Annex A
(informative)
Formal semantics

A.1 Introduction

This formal specification provides a clear unambiguous description of the meaning of the control constructs and most of the built-in predicates defined in this draft International Standard. Many features implicit in the informal definitions of the built-in predicates are explicitly described here.

NOTES

1 The following built-in predicates are not specified formally:

— `open/4`, `close/2`, `flush_output/1`, `stream_property/2`, `set_stream_position/2`, `read_term/3`, `write_term/3`, the system and the *read/write* options not being formalized.

— `op/3`, `current_op/3`, the operators not being formalized.

— `char_conversion/2`, `current_char_conversion/2`

2 This formal specification does not provide description of the concrete syntax of prolog texts.

3 Some error cases and other features may not be consistent due to last minute changes in the main parts of this draft International Standard. Disagreements may correspond to points in discussion or imprecisions.

The formal semantics is presented in four steps which should be read in the order:

A.2 — An informal introduction to the main features of the formal specification. This is also an informal introduction to standard Prolog and the semantics of control constructs and some built-in predicates (like assert, retract). It describes the main general properties of the formal specification which are needed to understand it.

A.3 — A description of the data structures used in the formal text and the comments of the A.4. Some structures are assumed to be defined by other means for example, arithmetic.

A.4 — The kernel of the specification and utilities written with clauses and local comments. One short comment is associated with each packet of clauses.

A.5 — The specification of the control constructs and built-in predicates.

The rest of this clause may be skipped, if familiar with logic programming. The other clauses need not be read sequentially. A better approach is to read the informal presentation (A.2) and then to start reading a built-in predicate defined in clause A.5, following the references to find the meaning of the predicates used in its definition.

A.1.1 Specification language: syntax

The formal specification is written in a specification language which is a first order logical language. It is a subset of most known dialects, in particular of standard Prolog (but, in order to avoid circular definition, with a proper syntax).

This language uses normal clauses (i.e. implications with possibly negative hypotheses). They are logical formulae written with:

— three logical connectors: "⇐" (implication), " **,** (conjunction)", "*not*" (negation).

— a finite set of semantical predicates which are themselves defined by normal clauses in clause A.4. (e.g. **semantics**, **buildforest**, etc.)

— a finite set of data structure predicates which are defined in clause A.3 and whose names are prefixed by **L-** or by **D-**.

— a finite set of special predicates.

— the arguments of the predications of the specification are either a variable, written using the syntax:

```
variable =
    capital letter char, { alpha numeric char}
    | _ ;
```

or some term built with all the function symbols used in the formal specification and representing databases, goals, search-trees, and other objects. Every value and constant of standard Prolog is denoted in the formal specification as specified in the abstract syntax in clause A.3.1.

NOTE — No confusion arises between symbols denoting a variable of a standard program and a variable of the specification language. In a standard program as in any feature related to the description of its behaviour (terms, database, streams, ...) all the objects are represented by ground terms. So they have a different syntax. However as variables and constants do not receive formally described treatment, no representation for these objects is provided in the formal specification.

A.1.2 Specification language: semantics

At first glance it may come as a surprise to give the semantics of standard Prolog using a strict subset of itself. There is no paradox: (1) Prolog programs and the formal specification have a different syntax, and (2) the semantics of the specification language is purely declarative whereas the semantics of standard Prolog can only be described operationally. The formal specification is a pure logical description of some meta-interpreter of standard Prolog programs.

The formal specification is axiomatic. It contains universally quantified first order logic axioms only. It can either be read logically (without specific knowledge of any existing Prolog dialect), or procedurally. But the semantics does not depends on any particular execution model, and the order of clauses and the predications in the bodies of clauses are irrelevant. Nevertheless they are given in an order which will aid readers to understand them. This axiomatic specification may be used to perform proofs of particular properties of the language. It may also be used to derive prototypes.

The semantics of clauses without negation is well-known. This is an advantage of this specification language; however, without

negation, its expressiveness is insufficient. With negation the specification language becomes extremely powerful.

Even if the formal specification can be considered as purely logical, its semantics is denoted by a specific model defined as *the set of the proof-tree roots*. The proof-trees are obtained by pasting together ground instances of normal clauses such that argument of a negative predication is not itself a proof-tree root.

Such a condition is not paradoxical because of the notion of **stratification**. Negation is stratified, i.e. a predicate is never defined recursively in terms of its negation.

NOTES

1 The stratification of negation is introduced to avoid a Russell-like paradox.

2 The specification uses five levels of stratification.

3 The use of negation by the specification fits with the usual notion of negation by failure, and thus simplifies the production of a consistent runnable specification from the formal one.

4 In the specification, a negated predication will never contain unbound variables. However the formal specification is not an "allowed" program.

5 The notion of stratification does not influence the logical reading of the axioms.

6 The semantics of the specification language fits with most of the known semantics for normal databases in logic programming, in particular it corresponds to the unique *stable model* or one of the *minimal term models of the completion*, or the *(two valued) well-founded model*.

Observe that only ground proof-trees are considered, but that other proof-trees can be constructed from the clauses of the formal specification. Only the subset of the ground proof-trees whose root is the predication **semantics** with arguments which are well-formed abstract objects (i.e. abstract database, goal and environment) are considered. This is a sufficient condition to guarantee that all such proof-trees are ground and with well-formed arguments in the formal specification.

The **D–** predicates are mostly simple relations, but necessary to make precise definitions.

The **L–** predicates are not defined in the formal semantics: they are an interface between the formal semantics and other specifications provided elsewhere in the standard. The semantics of **L-** predicates is defined by means of **relative denotation**. This means that their semantics is implicitly given by a possibly infinite set of ground predications. So the semantics of the whole formal specification is the set of the ground proof-tree roots (where the arguments are well-formed data structures) extended with the possibly infinite set of facts corresponding to the **L-** predicates.

A.1.3 Comments in the formal specification

Comments within the formal specification are of two kinds: **general comments** and **specific comments**.

General comments are all grouped in the clause A.2 (Informal description). They describe general properties of the specification which are difficult to deduce just by reading the axioms of the formal specification. They do not answer all possible questions about the behaviour of a standard database, but do assist its understanding.

Elsewhere only specific comments are given. Exactly one comment is associated with each data structure predicate or semantical predicate. The comments have the form:

pred*(X, Y)* — **if** *P(X)* **then** *Q(X, Y)*

or

pred*(X, Y)* — *iff Q(X, Y)*

where X, Y denote a partition of the arguments of **pred** and P and Q are assertions.

Such a comment is an informal description of the meaning of **pred**. It also corresponds to a partial correctness assertion: this means that all the predications in the semantics of the formal specification satisfy this assertion. If the comment contains *iff*, the assertion is also a completeness condition, i.e. the comment defines exactly all the predications in the semantics of this predicate. When there is a negative predication (e.g. *not Q(X, Y)*) in the body of a clause of the formal specification the comment required to understand it is usually the negation of the formula *Q(X, Y)* in the comment of the predicate of the predication.

In the formal specification every axiom is accompanied by cross references to the definitions of the predications in its body.

NOTE — More information on the specification method may be found in the following articles:

P. Deransart, G. Ferrand: An Operational Formal Definition of PROLOG: a Specification Method and its Application. New Generation Computing 10 (1992) 121-171. (The method)

A. Ed-Dbali, P. Deransart: Software Formal Specifications by Logic Programming: The example of Standard Prolog. LNAI 636, Springer Verlag, LPSS'92, September 1992. (On how the Formal Specification is used to improve the standard)

S. Renault, P. Deransart: Design of Formal Specifications by Logic Normal Programs: Merging Formal Text and Good Comments. INRIA document, February 1993. (Validation method by proofs)

A runnable specification (QUINTUS and SICStus compatible) is distributed by E-Mail on request to: AbdelAli Ed-Dbali (AbdelAli.Ed-Dbali@univ-orleans.fr)

A.1.4 About the style of the Formal Specification

The style of the formal specification may be surprising at first glance. Here are some observations which may help to understand it.

Terms of the form $f(t_1, ..., t_n)$ are denoted $func(f, t_1.....t_n.nil)$ in the formal specification. This is necessary to keep the specification first order. It helps also to understand what is the result of the unification performed on such terms (as defined in clause 7.3) which works the same way on abstract terms.

In the body of a clause a negated predication of the form

not **pred***(···)*

does not contain any anonymous variable in its arguments. This is because such variable is existentially quantified inside the

negation (it is universally outside of the clause, hence outside of the negation). As a result if such quantification is required an intermediate predicate must be introduced. See for example the predicates **error** A.4.1.14 and **in-error** A.4.1.15. Furthermore this facilitate production of executable specification using the standard negation.

The systematic use of "special predicates" where bootstrapped or auxiliary definitions are given is necessary to avoid clashes of names with user-defined predicates. In fact "bootstrapping" consists of adding to the initial complete database new predicate definitions. In the case of bootstrapped control construct or built-in predicates this is not needed because they cannot be redefined by the user.

A.2 An informal description

The semantics of a standard-conforming Prolog database is defined by the relation between the database, a goal, an environment and the corresponding search-tree which represents all the possible attempts to satisfy the goal (see A.2.7).

This kind of semantics takes into account non-determinism, i.e. the multiple (perhaps infinite number of) solutions, the unsuccessful attempts to resolve a query, and the control aspects as well. The representation of all the computations is usually defined by the so-called "search-tree" (also called SLD-tree in the case of "pure" Horn clause style). This notion is introduced in the next clause (A.2.1).

NOTE — The semantics can be viewed as being essentially declarative. The main difference with denotational semantics comes from the semantic domains (i.e. search-trees). An advantage of this approach is the relative familiarity of search-trees to Prolog programmers. It can also be considered as operational since the associated search-tree cannot be defined without simulating the execution of the database P for the goal G.

The process of the **construction of a branch** of the search-tree for a database and a goal (and an environment) corresponds to an attempt to **satisfy a goal**. The purpose of the formal specification is to describe all the possible attempts to satisfy a goal for a given standard Prolog database. It describes the **execution** of a goal. Any action performed before starting the execution is implementation defined or implementation dependent. It will be assumed that databases, goals and environments are already prepared for execution (in particular the **database** contains the clauses of the database to be executed and if a variable occurs as predication it has been included as argument of a `call` predication). The body of a fact contains only the predication `true`.

It is not required that the semantics should be a complete implementation of this search-tree. It should respect the following points:

a) the control flow: the order in which the nodes of the search-tree containing an executed user-defined procedure or BIP are visited.

b) failures, successes and/or successive instantiations of a goal in the same order.

c) effects of the BIPs.

The formal semantics is explained by progressively introducing the constructs and built-in predicates.

A.2.1 ("pure" Prolog) — The databases use only user-defined procedures and conjunction. "true" and "fail" are introduced.

A.2.2 ("pure" Prolog with cut) — Databases with cut.

A.2.3 (kernel Prolog) — All control constructs except "catch" and "throw" are considered. The notions of well-formed and transformed goal and of "scope of cut" are introduced.

A.2.4 — Structure of the database and "assert" and "retract" built-in predicates. The database update view is defined.

A.2.5 — Exception handling ("catch" and "throw").

A.2.6 — Environments.

A.2.7 — What is the semantics of a program conforming to this draft International Standard.

A.2.8 — Getting acquainted: general approach of the formal specification.

A.2.9 — Built-in predicates.

A.2.10 — Relationships with the informal semantics 7.7.

A.2.1 Search-tree for "pure" Prolog

Assume first that databases and goals use user-defined procedures and conjunction (,/2) only, and that a predication in the body of a clause cannot be a variable. A goal or the body of a clause is a possibly empty sequence of predications, denoted by the conjunction.

NOTE — This is "pure" Prolog. The notion of a search-tree was introduced for "pure" Prolog in the history of logic programming in order to explain the resolution and the backtracking as they are fixed in Standard Prolog, and it will serve as a basis to define and understand the semantics of further constructs.

Let us recall the notion of search-tree for pure Prolog, and thus the semantics of pure Prolog in the formal specification (because pure Prolog is a proper subset of standard Prolog). We will describe here what is known in the literature as the "standard" operational semantics of definite programs, or definite program with the left-to-right computation rule.

The clauses are ordered (by the sequential order in which they are written) and grouped into **packets** of clauses defining one procedure. The clauses have a head (a non-variable term) and a body consisting of an ordered conjunction of predications. If the body is empty it is denoted by **true**.

A database can be viewed a set of packets in which a procedure is defined only once by a single packet.

NOTE — In the formal specification all the clauses defining a predicate are grouped in a single packet. The way they are grouped is implementation defined according to the directive `discontiguous/1` 7.4.2.

The semantics of standard Prolog is based on the general resolution of a goal.

A.2.1.1 The General Resolution Algorithm

The general resolution of a goal G of a database P is defined by the following non-deterministic algorithm:

a) Start with the initial goal G which is an ordered conjunction of predications.

b) If G is empty then stop ($success$).

c) Choose a predication A in G (**predication-choice**)

d) If A is **true**, delete it, and proceed to step (b).

e) If no renamed clause in P has a head which unifies with A then stop ($failure$).

f) Choose a freshly renamed clause in P whose head H unifies with A (**clause-choice**) where $\sigma = MGU(H, A)$ and B is the body of the clause,

g) Replace in G the predication A by the body B, flatten and apply the substitution σ.

h) Proceed to step (b).

NOTES

1 The steps (c), (f), and (g) are called **resolution step**.

2 The MGU (most general unifier) of two terms is defined in clause 7.3.2.

3 A "freshly renamed clause" means a clause in which the variables are different from all the variables in all the previous resolution steps.

4 In standard Prolog, there is no flattening of goals. If not identical to true, a goal can always be viewed as a conjunction of (sub) goals.

A.2.1.2 The Prolog resolution algorithm

In standard Prolog this algorithm is deterministic:

a) The predication-choice function chooses the first predication in the sequence G (step (c)).

b) The clause-choice function chooses the unifiable clauses according to their sequential order in the packet (step (f)).

A.2.1.3 The search-tree

The different computations defined by this algorithm will be represented by a (search-)tree in which a node is labelled by the **current goal** and has as many children as there as unifiable heads with the chosen predication in the current goal. The children have the order of the selected clauses in the database.

NOTE — The search-tree is a suitable tool at the right level of abstraction. It is a well-known notion in the logic programming community.

The notion of search-tree permits to represent dynamic computations as a unique object. It formalizes the idea of "time" which is implicitly present in the total order of its nodes (total visit order).

We give now a more precise definition. Each node is labelled by two elements:

— Either a non-empty goal and a distinguished predication (**the chosen predication**), or the predication `true` and the node is a leaf called **success node**.

— a substitution.

The label of the root is the goal to be resolved and the empty substitution.

Each node has as many children as there are clauses whose head (with a suitable renaming) is unifiable with the chosen predication. So if there is no such clause the node is a leaf called **failure node**. It corresponds to a **failed branch**. A success node corresponds to a *success branch*. To every success branch it corresponds an *answer substitution* obtained by the composition 7.3 of all the substitutions of the nodes along the branch, restricted to the variables of the goal of the root.

There are three kinds of branches: success, failure, *infinite*. If there is no infinite branch in a search-tree, it is a *finite* search-tree.

The order of the children corresponds to the order of the clauses used to build them in the database. If $B_1, \cdots, B_n$ is the goal associated with a node, B_1 being the chosen predication, and A :- $C_1, \ldots, C_m$ is a freshly renamed clause with A and B_1 unifiable, then with the corresponding child the associated substitution is a MGU (most general unifier) σ of B_1 and A, and the associated sequence of predications is

$\sigma((C_1, \ldots, C_m), B_2, \ldots, B_n)$

or equivalently, if flattened:

$\sigma(C_1, \ldots, C_m, B_2, \ldots, B_n)$

In the General Resolution Algorithm (A.2.1.1) the search-tree is defined by the **predication-choice** function (also called **computation rule**) which determines the chosen predication for each node. The predication-choice function could select any predication in a goal and not just the first one. The Prolog search-tree is defined by the predication-choice function which always chooses the first predication.

All search-trees (i.e. corresponding to different computation rules) are equivalent in the sense that given the database and the goal, all the different search-trees have the same success nodes with the same answer substitutions up to a renaming of the variables.

A.2.1.4 The visited search-tree

Given a predication-choice function, i.e. a search-tree, the computation of a database and goal is defined by depth-first left-to-right visit of the search-tree. This visit defines the output order of the answer substitutions as the visit order of the success leaves. It also explains why the execution loops when the traversal visits an infinite branch.

To sum up, the semantics of a database and a goal is formalized by the search-tree with its **visit order**. We call **visited search-tree** (VST) a search-tree provided with a visit order. The semantics of standard Prolog is defined by two components: the *predication-choice* function (search-tree) and the *visit order* (of this search-tree).

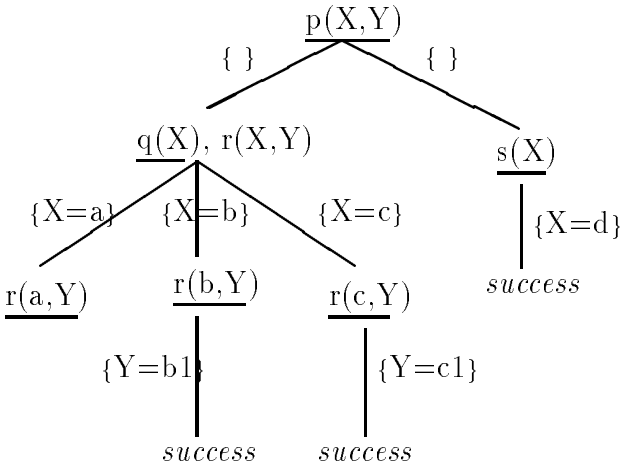

Figure A.1 — A search-tree example

A.2.1.5 A search-tree example

Consider the following database and the goal `p(X, Y)`

```
p(X, Y) :-
   q(X), r(X, Y).
p(X, Y) :-
   s(X).

q(a).
q(b).
q(c).

r(b, b1).
r(c, c1).

s(d).
```

Figure A.1 shows the standard search-tree with the chosen predication underlined, upper case letters denote variables and lower case constants.

The standard visit gives the following answer substitutions, in this order:

```
X = b, Y = b1

X = c, Y = c1

X = d
```

A.2.1.6 Building the visited search-tree

The semantics of a database P and a goal G is thus represented by a partially visited search-tree whose root is labelled by the goal G. Successive transformations modify the initial partially visited search-tree during the resolution.

When a node N is first visited it is immediately expanded with all its children. The representation of the search-tree respects the order of visits; the non-visited brothers of an already visited node are all "on the right" of this node. These nodes are called "hanging nodes" (see figure A.2). In a partially visited search-tree all the hanging nodes are "on the slice" and represent the next possible developments of the search-tree. The **clause-choice** function selects the next node to be visited, following the visit order.

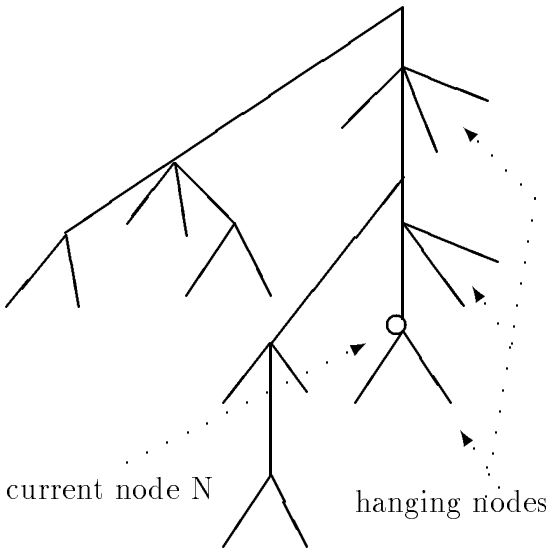

Figure A.2 — A visited search-tree

Observe now that there is no way to visit the search-tree beyond the first (i.e. left-most) infinite branch with the Prolog visit order. This is why in the formal specification the semantics is represented by all the finite partial search-trees which are partially visited up to some current node.

If the search-tree is finite (no infinite branch), then the semantics contains a greatest tree which corresponds to the complete visited search-tree (up to the root).

If the search-tree is infinite the semantics consists of all the partially visited search-trees containing all the visited nodes from the root up to some node of the first infinite branch.

A.2.1.7 Semantics terminology

Let us now introduce some vocabulary as defined in clause 7.7. Given a branch of a search-tree whose current node N is labelled by a goal G (called the **current goal**) such that A is the chosen predication in G, the **activation period** of A corresponds to the construction of the sub-search-tree issued from N. Of course, the activation period has no end if this sub-search-tree has an infinite branch.

If a node has more than one child it is **non-deterministic**. Such a node for which A is **re-executable** is called a **choice point**. If a node has only one child after its first visit it is a **deterministic node**. A node is said **completely visited** after all its branches have been completely developed. New visits to a choice point is called **backtracking**.

A.2.1.8 An analogy with Byrd's box model

Comparing this semantics with Byrd's trace model helps show how nodes are visited.

Byrd's box (figure A.3) represents what happens during the activation of a predication, i.e. between its choice at the current node ("call") and the last visit to this node ("redo fail"). The different visits correspond to different choices of clauses leading

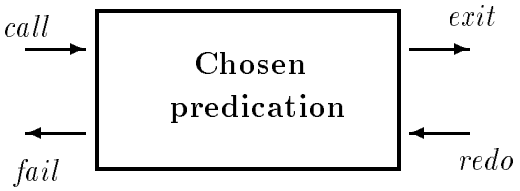

Figure A.3 — Byrd's trace model

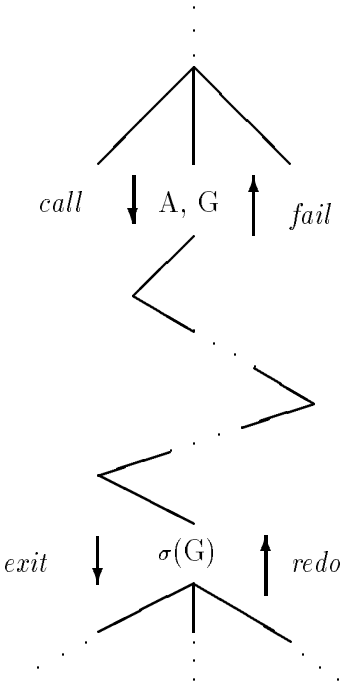

Figure A.4 — Byrd's model: a search-tree point of view

to success branches.

Figure A.4 shows the elements of Byrd's box from the search-tree point of view.

By analogy with Byrd's model the visits of a node N will be denoted by "call" for the first and "fail" for the last one of the same node. They correspond to the call of a predication and the end of all the attempts to resolve it. The "fail" mark must be distinguished from the failure nodes introduced previously. In fact many branches issued from the node N may be failed. The other attempts to re-execute it correspond to "redo" for obvious reasons (try a new clause at some ancestor choice point and continue the resolution). "exit" corresponds to one successful attempt to resolve the chosen predication of the node N.

NOTE — In this draft International Standard "a predication fails" means failure if there is no way to satisfy it, or just last visit if after different attempts to re-execute it (after backtracking).

A.2.2 Search tree for "pure" Prolog with cut

"Pure" Prolog is now extended by allowing the constant predication cut(!/0) in the body of the clauses.

From the logical point of view this cut has no effect (it is always true), but from the point of view of the computations (the search-tree) it has a drastic effect: a cut deletes some

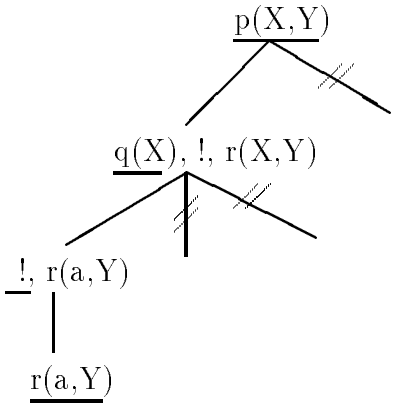

Figure A.5 — A search-tree example showing the effect of cut

search-tree branches in order to force a predication to execute quickly without visiting all its children.

A.2.2.1 A search-tree example with cut

If the first clause of the database (A.2.1.5) is replaced by

```
p(X, Y):- q(X), !, r(X, Y).
```

```
p(X, Y) :-
   q(X), !, r(X, Y).
p(X, Y) :-
   s(X).

q(a).
q(b).
q(c).

r(b, b1).
r(c, c1).

s(d).
```

Figure A.5 shows that the search-tree corresponding to the goal `p(X, Y)` has one success branch instead of three.

NOTE — Cuts sometimes increase the number of success branches. This may be understood by the use of the cut to specify negation by failure. The composition of two negations may increase the number of successes.

The effect of the "cut" is thus to erase some hanging nodes: all the hanging nodes between the current node and the parent node of the goal in which it first appeared.

A.2.3 Search-tree for kernel Prolog

In **kernel Prolog** only the control constructs (true/0, fail/0, !/0, ','/2, ';'/2, call/1, `'->'`/2, "if-then-else"/3) and the user-defined procedures are authorized.

A.2.3.1 Syntax: well-formed clause, body and goal, and transformation

In this draft International Standard (as in **kernel Prolog**) a clause in the database, a goal, or the body of a clause, must be well-formed. So a variable cannot occur in the position of a predication (it must be embedded in a call like `call(X)` in this

case), and a predication must be a callable term (i.e. neither a variable, nor a number).

By definition well-formed clause, body of clause, or goal must respect the following abstract syntax (formally defined in A.3.1):

```
clause = predication :- body

body =
    | ',' '(' body ',' body ')'
    | ';' '(' body ',' body ')'
    | '->' '(' body ',' body ')'
    | predication

predication =  "pred(list of terms)"
```

where `pred` is not in {`','`, `';'`, `'->'`} and `predication` is not a number.

If a clause or a goal is well-formed, a transformation may be performed as follows.

An (abstract) clause term of the form ':-'(H,G) is transformed into the term ':-'(H,trans_goal(G)) where trans_goal defines the transformation of a goal as follows.

An (abstract) goal term is transformed in a new goal whose behaviour is equivalent, according to this draft International Standard, to the same goal in which each variable "X" occurring in the position of a predication according to the abstract syntax above is replaced by "call(X)".

NOTE — This specification is weaker than what is specified in the term to body conversions 7.6.3: it suggests that some transformations may be implementation dependent . However as the effect must be equivalent to the given minimal transformation, only this minimal transformation is considered in the formal specification (see **D-term-to-body** A.3.1) as in 7.6.3.

A.2.3.2 An operational view of the conjunction (,/2)

The conjunction may be viewed now as a control construct combining goals. The semantics of this construction is defined by the mechanism of the search-tree construction and visit. It may be also informally described as follows:

if G_1 and G_2 are two goals then (G_1, G_2) is equivalent to execute G_1 and execute G_2 in sequence each time G_1 is satisfied.

The conjunction satisfies also the following obvious properties:

$goal,\ true = true,\ goal = goal$ and
$((gl_1, gl_2), gl_3) = (gl_1, (gl_2, gl_3)) = (gl_1, gl_2, gl_3)$

NOTE — These properties hold not only for kernel Prolog but also in standard Prolog.

A.2.3.3 `true` and `fail`

The meaning of `true` and `fail` is now clear. If the chosen predication is `true`, then it will be removed and the resolution continues with the following predications of the current goal. If there are no more predications, the branch is a success branch, and resolution continues from the closest choice point not yet completely visited.

`fail` can be viewed as a constant predicate with no definition at all. Hence its choice leads to a failed branch and resolutions continues from the closest choice point not yet completely visited.

A.2.3.4 Disjunction

disjunction is the control construct of two goals $G_1$and G_2 denoted $(G_1; G_2)$ whose meaning is equivalent to:

Execute G_1 and skip G_2 each time G_1 is satisfied, and execute G_2 when G_1 fails if this alternative has not been cut by the execution of G_1.

The disjunction corresponds to a non-deterministic choice-point. The simplest semantics for the disjunction is given by the two pseudo-clauses ("pseudo" because the disjunction is a control construct and is not authorized as functor of a clause head) :

```
';'(G1, G2) :- G1.
';'(G1, G2) :- G2.
```

A.2.3.5 Cut in kernel prolog and its scope

A cut may occur any where, embedded inside conjunctions, disjunctions or if-then constructs according to the abstract syntax above. Then the *(static) scope* of the cut is defined by the visible choice points which will be cut when it will be chosen. In a clause the visible choice points are the head of the clause and the disjunctions associated with the control construct ';' in which the cut is embedded. There are also "non visible" choice points which are introduced by the development of subgoals which have been chosen before the cut. Hence the scope of cut in a clause corresponds to all the predications or disjunctions which are on its left in the body of the clause together with all its embedding disjunctions and the head of the clause.

However there is one exception to this rule if the cut is inside the control part of a the `if-then` construct (see A.2.3.6).

In the formal semantic the scope of a cut is represented by flagging cut (!(flag)) where the flag is to the parent node of the node in which the instance of its body has been introduced first. When a cut flagged by N is chosen all the ancestor choice points including N are made deterministic.

NOTE — Due to the well-formedness of goals, there is no way to execute an unflagged cut. Hence if a cut is the argument of some disjunction, the variables of the bootstrapped definition contain this already flagged cut.

A.2.3.6 If-then

The **conditional construct** `'->'(Cond, Then)` is defined as follows:

if `Cond` succeeds then cut the choice points of `Cond` and execute `Then`.

It can be defined by the following pseudo-clause:

```
'->'(Cond, Then) :-
        call(Cond),!,Then.
```

NOTE — The 'call' in the condition is introduced with the only purpose to limit the scope of cut in the condition. In this draft International Standard a goal is assumed well-formed, thus the condition of an `if-then` construct is already well-formed.

A.2.3.7 If-then-else

The conditional construct "if-then-else" is denoted by a syntactical combination of if-then and the disjunction as follows:

```
(Cond -> Then); Else
```

It is defined as follows:

if `Cond` succeeds then cut the choice points of `Cond`, execute `Then` and skip `Else`. If `Cond` fails then execute `Else`.

It could be defined by the following pseudo-clause:

```
((Cond -> Then); Else) :-
   (call(Cond),!,Then);Else.
```

In the formal specification it is bootstrapped together with the disjunction.

A.2.3.8 Call

`call` is a control construct which permits the use of a variable as a predication and limit the scope of cut.

Its syntax is `call(Term)` where `Term` must be a term. When `call` is executed its argument must be a well-formed term (see **is-a-body** A.3.1), the scope of a cut in this goal is limited to this goal. Then the argument is transformed according to clause A.2.3.1 and executed, after local cuts of the goal, in the position of a predication according to the syntax above, have been flagged.

A.2.4 Database and database update view

In this draft International Standard it is assumed that the database contains at least three informations for every user-defined procedure: the predicate indicator, an indication whether it is dynamic or static and the packet of clauses (**D-is-a-database** A.3.1. Each clause can be viewed as an abstract term (**D-is-a-term** A.3.1).

The semantics described so far assumes that the database remains unchanged during the the execution of a goal. Standard Prolog contains three built-in predicates which may modify the database: asserta/1, assertz/1, retract/1, with an argument which is a clause or explore it: clause/2. Intuitively asserta/1 adds a clause at the beginning of a packet, assertz/1 does the same at the end, and retract/1 removes the first clause which unifies with the argument. retract/1 is resatisfiable and removes successive clauses in the packet. Notice that an asserted clause must be well-formed and that `retract(predication)` seeks for clauses of the form `predication :- true`.

To understand the semantics of these built-in predicates in standard Prolog it is useful to understand the problem of the database update view. As the search-tree is constructed the database may be modified. Add to each node an additional label corresponding to the current database used to build the children of this node. Assume first that all the clauses are used to build these children. Each child (say $1, 2, \ldots, n$) is now labelled by a new database (say $NewP_1, NewP_2, \ldots, NewP_n$). This situation is depicted in Figure A.6 (the c_i's correspond to the clauses chosen to build the child).

If there is no modification of the database all the $NewP_i$ and P are the same and all the children are visited and expanded.

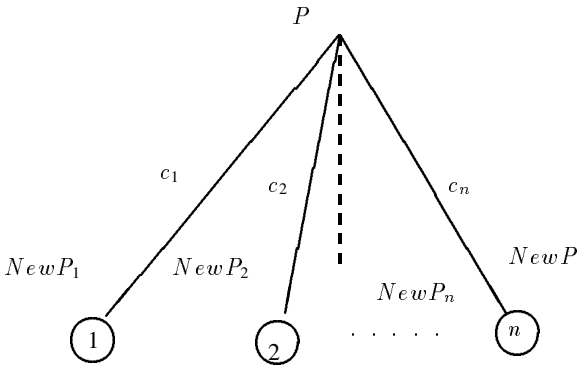

Figure A.6 — Standard database update view

Now consider a child i different from the first ($i > 1$) and assume that the clause to which it corresponds has been removed during the constructions of an older brother (i.e. $NewP_i$ does not contain anymore the clause c_i). Is it normal to choose and to try to resolve it or not? Assume now that the youngest child n has been reached and resolved, and the current database, say $NewP$ corresponding to the "fail" visit of n, contains new clauses appended to the corresponding packet in P. Should these new clauses be considered to create dynamically new children or not? Notice that such situation happens with assertz/1 only. With asserta/1 no new child will be created (although subsequent uses will consider the modified database).

The database update view depends on the way the previous questions are answered. The standard adopts the following view: the retracted clauses are selected but not the appended ones. It is called the **logical view**.

NOTE — In the formal specification the logical view is taken into account as follows: the packet associated to a node corresponds to the clauses available to build new children. It is fixed by the first visit.

Although the logical view has been adopted, some programmers are used to the so-called immediate view. There is a "minimal" way of thinking about the views, that is to say to database in a such manner which does not depend on the view (it is of course undecidable whether a given database satisfies the requirements of some view, hence in particular of the logical one). Here are some possible rules:

a) Using asserta/1 is always free of danger.

b) Never use retract/1 or assertz/1 on a predicate which is active except to retract already used clauses.

These restrictions fit with a prudent use of database updates. However note that, even without "call", these apparently simple rules remain undecidable.

NOTE — In A.2.2 where "pure" Prolog is described there is no data base updates and the database is invariant. In standard Prolog it is not the case. Thus a different database may be stored at each node of the search-tree. In the formal semantics only one database is stored at a node (instead of one "before" and one "after"): it is the database resulting from the complete development of the sub search-tree issued from that node; it may be different from the database associated to this node when it is the current node.

A.2.5 Exception handling

An exception may be raised during the resolution of a goal G by the system or by the user (with the control construct throw/1) and captured anywhere by some ancestor control construct catch/3 if the resolution of this goal G is performed in the context of this bip. The mechanism of the exception handling can be informally described as follows.

The bip catch/3 has three arguments: a goal to be executed (say `Goal`), a catcher which is term (say `Catcher`) and another goal to be executed in case an error occurs during the resolution of `Goal` trapped by this predication (say `Recovergoal`). Its semantics is the following: assume that `catch (Goal, Catcher, Recovergoal)` is chosen at node N. Unless some syntactic error on the form of this predication arises, it succeeds and two children are created labelled by the two goals: `(Goal, inactivate(...), Cont)` and `(Recovergoal, Cont)`, where `Cont` is the continuation defined by the goal of the node N (the goal at node N has the form `(catch (...),Cont)`). Note that N is non-deterministic. However if no error occurs the second child will never be visited and the node N will be considered as deterministic by **clause-choice**).

When an error is raised by a predication `throw(Ball)`, this predication succeeds if a freshly renamed copy of its argument `Ball` can be unified with the catcher of some calling ancestor catch/3 (else a system error is raised). If some ancestor catch is thus selected all the hanging nodes of its sub-search-tree are removed and its second child is developed, hence the goal `(Recovergoal, Cont)` is now resolved.

NOTE — In the formal specification the second node is not immediately constructed. It is by `throw`.

The role of the special bip inactivate/1 defined in the formal specification only is to avoid the capture of an error by the catcher of a calling catch when this error occurs during the resolution of the continuations. In fact, an error may be trapped by different catchers in different embedded catches, and an error in the continuation must be trapped by ancestor catches only. For this purpose the set of the active catchers is stored at the current node (i.e. catchers which must be tried if some error is raised) and the effect of inactivate whose argument is the node N is to remove this node from this set. Hence subsequent errors raised by the developments of `Cont` are no longer caught by the catcher of node N.

Notice also that there are two kinds or exceptions:

a) Explicit ones specified by the programmer by throw/1, and

b) Implicit ones raised by control constructs or bip errors. This case is exactly as though a user error is raised by calling `throw(`*Type_of_error_term*`)`.

Furthermore if the user for any reason omits to specify an appropriate catcher, everything will happen as if there is a catcher at the root (see A.4.1.1) resolving a special predication called *system-error-action* whose effect is implementation dependent.

Finally observe that the exception handling introduces a new kind of failed branch. In the leaf of such failed branch the chosen predication may be a `throw` or a bip in error or a special predicate called *system_error_action*. In the case of `halt` as for some other special predicates as well new leaves can be added to the search-tree which do not correspond either to any success or failure branch. The possible development of such branch is implementation defined or implementation dependent.

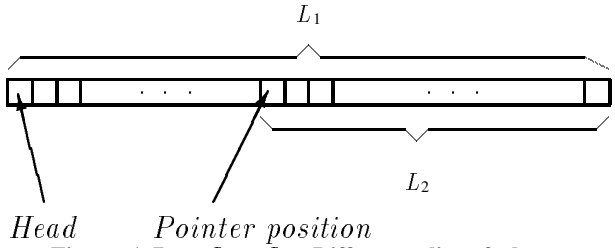

Figure A.7 — $L_1 - L_2$: Difference list of characters

A.2.6 Environments

In this draft International Standard it is required that an environment is defined at least by the values of the flags and the input and output text streams. The environment may be updated at each step of execution which affects flags or streams.

In this formal specification the current environment is attached to the current node. An environment is a quintuple which contains the current list of flags, the input and output streams and two lists of currently opened input and output streams respectively. It is denoted *env(PF, IF, OF, IFL, OFL)* (**D-is-an-environment** A.3.7).

>From the formal point of view a stream is considered as a sequence of characters which ends with an "eof" character. A stream is represented by a name and a difference list of characters. This representation permits the manipulation of both the head and the current character of the stream (see **D-is-a-stream** A.3.7).

To denote a stream, we use *stream(N, L1 - L2)*, where N is the abstract name of the stream and *L1 - L2* is a difference list of characters. *L1* represents the whole contents of the stream and *L2* represents the characters after the pointer (including the pointed first character). Thus an empty stream is denoted $nil - nil$ and points on the "eof". A stream $L - L$, L being a non empty list of characters points on the first character. A stream $L - nil$ points on the end of file (the current character is "eof". A stream $A.L - L$ points on the second and $L - A.nil$ on the last. Initially a non empty stream pointing on its first element A will be denoted $A.L - A.L$.

A.2.7 The semantics of a standard program

The semantics is defined by a relation with four arguments, called **semantics** (A.4.1.1) whose arguments are: a database (the initial database), a goal, an environment and a forest. The forest corresponds to the partially visited search-tree up to the **current node**, usually denoted by N in the formal specification. If for a given database P, goal G and environment E there is a finite search-tree, then in the semantics of this relation there is a proof-tree such that the fourth argument of the root represents this finite complete search-tree. The search-tree is represented by a data structure called "forest" (see A.3.3).

NOTES

1 It is important to observe that the semantics is not unique: there may be many search-trees for the same database, goal and environment,

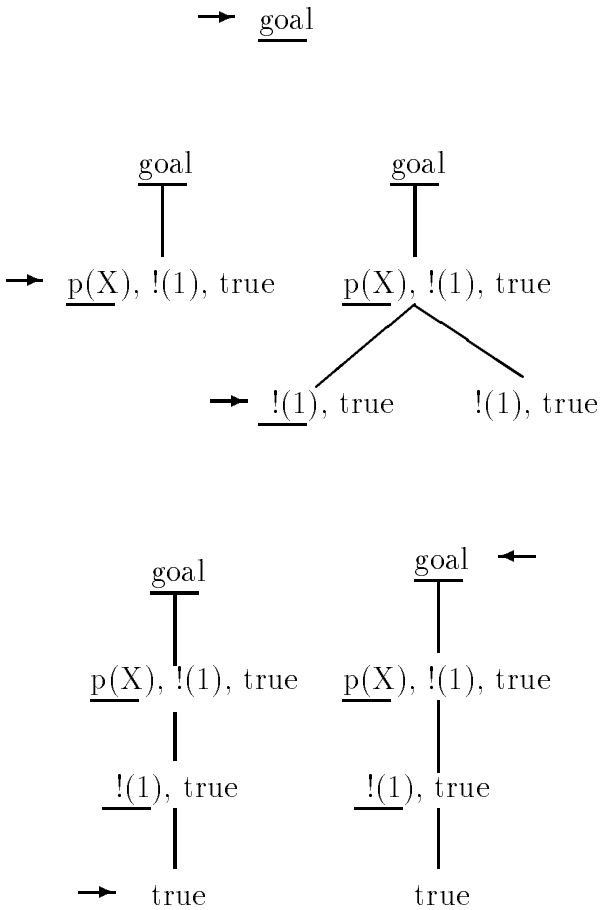

Figure A.8 — Partially visited search-tree

even if they are finite, each denoting a standard conforming semantics. This is due to undefined or implementation dependent features.

For example exceptions occurring during the computation of different subexpressions may lead, in an implementation dependent but also programmed manner, to completely different executions. Other cases are illustrated by the term-ordering (A.4.1.40) which is implementation dependent in the comparison of variables, or renaming.

2 The semantics specifies all the partially visited search-trees up to some current node. This is needed to take into account infinite executions.

To illustrate the semantics we give a short example with a simplified notation (the current goals and environments are not depicted).

Consider the database:

```
p(a).
p(b).

goal :- p(X), !.
```

and the goal: `goal`.

Its semantics contains all the partially visited search-trees depicted in Figure A.8 ($\rightarrow$ denotes the node N to be executed and $\leftarrow$ the last completely visited node) representing the evolution of the search-tree.

NOTE — In A.4 the relation **semantics**(P, G, E, F) defines the execution of `true & catch`(G, X, system_error_action)) instead of G. The `catch` serves to take into account untrapped errors during execution. The conjunction of `true` and `catch` serves in the description of `halt` which creates a new child to the root. Such behaviour is indeed implementation defined .

A.2.8 Getting acquainted with the formal specification

The general structure of the formal specification can now be described. The details are of course defined in the formal text (A.3, A.4).

The key predicate of the relation semantics is a predicate **buildforest** (A.4.1.3). It is non-deterministic in order to include in the semantics all the finite approximations of the (eventually infinite partial) search-tree. Each approximation includes the nodes of the previous approximation but some elements on the slice may have their labels altered by performing the transformations called "expansion".

The predicate **buildforest** simulates the search-tree walk construction. It uses the predicate **clause-choice** (A.4.1.4) which, in standard Prolog, selects the next not yet completely visited node other than a `catch` following the standard visit order or the root if there is no eligible node.

NOTE — A catch node is not completely visited because its alternative is chosen only after an error or throw occurring in the development of its first branch.

The predicate **treatment** (A.4.1.13) analyses the current goal (the goal labelling the current node) and expands it according to the selected predication in the current goal.

Notice that the current node "before" or "after" **treatment** is the same, but the search-tree may have been expanded "after". Hence "before" **treatment***(F1, N, F2)* N is a hanging node of $F1$ on the slice, but N may have children in $F2$.

The different cases of **treatment** deal with success, bip not in error, error case, special predicate and failure.

The addition of a new node is made by **expand**(A.4.1.18) in which **buildchild** (A.4.1.24) constructs a new node following the logical database update view and **addchild** makes the search-tree expansion by adding this new child. As soon as a search-tree issued form a node N is completely built and visited, the node N is marked *completely visited* and cannot be chosen any more for new visits (this happens when all the choices are cut inside a sub search-tree for example).

NOTES

1 The children nodes of a node n are numbered $zero.n$, $s(zero).n$, ...

2 The hanging nodes (i.e. all the children of a current node) are not explicitly built. Only the next child not yet visited of the current node is.

3 The packet associated to a node corresponds in fact to the remaining children to be built. If it is nil thus no more children can be built.

Some resatisfiable bip's like bagof/3 use different kind of packets. However elements of a packet always have the abstract syntax of a clause.

A.2.9 Built-in predicates

Most built-in predicates are defined by a search-tree transformation using the predicate **treat-bip**. One or more clauses for **treat-bip** (A.4.1.31) are given in the clause for that predicate, together with clauses for **in-error** (A.4.1.15) to show error cases. Only positive and error cases are specifies. Other cases correspond to failure.

Some built-in predicates do not modify the search-tree other than by generating a new node in case of success and a (local) answer substitution. These predicates, called **substbip** (A.3.8), are described by the relation **execute-bip** (A.4.1.36) which defines this substitution. Clauses for **execute-bip** are thus given in the clause for that predicate.

Other built-in predicates are boot-strapped. Formally these predicates are defined by a piece of a Prolog database as an argument to **D-packet** (A.3.8). However a database given using the abstract syntax is less clear than using the concrete syntax, and also we have not chosen how to represent integers, variables, etc. Therefore the packet is given implicitly using the concrete syntax of Prolog.

For example `@>` is defined by:

```
X @> Y :- Y @< X.
```

This implies that the specification contains a clause like:

D-packet(_ *func*(`@>`, _._*nil*),
func(`:-`, *func*(`@>`, *Var1.Var2.nil*).
func(`@<`, *Var2.Var1.nil*).*nil*).*nil*) ⇐
L-var(*Var1*),
L-var(*Var2*),
not **D-equal**(*Var1, Var2*).

Boot-strapping is normally used if the boot-strapped definition is simpler and more understandable than a direct definition using **treat-bip**.

Each bip definition contains a formal definition or a boot-strapped one in a concrete syntax form and the clauses of **in-error** defining the error cases.

A.2.10 Relationships with the informal semantics of 7.7 and 7.8

The formal specification uses the search-tree model. In this model all the computations are denoted by one (possibly infinite) object. The informal semantics is based on a stack. It describes the execution of "kernel Prolog"(A.2.3) only. Each computation is described separately.

With this restriction in mind, there is a one-one correspondence between the nodes of the search-tree along a path and the elements of the stack (*execution states*). A goal associated to a node is coded in the informal semantics as a stack whose top element corresponds to the chosen predication. The order of the elements in the stack (from the top to the bottom) corresponds to the order in which the predications (called *activators*) are chosen. All its elements are called *decorated subgoal*. A decorated subgoal has a pointer (called *cutpointer*) which points to the equivalent choice point of the search-tree. The *current state* of the resolution stack corresponds to the current node in the formal semantics.

In short the model of the informal semantics reflects a possible implementation of the search-tree visits for "kernel Prolog".

A.3 Data structures

This clause introduces the **L-** and **D-** predicates.

The following data structures are considered:

— A.3.1 abstract database and term (abstract syntax)

— A.3.2 predicate indicator

— A.3.3 forest: structure and updates

— A.3.4 abstract list, atom, character and lists

— A.3.5 substitution and unification

— A.3.6 arithmetic

— A.3.7 difference lists and environments

— A.3.8 built-in predicates, packets and special predicates

— A.3.9 input and output

A.3.1 Abstract databases and terms

In clause 6, the abstract syntax of terms, goals and clauses is represented by terms of the form $f(t_1, ..., t_n)$. These terms are denoted $func(f, t_1.....t_n.nil)$ in the formal specification. $t_1.....t_n.nil$ is called an arg-list. A constant c has the form $func(c, nil)$. In the same clause 6, a Prolog text is denoted by an arg-list whose elements are terms (clauses).

The abstract syntax presented here in a clausal form defines the objects called in the formal specification: term, clause, predication (or activator), database and goal as they are ready for execution. Other objects: lists and environment are defined in the corresponding subclause.

NOTES

1 A clause in the body is defined as a term whose principal functor is `':-'` or a predication (if it is a fact). In the formal specification it is considered that in a database, prepared for execution, all the facts have also the form of a rule whose body is `true`.

2 A predication cannot be a variable. In a database prepared for execution all the predications reduced to a variable `X` occurring in a Prolog text must have been converted to `call(X)` which is a term.

3 It is assumed that a procedure is defined only once in the abstract database.

D-is-a-database(*DB*) — *iff* $D\,B$ is the abstract representation of a concrete database

D-is-a-database(*nil*).

D-is-a-database(*P.DB*) ⇐
D-is-a-pred-definition(*P*),
D-is-a-database(*DB*).

D-is-a-pred-definition(*P*) — *iff* P is a definition of a user-predicate.

D-is-a-pred-definition(*def(PI, SD, P)*) ⇐
D-is-a-predicate-indicator(*PI*),
D-is-a-static-dynamic-mark(*SD*),
D-is-a-packet-of-clauses(*P*).

NOTE — References: **D-is-a-predicate-indicator** A.3.2.

D-is-a-packet-of-clauses(*P*) — *iff* P is the abstract representation of a sequence of clauses prepared for execution.

D-is-a-packet-of-clauses(*nil*).

D-is-a-packet-of-clauses(*C.P*) ⇐
D-is-a-clause(*C*),
D-is-a-packet-of-clauses(*P*).

D-is-a-clause(*func(:–, H.B.nil)*) ⇐
D-is-a-predication(*H*),
D-is-a-body(*B*).

D-is-a-body(*func(&, G1.G2.nil)*) ⇐
D-is-a-body(*G1*),
D-is-a-body(*G2*).

D-is-a-body(*func(;, G1.G2.nil)*) ⇐
D-is-a-body(*G1*),
D-is-a-body(*G2*).

D-is-a-body(*func(–>, G1.G2.nil)*) ⇐
D-is-a-body(*G1*),
D-is-a-body(*G2*).

D-is-a-body(*B*) ⇐
D-is-a-predication(*B*).

D-is-a-predication(*func(N, A)*) ⇐
L-atom(*N*),
not **D-equal**(*N, &*),
not **D-equal**(*N, ;*),
not **D-equal**(*N, –>*),
D-is-an-arglist(*A*).

NOTE — **D-is-a-clause** (**D-is-a-body**) define what is a well-formed term clause (term goal), or convertible in the sense of 7.6.

D-is-an-arglist(*L*) — *iff* L is an arg-list of terms.

D-is-an-arglist(*nil*).

D-is-an-arglist(*X.L*) ⇐
D-is-a-term(*X*),
D-is-an-arglist(*L*).

D-is-a-term(*X*) ⇐
L-var(*X*).

D-is-a-term(*func(N, L)*) ⇐
D-is-a-constant(*N*),
D-is-an-arglist(*L*).

D-is-a-constant(*X*) ⇐
L-atom(*X*).

D-is-a-constant(*X*) ⇐
L-integer(*X*).

D-is-a-constant(*X*) ⇐
L-float(*X*).

D-is-a-static-dynamic-mark(*SD*) — *iff* SD is a static/dynamic mark.

D-is-a-static-dynamic-mark(*static*).

D-is-a-static-dynamic-mark(*dynamic*).

D-is-a-callable-term(*T*) — *iff* T is a callable term as it is defined in 3.18.

D-is-a-callable-term(*T*) ⇐
not **D-is-a-number**(*T*),
not **L-var**(*T*).

L-var(*X*) — *iff* X denotes a concrete variable, i.e. an element of V defined in clause 6.1.2.

L-witness(*L,V*) — *iff* T is an abstract list of terms and V a term such that all its variables occur in some element of L. (If L is a singleton $t.nil$, V is the witness of the term t as defined in 7.1.1.2.

L-atom(*X*) — *iff* X denotes a concrete atom (identifier), i.e. an element of A defined in clause 6.1.2 b.

L-integer(*X*) — *iff* X denotes a concrete integer, i.e. an element of I defined in clause 6.1.2 c.

L-float(*X*) — *iff* X denotes a concrete floating point number, i.e. an element of R defined in clause 6.1.2 d.

D-is-a-goal(*G*) — *iff* G is the abstract representation of a goal.

D-is-a-goal(*G*) ⇐
D-is-a-body(*G*).

The syntax of a goal is now extended as follows:

D-is-a-predication(*G*) ⇐
D-is-a-special-pred(*G*).

D-is-a-predication(*func(!, I.nil)*) ⇐
D-is-a-dewey-number(*I*).

D-is-a-special-pred*(special-pred(inactivate, I.nil))* ⇐
D-is-a-dewey-number*(I).*

D-is-a-special-pred*(special-pred(system-error-action, nil)).*

D-is-a-special-pred*(special-pred(halt-system-action, nil)).*

D-is-a-special-pred*(special-pred(halt-system-action, I.nil))* ⇐
D-is-an-integer*(I).*

D-is-a-special-pred*(special-pred(value, __nil)).*

D-is-a-special-pred*(special-pred(compare, __nil)).*

D-is-a-special-pred*(special-pred(simple-comparison, ___nil)).*

D-is-a-special-pred*(special-pred(operation-value, ____nil)).*

D-is-a-special-pred*(special-pred(sorted, __nil)).*

NOTE — This additional abstract syntax defines the notion of extended goals. The formal specification uses flagged cuts and special predicates (in order to avoid clashes with user defined procedures) as predications. Except for the predicate **semantics** the comments will refer to the extended well-formed database.

This abstract syntax takes into account these new predicates:

— *func(*`!`*, D.nil)* where D is a dewey number, is allowed as a predication. This is to allow each cut to be flagged.

— *special-pred(system-error-action, nil)*,
special-pred(halt-system-action, nil),
special-pred(halt-system-action, _nil),
special-pred(inactivate, _nil),
special-pred(value, __nil),
special-pred(compare, __nil),
special-pred(simple-comparison, ___nil),
special-pred(operation-value, ____nil) and
special-pred(sorted, __nil) are allowed as predications.

D-is-a-conjunction*(G)* — **if** G is a goal **then** G is a conjunction of goals.

D-is-a-conjunction*(func(*`&`*, __nil)).*

D-is-a-dewey-number*(D)* — *iff* D is a dewey number.

D-is-a-dewey-number*(nil).*

D-is-a-dewey-number*(X.L)* ⇐
D-is-a-natural*(X),*
D-is-a-dewey-number*(L).*

D-is-a-list-of-dewey-number*(L)* — *iff* L is an abstract list of dewey numbers.

D-is-a-list-of-dewey-number*(nil).*

D-is-a-list-of-dewey-number*(X.L)* ⇐
D-is-a-dewey-number*(X),*
D-is-a-list-of-dewey-number*(L).*

D-is-a-natural*(N)* — *iff* N is a natural number.

D-is-a-natural*(zero).*

D-is-a-natural*(s(X))* ⇐
D-is-a-natural*(X).*

D-is-a-number*(N)* — *iff* N is a number.

D-is-a-number*(X)* ⇐
D-is-an-integer*(X).*

D-is-a-number*(X)* ⇐
D-is-a-float*(X).*

D-is-a-character-code*(C)* — *iff* C is a byte (an integer between 0 and 255) as defined in 7.1.2.2.

D-is-a-character-code*(func(N, nil))* ⇐
L-integer*(N),*
L-integer-less*(*`-1`*, N),*
L-integer-less*(N,* `256`*).*

D-is-an-integer*(func(N, nil))* ⇐
L-integer*(N).*

D-is-a-neg-integer*(I)* — *iff* X is a negative integer.

D-is-a-neg-integer*(func(N, nil))* ⇐
L-integer-less*(N,* `0`*).*

NOTE — References: **L-integer-less** A.3.6

D-is-a-non-neg-int*(I)* — *iff* I is a positive integer.

D-is-a-non-neg-int*(X)* ⇐
D-is-an-integer*(X),*
not **D-is-a-neg-integer***(X).*

D-is-a-float*(R)* — *iff* R is a real.

D-is-a-float*(func(N, nil))* ⇐
L-float*(N).*

D-equal*(X, Y)* — *iff* X and Y are any identical terms built any symbol used in this formal specification.

D-equal*(X, X).*

D-term-to-clause*(T, C)* — **if** T is a term **then** if its principal functor is `':-'`, then C is the corresponding transformed clause according to A.2.3.1 (also 7.6), else C is the clause whose head is T and body *func(*`true`*, nil).*

D-term-to-clause*(func(*`:-`*, H.B.nil), func(*`:-`*, H1.B1.nil))* ⇐
D-term-to-predication*(H, H1),*
D-term-to-body*(B, B1).*

D-term-to-clause*(A, C)* ⇐
not **D-name***(A, func(*`:-`*, nil)),*
D-fact-to-clause*(A, C).*

D-term-to-body*(T, B)* — **if** T is a term **then** C is the transformed body according to A.2.3.1 (also 7.6) **and if** B is a body **then** T is the corresponding term. (Variables `V` in the position of a predication are transformed into `call(V)`)

D-term-to-body*(func(*`&`*, G1.G2.nil), func(*`&`*, G3.G4.nil))* ⇐
D-term-to-body*(G1, G3),*
D-term-to-body*(G2, G4).*

D-term-to-body(*func(;, G1.G2.nil), func(;, G3.G4.nil)*) ⇐
D-term-to-body(*G1, G3*),
D-term-to-body(*G2, G4*).

D-term-to-body(*func(->, G1.G2.nil), func(->, G4.G5.nil)*) ⇐
D-term-to-body(*G1, G4*),
D-term-to-body(*G2, G5*).

D-term-to-body(*T, B*) ⇐
D-term-to-predication(*T, B*).

D-term-to-predication(*F, A), func(F, A)*) ⇐
not **D-equal**(*F, &*),
not **D-equal**(*F, ;*),
not **D-equal**(*F, ->*).

D-term-to-predication(*V, func(*`call`*, V.nil)*) ⇐
L-var(*V*).

D-fact-to-clause(*B, C*) — **if** B is a term **then** if its principal functor is not `':-'`, then C is the clause with head B and body *func(*`true`*, nil)*, else C is identical to B.

D-fact-to-clause(*func(:-, H.B.nil), func(:-, H.B.nil)*).

D-fact-to-clause(*B, func(:-, B.func(*`true`*, nil).nil)*) ⇐
not **D-name**(*B, func(:-, nil)*)).

D-clause-to-pred-indicator(*Cl, PI*) — **if** Cl is a clause **then** PI is the indicator of the head of Cl.

D-clause-to-pred-indicator(*func(:-, H._nil), func(/, At.Ar.nil)*)
⇐
D-name(*H, At*),
D-arity(*H, Ar*).

D-name(*B, K*) — **if** B is a term **then** K is the functor name of B.

D-name(*func(K, _), func(K, nil)*).

D-arity(*B, A*) — **if** B is a term **then** A is the arity of the term B.

D-arity(*func(K, L), func(A, nil)*) ⇐
D-length-list(*L, A*).

NOTE — References: **D-length-list** A.3.4

A.3.2 Predicate indicator

D-is-a-predicate-indicator(*PI*) — *iff* PI is a completely instantiated predicate indicator.

D-is-a-predicate-indicator(*func(/, At.Ar.nil)*) ⇐
D-is-an-atom(*At*),
D-is-an-integer(*Ar*).

NOTE — References: **D-is-an-atom** A.3.4, **D-is-an-integer** A.3.1

D-is-a-pred-indicator-pattern(*PI*) — *iff* PI is a compound term whose functor name is `'/'`, and arity `2`, and its first arguments may be instantiated by an atom and the second by an integer.

D-is-a-pred-indicator-pattern(*func(/, At.Ar.nil)*) ⇐
L-var(*At*),
L-var(*Ar*).

D-is-a-pred-indicator-pattern(*func(/, At.Ar.nil)*) ⇐
L-var(*At*),
D-is-an-integer(*Ar*).

D-is-a-pred-indicator-pattern(*func(/, At.Ar.nil)*) ⇐
L-var(*Ar*),
D-is-an-atom(*At*).

D-is-a-pred-indicator-pattern(*func(/, At.Ar.nil)*) ⇐
D-is-an-atom(*At*),
D-is-an-integer(*Ar*).

NOTE — References: **L-var** A.3.1, **D-is-an-integer** A.3.1, **D-is-an-atom** A.3.4

D-is-a-bip-indicator(*BI*) — *iff* BI is the indicator of a built-in predicate.

D-is-a-bip-indicator(*func(/, At.Ar.nil)*) ⇐
D-is-a-bip(*B*),
D-name(*B, At*),
D-arity(*B, Ar*).

NOTE — References: **D-is-a-bip** A.3.8, **D-name** A.3.1, **D-arity** A.3.1

A.3.3 Forest

A node of the search tree is represented as *nd(I, G, P, Q, E, S, L, M)* where:

— I is a node. If the node is the root of the search-tree, I = *nil*, otherwise the node is the Nth child of another node identified by J, and $I = N\ .\ J$;

— G is an extended goal;

— P is a well-formed extended database (called simply database);

— Q is a list of clauses available for the current computation and denotes the potential choices: i. e. the clauses to be used to build new children;

— E is an environment representing all available streams for the current computation;

— S is a substitution (the local substitution used to obtain this node);

— L is a list of nodes (dewey numbers) containing the active ancestor nodes at this step of resolution (i.e. the `catch` goals which could be chosen if `throw` is called). The

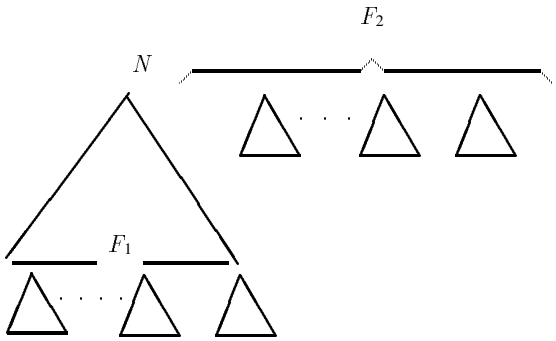

Figure A.9 — The non-empty forest: $for(N, F_1, F_2)$

nodes are ordered in this list from the youngest to the oldest ancestor.

— M is a marker which indicates if the node is completely treated or not (i.e. if the sub-search tree has been completely developed), and is either *partial* or *complete*.

The partially visited search tree is represented by a forest. A forest is either:

— *vid* : the empty forest; or

— *for(N, F1, F2)* : a non-empty forest, where N is a labelled node, and *F1* and *F2* are forests. A forest term denotes a sequence of $n+1$ trees if $F2$ has n trees as depicted in Figure A.9.

A.3.3.1 Forest structure

D-is-a-forest*(F)* — *iff* F is a forest.

D-is-a-forest*(vid).*

D-is-a-forest*(for(N, F1, F2))* ⇐
D-is-a-label-node*(N),*
D-is-a-forest*(F1),*
D-is-a-forest*(F2).*

D-is-a-label-node*(nd(I, G, P, Q, E, S, L, M))* ⇐
D-is-a-dewey-number*(I),*
D-is-a-body*(G),*
D-is-a-database*(P),*
D-is-a-packet-of-clauses*(Q),*
D-is-an-environment*(E),*
D-is-a-substitution*(S),*
D-is-a-list-of-dewey-number*(L),*
D-is-a-visit-mark*(M).*

NOTE — References: **D-is-a-dewey-number** A.3.1, **D-is-a-body** A.3.1, **D-is-a-database** A.3.1, **D-is-a-packet-of-clauses** A.3.1, **D-is-an-environment** A.3.7, **D-is-a-substitution** A.3.5, **D-is-a-list-of-dewey-number** A.3.1,

D-is-a-visit-mark*(complete).*

D-is-a-visit-mark*(partial).*

A.3.3.2 Root manipulation

D-root*(F,N)* — **if** F is a forest **then** N is one of the roots of F.

D-root*(for(N1, F1, F2), N)*⇐
D-equal*(N1,nd(N, _ _ _ _ _ _ _ _)).*

D-root*(for(N, F1, F2), M)* ⇐
D-root*(F2, M).*

NOTE — References: **D-equal** A.3.1

D-lastroot*(F,N)* — **if** F is a forest **then** N is the last (right-most) root of F.

D-lastroot*(for(N1, F1, vid), N)* ⇐
D-equal*(N1,nd(N, _ _ _ _ _ _ _ _)).*

D-lastroot*(for(N, F1, F2), M)* ⇐
D-lastroot*(F2, M).*

NOTE — References: **D-equal** A.3.1

D-number-of-root*(F,J)* — **if** F is a forest **then** F has J roots.

D-number-of-root*(vid, zero).*

D-number-of-root*(for(N, F1, F2), s(J))* ⇐
D-number-of-root*(F2, J).*

D-addroot*(F, N, F1)* — **if** F is a forest, and N a label node **then** $F1$ is F with a new root labelled by N at the right-most position.

D-addroot*(vid, N, for(N, vid, vid)).*

D-addroot*(for(M, F1, F2), N, for(M, F1, F3))* ⇐
D-addroot*(F2, N, F3).*

A.3.3.3 Children

D-child*(N, F, M)* — **if** F is a forest **then** M and N are nodes of F and M is one of the children of N.

D-child*(N, for(N1, F1, F2), M)* ⇐
D-equal*(N1,nd(N, _ _ _ _ _ _ _ _)),*
D-root*(F1, M).*

D-child*(N, for(N1, F1, F2), M)* ⇐
D-child*(N, F1, M).*

D-child*(N, for(N1, F1, F2), M)* ⇐
D-child*(N, F2, M).*

NOTE — References: **D-equal** A.3.1, **D-root** A.3.3.2

D-has-a-child*(N, F)* — **if** F is a forest and N is a node **then** N is a node of F and N has a child in F.

D-has-a-child*(N, F)* ⇐
D-child*(N, F, _).*

D-number-of-child*(N, F, J)* — **if** F is a forest **then** N is a node of F and N has J children.

D-number-of-child(*N, for(N1, F1, F2), J)* ⇐
D-equal(*N1,nd(N, _ _ _ _ _ _ _)*),
D-number-of-root(*F1, J*).

D-number-of-child(*M, for(N, F1, F2), J)* ⇐
D-number-of-child(*M, F1, J*).

D-number-of-child(*M, for(N, F1, F2), J)* ⇐
D-number-of-child(*M, F2, J*).

NOTE — References: **D-equal** A.3.1, **D-number-of-root** A.3.3.2

D-lastchild(*N, F, M*) — **if** F is a forest **then** M and N are nodes of F and M is the last (right-most) child of N.

D-lastchild(*N, for(N1, F1, F2), M)* ⇐
D-equal(*N1,nd(N, _ _ _ _ _ _ _)*),
D-lastroot(*F1, M*).

D-lastchild(*N, for(N1, F1, F2), M)* ⇐
D-lastchild(*N, F1, M*).

D-lastchild(*N, for(N1, F1, F2), M)* ⇐
D-lastchild(*N, F2, M*).

NOTE — References: **D-equal** A.3.1, **D-lastroot** A.3.3.2

D-parent-or-root(*M, F, P*) — **if** F is a forest **then** M and P are nodes of F and if M is the root of F then P is the root of F, else, P is the parent of M.

D-parent-or-root(*M, F, M)* ⇐
D-root(*F, M*).

D-parent-or-root(*M, F, P)* ⇐
not **D-root**(*F, M*),
D-parent(*M, F, P*).

NOTE — References: **D-lastroot** A.3.3.2

D-parent(*M, F, P*) — **if** F is a forest **then** M and P are nodes of F and P is the parent of M.

D-parent(*M, F, M)* ⇐
D-root(*F, M*).

D-parent(*M, F, P)* ⇐
not **D-root**(*F, M*),
D-child(*P, F, M*).

NOTE — References: **D-lastroot** A.3.3.2

A.3.3.4 Selector predicates

D-label of node *N* **in** *F* **is** *Nl* — **if** N is a node of the forest F **then** Nl is the node label of N.

D-label of node *N* **in** *for(Nl, F1, F2)* **is** *Nl* ⇐
D-equal(*Nl, nd(N, _ _ _ _ _ _ _)*).

D-label of node *N* **in** *for(Ml, F1, F2)* **is** *Nl* ⇐
D-label of node *N* **in** *F1* **is** *Nl*.

D-label of node *N* **in** *for(Ml, F1, F2)* **is** *Nl* ⇐
D-label of node *N* **in** *F2* **is** *Nl*.

D-goal of node *N* **in** *F* **is** *G* — **if** N is a node of the forest F **then** G is the goal in the corresponding label node in F.

D-goal of node *N* **in** *F* **is** *G* ⇐
D-label of node *N* **in** *F* **is** *nd(N, G, _ _ _ _ _ _)*.

and analogous:

D-database of node *N* **in** *F* **is** *P* — **if** N is a node of the forest F **then** P is the database in the corresponding label node in F.

D-choice of node *N* **in** *F* **is** *Q* — **if** N is a node of the forest F **then** Q is the choice in the corresponding label node in F.

D-environment of node *N* **in** *F* **is** *E* — **if** N is a node of the forest F **then** E is the environment in the corresponding label node in F.

D-substitution of node *N* **in** *F* **is** *S* — **if** N is a node of the forest F **then** S is the substitution in the corresponding label node in F.

D-active catchers of node *N* **in** *F* **is** *L* — **if** N is a node of the forest F **then** L is the active catcher list in the corresponding label node in F.

D-visit mark of node *N* **in** *F* **is** *M* — **if** N is a node of the forest F **then** M is visit mark in the corresponding label node in F.

D-root-database-and-env(*F, P, E*) — **if** F is a non-empty forest **then** P is the current database, and E the current environment at the first root of F.

D-root-database-and-env(*for(N, _ _), P, E)* ⇐
D-equal(*N, nd(_ _ P, _ E, _ _ _)*).

NOTE — References: **D-equal** A.3.1

A.3.3.5 Field node updates

D- *N1* **is** *N2* **where database is** *P* — **if** $N2$ is a node label and P a database **then** $N1$ is the same node label except that its database field is P.

D- *N1* **is** *N2* **where database is** *P* ⇐
D-equal(*N2, nd(N, G, _ Q, E, S, L, M)*),
D-equal(*N1, nd(N, G, P, Q, E, S, L, M)*).

and analogous:

D- *N1* **is** *N2* **where choices are** *C* — **if** $N2$ is a node label **then** $N1$ is the same node label except that its choice field is C.

D- *N1* **is** *N2* **where environment is** *E* — **if** $N2$ is a node label **then** $N1$ is the same node label except that its environment field is E.

D- *N1* **is** *N2* **where substitution is** *S* — **if** $N2$ is a node label **then** $N1$ is the same node label except that its substitution field is S.

D- *N1* **is** *N2* **where active catchers are** *L* — **if** $N2$ is a node label **then** $N1$ is the same node label except that its active catcher field is L.

D- *N1* **is** *N2* **where visit mark is** *M* — **if** $N2$ is a node label and M is a visit mark **then** $N1$ is the same node label except that its visit mark field is M.

NOTE — References: **D-equal** A.3.1

A.3.3.6 Label of node updates

D-modify-database(*F1, Newpg, F2*) — **if** $F1$ is a forest and *Newpg* the new database **then** $F2$ is identical to $F1$ except that in all nodes on the right most branch of $F1$, the old database is replaced by *Newpg*.

D-modify-database(*vid, _, vid*).

D-modify-database(*for(N, F1, vid), Newpg, for(N1, F2, vid)*) ⇐
D- *N1* **is** *N* **where database is** *Newpg*,
D-modify-database(*F1, Newpg, F2*).

D-modify-database(*for(N, F1, F2), Newpg, for(N, F1, F3)*) ⇐
not **D-equal**(*F2, vid*),
D-modify-database(*F2, Newpg, F3*).

NOTE — References: **D- _ is _ where database is _** A.3.3.4, **D-equal** A.3.1

D-modify-environment(*F1, Newenv, F2*) — **if** $F1$ is a forest and *Newenv* the new environment **then** $F2$ is $F1$ where, in all nodes on the right most branch of $F1$, the old environment is replaced by *Newenv*.

D-modify-environment(*vid, _, vid*).

D-modify-environment(*for(N, F1, vid), Newenv, for(N1, F2, vid)*) ⇐
D- *N1* **is** *N* **where environment is** *Newenv*,
D-modify-environment(*F1, Newenv, F2*).

D-modify-environment(*for(N, F1, F2), Newenv, for(N, F1, F3)*) ⇐
not **D-equal**(*F2, vid*),
D-modify-environment(*F2, Newenv, F3*).

NOTE — References: **D- _ is _ where environment is _** A.3.3.4, **D-equal** A.3.1

D-modify-node(*F1, Nl1, Nl2, F2*) — **if** $Nl1$ is a node label of the forest $F1$ and $Nl2$ a new node label corresponding to the same node **then** $F2$ is $F1$ except that $Nl2$ replaces $Nl1$.

D-modify-node(*for(Nl, F1, F2), Nl, Nl1, for(Nl1, F1, F2)*).

D-modify-node(*for(Nl, F1, F2), Nl1, Nl2, for(Nl, F3, F2)*) ⇐
D-modify-node(*F1, Nl1, Nl2, F3*).

D-modify-node(*for(Nl, F1, F2), Nl1, Nl2, for(Nl, F1, F3)*) ⇐
D-modify-node(*F2, Nl1, Nl2, F3*).

D-create-child(*F1, Nl1, Nl2, F2*) — **if** $Nl1$ is a node label of the forest $F1$ and $Nl2$ a new node label corresponding to a new youngest child of $Nl1$ **then** $F2$ is $F1$ in which $Nl2$ is the new youngest child of $Nl1$.

D-create-child(*for(Nl, F1, F2), Nl, Nl1, for(Nl, F3, F2)*)⇐
D-addroot(*F1, Nl1, F3*).

D-create-child(*for(Nl, F1, F2), Nl1, Nl2, for(Nl, F3, F2)*) ⇐
D-create-child(*F1, Nl1, Nl2, F3*).

D-create-child(*for(Nl, F1, F2), Nl1, Nl2, for(Nl, F1, F3)*) ⇐
D-create-child(*F2, Nl1, Nl2, F3*).

NOTE — References: **D-addroot** A.3.3.2

A.3.4 Abstract lists, atoms, characters and lists

An abstract list has the form $B1.B2.....nil$ where the elements may be terms (it is thus an *arg-list*), clauses, extended goals, streams, dewey numbers, naturals or substitutions.

A **list** is the abstract representation of a concrete list of the form $[t_1, \cdots, t_n]$.

D-is-an-atom(*A*) — *iff* A is an atom.

D-is-an-atom(*func(N, nil)*) ⇐
L-atom(*N*).

NOTE — References: **L-atom** A.3.1,

D-is-atomic(*A*) — **if** A is a term **then** A is a constant (it has the form: *func(_, nil)*).

D-is-atomic(*A*) ⇐
D-is-an-atom(*A*).

D-is-atomic(*A*) ⇐
D-is-a-number(*A*).

D-char-instantiated-list(*L*) — *iff* L is a list whose elements are variables or characters.

D-char-instantiated-list(*func(*`[]`*, nil)*).

D-char-instantiated-list(*func(., X.L.nil)*) ⇐
L-var(*X*),
D-char-instantiated-list(*L*).

D-char-instantiated-list(*func(., X.L.nil)*) ⇐
D-is-a-char(*X*),
D-char-instantiated-list(*L*).

NOTE — References: **L-var** A.3.1, **D-is-a-char** A.3.7

D-is-a-partial-char-list(*L*) — *iff* L is a partial list of chars.

D-is-a-partial-char-list(*func(., X.L.[])*) ⇐
D-is-a-char(*X*),
L-var(*L*).

D-is-a-partial-char-list(*func(., X.L.[])*) ⇐
D-is-a-char(*X*),
D-is-a-partial-char-list(*L*).

NOTE — References: **L-var** A.3.1, **D-is-a-char** A.3.7

D-code-instantiated-list(*L*) — *iff* L is a list whose elements are variables or codes.

D-code-instantiated-list(*func(*`[]`*, nil)*).

D-code-instantiated-list(*func(., X.L.nil)*) ⇐
L-var(*X*),
D-code-instantiated-list(*L*).

D-code-instantiated-list(*func(., X.L.nil)*) ⇐
D-is-a-character-code(*X*),
D-code-instantiated-list(*L*).

NOTE — References: **L-var** A.3.1, **D-is-a-character-code** A.3.1

D-is-a-partial-code-list*(L)* — *iff* L is a partial list of codes.

D-is-a-partial-code-list*(func(., X.L.[]))* ⇐
D-is-a-character-code*(X),*
L-var*(L).*

D-is-a-partial-code-list*(func(., X.L.[]))* ⇐
D-is-a-character-code*(X),*
D-is-a-partial-code-list*(L).*

NOTE — References: **L-var** A.3.1, **D-is-a-character-code** A.3.7

D-is-a-list*(L)* — *iff* L is a list.

D-is-a-list*(func(*`[]`*, nil)).*

D-is-a-list*(func(.,* `X.L.nil`*))* ⇐
D-is-a-term*(X),*
D-is-a-list*(L).*

NOTE — References: **D-is-a-term** A.3.1

D-is-a-partial-list*(L)* — *iff* L is a partial list of terms.

D-is-a-partial-list*(func(.,* `X.L.nil`*))* ⇐
D-is-a-term*(X),*
L-var*(L).*

D-is-a-partial-list*(func(.,* `X.L.nil`*))* ⇐
D-is-a-term*(X),*
D-is-a-partial-list*(L).*

NOTE — References: **L-var** A.3.1, **D-is-a-term** A.3.1

D-conc*(L1, L2, L3)* — **if** $L1$ and $L2$ are abstract lists **then** $L3$ is the concatenation of $L1$ and $L2$, **and if** $L3$ is an abstract list **then** $L1$ and $L2$ are abstract lists such that $L3$ is the concatenation of $L1$ and $L2$.

D-conc*(nil, L, L).*

D-conc*(X.L1, L2, X.L3)* ⇐
D-conc*(L1, L2, L3).*

D-delete*(L, A, L1)* — **if** L is an abstract list **then** A is the first occurrence of A in L and $L1$ is L where this occurrence is deleted.

D-delete*(A.L, A, L).*

D-delete*(A.L, B, A.L1)* ⇐
not **D-equal***(A, B),*
D-delete*(L, B, L1).*

NOTE — References: **D-equal** A.3.1

D-one-delete*(L, A, L1)* — **if** L is an abstract list **then** A is an element of L and $L1$ is L where this element is deleted.

D-one-delete*(A.L, A, L).*

D-one-delete*(A.L, B, A.L1)* ⇐
D-one-delete*(L, B, L1).*

NOTE — References: **D-equal** A.3.1

D-member*(X, L)* — **if** L is an abstract list **then** X is an element of L.

D-member*(X, X.L).*

D-member*(X, Y.L)* ⇐
D-member*(X, L).*

D-position*(X, L, N)* — **if** L is an abstract list **then** N is a concrete integer and X is the N^{th} element of L.

D-position*(X, X.L,* `1`*).*

D-position*(Y, X.L, N)* ⇐
L-integer-plus*(P,* `1`*, N),*
D-position*(Y, L, P).*

NOTE — References: **L-integer-plus** A.3.6

D-length-list*(L, N)* — **if** L is an abstract list **then** N is the concrete integer corresponding to the number of elements of L.

D-length-list*(nil,* `0`*).*

D-length-list*(X.L, N)* ⇐
D-length-list*(L, P),*
L-integer-plus*(P,* `1`*, N).*

NOTE — References: **L-integer-plus** A.3.6

D-same-length*(L1, L2)* — **if** $L1$ and $L2$ are abstract lists **then** they have the same number of elements.

D-same-length*(nil, nil).*

D-same-length*(X.L1, Y.L2)* ⇐
D-same-length*(L1, L2).*

D-buildlist-of-var*(L, N)* — *iff* L is an abstract list of length N whose elements are distinct variables.

D-buildlist-of-var*(nil,* `0`*).*

D-buildlist-of-var*(X.L, N)* ⇐
D-buildlist-of-var*(L, P),*
L-integer-plus*(P,* `1`*, N),*
L-var*(X),*
not **D-member***(X, L).*

NOTE — References: **L-integer-plus** A.3.6, **L-var** A.3.1, **D-member** A.3.4

D-transform-list*(L1, L2)* — **if** $L1$ is an arg-list **then** $L2$ is the corresponding list of the elements of $L1$, **and if** $L2$ is a list of terms **then** $L1$ is an arg-list formed by terms in $L2$.

D-transform-list*(nil, func(*`[]`*, nil)).*

D-transform-list*(Term.L1, func(., Term.L2.nil))* ⇐
D-transform-list*(L1, L2).*

L-var-order*(X,Y)* — *iff* X and Y are variables such that X term-precedes Y (this order is implementation dependent, see 7.2.1).

L-char-code*(X, Y)* — *iff* X is a concrete character and Y its integer code (see `char_code/2` bip).

L-atom-chars*(X, Y)* — *iff* X is a concrete atom and Y the arg-list of characters such that the juxtaposition of their concrete form corresponds to X (see `atom_chars/2` bip).

L-atom-codes*(X, Y)* — *iff* X is a concrete atom and Y the arg-list of character codes such that the juxtaposition of the corresponding characters of these codes corresponds to X (see `atom_codes/2` bip).

L-number-chars*(X, Y)* — *iff* X is a concrete number and Y the arg-list of characters corresponding to a character sequence of X (see `number_chars/2` bip).

L-number-codes*(X, Y)* — *iff* X is a concrete number and Y the arg-list of character codes corresponding to a character sequence of X (see `number_codes/2` bip).

L-atom-order*(X, Y)* — *iff* X and Y are concrete atoms such that X is less than Y in the term order (see 7.2).

L-sorted*(X, Y)* — *iff* X and Y are lists and Y is the list X sorted according to term ordered (7.2) with duplicates removed except the same order is used when two variables are compared. (see also 7.1.6.5)

A.3.5 Substitutions and unification

D-is-a-substitution*(S)* — *iff* S is a substitution.

NOTE — No formal representation is defined for substitutions except for the empty substitution which is denoted *empsubs*.

L-unify*(X, Y, S)* — *iff* X and Y are $NSTO$ terms and S is one of their most general unifier (see clause 7.3).

L-unify-occur-check*(X, Y, S)* — *iff* X and Y are terms and S is one of their most general unifier (see clause 7.3).

L-unify-members-list*(L, S)* — *iff* S is a most general unifier of all the elements of the abstract list of terms L.

D-unifiable*(X, Y)* — *iff* X and Y are $NSTO$ terms and they are unifiable terms (see clause 7.3).

D-unifiable*(T, T1)* ⇐
L-unify*(T, T1, _)*.

L-not-unifiable*(X, Y)* — *iff* X and Y are NSTO terms and they are not unifiable (see clause 7.3).

L-occur-in*(T1, T2)* — *iff* $T1$ and $T2$ are terms and some variables of $T1$ occur in $T2$.

L-not-occur-in*(T1, T2)* — *iff* $T1$ and $T2$ are terms and do not share any variable.

L-composition*(S1, S2, S3)* — *iff* $S1$, $S2$ and $S3$ are substitutions on terms where $S3$ is the composition of $S1$ and $S2$ (see clause 7.3).

L-instance*(T1, S, T2)* — *iff* $T1$ is an any-term, S is a substitution and $T2$ is the any-term obtained by applying the substitution S to $T1$ (applying the substitution modifies only the concrete variables occurring in $T1$ (3.74)).

NOTE — *any-term* denotes any kind of term that is to say terms built with any functor used in the formal specification language.

L-rename*(F, X, Y)* — *iff* F is a search tree, and X and Y are any-terms such that Y is a copy of X except its variables are renamed so that they do not occur in F.

L-rename-except*(F, V, X, Y)* — *iff* F is a search tree, V a term and X and Y are any-terms such that Y is identical X except all its variables which do not occur in V are renamed so that they do not occur in F.

L-variants*(T1, T2)* — *iff* $T1$ and $T2$ are variant terms according to definition 7.1.6.1.

D-compose-list*(L, S, L1)* — **if** L is an abstract list of substitutions and S a substitution **then** $L1$ is the abstract list of substitutions obtained by composition with S of each substitution of L.

D-compose-list*(nil, S, nil)*.

D-compose-list*(S1.L1, S, S2.L2)* ⇐
L-composition*(S1, S, S2)*,
D-compose-list*(L1, S, L2)*.

A.3.6 Arithmetic

L-integer-less*(X, Y)* — *iff* X and Y are concrete integers such that $X < Y$.

L-integer-plus*(X, Y, Z)* — *iff* X, Y, and Z are concrete integers such that $Z = X + Y$.

L-float-less*(X, Y)* — *iff* X and Y are concrete reals such that $X < Y$.

L-error-in-expression*(E, T)* — *iff* E is an erroneous elementary expression and T is the type of the corresponding error (see 9).

L-value*(E, V)* — *iff* E is an elementary arithmetic expression (see 9.1) which can be successfully evaluated and V is the number corresponding to its value.

L-arithmetic-comparison*(X, Op, Y)* — *iff* X and Y denote numbers and Op an arithmetic comparison operator such that $X\ Op\ Y$ following the definition (see 8.7.1).

A.3.7 Difference lists and environments

D-is-an-environment*(E)* — *iff* E is an environment with all flags (defined only once) and all open streams (all streams have different stream names).

D-is-an-environment*(env(PF, IF, OF, IFL, OFL))* ⇐
D-is-a-list-of-flags*(PF)*,
D-is-a-stream*(IF)*,
D-is-a-stream*(OF)*,
D-is-a-list-of-streams*(LIF)*,
D-is-a-list-of-streams*(LOF)*.

D-is-a-list-of-flags*(PF)* — *iff* PF is an abstract list of flag terms.

D-is-a-list-of-flags*(nil)*.

D-is-a-list-of-flags*(F.PF)* ⇐
D-is-a-flag-term*(F)*,
D-is-a-list-of-flags*(PF)*.

D-is-a-flag-term*(Flag)* — *iff* $Flag$ is a term representing a flag.

D-is-a-flag-term(*func(flag,*
$name.actual-value_{Name}.possible-values_{Name}$.*nil)) ⇐*
D-is-a-flag(*Name).*

Where $name$, $actual-value_{name}$ and $possible-values_{name}$ stand for the name of a flag and its actual value and possible values as defined in clause 7.11,

with $name \in$ {`bounded`,
`max_integer`,
`min_integer`,
`integer_rounding_function`,
...
`debug`,
`max_arity`,
`undefined_predicate`}

D-is-a-flag(*Flag*) — *iff* $Flag$ is a flag term as defined in 7.11.

D-is-a-flag(*func(flag-name,nil)).*

with $flag-name \in$ {`bounded`,
`max_integer`,
`min_integer`,
`integer_rounding_function`,
...
`debug`,
`max_arity`,
`undefined_predicate`}

D-is-a-flag-value(*F, Flag, Value*) — **if** F is a forest and $Flag$ is a flag **then** $Value$ is a valid value of $Flag$ in F.

D-is-a-flag-value(*F, Flag, Value*) ⇐
D-root-database-and-env(*F, _, env(PF, _, _, _, _)),*
D-corresponding-flag-term(*Flag, PF, T),*
D-equal(*T, func(flag,_,_V.nil)),*
D-transform-list(*V,V1),*
D-member(*Value, V1).*

NOTE — References: **D-root-database-and-env** A.3.3.4, **D-equal** A.3.1, **D-transform-list** A.3.4, **D-member** A.3.4

D-corresponding-flag-term(*Flag, PF, T*) — **if** $Flag$ is a flag and PF is a non empty abstract list of flag terms **then** T is the flag term corresponding to $Flag$.

D-corresponding-flag-term(*Flag, func(flag, Flag.V.LV.nil).PF, func(flag, Flag.V.LV.nil)).*

D-corresponding-flag-term(*Flag, T1.PF, T*) ⇐
D-corresponding-flag-term(*Flag, PF, T).*

D-is-a-stream(*S*) — *iff* S represents a stream.

D-is-a-stream(*stream(S, L)*) ⇐
L-stream-name(*S),*
D-is-a-difference-list-of-char(*L).*

L-stream-name(*X*) — *iff* X is a ground term denoting a stream identifier defined in 7.10.2.1.

L-stream-property(*SP*) — *iff* SP is a stream property as defined in clause 7.10.2.13.

D-is-a-list-of-streams(*L*) — *iff* L represents an abstract list of streams.

D-is-a-list-of-streams(*nil).*

D-is-a-list-of-streams(*X.L*) ⇐
D-is-a-stream(*X),*
D-is-a-list-of-streams(*L).*

D-is-an-io-mode(*M*) — *iff* M is an input/output mode.

D-is-an-io-mode(*func(read, nil)).*

D-is-an-io-mode(*func(write, nil)).*

D-is-an-io-mode(*func(append, nil)).*

D-is-a-difference-list-of-char(*L-L*) ⇐
D-is-a-list-of-char(*L).*

D-is-a-difference-list-of-char(*C.L1-L2*) ⇐
D-is-a-char(*C),*
D-is-a-difference-list-of-char(*L1-L2).*

D-is-a-list-of-char(*nil).*

D-is-a-list-of-char(*C.L*) ⇐
D-is-a-char(*C),*
D-is-a-list-of-char(*L).*

D-is-a-char(*func(C, nil)*) ⇐
L-char(*C).*

L-char(*X*) — *iff* X is a concrete atom of length `1`.

L-io-option(*F, Op, V*) — **if** F is a stream, and Op a stream option **then** V is the value of option Op of the stream F as defined in 3.122.

A.3.8 Built-in predicates and packets

D-is-a-bip(*B*) — **if** B is a predication **then** it is the predication of a built-in predicate.

D-is-a-bip(*B*) ⇐
D-is-a-term-unification-bip(*B).*

D-is-a-bip(*B*) ⇐
D-is-a-term-comparison-bip(*B).*

D-is-a-bip(*B*) ⇐
D-is-an-all-solution-bip(*B).*

D-is-a-bip(*B*) ⇐
D-is-a-type-testing-bip(*B).*

D-is-a-bip(*B*) ⇐
D-is-a-term-creation-decomposition-bip(*B).*

D-is-a-bip(*B*) ⇐
D-is-a-database-bip(*B).*

D-is-a-bip(*B*) ⇐
D-is-an-arithmetic-bip(*B).*

D-is-a-bip(*B*) ⇐
D-is-an-atom-processing-bip(*B).*

D-is-a-bip(*B*) ⇐
D-is-an-input-output-bip(*B).*

D-is-a-bip*(B)* $\Leftarrow$
D-is-a-logic-control-bip*(B).*

D-is-a-bip*(B)* $\Leftarrow$
D-is-a-control-construct-bip*(B).*

D-is-a-bip*(B)* $\Leftarrow$
D-is-an-environment-bip*(B).*

D-is-a-term-unification-bip(B).

with $B \in$ {func(=, __.nil),
func(unify_with_occurs_check, __.nil),
func(\=, __.nil)}

D-is-a-term-comparison-bip(B).

with $B \in$ {func(==, __.nil).
func(\==, __.nil),
func(@<, __.nil),
func(@=<, __.nil),
func(@>, __.nil),
func(@>=, __.nil)}

D-is-an-all-solution-bip(B).

with $B \in$ {func(findall, ___.nil),
func(bagof, ___.nil),
func(setof, ___.nil)}

D-is-a-type-testing-bip(B).

with $B \in$ {func(var, _.nil),
func(nonvar, _.nil),
func(atom, _.nil),
func(atomic, _.nil),
func(number, _.nil),
func(integer, _.nil),
func(real, _.nil),
func(compound, _.nil)}

D-is-a-term-creation-decomposition-bip(B).

with $B \in$ {func(arg, ___.nil),
func(functor, ___.nil),
func(=.., __.nil),
func(copy_term, __.nil)}

D-is-a-database-bip*(B)* $\Leftarrow$
D-is-a-clause-retrieval-information-bip*(B).*

D-is-a-database-bip*(B)* $\Leftarrow$
D-is-a-clause-creation-destruction-bip*(B).*

D-is-a-clause-retrieval-information-bip(B).

with $B \in$ {func(clause, __.nil),
func(current_predicate, _.nil)}

D-is-a-clause-creation-destruction-bip(B).

with $B \in$ {func(asserta, _.nil),
func(assertz, _.nil),
func(retract, _.nil),
func(abolish, _.nil)}

D-is-an-arithmetic-bip(B).

with $B \in$ {func(is, __.nil),
func(=:=, __.nil),
func(=\=, __.nil),
func(<, __.nil),
func(>, __.nil),
func(=<, __.nil),
func(>=, __.nil)}

D-is-an-atom-processing-bip(B).

with $B \in$ {func(atom_length, __.nil),
func(atom_concat, ___.nil),
func(sub_atom, ____.nil),
func(atom_chars, __.nil),
func(atom_codes, __.nil),
func(number_chars, __.nil),
func(number_codes, __.nil),
func(char_code, __.nil)}

D-is-an-input-output-bip*(B)* $\Leftarrow$
D-is-a-char-input-output-bip*(B).*

D-is-an-input-output-bip*(B)* $\Leftarrow$
D-is-a-term-input-output-bip*(B).*

D-is-a-char-input-output-bip(B).

with $B \in$ {func(current_input, _.nil),
func(current_output, _.nil),
func(set_input, _.nil),
func(set_output, _.nil),
func(get_char, _.nil),
func(get_char, __.nil),
func(get_code, _.nil),
func(get_code, __.nil),
func(at_end_of_stream, nil),
func(at_end_of_stream, _.nil),
func(put_char, _.nil),
func(put_char, __.nil),
func(put_code, _.nil),
func(put_code, __.nil),
func(nl, nil),
func(nl, _.nil)}

D-is-a-term-input-output-bip(B).

with $B \in$ {func(read_term, ___.nil),
func(read_term, __.nil),
func(read, _.nil),
func(read, __.nil),
func(write_term, ___.nil),
func(write_term, __.nil),
func(write, _.nil),
func(write, __.nil),
func(writeq, _.nil),
func(writeq, __.nil),
func(write_canonical, _.nil),
func(write_canonical, __.nil),
func(op, ___.nil),
func(current_op, ___.nil)}

D-is-a-logic-control-bip(B).

with $B \in$ {func(fail_if, _.nil),
func(once, _.nil),
func(repeat, nil)}

D-is-a-control-construct-bip*(func(*!*, D.nil))* $\Leftarrow$
D-is-a-dewey-number*(D).*

D-is-a-control-construct-bip(B).

with $B \in$ {func(`;`, _._.nil),
func(`->`, _._.nil),
func(`true`, nil),
func(`fail`, nil),
func(`!`, nil),
func(`call`, _.nil),
func(`catch`, _._._.nil),
func(`throw`, _.nil)}

NOTE — References: **D-is-a-dewey-number** A.3.1

D-is-an-environment-bip(B).

with $B \in$ {func(`halt`, nil),
func(`halt`, _.nil),
func(`current_prolog_flag`, _._.nil),
func(`set_prolog_flag`, _._.nil)}

D-boot-bip*(B)* — **if** B is a predication **then** it is the predication of a boot-strapped built-in predicate.

with $B \in$ {func(`';'`, _._.nil),
func(`'->'`, _._.nil),
func(`fail`, nil),
func(`fail_if`, _.nil),
func(`get_char` , _.nil),
func(`get_code` , _.nil),
func(`number`, _.nil),
func(`is`, _._.nil),
func(`once`, _.nil),
func(`put_char`, _.nil),
func(`put_code`, _.nil),
func(`at_end_of_stream`, nil),
func(`read`, _.nil),
func(`repeat`, nil),
func(`sub_atom`, _._._._.nil),
func(`write`, _.nil),
func(`nl`, nil),
func(`nl`, _.nil),
func(`=:=`, _._.nil),
func(`=`, _._.nil),
func(`>`, _._.nil),
func(`<`, _._.nil),
func(`>=`, _._.nil),
func(`<=`, _._.nil)}

D-database-backtrack-bip*(B)* — **if** B is a predication **then** it is the predication of a re-executable built-in predicate on database.

with $B \in$ {func(`clause`, _._.nil),
func(`current_predicate`, _.nil),
func(`retract`, _.nil)}

D-is-a-subst-bip*(B)* — **if** B is a predication **then** it is the predication of a class of built-in predicates which do not affect the database or environment (the result of executing such a bip is either success leading to a substitution, or failure).

D-is-a-subst-bip*(B)* ⇐
D-is-a-term-unification-bip*(B).*

D-is-a-subst-bip*(B)* ⇐
D-is-a-type-testing-bip*(B).*

D-is-a-subst-bip*(B)* ⇐
D-is-a-term-creation-decomposition-bip*(B).*

D-is-a-subst-bip*(B)* ⇐
D-is-an-arithmetic-bip*(B).*

D-is-a-subst-bip*(B)* ⇐
D-is-a-term-comparison-bip*(B).*

D-is-a-subst-bip*(B)* ⇐
D-is-an-atom-processing-bip*(B).*

D-packet*(P, A, Q)* — **if** P is a database and A is a predication **then** Q is the list of clauses defining the procedure corresponding to A or all clauses of P if A corresponds to a re-executable built-in predicate.

D-packet*(nil, A, nil)* ⇐
not **D-is-a-bip***(A),*
not **D-is-a-special-pred***(A).*

D-packet*(DB, A, Q)* ⇐
not **D-is-a-bip***(A),*
D-name*(A, F),*
D-arity*(A, N),*
corresponding-pred-definition*(func(/, F.N.nil), DB, def(_, _, Q), _).*

D-packet*(DB, A, nil)* ⇐
not **D-is-a-bip***(A),*
D-name*(A, F),*
D-arity*(A, N),*
not **exist-corresponding-pred-definition***(func(/, F.N.nil), DB).*

D-packet*(DB, A, Q)* ⇐
D-is-a-bip*(A),*
D-database-backtrack-bip*(A),*
D-all-clauses*(DB, Q).*

D-packet*(_, A, nil)* ⇐
D-is-a-bip*(A),*
not **D-database-backtrack-bip***(A),*
not **D-boot-bip***(A).*

D-packet*(_, SP, nil).*

with $SP \in$ {special-pred(inactivate, _.nil),
special-pred(system-error-action, nil),
special-pred(halt-system-action, nil),
special-pred(halt-system-action, _.nil),
special-pred(value, _._.nil),
special-pred(compare, _._.nil),
special-pred(simple-comparison, _._._.nil),
special-pred(operation-value, _._._._.nil),
special-pred(sorted, _._.nil) }

NOTE — Further clauses for packet are given (implicitly) by the boot-strap definitions of so defined built-in predicates.

NOTE — References: **D-is-a-special-pred** A.3.1, **D-name** A.3.1, **D-arity** A.3.1, **corresponding-pred-definition** A.4.1.51, **exist-corresponding-pred-definition** A.4.1.52

D-all-clauses*(DB, Q)* — **if** DB is a database **then** Q is the list of clauses defining all the predicates of DB.

D-all-clauses*(nil, nil).*

D-all-clauses*(def(_, _, Q1).DB, Q)* ⇐
D-all-clauses*(DB, Q2),*
D-conc*(Q1, Q2, Q).*

NOTE — References: **D-conc** A.3.4

D-delete-packet*(P1, PI, P2)* — **if** $P1$ is an abstract list of clauses and PI a predicate indicator pattern **then** $P2$ is $P1$ from which all the clauses of the procedure whose predicate indicator unifies with PI have been removed.

D-delete-packet*(nil, PI, nil).*

D-delete-packet*(func(:-, H._nil).P1, PI, P2)* ⇐
D-name*(H, At),*
D-arity*(H, Ar),*
D-unifiable*(PI, func(/, At.Ar.nil)),*
D-delete-packet*(P1, PI, P2).*

D-delete-packet*(func(:-, H.B.nil).P1, PI, func(:-, H.B.nil).P2)* ⇐
D-name*(H, At),*
D-arity*(H, Ar),*
L-not-unifiable*(PI, func(/, At.Ar.nil)),*
D-delete-packet*(P1, PI, P2).*

NOTE — References: **D-name** A.3.1, **D-arity** A.3.1, **D-unifiable** A.3.5, **L-not-unifiable** A.3.5

D-same-predicate*(A, B)* — **if** A and B are predications **then** they correspond to the same predicate.

D-same-predicate*(A, B)* ⇐
D-equal*(A, func(N, L1)),*
D-equal*(B, func(N, L2)),*
D-same-length*(L1, L2).*

NOTE — References: **D-equal** A.3.1, **D-same-length** A.3.4

A.3.9 Input and output

L-coding-term*(T, L1 - L2)* — *iff* T is a term concretely represented by the sequence of characters of the difference list of characters $L1 - L2$ as specified by the concrete syntax in clause 6.

D-open-input*(Fn, Env)* — **if** Env is an environment and Fn a name of a stream in Env **then** the stream corresponding to Fn is open for input.

D-open-input*(Fn, env(_, IF, OF, IFL, OFL))* ⇐
streamname*(IF, Fn).*

D-open-input*(Fn, env(_, IF, OF, IFL, OFL))* ⇐
not **streamname***(IF, Fn),*
D-member*(F, IFL),*
streamname*(F, Fn).*

NOTE — References: **streamname** A.4.1.55, **D-member** A.3.4

D-open-output*(Fn, Env)* — **if** Env is an environment and Fn a name of a stream in Env **then** the stream corresponding to Fn is open for output.

D-open-output*(Fn, env(_, IF, OF, IFL, OFL))* ⇐
streamname*(OF, Fn).*

D-open-output*(Fn, env(_, IF, OF, IFL, OFL))* ⇐
not **streamname***(OF, Fn),*
D-member*(F, OFL),*
streamname*(F, Fn).*

NOTE — References: **streamname** A.4.1.55, **D-member** A.3.4

A.4 The Formal Semantics

A.4.1 The kernel

NOTES

1 PVST stands for Partially Visited Search Tree.

2 CVST stands for Completely Visited Search Tree.

A.4.1.1 semantics*(P, G, E, F)*

if P is a well-formed complete database, G is a well-formed goal, and E is an environment **then** F is a PVST up to some node which is any leaf before or on the first infinite branch or CVST if there is no infinite branch.

semantics*(P, G, E, F)* ⇐
D-equal*(N,*
nd(nil, `true & catch`*(G, X, system-error-action)),*
P, nil, E, empsubs, nil, partial)),
buildforest*(for(N, vid, vid), nil, F),*
L-var*(X),*
L-not-occur-in*(X, G).*

NOTES

1 Formally: func(&, func(true, nil).func(catch, G.X.func(special-pred, system-error-action.nil).nil))

2 in all other comments "database" means extended well-formed database and "goal" means extended well-formed goal.

3 References: **D-equal** A.3.1, **L-not-occur-in** A.3.5, **L-var** A.3.1, **buildforest** A.4.1.3

A.4.1.2 predication-choice*(G, A)*

if G is a goal **then** A is the chosen predication in G following the standard strategy (the "first" predication in the goal).

predication-choice*(A, A)* ⇐
not **D-is-a-conjunction***(A).*

predication-choice*(func(&, G._nil), A)* ⇐
predication-choice*(G, A).*

NOTE — References: **D-is-a-conjunction** A.3.1

A.4.1.3 buildforest*(F1, N, F2)*

if $F1$ is a PVST up to node N **then** $F2$ is the extension of the $F1$ up to some node after N which is any leaf before or on the first infinite branch of the complete extension or is a CVST if the complete extension is finite.

buildforest*(F1, N, F1)* ⇐
D-root*(F1, N).*

buildforest*(F1, N, F2)* ⇐
  treatment*(F1, N, F2).*

buildforest*(F1, N, F2)* ⇐
  not **D-root***(F1, N),*
  treatment*(F1, N, F3),*
  clause-choice*(N, F3, M),*
  buildforest*(F3, M, F2).*

NOTE — References: **D-root** A.3.3.2, **treatment** A.4.1.13, **clause-choice** A.4.1.4

A.4.1.4 clause-choice*(N, F, M)*

if F is a PVST up to node N **then** M is the next eligible node.

clause-choice*(N, F, M)* ⇐
  D-lastchild*(N, F, M),*
  not **completely-visited-node***(M, F).*

clause-choice*(N, F, M)* ⇐
  D-lastchild*(N, F, M1),*
  completely-visited-node*(M1, F),*
  next-ancestor*(N, F, M).*

clause-choice*(N, F, M)* ⇐
  not **D-has-a-child***(N, F),*
  next-ancestor*(N, F, M).*

NOTE — References: **D-lastchild** A.3.3.3, **completely-visited-node** A.4.1.5, **next-ancestor** A.4.1.7, **D-has-a-child** A.3.3.3

A.4.1.5 completely-visited-node*(N, F)*

if N is a node of the PVST F **then** N is a completely visited node.

completely-visited-node*(N, F)* ⇐
  D-choice of node *N* **in** *F* **is** *nil,*
  D-visit mark of node *N* **in** *F* **is** *complete.*

NOTE — References: **D-choice of node _ in _ is _** A.3.3.4, **D-visit mark of node _ in _ is _** A.3.3.4

A.4.1.6 completely-visited-tree*(F, N)*

if F is a PVST up to node N **then** F is a CVST of root N.

completely-visited-tree*(F, N)* ⇐
  D-root*(F, N),*
  completely-visited-node*(N, F).*

NOTE — References: **D-root** A.3.3.2, **completely-visited-node** A.4.1.5

A.4.1.7 next-ancestor*(N, F, M)*

if F is a PVST up to node N **then** M is the next ancestor of N which is an eligible node, if it exists, else the root.

next-ancestor*(N, F, M)* ⇐
  available-ancestor*(N, F, M).*

next-ancestor*(N, F, M)* ⇐
  not **has-an-available-ancestor***(N, F),*
  D-root*(F, N1),*
  D-lastchild*(N1, F, M),*
  not **completely-visited-node***(M, F).*

next-ancestor*(N, F, M)* ⇐
  not **has-an-available-ancestor***(N, F),*
  D-root*(F, M),*
  D-lastchild*(M, F, M1),*
  completely-visited-node*(M1, F).*

NOTE — References: **available-ancestor** A.4.1.8, **has-an-available-ancestor** A.4.1.9, **D-root** A.3.3.2, **D-lastchild** A.3.3.3, **completely-visited-node** A.4.1.5

A.4.1.8 available-ancestor*(N, F, M)*

if F is a PVST up to node N **then** M is the next ancestor of N which is an eligible node.

available-ancestor*(N, F, M)* ⇐
  D-parent*(N, F, M),*
  eligible-node*(M, F).*

available-ancestor*(N, F, M)* ⇐
  D-parent*(N, F, K),*
  not **eligible-node***(K, F),*
  available-ancestor*(K, F, M).*

NOTE — References: **D-parent** A.3.3.3, **eligible-node** A.4.1.10, **available-ancestor** A.4.1.8

A.4.1.9 has-an-available-ancestor*(N, F)*

if F is a PVST up to node N **then** N has an eligible node ancestor.

has-an-available-ancestor*(N, F)* ⇐
  available-ancestor*(N, F, _).*

NOTE — References: **available-ancestor** A.4.1.8

A.4.1.10 eligible-node*(N, F)*

if N is a node of the PVST F **then** N is neither completely visited nor is a catch node (a catch node cannot be chosen again even if it is marked not completely visited).

eligible-node*(N, F)* ⇐
  not **completely-visited-node***(N, F),*
  not **is-a-catch-node***(N, F).*

NOTE — References: **completely-visited-node** A.4.1.5, **is-a-catch-node** A.4.1.11

A.4.1.11 is-a-catch-node*(N, F)*

if N is a node of the PVST F **then** N is a node whose chosen predication is the bip `catch`.

is-a-catch-node*(N, F)* ⇐
  chosen predication of node *N* **in** *F* **is** *func*(`catch`, _).

NOTE — References: **chosen predication of node _ in _ is_** A.4.1.12

A.4.1.12 chosen predication of node *N* in *F* is *A*

if N is a node of the PVST F **then** A is the chosen predication in the goal field of the corresponding label node in F.

chosen predication of node *N* **in** *F* **is** *A* ⇐
  D-goal of node *N* **in** *F* **is** *G*,
  predication-choice(*G,A*).

NOTE — References: **D-goal of node _ in _ is _** A.3.3.4, **predication-choice** A.4.1.2

A.4.1.13 treatment(*F1, N, F2*)

if $F1$ is a PVST up to the first not completely visited node N **then** $F2$ is the extension of $F1$ obtained after one step of resolution from N.

treatment(*F1, N, F2*) ⇐
  success-node(*N, F1*),
  erasepack(*F1, N, F2*).

treatment(*F1, N, F2*) ⇐
  not **success-node**(*N, F1*),
  chosen predication of node *N* **in** *F1* **is** *A*,
  D-is-a-bip(*A*),
  not **error**(*F1, A*),
  D-boot-bip(*A*),
  expand(*F1, N, F2*).

treatment(*F1, N, F2*) ⇐
  not **success-node**(*N, F1*),
  chosen predication of node *N* **in** *F1* **is** *A*,
  D-is-a-bip(*A*),
  not **error**(*F1, A*),
  not **D-boot-bip**(*A*),
  treat-bip(*F1, N, A, F2*).

treatment(*F1, N, F2*) ⇐
  not **success-node**(*N, F1*),
  chosen predication of node *N* **in** *F1* **is** *A*,
  D-is-a-bip(*A*),
  in-error(*F1, A, T*),
  treat-bip(*F1, N, func*(`throw`, *T.nil*), *F2*).

treatment(*F1, N, F2*) ⇐
  not **success-node**(*N, F1*),
  chosen predication of node *N* **in** *F1* **is** *A*,
  D-is-a-special-pred(*A*),
  treat-special-pred(*F1, N, A, F2*).

treatment(*F1, N, F2*) ⇐
  not **success-node**(*N, F1*),
  chosen predication of node *N* **in** *F1* **is** *A*,
  not **D-is-a-bip**(*A*),
  not **D-is-a-special-pred**(*A*),
  expand(*F1, N, F2*).

NOTE — References: **success-node** A.4.1.16, **erasepack** A.4.1.23, **chosen predication of node _ in _ is_** A.4.1.12, **D-is-a-bip** A.3.8, **error** A.4.1.14, **D-boot-bip** A.3.8, **D-is-a-special-pred** A.3.1, **expand** A.4.1.18, **treat-bip** A.4.1.31, **in-error** A.4.1.15, **treat-special-pred** A.4.1.17,

A.4.1.14 error(*F, B*)

if F is a forest and B is a predication **then** it is a predication of a built-in predicate whose execution raises an error in F.

error(*F, B*) ⇐
  in-error(*F, B, _*).

NOTE — References: **in-error** A.4.1.15

A.4.1.15 in-error(*F, B, T*)

if F is a forest and B is a predication **then** it is a predication of a built-in predicate whose execution raises an error of type T.

in-error(_, *special-pred(operation-value, V1.func(Op, nil).V2.V.nil), T*) ⇐
  L-error-in-expression(*func(Op, V1.V2.nil), T*).

The appropriate clauses of **in-error** are given with the definitions of each built-in predicate.

A.4.1.16 success-node(*N, F*)

if F is a PVST up to node N **then** the goal carried by N is the goal `true`.

success-node(*N, F*) ⇐
  D-goal of node *N* **in** *F* **is** *func*(`true`,*nil*).

NOTE — References: **D-goal of node _ in _ is _** A.3.3.4

A.4.1.17 treat-special-pred(*F1, N, A, F2*)

if $F1$ is a PVST up to node N and the chosen predication A in the goal of N is a special predicate **then** $F2$ is the new PVST obtained after its execution.

treat-special-pred(*F1, N, special-pred(inactivate, J.nil), F2*) ⇐
  treat-inactivate(*F1, N, J, F2*).

treat-special-pred(*F1, N, special-pred(system-error-action, nil), F1*).

treat-special-pred(*F1, N, special-pred(halt-system-action, nil), F1*).

treat-special-pred(*F1, N, special-pred(halt-system-action, I.nil), F1*).

treat-special-pred(*F1, N, special-pred(value, E.V.nil), F2*) ⇐
  expand(*F1, N, F2*).

treat-special-pred(*F1, N, special-pred(compare, E1.Op.E2.nil), F2*) ⇐
  expand(*F1, N, F2*).

treat-special-pred(*F1, N, special-pred(operation-value, V1.func(Op, nil).V2.V.nil), F1*) ⇐
  not **error**(*F1, special-pred(operation-value, V1.func(Op, nil).V2.V.nil)*),
  L-value(*func(Op, V1.V2.nil), V*).

treat-special-pred(*F1, N, special-pred(operation-value, V1.func(Op, nil).V2.V.nil), F2*) ⇐
  in-error(*F1, special-pred(operation-value, V1.func(Op, nil).V2.V.nil), T*),
  treat-bip(*F1, N, func*(`throw`, *T.nil*), *F2*).

treat-special-pred(*F1, N, special-pred(sorted, L1.L2.nil), F1*) ⇐
  L-sorted(*L1, L2*).

treat-special-pred*(F1, N, special-pred(simple-comparison, V1.Op.V2.nil), F1)* ⇐
L-arithmetic-comparison*(V1, Op, V2).*

NOTE — References: **treat-inactivate** A.4.1.57, **expand** A.4.1.18, **error** A.4.1.14, **in-error** A.4.1.15, **L-value** A.3.6, **treat-bip** A.4.1.31, **L-sorted** A.3.4, **L-arithmetic-comparison** A.3.6

A.4.1.18 expand*(F1, N, F2)*

if $F1$ is a PVST up to node N and the chosen predication in the goal of N is a user defined predicate or a boot-strapped built-in predicate **then** $F2$ is the new PVST obtained after one step of resolution (So the node N in $F2$ either has a new youngest child or has no new child and is marked completely visited).

expand*(F1, N, F2)* ⇐
D-choice of node *N* **in** *F1* **is** *Q,*
chosen predication of node *N* **in** *F1* **is** *A,*
D-label of node *N* **in** *F1* **is** *Nl,*
not **possible-child***(Q, F1, Nl, A),*
undefined-pred-treatment*(F1, N, A, F2).*

expand*(F1, N, F2)* ⇐
chosen predication of node *N* **in** *F1* **is** *A,*
D-equal*(A, special-pred(value, func(Op, E1.E2.nil).V.nil)),*
D-label of node *N* **in** *F1* **is** *Nl,*
add-value-child*(F1, Nl, A, F2).*

expand*(F1, N, F2)* ⇐
chosen predication of node *N* **in** *F1* **is** *A,*
D-equal*(A, special-pred(compare, E1.Op.E2.nil)),*
D-label of node *N* **in** *F1* **is** *Nl,*
add-compare-child*(F1, Nl, A, F2).*

expand*(F1, N, F2)* ⇐
D-choice of node *N* **in** *F1* **is** *Q,*
chosen predication of node *N* **in** *F1* **is** *A,*
D-label of node *N* **in** *F1* **is** *Nl,*
buildchild*(Q, F1, Nl, A, Nl1, Q1),*
addchild*(F1, Nl, Nl1, Q1, F2).*

NOTE — References: **D-choice of node _ in _ is _** A.3.3.4, **chosen predication of node _ in _ is _** A.4.1.12, **D-label of node _ in _ is _** A.3.3.4, **possible-child** A.4.1.22, **undefined-pred-treatment** A.4.1.19, **add-value-child** A.4.1.20, **add-compare-child** A.4.1.21, **buildchild** A.4.1.24, **addchild** A.4.1.25

A.4.1.19 undefined-pred-treatment*(F1, N, A, F2)*

if $F1$ is a PVST up to node N, and A is an undefined predication **then** $F2$ is the extension of $F1$ according to the value of the flag `undefined_predicate` (7.11.2.4).

undefined-pred-treatment*(F1, N, A, F2)* ⇐
D-environment of node *N* **in** *F1* **is** *Env,*
D-equal*(Env, env(PF, _ _ _ _)),*
corresponding-flag-and-value*(func(*`undefined_predicate`*, nil), Value, PF, _ _),*
D-equal*(Value, func(fail, nil)),*
erasepack*(F1, N, F2).*

undefined-pred-treatment*(F1, N, A, F2)* ⇐
D-environment of node *N* **in** *F1* **is** *Env,*
D-equal*(Env, env(PF, _ _ _ _)),*
corresponding-flag-and-value*(func(*`undefined_predicate`*, nil), Value, PF, _ _),*
D-equal*(Value, func(error, nil)),*
treat-bip*(F1, N, func(*`throw`*, undefined_predicate_error.nil), F2).*

undefined-pred-treatment*(F1, N, A, F2)* ⇐
D-environment of node *N* **in** *F1* **is** *Env,*
D-equal*(Env, env(PF, _ _ _ _)),*
corresponding-flag-and-value*(func(*`undefined_predicate`*, nil), Value, PF, _ _),*
D-equal*(Value, func(warning, nil)),*
treat-bip*(F1, N, func(*`write`*, output_warning_stream.undefined_predicate_message.nil).nil), F2).*

NOTE — References: **D-environment of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **corresponding-flag-and-value** A.4.1.68, **erasepack** A.4.1.23, **treat-bip** A.4.1.31

expand(F1, N, F2) ⇐
chosen predication of node N **in** F1 **is** A,
D-equal(A, special-pred(value, func(Op, E1.E2.nil).V.nil)),
D-label of node N **in** F1 **is** Nl,
add-value-child(F1, Nl, A, F2).

A.4.1.20 add-value-child*(F1, Nl, A, F2)*

if A is the special predication *value* chosen in the goal of the node label Nl in the PVST F **then** $F2$ is the new PVST with one new child whose node label is identical to Nl except that the new goal contains explicit evaluation of the expression.

add-value-child*(F1, Nl, A, F2)* ⇐
D-equal*(Nl, nd(I, G, P, Q, E, _ L, _)),*
D-number-of-child*(I, F1, J),*
D-equal*(A, special-pred(value, Num.V.nil)),*
erase*(G, G1),*
D-equal*(G2, func(&, func(number, Num.nil).func(=, Num.V.nil).nil).G1.nil),*
predication-choice*(G2, A1),*
D-packet*(P, A1, Q1),*
D-equal*(Nl1, nd(J.I, G2, P, Q1, E, empsubs, L, partial)),*
addchild*(F1, Nl, Nl1, nil, F2).*

add-value-child*(F1, Nl, A, F2)* ⇐
D-equal*(Nl, nd(I, G, P, Q, E, _ L, _)),*
D-number-of-child*(I, F1, J),*
D-equal*(A, special-pred(value, func(Op, E1.E2.nil).V.nil)),*
erase*(G, G1),*
D-equal*(G2, func(&, (special-pred(value, E1.V1.nil), special-pred(value, E2.V2.nil), special-pred(operation-value, V1.Op.V2.V.nil)).G1.nil),*
D-equal*(Nl1, nd(J.I, G2, P, Q, E, empsubs, L, partial)),*
addchild*(F1, Nl, Nl1, nil, F2).*

add-value-child*(F1, Nl, A, F2)* ⇐
D-equal*(Nl, nd(I, G, P, Q, E, _ L, _)),*
D-number-of-child*(I, F1, J),*
D-equal*(A, special-pred(value, func(Op, E1.E2.nil).V.nil)),*
erase*(G, G1),*
D-equal*(G2, func(&, (special-pred(value, E2.V2.nil), special-pred(value, E1.V1.nil), special-pred(operation-value, V1.Op.V2.V.nil)).G1.nil),*
D-equal*(Nl1, nd(J.I, G2, P, Q, E, empsubs, L, partial)),*
addchild*(F1, Nl, Nl1, nil, F2).*

NOTE — References: **D-equal** A.3.1, **D-number-of-child** A.3.3.3, **erase** A.4.1.35, **predication-choice** A.4.1.2, **D-packet** A.3.8, **addchild** A.4.1.25

A.4.1.21 add-compare-child*(F1, Nl, A, F2)*

if A is the special predication *compare* chosen in the goal of the node label Nl in the PVST F **then** $F2$ is the new PVST with one new child whose node label is identical to Nl except that the new goal contains explicit comparison of the expression.

add-compare-child*(F1, Nl, A, F2)* ⇐
D-equal*(Nl, nd(I, G, P, Q, E, _ L, _)),*
D-number-of-child*(I, F1, J),*
D-equal*(A, special-pred(compare, E1.Op.E2.nil)),*
erase*(G, G1),*
D-equal*(G2, func(&, (special-pred(value, E1.V1.nil), special-pred(value, E2.V2.nil), special-pred(simple-comparison, V1.Op.V2.nil),*
D-equal*(Nl1, nd(J.I, G2, P, Q, E, empsubs, L, partial)),*
addchild*(F1, Nl, Nl1, nil, F2).*

add-compare-child*(F1, Nl, A, F2)* ⇐
D-equal*(Nl, nd(I, G, P, Q, E, _ L, _)),*
D-number-of-child*(I, F1, J),*
D-equal*(A, special-pred(compare, E1.Op.E2.nil)),*
erase*(G, G1),*
D-equal*(G2, func(&, (special-pred(value, E2.V2.nil), special-pred(value, E1.V1.nil), special-pred(simple-comparison, V1.Op.V2.nil),*
D-equal*(Nl1, nd(J.I, G2, P, Q, E, empsubs, L, partial)),*
addchild*(F1, Nl, Nl1, nil, F2).*

NOTE — References: **D-equal** A.3.1, **D-number-of-child** A.3.3.3, **erase** A.4.1.35, **predication-choice** A.4.1.2, **D-packet** A.3.8, **addchild** A.4.1.25

A.4.1.22 possible-child*(Q, F, N, A)*

if A is the chosen predication in the goal of the node label N in the PVST F, and Q is the clauses corresponding to the remaining choices **then** it is possible to build a child to N with one of these clauses.

possible-child*(Q, F, N, A)* ⇐
buildchild*(Q, F, N, A, _ _).*

NOTE — References: **buildchild** A.4.1.24

A.4.1.23 erasepack*(F1, N, F2)*

if $F1$ is the PVST up to node N **then** $F2$ is the same PVST except that the node N in $F2$ is completely visited.

erasepack*(F1, N, F2)* ⇐
D-label of node *N* **in** *F1* **is** *Nl1,*
complete-visit*(Nl1, Nl2),*
D-modify-node*(F1, Nl1, Nl2, F2).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4 **complete-visit** A.4.1.37 **D-modify-node** A.3.3.6

A.4.1.24 buildchild*(Q, F, Nl, A, Nl1, Q1)*

if A is the chosen predication in the goal of the node label Nl in the PVST F, and Q is the non empty remaining choices corresponding to A **then** $Nl1$ is the node label of the new child of Nl not completely visited built with a clause of Q, and $Q1$ is the remaining choices for building any further child of Nl.

buildchild*(func(:–, H.B.nil).R, F, Nl, A, Nl1, R)* ⇐
D-equal*(Nl, nd(I, G, P, _ E, _ L, _)),*
D-number-of-child*(I, F, J),*
L-rename*(F, func(:–, H.B.nil), func(:–, H1.B1.nil)),*
L-unify*(H1, A, S1),*
flag-cut*(B1, J.I, B2),*
replace*(G, A, B2, G2),*
L-instance*(G2, S1, G1),*
predication-choice*(G1, A1),*
D-packet*(P, A1, Q1),*
D-equal*(Nl1, nd(J.I, G1, P, Q1, E, S1, L, partial)).*

buildchild*(func(:–, H.B.nil).R, F, Nl, A, Nl1, R1)* ⇐
L-rename*(F, func(:–, H.B.nil), func(:–, H1.B1.nil)),*
L-not-unifiable*(H1, A),*
buildchild*(R, F, Nl, A, Nl1, R1).*

NOTE — References: **D-equal** A.3.1, **D-number-of-child** A.3.3.3, **L-rename** A.3.5, **L-unify** A.3.5, **flag-cut** A.4.1.28, **replace** A.4.1.27, **L-instance** A.3.5, **predication-choice** A.4.1.2, **D-packet** A.3.8, **L-not-unifiable** A.3.5

A.4.1.25 addchild*(F, Nl, Nl1, Q, F1)*

if F is a PVST up to the label node Nl, and Q are the remaining choices available to build the next children of Nl, and $Nl1$ is the node label to be added **then** $F1$ is the new PVST with $Nl1$ as youngest child of Nl, and Nl is updated with the remaining choices Q, and Nl is marked completely visited if Q is empty.

addchild*(F1, Nl1, Nl2, Q, F2)* ⇐
D- *Nl3* **is** *Nl1* **where choices are** *Q,*
case-nil-choice*(Q, Nl3, Nl4),*
D-modify-node*(F1, Nl1, Nl4, F3),*
D-create-child*(F3, Nl4, Nl2, F2).*

NOTE — References: **D- _ is _ where choices are _** A.3.3.5, **case-nil-choice** A.4.1.26, **D-modify-node** A.3.3.6, **D-create-child** A.3.3.6

A.4.1.26 case-nil-choice*(Q, Nl1, Nl2)*

if $Nl1$ is a node label and Q an abstract list of clauses **then** $Nl2$ is $Nl1$ unless Q is empty in which case $Nl2$ is marked completely visited.

case-nil-choice*(nil, Nl1, Nl2)* ⇐
D- *Nl2* **is** *Nl1* **where visit mark is** *complete.*

case-nil-choice*(Q, Nl, Nl)* ⇐
not **D-equal***(Q, nil).*

NOTE — References: **D- _ is _ where visit mark is _** A.3.3.5 **D-equal** A.3.1

A.4.1.27 replace*(G1, A, G2, G3)*

if $G1$ and $G2$ are goals, and A the first predication of $G1$ **then** $G3$ is the goal obtained from $G1$ by replacement of A by $G2$.

replace*(A, A, G1, G1)* ⇐
not **D-is-a-conjunction***(A).*

replace*(func(&, G1.G.nil), A, G2, func(&, G3.G.nil))* ⇐
replace*(G1, A, G2, G3).*

NOTE — References: **D-is-a-conjunction** A.3.1

A.4.1.28 flag-cut*(A, J, B)*

if A is a goal and J a node **then** the goal B is A in which all the cuts not already flagged with scope in B are flagged by J.

flag-cut*(func(!, nil), J, func(!, J.nil)).*

flag-cut*(A, J, A)* ⇐
  D-is-a-predication*(A),*
  not **is-a-not-flagged-cut***(A).*

flag-cut*(func(&, L), J, func(&, L1))* ⇐
  flag-list*(L, J, L1).*

flag-cut*(func(;, L), J, func(;, L1))* ⇐
  flag-list*(L, J, L1).*

flag-cut*(func(->, C.L), J, func(->, C.L1))* ⇐
  flag-list*(L, J, L1).*

NOTE — References: **D-is-a-predication** A.3.1, **is-a-not-flagged-cut** A.4.1.29, **flag-list** A.4.1.30

A.4.1.29 is-a-not-flagged-cut*(A)*

iff A is the control construct "cut" and is not flagged.

is-a-not-flagged-cut*(func(!, nil)).*

A.4.1.30 flag-list*(G, J, G1)*

if G is an abstract list of goals and J is a node **then** $G1$ is an abstract list whose elements are those of G in which all the control constructs "cut" have been flagged.

flag-list*(nil, J, nil).*

flag-list*(X.L, J, X1.L1)* ⇐
  flag-cut*(X, J, X1),*
  flag-list*(L, J, L1).*

NOTE — References: **flag-cut** A.4.1.28

A.4.1.31 treat-bip*(F, N, B, F1)*

if F is a PVST up to node N and B is the chosen goal in N and B is a bip but neither bootstrapped nor in error **then** $F1$ is the new PVST obtained after execution of B.

treat-bip*(F, N, B, F1)* ⇐
  D-is-a-subst-bip*(B),*
  bip-expand*(F, N, B, F1).*

NOTE — The other clauses for **treat-bip** defining each built-in predicate (other than the substitution bip's or the bootstrapped one's) are given in the subclause for that built-in predicate.

NOTE — References: **D-is-a-subst-bip** A.3.8, **bip-expand** A.4.1.32

A.4.1.32 bip-expand*(F, N, B, F1)*

if F is a PVST up to node N, and B is a substitution built-in predicate not in error **then** $F1$ is the PVST obtained after execution of B.

bip-expand*(F, N, B, F1)* ⇐
  D-label of node *N* **in** *F* **is** *Nl1,*
  D-equal*(Nl1, nd(N, G, P, _, E, _, L, _)),*
  execute-bip*(F, B, S1),*
  final-resolution-step*(G, S1, P, G1, Q1),*
  D-equal*(Nl2, nd(zero.N, G1, P, Q1, E, S1, L, partial)),*
  addchild*(F, Nl1, Nl2, nil, F1).*

bip-expand*(F, N, B, F1)* ⇐
  not **execsuccess***(F, B),*
  erasepack*(F, N, F1).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **execute-bip** A.4.1.36, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **execsuccess** A.4.1.34, **erasepack** A.4.1.23

A.4.1.33 final-resolution-step*(G, S, P, G1, Q)*

if G is a goal, S a substitution and P a database **then** $G1$ and Q are respectively the goal and the choices obtained by application of the resolution step **e)** (see A.2.1).

final-resolution-step*(G, S, P, G1, Q)* ⇐
  erase*(G, G2),*
  L-instance*(G2, S, G1),*
  predication-choice*(G1, A),*
  D-packet*(P, A, Q).*

NOTE — References: **erase** A.4.1.35, **L-instance** A.3.5, **predication-choice** A.4.1.2, **D-packet** A.3.8

A.4.1.34 execsuccess*(F, B)*

if B is a substitution built-in predicate not in error, and F is a PVST **then** B succeeds in F.

execsuccess*(F, B)* ⇐
  execute-bip*(F, B, _).*

NOTE — References: **execute-bip** A.4.1.36

A.4.1.35 erase*(G, G1)*

if G is a goal **then** $G1$ is the goal obtained by deleting the first predication of G.

erase*(G, func(*`true`*, nil))* ⇐
  not **D-is-a-conjunction***(G).*

erase*(func(&, A.G.nil), G)* ⇐
  not **D-is-a-conjunction***(A).*

erase*(func(&, func(&, G1.G2.nil).G3.nil), func(&, T.G3.nil))* ⇐
  erase*(func(&, G1.G2.nil), T).*

NOTE — References: **D-is-a-conjunction** A.3.1

A.4.1.36 execute-bip*(F, B, S)*

if B is a substitution bip in the context of the PVST F and B is not in error **then** execution of B succeeds with substitution S.

NOTE — Clauses for **execute-bip** are given in the subclause for relevant built-in predicates.

A.4.1.37 complete-visit*(Nl1, Nl2)*

if $Nl1$ is a node label, **then** $Nl2$ is the same node label except that its remaining choices are empty and the visit-mark indicates that $Nl2$ is completely visited.

complete-visit*(Nl1, Nl2)* ⇐
  D- *Nl3* **is** *Nl1* **where choices are** *nil,*
  D- *Nl2* **is** *Nl3* **where visit mark is** *complete.*

NOTE — References: **D- _ is _ where choices are _** A.3.3.5, **D- _ is _ where visit mark is _** A.3.3.5

A.4.1.38 cut-all-choice-point*(F1, N, M, F2)*

if $F1$ is a PVST up to node N, and M an ancestor node of N **then** $F2$ is $F1$ where all the nodes along the path from N to M inclusive are completely visited.

cut-all-choice-point*(F1, N, N, F2)* ⇐
  cut-choice-point*(F1, N, F2).*

cut-all-choice-point*(F1, A.N, M, F2)* ⇐
  cut-choice-point*(F1, A.N, F3),*
  cut-all-choice-point*(F3, N, M, F2).*

NOTE — References: **cut-choice-point** A.4.1.39

A.4.1.39 cut-choice-point*(F1, N, F2)*

if $F1$ is a PVST up to node N **then** $F2$ is $F1$ where N is marked completely visited.

cut-choice-point*(F1, N, F2)* ⇐
  D-label of node *N* **in** *F1* **is** *Nl,*
  complete-visit*(Nl, Nl1),*
  D-modify-node*(F1, Nl, Nl1, F2).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **complete-visit** A.4.1.37, **D-modify-node** A.3.3.6

A.4.1.40 term-ordered*(X, Y)*

if X and Y are terms **then** X term-precedes Y in the total order on the terms (see 7.2).

term-ordered*(X, Y)* ⇐
  L-var-order*(X, Y).*

term-ordered*(X, Y)* ⇐
  L-var*(X),*
  not **L-var***(Y).*

term-ordered*(func(X, nil), func(Y, nil))* ⇐
  L-float-less*(X, Y).*

term-ordered*(X, Y)* ⇐
  D-is-a-float*(X),*
  not **L-var***(Y),*
  not **D-is-a-float***(Y).*

term-ordered*(func(X, nil), func(Y, nil))* ⇐
  L-integer-less*(X, Y).*

term-ordered*(X, Y)* ⇐
  D-is-an-integer*(X),*
  not **L-var***(Y),*
  not **D-is-a-float***(Y),*
  not **D-is-an-integer***(Y).*

term-ordered*(func(X, L), func(Y, L1))* ⇐
  D-length-list*(L, N),*
  D-length-list*(L1, N1),*
  L-integer-less*(N, N1).*

term-ordered*(func(X, L), func(Y, L1))* ⇐
  D-same-length*(L, L1),*
  L-atom-order*(X, Y).*

term-ordered*(func(X, L), func(X, L1))* ⇐
  D-same-length*(L, L1),*
  before*(L, L1).*

NOTE — References: **L-var-order** A.3.4, **L-var** A.3.1, **L-float-less** A.3.6, **D-is-a-float** A.3.1, **L-integer-less** A.3.6, **D-is-an-integer** A.3.1, **D-length-list** A.3.4, **D-same-length** A.3.4, **L-atom-order** A.3.4, **before** A.4.1.41

A.4.1.41 before*(L, L1)*

if L and $L1$ are arg-lists of same length **then** they are different and the first element of L which is different from the corresponding element in $L1$ is less than this element.

before*(X.L, X1.L1)* ⇐
  term-ordered*(X, X1).*

before*(X.L, X.L1)* ⇐
  before*(L, L1).*

NOTE — References: **term-ordered** A.4.1.40

A.4.1.42 complete-semantics*(P, G, E, F)*

if P is a database, G is a goal, and E is an environment, and there exists a CVST for P, G, and E **then** F is this CVST.

complete-semantics*(P, G, E, F)* ⇐
  D-equal*(N,*
  nd(nil,func(catch, G.X.func(special-pred, system-error-action.nil).nil),
  P, nil, E, empsubs, nil, partial)),
  complete-forest*(for(N, vid, vid), nil, F),*
  L-var*(X),*
  L-not-occur-in*(X, G).*

NOTE — References: **D-equal** A.3.1, **L-var** A.3.1, **L-not-occur-in** A.3.5, **complete-forest** A.4.1.43

A.4.1.43 complete-forest*(F1, N, F2)*

if $F1$ is a PVST up to node N, and there exists a CVST which is an extension of $F1$ **then** $F2$ is this CVST.

complete-forest*(F1, N, F1)* ⇐
  completely-visited-tree*(F1, N).*

complete-forest*(F1, N, F2)* ⇐
  treatment*(F1, N, F3),*
  clause-choice*(N, F3, M),*
  complete-forest*(F3, M, F2).*

NOTE — References: **completely-visited-tree** A.4.1.6, **D-root** A.3.3.2, **treatment** A.4.1.13, **clause-choice** A.4.1.4

A.4.1.44 list-of-ans-subs*(F, L)*

if F is a forest **then** L is the abstract list of answer substitutions of F.

list-of-ans-subs*(vid, nil).*

list-of-ans-subs*(for(N, vid, F2), L)* ⇐
not **D-equal***(N, nd(_ func(*`true`*,nil), _ _ _ _ _ _)),*
list-of-ans-subs*(F2, L).*

list-of-ans-subs*(for(N, vid, F2), S.L)* ⇐
D-equal*(N, nd(_ func(*`true`*,nil), _ _ _ S, _ _)),*
list-of-ans-subs*(F2, L).*

list-of-ans-subs*(for(N, F1, F2), L)* ⇐
not **D-equal***(F1, vid),*
list-of-ans-subs*(F1, L1),*
D-equal*(N, nd(_ _ _ _ _ S, _ _)),*
D-compose-list*(L1, S, L2),*
list-of-ans-subs*(F2, L3),*
D-conc*(L2, L3, L).*

NOTE — References: **D-equal** A.3.1, **D-compose-list** A.3.5, **D-conc** A.3.4

A.4.1.45 list-of-instance*(L, T, B)*

if L is an abstract list of substitutions, and T is a term, **then** B is the abstract list of instances of T by all the substitutions of L.

list-of-instance*(nil, T, func(*`[]`*, nil)).*

list-of-instance*(S.L, T, func(*`.`*, T2.B.nil))* ⇐
L-instance*(T, S, T1),*
list-of-instance*(L, T, B).*

NOTE — References: **L-instance** A.3.5, **L-rename** A.3.5

A.4.1.46 untrapped-error*(F)*

if F is a forest **then** F gives rise to a node of untrapped error.

untrapped-error*(F)* ⇐
D-root*(F, N),*
D-number-of-child*(N, F, I),*
D-equal*(I, s(s(zero))).*

NOTE — References: **D-root** A.3.3.2, **D-number-of-child** A.3.3.3, **D-equal** A.3.1

A.4.1.47 first-match-clause*(F, P, B, Cl, S, Q)*

if P is a packet of clauses, B is a clause, and F is a forest **then** Cl is the first clause in P with its variables renamed such that they do not occur in F, Cl unifies with B by substitution S, and Q is the remaining packet of clauses in P that are after Cl.

first-match-clause*(F, Cl.P, B, Cl, S, P)* ⇐
L-rename*(F, Cl, Cl1),*
L-unify*(Cl1, B, S).*

first-match-clause*(F, Cl1.P, B, Cl, S, Q)* ⇐
L-rename*(F, Cl1, Cl2),*
L-not-unifiable*(Cl2, B),*
first-match-clause*(F, P, B, Cl, S, Q).*

NOTE — References: **L-rename** A.3.5, **L-unify** A.3.5, **L-not-unifiable** A.3.5

A.4.1.48 exist-match-clause*(F, P, B)*

if P is a packet of clauses, B a clause, and F is a forest **then** there exists a clause in P which unifies with B when its variables are renamed such that they do not occur in F.

exist-match-clause*(F, P, B)* ⇐
first-match-clause*(F, P, B, _ _ _).*

NOTE — References: **first-match-clause** A.4.1.47

A.4.1.49 corresponding-pred*(PI, P, PI1, S)*

if PI is a predicate indicator pattern and P is an abstract list of clauses **then** $PI1$ is the predicate indicator of one of the clause head in P which unifies with PI and S is the resulting substitution.

corresponding-pred*(func(*`/`*, At.Ar.nil),*
func(`:-`*, H._nil).P, func(*`/`*, At1.Ar1.nil), S)* ⇐
D-name*(H, At1),*
D-arity*(H, Ar1),*
L-unify*(func(*`/`*, At.Ar.nil), func(*`/`*, At1.Ar1.nil), S).*

corresponding-pred*(func(*`/`*, At.Ar.nil),*
func(`:-`*, __nil).P, PI1, S)* ⇐
corresponding-pred*(func(*`/`*, At.Ar.nil), P, PI1, S).*

NOTE — References: **D-name** A.3.1, **D-arity** A.3.1, **L-unify** A.3.5

A.4.1.50 exist-corresponding-pred*(PI, P)*

if PI is a predicate indicator pattern and P is an arg-list of clauses **then** P contains clauses for a predicate corresponding to the indicator PI.

exist-corresponding-pred*(PI, P)* ⇐
corresponding-pred*(PI, P, _ _).*

NOTE — References: **corresponding-pred** A.4.1.49

A.4.1.51 corresponding-pred-definition*(PI, DB, P, S)*

if PI is a predicate indicator pattern and DB is a non empty database **then** P is the definition of the predicate whose indicator unifies with PI and S is the resulting substitution.

corresponding-pred-definition*(PI, def(PI1, SD,P).DB, def(PI1, SD,P), S)* ⇐
L-unify*(PI, PI1, S).*

corresponding-pred-definition*(PI, def(PI1, _ _).DB, PD, S)* ⇐
L-not-unifiable*(PI, PI1),*
corresponding-pred-definition*(PI, DB, PD, S).*

NOTE — References: **L-unify** A.3.5, **L-not-unifiable** A.3.5

A.4.1.52 exist-corresponding-pred-definition*(PI, DB)*

if PI is a predicate indicator pattern and DB is a database **then** P is there exists a definition of the predicate whose indicator unifies with PI

exist-corresponding-pred-definition*(PI, DB)* ⇐
corresponding-pred-definition*(PI, DB, _, _)*.

NOTE — References: **corresponding-pred-definition** A.4.1.51

A.4.1.53 get-stream-in-env*(Old, F, C, New)*

if Old is the old environment which contains a stream whose first argument is F **then** C is the next character to be read in this stream and New is the new environment obtained by advancing the pointer in it.

get-stream-in-env*(Old, F, C, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
D-equal*(IF, stream(F, L1-C.L2))*,
D-delete*(LIF, IF, LIF1)*,
D-equal*(Newif, stream(F, L1-L2))*,
D-equal*(New, env(PF, Newif, OF, Newif.LIF1, LOF))*.

get-stream-in-env*(Old, F, C, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
not **streamname***(IF, F)*,
D-delete*(LIF, stream(F, L1 - C.L2), LIF1)*,
D-equal*(Newf, stream(F, L1 - L2))*,
D-equal*(New, env(PF, IF, OF, Newf.LIF1, LOF))*.

get-stream-in-env*(Old, F, func(eof, nil), Old)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
D-equal*(IF, stream(F, L - nil))*,
L-io-option*(F, eof-action, eof-code)*.

get-stream-in-env*(Old, F, C, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
D-equal*(IF, stream(F, C.L - nil))*,
L-io-option*(F, eof-action, reset)*,
D-delete*(LIF, IF, LIF1)*,
D-equal*(Newif, stream(F, C.L - L))*,
D-equal*(New, env(PF, Newif, OF, Newif.LIF1, LOF))*.

get-stream-in-env*(Old, F, func(eof, nil), Old)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
not **streamname***(IF, F)*,
D-member*(stream(F, L - nil), LIF)*,
L-io-option*(F, eof-action, eof-code)*.

get-stream-in-env*(Old, F, C, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
not **streamname***(IF, F)*,
D-delete*(LIF, stream(F, C.L - nil), LIF1)*,
L-io-option*(F, eof-action, reset)*,
D-equal*(Newf, stream(F, C.L - L))*,
D-equal*(New, env(PF, IF, OF, Newf.LIF1, LOF))*.

NOTE — References: **D-equal** A.3.1, **D-delete** A.3.4, **D-member** A.3.4, **L-io-option** A.3.7, **streamname** A.4.1.55

A.4.1.54 get-term-stream-in-env*(Old, F, T, New)*

if Old is the old environment which contains stream whose first argument is F **then** T is the first term beyond the current pointer to be read in this stream and New is the new environment obtained by advancing the pointer in it.

get-term-stream-in-env*(Old, F, T, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
D-equal*(IF, stream(F, L1 - L2))*,
L-coding-term*(T, L2 - L3)*,
D-delete*(LIF, IF, LIF1)*,
D-equal*(Newif, stream(F, L1 - L3))*,
D-equal*(New, env(PF, Newif, OF, Newif.LIF1, LOF))*.

get-term-stream-in-env*(Old, F, T, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
not **streamname***(IF, F)*,
D-delete*(LIF, stream(F, L1 - L2), LIF1)*,
L-coding-term*(T, L2 - L3)*,
D-equal*(Newf, stream(F, L1 - L3))*,
D-equal*(New, env(PF, IF, OF, Newf.LIF1, LOF))*.

get-term-stream-in-env*(Old, F, func(*`end_of_file`*, nil), Old)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
D-equal*(IF, stream(F, L - nil))*,
L-io-option*(F, eof-action, eof-code)*.

get-term-stream-in-env*(Old, F, T, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
D-equal*(IF, stream(F, L - nil))*,
L-io-option*(F, eof-action, reset)*,
L-coding-term*(T, L - L1)*,
D-delete*(LIF, IF, LIF1)*,
D-equal*(Newif, stream(F, L - L1))*,
D-equal*(New, env(PF, Newif, OF, Newif.LIF1, LOF))*.

get-term-stream-in-env*(Old, F, func(*`end_of_file`*, nil), Old)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
not **streamname***(IF, F)*,
D-member*(stream(F, L - nil), LIF)*,
L-io-option*(F, eof-action, eof-code)*.

get-term-stream-in-env*(Old, F, T, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,
not **streamname***(IF, F)*,
D-delete*(LIF, stream(F, L - nil), LIF1)*,
L-io-option*(F, eof-action, reset)*,
L-coding-term*(T, L - L1)*,
D-equal*(Newf, stream(F, L - L1))*,
D-equal*(New, env(PF, IF, OF, Newf.LIF1, LOF))*.

NOTE — References: **D-equal** A.3.1, **D-delete** A.3.4, **D-member** A.3.4, **L-io-option** A.3.7, **streamname** A.4.1.55, **L-coding-term** A.3.9

A.4.1.55 streamname*(F, Fn)*

if F is a stream **then** Fn is the name of the stream F.

streamname*(stream(Fn, _), Fn)*.

A.4.1.56 put-stream-in-env*(Old, F, LC, New)*

if Old is the old environment which contains stream whose first argument is F and the pointer of this stream is at the end-of-file, and LC is an abstract list of characters **then** New is the new environment made by adding LC to the end of this stream.

put-stream-in-env*(Old, F, LC, New)* ⇐
D-equal*(Old, env(PF, IF, OF, LIF, LOF))*,

D-equal(*OF, stream(F, L1 - nil)*),
D-delete(*LOF, OF, LOF1*),
D-conc(*L1, LC, L2*),
D-equal(*Newof, stream(F, L2 - nil)*),
D-equal(*New, env(PF, IF, Newof, LIF, Newof.LOF1)*).

put-stream-in-env(*Old, F, LC, New*) ⇐
D-equal(*Old, env(PF, IF, OF, LIF, LOF)*),
not **streamname**(*OF, F*),
D-delete(*LOF, stream(F, L1 - nil), LOF1*),
D-conc(*L1, LC, L2*),
D-equal(*Newf, stream(F, L2 - nil)*),
D-equal(*New, env(PF, IF, OF, LIF, Newf.LOF1)*).

NOTE — References: **D-equal** A.3.1, **D-delete** A.3.4, **D-conc** A.3.4, **streamname** A.4.1.55

A.4.1.57 treat-inactivate(*F, N, J, F1*)

if F is a PVST up to node N, and the goal carried by N is *special-pred(inactivate, J.nil)*, **then** $F1$ has one more node than F, the node of $F1$ corresponding to N is completely visited, and J is not on the list of active (catch) nodes of this child.

treat-inactivate(*F, N, J, F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, _, E, _, L, _)*),
D-delete(*L, J, L1*),
final-resolution-step(*G, empsubs, P, G1, Q1*),
D-equal(*Nl1, nd(zero.N, G1, P, Q1, E, empsubs, L1, partial)*),
addchild(*F, Nl, Nl1, nil, F1*).

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **D-delete** A.3.4, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25

A.4.1.58 active-node(*F, N1, L, T, N2, S*)

if F is a PVST up to node $N1$, T a type of error or catcher 'thrown' by `throw` and L is an abstract list of active (catch) nodes of F **then** $N2$ is the first active node in F which is in L such that T and the catcher carried by $N2$ unify by substitution S.

active-node(*F, N1, L, T, N2, S*) ⇐
unifiable-catcher-ancestor(*F, N1, L, T, N2, S*),
not **exist-younger-unifiable-catcher-ancestor**(*F, N1, L, T, N2*).

NOTE — References: **unifiable-catcher-ancestor** A.4.1.59, **exist-younger-unifiable-catcher-ancestor** A.4.1.60

A.4.1.59 unifiable-catcher-ancestor(*F, N1, L, T, N2, S*)

if F is a PVST up to node $N1$, T a type of error or catcher 'thrown' by `throw` and L is an abstract list of active (catch) nodes of F **then** $N2$ is an active node in F which is in L such that T and the catcher carried by $N2$ unify by substitution S.

unifiable-catcher-ancestor(*F, N1, L, T, N2, S*) ⇐
D-member(*N2, L*),
D-goal of node *N2* **in** *F* **is** *G*,
predication-choice(*G, func*(`catch`, *_T1._nil*)),
L-unify(*T, T1, S*).

NOTE — References: **D-member** A.3.4, **D-goal of node _ in _ is _** A.3.3.4, **predication-choice** A.4.1.2, **L-unify** A.3.5

A.4.1.60 exist-younger-unifiable-catcher-ancestor(*F, N1, L, T, N2*)

if F is a PVST up to node $N1$, T a type of error or catcher 'thrown' by `throw`, L is an abstract list of active (catch) nodes of F and $N2$ an ancestor of $N1$ **then** there exists a younger active ancestor in F which is in L such that T and the catcher it carries are unifiable.

exist-younger-unifiable-catcher-ancestor(*F, N1, L, T, N2*) ⇐
unifiable-catcher-ancestor(*F, N1, L, T, N3, _*),
not **D-equal**(*N2, N3*),
D-conc(*_, N2, N3*).

NOTE — References: **unifiable-catcher-ancestor** A.4.1.59, **D-equal** A.3.1, **D-conc** A.3.4

A.4.1.61 extract-solution-list(*L, L1*)

if L is an abstract list of terms of the form $func(.,V.T.nil)$ such that V and T are terms **then** $L1$ is the list of terms T in L.

extract-solution-list(*nil, func*(`[]`, *nil*)).

extract-solution-list(*func(., V.T.nil).L, func(., T.L1.nil)*) ⇐
extract-solution-list(*L, L1*).

A.4.1.62 variant-members(*L1, LV, LR, W*)

if $L1$ is an abstract list of terms, each of the form $func(.,V.T.nil)$ such that V and T are terms **then** LV is an abstract sublist of $L1$, and W is the corresponding abstract sublist consisting of the first arguments of the elements of LV which are all variants, and LR is $L1$ with all elements of LV removed.

variant-members(*nil, nil, nil, nil*).

variant-members(*L1, func(.,V.T.nil).LV, LR, V.W*) ⇐
D-one-delete(*L1, func(.,V.T.nil), L2*),
find-variant-members(*V, L2, LV, LR, W*).

NOTE — References: **D-one-delete** A.3.4, **find-variant-members** A.4.1.63

A.4.1.63 find-variant-members(*V1, L1, LV, LR, W*)

if $V1$ is a term and $L1$ is a list of terms of the form $func(.,V.T.nil)$ such that V and T are terms **then** LV is a sublist of $L1$ and W is the corresponding sublist of the first arguments of the elements of LV which are all variants of $V1$ and LR is $L1$ except all the elements of LV.

find-variant-members(*V1, L1, nil, L1, nil*)⇐
not **exist-variants**(*V1, L1*).

find-variant-members(*V1, L1, func(., V2, T2).LV, LR, V2.W*) ⇐
select-first-variant(*V1, L1, func(., V2.T2.nil)*),
D-delete(*L1, func(., V2.T2.nil), L2*),
find-variant-members(*V1, L2, LV, LR, W*).

NOTE — References: **exist-variants** A.4.1.64, **select-first-variant** A.4.1.67, **D-delete** A.3.4

A.4.1.64 exist-variants*(V, L)*

iff V is a set of variables and L a list of terms of the form $func(.,V.T.nil)$ such that V is a set of variables and T a term and some element of L has its first argument a variant of V.

exist-variants*(V, L)* ⇐
  select-first-variant*(V, L, _)*.

NOTE — References: **select-first-variant** A.4.1.67.

A.4.1.65 free-var*(T1, G1, V, G2)*

if $T1$ is a term and $G1$ a goal **then** V is a term containing the free variables of $G1$ with respect to T according to definition 7.1.1.4 and $G2$ is the iterated goal according to definition 7.1.6.3.

free-var*(T1, func(^, T.G1.nil), V, G2)* ⇐
  free-var*(T.T1, G1, V, G2)*.

free-var*(T, G, V, G)* ⇐
  not **bagof-goal***(G)*,
  L-witness*(T, V)*.

NOTE — References: **L-witness** A.3.1 **bagof-goal** A.4.1.66

A.4.1.66 bagof-goal*(G)*

iff G is a term whose principal functor is ^/2.

bagof-goal*(func(^, _._.nil))*.

A.4.1.67 select-first-variant*(V, L, T)*

if V is a term and L an abstract list of terms of the form $func(.,V1.T1.nil)$ **then** T is $T1$ (the second argument of the first element in L) such that V and $V1$ are variants.

select-first-variant*(V1, func(., V2.T2.nil).L, func(., V2.T2.nil))* ⇐
  L-variants*(V1, V2)*.

select-first-variant*(V1, func(., V2.T2.nil).L, T)* ⇐
  not **L-variants***(V1, V2)*,
  select-first-variant*(V1, L, T)*.

NOTE — References: **L-variants** A.3.5

A.4.1.68 corresponding-flag-and-value*(Flag, Value, P, C, S)*

if $Flag$ is a flag and $Value$ is a possible value of $Flag$ and P is a list of flags **then** C is the term representing $Flag$ with actual value $Value$ and S is the resulting substitution.

corresponding-flag-and-value*(Flag, Value, C.P, C, S)* ⇐
  D-equal*(C, func(flag, Flag1.Value1._.nil))*,
  L-unify*([Flag, Value], [Flag1, Value1], S)*.

corresponding-flag-and-value*(Flag, Value, C1.P, C, S)* ⇐
  D-equal*(C1, func(flag, Flag1.Value._.nil))*,
  L-not-unifiable*([Flag, Value], [Flag1, Value1])*,
  corresponding-flag-and-value*(Flag, P, C, S)*.

NOTE — References: **D-equal** A.3.1, **L-unify** A.3.5, **L-not-unifiable** A.3.5

A.4.1.69 exist-corresponding-flag-and-value*(Flag, Value, P)*

if $Flag$ is a flag and $Value$ is a possible value of $Flag$ and P is a list of flags **then** there exists a term representing $Flag$ with actual value $Value$.

exist-corresponding-flag-and-value*(Flag, Value, P)* ⇐
  corresponding-flag-and-value*(Flag, Value, P, _, _)*.

NOTE — References: **corresponding-flag-and-value** A.4.1.68

A.5 Control constructs and Built-in predicates

A.5.1 Control constructs

A.5.1.1 or/2 and if-then-else

```
';'('->'(Cond, Then), Else) :- call(Cond), !, Then.
';'('->'(Cond, Then), Else) :- !, Else.

';'(G1, G2) :- G1.
';'(G1, G2) :- G2.
```

NOTE — References: `call` A.5.1.4

A.5.1.2 if-then/2 - conditional

The conditional is a basic construct. It is described in section A.2.3 and corresponds to the bootstrapped definition (concrete syntax form):

```
'->'(Cond, Then) :-
        call(Cond),!,Then.
```

NOTE — References: `call` A.5.1.4

A.5.1.3 true/0

treat-bip*(F, N, func(*`true`*, nil), F1)* ⇐
  D-label of node *N* **in** *F* **is** *Nl*,
  D-equal*(Nl, nd(N, G, P, _, E, S, L, _))*,
  final-resolution-step*(G, empsubs, P, G1, Q)*,
  D-equal*(Nl1, nd(zero.N, G1, P, Q, E, empsubs, L, partial))*,
  addchild*(F, Nl, Nl1, nil, F1)*.

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25

A.5.1.4 fail/0

`fail/0` is formally defined by the fact that there is no clause in its packet.

A.5.1.5 !/0 - cut

treat-bip*(F, N, func(*`!`*, J.nil), F1)* ⇐
  D-label of node *N* **in** *F* **is** *Nl*,
  D-equal*(Nl, nd(N, G, P, _, E, _, L, _))*,
  final-resolution-step*(G, empsubs, P, G1, Q)*,
  D-equal*(Nl1, nd(zero.N, G1, P, Q, E, empsubs, L, partial))*,
  D-parent-or-root*(J, F, J1)*,
  cut-all-choice-point*(F, N, J1, F2)*,

D-label of node *N* **in** *F2* **is** *Nl2*,
addchild(*F2, Nl2, Nl1, nil, F1*).

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **final-resolution-step** A.4.1.33, **D-parent-or-root** A.3.3.3, **cut-all-choice-point** A.4.1.38, **addchild** A.4.1.25

A.5.1.6 call/1

treat-bip(*F, N, func*(`call`, *Goal.nil*), *F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, _, E, S, L, _)*),
D-term-to-body(*Goal, Goal1*),
flag-cut(*Goal1, zero.N, Goal2*),
erase(*G, G2*),
D-equal(*G1, func*(`&`, *Goal2.G2.nil*)),
predication-choice(*G1, A*),
D-packet(*P, A, Q*),
D-equal(*Nl1, nd(zero.N, G1, P, Q, E, empsubs, L, partial)*),
addchild(*F, Nl, Nl1, nil, F1*).

Error cases:

in-error(*_, func*(`call`, *Goal.nil*), *instantiation-error*) ⇐
L-var(*Goal*).

in-error(*_, func*(`call`, *Goal.nil*), *type-error(callable, Goal)*) ⇐
not **L-var**(*Goal*),
not **D-is-a-callable-term**(*Goal*).

in-error(*_, func*(`call`, *Goal.nil*), *type-error(callable, Goal)*) ⇐
not **D-is-a-body**(*Goal*).

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **D-term-to-body** A.3.1, **flag-cut** A.4.1.28, **erase** A.4.1.35, **predication-choice** A.4.1.2, **D-packet** A.3.8, **addchild** A.4.1.25, **L-var** A.3.1, **D-is-a-callable-term** A.3.1, **D-is-a-body** A.3.1

A.5.1.7 catch/3

treat-bip(*F, N, func*(`catch`, *Go.Cat.Rec.nil*), *F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, _, E, _, L, _)*),
erase(*G, G2*),
D-equal(*G1,* `call`*(Go)* `&` *inactivate(N)* `&` *G2*),
predication-choice(*G1, A1*),
D-packet(*P, A1, Q1*),
D-equal(*Nl1, nd(zero.N, G1, P, Q1, E, empsubs, N.L, partial)*),
addchild(*F, Nl, Nl1, nil, F1*).

NOTE — Formally: func(`&`, func(call, Go.nil).func(`&`, special-pred(inactivate, N.nil).G2.nil).nil)

Error cases:

in-error(*_, func*(`catch`, *Goal.Cat.Rec.nil*), *instantiation-error*) ⇐
L-var(*Goal*).

in-error(*_, func*(`catch`, *Goal.Cat.Rec.nil*), *type-error(callable, Goal)*) ⇐
not **L-var**(*Goal*),
not **D-is-a-callable-term**(*Goal*).

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **erase** A.4.1.35, **predication-choice** A.4.1.2, **D-packet** A.3.8, **addchild** A.4.1.25, **D-is-a-callable-term** A.3.1, **L-var** A.3.1

A.5.1.8 throw/1

treat-bip(*F, N, func*(`throw`, *T.nil*), *F1*) ⇐
D-active catchers of node *N* **in** *F* **is** *L*,
active-node(*F, N, L, T, M, S1*),
D-label of node *M* **in** *F* **is** *Ml*,
D-equal(*Ml, nd(M, G, P, nil, E, _, L1, _)*),
predication-choice(*G, A*),
D-equal(*A, func*(`catch`, *Go.T1.Rec.nil*)),
erase(*G, G2*),
D-equal(*G3, func*(`&`, *func(call, Rec.nil).G2.nil*)),
L-instance(*G3, S1, G1*),
predication-choice(*G1, A1*),
D-packet(*P, A1, Q1*),
D-equal(*Nl1, nd(s(zero).M, G1, P, Q1, E, S1, L1, partial)*),
cut-all-choice-point(*F, N, M, F2*),
D-label of node *M* **in** *F2* **is** *Ml1*,
addchild(*F2, Ml1, Nl1, nil, F1*).

NOTE — A system error is generated if the argument of `throw` does not unify with the catcher argument of any call of `catch/3`. This case is allowed by introducing a general catcher in the root (see **semantics** A.4.1.1).

NOTE — References: **D-active catchers of node _ in _ is _** A.3.3.4, **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **active-node** A.4.1.58, **predication-choice** A.4.1.2, **erase** A.4.1.35, **L-instance** A.3.5, **D-packet** A.3.8, **cut-all-choice-point** A.4.1.38, **addchild** A.4.1.25,

---

A.5.2 Term unification

A.5.2.1 =/2 - Prolog unify

execute-bip(*_, func(=, X.Y.nil), S*) ⇐
L-unify(*X, Y, S*).

NOTE — References: **L-unify** A.3.5

A.5.2.2 unify_with_occurs_check/2 - unify

execute-bip(*_, func*(`unify_with_occurs_check`, *X.Y.nil*), *S*) ⇐
L-unify-occur-check(*X, Y, S*).

NOTE — References: **L-unify-occur-check** A.3.5

A.5.2.3 \=/2 - not Prolog unifiable

execute-bip(*_, func*(`\=`, *X.Y.nil*), *empsubs*) ⇐
L-not-unifiable(*X, Y*).

NOTE — References: **L-not-unifiable** A.3.5

A.5.3 Type testing

A.5.3.1 var/1

execute-bip(*_, func*(`var`, *X.nil*), *empsubs*) ⇐
L-var(*X*).

NOTE — References: **L-var** A.3.1

A.5.3.2 atom/1

execute-bip(_, *func*(`atom`, *X.nil*), *empsubs*) ⇐
D-is-an-atom(*X*).

NOTE — References: **D-is-an-atom** A.3.4

A.5.3.3 integer/1

execute-bip(_, *func*(`integer`, *X.nil*), *empsubs*) ⇐
D-is-an-integer(*X*).

NOTE — References: **D-is-an-integer** A.3.1

A.5.3.4 real/1

execute-bip(_, *func*(`real`, *X.nil*), *empsubs*) ⇐
D-is-a-float(*X*).

NOTE — References: **D-is-a-float** A.3.1

A.5.3.5 atomic/1

execute-bip(_, *func*(`atomic`, *X.nil*), *empsubs*) ⇐
D-is-atomic(*X*).

NOTE — References: **D-is-atomic** A.3.4

A.5.3.6 compound/1

execute-bip(_, *func*(`compound`, *T.nil*), *empsubs*) ⇐
not **L-var**(*T*),
not **D-is-atomic**(*T*).

NOTE — References: **L-var** A.3.1, **D-is-atomic** A.3.4

A.5.3.7 nonvar/1

execute-bip(_, *func*(`nonvar`, *X.nil*), *empsubs*) ⇐
not **L-var**(*X*).

NOTE — References: **L-var** A.3.1

A.5.3.8 number/1

execute-bip(_, *func*(`number`, *X.nil*), *empsubs*) ⇐
D-is-a-number(*X*).

NOTE — References: **D-is-a-number** A.3.1

A.5.4 Term comparison

A.5.4.1 ==/2 - identical

execute-bip(_, *func*(==, *X.Y.nil*), *empsubs*) ⇐
D-equal(*X*, *Y*).

NOTE — References: **D-equal** A.3.1

A.5.4.2 \==/2 - not identical

execute-bip(_, *func*(`\==`, *X.Y.nil*), *empsubs*) ⇐
not **D-equal**(*X*, *Y*).

NOTE — References: **D-equal** A.3.1

A.5.4.3 @</2 - term less than

execute-bip(_, *func*(`@<`, *X.Y.nil*), *empsubs*) ⇐
term-ordered(*X*, *Y*).

NOTE — References: **term-ordered** A.4.1.40

A.5.4.4 @=</2 - term less than or equal

execute-bip(_, *func*(`@=<`, *X.Y.nil*), *empsubs*) ⇐
term-ordered(*X*, *Y*).

execute-bip(_, *func*(`@=<`, *X.Y.nil*), *empsubs*) ⇐
D-equal(*X*, *Y*).

NOTE — References: **term-ordered** A.4.1.40, **D-equal** A.3.1

A.5.4.5 @>/2 - term greater than

execute-bip(_, *func*(`@>`, *X.Y.nil*), *empsubs*) ⇐
term-ordered(*Y*, *X*).

NOTE — References: **term-ordered** A.4.1.40

A.5.4.6 @>=/2 - term greater than or equal

execute-bip(_, *func*(`@>=`, *X.Y.nil*), *empsubs*) ⇐
term-ordered(*Y*, *X*).

execute-bip(_, *func*(`@>=`, *X.Y.nil*), *empsubs*) ⇐
D-equal(*X*, *Y*).

NOTE — References: **term-ordered** A.4.1.40, **D-equal** A.3.1

A.5.5 Term creation and decomposition

A.5.5.1 functor/3

execute-bip(_, *func*(`functor`, *func(N, L).Functor.Arity.nil*), *S*) ⇐
D-length-list(*L*, *P*),
L-unify(`[`*Functor, Arity* `]`, `[`*func(N, nil), func(P, nil)*`]`, *S*).

NOTE — Formally:
L-unify(*func(*`.`*, Functor.func(*`.`*, Arity.func(*`[]`*, nil).nil).nil),*
func(`.`*, func(N, nil).func(*`.`*, func(P, nil).func(*`[]`*, nil).nil).nil), S).*

execute-bip(*F*, *func*(`functor`, *Term.func(N, nil).func(P, nil).nil*),
S) ⇐
L-var(*Term*),
D-buildlist-of-var(*L*, *P*),
L-rename(*F*, *L*, *L1*),
L-unify(*Term*, *func(N, L1)*, *S*).

Error cases:

in-error(_, *func*(`functor`, *Term.Functor.Arity.nil), instantiation-error)* ⇐
L-var(*Term*),
L-var(*Functor*).

in-error(_, *func*(`functor`, *Term.Functor.Arity.nil), instantiation-error)* ⇐
L-var(*Term*),
L-var(*Arity*).

in-error(_, *func*(`functor`, *Term.Functor.Arity.nil), type-error(integer,Arity))* ⇐
not **L-var**(*Arity*),
not **D-is-an-integer**(*Arity*).

in-error(_, *func*(`functor`, *Term.Functor.Arity.nil), representation-error(exceeded_max_arity, Arity))* ⇐
D-is-an-integer(*Arity*),
D-equal(*Arity, func(N, nil)*),
L-integer-less(`max-arity`, *N*).

in-error(_, *func*(`functor`, *Term.Functor.Arity.nil), domain-error(> (0), Arity))* ⇐
not **L-var**(*Arity*),
D-is-a-neg-integer(*Arity*).

in-error(_, *func*(`functor`, *Term.Functor.Arity.nil), type-error(atomic, Functor))* ⇐
not **L-var**(*Functor*),
not **D-is-atomic**(*Functor*).

in-error(_, *func*(`functor`, *Term.Functor.Arity.nil), type-error(atom, Functor))* ⇐
D-is-an-integer(*Arity*),
D-equal(*Arity, func(N, nil)*),
L-integer-less(*N,0*),
not **D-is-an-atom**(*Functor*).

NOTE — References: **D-length-list** A.3.4, **L-unify** A.3.5, **L-var** A.3.1, **D-buildlist-of-var** A.3.4, **L-rename** A.3.5, **D-is-atomic** A.3.4, **D-is-an-atom** A.3.4, **D-is-a-neg-integer** A.3.1, **D-is-an-integer** A.3.1, **D-equal** A.3.1, **L-integer-less** A.3.6,

A.5.5.2 arg/3

execute-bip(_, *func*(`arg`, *func(I, nil).func(F, L).Arg.nil), S)* ⇐
D-position(*Arg1, L, I*),
L-unify(*Arg, Arg1, S*).

Error cases:

in-error(_, *func*(`arg`, *N.Term.Arg.nil), instantiation-error)* ⇐
L-var(*N*).

in-error(_, *func*(`arg`, *N.Term.Arg.nil), instantiation-error)* ⇐
L-var(*Term*).

in-error(_, *func*(`arg`, *N.Term.Arg.nil), type-error(compound,Term))* ⇐
D-is-atomic(*Term*).

in-error(_, *func*(`arg`, *N.Term.Arg.nil), type-error(integer,N))* ⇐
not **L-var**(*N*),
not **D-is-an-integer**(*N*).

in-error(_, *func*(`arg`, *N.Term.Arg.nil), domain-error(between(1, Arity),N))* ⇐
not **L-var**(*N*),
D-is-a-neg-integer(*N*),
D-arity(*Term, func(Arity,nil)*).

in-error(_, *func*(`arg`, *N.Term.Arg.nil), domain-error(between(1, Arity),N))* ⇐
not **L-var**(*N*),
D-equal(*N, func*(`0`, *nil)*),
D-arity(*Term, func(Arity,nil)*).

in-error(_, *func*(`arg`, *N.Term.Arg.nil), domain-error(between(1, Arity),N))* ⇐
D-is-an-integer(*N*),
D-equal(*N, func(I, nil)*),
D-arity(*Term, func(Arity,nil)*),
L-integer-less(*Arity, I*).

NOTE — References: **D-position** A.3.4, **L-unify** A.3.5, **D-is-an-integer** A.3.1, **D-is-a-neg-integer** A.3.1, **D-arity** A.3.1, **D-equal** A.3.1, **D-length-list** A.3.4, **L-integer-less** A.3.6, **D-is-atomic** A.3.4, **L-var** A.3.1

A.5.5.3 =../2 - univ

execute-bip(_, *func*(=. ., *func(F, L).List.nil), S)* ⇐
D-transform-list(*L, L1*),
L-unify(*List, func*(., *func(F, nil).L1.nil), S*).

execute-bip(_, *func*(=. ., *Term.func*(., *func(F, nil).L.nil).nil), S)* ⇐
D-transform-list(*L1, L*),
L-unify(*Term, func(F, L1), S*).

Error cases:

in-error(_, *func*(=. ., *Term.List.nil), instantiation-error)* ⇐
L-var(*Term*),
L-var(*List*).

in-error(_, *func*(=. ., *Term.List.nil), type-error(proper_list, List))* ⇐
not **L-var**(*List*),
not **D-equal**(*List, func*(., _._nil)).

in-error(_, *func*(=. ., *Term.List.nil), instantiation-error)* ⇐
L-var(*Term*),
D-is-a-partial-list(*List*).

in-error(_, *func*(=. ., *Term.List.nil), type-error(proper_list, List))* ⇐
L-var(*Term*),
D-equal(*List, func*(., *H.T.nil)*),
not **L-var**(*T*),
not **D-is-a-list**(*T*),
not **D-is-a-partial-list**(*T*).

in-error(_, *func*(=. ., *Term.List.nil), instantiation-error)* ⇐
L-var(*Term*),
D-equal(*List, func*(., *H.T.nil)*),
L-var(*H*).

in-error(_, *func*(=. ., *Term.List.nil), type-error(atomic, H))* ⇐
D-equal(*List, func*(., *H.T.nil)*),
not **L-var**(*H*),
not **D-is-atomic**(*H*).

in-error(_, *func*(=. ., *Term.List.nil), type-error(atom, H))* ⇐
D-equal(*List, func*(., *H.T.nil)*),

not **D-equal**(*T, func(*[], *nil)),*
not **D-is-an-atom**(*H).*

in-error(_ *func(=.., Term.List.nil), representation-error(exceeded_max_arity))* ⇐
D-is-a-list(*List),*
D-length-list(*List, N),*
L-integer-plus(*N,* 1*, N1),*
L-integer-less(max-arity*, N1).*

NOTE — References: **D-transform-list** A.3.4, **L-var** A.3.1, **D-is-a-partial-list** A.3.4, **D-length-list** A.3.4, **L-integer-less** A.3.6, **L-integer-plus** A.3.6, **D-is-a-list** A.3.4, **D-equal** A.3.1, **L-unify** A.3.5, **D-is-a-number** A.3.1, **D-is-atomic** A.3.4, **D-is-an-atom** A.3.4

A.5.5.4 copy_term/2

execute-bip(*F, func(*copy_term*, Term1.Term2.nil), S)* ⇐
L-rename(*F, Term1, Term),*
L-unify(*Term, Term2, S).*

NOTE — References: **L-rename** A.3.5, **L-unify** A.3.5

A.5.6 Arithmetic evaluation - is/2

```
is(R, E) :-
```
value (E, V),
R = V.

Error cases:

in-error(_ *func(*is*, Result.Exp.nil), instantiation-error)* ⇐
L-var(*Exp).*

NOTE — Other errors are raised by computation of the expression (see A.4.1.15).

NOTE — References: **value** A.3.8 and A.4.1.17, **L-var** A.3.1

A.5.7 Arithmetic comparison

Op1 (Exp1, Exp2) :-
compare (Exp1, *Op*1, Exp2).

with *Op*1 ∈ { =:= ,=\= , < , > , =< , => , }.

These operations are defined in the body (section 8.7.1).

Error cases:

in-error(_ *func(Op1, Exp1.Exp2.nil), instantiation-error)* ⇐
L-var(*Exp1).*

in-error(_ *func(Op1, Exp1.Exp2.nil), instantiation-error)* ⇐
L-var(*Exp2).*

NOTE — Other errors are raised by computation of the expressions.

NOTE — References: **value** A.3.8 and A.4.1.17, **L-var** A.3.1

A.5.8 Clause retrieval and information

A.5.8.1 clause/2

NOTE — clause/2 is re-executable.

treat-bip(*F, N, func(*clause*, Head.Body.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(N, G, P, Q, E,* _ *L,* _*)),*
first-match-clause(*F, Q, func(*:–*,Head.Body.nil), Cl, S1, R),*
final-resolution-step(*G, S1, P, G1, Q1),*
D-number-of-child(*N, F, N1),*
D-equal(*Nl1, nd(N1.N, G1, P, Q1, E, S1, L, partial)),*
addchild(*F, Nl, Nl1, R, F1).*

treat-bip(*F, N, func(*clause*, Head.Body.nil), F1)* ⇐
D-choice of node *N* **in** *F* **is** *Q,*
not **exist-match-clause**(*F, Q, func(*:–*,Head.Body.nil)),*
erasepack(*F, N, F1).*

Error cases:

in-error(_ *func(*clause*, Head.Body.nil), instantiation-error)* ⇐
L-var(*Head).*

in-error(_ *func(*clause*, Head.Body.nil), type-error(callable, Head))* ⇐
not **L-var**(*Head),*
not **D-is-a-callable-term**(*Head).*

in-error(*F, func(*clause*, Head.Body.nil), permission-error(static_procedure,Func))* ⇐
D-root-database-and-env(*F, DB,* _*),*
D-name(*Head, func(Func,* _*)),*
D-arity(*Head, Ar),*
corresponding-pred-definition(*func(/, func(Func,* _*).Ar.nil), DB, def(*_*DS,*_*),* _*),*
D-equal(*DS, static).*

in-error(_ *func(*clause*, Head.Body.nil), permission-error(built_in,Func))* ⇐
D-is-a-bip(*Head),*
D-equal(*Head,func(Func,* _*)).*

NOTE — References: **D-label of node** _ **in** _ **is** _ A.3.3.4, **D-choice of node** _ **in** _ **is** _ A.3.3.4, **D-equal** A.3.1, **final-resolution-step** A.4.1.33, **first-match-clause** A.4.1.47, **D-number-of-child** A.3.3.3, **addchild** A.4.1.25, **exist-match-clause** A.4.1.48, **erasepack** A.4.1.23, **D-is-a-callable-term** A.3.1, **D-root-database-and-env** A.3.3.4, **D-name** A.3.1, **D-arity** A.3.1, **corresponding-pred-definition** A.4.1.51 **L-var** A.3.1, **D-is-a-bip** A.3.8

A.5.8.2 current_predicate/1

NOTE — current_predicate/1 is re-executable.

treat-bip(*F, N, func(*current_predicate*, PI.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(N, G, P, Q, E,* _ *L,* _*)),*
corresponding-pred(*PI, Q, PI1, S),*
D-delete-packet(*Q, PI1, R),*
final-resolution-step(*G, S, P, G1, Q1),*
D-number-of-child(*N, F, N1),*
D-equal(*Nl1, nd(N1.N, G1, P, Q1, E, S, L, partial)),*
addchild(*F, Nl, Nl1, R, F1).*

treat-bip(*F, N, func(*current_predicate*, PI.nil), F1)* ⇐
D-choice of node *N* **in** *F* **is** *Q,*

not **exist-corresponding-pred***(PI, Q),*
erasepack*(F, N, F1).*

Error cases:

in-error*(_, func(*`current_predicate`*, PI.nil), type-error(predicate_indicator,PI))* ⇐
not **L-var***(PI),*
not **D-is-a-pred-indicator-pattern***(PI).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-choice of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **final-resolution-step** A.4.1.33, **corresponding-pred** A.4.1.49, **D-number-of-child** A.3.3.3, **addchild** A.4.1.25, **D-delete-packet** A.3.8, **exist-corresponding-pred** A.4.1.50, **erasepack** A.4.1.23, **L-var** A.3.1, **D-is-a-pred-indicator-pattern** A.3.2

A.5.9 Clause creation and destruction

A.5.9.1 asserta/1

treat-bip*(F, N, func(*`asserta`*, T.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal*(Nl, nd(N, G, OldDB, _, E, _, L, _)),*
D-clause-to-pred-indicator*(T,PI),*
corresponding-pred-definition*(PI, OldDB, def(PI,SD,P), _),*
D-term-to-clause*(T, T1),*
D-conc*(T1.nil, P, P1),*
D-delete*(OldDB, def(PI,SD,P).OldDB1),*
D-equal*(NewDB, def(PI,SD,P1).OldDB1),*
final-resolution-step*(G, empsubs, NewDB, G1, Q),*
D-equal*(Nl1, nd(zero.N, G1, NewDB, Q, E, empsubs, L, partial)),*
addchild*(F, Nl, Nl1, nil, F2),*
D-modify-database*(F2, NewDB, F1).*

treat-bip*(F, N, func(*`asserta`*, T.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal*(Nl, nd(N, G, OldDB, _, E, _, L, _)),*
D-clause-to-pred-indicator*(T,PI),*
not **exist-corresponding-pred-definition***(PI, OldDB),*
D-term-to-clause*(T, T1),*
D-equal*(NewDB, def(PI,dynamic,T1.nil).OldDB),*
final-resolution-step*(G, empsubs, NewDB, G1, Q),*
D-equal*(Nl1, nd(zero.N, G1, NewDB, Q, E, empsubs, L, partial)),*
addchild*(F, Nl, Nl1, nil, F2),*
D-modify-database*(F2, NewDB, F1).*

Error cases:

in-error*(_, func(*`asserta`*, X.nil), instantiation-error)* ⇐
L-var*(X).*

in-error*(_, func(*`asserta`*, X.nil), instantiation-error)* ⇐
D-equal*(X, func(*`:-`*, H.B.nil)),*
L-var*(H).*

in-error*(_, func(*`asserta`*, X.nil), type-error(callable,X)* ⇐
not **L-var***(X),*
not **D-is-a-clause***(X).*

in-error*(_, func(*`asserta`*, C.nil), permission-error(built_in, PI))*
⇐
D-root-database-and-env*(F, DB, _),*
D-clause-to-pred-indicator*(C,PI),*
corresponding-pred-definition*(PI, DB, def(_,DS,_), _),*
D-equal*(DS, static).*

in-error*(_, func(*`asserta`*, func(*`:-`*, H.B.nil).nil), permission-error(static_procedure, func(*`/`*, F.A.nil)))* ⇐
D-is-a-bip*(H)*
D-equal*(H, F),*
D-arity*(H, A).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **corresponding-pred-definition** A.4.1.51 **exist-corresponding-pred-definition** A.4.1.52 **D-clause-to-pred-indicator** A.3.1, **D-term-to-clause** A.3.1, **D-conc** A.3.4, **D-delete** A.3.4, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-modify-database** A.3.3.6, **D-is-a-clause** A.3.1, **L-var** A.3.1, **D-root-database-and-env** A.3.3.4, **D-is-a-bip** A.3.8, **D-arity** A.3.1

A.5.9.2 assertz/1

treat-bip*(F, N, func(*`assertz`*, T.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal*(Nl, nd(N, G, OldDB, _, E, _, L, _)),*
D-clause-to-pred-indicator*(T,PI),*
corresponding-pred-definition*(PI, OldDB, def(PI,SD,P), _),*
D-term-to-clause*(T, T1),*
D-conc*(P, T1.nil, P1),*
D-delete*(OldDB, def(PI,SD,P).OldDB1),*
D-equal*(NewDB, def(PI,SD,P1).OldDB1),*
final-resolution-step*(G, empsubs, NewDB, G1, Q),*
D-equal*(Nl1, nd(zero.N, G1, NewDB, Q, E, empsubs, L, partial)),*
addchild*(F, Nl, Nl1, nil, F2),*
D-modify-database*(F2, NewDB, F1).*

treat-bip*(F, N, func(*`assertz`*, T.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal*(Nl, nd(N, G, OldDB, _, E, _, L, _)),*
D-clause-to-pred-indicator*(T,PI),*
not **exist-corresponding-pred-definition***(PI, OldDB),*
D-term-to-clause*(T, T1),*
D-equal*(NewDB, def(PI,dynamic,T1.nil).OldDB),*
final-resolution-step*(G, empsubs, NewDB, G1, Q),*
D-equal*(Nl1, nd(zero.N, G1, NewDB, Q, E, empsubs, L, partial)),*
addchild*(F, Nl, Nl1, nil, F2),*
D-modify-database*(F2, NewDB, F1).*

Error cases:

in-error*(_, func(*`assertz`*, X.nil), instantiation-error)* ⇐
L-var*(X).*

in-error*(_, func(*`assertz`*, X.nil), instantiation-error)* ⇐
D-term-to-clause*(X, func(*`:-`*, H.B.nil)),*
L-var*(H).*

in-error*(_, func(*`assertz`*, X.nil), type-error(callable, X)* ⇐
not **L-var***(X),*
not **D-is-a-clause***(X).*

in-error*(_, func(*`assertz`*, C.nil), permission-error(built_in, PI))*
⇐
D-root-database-and-env*(F, DB, _),*
D-clause-to-pred-indicator*(C,PI),*
corresponding-pred-definition*(PI, DB, def(_,DS,_), _),*
D-equal*(DS, static).*

in-error(_, *func*(`assertz`, *func*(`:-`, *H.B.nil).nil), *permission-error(static_procedure, func(/, F.A.nil))*) ⇐
D-is-a-bip(*H*)
D-equal(*H, F*),
D-arity(*H, A*).

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **corresponding-pred-definition** A.4.1.51 **exist-corresponding-pred-definition** A.4.1.52 **D-clause-to-pred-indicator** A.3.1, **D-term-to-clause** A.3.1, **D-conc** A.3.4, **D-delete** A.3.4, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-modify-database** A.3.3.6, **D-is-a-clause** A.3.1, **L-var** A.3.1, **D-root-database-and-env** A.3.3.4, **D-is-a-bip** A.3.8, **D-arity** A.3.1

A.5.9.3 retract/1

NOTE — `retract/1` is re-executable.

treat-bip(*F, N, func*(`retract`, *T.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, OldDB, _, E, _, L, _)*),
D-clause-to-pred-indicator(*T,PI*),
corresponding-pred-definition(*PI, OldDB, def(PI,SD,P), _*),
D-fact-to-clause(*T, C*),
first-match-clause(*F, P, C, Cl, S1, R*),
D-delete(*P, Cl, P1*),
D-delete(*OldDB, def(PI,SD,P).OldDB1*),
D-equal(*NewDB, def(PI,SD,P1).OldDB1*),
final-resolution-step(*G, S1, NewDB, G1, Q1*),
D-number-of-child(*N, F, N1*),
D-equal(*Nl1, nd(N1.N, G1, NewDB, Q1, E, S1, L, partial)*),
addchild(*F, Nl, Nl1, R, for(M, F2, F3)*),
D-modify-database(*F2, NewDB, F4*),
D-modify-database(*F3, NewDB, F5*),
D-equal(*F1, for(M, F4, F5)*).

treat-bip(*F, N, func*(`retract`, *T.nil), F1*) ⇐
D-database of node *N* **in** *F* **is** *DB*,
D-clause-to-pred-indicator(*T,PI*),
not **exist-corresponding-pred-definition**(*PI, DB*),
erasepack(*F, N, F1*).

treat-bip(*F, N, func(retract, T.nil), F1*) ⇐
D-database of node *N* **in** *F* **is** *DB*,
D-clause-to-pred-indicator(*T,PI*),
corresponding-pred-definition(*PI, DB, def(_,_,P), _*),
D-fact-to-clause(*T, C*),
not **exist-match-clause**(*F, P, C*),
erasepack(*F, N, F1*).

Error cases:

in-error(_, *func*(`retract`, *C.nil), instantiation-error*) ⇐
L-var(*C*).

in-error(_, *func*(`retract`, *func*(`:-`, *H.B.nil).nil), instantiation-error*) ⇐
L-var(*H*).

in-error(_, *func*(`retract`, *C.nil), type-error(callable, H)*) ⇐
not **L-var**(*C*),
not **D-is-a-callable-term**(*C*).

in-error(_, *func*(`retract`, *func*(`:-`, *H.B.nil).nil), type-error(callable, H)*) ⇐
not **L-var**(*H*),
not **D-is-a-callable-term**(*H*).

in-error(*F*, *func*(`retract`, *C.nil), permission-error(static_procedure, PI)*) ⇐
D-root-database-and-env(*F, DB, _*),
D-clause-to-pred-indicator(*C,PI*),
corresponding-pred-definition(*PI, DB, def(_,DS,_), _*),
D-equal(*DS, static*).

in-error(_, *func*(`retract`, *func*(`:-`, *H.B.nil).nil), permission-error(built_in, func(/, F.A.nil))*) ⇐
D-is-a-bip(*H*),
D-name(*H, F*),
D-arity(*H, A*).

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **D-clause-to-pred-indicator** A.3.1, **corresponding-pred-definition** A.4.1.51, **D-fact-to-clause** A.3.1, **first-match-clause** A.4.1.47, **D-delete** A.3.4, **final-resolution-step** A.4.1.33, **D-number-of-child** A.3.3.3, **addchild** A.4.1.25, **D-modify-database** A.3.3.6, **exist-match-clause** A.4.1.48, **erasepack** A.4.1.23, **D-is-a-callable-term** A.3.1, **L-var** A.3.1, **D-root-database-and-env** A.3.3.4, **D-is-a-bip** A.3.8, **D-arity** A.3.1

A.5.9.4 abolish/1

treat-bip(*F, N, func*(`abolish`, *PI.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, OldDB, _, E, _, L, _)*),
corresponding-pred-definition(*PI, OldDB, def(PI,SD,P), _*),
D-delete(*OldDB, def(PI,SD,P), NewDB*),
final-resolution-step(*G, empsubs, NewDB, G1, Q*),
D-equal(*Nl1, nd(zero.N, G1, NewDB, Q, E, empsubs, L, partial)*),
addchild(*F, Nl, Nl1, nil, F2*),
D-modify-database(*F2, NewDB, F1*).

Error cases:

in-error(_, *func*(`abolish`, *PI.nil), instantiation-error*) ⇐
L-var(*PI*).

in-error(_, *func*(`abolish`, *PI.nil), type-error(predicate_indicator, PI)*)⇐
not **L-var**(*PI*),
not **D-name**(*PI, func(/, nil)*).

in-error(_, *func*(`abolish`, *func(/, At.Ar.nil).nil), type-error(atom, At)*) ⇐
not **L-var**(*At*),
not **D-is-an-atom**(*At*).

in-error(_, *func*(`abolish`, *func(/, At.Ar.nil).nil), domain-error(>= (0), Ar)*) ⇐
D-equal(*Ar, func(N, nil)*),
L-integer-less(*N, 0*).

in-error(_, *func*(`abolish`, *func(/, At.Ar.nil).nil), representation-error(<= (max_arity), Ar)*) ⇐
D-equal(*Ar, func(N, nil)*),
L-integer-less(*max_arity, N*).

in-error(_, *func*(`abolish`, *func(/, At.Ar.nil).nil), type-error(integer, Ar)*) ⇐
not **L-var**(*Ar*),
not **D-is-an-integer**(*Ar*).

in-error(_, *func*(`abolish`, *PI.nil), permission-error(static_procedure, PI)*) ⇐
D-root-database-and-env(*F, DB, _*),

corresponding-pred-definition(*PI, DB, def(_,DS,_), _*),
D-equal(*DS, static*).

in-error(*_ func*(`abolish`, *PI.nil), permission-error(built_in, PI)*) ⇐
D-is-a-bip-indicator(*PI*).

NOTE — References: **D-label of node _ in _ is _** A.3.3.4 **D-equal** A.3.1, **D-delete** A.3.4, **corresponding-pred-definition** A.4.1.51, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-modify-database** A.3.3.6, **L-var** A.3.1, **D-name** A.3.1, **D-is-an-atom** A.3.4 **L-integer-less** A.3.6, **D-is-an-integer** A.3.1, **D-root-database-and-env** A.3.3.4, **D-is-a-bip-indicator** A.3.2

A.5.10 All solutions

A.5.10.1 findall/3

treat-bip(*F, N, func*(`findall`, *T.Go.B.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, _, E, _, L, _)*),
complete-semantics(*P, func(call, Go.nil), E, F2*),
not **untrapped-error**(*F2*),
list-of-ans-subs(*F2, L2*),
list-of-instance(*L2, T, L3*),
L-rename-except(*F, T, L3, L1*),
L-unify(*B, L1, S1*),
final-resolution-step(*G, S1, P, G1, Q1*),
D-equal(*Nl1, nd(zero.N, G1, P, Q1, E, S1, L, partial)*),
addchild(*F, Nl, Nl1, nil, F4*),
D-root-database-and-env(*F2, P2, E2*),
D-modify-database(*F4, P2, F3*),
D-modify-environment(*F3, E2, F1*).

treat-bip(*F, N, func*(`findall`, *T.Go.B.nil), F1*) ⇐
D-database of node *N* **in** *F* **is** *P*,
D-environment of node *N* **in** *F* **is** *E*,
complete-semantics(*P, func(call, Go.nil), E, F2*),
not **untrapped-error**(*F2*),
list-of-ans-subs(*F2, L2*),
list-of-instance(*L2, F, T, L3*),
L-rename-except(*F, T, L3, L1*),
L-not-unifiable(*B, L1*),
erasepack(*F, N, F1*).

treat-bip(*F, N, func*(`findall`, *T.Go.B.nil), F1*) ⇐
D-database of node *N* **in** *F* **is** *P*,
D-environment of node *N* **in** *F* **is** *E*,
complete-semantics(*P, func(call, Go.nil), E, F2*),
untrapped-error(*F2*),
treat-bip(*F, N, func*(`throw`, *untrapped-error-in-findall.nil), F1*).

Error cases:

in-error(*_ func*(`findall`, *Term.Goal.Bag.nil), instantiation-error*) ⇐
L-var(*Goal*).

in-error(*_ func*(`findall`, *Term.Goal.Bag.nil), type-error(callable, Goal)*) ⇐
not **L-var**(*Goal*),
not **D-is-a-callable-term**(*Goal*).

in-error(*_ func*(`findall`, *Term.Goal.Bag.nil), type-error(list, Bag)*) ⇐
not **L-var**(*Bag*),
not **D-is-a-list**(*Bag*).

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-database of node _ in _ is _** A.3.3.4, **D-environment of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **complete-semantics** A.4.1.42, **untrapped-error** A.4.1.46, **list-of-ans-subs** A.4.1.44, **list-of-instance** A.4.1.45, **L-unify** A.3.5, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-root-database-and-env** A.3.3.4, **D-modify-database** A.3.3.6, **D-modify-environment** A.3.3.6, **L-not-unifiable** A.3.5, **erasepack** A.4.1.23, **treat-bip** A.4.1.31, **L-var** A.3.1, **D-is-a-callable-term** A.3.1, **D-is-a-list** A.3.4

A.5.10.2 bagof/3

First visit of node N: empty list of choices (see "nil" in Nl in the second literal of the body)
Case of success

treat-bip(*F, N, func*(`bagof`, *Term.Goal.Bag.nil), F1*) ⇐
% As in `findall`*:*
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, nil, E, _, L, _)*),
free-var(*nil, Term ^Goal, V, Goal1*),
complete-semantics(*P, Goal1, E, F2*),
not **untrapped-error**(*F2*),
list-of-ans-subs(*F2, L2*),
list-of-instance(*L2, func(., V.Term.nil), L4*),
L-rename-except(*F, func(., V.Term.nil), L4, L1*),
% Test of success:
not **D-equal**(*L2, nil*),
variant-members(*L1, VL, LR, W*),
L-unify-members-list(*V.W, Phi*),
extract-solution-list(*VL, LT*),
L-instance(*LT, Phi, LT1*),
L-unify(*Bag, LT1, Mu*),
L-composition(*Phi, Mu, S*),
% As in `findall` *except backtracking*
% (LR stored in node N; if LR is empty the node is
% completely visited. Notice that the backtracking
% information is a list of pairs instead of a list of clauses)
final-resolution-step(*G, S, P, G1, Q1*),
D-equal(*Nl1, nd(zero.N, G1, P, Q1, E, S, L, partial)*),
addchild(*F, Nl, Nl1, LR, F4*),
D-root-database-and-env(*F2, P2, E2*),
D-modify-database(*F4, P2, F3*),
D-modify-environment(*F3, E2, F1*).

% Case of failure in the final unification

treat-bip(*F, N, func*(`bagof`, *Term.Goal.Bag.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, nil, E, _, L, _)*),
free-var(*nil, Goal, V, Goal1*),
complete-semantics(*P, Goal1, E, F2*),
not **untrapped-error**(*F2*),
list-of-ans-subs(*F2, L2*),
not **D-equal**(*L2, nil*),
list-of-instance(*L2, F, func(., V.Term.nil), L4*),
L-rename-except(*F, func(., V.Term.nil), L4, L1*),
variant-members(*L1, VL, LR, W*),
L-unify-members-list(*V.W, Phi*),
extract-solution-list(*VL, LT*),
L-instance(*LT, Phi, LT1*),
L-not-unifiable(*Bag, LT1*),
erasepack(*F, N, F1*).

% Case of failure: empty list of solutions

treat-bip(*F, N, func*(`bagof`, *Term.Goal.Bag.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, nil, E, _, L, _)*),
free-var(*nil, Term ^Goal, _, Goal1*),
complete-semantics(*P, Goal1, E, F2*),
not **untrapped-error**(*F2*),
list-of-ans-subs(*F2, L2*),
D-equal(*L2, nil*),
erasepack(*F, N, F1*).

% Case of error in subcomputation

treat-bip(*F, N, func*(`bagof`, *Term.Goal.Bag.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, nil, E, _, L, _)*),
free-var(*nil, Term ^Goal, _, Goal1*),
complete-semantics(*P, Goal1, E, F2*),
untrapped-error(*F2*),
treat-bip(*F, N, func*(`throw`, *untrapped-error-in-findall.nil*), *F1*).

% Other visits of node N (on backtracking):
% the list of choices LR is not empty.
% Case of success

treat-bip(*F, N, func*(`bagof`, *Term.Goal.Bag.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, LR, E, _, L, _)*),
not **D-equal**(*LR, nil*),
free-var(*nil, Term ^Goal, V, _*),
variant-members(*LR, VL, LR1, W*),
L-unify-members-list(*V.W, Phi*),
extract-solution-list(*VL, LT*),
L-instance(*LT, Phi, LT1*),
L-unify(*Bag, LT1, Mu*),
L-composition(*Phi, Mu, S*),
final-resolution-step(*G, S, P, G1, Q1*),
D-number-of-child(*N, F, N1*),
D-equal(*Nl1, nd(N1.N, G1, P, Q1, E, S, L, partial)*),
addchild(*F, Nl, Nl1, LR1, F1*).

% Case of failure in the final unification

treat-bip(*F, N, func*(`bagof`, *Term.Goal.Bag.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, LR, E, _, L, _)*),
not **D-equal**(*LR, nil*),
free-var(*nil, Term ^Goal, V, _*),
variant-members(*LR, VL, LR1, W*),
L-unify-members-list(*V.W, Phi*),
extract-solution-list(*VL, LT*),
L-instance(*LT, Phi, LT1*),
L-not-unifiable(*Bag, LT1*),
erasepack(*F, N, F1*).

Error cases:

in-error(*_, func*(`bagof`, *Term.Goal.Bag.nil), instantiation-error*)
⇐
L-var(*Goal*).

in-error(*_, func*(`bagof`, *Term.Goal.Bag.nil), type-error(callable, Goal)*) ⇐
not **L-var**(*Goal*),
free-var(*nil, Term ^Goal, _, Goal1*),
not **D-is-a-callable-term**(*Goal1*).

in-error(*_, func*(`bagof`, *Term.Goal.Bag.nil), type-error(list, Bag)*)
⇐
not **L-var**(*Bag*),
not **D-is-a-list**(*Bag*).

NOTE — References: **D-label of node** _ **in** _ **is** _ A.3.3.4, **D-equal** A.3.1, **free-var** A.4.1.65, **complete-semantics** A.4.1.42, **untrapped-error** A.4.1.46, **list-of-ans-subs** A.4.1.44, **list-of-instance** A.4.1.45, **variant-members** A.4.1.62, **L-unify-members-list** A.3.5, **extract-solution-list** A.4.1.61, **L-instance** A.3.5, **L-composition** A.3.5, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-root-database-and-env** A.3.3.4, **D-modify-database** A.3.3.6, **D-modify-environment** A.3.3.6, **L-unify** A.3.5, **L-not-unifiable** A.3.5, **erasepack** A.4.1.23, **treat-bip** A.4.1.31, **D-number-of-child** A.3.3.3, **L-var** A.3.1, **D-is-a-callable-term** A.3.1, **D-is-a-list** A.3.4, **L-rename-except** A.3.5

A.5.10.3 setof/3

```
setof(Term, Goal, List) :-
  bagof(Term, Goal, Bag),
  sorted(Bag, List).
```

sorted($Bag, List$) is a special predicate which sorts the list Bag and eliminates duplicate elements in the resulting list (see A.4.1.17).
Error cases:

in-error(*_, func*(`setof`, *Term.Goal.List.nil), instantiation-error*)
⇐
L-var(*Goal*).

in-error(*_, func*(`setof`, *Term.Goal.List.nil), type-error(callable, Goal)*) ⇐
not **L-var**(*Goal*),
free-var(*nil, Term ^Goal, _, Goal1*),
not **D-is-a-callable-term**(*Goal1*).

in-error(*_, func*(`setof`, *Term.Goal.List.nil), type-error(list, List)*)
⇐
not **L-var**(*List*),
not **D-is-a-list**(*List*).

NOTE — References: `bagof` A.5.10.2, **sorted** A.4.1.17, **L-var** A.3.1, **free-var** A.4.1.65, **D-is-a-callable-term** A.3.1, **D-is-a-list** A.3.4

A.5.11 Stream selection and control

A.5.11.1 current_input/1

execute-bip(*F, func*(`current_input`, *Fn.nil*), *S*) ⇐
D-root-database-and-env(*F, _, E*),
D-equal(*E, env(PF, IF, OF, LIF, LOF)*),
streamname(*IF, File*),
L-unify(*Fn, File, S*).

Error cases:

in-error(*_, func*(`current_input`, *Fn.nil*), type-error(stream, Fn)*)
⇐
not **L-var**(*Fn*),
not **L-stream-name**(*Fn*).

NOTE — References: **D-root-database-and-env** A.3.3.4, **D-equal** A.3.1, **streamname** A.4.1.55, **L-unify** A.3.5, **L-var** A.3.1, **L-stream-name** A.3.7

A.5.11.2 current_output/1

execute-bip(*F, func(*`current_output`*, Fn.nil), S)* ⇐
D-root-database-and-env(*F, _ E),*
D-equal(*E, env(PF, IF, OF, LIF, LOF)),*
streamname(*OF, File),*
L-unify(*Fn, func(File, nil), S).*

Error cases:

in-error(*_ func(*`current_output`*, Fn.nil), type-error(stream, Fn))* ⇐
not **L-var**(*Fn),*
not **L-stream-name**(*Fn).*

NOTE — References: **D-root-database-and-env** A.3.3.4, **D-equal** A.3.1, **streamname** A.4.1.55, **L-unify** A.3.5, **L-var** A.3.1, **L-stream-name** A.3.7

A.5.11.3 set_input/1

treat-bip(*F, N, func(*`set_input`*, Fn.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(N, G, P, _ Oldenv, S, L, _)),*
D-equal(*Oldenv, env(PF, IF, OF, LIF, LOF)),*
streamname(*IF, Fn),*
final-resolution-step(*G, empsubs, P, G1, Q),*
D-equal(*Nl1, nd(zero.N, G1, P, Q, Oldenv, empsubs, L, partial)),*
addchild(*F, Nl, Nl1, nil, F1).*

treat-bip(*F, N, func(*`set_input`*, Fn.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(N, G, P, _ Oldenv, S, L, _)),*
D-equal(*Oldenv, env(PF, IF, OF, LIF, LOF)),*
not **streamname**(*IF, Fn),*
D-delete(*LIF, stream(Fn, L1 - L2), LIF1),*
D-equal(*Newenv, env(PF, stream(Fn, L1 - L2), OF, Fn.LIF1, LOF)),*
final-resolution-step(*G, empsubs, P, G1, Q),*
D-equal(*Nl1, nd(zero.N, G1, P, Q, Oldenv, empsubs, L, partial)),*
addchild(*F, Nl, Nl1, nil, F2),*
D-modify-environment(*F2, Newenv, F1).*

Error cases:

in-error(*_ func(*`set_input`*, Fn.nil), instantiation-error)* ⇐
L-var(*Fn).*

in-error(*_ func(*`set_input`*, Fn.nil), type-error(stream_or_alias, Fn))* ⇐
not **L-var**(*Fn),*
not **L-stream-name**(*Fn).*

in-error(*F, func(*`set_input`*, Fn.nil), existence-error(stream, Fn))* ⇐
D-root-database-and-env(*F, _ Env),*
not **D-open-input**(*Fn, Env),*
not **D-open-output**(*Fn, Env).*

in-error(*F, func(*`set_input`*, Fn.nil), permission-error(stream, Fn))* ⇐
D-root-database-and-env(*F, _ Env),*
not **D-open-input**(*Fn, Env),*
D-open-output(*Fn, Env).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **streamname** A.4.1.55, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-delete** A.3.4, **D-modify-environment** A.3.3.6, **L-var** A.3.1, **L-stream-name** A.3.7, **D-root-database-and-env** A.3.3.4, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.11.4 set_output/1

treat-bip(*F, N, func(*`set_output`*, Fn.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(N, G, P, _ Oldenv, S, L, _)),*
D-equal(*Oldenv, env(PF, IF, OF, LIF, LOF)),*
streamname(*OF, Fn),*
final-resolution-step(*G, empsubs, P, G1, Q),*
D-equal(*Nl1, nd(zero.N, G1, P, Q, Oldenv, empsubs, L, partial)),*
addchild(*F, Nl, Nl1, nil, F1).*

treat-bip(*F, N, func(*`set_output`*, Fn.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(N, G, P, _ Oldenv, S, L, _)),*
D-equal(*Oldenv, env(PF, IF, OF, LIF, LOF)),*
not **streamname**(*OF, Fn),*
D-delete(*LOF, stream(Fn, L1 - L2), LOF1),*
D-equal(*Newenv, env(PF, IF, stream(Fn, L1 - L2), LIF, OF.LOF)),*
final-resolution-step(*G, empsubs, P, G1, Q),*
D-equal(*Nl1, nd(zero.N, G1, P, Q, Oldenv, empsubs, L, partial)),*
addchild(*F, Nl, Nl1, nil, F2),*
D-modify-environment(*F2, Newenv, F1).*

Error cases:

in-error(*_ func(*`set_output`*, Fn.nil), instantiation-error)* ⇐
L-var(*Fn).*

in-error(*_ func(*`set_output`*, Fn.nil), type-error(stream_or_alias, Fn))* ⇐
not **L-var**(*Fn),*
not **L-stream-name**(*Fn).*

in-error(*F, func(*`set_output`*, Fn.nil), existence-error(stream, Fn))* ⇐
D-root-database-and-env(*F, _ Env),*
not **D-open-input**(*Fn, Env),*
not **D-open-output**(*Fn, Env).*

in-error(*F, func(*`set_output`*, Fn.nil), permission-error(stream, Fn))* ⇐
D-root-database-and-env(*F, _ Env),*
not **D-open-output**(*Fn, Env),*
D-open-input(*Fn, Env).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **streamname** A.4.1.55, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-delete** A.3.4, **D-modify-environment** A.3.3.6, **L-var** A.3.1, **L-stream-name** A.3.7, **D-root-database-and-env** A.3.3.4, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.11.5 open/3

```
open( Source, Mode, Stream ) :-
  open( Source, Mode, Stream, [] ).
```

Error cases:

in-error(*_ func(*`open`*, So.M.St.nil), instantiation-error)* ⇐
L-var(*So).*

in-error(_, *func*(`open`, *So.M.St.nil*), *instantiation-error*) ⇐
L-var(*M*).

in-error(_, *func*(`open`, *So.M.St.nil*), *type-error(file_name, So)*) ⇐
not **L-var**(*So*),
not **L-stream-name**(*So*).

in-error(_, *func*(`open`, *So.M.St.nil*), *type-error(atom, M)*) ⇐
not **L-var**(*M*),
not **D-is-an-atom**(*M*).

in-error(_, *func*(`open`, *So.M.St.nil*), *type-error(var, St)*) ⇐
not **L-var**(*St*),

in-error(_, *func*(`open`, *So.M.St.nil*), *domain-error(io_mode, M)*) ⇐
D-is-an-atom(*M*),
not **D-is-an-io-mode**(*M*).

in-error(_, *func*(`open`, *So.M.St.nil*), *existence-error(file, So)*) ⇐
unspecified

NOTE — The file specified by So does not exist (physically).

in-error(_, *func*(`open`, *So.M.St.nil*), *permission-error(file, So)*) ⇐
unspecified

NOTE — The file specified by So does not allowed by the system.

in-error(_, *func*(`open`, *So.M.St.nil*), *resource-error(open_streams)*) ⇐
unspecified

NOTE — So cannot be opened, too many open streams.

NOTE — References: `open/4` A.5.11.6, **L-var** A.3.1, **D-is-an-atom** A.3.4, **L-stream-name** A.3.7, **D-is-an-io-mode** A.3.7

A.5.11.6 open/4

NOTE — No formal definition. The system and *I/O options* being not formalized.

A.5.11.7 close/1

```
close( Stream ) :-
  close( Stream, [] ).
```

Error cases:

in-error(_, *func*(`close`, *S.nil*), *instantiation-error*) ⇐
L-var(*S*).

in-error(_, *func*(`close`, *S.nil*), *instantiation-error*) ⇐
not **L-var**(*S*),
not **L-stream-name**(*S*).

in-error(_, *func*(`close`, *S.nil*), *resource-error(disk_space)*) ⇐
unspecified.

NOTE — Disk space is insufficient.

in-error(_, *func*(`close`, *S.nil*), *system-error*) ⇐
unspecified.

NOTE — Operation cannot be completed.

NOTE — References: `close/2` A.5.11.8, **L-var** A.3.1, **L-stream-name** A.3.7

A.5.11.8 close/2

NOTE — No formal definition. The system and *close options* being not formalized.

A.5.11.9 flush_output/0

```
flush_output :-
  current_output( Stream )
, flush_output( Stream ).
```

NOTE — References: `current_output` A.5.11.2, `flush_output/1` A.5.11.10

A.5.11.10 flush_output/1

NOTE — No formal definition. The system being not formalized.

A.5.11.11 stream_property/2

NOTE — No formal definition. The system being not formalized.

Error cases:

in-error(_, *func*(`stream_property`, *S.P.nil*), *instantiation-error*) ⇐
not **L-var**(*S*),
not **L-stream-name**(*S*).

in-error(_, *func*(`stream_property`, *S.P.nil*), *type-error(stream_property, P)*) ⇐
not **L-var**(*P*),
not **L-stream-property**(*P*).

NOTE — References: **L-var** A.3.1, **L-stream-name** A.3.7, **L-stream-property** A.3.7

A.5.11.12 at_end_of_stream/0

```
at_end_of_stream :-
  current_output( Stream )
, at_end_of_stream( Stream ).
```

NOTE — References: `current_output` A.5.11.2, `at_end_of_stream/1` A.5.11.13

A.5.11.13 at_end_of_stream/1

execute-bip(*F, func*(`at_end_of_stream`, *Fn.nil*), *empsubs*) ⇐
D-root-database-and-env(*F*, _, *env(PF, IF, OF, IFL, OFL)*),
D-equal(*IF, stream(Fn, L - nil)*).

execute-bip(*F, func*(`at_end_of_stream`, *Fn.nil*), *empsubs*) ⇐
D-root-database-and-env(*F*, _, *env(PF, IF, OF, IFL, OFL)*),
not **streamname**(*IF, Fn*),
D-member(*stream(Fn, L - nil), IFL*).

Error cases:

in-error(_, *func*(`at_end_of_stream`, *Fn.nil*), *instantiation-error*) ⇐
L-var(*Fn*).

in-error(_, *func*(`at_end_of_stream`, *Fn.nil), type-error(stream_or_alias, Fn))* ⇐
not **L-var**(*Fn*),
not **L-stream-name**(*Fn*).

in-error(*F, func*(`at_end_of_stream`, *Fn.nil), existence-error(stream, Fn))* ⇐
D-root-database-and-env(*F*, _, *Env*),
not **D-open-input**(*Fn, Env*),
not **D-open-output**(*Fn, Env*).

in-error(*F, func*(`at_end_of_stream`, *Fn.nil), permission-error(stream, Fn))* ⇐
D-root-database-and-env(*F*, _, *Env*),
D-open-output(*Fn, Env*),
not **D-open-input**(*Fn, Env*).

in-error(*F, func*(`at_end_of_stream`, *Fn.nil), system-error)* ⇐
unspecified.

NOTE — Unexpected read error indicated by the operating system.

NOTE — References: **D-root-database-and-env** A.3.3.4, **D-equal** A.3.1, **streamname** A.4.1.55, **D-member** A.3.4, **L-var** A.3.1, **L-stream-name** A.3.7, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.11.14 set_stream_position/2

NOTE — No formal definition.

A.5.12 Character input/output

A.5.12.1 get_char/1

`get_char/1` is a bootstrapped bip. Possible definition is:

```
get_char( Char ) :-
  current_input( Stream )
, get_char( Stream, Char ).
```

Error cases:

in-error(_, *func*(`get_char`, *Char.nil), type-error(character, Char))* ⇐
not **L-var**(*Char*),
not **D-is-a-char**(*Char*).

in-error(*F, func*(`get_char`, *Char.nil), existence-error(past_end_of_stream, Fn))* ⇐
D-root-database-and-env(*F*, _, *E*),
D-equal(*E, env(PF, IF, OF, LIF, LOF)*),
D-equal(*IF, stream(Fn, L - nil)*),
L-io-option(*Fn, eof-action, error*).

in-error(*F, func*(`get_char`, *Char.nil), system-error)* ⇐
unspecified.

NOTE — Unexpected read error indicated by the operating system.

NOTE — References: `current_input` A.5.11.1, `get_char/2` A.5.12.2, **L-var** A.3.1, **D-is-a-char** A.3.7, **D-root-database-and-env** A.3.3.4, **D-equal** A.3.1, **L-io-option** A.3.7,

A.5.12.2 get_char/2

treat-bip(*F, N, func*(`get_char`, *Fn.Char.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, _, Oldenv, S, L, _)*),
get-stream-in-env(*Oldenv, Fn, Char1, Newenv*),
L-unify(*Char, Char1, S1*),
final-resolution-step(*G, S1, P, G1, Q*),
D-equal(*Nl1, nd(zero.N, G1, P, Q, Newenv, S1, L, partial)*),
addchild(*F, Nl, Nl1, nil, F2*),
D-modify-environment(*F2, Newenv, F1*).

treat-bip(*F, N, func*(`get_char`, *Fn.Char.nil), F1)* ⇐
D-environment of node *N* **in** *F* **is** *Oldenv*,
get-stream-in-env(*Oldenv, Fn, Char1, Newenv*),
L-not-unifiable(*Char, Char1*),
erasepack(*F, N, F2*),
D-modify-environment(*F2, Newenv, F1*).

Error cases:

in-error(_, *func*(`get_char`, *Fn.Char.nil), instantiation-error)* ⇐
L-var(*Fn*).

in-error(_, *func*(`get_char`, *Fn.Char.nil), type-error(stream, Fn))* ⇐
not **L-var**(*Fn*),
not **L-stream-name**(*Fn*).

in-error(_, *func*(`get_char`, *Fn.Char.nil), type-error(character, Char))* ⇐
not **L-var**(*Char*),
not **D-is-a-char**(*Char*).

in-error(*F, func*(`get_char`, *Fn.Char.nil), existence-error(stream, Fn))* ⇐
D-root-database-and-env(*F*, _, *Env*),
not **D-open-input**(*Fn, Env*),
not **D-open-output**(*Fn, Env*).

in-error(*F, func*(`get_char`, *Fn.Char.nil), existence-error(past_end_of_stream, Fn))* ⇐
D-root-database-and-env(*F*, _, *E*),
D-equal(*E, env(PF, IF, OF, LIF, LOF)*),
D-equal(*IF, stream(Fn, L - nil)*),
L-io-option(*Fn, eof-action, error*).

in-error(*F, func*(`get_char`, *Fn.Char.nil), existence-error(past_end_of_stream, Fn))* ⇐
D-root-database-and-env(*F*, _, *E*),
D-equal(*E, env(PF, IF, OF, LIF, LOF)*),
not **streamname**(*IF, Fn*),
D-member(*stream(Fn, L - nil), LIF*),
L-io-option(*Fn, eof-action, error*).

in-error(*F, func*(`get_char`, *Fn.Char.nil), permission-error(stream, Fn))* ⇐
D-root-database-and-env(*F*, _, *Env*),
not **D-open-input**(*Fn, Env*),
D-open-output(*Fn, Env*).

in-error(*F, func*(`get_char`, *Fn.Char.nil), system-error)* ⇐
unspecified.

NOTE — Unexpected read error indicated by the operating system.

NOTE — References: **D-label of node** _ **in** _ **is** _ A.3.3.4, **D-environment of node** _ **in** _ **is** _ A.3.3.4, **D-equal** A.3.1, **get-stream-in-env** A.4.1.53, **L-unify** A.3.5, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-modify-environment** A.3.7, **L-not-unifiable** A.3.5, **erasepack** A.4.1.23, **L-var** A.3.1, **L-stream-name** A.3.7, **D-is-a-char** A.3.7, **D-root-database-and-env** A.3.3.4, **D-member** A.3.4,

streamname A.4.1.55, **L-io-option** A.3.7, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.12.3 put_char/1

```
put_char( Char ) :-
  current_output( Stream )
, put_char( Stream, Char ).
```

Error cases:

in-error(_, *func*(put_char, *Char.nil*), *instantiation-error*) ⇐
  L-var(*Char*).

in-error(_, *func*(put_char, *Char.nil*), *type-error*(*character*, *Char*)) ⇐
  not **L-var**(*Char*),
  not **D-is-a-char**(*Char*).

in-error(*F*, *func*(put_char, *Char.nil*), *system-error*) ⇐
  unspecified.

NOTE — Unexpected write error indicated by the operating system.

NOTE — References: current_output A.5.11.2, put_char/2 A.5.12.4, **L-var** A.3.1, **D-is-a-char** A.3.7,

A.5.12.4 put_char/2

treat-bip(*F*, *N*, *func*(put_char, *Fn.Char.nil*), *F1*) ⇐
  D-label of node *N* **in** *F* **is** *Nl*,
  D-equal(*Nl*, *nd*(*N*, *G*, *P*, _, *Oldenv*, *S*, *L*, _)),
  put-stream-in-env(*Oldenv*, *Fn*, *Char.nil*, *Newenv*),
  final-resolution-step(*G*, *empsubs*, *P*, *G1*, *Q*),
  D-equal(*Nl1*, *nd*(*zero.N*, *G1*, *P*, *Q*, *Newenv*, *empsubs*, *L*, *partial*)),
  addchild(*F*, *Nl*, *Nl1*, *nil*, *F2*),
  D-modify-environment(*F2*, *Newenv*, *F1*).

Error cases:

in-error(_, *func*(put_char, *Fn.Char.nil*), *instantiation-error*) ⇐
  L-var(*Fn*).

in-error(_, *func*(put_char, *Fn.Char.nil*), *instantiation-error*) ⇐
  L-var(*Char*).

in-error(_, *func*(put_char, *Fn.Char.nil*), *type-error*(*stream*, *Fn*)) ⇐
  not **L-var**(*Fn*),
  not **L-stream-name**(*Fn*).

in-error(_, *func*(put_char, *Fn.Char.nil*), *type-error*(*character*, *Char*)) ⇐
  not **L-var**(*Char*),
  not **D-is-a-char**(*Char*).

in-error(*F*, *func*(put_char, *Fn.Char.nil*), *existence-error*(*stream*, *Fn*)) ⇐
  D-root-database-and-env(*F*, _, *Env*),
  not **D-open-input**(*Fn*, *Env*),
  not **D-open-output**(*Fn*, *Env*).

in-error(*F*, *func*(put_char, *Fn.Char.nil*), *permission-error*(*stream*, *Fn*)) ⇐
  D-root-database-and-env(*F*, _, *Env*),
  not **D-open-output**(*Fn*, *Env*),
  D-open-input(*Fn*, *Env*).

in-error(*F*, *func*(put_char, *Fn.Char.nil*), *system-error*) ⇐
  unspecified.

NOTE — Unexpected write error indicated by the operating system.

in-error(*F*, *func*(put_char, *Fn.Char.nil*), *resource-error*(*disk_space*)) ⇐
  unspecified.

NOTE — No more data can be accepted.

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **put-stream-in-env** A.4.1.56, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-modify-environment** A.3.7, **L-var** A.3.1, **L-stream-name** A.3.7, **D-is-a-char** A.3.7, **D-root-database-and-env** A.3.3.4, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.12.5 nl/0

```
nl :-
  current_output( Stream )
, nl( Stream ).
```

Error cases:

in-error(*F*, *func*(nl, *nil*), *system-error*) ⇐
  unspecified.

NOTE — Unexpected write error indicated by the operating system.

in-error(*F*, *func*(*nl*, *nil*), *resource-error*(*disk_space*)) ⇐
  unspecified.

NOTE — No more data can be accepted.

NOTE — References: current_output A.5.11.2, nl/1 A.5.12.6,

A.5.12.6 nl/1

```
nl( Stream ) :-
  put_char( Stream, '\n' ).
```

Error cases:

in-error(_, *func*(nl, *Fn.nil*), *type-error*(*stream*, *Fn*)) ⇐
  not **L-var**(*Fn*),
  not **L-stream-name**(*Fn*).

in-error(*F*, *func*(nl, *Fn.nil*), *existence-error*(*stream*, *Fn*)) ⇐
  D-root-database-and-env(*F*, _, *Env*),
  not **D-open-input**(*Fn*, *Env*),
  not **D-open-output**(*Fn*, *Env*).

in-error(*F*, *func*(nl, *Fn.nil*), *permission-error*(*stream*, *Fn*)) ⇐
  D-root-database-and-env(*F*, _, *Env*),
  not **D-open-output**(*Fn*, *Env*),
  D-open-input(*Fn*, *Env*).

in-error(*F*, *func*(nl, *Fn.nil*), *system-error*) ⇐
  unspecified.

NOTE — Unexpected write error indicated by the operating system.

in-error(*F*, *func*(nl, *Fn.nil*), *resource-error*(*disk_space*)) ⇐
  unspecified.

NOTE — No more data can be accepted.

NOTE — References: put_char A.5.12.4, **L-var** A.3.1, **L-stream-name** A.3.7, **D-root-database-and-env** A.3.3.4, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.13 Character code input/output

A.5.13.1 get_code/1

```
get_code( Code ) :-
  current_input( Stream )
, get_code( Stream, Code ).
```

Error cases:

in-error(_, *func*(`get_code`, *Code.nil), type-error(character_code, Code)*) ⇐
not **L-var**(*Code*),
not **D-is-a-character-code**(*Code*).

in-error(*F, func*(`get_code`, *Code.nil), existence-error(past_end_of_stream, Fn)*) ⇐
D-root-database-and-env(*F*, _, *E*),
D-equal(*E, env(PF, IF, OF, LIF, LOF)*),
D-equal(*IF, stream(Fn, L - nil)*),
L-io-option(*Fn, eof-action, error*).

in-error(*F, func*(`get_code`, *Code.nil), system-error*) ⇐
unspecified.

NOTE — Unexpected read error indicated by the operating system.

NOTE — References: `current_input` A.5.11.1, `get_code` A.5.13.2, **L-var** A.3.1, **D-is-a-character-code** A.3.1, **D-equal** A.3.1, **D-root-database-and-env** A.3.3.4, **L-io-option** A.3.7,

A.5.13.2 get_code/2

treat-bip(*F, N, func*(`get_code`, *Fn.Code.nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, _, Oldenv, S, L, _)*),
get-stream-in-env(*Oldenv, Fn, func(Char, nil), Newenv*),
L-char-code(*Char, C*),
L-unify(*Code, func(C, nil), S1*),
final-resolution-step(*G, S1, P, G1, Q*),
D-equal(*Nl1, nd(zero.N, G1, P, Q, Newenv, S1, L, partial)*),
addchild(*F, Nl, Nl1, nil, F2*),
D-modify-environment(*F2, Newenv, F1*).

treat-bip(*F, N, func*(`get_code`, *Fn.Code.nil), F1*) ⇐
D-environment of node *N* **in** *F* **is** *Oldenv*,
get-stream-in-env(*Oldenv, Fn, func(Char, nil), Newenv*),
L-char-code(*Char, C*),
L-not-unifiable(*Code, func(C, nil)*),
erasepack(*F, N, F2*),
D-modify-environment(*F2, Newenv, F1*).

Error cases:

in-error(_, *func*(`get_code`, *Fn.Code.nil), instantiation-error*) ⇐
L-var(*Fn*).

in-error(_, *func*(`get_code`, *Fn.Code.nil), type-error(stream, Fn)*) ⇐
not **L-var**(*Fn*),
not **L-stream-name**(*Fn*).

in-error(_, *func*(`get_code`, *Fn.Code.nil), type-error(character_code, Code)*) ⇐
not **L-var**(*Code*),
not **D-is-a-character-code**(*Code*).

in-error(*F, func*(`get_code`, *Fn.Code.nil), existence-error(stream, Fn)*) ⇐
D-root-database-and-env(*F*, _, *Env*),
not **D-open-input**(*Fn, Env*),
not **D-open-output**(*Fn, Env*).

in-error(*F, func*(`get_code`, *Fn.Code.nil), existence-error(past_end_of_stream, Fn)*) ⇐
D-root-database-and-env(*F*, _, *E*),
D-equal(*E, env(PF, IF, OF, LIF, LOF)*),
D-equal(*IF, stream(Fn, L - nil)*),
L-io-option(*Fn, eof-action, error*).

in-error(*F, func*(`get_code`, *Fn.Code.nil), existence-error(past_end_of_stream, Fn)*) ⇐
D-root-database-and-env(*F*, _, *E*),
D-equal(*E, env(PF, IF, OF, LIF, LOF)*),
not **streamname**(*IF, Fn*),
D-member(*stream(Fn, L - nil), LIF*),
L-io-option(*Fn, eof-action, error*).

in-error(*F, func*(`get_code`, *Fn.Code.nil), permission-error(stream, Fn)*) ⇐
D-root-database-and-env(*F*, _, *Env*),
not **D-open-input**(*Fn, Env*),
D-open-output(*Fn, Env*).

in-error(*F, func*(`get_code`, *Fn.Code.nil), system-error*) ⇐
unspecified.

NOTE — Unexpected read error indicated by the operating system.

NOTE — References: **D-label of node** _ **in** _ **is** _ A.3.3.4, **D-environment of node** _ **in** _ **is** _ A.3.3.4, **D-equal** A.3.1, **get-stream-in-env** A.4.1.53, **L-char-code** A.3.4, **L-unify** A.3.5, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **L-not-unifiable** A.3.5, **erasepack** A.4.1.23, **D-modify-environment** A.3.7, **L-var** A.3.1, **L-stream-name** A.3.7, **D-is-a-character-code** A.3.1, **D-member** A.3.4, **streamname** A.4.1.55, **D-root-database-and-env** A.3.3.4, **L-io-option** A.3.7, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.13.3 put_code/1

```
put_code( Code ) :-
  current_output( Stream )
, put_code( Stream, Code ).
```

Error cases:

in-error(_, *func*(`put_code`, *Code.nil), instantiation-error*) ⇐
L-var(*Code*).

in-error(_, *func*(`put_code`, *Code.nil), type-error(character_code, Code)*) ⇐
not **L-var**(*Code*),
not **D-is-a-character-code**(*Code*).

in-error(*F, func*(`put_code`, *Code.nil), system-error*) ⇐
unspecified.

NOTE — Unexpected write error indicated by the operating system.

NOTE — References: `current_output` A.5.11.2, `put_code` A.5.13.4, **L-var** A.3.1, **D-is-a-character-code** A.3.1,

A.5.13.4 put_code/2

treat-bip(*F, N, func*(`put_code`, *Fn.func(Code, nil).nil), F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,

D-equal(*Nl, nd(N, G, P, _, Oldenv, S, L, _)*),
L-char-code(*Char, Code*),
put-stream-in-env(*Oldenv, Fn, func(Char, nil).nil, Newenv*),
final-resolution-step(*G, empsubs, P, G1, Q*),
D-equal(*Nl1, nd(zero.N, G1, P, Q, Newenv, empsubs, L, partial)*),
addchild(*F, Nl, Nl1, nil, F2*),
D-modify-environment(*F2, Newenv, F1*).

Error cases:

in-error(*_, func*(`put_code`, *Fn.Code.nil), instantiation-error*) ⇐
L-var(*Fn*).

in-error(*_, func*(`put_code`, *Fn.Code.nil), instantiation-error*) ⇐
L-var(*Code*).

in-error(*_, func*(`put_code`, *Fn.Code.nil), type-error(stream, Fn)*) ⇐
not **L-var**(*Fn*),
not **L-stream-name**(*Fn*).

in-error(*_, func*(`put_code`, *Fn.Code.nil), type-error(character_code, Code)*) ⇐
not **L-var**(*Code*),
not **D-is-a-character-code**(*Code*).

in-error(*F, func*(`put_code`, *Fn.Code.nil), existence-error(stream, Fn)*) ⇐
D-root-database-and-env(*F, _, Env*),
not **D-open-input**(*Fn, Env*),
not **D-open-output**(*Fn, Env*).

in-error(*F, func*(`put_code`, *Fn.Code.nil), permission-error(stream, Fn)*) ⇐
D-root-database-and-env(*F, _, Env*),
not **D-open-output**(*Fn, Env*),
D-open-input(*Fn, Env*).

in-error(*F, func*(`put_code`, *Fn.Code.nil), system-error*) ⇐
unspecified.

NOTE — Unexpected write error indicated by the operating system.

in-error(*F, func*(`put_code`, *Fn.Code.nil), resource-error(disk_space)*) ⇐
unspecified.

NOTE — No more data can be accepted.

NOTE — **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **L-char-code** A.3.4, **put-stream-in-env** A.4.1.56, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-modify-environment** A.3.7, **L-var** A.3.1, **L-stream-name** A.3.7, **D-is-a-character-code** A.3.1, **D-root-database-and-env** A.3.3.4, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.14 Term input/output

A.5.14.1 read_term/2

```
read_term( Term, Options ) :-
  current_input( Stream )
, read_term( Stream, Term, Options ).
```

NOTE — References: `current_input` A.5.11.1, `read_term/3` A.5.14.2

A.5.14.2 read_term/3

NOTE — No formal definition. The system and *read options* being not formalized.

A.5.14.3 read/1

```
read( Term ) :-
  current_input( Stream )
, read( Stream, Term ).
```

Error cases:

in-error(*F, func*(`read`, *Term.nil*), *system-error*) ⇐
unspecified.

NOTE — Unexpected read error indicated by the operating system.

NOTE — References: `current_input` A.5.11.1, `read/2` A.5.14.4

A.5.14.4 read/2

treat-bip(*F, N, func*(`read`, *Fn.Term.nil*), *F1*) ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, _, Oldenv, S, L, _)*),
get-term-stream-in-env(*Oldenv, Fn, Term1, Newenv*),
L-unify(*Term, Term1, S1*),
final-resolution-step(*G, S1, P, G1, Q*),
D-equal(*Nl1, nd(zero.N, G1, P, Q, Newenv, S1, L, partial)*),
addchild(*F, Nl, Nl1, nil, F2*),
D-modify-environment(*F2, Newenv, F1*).

treat-bip(*F, N, func*(`read`, *Fn.Term.nil*), *F1*) ⇐
D-environment of node *N* **in** *F* **is** *Oldenv*,
get-term-stream-in-env(*Oldenv, Fn, Term1, Newenv*),
L-not-unifiable(*Term, Term1*),
erasepack(*F, N, F2*),
D-modify-environment(*F2, Newenv, F1*).

Error cases:

in-error(*_, func*(`read`, *Fn.Term.nil*), *instantiation-error*) ⇐
L-var(*Fn*).

in-error(*_, func*(`read`, *Fn.Term.nil*), *type-error(stream, Fn)*) ⇐
not **L-var**(*Fn*),
not **L-stream-name**(*Fn*).

in-error(*F, func*(`read`, *Fn.Term.nil*), *existence-error(stream, Fn)*) ⇐
D-root-database-and-env(*F, _, Env*),
not **D-open-input**(*Fn, Env*),
not **D-open-output**(*Fn, Env*).

in-error(*F, func*(`read`, *Fn.Term.nil*), *permission-error(stream, Fn)*) ⇐
D-root-database-and-env(*F, _, Env*),
D-open-output(*Fn, Env*),
not **D-open-input**(*Fn, Env*).

in-error(*F, func*(`read`, *Fn.Term.nil*), *system-error*) ⇐
unspecified.

NOTE — Unexpected read error indicated by the operating system.

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-environment of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **get-term-stream-in-env** A.4.1.54, **L-unify** A.3.5, **final-resolution-step**

A.4.1.33, **addchild** A.4.1.25, **L-not-unifiable** A.3.5, **erasepack** A.4.1.23, **D-modify-environment** A.3.7, **L-var** A.3.1, **L-stream-name** A.3.7, **D-root-database-and-env** A.3.3.4, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.14.5 write_term/2

```
write_term( Term, Options ) :-
  current_output( Stream )
, write_term( Stream, Term, Options ).
```

NOTE — References: `current_output` A.5.11.2, `write_term/3` A.5.14.6

A.5.14.6 write_term/3

NOTE — No formal definition. The system and *write options* being not formalized.

A.5.14.7 write/1

```
write( Term ) :-
  current_output( Stream )
, write( Stream, Term ).
```

NOTE — References: `current_output` A.5.11.2, `write/2` A.5.14.8

A.5.14.8 write/2

treat-bip(*F, N, func(write, Fn.Term.nil), F1)* ⇐
  D-label of node *N* **in** *F* **is** *Nl,*
  D-equal(*Nl, nd(N, G, P, _, Oldenv, S, L, _)),*
  L-coding-term(*Term, Term1 - nil),*
  put-stream-in-env(*Oldenv, Fn, Term1, Newenv),*
  final-resolution-step(*G, empsubs, P, G1, Q),*
  D-equal(*Nl1, nd(zero.N, G1, P, Q, Newenv, empsubs, L, partial)),*
  addchild(*F, Nl, Nl1, nil, F2),*
  D-modify-environment(*F2, Newenv, F1).*

Error cases:

in-error(*_, func(*`write`*, Fn.Term.nil), instantiation-error)* ⇐
  L-var(*Fn).*

in-error(*_, func(*`write`*, Fn.Term.nil), type-error(stream, Fn))* ⇐
  not **L-var**(*Fn),*
  not **L-stream-name**(*Fn).*

in-error(*F, func(*`write`*, Fn.Term.nil), existence-error(stream, Fn))* ⇐
  D-root-database-and-env(*F, _, Env),*
  not **D-open-input**(*Fn, Env),*
  not **D-open-output**(*Fn, Env).*

in-error(*F, func(*`write`*, Fn.Term.nil), permission-error(stream, Fn))* ⇐
  D-root-database-and-env(*F, _, Env),*
  not **D-open-output**(*Fn, Env),*
  D-open-input(*Fn, Env).*

in-error(*F, func(*`write`*, Fn.Term.nil), system-error)* ⇐
  unspecified.

NOTE — Unexpected write error indicated by the operating system.

in-error(*F, func(*`write`*, Fn.Term.nil), resource-error(disk_space))* ⇐
  unspecified.

NOTE — No more data can be accepted.

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **L-coding-term** A.3.9, **put-stream-in-env** A.4.1.56, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-modify-environment** A.3.7, **L-var** A.3.1, **L-stream-name** A.3.7, **D-root-database-and-env** A.3.3.4, **D-open-input** A.3.9, **D-open-output** A.3.9.

A.5.14.9 writeq/1

```
writeq( Term ) :-
  current_output( Stream )
, writeq( Stream, Term ).
```

NOTE — References: `current_output` A.5.11.2, `writeq/2` A.5.14.10

A.5.14.10 writeq/2

```
writeq( Stream, Term ) :-
   write_term( Stream, Term,
      [quoted(true), numbervars(true)] ).
```

NOTE — References: `write_term/2` A.5.14.6

A.5.14.11 write_canonical/1

```
write_canonical( Term ) :-
  current_output( Stream )
, write_canonical( Stream, Term ).
```

NOTE — References: `current_output` A.5.11.2, `write_canonical/2` A.5.14.12

A.5.14.12 write_canonical/2

```
write_canonical( Stream, Term ) :-
   write_term( Stream, Term,
      [quoted(true), ignore_ops(true)] ).
```

A.5.14.13 op/3

NOTE — No formal definition. Operators being not formalized.

A.5.14.14 current_op/3

NOTE — No formal definition. Operators being not formalized.

A.5.14.15 char_conversion/2

NOTE — No formal definition. Character conversion being not formalized.

A.5.14.16 current_char_conversion/2

NOTE — No formal definition. Character conversion being not formalized.

A.5.15 Logic and control

A.5.15.1 fail_if/1

```
fail_if(X) :-
   call(X),
   !,
   fail.

fail_if(_).
```

Error cases:

in-error(_, *func*(`fail_if`, *X.nil), instantiation-error)* ⇐
L-var(*X).*

in-error(_, *func*(`fail_if`, *X.nil), type-error(callable, X))* ⇐
not **D-is-a-callable-term**(*X).*

NOTE — References: `call` A.5.1.6, `fail` A.5.1.4, **D-is-a-callable-term** A.3.1, **L-var** A.3.1

A.5.15.2 once/1

```
once(X) :-
   call(X),
   !.
```

Error cases:

in-error(_, *func*(`once`, *X.nil), instantiation-error)* ⇐
L-var(*X).*

in-error(_, *func*(`once`, *X.nil), type-error(callable, X))* ⇐
not **D-is-a-callable-term**(*X).*

NOTE — References: `call` A.5.1.6, **D-is-a-callable-term** A.3.1, **L-var** A.3.1

A.5.15.3 repeat/0

```
repeat.
repeat :-
   repeat.
```

A.5.16 Atom processing

A.5.16.1 atom_length/2

execute-bip(_, *func*(`atom_length`, *func(Atom, nil).Length.nil), S)* ⇐
L-atom-chars(*Atom, List),*
D-length-list(*List, N),*
L-unify(*Length, func(N, nil), S).*

Error cases:

in-error(_, *func*(`atom_length`, *Atom.Length.nil), instantiation-error)* ⇐
L-var(*Atom).*

in-error(_, *func*(`atom_length`, *Atom.Length.nil), type-error(atom, Atom))* ⇐
not **L-var**(*Atom),*
not **D-is-an-atom**(*Atom).*

in-error(_, *func*(`atom_length`, *Atom.Length.nil), type-error(integer, Length))* ⇐
not **L-var**(*Length),*
not **D-is-an-integer**(*Length).*

NOTE — **L-atom-chars** A.3.4, **D-length-list** A.3.4, **L-unify** A.3.5, **L-var** A.3.1, **D-is-an-atom** A.3.4 **D-is-an-integer** A.3.1

A.5.16.2 atom_concat/3

execute-bip(_, *func*(`atom_concat`, *func(Atom1, nil).func(Atom2, nil).Atom3.nil), S)* ⇐
L-atom-chars(*Atom1, List1),*
L-atom-chars(*Atom2, List2),*
D-conc(*List1, List2, L),*
L-atom-chars(*A, L),*
L-unify(*Atom3, func(A, nil), S).*

execute-bip(_, *func*(`atom_concat`, *Atom1.Atom2.func(Atom3, nil).nil), S)* ⇐
L-atom-chars(*Atom3, List3),*
D-conc(*List1, List2, List3),*
L-atom-chars(*A1, List1),*
L-atom-chars(*A2, List2),*
L-unify(*func(., Atom1.Atom2.nil), func(., func(A1, nil).func(A2, nil).nil), S).*

Error cases:

in-error(_, *func*(`atom_concat`, *Atom1.Atom2.Atom3.nil), instantiation-error)* ⇐
L-var(*Atom1),*
L-var(*Atom3).*

in-error(_, *func*(`atom_concat`, *Atom1.Atom2.Atom3.nil), instantiation-error)* ⇐
L-var(*Atom2),*
L-var(*Atom3).*

in-error(_, *func*(`atom_concat`, *Atom1.Atom2.Atom3.nil), type-error(atom, Atom1))* ⇐
not **L-var**(*Atom1),*
not **D-is-an-atom**(*Atom1).*

in-error(_, *func*(`atom_concat`, *Atom1.Atom2.Atom3.nil), type-error(atom, Atom2))* ⇐
not **L-var**(*Atom2),*
not **D-is-an-atom**(*Atom2).*

in-error(_, *func*(`atom_concat`, *Atom1.Atom2.Atom3.nil), type-error(atom, Atom3))* ⇐
not **L-var**(*Atom3),*
not **D-is-an-atom**(*Atom3).*

NOTE — References: **L-var** A.3.1, **D-is-an-atom** A.3.4, **atom_chars** A.5.16.4 **L-unify** A.3.5, **L-atom-chars** A.3.4, **D-conc** A.3.4

A.5.16.3 sub_atom/4

```
sub_atom( Atom, Start, Length, Sub_atom ) :-
  atom_concat( X,  Y,  Atom )
, atom_length( X,  N1 )
, Start is N1 + 1
, atom_concat( Sub_atom,  Z,  Y )
, atom_length( Sub_atom,  Length ).
```

Error cases:

in-error(_, *func*(`sub_atom`, *Atom.Start.Length.Sub-atom.nil), instantiation-error)* ⇐
not **D-is-an-atom**(*Atom).*

in-error(_, *func*(`sub_atom`, *Atom.Start.Length.Sub-atom.nil), type-error(atom, Sub-atom))* ⇐
not **L-var**(*Sub-atom),*
not **D-is-an-atom**(*Sub-atom).*

in-error(_, *func*(`sub_atom`, *Atom.Start.Length.Sub-atom.nil), type-error(integer, Start))* ⇐
not **L-var**(*Start),*
not **D-is-an-integer**(*Start).*

in-error(_, *func*(`sub_atom`, *Atom.Start.Length.Sub-atom.nil), type-error(integer, Length))* ⇐
not **L-var**(*Length),*
not **D-is-an-integer**(*Length).*

NOTE — References: `atom_concat` A.5.16.2, `is` A.5.6, `atom_length` A.5.16.1, **L-var** A.3.1, **D-is-an-atom** A.3.4, **D-is-an-integer** A.3.1.

A.5.16.4 atom_chars/2

execute-bip(_, *func*(`atom_chars`, *func(A, nil).List.nil), S)* ⇐
L-atom-chars(*A, List1),*
D-transform-list(*List1, List2),*
L-unify(*List, List2, S).*

execute-bip(_, *func*(`atom_chars`, *Atom.List.nil), S)* ⇐
D-transform-list(*List1, List),*
L-atom-chars(*Atom1, List1),*
L-unify(*Atom, func(Atom1, nil), S).*

Error cases:

in-error(_, *func*(`atom_chars`, *Atom.List.nil), instantiation-error)* ⇐
L-var(*Atom),*
L-var(*List).*

in-error(_, *func*(`atom_chars`, *Atom.List.nil), instantiation-error)* ⇐
L-var(*Atom),*
D-is-a-partial-char-list(*List).*

in-error(_, *func*(`atom_chars`, *Atom.List.nil), type-error(atom, Atom))* ⇐
not **L-var**(*Atom),*
not **D-is-an-atom**(*Atom).*

in-error(_, *func*(`atom_chars`, *Atom.List.nil), type-error(proper_list, List))* ⇐
not **L-var**(*List),*
not **D-char-instantiated-list**(*List).*

in-error(_, *func*(`atom_chars`, *Atom.List.nil), type-error(partial_list, List))* ⇐
D-is-an-atom(*Atom),*
not **L-var**(*List),*
not **D-char-instantiated-list**(*List),*
not **D-is-a-partial-char-list**(*List).*

NOTE — References: **L-atom-chars** A.3.4, **D-transform-list** A.3.4, **L-unify** A.3.5, **L-var** A.3.1, **D-is-an-atom** A.3.4, **D-char-instantiated-list** A.3.4, **D-is-a-partial-char-list** A.3.4

A.5.16.5 atom_codes/2

execute-bip(_, *func*(`atom_codes`, *func(A, nil).List.nil), S)* ⇐
L-atom-codes(*A, List1),*
D-transform-list(*List1, List2),*
L-unify(*List, List2, S).*

execute-bip(_, *func*(`atom_codes`, *Atom.List.nil), S)* ⇐
D-transform-list(*List1, List),*
L-atom-codes(*Atom1, List1),*
L-unify(*Atom, func(Atom1, nil), S).*

Error cases:

in-error(_, *func*(`atom_codes`, *Atom.List.nil), instantiation-error)* ⇐
L-var(*Atom),*
L-var(*List).*

in-error(_, *func*(`atom_codes`, *Atom.List.nil), instantiation-error)* ⇐
L-var(*Atom),*
D-is-a-partial-code-list(*List).*

in-error(_, *func*(`atom_codes`, *Atom.List.nil), type-error(atom, Atom))* ⇐
not **L-var**(*Atom),*
not **D-is-an-atom**(*Atom).*

in-error(_, *func*(`atom_codes`, *Atom.List.nil), type-error(partial_list, List))* ⇐
L-var(*Atom),*
not **L-var**(*List),*
not **D-code-instantiated-list**(*List).*

in-error(_, *func*(`atom_codes`, *Atom.List.nil), type-error(partial_list, List))* ⇐
D-is-an-atom(*Atom),*
not **L-var**(*List),*
not **D-code-instantiated-list**(*List),*
not **D-is-a-partial-code-list**(*List).*

NOTE — References: **L-atom-codes** A.3.4, **D-transform-list** A.3.4, **L-unify** A.3.5, **L-var** A.3.1, **D-is-an-atom** A.3.4, **D-code-instantiated-list** A.3.4, **D-is-a-partial-code-list** A.3.4

A.5.16.6 char_code/2

execute-bip(_, *func*(`char_code`, *Char.func(N, nil).nil), S)* ⇐
L-char-code(*C, N*),
L-unify(*Char, func(C, nil), S*).

execute-bip(_, *func*(`char_code`, *func(C, nil).Int.nil*), *S)* ⇐
L-char-code(*C, N),*
L-unify(*Int, func(N, nil), S).*

Error cases:

in-error(_, *func*(`char_code`, *Char.Int.nil) ,instantiation-error)* ⇐
L-var(*Char),*
L-var(*Int).*

in-error(_, *func*(`char_code`, *Char.Int.nil) ,type-error(character, Char))* ⇐
not **L-var**(*Char),*
not **D-is-a-char**(*Char).*

in-error(_, *func*(`char_code`, *Char.Int.nil)*
,type-error(character_code, Code)) ⇐
not **L-var**(*Int*),
not **D-is-a-character-code**(*Int*).

NOTE — References: **L-char-code** A.3.4, **L-unify** A.3.5, **L-var** A.3.1, **D-is-a-char** A.3.7, **D-is-a-character-code** A.3.1

A.5.16.7 number_chars/2

execute-bip(_, *func*(`number_chars`, *func(I, nil).List.nil), S)* ⇐
L-number-chars(*I, List1*),
D-transform-list(*List1, List2*),
L-unify(*List, List2, S*).

execute-bip(_, *func*(`number_chars`, *N.List.nil), S)* ⇐
D-transform-list(*List1, List*),
L-number-chars(*I, List1*),
L-unify(*N, func(I, nil), S*).

Error cases:

in-error(_, *func*(`number_chars`, *N.List.nil), instantiation-error)* ⇐
L-var(*N*),
L-var(*List*).

in-error(_, *func*(`number_chars`, *N.List.nil), instantiation-error)* ⇐
L-var(*N*),
D-is-a-partial-char-list(*List*).

in-error(_, *func*(`number_chars`, *N.List.nil), type-error(number, Atom))* ⇐
not **L-var**(*N*),
not **D-is-a-number**(*N*).

in-error(_, *func*(`number_chars`, *N.List.nil), type-error(character_list, List))* ⇐
not **L-var**(*List*),
not **D-char-instantiated-list**(*List*).

in-error(_, *func*(`number_chars`, *N.List.nil), type-error(partial_list, List))* ⇐
D-is-a-number(*N*),
not **L-var**(*List*),
not **D-char-instantiated-list**(*List*),
not **D-is-a-partial-char-list**(*List*).

NOTE — References: **L-number-chars** A.3.1, **D-transform-list** A.3.4, **L-unify** A.3.5, **L-var** A.3.1, **D-is-a-number** A.3.1, **D-char-instantiated-list** A.3.4, **D-is-a-partial-char-list** A.3.4

A.5.16.8 number_codes/2

execute-bip(_, *func*(`number_codes`, *func(I, nil).List.nil), S)* ⇐
L-number-codes(*I, List1*),
D-transform-list(*List1, List2*),
L-unify(*List, List2, S*).

execute-bip(_, *func*(`number_codes`, *N.List.nil), S)* ⇐
D-transform-list(*List1, List*),
L-number-codes(*I, List1*),
L-unify(*N, func(I, nil), S*).

Error cases:

in-error(_, *func*(`number_codes`, *N.List.nil), instantiation-error)* ⇐
L-var(*N*),
L-var(*List*).

in-error(_, *func*(`number_codes`, *N.List.nil), instantiation-error)* ⇐
L-var(*N*),
D-is-a-partial-code-list(*List*).

in-error(_, *func*(`number_codes`, *N.List.nil), type-error(number, Atom))* ⇐
not **L-var**(*N*),
not **D-is-a-number**(*N*).

in-error(_, *func*(`number_codes`, *N.List.nil), type-error(character_code_list, List))* ⇐
not **L-var**(*List*),
not **D-code-instantiated-list**(*List*).

in-error(_, *func*(`number_codes`, *N.List.nil), type-error(partial_list, List))* ⇐
D-is-a-number(*N*),
not **L-var**(*List*),
not **D-code-instantiated-list**(*List*),
not **D-is-a-partial-code-list**(*List*).

NOTE — References: **L-number-codes** A.3.1, **D-transform-list** A.3.4, **L-unify** A.3.5, **L-var** A.3.1, **D-is-a-number** A.3.1, **D-code-instantiated-list** A.3.4, **D-is-a-partial-code-list** A.3.4

A.5.17 Implementation defined hooks

A.5.17.1 set_prolog_flag/2

treat-bip(*F, N, func*(`set_prolog_flag`, *Flag.Value.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl*,
D-equal(*Nl, nd(N, G, DB, _, OldEnv, _, L, _)*),
D-equal(*OldEnv, env(PF, IF, OF, IFL, OFL)*),
D-corresponding-flag-term(*Flag, PF, C*),
D-equal(*C1, func(flag,Flag.Value._nil)*),
D-delete(*PF, C, PF1*),
D-conc(*C1.nil, PF1, PF2*),
D-equal(*NewEnv, env(PF2, IF, OF, IFL, OFL)*),
final-resolution-step(*G, empsubs, DB, G1, Q*),
D-equal(*Nl1, nd(zero.N, G1, DB, Q, OldEnv, empsubs, L, partial)*),
addchild(*F, Nl, Nl1, nil, F2*),
D-modify-environment(*F2, NewEnv, F1*).

Error cases:

in-error(_, *func*(`set_prolog_flag`, *Flag.Value.nil), instantiation-error)* ⇐
L-var(*Flag*).

in-error(_, *func*(`set_prolog_flag`, *Flag.Value.nil), type-error(atom, Flag))* ⇐
not **L-var**(*Flag*),
not **D-is-an-atom**(*Flag*).

in-error(_, *func*(`set_prolog_flag`, *Flag.Value.nil), domain-error(prolog_flag, Flag))* ⇐
not **L-var**(*Flag*),
not **D-is-a-flag**(*Flag*).

in-error(*F, func(*`set_prolog_flag`*, Flag.Value.nil), domain-error(domain, Value))* ⇐
D-is-a-flag(*Flag),*
not **D-is-a-flag-value**(*F, Flag, Value).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **D-corresponding-flag-term** A.3.7, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-delete** A.3.4, **D-modify-environment** A.3.3.6, **L-var** A.3.1, **D-is-an-atom** A.3.4, **D-is-a-flag** A.3.7, **D-is-a-flag-value** A.3.7

A.5.17.2 current_prolog_flag/2

`current_prolog_flag/2` is re-executable.

treat-bip(*F, N, func(*`current_prolog_flag`*, Flag.Value.nil), F1)* ⇐
D-label of node *N* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(N, G, DB, _, Env, _, L, _)),*
D-equal(*Env, env(PF, IF, OF, IFL, OFL)),*
corresponding-flag-and-value(*Flag, Value, PF, C, S),*
final-resolution-step(*G, S, DB, G1, Q),*
D-equal(*Nl1, nd(zero.N, G1, DB, Q, Env, empsubs, L, partial)),*
addchild(*F, Nl, Nl1, nil, F1).*

treat-bip(*F, N, func(*`current_prolog_flag`*, Flag.Value.nil), F1)* ⇐
D-environment of node *N* **in** *F* **is** *Env,*
D-equal(*Env, env(PF, IF, OF, IFL, OFL)),*
not **exist-corresponding-flag-and-value**(*Flag, Value, PF),*
erasepack(*F, N, F1).*

Error cases:

in-error(*_, func(*`current_prolog_flag`*, Flag.Value.nil), type-error(atom,Flag))* ⇐
not **L-var**(*Flag),*
not **D-is-an-atom**(*Flag).*

in-error(*_, func(*`current_prolog_flag`*, Flag.Value.nil), type-error(prolog_flag,Flag))* ⇐
not **L-var**(*Flag),*
not **D-is-a-flag**(*Flag).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-environment of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **corresponding-flag-and-value** A.4.1.68, **final-resolution-step** A.4.1.33, **addchild** A.4.1.25, **D-delete** A.3.4, **D-modify-environment** A.3.3.6, **exist-corresponding-flag-and-value** A.4.1.69, **L-var** A.3.1, **D-is-an-atom** A.3.4, **D-is-a-flag** A.3.7,

A.5.17.3 halt/0

treat-bip(*F, N, func(*`halt`*, nil), F1)* ⇐
D-label of node *nil* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(nil, _, P, _, E, _, _, _)),*
D-equal(*Nl1, nd(s(zero).nil, special-pred(halt-system-action, nil), P, nil, E, empsubs, nil, partial)),*
cut-all-choice-point(*F, N, nil, F2),*
D-label of node *nil* **in** *F2* **is** *Nl2,*
addchild(*F2, Nl2, Nl1, nil, F1).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **cut-all-choice-point** A.4.1.38, **addchild** A.4.1.25

A.5.17.4 halt/1

treat-bip(*F, N, func(*`halt`*, Int.nil), F1)* ⇐
D-label of node *nil* **in** *F* **is** *Nl,*
D-equal(*Nl, nd(nil, _, P, _, E, _, _, _)),*
D-equal(*Nl1, nd(s(zero).nil, special-pred(halt-system-action, Int.nil), P, nil, E, empsubs, nil, partial)),*
cut-all-choice-point(*F, N, nil, F2),*
D-label of node *nil* **in** *F2* **is** *Nl2,*
addchild(*F2, Nl2, Nl1, nil, F1).*

Error cases:

in-error(*_, func(*`halt`*, Int.nil), instantiation-error)* ⇐
L-var(*Int).*

in-error(*_, func(*`halt`*, Int.nil), type-error(integer, Int))* ⇐
not **L-var**(*Int),*
not **D-is-an-integer**(*Int).*

NOTE — References: **D-label of node _ in _ is _** A.3.3.4, **D-equal** A.3.1, **cut-all-choice-point** A.4.1.38, **addchild** A.4.1.25 **L-var** A.3.1, **D-is-an-integer** A.3.1,

---