Contents

Programming language Prolog

Introduction

This is the second Committee Draft of Part 1 of an International Standard for Prolog; it was produced on 18 March 1993. It replaces the draft for CD registration: ISO/IEC JTC1 SC22 WG17 N92 (= SC22 N1133, ballot results in SC22 N1205, N1210).

There is no existing International Standard for Prolog.

---

1 Scope

This draft International Standard is designed to promote the applicability and portability of Prolog text and data among a variety of data processing systems.

This draft International Standard specifies:

a) The representation of Prolog text, and

b) The syntax and constraints of the Prolog language, and

c) The semantic rules for interpreting Prolog text, and

d) The representation of input data to be processed by Prolog, and

e) The representation of output produced by Prolog, and

f) The restrictions and limits imposed on a conforming Prolog processor.

NOTE — This draft International Standard does not specify:

a) the size or complexity of Prolog text that will exceed the capacity of any specific data processing system or language processor, or the actions to be taken when the corresponding limits are exceeded;

b) the minimal requirements of a data processing system that is capable of supporting an implementation of a Prolog processor;

c) the methods of activating the Prolog processor or the set of commands used to control the environment in which Prolog text is prepared for execution and executed;

d) the mechanisms by which Prolog text is prepared for use by a data processing system;

e) the typographical representation of Prolog text published for human reading;

f) the user environment (top level loop, debugger, library system, editor, compiler etc.) of a Prolog processor.

This draft International Standard is intended for use by implementors and knowledgeable programmers, and is not a tutorial.

1.1 Notes

Notes in this draft International Standard have no effect on the language, Prolog text or Prolog processors that are defined as conforming by this draft International Standard. Reasons for including a note include:

a) Cross references to other clauses of this draft International Standard in order to help readers find their way around,

b) Warnings when a built-in predicate as defined in this draft International Standard has a different meaning in some existing implementations.

---

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this draft International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this draft International Standard are encouraged to investigate the possibility of applying the most recent editions of the

standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 646, *Information processing — ISO 7-bit coded character set for information interchange, 1983.*

ISO 2382, *Data processing — vocabulary.*

ISO 8859, *Information processing — 8-bit single-byte coded graphic character sets (for example — Part 1: Latin alphabet No. 1, 1987; Part 2: Latin alphabet No. 2, 1987; Part 5: Latin/Cyrillic alphabet, 1988; Part 6: Latin/Arabic alphabet, 1987; Part 8: Latin/Hebrew alphabet, 1988).*

ISO/IEC 9899 : 1990, *Information processing — Programming languages — C.*

ISO/IEC TR 10034 : 1990, *Guidelines for the preparation of conformity clauses in programming languages.*

ISO/IEC 10967-1, *Information processing — Programming languages — Language independent arithmetic — Part 1: Integer and floating point arithmetic. Second committee draft, August 1992.*

BS 6154, *Method of defining — Syntactic metalanguage, 1981.*

---

3 Definitions

This terminology for Prolog has a format modelled on that of ISO 2382 — Data Processing — Vocabulary.

An entry consists of a phrase (in bold type) being defined, followed by its definition. Words and phrases defined in the glossary are printed in italics when they are used in other entries. When a definition contains two terms defined in separate entries directly following each other (or separated only by a punctuation sign), an asterisk separates the terms.

Words and phrases not defined in this glossary are assumed to have the meaning given in ISO 2382; if they do not appear in ISO 2382, then they are assumed to have their usual meaning.

In this draft International Standard, the following definitions apply:

3.1 A**:** The set of *atoms* (see 6.1.2b, 7.1.4).

3.2 activation: The process of *executing* an *activator*.

3.3 activator: The result of preparing a *goal* for *execution* (see 7.7.3).

NOTE — **algorithm, Herbrand** is defined as: **Herbrand algorithm** (3.65).

3.4 alias: An *alias* is an *atom* associated with an open *stream* (see 7.10.2.2).

The standard input *stream* has the alias `user_input`, and the standard output *stream* has the alias `user_output` (see 7.10.2.3).

NOTE — A *stream* can have many aliases, but an *atom* can be the *alias* of at most one *stream*, with the exception of the *atom* `user` which identifies both the standard input and the standard output *streams*.

3.5 anonymous variable: A *variable*, represented in a *term* or *Prolog text* by _, which differs from every other *variable* (and anonymous variable) (see 6.4.3).

3.6 argument: A *term* which is associated with a *predication* or *compound term*.

3.7 arithmetic data type: A *data type* whose values are members of $\mathcal{Z}$ or $\mathcal{R}$.

3.8 arity: The number of *arguments* of a *compound term*. Syntactically, a non-negative integer associated with a *functor* or *predicate*.

3.9 assert, to: To assert a *clause* is to add it to the *user-defined procedure* in the *database* defined by the *predicate* of that *clause*.

NOTE — It is unnecessary for the *user-defined procedure* to already exist.

3.10 atom: A basic object, denoted by an *identifier*.

NOTE — **atom, null** is defined as: **null atom** (3.82).

3.11 axiom: A general rule satisfied by an operation and all values of the *data type* to which the operation belongs.

3.12 backtrack, to: To return to the *choicepoint* of the current *goal* in order to attempt to *resatisfy* it (see 7.7.8).

NOTE — **bias, exponent** is defined as: **exponent bias** (3.51).

3.13 bind, to: To *substitute* a *variable* with a different *term*.

3.14 body: A *goal*, distinguished by its context as part of a *rule* (see 3.109).

3.15 built-in predicate: A *procedure* whose *execution* is implemented by the *processor* (see 8).

3.16 byte: An integer in the range $[0..255]$ (see 7.1.2.1).

3.17 C: The set of *characters* (see 7.1.4.1).

3.18 callable term: An *atom* or a *compound term*.

3.19 CC: The set of character codes (see 7.1.2.2).

3.20 character: An element of C — an *implementation defined* character set (see 6.5, 7.1.4.1).

3.21 character-conversion relation: A *relation* on the set of *characters*, C, which specifies that, in some *Prolog text* units and *sources*, some *characters* are intended to be equivalent to other *characters*, and *converted* to those *characters* (see 7.4.2.5, 8.14.15).

3.22 choicepoint: A state during *execution* from which a *goal* can be *executed* in more than one way.

NOTE — There is a choicepoint even if all the attempts to *satisfy* the *goal* fail (see 7.7.8, 7.8.4).

3.23 class (of an *operator*): The class of an *operator* defines whether it is a prefix, infix, or postfix *operator* (see 6.3.4).

3.24 clause: A *fact* or a *rule*.

NOTE — In ISO/IEC International Standards "clause" has the meaning: one of the numbered paragraphs of a standard. In this draft International Standard, the context distinguishes the two meanings.

3.25 collating sequence: An *implementation defined* ordering defined on the set C of *characters* (see 6.6).

3.26 complete database: The set of *procedures* with respect to which *execution* is performed (see 7.5).

3.27 composition (of two *substitutions*): The mapping resulting from the application of the first substitution followed by the application of the second. Composition of the substitutions σ_1 and σ_2 is denoted $\sigma_1\sigma_2$. When the composition acts on a *term* t, it is denoted by $t\sigma_1\sigma_2$, with the meaning $((t\sigma_1)\sigma_2)$.

3.28 compound term: A *functor* of *arity* N, N positive, together with a sequence of N *arguments* (see 7.1.5).

3.29 configuration: Host and target computers, any operating system(s) and software used to operate a *processor*.

3.30 conforming processor: A *processor* which prepares *conforming Prolog text* for execution, and which obeys all the compliance clauses (see 5) for *processors* in this draft International Standard.

3.31 conforming Prolog data: A sequence of *characters* which obeys all the compliance clauses for *Prolog data* in this draft International Standard (see 5, 6.2.2).

3.32 conforming Prolog text: A sequence of *characters* which obeys all the compliance clauses for *Prolog text* in this draft International Standard (see 5, 6.2).

3.33 constant: An *atom* or a *number*.

NOTE — **construct, control** is defined as: **control construct** (3.35).

NOTE — **constructor, list** is defined as: **list constructor** (3.77).

3.34 contain, to: A *term* contains another *term* if the two are *identical terms*, or it is a *compound term*, one of whose *arguments* contains the other *term*.

3.35 control construct: A *procedure* whose definition is part of the Prolog *processor* (see 7.8).

3.36 convert (from type A to type B): A function whose signature is

$convert_{A \to B} : A \to B \cup \{\mathbf{error}\}$

which transforms a value of type A to type B. It shall be an error if the transformation cannot be made.

For example, see converting a term to a clause and vice versa (7.6), character conversion (3.21, 7.4.2.5, 8.14.15), and converting a floating point value to an integer value and vice versa (9.1.5).

3.37 CT**:** The set of *compound terms* (see 6.1.2e, 7.1.5).

3.38 cut: A *control construct* whose effect is to remove all *choicepoints* back to, and including, its *parent* (see 7.8.4).

NOTE — **data, conforming Prolog** is defined as: **conforming Prolog data** (3.31).

3.39 database: The set of *user-defined procedures* which currently exist during *execution* (see 7.5).

NOTE — **database, complete** is defined as: **complete database** (3.26).

3.40 data type: A set of values and a set of operations that manipulate those values.

NOTE — **data type, arithmetic** is defined as: **arithmetic data type** (3.7).

3.41 denormalized value: A denormalized floating point value provides less than the full precision allowed by that type (see F_D, 7.1.3).

3.42 directive: A *clause* in *Prolog text* which has *principal functor* `:-/1` (see 6.2.1.1) which affects the meaning of that *Prolog text* (see 7.4.2).

3.43 dynamic (of a ***procedure*****):** A dynamic *procedure* is one whose *clauses* can be changed during *execution*, for example by *asserting* or *retracting* $*$ *clauses* (see 7.5.2).

NOTE — **effect, side** is defined as: **side effect** (3.111).

3.44 element (of a ***list*****):** An element of a *non-empty list* is either the *head* of the *list* or an element of the *tail* of the *list*. The *empty list* has no elements.

3.45 empty list: The *atom* `[]` (nil).

3.46 error: An error is a special circumstance which causes the normal process of *execution* to be interrupted (see 7.12).

3.47 evaluable functor: The *principal functor* of a *term* which may be evaluated as an *expression* (see 7.9, 9).

3.48 evaluate: To reduce an *expression* to a *constant* (see 8.6.1, 9).

3.49 exceptional value: A non-numeric result of an *expression* (see 7.9.4).

3.50 execution (verb: to execute): The process by which a Prolog *processor* tries to *satisfy* a *goal* (see 7.7.1).

3.51 exponent bias: A number added to the exponent of a floating point number, usually to transform the exponent to an unsigned integer.

3.52 expression: A *constant* or a *compound term* which may be *evaluated* to produce a result (see 8.6.1, 9).

3.53 extension: A facility in the *processor* that is not specified in this draft International Standard but that would not cause any ambiguity or contradiction if added to this draft International Standard.

3.54 F**:** The set of floating point values (see 6.1.2d, 7.1.3).

3.55 fact: A *clause* whose *body* is the *goal* `true`.

NOTE — A fact is represented in *Prolog text* by a *term* whose *principal functor* is not `:-/2`.

3.56 fail, to: *Execution* of a *goal* fails if it is not *satisfied*.

3.57 file name: An *implementation defined* $*$ *ground term* which identifies to the *processor* a file which will be used for input/output during the *execution* of the *Prolog text*.

3.58 flag: An *atom* which is associated with an *implementation defined* or user-defined value (see 7.11).

3.59 functor: An *identifier* together with a non-zero *arity*.

3.60 functor name: The *identifier* of a *functor*.

NOTE — **function, range checking** is defined as: **range checking function** (3.99).

NOTE — **function, rounding** is defined as: **rounding function** (3.108).

NOTE — **functor, principal** is defined as: **principal functor** (3.93).

3.61 goal: A *predication* which is to be *executed* (see *body*, *query*, and 7.7.3).

3.62 ground term: A *constant* or a *compound term* whose *arguments* are all ground. A *term* is ground with respect to a *substitution* if application of the *substitution* yields a ground term.

3.63 head (of a *list*): The first *argument* of a *non-empty list*.

3.64 head (of a *rule*): A *predication*, distinguished by its context.

3.65 Herbrand algorithm: An algorithm which computes the *most general unifier* MGU of a set of equations (see 7.3.2).

3.66 I**:** The set of integers (see 6.1.2c, 7.1.2).

3.67 identical terms: Two *terms* are identical if they have the same abstract syntax (see 6.1.2).

3.68 identifier: A basic unstructured object used to denote an *atom*, *functor name* or *predicate name*.

NOTE — **identifier, stream** is defined as: **stream identifier** (3.121).

3.69 iff: If and only if.

3.70 implementation defined: An implementation defined feature is dependent on the *processor*, and is required by this draft International Standard to be defined in the accompanying *processor* documentation (see 5).

3.71 implementation dependent: An implementation dependent feature is dependent on the *processor*.

NOTE — It is not required by this draft International Standard to be defined in the accompanying *processor* documentation.

3.72 improper list: A *compound term* whose *functor* is the *list constructor* and which has two *arguments*, the second of which is neither a *variable*, nor a *list*, nor a *partial list*.

NOTE — The concept of an improper list is used in 8.5.3.

Examples: `[a | b]`, `[1, 2 | 3]`

3.73 I/O mode: An *atom* which represents an attribute of a *stream*. A *processor* shall support the *I/O modes*: `read`, `write`, `append` (see 8.11.5, 7.10.1.1).

NOTE — **indicator, predicate** is defined as: **predicate indicator** (3.90).

3.74 instance (of a *term*): The result of applying a *substitution* to the *term*.

If t is a term and σ a substitution, the instance of t by σ is denoted $t\sigma$.

3.75 instantiated: A *variable* is instantiated with respect to a *substitution* if application of the *substitution* yields a *constant* or a *compound term*.

A *term* is instantiated if any of its *variables* are instantiated.

NOTE — **level, top** is defined as: **top level** (3.129).

3.76 list: Either the *empty list* or a *non-empty list*.

NOTE — Examples: `[]`, `[a, X]`, `[1, 2, _]`, `[a | [b]]`

3.77 list constructor: The *functor* `'.'` (dot), of *arity* two, used for constructing *lists*.

NOTE — **list, empty** is defined as: **empty list** (3.45).

NOTE — **list, improper** is defined as: **improper list** (3.72).

NOTE — **list, non-empty** is defined as: **non-empty list** (3.79).

NOTE — **list, partial** is defined as: **partial list** (3.86).

NOTE — **list, read-options** is defined as: **read-options list** (3.101).

NOTE — **list, write-options** is defined as: **write-options list** (3.141).

NOTE — **mode, I/O** is defined as: **I/O mode** (3.73).

3.78 most general unifier (MGU**):** The most general unifier (MGU) of *terms* is a minimal *substitution* which acts on the *terms* to make them *identical*. Any unifier is an instance of some MGU.

NOTE — It is defined up to a renaming of the *variables*. If idempotent no *variable* of its domain appears in the resulting *terms*. An idempotent MGU can be computed by the *Herbrand algorithm* (see 7.3.2).

NOTE — **name, file** is defined as: **file name** (3.57).

NOTE — **name, functor** is defined as: **functor name** (3.60).

NOTE — **name, predicate** is defined as: **predicate name** (3.91).

3.79 non-empty list: A *compound term* whose *functor* is the *list constructor* and which has two *arguments*, the second of which is a *list*.

3.80 normalized value: A normalized floating point value provides the full precision allowed by that *type* (see 7.1.3).

3.81 NSTO: Not subject to occurs-check (see 7.3.3).

3.82 null atom: The *atom* `''`.

3.83 number: An integer or floating point number.

3.84 operator: A special *functor* or *predicate* which allows *compound terms*, or *predications* respectively, to be expressed in prefix, infix or postfix form (see 6.3.4).

NOTE — **operator, predefined** is defined as: **predefined operator** (3.88).

NOTE — **options, stream** is defined as: **stream options** (3.122).

3.85 parent: The parent of a *predication* is the *head* of the *rule* in whose *body* it occurs.

3.86 partial list: A *compound term* whose *functor* is the *list constructor* and which has two *arguments*, the second of which is either a *variable* or a partial list.

NOTE — The concept of a partial list is used in 8.5.3.

Examples: `[a | X]`, `[1, 2 | B]`

NOTE — **position, stream** is defined as: **stream position** (3.123).

3.87 precision: The number of digits in the fraction of a floating point number (see 7.1.3).

3.88 predefined operator: An *operator* which is initially provided by the *processor*.

3.89 predicate: An *identifier* together with an *arity*.

NOTE — **predicate, built-in** is defined as: **built-in predicate** (3.15).

3.90 predicate indicator: A *compound term* `A/N`, where `A` is an *atom* and `N` is an integer, denoting one particular *procedure*.

3.91 predicate name: The *identifier* of a *predicate*.

3.92 predication: A *predicate* of *arity* `N` and a sequence of `N` *arguments*.

3.93 principal functor: The principal functor of a *compound term* is `F/N` if the *functor* of the *compound term* is `F` and its *arity* is `N`.

The principal functor of a *constant* is `C/0` if the *constant* is `C`.

3.94 procedure: A *control construct*, a *built-in predicate*, or a *user-defined procedure*. A procedure is either *static* or *dynamic* (see 7.5).

NOTE — **procedure, user-defined** is defined as: **user-defined procedure** (3.137).

3.95 processor: A compiler or interpreter working in combination with a *configuration*.

NOTE — **processor, conforming** is defined as: **conforming processor** (3.30).

3.96 Prolog text: A sequence of *terms* representing *directives* and *clauses* (see 6.2, 7.4).

3.97 query: A *goal* given as interactive input to the *top level*.

NOTE — This draft International Standard does not define or require a *processor* to support the concept of *top level*.

3.98 $\mathcal{R}$**:** The set of real numbers (see 4).

3.99 range checking function: A function whose *signature* is:

$$chk_F : \mathcal{R} \times F^* \rightarrow F \cup \{\mathbf{overflow}, \mathbf{underflow}\}$$

A range checking function $chk_F(x, y)$ determines the final result of a floating point operation based on a before-rounding value x and an after-rounding value y.

NOTE — The permissible range checking functions are defined in 9.1.3.2.

3.100 read-option: A *read-option* controls the output produced by the predicate `read_term/3` (8.14.2) and all the other predicates based on it (see 7.10.3).

3.101 read-options list: A list of *read-options*.

3.102 read-term: A *term* followed by an end token. (see 6.2.2, 6.4.8).

3.103 re-execute, to: To re-execute a *goal* is to attempt to *resatisfy* it (see 7.7.6, 7.7.8).

3.104 relation: A *data type* R_T where T is a *data type* (see 4.3).

3.105 resatisfy, to: To successfully *re-execute* a *goal* having already *satisfied* it.

3.106 retract, to: To retract a *clause* is to remove it from the *user-defined procedure* in the *database* defined by the *predicate* of that *clause*.

3.107 rounding: The act of computing a representable final result (for an operation) which is close to the exact (but unrepresentable) result for that operation (see 9.1.2.1, 9.1.3.1).

3.108 rounding function: A function whose *signature* is
$rnd : \mathcal{R} \rightarrow X$ (where X is a discrete subset of $\mathcal{R}$)
which maps each element of X to itself, and is monotonic non-decreasing. Formally, if x and y are in $\mathcal{R}$,

$$x \in X \Rightarrow rnd(x) = x$$
$$x < y \Rightarrow rnd(x) \leq rnd(y)$$

NOTE — If $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects one of those adjacent values.

3.109 rule: A rule has a *head* and a *body*. During *execution*, if the *body* is true for some *substitution*, then the *head* is also true for that *substitution*. A rule is represented in *Prolog text* by a *term* whose *principal functor* is `:-/2` where the first *argument* is *converted* to the *head*, and the second *argument* is *converted* to the *body*.

3.110 satisfy, to: To satisfy a *goal* is to *execute* it successfully.

NOTE — **sequence, collating** is defined as: **collating sequence** (3.25).

3.111 side effect: A non-logical effect of an *activator* during *execution*, for example, input, output or *database* modification (see 7.7.9).

3.112 signature: A specification of an operation which defines its name, and the *type* of its argument(s) and result.

NOTE — The operation is further defined by one or more axioms.

For example, the signature:

$$add_I : I \times I \rightarrow I \cup \{\mathbf{overflow}\}$$

defines the operation add_I which takes two integer arguments ($I \times I$) and produces either a single integer result (I) or the exceptional value **overflow**.

3.113 sink: A physical object to which a *processor* outputs results, for example a file, terminal, or interprocess communication channel (see 7.10.1).

3.114 source: A physical object from which a *processor* reads data, for example a file, terminal, or interprocess communication channel (see 7.10.1).

3.115 source/sink: A *source* or a *sink*.

3.116 specifier (of an *operator*): One of the *atoms* `fx`, `fy`, `xfx`, `xfy`, `yfx`, `xf` or `yf`. A specifier denotes the *class* and associativity of an *operator* (see 6.3.4).

3.117 stack: A *data type* S_D where D is a *data type* (see 4.2).

3.118 static (of a *procedure*): A static *procedure* is one whose *clauses* shall not be changed (see 7.5.2).

3.119 STO: Subject to occurs-check (see 7.3.3).

3.120 stream: A logical view of a *source* or *sink* (see 7.10.2).

3.121 stream identifier: An *implementation dependent* $*$ *ground term* which identifies a *source/sink* inside *Prolog text* (see 7.10.2.1).

3.122 stream options: A list of zero or more *terms* which specify additional characteristics over and above those given by the *mode* of a *stream* (see 7.10.2.11).

3.123 stream position: An absolute position in a *source/sink* to which the *stream* is connected (see 7.10.2.8).

3.124 substitution: A mapping from *variables* to *terms*. By extension a substitution acts on a *term* by acting on each *variable* in the *term*.

NOTE — A substitution is represented by a Greek letter (for example Σ, σ, μ) acting as a postfix *operator*, for example:

Substitution	σ	{ X → a, Y → 3+Z }
Term	T	foo(X, Y)
New term	Tσ	foo(a, 3 + Z)

3.125 succeed, to: *Execution* of a *goal* succeeds if it is *satisfied*.

3.126 tail: The second *argument* of a *non-empty list*.

3.127 term: A *constant*, a *compound term* or a *variable* (see 7.1).

NOTE — **term, callable** is defined as: **callable term** (3.18).

NOTE — **term, compound** is defined as: **compound term** (3.28).

NOTE — **term, ground** is defined as: **ground term** (3.62).

NOTE — **terms, identical** is defined as: **identical terms** (3.67).

3.128 term-precedes: A function whose *signature* (where T is a term) is:
$term\_precedes : T \times T \rightarrow Boolean$
and which defines an order on any two *terms* (see 7.2).

NOTE — **text, conforming Prolog** is defined as: **conforming Prolog text** (3.32).

NOTE — **text, Prolog** is defined as: **Prolog text** (3.96).

3.129 top level: A process whereby a Prolog *processor* repeatedly reads and *executes* $*$ *queries*.

NOTE — This draft International Standard does not define or require a *processor* to support the concept of *top level*.

3.130 true: A *predication* is true if it is logically true.

3.131 type: The type of a *term* is a property of the term depending on its syntax and is one of: *atom*, integer, floating point, *variable* or *compound term* (see 7.1).

NOTE — **type, data** is defined as: **data type** (3.40).

3.132 undefined: A feature is undefined if this draft International Standard (a) states it is undefined, or (b) makes no requirements concerning its *execution*.

3.133 unifiable: Two or more *terms* are unifiable *iff* there exists a *unifier* for them.

3.134 unifier (of two or more *terms*): A *substitution* such that applying this *substitution* to each *term* results in *identical terms*.

NOTE — **unifier, most general** is defined as: **most general unifier** (3.78).

3.135 unify, to: To find and apply a *most general unifier* of two *terms* by successfully executing (explicitly or implicitly) the *built-in predicate* =/2 (unify) (see 8.2.1).

3.136 uninstantiated: A *variable* is *uninstantiated* when it is not *instantiated*.

3.137 user-defined procedure: A *procedure* which is defined by a sequence of *clauses* where the *head* of each *clause* has the same *principal functor*, and each *clause* is expressed by *Prolog text* or has been *asserted* during *execution* (see 8.9).

3.138 V**:** The set of *variables*, (see 6.1.2a, 7.1.1).

NOTE — **value, denormalized** is defined as: **denormalized value** (3.41).

NOTE — **value, exceptional** is defined as: **exceptional value** (3.49).

NOTE — **value, normalized** is defined as: **normalized value** (3.80).

3.139 variable: An object which may become *instantiated* to a *term* during *execution*. (see 6.1.2a, 7.1.1).

NOTE — **variable, anonymous** is defined as: **anonymous variable** (3.5).

NOTE — **variant (of a term)** is defined at (7.1.6.1).

NOTE — **witness (of a set of variables)** is defined at (7.1.1.2).

3.140 write-option: A *write-option* controls the output produced by the predicate `write_term/3` (8.14.6) and all the other predicates based on it (see 7.10.5).

3.141 write-options list: A list of *write-options*.

3.142 $\mathcal{Z}$**:** The set of mathematical integers (see 4).

---

4 Symbols and abbreviations

The following symbols and abbreviations are used in this draft International Standard.

4.1 Notation

$\mathcal{Z}$ denotes the set of mathematical integers, and $\mathcal{R}$ denotes the set of real numbers.

The following prefix and infix operators have their conventional (exact) mathematical meaning:

$+$, $-$, $*$, $/$, x^y, $log_x y$, $\sqrt{x}$, $|x|$, and $\lfloor x \rfloor$ on $\mathcal{R}$
$<$, $\leq$, $=$, $\neq$, $\geq$, and $>$ on $\mathcal{R}$
max and min on sets of $\mathcal{Z}$ and $\mathcal{R}$

The following infix operators have their conventional meaning for set definition and manipulation:

$\Rightarrow$ and $\Longleftrightarrow$ for logical implication and equivalence
$\rightarrow$ for a mapping between sets
$\times$, $\cup$, $\in$, $\subset$, and $=$ on sets
$\circ$ for composition of functions

For $x \in \mathcal{R}$, the notation $\lfloor x \rfloor$ designates the largest integer not greater than x:

$$\lfloor x \rfloor \in \mathcal{Z} \quad \text{and} \quad x - 1 < \lfloor x \rfloor \leq x$$

The type *Boolean* shall consist of the two values **true** and **false**. Mathematical predicates (like $<$ and $=$) shall produce values of type Boolean.

NOTES

1 Mathematical integers are a subset of the real numbers: $\mathcal{Z} \subset \mathcal{R}$.

2 This draft International Standard uses $*$ for multiplication, and $\times$ for the Cartesian product of sets.

4.2 Data type: stack

The following functions are specified for a stack S_D where D is a data type:

$push_D : D \times S_D \rightarrow S_D$
$top_D : S_D \rightarrow D \cup \{\mathbf{error}\}$
$pop_D : S_D \rightarrow S_D \cup \{\mathbf{error}\}$
$newstack : \rightarrow S_D$
$isempty_D : S_D \rightarrow Boolean$
$equalstack : S_D \times S_D \rightarrow Boolean$

For all values d in D, s, s' in S_D, the following axioms shall apply:

$top_D(push_D(d,s)) = d$

$top_D(newstack_D) = \mathbf{error}$

$pop_D(push_D(d,s)) = s$

$pop_D(newstack_D) = \mathbf{error}$

$isempty_D(newstack_D) = \mathbf{true}$

$isempty_D(push_D(d,s)) = \mathbf{false}$

$equalstack_D(s,s')$
$= \mathbf{true}$ if $isempty_D(s)$ and $isempty_D(s')$
$= \mathbf{false}$ if $isempty_D(s)$ and (not $isempty_D(s')$)
$= equalstack_D(pop_D(s), pop_D(s'))$
if $top_D(s) = top_D(s')$
and (not $isempty_D(s)$)
and (not $isempty_D(s')$)

NOTE — Stacks are used in the definition of executing a goal (7.7) and control constructs (7.8).

4.3 Data type: relation

The following functions are specified for a relation R_T where T is a data type:

$identity\_relation_T : \rightarrow R_T$
$relation_T : T \times T \times R_T \rightarrow R_T$
$apply\_relation_T : T \times R_T \rightarrow T$
$update\_relation_T : T \times T \times R_T \rightarrow R_T$

For all values a, a', b, b' in T, r, r' in R_T, the following axioms shall apply:

$apply\_relation_T(a, identity\_relation_T) = a$

$apply\_relation_T(a, relation_T(a', b, r))$
$= b$ if $a = a'$
$= apply\_relation_T(a, r)$ if $a \neq a'$

$update\_relation_T(a, b, identity\_relation_T)$
$= identity\_relation_T$ if $a = b$
$= relation_T(a, b, identity\_relation_T)$ if $a \neq b$

$update\_relation_T(a, b, relation_T(a', b', r))$
$= relation_T(a', b', update\_relation_T(a, b, r))$ if $a \neq a'$
$= relation_T(a, b, r)$ if $a = a'$ and $a \neq b$
$= r$ if $a = a'$ and $a = b$

NOTE — Relations are used in the description of the character-conversion relation (see 3.21).

Initially, the character-conversion relation is an $identity\_relation_C$, each call of `char_conversion/2` (8.14.15 updates this relation, and `read_term/3` (8.14.2) applies the relation on each unquoted character which is read.

---

5 Compliance

5.1 Prolog processor

A conforming Prolog processor shall:

a) Correctly prepare for execution Prolog text which conforms to both:

1) the requirements of this draft International Standard and

2) the implementation defined features of the Prolog processor, and

b) Correctly execute Prolog goals which have been prepared for execution and which conform to both:

1) the requirements of this draft International Standard and

2) the implementation defined features of the Prolog processor, and

c) Reject any term whose syntax fails to conform to the requirements of this draft International Standard, and

d) Specify all permitted variations from this draft International Standard in the manner prescribed by this draft International Standard, and

e) Offer a facility to reject the use in Prolog text of any violation of:

1) the requirements of this draft International Standard or

2) the implementation defined features of the Prolog processor.

5.2 Prolog text

Conforming Prolog text shall use only the constructs specified in this draft International Standard.

5.3 Prolog goal

A conforming Prolog goal is one whose execution is defined by this draft International Standard.

5.4 Documentation

A conforming Prolog processor shall be accompanied by documentation that includes:

a) A list of all definitions or values for the implementation defined features of the language, and

b) A list of all the features implemented by the Prolog processor which are extensions to the Prolog language defined by this draft International Standard.

---

6 Syntax

This clause defines the abstract and concrete syntaxes of a term, Prolog text and data.

Terms are the data structures manipulated at runtime by a Prolog application. Clause 6.2 defines how terms form Prolog text and data, clause 6.3 defines how tokens are combined to form terms, and clause 6.4 defines how sequences of characters form Prolog tokens.

NOTES

1 The concept of a program is different in Prolog from that in many other programming languages. The closest equivalent concept in this draft International Standard is the concept of “Prolog text”.

2 Different sequences of characters in Prolog text and data can have identical semantic meanings. The semantics is therefore based on an abstract syntax (6.1.2).

Table 1 — BS6154 syntactic metalanguage

BS6154 symbol	Meaning
Unquoted characters	Non-terminal symbol
" ... "	Terminal symbol
' ... '	Terminal symbol
(...)	Brackets
[...]	Optional symbols
{ ... }	Symbols repeated zero or more times
=	Defining symbol
;	Rule terminator
|	Alternative
,	Concatenation
(* ... *)	Comment

6.1 Notation

6.1.1 Backus Naur Form

Syntax productions are written in a tabular notation, where the first line uses the extended BNF notation standardized as BS6154 and summarized in tables 1.

The metalanguage symbols '=' '|' ',' are right-associative infix operators which bind increasingly tightly.

The remaining lines of each syntax production link different attributes of each production and express context-sensitive constraints. Each entry can be considered as a parameter of a logical grammar (i.e. a definite clause or metamorphosis grammar). Parameters apply to non-terminal and terminal symbols. In these lines, variables are written in italic type style, and constants in typewriter type style. Each attribute of the grammar is on a separate line which is identified at the start of the line.

The facets of the term grammar are:

Abstract — The abstract term (or list of abstract terms in the case of an arg list) associated with the grammar symbol.

Priority — The context-sensitive aspects of the precedence grammars on which the concrete Prolog operator notation is based.

Each term and operator is associated with a priority, i.e. an integer between 0 and 1201. A constant term and a compound term expressed in functional notation have a zero priority. A compound term expressed in operator notation (i.e. its principal functor occurs as an operator) has a priority which is equal or greater than the priority of its principal functor (see 6.3.4.1).

Specifier — The specifier of an operator (which defines its class and associativity, see table 2).

Condition — A condition which must be satisfied for the rule of the term grammar to apply.

6.1.2 Abstract term syntax

Prolog is typeless in the sense that it includes only one data type, whose elements are called terms. The enumerable set of terms is defined as the union of disjoint sets V, A, I, F, and CT where:

a) V is a set of variables such that for each form of variable token (6.4.3):

1) Every occurrence of the same named variable in a read-term corresponds to the same element of V, and

2) Every other named variable corresponds to a different element of V, and

3) Every anonymous variable corresponds to a different element of V.

b) A is a set of atoms such that for each form of atom token (6.4.3) $a \in A$ is defined by:

1) a is the two characters `[]` for the empty list.

2) a is the two characters $\{\}$ for a curly brackets.

3) a is the concatenation of the characters defined below for each form of name token (6.4.2):

Identifier token — The initial `small letter char` followed by each `alpha numeric char`.

Graphic token — Each `graphic token char`.

Quoted token — The character denoted by each `single quoted char`.

Semicolon token — The character `;`.

Cut token — The character `!`.

The characters of a non-empty atom are numbered from one upwards.

c) I is a set of integers (see 7.1.2) and $i \in I$ is defined for each form of integer token (6.4.4) as:

Integer constant — The number obtained by interpreting as a decimal number the concatenation of

the decimal digit char characters forming the integer constant.

Binary constant — The number obtained by interpreting as a binary number the concatenation of the binary digit char characters forming the binary constant.

Octal constant — The number obtained by interpreting as a octal number the concatenation of the octal digit char characters forming the octal constant.

Hexadecimal constant — The number obtained by interpreting as a hexadecimal number the concatenation of the hexadecimal digit char characters forming the hexadecimal constant.

Character code constant — The value in the collating sequence (6.6) of the character denoted by the single quoted char.

d) F is a set of floating point numbers (see 7.1.3) and $f \in F$ is defined for each float number by rounding (see 9.1.3.1) the real number defined by $(integer + fraction) * 10^{exponent}$ where:

$integer$ — The number obtained by interpreting as a decimal number the concatenation of the characters forming the integer constant of the float number token.

$fraction$ — The number obtained by interpreting as a decimal fraction the concatenation of "0." and the characters forming fraction.

$exponent$ — The number obtained by interpreting as a signed decimal number the concatenation of the characters sign and integer constant forming the exponent.

e) CT is a set where $c \in CT$ is defined for each compound term, and c is defined as $f(x_1, \ldots, x_n)$ where:

1) f is the functor name of the compound term, and

2) n is the arity of the compound term, and

3) $x_1, \ldots, x_n$ for all $n > 0$, are the arguments of the compound term.

Prolog text (6.2) is represented abstractly by an abstract list x where x is:

a) $d \cdot t$ where d is the abstract syntax for a directive, and t is Prolog text, or

b) $c \cdot t$ where c is the abstract syntax for a clause, and t is Prolog text, or

c) nil, the empty list.

6.2 Prolog text and data

Prolog text is a sequence of read-terms which denote (1) directives, and (2) clauses of predicates.

Clause 7.4 defines the correspondence between Prolog text and the complete database.

6.2.1 Prolog text

Prolog text is a sequence of directives and clauses.

	prolog text =	directive,	prolog text ;
Abstract:	$d \cdot t$	d	t

	prolog text =	clause,	prolog text ;
Abstract:	$c \cdot t$	c	t

	prolog text = ;
Abstract:	nil

6.2.1.1 Directives

	directive =	directive term, end ;
Abstract:	d	d

	directive term =	term ;
Abstract:	$:-(d)$	$:-(d)$
Priority:		1201

NOTE — Clause 7.4.2 defines the possible directives and their meaning.

6.2.1.2 Clauses

	clause =	clause term, end ;
Abstract:	c	c

	clause term =	term ;
Abstract:	c	c
Priority:		1201

Condition: The principal functor of c is not `:-/1`

NOTE — Clause 7.4.3 defines how each clause becomes part of the database.

6.2.2 Data

A Prolog read-term can be read as data by calling the predicate `read_term/3` (8.14.2).

```
        read term = term, end ;
Abstract: a       a
Priority:         1201
```

Any layout text before the term is ignored. A read-term ends with the end token.

6.3 Terms

Every Prolog term is either a constant (6.3.1), a variable (6.3.2), or a compound term (6.3.3).

6.3.1 Constants

6.3.1.1 Numbers

```
        term = integer ;
Abstract: n      n
Priority: 0
```

```
        term = float number ;
Abstract: r      r
Priority: 0
```

6.3.1.2 Negative numbers

```
        term = atom,  integer ;
Abstract: −n     −    n
Priority: 0
```

```
        term = atom,  float number ;
Abstract: −r     −    r
Priority: 0
```

The prefix operator - with a numeric constant as operand denotes the corresponding negative constant.

6.3.1.3 Atoms

```
        term = atom ;
Abstract: a      a
Priority: 0
Condition: a is not an operator
```

```
        term = atom ;
Abstract: a      a
Priority: 1201
Condition: a is an operator
```

An atom which is an operator shall not be the immediate operand of an operator. The priority of a term consisting of an operator is therefore given the priority 1201 rather than the normal 0. An atom which is an operator shall be bracketed in order to denote a term.

```
        atom = name ;
Abstract: a      a
```

```
        atom = empty list ;
Abstract: []
```

```
        atom = curly brackets ;
Abstract: {}
```

```
        empty list = open list, close list ;
```

```
        curly brackets = open curly, close curly ;
```

An atom can be a name, the empty list, or a curly brackets.

6.3.2 Variables

```
        term = variable ;
Abstract: v      v
Priority: 0
```

6.3.3 Compound terms – functional notation

Every compound term can be expressed in functional notation. When the principal functor is an operator, it can also be expressed in operator notation (6.3.4). When the principal functor is `./2` it can also be expressed as a list (6.3.5), and sometimes it can be expressed as a character code list (6.3.7). When the principal functor is `{}/1` it can also be expressed as a curly bracketed expression (6.3.6).

Functional notation is a subset of the Prolog syntax in which all compound terms can be expressed. Brackets are needed around an atom which is defined as an operator (see 6.3.4.4) when it is an operand.

A compound term written in functional notation has the form `f(A1,...,An)` where the `arguments Ai` are separated by `,` (comma). Each argument is an expression.

```
        term =  atom, open ct,  arg list, close ;
Abstract: f(l)      f               l
Priority: 0
```

```
        arg list =  exp ;
Abstract: a         a
```

```
        arg list =  exp, comma,  arg list ;
Abstract: a,l       a            l
```

6.3.3.1 Expressions

An expression (represented by `exp` in the syntax rules) occurs as the argument of a compound term or element of a list. It can be an atom which is an operator, or a term with priority less than 999. When an expression is an arbitrary term, its priority is less than the priority of the `,` (comma) operator so that there is no conflict between comma as an infix operator and comma as an argument or list element separator.

```
        exp =  atom ;
Abstract: a    a
Condition: a is an operator but not a comma
```

```
        exp =  term ;
Abstract: a    a
Priority:      999
```

NOTE — This concept of an "expression" ensures that both the terms `f(x,y)` and `f(:-, ;, [:-, :-|:-])` are syntactically valid whatever operator definitions are currently defined. Comma is special, and the following terms are syntax errors: `f(,,a)`, `[a,,|v]`, and `[a,b|,]`.

6.3.4 Compound terms – operator notation

Operator notation can be used for writing compound terms whose functor symbol is an operator. There are some predefined operators (6.3.4.4), and the programmer can define others with `op/3` (8.14.13).

An operator is defined by its name, specifier and priority.

Table 2 — Specifiers for operators

Specifier	Class	Associativity
`fx`	prefix	non-associative
`fy`	prefix	right-associative
`xfx`	infix	non-associative
`xfy`	infix	right-associative
`yfx`	infix	left-associative
`xf`	postfix	non-associative
`yf`	postfix	left-associative

Table 3 — Valid and invalid expressions

Invalid expression	Valid expression
`fx fx 1`	`fx (fx 1)`
`1 xf xf`	`(1 xf) xf`
`1 xfx 2 xfx 3`	`(1 xfx 2) xfx 3`
`1 xfx 2 xfx 3`	`1 xfx (2 xfx 3)`

The name of an operator is a `name` or a `comma`.

The priority of an operator is an integer in the range R, where

$$R = \{r, r \in \mathcal{Z} \mid 1 \leq r \leq 1200\}$$

The lower the priority the stronger binds the operator.

The specifier of an operator (defined by table 2) is a mnemonic that defines the class (prefix, infix or postfix) and the associativity (right-, left- or non-) of the operator.

An operand with the same (or smaller) priority as a right-associative operator which follows that operator need not be bracketed.

An operand with smaller priority than a left-associative operator which precedes that operator need not be bracketed.

An operand with the same priority as a left-associative operator which precedes that operator need only be bracketed if the principal functor of the operand is a right-associative operator.

An operand with the same priority as a non-associative operator must be bracketed.

The `lterm` non-terminal denotes a subset of terms, namely those allowed as the left operand of a left-associative operator with a given priority.

NOTES

Table 4 — Equivalent expressions

Unbracketed expression	Equivalent bracketed expression
`fy fy 1`	`fy (fy 1)`
`1 xfy 2 xfy 3`	`1 xfy (2 xfy 3)`
`1 xfy 2 yfx 3`	`1 xfy (2 yfx 3)`
`fy 2 yf`	`fy (2 yf)`
`1 yf yf`	`(1 yf) yf`
`1 yfx 2 yfx 3`	`(1 yfx 2) yfx 3`

1 The examples of expressions in tables 3 and 4 assume that each atom `fx`, `fy`, `xfx`, `xfy`, `yfx`, `xf` and `yf`, is an operator with the corresponding specifier.

2 Table 3 shows some invalid expressions and how they need to be bracketed to be valid.

3 Table 4 shows equivalent bracketed and unbracketed expressions. The operators `xfy` and `yfx` are assumed to have the same priority, and the operators `fy` and `yf` are also assumed to have the same priority.

6.3.4.1 Operand

An operand is a term.

	`term =`	`lterm ;`
Abstract:	a	a
Priority:	n	n

	`lterm =`	`term ;`
Abstract:	a	a
Priority:	n	$n-1$

A term with smaller priority can always occur where a term of larger priority is allowed.

	`term = open,`	`term, close ;`
Abstract:	a	a
Priority:	0	1201

	`term = open ct,`	`term, close ;`
Abstract:	a	a
Priority:	0	1201

Brackets are used to override the priority of operators.

6.3.4.2 Operators as functors

	`lterm =`	`term,`	`op,`	`term ;`
Abstract:	$f(a,b)$	a	f	b
Priority:	n	$n-1$	n	$n-1$
Specifier:			`xfx`	

	`lterm =`	`lterm,`	`op,`	`term ;`
Abstract:	$f(a,b)$	a	f	b
Priority:	n	n	n	$n-1$
Specifier:			`yfx`	

	`term =`	`term,`	`op,`	`term ;`
Abstract:	$f(a,b)$	a	f	b
Priority:	n	$n-1$	n	n
Specifier:			`xfy`	

	`lterm =`	`lterm,`	`op ;`
Abstract:	$f(a)$	a	f
Priority:	n	n	n
Specifier:			`yf`

	`lterm =`	`term,`	`op ;`
Abstract:	$f(a)$	a	f
Priority:	n	$n-1$	n
Specifier:			`xf`

	`term =`	`op,`	`term ;`
Abstract:	$f(a)$	f	a
Priority:	n	n	n
Specifier:		`fy`	
Condition:	If a is a numeric constant, f is not $-$		
Condition:	The first token of a is not `open ct`		

	`lterm =`	`op,`	`term ;`
Abstract:	$f(a)$	f	a
Priority:	n	n	$n-1$
Specifier:		`fx`	
Condition:	If a is a numeric constant, f is not `-`		
Condition:	The first token of a is not `open ct`		

NOTES

1 The last condition defines the use of `-` in the term `-(1,2)` as functor and the use in `- (1,2)` as prefix operator.

2 The `lterm` non-terminal was introduced to assign an unambiguous reading to expressions such as `fy t1 yf` where the operators have the same priority.

6.3.4.3 Operators

An operator is a name (6.4.2) or a comma (6.4.8).

```
          op =  name ;
Abstract: a     a
Priority:
Specifier:
Condition: a is a predefined operator
```

```
          op =  name ;
Abstract: a     a
Priority: n     n
Specifier: s    s
Condition: a is an operator by using op(n, s, a)
```

```
          op = comma ;
Abstract: ,
Priority: 1000
Specifier: xfy
```

There shall not be two operators with the same class and name.

There shall not be an infix and a postfix operator with the same name.

NOTES

1 comma is a solo character (6.5.3), and a token (6.4) but not an atom.

2 `','` is a predefined operator and is equivalent to comma (6.3.4.4)

3 the third argument of `op/3` (8.14.13) is any name except `','`, `'[]'`, and `'{}'`, so the precedence of the comma operator cannot be changed, and so empty lists and a curly brackets cannot be treated as operators.

4 The constraints on multiple operators allow a parser to decide immediately the specifier of an operator without too much look ahead. For example

```
  t1 yf_or_yfx   fy_or_yf t2
= t1 yf_or_yfx ( fy_or_yf t2 )

     t1 yf_or_yfx   fy_or_yf   yf
= ( ( t1 yf_or_yfx ) fy_or_yf ) yf
```

In these cases knowledge about the complete term is necessary in order to decide whether to interpret the `yf_or_yfx` as a `yf` or `yfx` operator.

6.3.4.4 Predefined operators

Table 5 defines the predefined operators which are initially recognized by a conforming processor.

Table 5 — The predefined operators

Priority	Specifier	Operator(s)
1200	xfx	:- -->
1200	fx	:- ?-
1100	xfy	;
1050	xfy	->
1000	xfy	,
700	xfx	= \=
700	xfx	== \== @< @=< @> @>=
700	xfx	=..
700	xfx	is =:= =\= < =< > >=
500	yfx	+ - /\ \/
400	yfx	* / // rem mod << >>
200	xfx	**
200	xfy	^
200	fy	- \
100	xfx	@
50	xfx	:

NOTES

1 The predicates defined as operators are: (a) Term unification, (b) Term comparison, (c) `=../2` (univ), (d) Arithmetic evaluation.

2 The control constructs defined as operators are: (a) `,/2` (conjunction), (b) `;/2` (disjunction), (c) `->/2` (if-then).

3 The evaluable functors defined as operators are: (a) binary arithmetic functors, (b) `-/1` (negation), (c) logical functors.

4 The predefined operators may be altered during execution, see `op/3` (8.14.13).

5 The meaning and use of the operators `@` and `:` is defined in Part 2 of this draft International Standard.

6.3.5 Compound terms – list notation

A list is either the empty list (6.3.1.3) or a non-empty list, which is a compound term with principal functor `./2` (dot).

```
          term = open list,  items, close list ;
Abstract: l                  l
Priority: 0
```

```
          items =  exp, comma,  items ;
Abstract: .(h, l)  h            l
```

```
          items =  exp, ht sep,  exp ;
Abstract: .(h, t)  h             t
```

	items =	exp ;
Abstract:	$.(t, [])$	t

A list is generally of the form `[E1,...,En | Tail]` where the items are separated by `,` (comma).

6.3.5.1 Examples

The following examples show terms expressed in list and functional notation.

```
[a] == .(a, []).

[a, b] == .(a, .(b, [])).

[a | b] == .(a, b).
```

6.3.6 Compound terms – curly bracket notation

A term can also be a curly brackets (6.3.1.3) or a curly bracketed expression, which is a compound term with principal functor `{}/1`.

	term = open curly,	term, close curly ;
Abstract:	$\{\}(l)$	l
Priority:	0	1201

6.3.6.1 Examples

The following examples show terms expressed in curly bracket and functional notation.

```
{a} == {}(a).

{a, b} == {}(',',(a, b)).
```

6.3.7 Compound terms – character code list notation

A character code list is either the empty list (6.3.1.3) or a non-empty list, which is a compound term with principal functor `./2` (dot).

If the `char code list token` of the `char code list` ccl contains L `double quoted chars` then the list l has L elements, and the N-th element of the list is the collating sequence integer of the N-th `double quoted char` of the `char code list token`.

	term =	char code list ;
Abstract:	l	ccl
Priority:	0	

6.4 Tokens

The syntax of Prolog terms (6.3) has been defined in terms of tokens. This clause defines how characters are combined to form tokens.

```
token (* 6.4 *)
   = name token (* 6.4.2 *)
   | variable token (* 6.4.3 *)
   | integer token (* 6.4.4 *)
   | float number token (* 6.4.5 *)
   | char code list token (* 6.4.6 *)
   | open token (* 6.4.8 *)
   | close token (* 6.4.8 *)
   | open list token (* 6.4.8 *)
   | close list token (* 6.4.8 *)
   | open curly token (* 6.4.8 *)
   | close curly token (* 6.4.8 *)
   | head tail separator token (* 6.4.8 *)
   | comma token (* 6.4.8 *)
   | end token (* 6.4.8 *) ;

name (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     name token (* 6.4.2 *) ;
variable (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     variable token (* 6.4.3 *) ;
integer (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     integer token (* 6.4.4 *) ;
float number (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     float number token (* 6.4.5 *) ;
char code list (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     char code list token (* 6.4.6 *) ;
open (* 6.4 *)
   = layout text sequence (* 6.4.1 *) ,
     open token (* 6.4.8 *) ;
open ct (* 6.4 *)
   = open token (* 6.4.8 *) ;
close (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     close token (* 6.4.8 *) ;
open list (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     open list token (* 6.4.8 *) ;
close list (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     close list token (* 6.4.8 *) ;
open curly (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     open curly token (* 6.4.8 *) ;
close curly (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     close curly token (* 6.4.8 *) ;
ht sep (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     head tail separator token (* 6.4.8 *) ;
comma (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     comma token (* 6.4.8 *) ;
end (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     end token (* 6.4.8 *) ;
```

A token shall not be followed by characters such that concatenating the characters of the token with these

characters forms a valid token as specified by the above syntax.

NOTES

1 This is the eager consumer rule: `123.e` defines the tokens `123` `.` `e`. A `layout text` is sometimes necessary to separate two tokens.

2 A quoted token begins and ends with the same quote character, and can contain that quote character only as two adjacent quote characters, for example `'ab''cd''e'`, or `"f""""g"`, or prog““.

6.4.1 Layout text

Layout text separates tokens and is also used to resolve two ambiguities. Layout text determines whether:

a) `.` (dot) is a graphic token or an end token,

b) An atom followed by an open token is a functor or an operator.

NOTE — See the definitions of compound term (6.3.3) and prefix operator (6.3.4.2).

```
layout text sequence (* 6.4.1 *)
   = layout text (* 6.4.1 *),
     { layout text (* 6.4.1 *) } ;

layout text (* 6.4.1 *)
   = layout char (* 6.5.4 *)
   | comment (* 6.4.1 *) ;
```

The comment text of a single line comment shall not contain a new line char.

The comment text of a bracketed comment shall not contain the comment close sequence.

```
comment (* 6.4.1 *)
   = single line comment (* 6.4.1 *)
   | bracketed comment (* 6.4.1 *) ;

single line comment (* 6.4.1 *)
   = end line comment char (* 6.5.3 *),
     comment text (* 6.4.1 *),
     new line char (* 6.5.4 *) ;

bracketed comment (* 6.4.1 *)
   = comment open (* 6.4.1 *),
     comment text (* 6.4.1 *),
     comment close (* 6.4.1 *) ;

comment open (* 6.4.1 *)
   = comment 1 char (* 6.4.1 *),
     comment 2 char (* 6.4.1 *) ;
comment close (* 6.4.1 *)
   = comment 2 char (* 6.4.1 *),
     comment 1 char (* 6.4.1 *) ;
comment text (* 6.4.1 *)
   = { char (* 6.5 *) } ;

comment 1 char (* 6.4.1 *) = "/" ;
comment 2 char (* 6.4.1 *) = "*" ;
```

6.4.2 Names

```
name token (* 6.4.2 *)
   = identifier token (* 6.4.2 *)
   | graphic token (* 6.4.2 *)
   | quoted token (* 6.4.2 *)
   | semicolon token (* 6.4.2 *)
   | cut token (* 6.4.2 *) ;

identifier token (* 6.4.2 *)
   = small letter char (* 6.5.2 *),
     { alpha numeric char (* 6.5.2 *) } ;
```

A graphic token shall not begin with the character sequence `comment open (* 6.4.1 *)`.

A graphic token shall not be the single character `.` (dot) when it is followed by a layout character.

```
graphic token (* 6.4.2 *)
   = graphic token char (* 6.4.2 *),
     { graphic token char (* 6.4.2 *) } ;

graphic token char (* 6.4.2 *)
   = graphic char (* 6.5.1 *)
   | backslash char (* 6.5.5 *) ;
```

A quoted token consists of the characters denoted by the sequence of `single quoted char (* 6.4.2.1 *)` appearing within the quoted token. If this character sequence forms a valid atom without quotes the quoted token shall denote that atom.

A quoted token which contains no `single quoted char`s is the null atom.

A quoted token can be spread over two or more lines by means of continuation escape sequences.

A quoted token containing one or more continuation escape sequences denotes the same atom as the quoted token obtained by removing the continuation escape sequences from the original quoted token.

```
quoted token (* 6.4.2 *)
   = single quote char (* 6.5.5 *),
     { single quoted item (* 6.4.2 *) } ,
     single quote char (* 6.5.5 *) ;

single quoted item (* 6.4.2 *)
   = single quoted char (* 6.4.2.1 *)
   | continuation escape sequence (* 6.4.2 *) ;

continuation escape sequence (* 6.4.2 *)
   = backslash char (* 6.5.5 *),
     new line char (* 6.5.4 *) ;
```

NOTE — `'abc'` and `abc` denote the same atom.

But `'\\/'` and `\\/` do not denote the same atom because `\` is used to start an escape sequence in a quoted token.

```
semicolon token (* 6.4.2 *)
   = semicolon char (* 6.5.3 *) ;
```

```
cut token (* 6.4.2 *)
   = cut char (* 6.5.3 *) ;
```

6.4.2.1 Quoted characters

```
single quoted char (* 6.4.2.1 *)
   = non quote char (* 6.4.2.1 *)
   | single quote char (* 6.5.5 *),
     single quote char (* 6.5.5 *)
   | double quote char (* 6.5.5 *)
   | back quote char (* 6.5.5 *) ;

double quoted char (* 6.4.2.1 *)
   = non quote char (* 6.4.2.1 *)
   | single quote char (* 6.5.5 *)
   | double quote char (* 6.5.5 *),
     double quote char (* 6.5.5 *)
   | back quote char (* 6.5.5 *) ;

back quoted char (* 6.4.2.1 *)
   = non quote char (* 6.4.2.1 *)
   | single quote char (* 6.5.5 *)
   | double quote char (* 6.5.5 *)
   | back quote char (* 6.5.5 *),
     back quote char (* 6.5.5 *) ;

non quote char (* 6.4.2.1 *)
   = graphic char (* 6.5.1 *)
   | alpha numeric char (* 6.5.2 *)
   | solo char (* 6.5.3 *)
   | space char (* 6.5.4 *)
   | meta escape sequence (* 6.4.2.1 *)
   | control escape sequence (* 6.4.2.1 *)
   | octal escape sequence (* 6.4.2.1 *)
   | hexadecimal escape sequence (* 6.4.2.1 *) ;
```

A quoted char can be a `single quoted char` or a `double quoted char` or a `back quoted char`.

A single quoted char which consists of two adjacent single quote chars denotes a single quote char. A double quoted char which consists of two adjacent double quote chars denotes a double quote char. A back quoted char which consists of two adjacent back quote chars denotes a back quote char.

A quoted char which consists of a graphic char, or an alpha numeric char, or a solo char, or a space char denotes that char.

A meta escape sequence denotes the escaped meta char.

```
meta escape sequence (* 6.4.2.1 *)
   = backslash char (* 6.5.5 *),
     meta char (* 6.5.5 *) ;
```

A control escape sequence denotes the control character indicated by the name of the `symbolic control char`, iff that control character is an extended character of the processor character set (6.5).

```
control escape sequence (* 6.4.2.1 *)
   = backslash char (* 6.5.5 *),
     symbolic control char (* 6.4.2.1 *) ;

symbolic control char (* 6.4.2.1 *)
   = symbolic alert char (* 6.4.2.1 *)
   | symbolic vertical tab char (* 6.4.2.1 *)
   | symbolic horizontal tab char (* 6.4.2.1 *)
   | symbolic backspace char (* 6.4.2.1 *)
   | symbolic form feed char (* 6.4.2.1 *)
   | symbolic new line char (* 6.4.2.1 *)
   | symbolic carriage return char (* 6.4.2.1 *) ;

symbolic alert char (* 6.4.2.1 *)
   = "a" ;
symbolic backspace char (* 6.4.2.1 *)
   = "b" ;
symbolic form feed char (* 6.4.2.1 *)
   = "f" ;
symbolic new line char (* 6.4.2.1 *)
   = "n" ;
symbolic carriage return char (* 6.4.2.1 *)
   = "r" ;
symbolic horizontal tab char (* 6.4.2.1 *)
   = "t" ;
symbolic vertical tab char (* 6.4.2.1 *)
   = "v" ;
symbolic hexadecimal char (* 6.4.2.1 *)
   = "x" ;
```

An octal or hexadecimal escape sequence denotes the character from the processor character set (6.5) whose value according to the collating sequence (6.6) is equal to the value denoted by the octal or hexadecimal constant.

```
octal escape sequence (* 6.4.2.1 *)
   = backslash char (* 6.5.5 *),
     octal digit char (* 6.5.2 *),
     { octal digit char (* 6.5.2 *) } ,
     backslash char (* 6.5.5 *) ;

hexadecimal escape sequence (* 6.4.2.1 *)
   = backslash char (* 6.5.5 *),
     symbolic hexadecimal char (* 6.4.2.1 *),
     hexadecimal digit char (* 6.5.2 *),
     { hexadecimal digit char (* 6.5.2 *) } ,
     backslash char (* 6.5.5 *) ;
```

NOTES

1 `\` cannot be followed by a space in a quoted token, and a `new line char` occurs in a quoted token only as part of a continuation escape sequence (6.4.2), so an atom `'a\`
`b'` does not conform to this syntax unless `\` is followed by a `new line char` in which case the atom is equivalent to the atoms `'ab'` and `ab`.

2 The representations of the symbolic control characters are those recommended by the International Standard for C (IS 9899).

3 A back quoted string (6.4.7) contains back quoted characters, but this draft International Standard does not define a token (or term) based on a back quoted string.

6.4.3 Variables

```
variable token (* 6.4.3 *)
   = anonymous variable (* 6.4.3 *)
```

```
   | named variable (* 6.4.3 *) ;

anonymous variable (* 6.4.3 *)
   = variable indicator char (* 6.4.3 *) ;

named variable (* 6.4.3 *)
   = variable indicator char (* 6.4.3 *),
     alpha numeric char (* 6.5.2 *),
     { alpha numeric char (* 6.5.2 *) }
   | capital letter char (* 6.5.2 *),
     { alpha numeric char (* 6.5.2 *) } ;

variable indicator char (* 6.4.3 *)
   = underscore char (* 6.5.2 *) ;
```

6.4.4 Integer numbers

```
integer token (* 6.4.4 *)
   = integer constant (* 6.4.4 *)
   | character code constant (* 6.4.4 *)
   | binary constant (* 6.4.4 *) ;
   | octal constant (* 6.4.4 *) ;
   | hexadecimal constant (* 6.4.4 *) ;

integer constant (* 6.4.4 *)
   = decimal digit char (* 6.5.2 *),
     { decimal digit char (* 6.5.2 *) } ;

character code constant (* 6.4.4 *)
   = "0", single quote char (* 6.5.5 *),
     single quoted char (* 6.4.2.1 *) ;

binary constant (* 6.4.4 *)
   = binary constant indicator (* 6.4.4 *),
     binary digit char (* 6.5.2 *),
     { binary digit char (* 6.5.2 *) } ;

binary constant indicator (* 6.4.4 *)
   = "0b" ;

octal constant (* 6.4.4 *)
   = octal constant indicator (* 6.4.4 *),
     octal digit char (* 6.5.2 *),
     { octal digit char (* 6.5.2 *) } ;

octal constant indicator (* 6.4.4 *)
   = "0o" ;

hexadecimal constant (* 6.4.4 *)
   = hexadecimal constant indicator (* 6.4.4 *),
     hexadecimal digit char (* 6.5.2 *),
     { hexadecimal digit char (* 6.5.2 *) } ;

hexadecimal constant indicator (* 6.4.4 *)
   = "0x" ;
```

An integer constant is unsigned. Negative integers are defined by the term syntax (6.3.1.2).

A character code constant denotes the value of the character according to the collating sequence (6.6).

6.4.5 Floating point numbers

```
float number token (* 6.4.5 *)
   = integer constant (* 6.4.4 *),
     fraction (* 6.4.5 *),
     [ exponent (* 6.4.5 *) ] ;

fraction (* 6.4.5 *)
   = decimal point char (* 6.4.5 *),
     decimal digit char (* 6.5.2 *),
     { decimal digit char (* 6.5.2 *) } ;

exponent (* 6.4.5 *)
   = exponent char (* 6.4.5 *),
     sign (* 6.4.5 *),
     integer constant (* 6.4.4 *) ;

sign (* 6.4.5 *)
   = negative sign char (* 6.4.5 *)
   | [ positive sign char (* 6.4.5 *) ] ;

positive sign char (* 6.4.5 *) = "+" ;
negative sign char (* 6.4.5 *) = "-" ;
decimal point char (* 6.4.5 *) = "." ;
exponent char (* 6.4.5 *) = "e" | "E" ;
```

A floating-point constant is unsigned. Negative floating-point constants are defined by the term syntax (6.3.1.2).

6.4.6 Character code lists

A char code list token denotes the list (6.3.5) of collating sequence integers (6.6) for a sequence of double quoted chars appearing within the char code list token.

A char code list token can be spread over two or more lines by means of continuation escape sequences.

A char code list token containing one or more continuation escape sequences denotes the same list as the char code list token obtained by removing the continuation escape sequences from the original char code list token.

```
char code list token (* 6.4.6 *)
   = double quote char (* 6.5.5 *),
     { double quoted item (* 6.4.6 *) } ,
     double quote char (* 6.5.5 *) ;

double quoted item (* 6.4.6 *)
   = double quoted char (* 6.4.2.1 *)
   | continuation escape sequence (* 6.4.2 *) ;
```

6.4.7 Back quoted strings

A back quoted string is a sequence of back quote chars appearing within the back quoted string.

A back quoted string can be spread over two or more lines by means of continuation escape sequences.

A back quoted string BS containing one or more continuation escape sequences shall be equivalent to back quoted string which would be obtained by removing the continuation escape sequences from BS.

```
back quoted string (* 6.4.7 *)
```

```
   = back quote char (* 6.5.5 *),
     { back quoted item (* 6.4.7 *) } ,
     back quote char (* 6.5.5 *) ;

back quoted item (* 6.4.7 *)
   = back quoted char (* 6.4.2.1 *)
   | continuation escape sequence (* 6.4.2 *) ;
```

NOTE — This draft International Standard does not define a token (or term) based on a back quoted string.

It would be a valid extension of this draft International Standard to define a `back quoted string` as denoting a character string constant.

6.4.8 Other tokens

```
open token (* 6.4.8 *)
   = open char (* 6.5.3 *) ;
close token (* 6.4.8 *)
   = close char (* 6.5.3 *) ;
open list token (* 6.4.8 *)
   = open list char (* 6.5.3 *) ;
close list token (* 6.4.8 *)
   = close list char (* 6.5.3 *) ;
open curly token (* 6.4.8 *)
   = open curly char (* 6.5.3 *) ;
close curly token (* 6.4.8 *)
   = close curly char (* 6.5.3 *) ;
head tail separator token (* 6.4.8 *)
   = head tail separator char (* 6.5.3 *) ;
comma token (* 6.4.8 *)
   = comma char (* 6.5.3 *) ;

end token (* 6.4.8 *)
   = end char (* 6.4.8 *) ;

end char (* 6.4.8 *) = "." ;
```

An end char shall not be followed by any character other than layout text, that is a layout character or a `%`.

NOTES

1 A `,` (comma) has three different meanings, depending on the context where it appears: it can separate arguments of a compound term (6.3.3), it can separate elements of a list (6.3.5), or can be an operator (6.3.4.2).

2 A read-term is terminated by `.` (end char).

3 The eager consumer rule applies to the parsing of an end token. An `end char` is not an end token if it could be one character of a `graphic token` (6.4.2).

6.5 Processor character set

The processor character set PCS is an implementation defined character set. The elements of PCS shall include each character defined by `char (* 6.5 *)`.

PCS may include additional elements, known as extended characters. It shall be implementation defined for each extended character whether it is a graphic char, or an alpha numeric char, or a solo char, or a layout char, or a meta char.

```
char (* 6.5 *)
   = graphic char (* 6.5.1 *)
   | alpha numeric char  (* 6.5.2 *)
   | solo char (* 6.5.3 *)
   | layout char (* 6.5.4 *)
   | meta char (* 6.5.5 *) ;
```

NOTES

1 Prolog text and data read from text streams consist of a sequence of characters taken from PCS.

2 Examples of extended characters are single-octet characters such as G1 graphic characters in IS8859/1, or multi-octet characters such as Chinese, Japanese, or Korean characters. Examples of extended small letter char (6.5.2) are small letters with grave or acute accent and Japanese Kanji characters. Examples of extended capital letter char (6.5.2) are capital letters with grave or acute accent.

6.5.1 Graphic characters

```
graphic char (* 6.5.1 *)
   = "#" | "$" | "&" | "*" | "+" | "-" | "."
   | "/" | ":" | "<" | "=" | ">" | "?" | "@"
   | "^" | "~" ;
```

A graphic character denotes itself in a quoted character.

6.5.2 Alphanumeric characters

```
alpha numeric char (* 6.5.2 *)
   = alpha char (* 6.5.2 *)
   | decimal digit char (* 6.5.2 *) ;

alpha char (* 6.5.2 *)
   = underscore char (* 6.5.2 *)
   | letter char (* 6.5.2 *) ;

letter char (* 6.5.2 *)
   = capital letter char (* 6.5.2 *)
   | small letter char (* 6.5.2 *) ;

small letter char (* 6.5.2 *)
   = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h"
   | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p"
   | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
   | "y" | "z" ;
capital letter char (* 6.5.2 *)
   = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
   | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P"
   | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X"
   | "Y" | "Z" ;
decimal digit char (* 6.5.2 *)
   = "0" | "1" | "2" | "3" | "4"
   | "5" | "6" | "7" | "8" | "9" ;
binary digit char (* 6.5.2 *)
   = "0" | "1" ;
octal digit char (* 6.5.2 *)
```

```
   = "0" | "1" | "2" | "3" | "4"
   | "5" | "6" | "7" ;
hexadecimal digit char (* 6.5.2 *)
   = "0" | "1" | "2" | "3" | "4"
   | "5" | "6" | "7" | "8" | "9"
   | ("A" | "a") | ("B" | "b") | ("C" | "c")
   | ("D" | "d") | ("E" | "e") | ("F" | "f") ;
underscore char (* 6.5.2 *) = "_" ;
```

An alpha numeric character denotes itself in a quoted character.

NOTE — The alpha numeric characters can be concatenated to form:

a) an atom when they follow a small letter char, or

b) a variable when they follow an underscore char or a capital letter char.

Two such atoms and variables that are adjacent must be separated by a layout character or comment.

6.5.3 Solo characters

```
solo char (* 6.5.3 *)
   = cut char (* 6.5.3 *)
   | open char (* 6.5.3 *)
   | close char (* 6.5.3 *)
   | comma char (* 6.5.3 *)
   | semicolon char (* 6.5.3 *)
   | open list char (* 6.5.3 *)
   | close list char (* 6.5.3 *)
   | open curly char (* 6.5.3 *)
   | close curly char (* 6.5.3 *)
   | head tail separator char (* 6.5.3 *)
   | end line comment char (* 6.5.3 *) ;

cut char (* 6.5.3 *) = "!" ;
open char (* 6.5.3 *) = "(" ;
close char (* 6.5.3 *) = ")" ;
comma char (* 6.5.3 *) = "," ;
semicolon char (* 6.5.3 *) = ";" ;
open list char (* 6.5.3 *) = "[" ;
close list char (* 6.5.3 *) = "]" ;
open curly char (* 6.5.3 *) = "{" ;
close curly char (* 6.5.3 *) = "}" ;
head tail separator char (* 6.5.3 *) = "|" ;
end line comment char (* 6.5.3 *) = "%" ;
```

A solo character denotes itself in a quoted character.

NOTE — An unquoted solo character is a single character token except that `%` and the remaining characters on the line are a comment that has no significance in Prolog text or a Prolog read-term.

A solo character need not be separated from the previous and following tokens by a layout character or comment.

6.5.4 Layout characters

```
layout char (* 6.5.4 *)
   = space char (* 6.5.4 *)
   | new line char (* 6.5.4 *) ;
space char (* 6.5.4 *) = " " ;
new line char (* 6.5.4 *)
    = implementation dependent ;
```

A space char denotes itself in a quoted character.

NOTE — An unquoted layout character is sometimes necessary to separate tokens, but is not itself a token or part of a token.

A new line char is not allowed in a quoted character.

6.5.5 Meta characters

```
meta char (* 6.5.5 *)
   = backslash char (* 6.5.5 *)
   | single quote char (* 6.5.5 *)
   | double quote char (* 6.5.5 *)
   | back quote char (* 6.5.5 *) ;

backslash char (* 6.5.5 *) = "\" ;
single quote char (* 6.5.5 *) = "'" ;
double quote char (* 6.5.5 *) = '"' ;
back quote char (* 6.5.5 *) = "`" ;
```

NOTE — A meta character modifies the meaning of the following characters, for example:

a) A backslash character starts an escape sequence in a quoted token, a char code list token, and a character code constant; but in a graphic token, it behaves like a `graphic char (* 6.5.1 *)` (see 6.4.2).

b) A single quote char is used to indicate the start and end of a quoted token (see 6.4.2).

c) A double quote char is used to indicate the start and end of a char code list token (see 6.4.6).

d) A back quote char is used to indicate the start and end of a back quoted string.

6.6 Collating sequence

The collating sequence is defined implicitly by associating a unique collating sequence integer with each character.

The collating sequence integer for an unquoted character (6.5) is implementation defined subject to the restrictions:

a) The collating sequence integers for each `capital letter char` from `A` to `Z` shall be monotonically increasing.

b) The collating sequence integers for each `small letter char` from `a` to `z` shall be monotonically increasing.

c) The collating sequence integers for each `decimal digit char` from `0` to `9` shall be monotonically increasing and contiguous.

The collating sequence integer for a quoted character (6.4.2.1) which is not a `control escape sequence` or an `octal escape sequence` or a `hexadecimal escape sequence` is the collating sequence integer for the unquoted character that the quoted character denotes.

The collating sequence integer for a quoted character which is a `control escape sequence` is implementation defined.

The collating sequence integer for a quoted character which is an `octal escape sequence` is the value of the octal characters interpreted as an octal integer.

The collating sequence integer for a quoted character which is a `hexadecimal escape sequence` is the value of the hexadecimal characters interpreted as a hexadecimal integer.

The collating sequence integer for each extended character shall also be implementation defined.

---

7 Language concepts and semantics

This clause defines the semantic concepts of Prolog:

a) Clause 7.1 defines a type to be associated with each term,

b) Clause 7.2 defines an ordering for any two terms,

c) Clause 7.3 defines unification in Prolog,

d) Clause 7.4 defines the meaning of Prolog text,

e) Clause 7.5 defines the database,

f) Clause 7.6 defines the process of converting terms to goals, and vice versa,

g) Clause 7.7 defines the execution of a goal,

h) Clause 7.8 defines the control constructs of Prolog,

i) Clause 7.9 defines the evaluation of a Prolog term as an expression.

j) Clause 7.10 defines input/output concepts,

k) Clause 7.11 defines flags,

l) Clause 7.12 defines errors.

7.1 Types

The type of any term is determined by its abstract syntax (6.1.2).

Every term has one of the following mutually-exclusive types: V (variables), I (integers), F (floating point values), A (atoms), CT (compound terms).

A term with type I, F, or A is a constant.

Built-in predicates which test explicitly the type of a term are defined in 8.3.

NOTE — Prolog is not a typed language, and any argument of a predicate can be any term whatsoever. Nonetheless, some predicates can be satisfied only when the arguments possess particular properties, and some evaluable functors are defined only when the operands possess some particular property. Note also that although the control constructs, built-in predicates and evaluable functors are defined for all arguments and operands, it is often an error unless an argument has a particular sort of value.

It is therefore convenient when defining the language to classify any term as belonging to one of several disjoint types.

7.1.1 Variable

A variable is an element of a set V (see 6.1.2a). While a goal is being executed, unification may cause a variable to become unified with another term.

NOTE — The syntax of a variable is defined in 6.3.2 and 6.4.3.

7.1.1.1 Variable set of a term

The variable set, S_V, of a term `T` is a set of variables defined recursively as:

a) If `T` is a constant then S_V is the empty set,

b) Else if `T` is a variable then S_V is the set { `T` },

c) Else if `T` is a compound term then S_V is the union of the variable sets for each of the arguments of `T`.

NOTE — For example, { `X, Y` } is the variable set of each of the terms `f(X,Y)`, `f(Y,X)`, `X+Y`, and `Y-X-X`.

7.1.1.2 Witness of a variable set

A witness of a set of variables is a term in which each of those variables occurs exactly once.

NOTES

1 For example, `f(X,Y)`, `f(Y,X)`, `X+Y`, `Y-1-X` are all witnesses of the variable set { `X`, `Y` }.

2 The concept of a witness is required when defining `bagof/3` (8.10.2) and `setof/3` (8.10.3).

7.1.1.3 Existential variables set of a term

The existential variables set, EV, of a term `T` is a set of variables defined recursively as follows:

a) If `T` is a variable or a constant then EV is the empty set,

b) Else if `T` unifies with `^(V, G)` then EV is the union of the variable set (7.1.1.1) of `V` and the existential variables set of the term `G`,

c) Else EV is the empty set.

NOTE — For example, { `X`, `Y` } is the existential variables set of each of the terms `X^Y^f(X,Y,Z)`, `(X,Y)^f(Z,Y,X)`, and `(X+Y)^3`.

7.1.1.4 Free variables set of a term

The free variables set, FV, of a term `T` with respect to a term `V` is a set of variables defined as the set difference of the variable set (7.1.1.1) of `T` and BV where BV is a set of variables defined as the union of the variable set of `V` and the existential variables set (7.1.1.3) of `T`.

NOTES

1 For example, { `X`, `Y` } is the free variables set of `X+Y+Z` with respect to `f(Z)`, and also of `Z^(A+X+Y+Z)` with respect to `A`.

2 The concept of a free variables set is required when defining `bagof/3` (8.10.2) and `setof/3` (8.10.3).

7.1.2 Integer

An integer is an element of a set I (see 6.1.2c) where I is a subset of $\mathcal{Z}$ characterized by one or three parameters. The first parameter is

$bounded \in Boolean$ (whether the set I is finite)

If $bounded$ is **false**, it is the only parameter. In this case,

$I = \mathcal{Z}$

If $bounded$ is **true**, the other two parameters are

$minint \in \mathcal{Z}$ (the smallest integer in I)
$maxint \in \mathcal{Z}$ (the largest integer in I)

$minint$ and $maxint$ shall satisfy:

$maxint > 0$

and one of: $minint = -(maxint)$
$minint = -(maxint + 1)$

Given specific values for $maxint$ and $minint$,

$I = \{x \in \mathcal{Z} \mid minint \leq x \leq maxint\}$

NOTES

1 LCAS also allows $minint = 0$ for an integer type I, but this is not allowed for the integer type provided by a Prolog processor because some of the floating point operations (for example, $exponent_F$ and $trunc_F$) may have a negative value as operand or result.

2 When $bounded$ is **false**, operations with an integer result will not have a result **overflow**, but might produce a resource error (7.12, 7.12.2h) because of exhaustion of resources.

3 During execution the values of the parameters $bounded$, $minint$, and $maxint$ are values associated with various flags (see 7.11.1).

4 A processor may provide as an extension more than one integer type. Each integer type shall have a distinct set of the operations described in 9.1.2.

5 The syntax of an integer number is defined in 6.3.1.1, 6.3.1.2, and 6.4.4.

7.1.2.1 Bytes

B, a set of bytes, is a subset of I where:

$B = \{i \in I \mid 0 \leq i \leq 255\}$

7.1.2.2 Character codes

CC, a set of character codes, is a subset of I where:

$CC = \{i \in I \mid \exists c \in C \cdot i = character\_code(c)\}$

where $character\_code(c)$ is a function giving the collating sequence integer (6.6) for a character c (7.1.4.1) of the processor character set (6.5).

The mapping between a character code and a sequence of bytes shall be implementation defined.

NOTE — A character code may correspond to more than one byte in a stream. Thus, reading a single character may consume several bytes from an input stream, and writing a single character may output several bytes to an output stream.

There is a one-to-one mapping between elements of C (characters) (7.1.4.1) and elements of CC (character codes).

7.1.3 Floating point

A floating point value is an element of a set F (see 6.1.2d) where F is a finite subset of $\mathcal{R}$ characterized by five parameters:

$r \in \mathcal{Z}$ (the radix of F)
$p \in \mathcal{Z}$ (the precision of F)
$emin \in \mathcal{Z}$ (the smallest exponent of F)
$emax \in \mathcal{Z}$ (the largest exponent of F)
$denorm \in Boolean$ (whether F contains denormalized values)

These parameters shall satisfy:

r is a positive even integer
$r^{p-1} \geq 10^6$
$(emin - 1) < -2 * (p - 1)$
$emax > 2 * (p - 1)$

These parameters should also satisfy:

$$-2 \leq (emin - 1) + emax \leq 2$$

Given specific values for r, p, $emin$, $emax$, and $denorm$,

$$F_N = \{0, \pm i * r^{e-p} \mid i, e \in \mathcal{Z},\ r^{p-1} \leq i \leq r^p - 1,\ emin \leq e \leq emax\}$$

$$F_D = \{\pm i * r^{emin-p} \mid i \in \mathcal{Z},\ 1 \leq i \leq r^{p-1} - 1\}$$

$$\begin{aligned} F &= F_N \cup F_D && \text{if } denorm = \mathbf{true} \\ &= F_N && \text{if } denorm = \mathbf{false} \end{aligned}$$

The elements of F_N are called *normalized* floating point values because of the constraint $r^{p-1} \leq i \leq r^p - 1$. The elements of F_D are called *denormalized* floating point values.

The type F is called *normalized* if it contains only normalized values, and called *denormalized* if it contains denormalized values as well.

NOTES

1 This draft International Standard does not advocate any particular representation for floating point values. However, concepts such as *radix*, *precision*, and *exponent* are derived from an abstract model of such values that is discussed in LCAS (A.4.2).

2 The floating point type has commonly, but misleadingly, been known as "real" in many Prolog processors.

3 The constraints on the parameters are justified and explained in the LCAS rationale.

4 The terms *normalized* and *denormalized* refer to the mathematical values involved, not to any method of representation.

5 The syntax of a floating point number is defined in 6.3.1.1, 6.3.1.2, and 6.4.5.

7.1.3.1 Additional floating point constants and sets

For convenience, five constants, and an unbounded set are defined:

$$\begin{aligned} fmax &= max\ \{z \in F \mid z > 0\} \\ &= (1 - r^{-p}) * r^{emax} \end{aligned}$$

$$\begin{aligned} fmin_N &= min\ \{z \in F_N \mid z > 0\} \\ &= r^{emin-1} \end{aligned}$$

$$\begin{aligned} fmin_D &= min\ \{z \in F_D \mid z > 0\} \\ &= r^{emin-p} \end{aligned}$$

$$\begin{aligned} fmin &= min\ \{z \in F \mid z > 0\} \\ &= fmin_D && \text{if } denorm = \mathbf{true} \\ &= fmin_N && \text{if } denorm = \mathbf{false} \end{aligned}$$

$epsilon = r^{1-p}$ (the maximum relative error in F_N)

$$F^* = F_N \cup F_D \cup \{\pm i * r^{e-p} \mid i, e \in \mathcal{Z},\ r^{p-1} \leq i \leq r^p - 1,\ e > emax\}$$

NOTES

1 F^* contains values beyond those that are representable in the type F.

7.1.4 Atom

An atom is an element of the set A (see 6.1.2b) and serves as an identifier, for example the name of a predicate or functor, or as a programmer's mnemonic for one of several distinct items.

7.1.4.1 Characters

C, a set of characters, is a subset of A where PCS is the processor character set (6.5).

$$C = \{a \in A \mid a \in PCS\}$$

NOTE — There is a one-to-one mapping between elements of C (characters) and elements of CC (character codes) (7.1.2.2).

7.1.4.2 Boolean

$Bool$ is a subset of A.

$Bool = \{$ `true` , `false` $\}$

NOTE — An element of *Bool* is used during input/output to indicate whether or not a particular option is applicable (7.10).

7.1.5 Compound term

A compound term is an element of a set *CT* (see 6.1.2e) and is an arbitrary data structure. It has a functor which is an identifier with an arity, and a number of terms as the arguments.

NOTE — The syntax of a compound term is defined in 6.1.2e, 6.3.3, 6.3.4, and 6.3.5.

7.1.6 Related terms

7.1.6.1 Variants of a term

Two terms are variants if there is a bijection `s` of the variables of the former to the variables of the latter such that the latter term results from replacing each variable `X` in the former by `Xs`.

NOTES

1 For example, `f(A, B, A)` is a variant of `f(X, Y, X)`, `g(A, B)` is a variant of `G(_, _)`, and `P+Q` is a variant of `P+Q`.

2 The concept of a witness is required when defining `bagof/3` (8.10.2) and `setof/3` (8.10.3).

7.1.6.2 Renamed copy of a term

A term `T2` is a renamed copy of a term `T1` if:

a) `T2` is a variant of `T1`, and

b) None of the variables in the variable set of `T2` exist in the complete database or execution stack.

NOTE — The concept of a renamed copy of a term is required when defining the execution of a user-defined procedure (7.7.10), and the built-in predicates `functor/3` (8.5.1), `copy_term/2` (8.5.4), `clause/2` (8.8.1), etc.

7.1.6.3 Iterated-goal term

The iterated-goal term `G` of a term `T` is a term defined recursively as follows:

a) If `T` unifies with `^(_, Goal)` then `G` is the iterated-goal term of `Goal`,

b) Else `G` is `T`.

NOTES

1 For example, `foo(X)` is the iterated-goal term of `^(X, foo(X))`.

2 The concept of an iterated-goal term is required when defining `bagof/3` (8.10.2) and `setof/3` (8.10.3).

7.1.6.4 Proper sublist of a list

`SL` is a proper sublist of a list `L` if:

a) `SL` and `L` are both the empty list, or

b) `SL` is a proper sublist of the tail of `L`, or

c) The head of `SL` and `L` are identical, and the tail of `SL` is a proper sublist of the tail of `L`.

NOTES

1 For example, `[1,3,4]`, `[2,3]`, `[5]`, and `[]` are all proper sublists of `[1,2,3,4,5]`.

2 The concept of a proper sublist is required when defining `bagof/3` (8.10.2) and `setof/3` (8.10.3).

7.1.6.5 Sorted list of a list

`SL` is the sorted list of a list `L` if:

a) `L_element` is an element of `SL` iff `L_element` is an element of `L`, and

b) `L1_element` and `L2_element` are successive elements of `SL` iff `L1_element` term-precedes (see 7.2, 7.2.1) `L2_element`.

NOTES

1 For example, `[1,2,3]` is the sorted list of `[2,3,1,2,1]`; and `[X,Y,-X,-Y]` (but not `[X,Y,-Y-X]`) may be the sorted list of `[-X,Y,-Y,X]`.

2 The concept of a sorted list is required when defining `setof/3` (8.10.3).

7.2 Term order

An ordering *term_precedes* is defined for any two terms.

If `X` and `Y` are identical terms then `X` *term_precedes* `Y` and `Y` *term_precedes* `X` are both false.

If `X` and `Y` have different types: `X` *term_precedes* `Y` iff the type of `X` precedes the type of `Y` in the following

order: `variable` precedes `floating point` precedes `integer` precedes `atom` precedes `compound`.

NOTE — Built-in predicates which test the ordering of terms are defined in 8.4.

7.2.1 Variable

If `X` and `Y` are variables which are not identical then `X` *term_precedes* `Y` shall be implementation dependent except that in any sorted list (7.1.6.5) the ordering shall remain constant.

NOTE — If `X` and `Y` are both anonymous variables then they are not identical terms (see 6.1.2a).

7.2.2 Floating point

If `X` and `Y` are floating point numbers then `X` *term_precedes* `Y` iff `'<'(X, Y)`.

7.2.3 Integer

If `X` and `Y` are integers then `X` *term_precedes* `Y` iff `'<'(X, Y)`.

7.2.4 Atom

If `X` and `Y` are atoms then `X` *term_precedes* `Y` iff:

a) `X` is the null atom and `Y` is not the null atom, or

b) the value in the collating sequence (6.6) of the first character (6.1.2b) of `X` is less than the the value in the collating sequence of the first character of `Y`, or

c) the value in the collating sequence of the first character of `X` is equal to the the value in the collating sequence of the first character of `Y`, and `XT` *term_precedes* `YT` where `XT` is the atom obtained by deleting the first character of `X`, and `YT` is the atom obtained by deleting the first character of `Y`.

NOTE — The collating sequence 6.6 is implementation defined.

7.2.5 Compound

If `X` and `Y` are compound terms then `X` *term_precedes* `Y` iff:

a) The arity of `X` is less than the arity of `Y`, or

b) `X` and `Y` have the same arity, and the functor name of `X` is `FX`, and the functor name of `Y` is `FY`, and `FX` *term_precedes* `FY` or

c) `X` and `Y` have the same functor name and arity, and there is a positive integer N such that:

1) if, for all `I` less than N `Xi` is the `I`th argument of `X` and `Yi` the `I`th argument of `Y` then `'=='(Xi, Yi)`, and

2) if `XN` is the Nth argument of `X` and `YN` the Nth argument of `Y` and `XN` *term_precedes* `YN`.

7.3 Unification

Unification is a basic device of Prolog which affects the success or failure of goals, and causes the instantiation of variables. It is defined on terms as specified by their abstract syntax.

Built-in predicates which unify two terms explicitly are defined in 8.2.

7.3.1 The mathematical definition

A substitution σ is a unifier of two terms if the instances of these terms by the substitution are identical. Formally, σ is a unifier of t_1 and t_2 iff $t_1\sigma$ and $t_2\sigma$ are identical. It is also a *solution* of the equation $t_1 = t_2$, which by analogy is called the unifier of the equation. The notion of unifier extends straightforwardly to several terms or equations. Terms or equations are said *unifiable* if there exists a unifier for them. They are *not unifiable* otherwise.

A unifier is a *most general unifier MGU* of terms if any unifier of these terms is an instance of it. A most general unifier always exists for terms if they are unifiable. There are infinitely many equivalent unifiers through renaming. A substitution is *idempotent* if successive application to itself yield to the same substitution (it is equivalent to say that no variable of its domain occurs in the resulting terms). There is only one most general idempotent unifier for terms, whose domain is limited to the variables of the terms, up to a renaming. It is sometimes called the *unique most general unifier*.

7.3.2 Herbrand algorithm

A non-deterministic algorithm, called the “Herbrand algorithm”, computes the unique most general unifier MGU of a set of equations.

It is given with the only purpose to define the concepts presented in the next section ($NSTO, STO$). Conforming processors are not required to implement this algorithm.

The Herbrand algorithm is:

Given a set of equations of the form $t_1 = t_2$ apply in any order one of the following non-exclusive steps:

a) If there is an equation of the form:

1) $f = g$ where f and g are different constants, or

2) $f = g$ where f is a constant and g is a compound term, or f is a compound term and g is a constant, or

3) $f(...) = g(...)$ where f and g are different functors, or

4) $f(a_1, a_2, ...a_N) = f(b_1, b_2, ...b_M)$ where N and M are different.

then exit with failure (*not unifiable*).

b) If there is an equation of the form $X = X$, X being a variable, then remove it.

c) If there is an equation of the form $c = c$, c being a constant, then remove it.

d) If there is an equation of the form $f(a_1, a_2, ...a_N) = f(b_1, b_2, ...b_N)$ then replace it by the set of equations $a_i = b_i$.

e) If there is an equation of the form $t = X$, X being a variable and t a non-variable term, then replace it by the equation $X = t$,

f) If there is an equation of the form $X = t$ where:

1) X is a variable and t a term in which the variable X does not occur, and

2) the variable X occurs in some other equation,

then substitute in all other equations every occurrence of the variable X by the term t.

g) If there is an equation of the form $X = t$ such that X is a variable and t is a non-variable term which contains this variable, then exit with failure (*not unifiable, positive occurs-check*).

h) If no transformation can be applied any more, then exit with success ($unifiable$).

This algorithm always terminates. If it terminates with success ($unifiable$) the remaining set of equations

$$(v_1 = t_1, v_2 = t_2, ..., v_N = t_N)$$

defines an MGU

$$\{v_1 \rightarrow t_1, v_2 \rightarrow t_2, ...v_N \rightarrow t_N\}$$

Examples in table 6 show the operation of the algorithm. The final two examples show that the result of the algorithm is not necessarily unique.

7.3.3 Subject to occurs-check (STO) and not subject to occurs-check ($NSTO$)

A set of equations (or two terms) are "subject to occurs-check" (STO) iff there exists a way to proceed through the steps of the Herbrand Algorithm such that 7.3.2g happens.

A set of equations (or two terms) are "not subject to occurs-check" ($NSTO$) iff there exists no way to proceed through the steps of the Herbrand Algorithm such that 7.3.2g happens.

NOTE — STO and $NSTO$ are decidable properties for a single unification. However processors are not required to include such a test.

A Prolog text (including goals) is $NSTO$ if and only if all unifications during its execution are $NSTO$. It is STO otherwise. The property STO (or $NSTO$) for a program is not decidable. However there are tests which guarantee that for a given processor, a program is $NSTO$. These tests are just sufficient conditions.

7.3.4 Normal unification in Prolog

Unification of two terms is defined in Prolog as:

a) If two terms are STO then the result is undefined.

b) If two terms are $NSTO$ and the two terms are unifiable, then the result is an MGU.

c) If two terms are $NSTO$ and the two terms are not unifiable, then the result is failure.

This definition of unification applies both to the normal unification predicate `=/2` and also when unification is invoked implicitly in this draft International Standard.

It is the responsibility of the programmer to ensure that Prolog text will be $NSTO$ when executed on a standard-conforming processor. Programs are standard-conforming with respect to unification iff:

Table 6 — Unification examples

Step	The set of equations
	$3 = 3$
(7.3.2c)	
(7.3.2h)	success ($unifiable$)
	$MGU = \{\ \}$
	$X = Y$
(7.3.2h)	success ($unifiable$)
	$MGU = \{X \rightarrow Y\}$
	$3 = 4$
(7.3.2a1)	failure (*not unifiable*)
	$3 = f(X)$
(7.3.2a2)	failure (*not unifiable*)
	$f(X) = g(X)$
(7.3.2a3)	failure (*not unifiable*)
	$f(X) = f(g(X),\ 1)$
(7.3.2a4)	failure (*not unifiable*)
	$f(X) = f(X)$
(7.3.2d)	$X = X$
(7.3.2b)	
(7.3.2h)	success ($unifiable$)
	$MGU = \{\ \}$
	$f(X,\ Y) = f(g(Y),\ a)$
(7.3.2d)	$X = g(Y),\ Y = a$
(7.3.2f)	$X = g(a),\ Y = a$
(7.3.2h)	success ($unifiable$)
	$MGU = \{X \rightarrow g(a),\ Y \rightarrow a\}$
	$f(X,\ X,\ X) = f(Y,\ g(Y),\ a)$
(7.3.2d)	$X = Y,\ X = g(Y),\ X = a$
(7.3.2f)	$a = Y,\ a = g(Y),\ X = a$
(7.3.2e)	$Y = a,\ a = g(Y),\ X = a$
(7.3.2f)	$Y = a,\ a = g(a),\ X = a$
(7.3.2a2)	failure (*not unifiable*)
	$f(X,\ X,\ X) = f(Y,\ g(Y),\ a)$
(7.3.2d)	$X = Y,\ X = g(Y),\ X = a$
(7.3.2f)	$X = Y,\ Y = g(Y),\ Y = a$
(7.3.2g)	failure (*failure, positive occurs-check*)

a) they are $NSTO$ on a standard-conforming processor or,

b) all unifications which are STO are made using the predicate `unify_with_occurs_check/2` (8.2.2).

NOTES

1 When a built-in predicate can be called in a way which is undefined by this draft International Standard because there is implicit unification of two terms which are STO, the examples accompanying the definition of the built-in predicate often include one such example.

2 This is compatible with most implementations which do not include the occurs-check test for efficiency reasons. In this case the unification algorithm may or may not terminate. But most practical programs are $NSTO$, and for those that are STO, existing implementations often have the same behaviour. This is why `=/2` is not defined when its arguments are STO.

3 Although the $NSTO$ property is undecidable, it is possible to avoid testing for it by using explicitly a unification with occurs-check in a program. This will guarantee that the execution of a program remains defined by this draft International Standard. It is thus possible to apply explicitly unification with occurs-check whenever it is needed by calling the built-in predicate `unify_with_occurs_check/2` whose semantics is:

a) If two terms are unifiable, then the result is an MGU.

b) If two terms are not unifiable, then the result is failure.

7.3.4.1 Example

The predicate `unify_with_occurs_check/2` enables the programmer to avoid unsafe unifications whether they are explicit (replacing calls of `=/2`) or implicit, for example when a goal is being matched with clause heads. But in the latter case, care is needed, for example consider the utility predicate `append/3` defined by the clauses:

```
append([], L, L) :-
   is_list(L).
append([H|L1], L2, [H|L12]) :-
   append(L1, L2, L12).
```

The goals

```
append([], L, [a|L])
```

and

```
append([f(X,Y,X)], [], [f(g(X),g(Y),Y)])
```

are STO. If there might be such a call, and the programmer wishes to ensure that execution is standard-conforming, then calls of `append/3` must be replaced by calls of `safe_append/3` which is defined as:

```
safe_append([], L1, L2) :-
   unify_with_occurs_check(L1, L2),
   is_list(L1).
safe_append([H1|L1], L2, [H2|L12]) :-
   unify_with_occurs_check(H1, H2),
   safe_append(L1, L2, L12).
```

7.4 Prolog text

Prolog text specifies:

a) User-defined procedures in a textual form, and

b) Goals which are to be executed as soon as execution begins, and

c) Operators which affect the syntax of Prolog text.

NOTES

1 The concrete and abstract syntax for Prolog text is defined in 6.2 and 6.2.1.

2 Preparing a Prolog text for execution is defined in 7.5.1.

7.4.1 Undefined features

This draft International Standard leaves undefined:

a) The mechanisms for transforming clauses and directives of Prolog text into procedures of the database, or

b) The complete rules for combining Prolog text occurring in more than one text unit into a single equivalent sequence of Prolog text, or

c) The action to be taken if the read-terms forming Prolog text do not conform to the requirements of this draft International Standard.

NOTE — This draft International Standard does not define a built-in predicate `consult/1`, nor any similar predicate.

7.4.2 Directives

A directive d in Prolog text (6.2.1.1) specifies:

a) properties of the procedures defined in Prolog text, or

b) the format of read-terms in Prolog text, or

c) a goal to be executed after the Prolog text has been prepared for execution, or

d) another text unit of Prolog text which is to be prepared for execution.

NOTE — The usage and semantics of directives may be altered in Part 2 (Modules) of this draft International Standard.

7.4.2.1 Directive `dynamic/1`

A directive `dynamic(P)` where `P` is a predicate indicator defines the user-defined procedure specified by `P` to be dynamic.

`P` shall not be a control construct or built-in predicate.

If Prolog text contains a directive `dynamic(P)`, then that directive may occur any number of times in that Prolog text. The first such directive shall precede all clauses for the user-defined procedure `P`.

7.4.2.2 Directive `multifile/1`

The clauses for each user-defined procedure with predicate indicator `UP` shall be read-terms of a single Prolog text unless there is a `multifile(UP)` directive in that Prolog text.

A directive `multifile(P)` where `P` is a predicate indicator defines the user-defined procedure specified by `P` may be defined by clauses in more than one Prolog text.

`P` shall not be a control construct or built-in predicate.

If Prolog text contains a directive `multifile(P)`, then that directive may occur any number of times in that Prolog text. The first such directive shall precede all clauses for the user-defined procedure `P`.

If Prolog text contains a directive `multifile(P)`, then that directive shall occur in every Prolog text which contains clauses for the user-defined procedure `P`. Further, if `P` is defined to be a dynamic procedure in one Prolog text, then a directive `dynamic(P)` shall occur in every Prolog text which contains clauses for the user-defined procedure `P`.

7.4.2.3 Directive `discontiguous/1`

The clauses for each user-defined procedure with predicate indicator `UP` shall be consecutive read-terms of a single Prolog text unless there is a `discontiguous(UP)` directive in that Prolog text.

A directive `discontiguous(P)` where `P` is a predicate indicator defines the user-defined procedure specified by `P` may be defined by clauses which are not contiguous in the Prolog text.

`P` shall not be a control construct or built-in predicate.

If Prolog text contains a directive `discontiguous(P)`, then that directive may occur any number of times in

that Prolog text. The first such directive shall precede all clauses for the user-defined procedure `P`.

7.4.2.4 Directive `op/3`

A directive `op(Priority, Op_specifier, Operator)` specifies that, in subsequent read-terms of the Prolog text, `Operator` shall be an operator with properties defined by specifier `Op_specifier` and precedence `Priority`.

The arguments `Priority`, `Op_specifier`, and `Operator` shall satisfy the same constraints as those required for a successful execution of the built-in predicate `op/3` (8.14.13).

It shall be implementation defined whether or not an operator defined in `op(P, S, O)` directive shall affect the syntax of read-terms in other Prolog texts or during execution.

7.4.2.5 Directive `char_conversion/2`

A directive `char_conversion(In, Out)` specifies that, in this Prolog text, when the value associated with the flag `char_conversion` (7.11.2.1) is `on`, an unquoted character `In` will be replaced by a character `Out`.

A character is a unquoted character iff it is not a quoted character (6.4.2.1).

The arguments `Input_char` and `Out_char` shall satisfy the same constraints as those required the built-in predicate `char_conversion/2` (8.14.15) to succeed.

It shall be implementation defined whether or not the character-conversion relation defined by `char_conversion(In, Out)` directives shall affect the characters of read-terms in other Prolog texts or during execution.

NOTE — The rationale for providing this facility is because some extended character sets (for example, Japanese JIS character sets) are used with the basic character set and contain the characters equivalent to those in the basic character set with different encoding. In such cases, users will often wish the meaning of characters in data and Prolog text to be the same regardless of the encoding.

7.4.2.6 Directive `initialization/1`

A directive `initialization(G)` includes the goal `G` in a set of goals which shall be executed immediately after the Prolog text has been prepared for execution. The order in which any such goals will be executed shall be implementation defined.

7.4.2.7 Directive `include/1`

Prolog text $P1$ which contains an `include/1` directive is identical to a Prolog text $P2$ obtained by replacing the directive `include(F)` in $P1$ by the Prolog text denoted by `F` where `F` is an implementation defined ground term designating a Prolog text unit.

7.4.2.8 Directive `ensure_loaded/1`

A directive `ensure_loaded(P_text)` specifies that the Prolog text being prepared for execution shall include the Prolog text denoted by `P_text` where `P_text` is an implementation defined ground term designating a Prolog text unit.

When there are multiple `ensure_loaded/1` directives for the same Prolog text, only one copy is included in the Prolog text prepared for execution. The position where the copy is included in the Prolog text is implementation defined.

7.4.3 Clauses

A read-term c representing a clause in Prolog text (6.2.1.2) defines a clause of the user-defined procedure with predicate indicator `P/N`:

a) If the principal functor of c is `:-/2`, and the principal functor of the first argument of c is `P/N`,

b) Else the principal functor of c is not `:-/2`, and the principal functor of c is `P/N`.

`P/N` shall not be the predicate indicator of a built-in predicate or a control construct.

NOTE — It is not possible to define a user-defined procedure with predicate indicator `:-/1` or `:-/2` in Prolog text.

7.5 Database

The database is the set of user-defined procedures which currently exist during execution.

The complete database is the collection of procedures with respect to which execution is performed. Each procedure is:

a) a control construct, or

b) a built-in predicate, or

c) a user-defined procedure.

Each procedure has a unique predicate which can be used to identify it. A predicate has two parts: an identifier (which is an atom), and an arity (which is an integer).

Built-in predicates and control constructs are provided by the processor. They have properties which are defined by the clauses in this draft International Standard. In particular, they cannot be altered or deleted during execution (see 7.5.3).

A user-defined procedure is a sequence of (zero or more) clauses prepared for execution.

7.5.1 Preparing a Prolog text for execution

Preparing a Prolog text for execution shall result in the complete database and processor being in an initial state of execution.

The means by which a Prolog processor is asked to prepare standard-conforming Prolog texts (6.2) for execution shall be implementation defined. The manner in which a Prolog processor prepares standard-conforming Prolog texts for execution shall be implementation dependent. This process converts the read-terms in a Prolog text to the clauses of user-defined procedures in the database.

All clauses of a procedure are ordered for execution according to the textual (or temporal) order of these clauses as they were prepared for execution.

Any effects of reordering, adding or removing clauses by directives during preparation for execution are implementation defined.

The clauses of different procedures have no temporal or spatial correlation.

The effect of directives while preparing a Prolog text for execution is defined in (7.4.2).

7.5.2 Static and dynamic procedures

Each procedure is either dynamic or static. Each built-in predicate and control construct shall be static, and a user-defined procedure shall be either dynamic or static.

By default a user-defined procedure shall be static, but (1) a directive in Prolog text can override the default, and (2) asserting a clause of a previously non-existent procedure shall create a dynamic procedure.

A clause of a dynamic procedure can be inspected or altered, a clause of a static procedure cannot be inspected or altered.

Attempts to perform invalid operations on the complete database cause a permission error (7.12.2e).

NOTES

1 While Prolog was implemented as a simple interpreted system, it was sufficient to classify procedures as built-in (and static) or user-defined (and dynamic). But the subsequent development of compilers and libraries requires a more sophisticated classification in order to achieve greater efficiency.

2 The restriction that only dynamic procedures can be altered enables “partial evaluation” to be performed on any procedure which is static.

3 The distinction between static and dynamic is also important for users, for example, when developing a library, procedures can be dynamic during development, but then be made static for users of the library.

4 A processor which regards all user-defined procedures as dynamic can execute standard-conforming programs correctly if the program does not cause a Database Error by attempting to alter a static predicate.

7.5.3 A logical database

The database is frozen during the execution of a goal, that is, any change in the database that occurs as the result of executing a goal (for example, `assertz/1`, `retract/1`) affects only an activation whose execution begins afterwards. The change shall not affect any activation that is currently being executed.

7.6 Converting a term to a clause, and a clause to a term

Prolog provides the ability to convert data to and from code. But an argument of a goal is a term, while the complete database contains procedures with the user-defined procedures being formed from clauses. Some predicates (for example `asserta/1`) convert a term to a corresponding clause, and others (for example `clause/2`) convert a clause to a corresponding term.

NOTES

1 Converting a term `T` to a body `B` and back may result in non-identical term `T'`.

2 Part 2 (Modules) of this draft International Standard may require additional operations when converting a term to a body.

7.6.1 Converting a term to the head of a clause

A term T can be converted to a predication which is the head H of a clause:

a) If T is a compound term whose functor name is FT then the predicate name PH of H is FT, and the arguments of T and H are identical.

b) If T is an atom denoted by the identifier A then the predicate name PH of H is A, and H has no arguments.

NOTE — If T is a number or variable, then T cannot be converted to a head.

7.6.2 Converting a term to the body of a clause

A term T can be converted to a goal G which is the body of a clause:

a) If T is a variable then G is the control construct call (7.8.3), whose argument is T.

b) If T is a term whose principal functor appears in table 7 then G is the corresponding control construct. If the principal functor of T is `call/1` or `catch/3` or `throw/1` then the arguments of T and G are identical, else if the principal functor of T is `,/2` or `;/2` or `->/2` then each argument of T shall also be converted to a goal.

c) If T is an atom or compound term whose principal functor FT does not appear in table 7 then G is a predication whose predicate indicator is FT, and the arguments, if any, of T and G are identical.

NOTES

1 A variable X and a term `call(X)` are converted to identical bodies.

2 If T is a number then there is no goal which corresponds to T.

7.6.3 Converting the head of a clause to a term

A head H with predicate indicator P/N can be converted to a term T:

a) If N is zero then T is the atom P.

b) If N is non-zero then T is a renamed copy (7.1.6.2) of TT where TT is the compound term whose principal functor is P/N and the arguments of H and TT are identical.

Table 7 — Principal functors and control constructs

Principal functor	Control construct
`,/2`	Conjunction
`;/2`	Disjunction
`->/2`	If-then
`!/0`	Cut
`call/1`	Call
`true/0`	True
`fail/0`	Fail
`catch/3`	Catch
`throw/1`	Throw

7.6.4 Converting the body of a clause to a term

A goal G which is a predication with predicate indicator P/N can be converted to a term T:

a) If N is zero then T is the atom P.

b) If N is non-zero then T is a renamed copy (7.1.6.2) of TT where TT is the compound term whose principal functor is P/N and the arguments of G and TT are identical.

c) If G is a control construct which appears in table 7 then T is a term with the corresponding principal functor. If the principal functor of T is `call/1` or `catch/3` or `throw/1` then the arguments of G and T are identical, else if the principal functor of T is `,/2` or `;/2` or `->/2` then each argument of G shall also be converted to a term.

7.7 Executing a Prolog goal

This clause defines the flow of control through Prolog clauses as a goal is executed.

NOTES

1 This description is consistent with the formal definition in annex A.

2 This clause does not define:

a) The meaning of each built-in predicate,

b) The checks to see whether or not an error condition is satisfied,

c) Side-effects, for example database updates, input/output.

3 The execution model described here is based on a stack (4.2) of execution states.

7.7.1 Execution

Execution is a sequence of activations which attempt to satisfy a goal. Side effects can occur, and the database can be altered during execution.

Each execution step is represented by a sequence of execution states.

Execution may or may not terminate. If it does, the result shall be to realize side effects during execution (for example, input/output, file and screen handling, user dialogue, modification of the database etc.), and:

a) To fail the initial goal (that is, to give a negative answer in an implementation defined form) with respect to the complete database, or

b) To satisfy the initial goal (that is, to give a positive answer in an implementation defined form) with respect to the complete database, and perhaps instantiating some or all of the variables of the initial goal.

7.7.2 Data types for the execution model

The execution model of Prolog is based on a execution stack S of execution states ES.

ES is a structured data type with components:

S_index — A value which identifies a component of S.

$decsglstk$ — A stack of decorated subgoals, that defines an activation path of the activators which might be activated during execution.

$subst$ — A substitution which defines the state of the instantiations of the variables.

BI — Backtrack Information: a value which defines how to re-execute a goal.

The $choicepoint$ for the execution state ES_{i+1} is ES_i.

A decorated subgoal is a structured data type with components:

$activator$ — A predication prepared for execution which must be executed successfully in order to execute the goal.

$cutparent$ — A pointer to a deeper execution state that indicates where control is resumed should a cut be re-executed (see 7.8.4.1).

$currstate$, the current execution state, is $top(S)$. It contains:

a) An index which identifies its position in S, and

b) The current decorated subgoal stack (i.e. the current goal), and

c) The current substitution, and

d) Backtracking information.

$currdecsgl$, the current decorated subgoal, is $top(decsglstk)$ of $currstate$. It contains:

a) The current activator, $curract$, and

b) its $cutparent$.

BI has a value:

nil — Its initial value, or

$ctrl$ — The procedure is a control construct, or

bip — The activated procedure is a built-in predicate, or

$up(CL)$ — CL is a list of the clauses of a user-defined procedure whose predicate is identical to $curract$, and which are still to be executed.

NOTES

1 Thus the data structures are:

S = (ES_N, ES_{N-1}, ES_{N-2}, ... ES_1, nil)

ES_i = (S_i, $currentgoal_i$, $subst_i$, BI_i)

$currstate$ = $top(S)$ = ES_N

$currentgoal$ = ($decoratedsubgoal_J$, ... $decoratedsubgoal_1$, nil)

$currdecsgl$ = $decoratedsubgoal_J$

$decoratedsubgoal_j$ = ($activator_j$, $cutparent_j$)

$curract$ is the activator of $currdecsgl$.

2 The concept of a stack is defined in clause 4.2.

7.7.3 Initialization

The method by which a user delivers a goal to the Prolog processor shall be implementation defined.

Table 8 — The execution stack after initialization

S_index	Decorated subgoal stack	Substitution	BI
1	((goal, 0), nil)	$\{\}$	nil
0	nil		

Table 9 — The goal succeeds

S_index	Decorated subgoal stack	Substitution	BI
N	(nil)	Σ	nil
...			
1	((goal, 0), nil)	$\{\}$	...
0	nil		

A goal is prepared for execution by transforming it into an activator.

Table 8 shows the execution stack after it has been initialized and is ready to execute `goal` after it has been transformed into an activator.

Execution can then begin (7.7.7).

NOTE — A processor may support the concept of a query, that is a goal given as interactive input to the top level. But this draft International Standard does not define a means of delivering a goal to the processor except that Prolog text may include a set of goals to be executed immediately after it has been prepared for execution (7.4.2.6).

Nor does this draft International Standard define a means of instructing a processor to find multiple solutions for a goal.

7.7.4 A goal succeeds

A goal is satisfied, i.e. execution succeeds when the decorated subgoal stack of $currstate$ is empty, as in Table 9. A solution for the goal `goal` is represented by the substitution Σ.

7.7.5 A goal fails

Execution fails when the execution stack S is empty, as in table 10.

Table 10 — The goal fails

S_index	Decorated subgoal stack	Substitution	BI
0	nil		

7.7.6 Re-executing a goal

After satisfying an initial goal, execution may continue by trying to resatisfy it.

Procedurally,

a) Pop $currstate$ from S.

b) Continue execution at 7.7.8.

7.7.7 Selecting a clause for execution

Execution proceeds in a succession of steps:

a) The processor searches in the complete database for a procedure p whose predicate is identical with the functor and arity of $curract$.

b) If no procedure has a functor and arity agreeing with the functor and arity of $curract$, then action depends on the value of the flag `undefined_predicate` (7.11.2.4):

`error` — There shall be an error `existence_error(procedure, PF)` where `PF` is the principal functor of $curract$, or

`warning` — an implementation dependent warning shall be generated, and $curract$ replaced by the control construct fail, or

`fail` — $curract$ shall be replaced by the control construct fail.

c) If p is a control construct (true, fail, call, cut, conjunction, disjunction, if-then, if-then-else, catch, throw), then BI is set to $ctrl$ and continue execution according to the rules defined in 7.8,

d) If p is a built-in predicate BP, BI is set to bip, and continue execution at 7.7.12,

e) If p is a user-defined procedure, BI is set to $up(CL)$ where CL is a list of the current clauses of p and continue execution at 7.7.10.

NOTE — BI has the value $up([])$ when (1) all the clauses of p have been examined to see if their head matches $curract$, or (2) If p has no clauses at all.

There is a difference between a procedure which does not exist, and one which exists but has no clauses.

7.7.8 Backtracking

The processor backtracks (a) if a goal has failed, or (b) if the initial goal has been satisfied, and the processor is asked to re-execute it.

Procedurally, backtracking shall be executed as follows:

a) Examine the value of BI for the new $currstate$.

b) If BI is $up(CL)$ then p is a user-defined procedure is being executed, remove the head of CL and continue execution at 7.7.10.

c) If BI is bip then p is a built-in predicate, and continue execution at 7.7.12b.

d) If BI is $ctrl$ then p is a control construct, and the effect of re-executing it is defined in 7.8.

e) If BI is nil, then the new $curract$ has not yet been executed, and continue execution at 7.7.7.

NOTES

1 The control constructs true, fail and throw can never be re-executed because they are removed from $currstate$ as they are executed.

2 The control constructs call, cut, conjunction, disjunction, if-then, and catch all behave like fail when they are re-executed.

3 The control construct if-then-else is re-executed if the if fails so that the $else$ can be executed instead of $then$.

4 Step 7.7.8e happens after the $either$ branch of a disjunction has failed.

7.7.9 Side effects

Side effects that occur during the execution of a goal shall not be undone if the program subsequently backtracks over the goal. Examples include:

a) Changes to the database, for example `abolish/1`, `asserta/1`, `assertz/1`, `retract/1`.

b) Input, for example `read/1`.

c) Output, for example `write/1`.

7.7.10 Executing a user-defined procedure

Procedurally, a user-defined procedure shall be executed as follows:

a) If there are no (more) clauses for p, BI has the value $up([])$ and continue execution at 7.7.11.

b) Else consider clause c where BI has the value $up([c|CT])$,

c) If c has a head which matches $curract$, then it is selected for execution and continue execution at 7.7.10e,

d) Else BI is replaced by a value $up(CT)$ and continue execution at 7.7.10a.

e) Let c' be a renamed copy (7.1.6.2) of the clause c of $up([c|\_])$.

f) Unify the head of c' and $curract$ producing a most general unifier MGU.

g) Apply the substitution MGU to the body of c'.

h) Make a copy CCS of $currstate$. It contains a copy of the current goal which is called CCG.

i) Apply the substitution MGU to CCG (so that variables of CCG which are variables of $curract$ become (partially) instantiated).

j) Replace the current activator of CCG by the MGU-modified body of c'.

k) Set BI of CCS to nil.

l) Set the substitution of CCS to a composition of the substitution of $curract$ and MGU.

m) Set $cutparent$ of the new first subgoal of the decorated subgoal stack of CCS to the current $choicepoint$. Note that the $cutparent$ of the other decorated subgoals are unaltered.

n) Push CCS on to S. It becomes the new $currstate$, and the previous $currstate$ becomes its $choicepoint$.

o) Continue execution at 7.7.7.

NOTES

1 $choicepoint$ will be re-executed if backtracking becomes necessary (7.7.8).

2 The $choicepoint$ is the next execution state, but $cutparent$ points to the execution state below $choicepoint$ because backtracking a cut removes the total activation of a procedure including its activator and $choicepoint$.

Table 11 — Before executing a rule `p(X, Y)`

S_index	Decorated subgoal stack	Substitution	BI
N	((`p(X, Y)`, CP), ...)	Σ	nil
...			

3 When the clause which is selected for execution is a fact, then its body is `true` with an activator $true$ whose activation is described in (7.8.1.1).

7.7.10.1 Example – A user-defined rule

If the first clause of the user-defined procedure `p/2` is
`p(M, W) :- m(M), f(W).`
then the body of this clause in the database will be a conjunction:
`( p(M, W), ',' (m(M), f(W)) )`
and Table 11 shows the execution stack ready to execute a $curract$ `p(X, Y)` using this clause.

The actions to execute this subgoal:

a) $clause\_copy$ =
`( p(M, W), ',' (m(M), f(W)) )`

b) MGU = {X → M, Y → W}

c) Applying MGU to the $clause\_copy$ body, `',' (m(M), f(W))`

d) Make a copy CCS of $currstate$,
CCS = ((`p(X, Y)`, CP), ...), Σ, 1

e) Apply MGU to CCS

f) Replace the activator by the body of $clause\_copy$

g) Set BI to nil

h) Set $cutparent$ of subgoal to the current $choicepoint$ so that now
CCS = ((`',' (m(M), f(W))`, $N-1$), ...),
{ X → M, Y → W } ∘ Σ, nil

i) Push CCS on to S.

Table 12 shows the execution stack after executing the subgoal `p(X, Y)` using the clause `( p(M, W), ',' (m(M), f(W)) )`.

Table 12 — After executing a rule `p(X, Y)`

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((`',' (m(M), f(W))`, $N-1$), ...)	{X → M, Y → W} ∘ Σ	nil
N	((`p(X, Y)`, CP), ...)	Σ	$up[P_1 \mid P_T]$
...			

Table 13 — Before executing a fact `m(pete)`

S_index	Decorated subgoal stack	Substitution	BI
N	((`m(X)`, CP), (`f(W)`, CP), ...)	Σ	nil
...			

7.7.10.2 Example – A user-defined fact

If the first clause of the user-defined procedure `m/1` is
`m(pete).`
then the body of this clause in the database will be `true` because the clause is a fact. Table 13 shows the execution stack ready to execute a $curract$ `m(X)` using this clause.

The actions to execute this subgoal:

a) $clause\_copy$ = `(m(pete), true)`

b) MGU = {$X \rightarrow pete$}

c) Applying MGU to the $clause\_copy$ body, `true`

d) Make a copy CCS of $currstate$,
CCS = ((`m(X)`, CP), (`f(W)`, CP), ...), Σ, 1

e) Apply MGU to CCS

f) Replace the activator by the body of $clause\_copy$

g) Set BI to nil

h) Set $cutparent$ of subgoal to the current $choicepoint$ so that now
CCS = ((`true`, $N-1$), (`f(W)`, CP), ...),
{ X → pete } ∘ Σ, nil

Table 14 — After executing a fact `m(pete)`

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((`true`, $N-1$), (`f(W)`, CP), ...)	{X → pete} ∘ Σ	nil
N	((`m(X)`, CP), (`f(W)`, CP), ...)	Σ	$up[M_1\|M_T]$
...			

i) Push CCS on to S.

Table 14 shows the execution stack after executing the subgoal `m(X)` using the clause `(m(pete), true)`.

7.7.11 Executing a user-defined procedure with no more clauses

When a user-defined procedure has been selected for execution (7.7.7) but has no more clauses, i.e. BI has a value $up([])$, it shall be executed as follows:

a) Pop $currstate$ from S.

b) Continue execution at 7.7.8.

NOTE — The current substitution (whatever was contributed by the current MGU) is thereby lost forever.

Execution has failed completely when S is empty (see 7.7.5).

7.7.12 Executing a built-in predicate

A built-in predicate BP shall be executed as follows:

a) Unify $curract$ and the calling interface of the built-in predicate BP producing a most general unifier MGU.

b) Make a copy CCS of $currstate$. It contains a copy of the current goal which is called CCG.

c) Push CCS on to S. It becomes the new $currstate$, and the previous $currstate$ becomes its $choicepoint$ if backtracking becomes necessary (7.7.8).

d) Execute, or re-execute after backtracking (7.7.8), $curract$ and perform any side effects according to the rules for BP (see 8) This sometimes leads to a further instantiation of variables in the activator; if so the substitution is applied to the appropriate variables of the current goal.

e) If the activation of BP succeeds, then replace the current activator of CCG by an activator $true$ whose activation is described in (7.8.1.1).

f) Else if the activation of BP fails, then replace the current activator of CCG by an activator $fail$ whose activation is described in (7.8.2).

NOTE — Strictly speaking a new stack entry is needed only if the built-in procedure is designated as re-executable. Then the built-in activator could lead to re-activation of that built-in predicate and thereby to different substitutions.

7.8 Control constructs

This definition of each control construct gives its logical meaning, the procedural effect of satisfying it (by describing the changes on the execution stack S), the effect of re-executing it, and some examples.

NOTES

1 A control construct is static.

2 Each control construct described in clause 7.8.x is defined formally in clause A.5.1.x.

7.8.1 true/0

7.8.1.1 Description

`true` is true.

Procedurally, a control construct true, denoted by `true`, shall be executed as follows:

a) Pop $currdecsgl$ (= (`true`, CP)) from $currentgoal$ of $currstate$.

b) Set BI to nil indicating that a new activation of the new $curract$ is to take place.

c) Continue execution at 7.7.7.

NOTES

1 No new execution stack entry is created, and the current substitution remains unchanged.

2 Execution is complete when all activators in the activation path have been replaced by $true$ and deleted so that the decorated subgoal stack becomes empty (see 7.7.4).

Table 15 — Before executing `true`

S_index	Decorated subgoal stack	Substitution	BI
N	((`true`, $N-2$), (`f(W)`, CP), ...)	Σ	$ctrl$
...			

Table 16 — After executing `true`

S_index	Decorated subgoal stack	Substitution	BI
N	((`f(W)`, CP), ...)	Σ	nil
...			

7.8.1.2 Examples

Tables 15 and 16 show the execution stack before and after executing the control construct `true`.

```
true.
   Succeeds.
```

7.8.2 fail/0

7.8.2.1 Description

`fail` is false.

Procedurally, a control construct fail, denoted by `fail`, shall be executed as follows:

a) Pop $currstate$ from S.

b) Continue execution at 7.7.8.

NOTES

1 The current substitution (whatever was contributed by the current MGU) is thereby lost forever.

2 Execution has failed completely when S is empty (see 7.7.5).

Table 17 — Before executing fail

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((`fail`, CP), ...)	Σ	$ctrl$
N	((`f(Y)`, CP), ...)	Σ	$up([F_1\|F_T])$
...			

Table 18 — After executing fail

S_index	Decorated subgoal stack	Substitution	BI
N	((`f(Y)`, CP), ...)	Σ	$up([F_1\|F_T])$
...			

7.8.2.2 Examples

Tables 17 and 18 show the execution stack before and after executing the control construct `fail`.

```
fail.
   Fails.
```

7.8.3 call/1

7.8.3.1 Description

`call(G)` is true iff `G` represents a goal which is true.

When `G` contains `!` as a subgoal, the effect of `!` shall not extend outside `G`.

Procedurally, a control construct call, denoted by `call(G)`, shall be executed as follows:

a) Make a copy CCS of $currstate$.

b) Set BI of CCS to nil.

c) Pop $currdecsgl$ (= (`call(G)`), CP) from $currentgoal$ of CCS.

d) If the term `G` is a variable, there shall be an instantiation error (7.12.2a),

Table 19 — Before executing `call(G)`

S_index	Decorated subgoal stack	Substitution	BI
N	((`call(G)`, CP), ...)	Σ	$ctrl$
...			

Table 20 — After executing `call(G)`

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((`G`, $N-1$), ...)	Σ	nil
N	((`call(G)`, CP), ...)	Σ	$ctrl$
...			

e) Else if the term `G` is a number, there shall be a type error (7.12.2b),

f) Else convert the term `G` to a goal $goal$ (7.6.2).

g) Push ($goal$, N) on to $currentgoal$ of CCS.

h) Push CCS on to S.

i) Continue execution at 7.7.7.

j) Pop $currstate$ from S.

k) Continue execution at 7.7.8.

`call(G)` is re-executable. On backtracking, continue at 7.8.3.1j.

NOTE — Executing a call has the effect that:

a) If $goal$ should fail, then the call will fail, and

b) $goal$ can be re-executed, and

c) Any cut inside $goal$ is local to $goal$ because the $cutparent$ for $goal$ is the $choicepoint$ for the call.

7.8.3.2 Errors

a) `G` is a variable
— `instantiation_error`.

b) `G` is not a callable term
— `type_error(callable, G)`.

c) `G` cannot be converted to a goal
— `type_error(callable, G)`.

7.8.3.3 Examples

Tables 19 and 20 show the execution stack before and after executing the control construct `call(G)`.

The examples defined in this clause assume the database has been created from the following Prolog text:

```
b(X) :-
   Y = (write(X), X),
   call(Y).

a(1).
a(2).


call(!).
   Succeeds.

call(fail).
   Fails.

call((fail,X)).
   Fails.

call( (fail, call(1)) ).
   Fails.

b(_).
   Outputs 'X', then
   instantiation_error.

b(3).
   Outputs '3', then
   type_error(callable, 3).

Z = !, call( (Z=!, a(X), Z) ).
   Succeeds, unifying X with 1, and Z with !.
   On re-execution, fails.

call( (Z=!, a(X), Z) ).
   Succeeds, unifying X with 1, and Z with !.
   On re-execution, succeeds, unifying X with 2,
      and Z with !.
   On re-execution, fails.

call((write(3), X)).
   Outputs '3', then
   instantiation_error.

call((write(3), call(1)).
   Outputs '3', then
   type_error(callable, 1).

call(X).
   instantiation_error.

call(1).
   type_error(callable, 1).

call((fail, 1)).
   type_error(callable, (fail, 1)).
```

```
call((write(3), 1)).
   type_error(callable, (write(3), 1)).

call((1;true)).
   type_error(callable, (1;true)).
```

7.8.4 !/0 – cut

7.8.4.1 Description

`!` is true.

Procedurally, a control construct cut, denoted by `!`, shall be executed as follows:

a) Make a copy CCS of $currstate$.

b) Replace the $curract$, `!`, of CCS by `true`.

c) Push CCS on to S.

d) Continue execution at 7.7.7.

e) Make a copy cut of $cutparent$ of $currstate$.

f) Pop $currstate$ from S.

g) If $cut = S\_index$ of $top(S)$ then $top(S)$ becomes the new $currstate$, and continue execution by backtracking at 7.7.8.

h) Else continue execution at 7.8.4.1f.

`!` is re-executable. On backtracking, continue at 7.8.4.1e.

NOTES

1 Executing a cut has the effect that:

a) A cut always succeeds, but

b) No attempts are made to resatisfy the goals on S between the cut and its $cutparent$.

c) Re-executing a cut always fails, but unlike fail where the $choicepoint$ for $currstate$ is then re-executed, elements of S are popped until the $cutparent$ associated with the cut equals the S_index for $currstate$.

2 The execution states between a current execution state which has a cut as current activator, and the $cutparent$ of the current decorated subgoal, could be removed as soon as the cut is executed because they can never be reached by backtracking. But these (dead) execution states are left on S so that it always indicates how the current state of execution has been reached.

Table 21 — Before executing cut

S_index	Decorated subgoal stack	Substitution	BI
N	((!, CP), ...)	Σ	$ctrl$
...			

Table 22 — After executing cut

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((true, CP), ...)	Σ	nil
N	((!, CP), ...)	Σ	$ctrl$
...			

7.8.4.2 Examples

Tables 21 and 22 show the execution stack before and after executing the control construct `!`.

Tables 23 and 24 show the effect of re-executing a cut.

The following examples assume the database contains the following clauses:

```
   twice(!) :- write('C ').
   twice(true) :- write('Moss ').

   goal((twice(_), !)).
   goal(write('Three ')).


!.
   Succeeds.

(!, fail; true).
   Fails.

(call(!), fail; true).
   Succeeds.

twice(_), !, write('Forwards '), fail.
   Outputs "C Forwards ",
   Fails.

(! ; write('No ')), write('Cut disjunction '),
      fail.
   Outputs "Cut disjunction ",
   Fails.

twice(_), (write('No '); !), write('Cut '), fail.
   Outputs "C No Cut Cut ",
   Fails.
```

Table 23 — Before re-executing cut

S_index	Decorated subgoal stack	Substitution	BI
N	((!, CP), ...)	Σ	$ctrl$
...			
CP	((p(X, Y), CP), ...)	σ	$up[P_1\|P_T]$
...			

Table 24 — After re-executing cut

S_index	Decorated subgoal stack	Substitution	BI
CP	((p(X, Y), CP), ...)	σ	$up[P_1\|P_T]$
...			

```
twice(_), (!, fail; write('No ')).
   Outputs "C ",
   Fails.

twice(X), X, write('Forwards '), fail.
   Outputs "C Forwards Moss Forwards ",
   Fails.

goal(X), X, write('Forwards '), fail.
   Outputs "C Forwards Three Forwards ",
   Fails.

twice(_), fail_if(fail_if(!)),
     write('Forwards '), fail.
   Outputs "C Forwards Moss Forwards ",
   Fails.

twice(_), once(!),
     write('Forwards '), fail.
   Outputs "C Forwards Moss Forwards ",
   Fails.

twice(_), call(!),
     write('Forwards '), fail.
   Outputs "C Forwards Moss Forwards ",
   Fails.
```

7.8.5 ','/2 – conjunction

7.8.5.1 Description

`','(First, Second)` is true iff `First` is true and `Second` is true.

Procedurally, a control construct denoted by `','(First, Second)`, shall be executed as follows:

a) Make a copy CCS of $currstate$. It contains a copy of the current goal which is called CCG.

b) Replace the current activator of CCG by a pair of activators $first$ and $second$.

c) Set BI of CCS to nil.

d) Push CCS on to S.

e) Continue execution at 7.7.7.

f) Pop $currstate$ from S.

g) Continue execution at 7.7.8.

`','(First, Second)` is re-executable. On backtracking, continue at 7.8.5.1f.

NOTES

1 `','` is a predefined infix operator.

2 Step 7.8.5.1d makes CCS the new $currstate$, and the previous $currstate$ becomes its $choicepoint$. $first$ becomes the new $curract$, if it succeeds $second$ shall be executed.

The $cutparent$ of the new first subgoal of the decorated subgoal stack of CCS is the same as the previous $choicepoint$ because a conjunction is transparent to cut.

3 Executing a conjunction has the effect that:

a) The activator $first$ must succeed, and then the activator $second$ must succeed for the conjunction to effectively succeed, and

b) Conjunction is transparent to cut because the $cutparent$ for $first$ and $second$ are the same as that for the conjunction.

7.8.5.2 Examples

Tables 25 and 26 show the execution stack before and after executing the control construct `','(First, second)`.

```
','(X=1, var(X)).
   Fails.

','(var(X), X=1).
```

Table 25 — Before executing a conjunction

S_index	Decorated subgoal stack	Substitution	BI
N	((',' (First, Second), CP), ...)	Σ	$ctrl$
...			

Table 26 — After executing a conjunction

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((First, CP), (Second, CP), ...)	Σ	nil
N	((',' (First, Second), CP), ...)	Σ	$ctrl$
...			

```
   Succeeds.

','(foo(X), call(X)).
   [It may succeed, or fail, or be an error]
   Undefined.

','(X=foo, call(X)).
   [Equivalent to unifying X with foo, and
     executing the goal 'foo'.]

','(X = true, call(X)).
   Succeeds, unifying X with true.
```

7.8.6 ';'/2 – disjunction

A control construct which is a disjunction of two activators $either$ and or where $either$ is an if-then control construct is an if-then-else control construct and its execution is described in 7.8.8.1.

7.8.6.1 Description

`';'(Either, Or)` is true iff `Either` is true or `Or` is true.

Procedurally, a disjunction of two activators $either$ and or, denoted by `';'(Either, Or)`, shall be executed as follows:

Table 27 — Before executing a disjunction

S_index	Decorated subgoal stack	Substitution	BI
N	((';' (Either, Or), CP), ...)	Σ	$ctrl$
...			

a) Make two copies $CCS1$ and $CCS2$ of $currstate$.

b) Set BI of $CCS1$ and $CCS2$ to nil.

c) Replace the current activator $curract$ of $CCG2$ by or.

d) Push $CCG2$ on to S.

e) Replace the current activator $curract$ of $CCG1$ by $either$.

f) Push $CCG1$ on to S.

g) Continue execution at 7.7.7.

h) Pop $currstate$ from S.

i) Continue execution at 7.7.8.

`';'(Either, Or)` is re-executable. On backtracking, continue at 7.8.6.1h.

NOTES

1 `';'` is a predefined infix operator.

2 Step 7.8.6.1f makes is now a $choicepoint$ of execution state $CCG1$.

3 Executing a disjunction has the effect that:

a) If $either$ should fail, then or will be executed on backtracking, and

b) Disjunction is transparent to cut because the $cutparent$ for $either$ and or are the same as that for the disjunction.

7.8.6.2 Examples

Tables 27 and 28 show the execution stack before and after executing the control construct `';'(Either, Or)`.

Table 28 — After executing a disjunction

S_index	Decorated subgoal stack	Substitution	BI
$N+2$	((Either, CP), ...)	Σ	nil
$N+1$	((Or, CP), ...)	Σ	nil
N	((';'(Either, Or), CP), ...)	Σ	$ctrl$
...			

```
';'(true, fail).
   Succeeds.

';'((!, fail), true).
   Fails.
   [Equivalent to (!, fail).]

';'(!, call(3)).
   Succeeds.
   [Equivalent to !.]

';'((X = 1, !), X = 2).
   Succeeds, unifying X with 1.
```

7.8.7 '-¿'/2 – if-then

7.8.7.1 Description

'->'(If, Then) is true iff If and Then are true.

Procedurally, a control construct if-then of two activators if and $then$, denoted by '->'(If, Then), shall be executed as follows:

a) Make a copy CCS of $currstate$.

b) Set BI of CCS to nil.

c) Pop $currdecsgl$ (= ('->'(If, Then), CP)) from $currentgoal$ of CCS.

d) Push ($then$, CP) on to $currentgoal$ of CCS.

e) Push (!, $N-1$) on to $currentgoal$ of CCS.

f) Push (if, $N-1$) on to $currentgoal$ of CCS.

g) Push CCS on to S.

Table 29 — Before executing an if-then

S_index	Decorated subgoal stack	Substitution	BI
N	(('->'(If, Then), CP), ...)	Σ	$ctrl$
...			

h) Continue execution at 7.7.7.

i) Pop $currstate$ from S.

j) Continue execution at 7.7.8.

'->'(If, Then) is re-executable. On backtracking, continue at 7.8.7.1i.

NOTES

1 -> is a predefined infix operator.

2 Executing an if-then has the effect that:

a) If if should fail, then the if-then will fail, and

b) If if should succeed, then $then$ will be executed, and

c) If if should succeed and $then$ later fails, the if will not be re-executed because of the cut which has been executed, and

d) The if in an if-then is not transparent to cut because the $cutparent$ for if is the $choicepoint$ for the if-then conditional.

e) A cut in $then$ is transparent to if-then because its $cutparent$ is the same as that for the if-then.

7.8.7.2 Examples

Tables 29 and 30 show the execution stack before and after executing the control construct '->'(If, Then).

```
'->'(true, true).
   Succeeds.

'->'(true, fail).
   Fails.

'->'(fail, true).
   Fails.

'->'(true, X=1).
   Succeeds, unifying X with 1.
   On re-execution, fails.

'->'(';'(X=1, X=2)), true.
```

Table 30 — After executing an if-then

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((If, $N-1$), (!, $N-1$), (Then, CP), ...)	Σ	nil
N	(('->'(If, Then), CP), ...)	Σ	$ctrl$
...			

```
   Succeeds, unifying X with 1.
   On re-execution, fails.

'->'(true, ';'(X=1, X=2)).
   Succeeds, unifying X with 1.
   On re-execution, succeeds, unifying X with 2.
   On re-execution, fails.
```

7.8.8 ';'/2 – if-then-else

NOTE — `;/2` serves two different functions depending on whether or not the first argument is a compound term with functor `->/2`.

See (7.8.6) for the use of `;/2` for disjunctive goals, that is when the first argument of `';'(_, _)` does not unify with `'->'(_, _)`.

7.8.8.1 Description

`';'('->'(If, Then), Else)` is true iff (1) `If` and `Then` are true, or (2) `If` is false and `Else` is true.

Procedurally, a control construct if-then-else of three activators if, $then$ and $else$, denoted by `';'('->'(If, Then), Else)`, shall be executed as follows:

a) Make a copy CCS of $currstate$.

b) Set BI of CCS to nil.

c) Pop $currdecsgl$ (`';'('->'(If, Then), Else)`, CP) from $currentgoal$ of CCS.

d) Push ($then$, CP) on to $currentgoal$ of CCS.

e) Push (!, $N-1$) on to $currentgoal$ of CCS.

f) Push (if, N) on to $currentgoal$ of CCS.

g) Push CCS on to S.

h) Continue execution at 7.7.7.

i) Make a copy CCS of $currstate$.

j) Set BI of CCS to nil.

k) Pop $currdecsgl$ (`';'('->'(If, Then), Else)`, CP) from $currentgoal$ of CCS.

l) Push ($else$, CP) on to $currentgoal$ of CCS.

m) Push (!, $N-1$) on to $currentgoal$ of CCS.

n) Push CCS on to S.

o) Continue execution at 7.7.7.

`';'('->'(If, Then), Else)` is re-executable. On backtracking, continue at 7.8.8.1i.

The cut prevents an if-then-else from being re-executed a second time.

NOTES

1 `';'` and `'->'` are predefined infix operators so that
`(If -> Then ; Else)`
is parsed as
`';'('->'(If, Then), Else)`

2 Executing an if-then-else has the effect that:

a) If if should fail, then the if-then-else will fail, and

b) If if should succeed, then $then$ will be executed, and

c) If if should succeed and $then$ later fails, the if-then-else will not be re-executed because of the cut which has been executed, and

d) The if in an if-then-else is not transparent to cut because the $cutparent$ for if is the S_index for the if-then-else.

e) A cut in $then$ is transparent to $then$ because its $cutparent$ is the $cutparent$ for the if-then-else.

3 Re-executing an if-then-else has the effect that:

a) After if fails and the if-then-else is re-executed, the $else$ will be executed, and

b) If $else$ later fails, the if-then-else will not be re-executed again because of the cut which has been executed, and

c) A cut in $else$ is transparent to $else$ because its $cutparent$ is the $cutparent$ for the if-then-else.

Table 31 — Before executing an if-then-else

S_index	Decorated subgoal stack	Substitution	BI
N	((';'('->'(If, Then), Else), CP), ...)	Σ	$ctrl$
...			

Table 32 — After executing an if-then-else

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((If, N), (!, $N-1$), (Then, CP), ...)	Σ	nil
N	((';'('->'(If, Then), Else), CP), ...)	Σ	$ctrl$
...			

7.8.8.2 Examples

Tables 31 and 32 show the execution stack before and after executing the control construct `';'('->'(If, Then), Else)`.

Table 33 shows what happens after `(';'('->'(If, Then), Else)` is re-executed because `If` failed.

```
';'('->'(true, true), fail).
  Succeeds.

';'('->'(fail, true), true).
  Succeeds.

';'('->'(true, fail), fail).
  Fails.

';'('->'(fail, true), fail).
  Fails.

';'('->'(true, X=1), X=2).
  Succeeds, unifying X with 1.
  On re-execution, fails.

';'('->'(false, X = 1), X = 2).
  Succeeds, unifying X with 2.

';'('->'(';'(X=1, X=2), true), true).
  Succeeds, unifying X with 1.
```

Table 33 — After re-executing an if-then-else because `If` failed.

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((!, $N-1$), (else(W), CP), ...)	Σ	nil
N	((';'('->'(If, Then), Else), CP), ...)	Σ	$ctrl$
...			

```
On re-execution, succeeds, unifying X with 2.
```

7.8.9 catch/3

The `catch` and `throw` control constructs enable a program to continue execution after an error without intervention from the user.

`catch(Goal, Catcher, Recovery)` is similar to `call(Goal)`, however when `throw(Ball)` is called, the current flow of control through predicates is interrupted, and control returns to a call of `catch/3` that is being executed. This can happen in one of two ways:

a) Implicitly, when one or more of the error conditions for a built-in predicate are satisfied, and

b) Explicitly, when the program executes a call of `throw/1` because the program wishes to abandon the current processing, and instead to take corrective action.

NOTES

1 The names of the arguments have been chosen because `throw/1` behaves as though it is throwing a ball to be caught by an active call of `catch/3`.

2 There are several advantages for this method of error recovery:

a) The programmer can localise such code at points where it is convenient,

b) The trap is placed round a goal, rather than being simply switched on by asserting clauses into an error handler. Thus there is much less chance of a program looping because unanticipated errors are trapped,

c) Unforeseen errors in an application embedded in Prolog need no longer suddenly print Prolog error messages and diagnostics to a mystified user.

3 One use of this mechanism is error handling. Typically a simple interactive program might have a top level looking something like:

```
main :-
   repeat,
   catch(run, Fault, recover(Fault)),
   fail.
```

7.8.9.1 Description

`catch(G, C, R)` is true iff `call(G)` is true, or the call of `G` is interrupted by a call of `throw/1` whose argument unifies with `C`, and `R` is true.

Procedurally, a control construct catch, denoted by `catch(G, C, R)`, shall be executed as follows:

a) Make a copy CCS of $currstate$.

b) Replace $curract$ of CCS by `call(G)`.

c) Set BI to nil.

d) Push CCS on to S.

e) Continue execution at 7.7.7.

f) Pop $currstate$ from S.

g) Continue execution at 7.7.8.

`catch(G, C, R)` is re-executable. On backtracking, continue at 7.8.9.1f.

7.8.9.2 Errors

a) `G` is a variable
— `instantiation_error`.

b) `G` is not a callable term
— `type_error(callable, G)`.

7.8.9.3 Examples

Tables 34 and 35 show the execution stack before and after executing the control construct `catch(G, C, R)`.

The following examples assume the database contains the following clauses:

```
foo(X) :-
   Y is X * 2, throw(test(Y)).
```

Table 34 — Before executing `catch(G, C, R)`

S_index	Decorated subgoal stack	Substitution	BI
N	((`catch(G, C, R)`, CP), ...)	Σ	$ctrl$
...			

Table 35 — After executing `catch(G, C, R)`

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((`call(G)`, CP), ...)	Σ	nil
N	((`catch(G, C, R)`, CP), ...)	Σ	$ctrl$
...			

```
bar(X) :-
   X = Y, throw(Y).

coo(X) :-
   throw(X).

car(X) :-
   X = 1, throw(X).


catch(foo(5), test(Y), true).
   Succeeds, unifying Y with 10.

catch(bar(3), Z, true).
   Succeeds, unifying Z with 3.

catch(true, C, write(demoen)), throw(bla).
   system_error.

catch(coo(X), Y, true).
   Succeeds, unifying Y with a renamed copy of X.

catch(car(X), Y, true).
   Succeeds, unifying Y with 1.
```

7.8.10 throw/1

7.8.10.1 Description

`throw(B)` is a control construct that is neither true nor false. It exists only for its procedural effect of causing the normal flow of control to be transferred back to an existing call of `catch/3` (see 7.8.9).

Procedurally, a control construct throw, denoted by `throw(B)`, shall be executed as follows:

a) Make a copy CA of $curract$, and a copy CP of $cutparent$.

b) Pop $currstate$ from S.

c) It shall be a system error (7.12.2j) if S is now empty,

d) Else if (1) the new $curract$ is a call of the control construct `catch/3`, and (2) the argument of CA unifies with the second argument `C` of the catch with most general unifier MGU, and (3) the $cutparent$ for the new $curract$ is less than CP, then continue at 7.8.10.1f.

e) Else replace CP by the $cutparent$ for the new $curract$, and continue at 7.8.10.1b.

f) Apply MGU to $currentgoal$.

g) Replace $curract$ by `call(R)`.

h) Set BI to nil.

i) Continue execution at 7.7.7.

NOTE — Executing a catch and throw has the effect that:

a) A catch is initially the same as a call of its first argument, and

b) A throw (or error) is like a cut, and no attempts are made to re-execute the goals on S between the throw and the first suitable catch, which is then replaced by a call of its third argument.

7.8.10.2 Errors

a) `B` does not unify with the `C` argument of any call of `catch/3`
— `system_error`.

7.8.10.3 Examples

Tables 36 and 37 show the execution stack before and after executing the control construct `throw(B)`, assuming where μ is the substitution which resulted from unifying `B` and `C`.

See also 7.8.9.3.

Table 36 — Before executing `throw(B)`

S_index	Decorated subgoal stack	Substitution	BI
$N+M$	((`throw(B)`, $CP2$), ...)	Σ	$ctrl$
...			
N	((`catch(G, C, R)`, $CP1$), ...)	σ	$ctrl$
...			

Table 37 — After executing `throw(B)`

S_index	Decorated subgoal stack	Substitution	BI
N	((`call(R)`, $CP1$), ...)	$\mu \circ \sigma$	nil
...			

7.9 Evaluating an expression

This clause defines the evaluation of a Prolog term as an expression.

7.9.1 A constant

The value of a constant `C` which is evaluated as an expression is `C`.

7.9.2 A compound term

The value R of a compound term `CT` which is evaluated as an expression is executed as follows:

a) If an argument of `CT` is a variable then it shall be an `instantiation_error`,

b) If argument `A` of `CT` is neither a variable nor the correct type `Type` then it shall be a `type_error(Type, A)`,

c) Evaluates each argument A_i of `CT` as an expression giving a value V_i,

d) Selects the operation F corresponding to the evaluable functor of `CT` and the types of V_i,

e) Computes the value R' of the operation F with operands V_i,

f) The value R of the expression is R'.

NOTE — The evaluable functors supported by this draft International Standard are defined in clause 9.

7.9.3 Order of evaluation of operands

At 7.9.2c a processor evaluates the arguments which are the operands of an evaluable functor in an implementation dependent order.

7.9.4 Exceptional values

An exceptional value is **overflow**, **underflow**, **zero_divisor**, or **undefined**. It shall be a `calculation_error(E)` if the result of an expression is `E`, one of those exceptional values.

7.9.5 Errors

The following errors may occur during the evaluation of an expression:

a) An argument is a variable
— `instantiation_error`.

b) An argument `X` is neither a variable nor an integer
— `type_error(integer, X)`.

c) An argument `X` is neither a variable nor a floating point value
— `type_error(real, X)`.

d) An argument `X` is neither a variable nor an integer nor a floating point value
— `type_error(number, X)`.

e) The result of computing the value of the operation is **overflow**
— `calculation_error(overflow)`.

f) The result of computing the value of the operation is **underflow**
— `calculation_error(underflow)`.

g) The result of computing the value of the operation is **zero_divisor**
— `calculation_error(zero_divide)`.

h) The result of computing the value of the operation is **undefined**
— `calculation_error(undefined)`.

NOTE — See clause 9.1.7 for examples of evaluating expressions.

7.10 Input/output

7.10.1 Sources and sinks

A source/sink (3.115) is a fundamental notion. A program can output results to a sink or input data from a source.

A source/sink always has a beginning, but has an end only if it is finite. In principle, there is a sense of location (stream position, see 7.10.2.8), though relocation to an arbitrary position may not be possible due to the nature of a particular source/sink.

A source/sink may be a file, the user's terminal, or other implementation defined possibility permitted by the processor.

Each source/sink is associated with a finite or potentially infinite sequence of bytes or characters.

A source/sink is specified as an implementation defined ground term in a call of `open/[3,4]` (8.11.5, 8.11.6). All subsequent references to the source/sink are made by referring to a stream identifier (7.10.2) or alias (7.10.2.2).

The effect of opening a source/sink more than once is undefined in this draft International Standard.

7.10.1.1 I/O modes

An I/O mode is an atom which defines in a call of `open/[3,4]` the I/O operations that may be performed on a source/sink. A processor shall support the I/O modes:

`read` — Input. The source/sink is a source. If it is a file, it shall already exist.

`write` — Output. The source/sink is a sink. If it is not empty, it shall be emptied.

`append` — Output. The source/sink is a sink. If the sink already exists then output shall start at the end of that sink, else an empty sink shall be created.

NOTES

1 If the sink is a file which already exists, and the I/O mode is `write`, the contents are lost.

2 A processor may support additional I/O modes, such as a mode for both reading and writing.

7.10.2 Streams

A stream is a connection to a source or sink.

7.10.2.1 Stream identifier

A stream identifier identifies a stream during a call of an input/output predicate. It is an implementation dependent ground term which is created as a result of opening a source/sink by a call of `open/[3,4]` (8.11.5, 8.11.6). A stream identifier shall not be an atom.

A standard-conforming program shall make no assumptions about the form of the stream identifier term, except that:

a) It is a ground term.

b) It is not an atom.

c) It uniquely identifies a particular stream during the time that the stream is open.

It is implementation dependent whether or not the processor uses the same stream identifier to represent different source/sinks at different times.

NOTE — A stream identifier is not an atom so that it can be distinguished from an alias.

7.10.2.2 Stream aliases

Any stream may be associated with a stream alias which is an atom which may be used to refer to that stream. The association is created when a stream is opened, and automatically ends when the stream is closed. A particular alias shall refer to at most one stream at any one time.

NOTES

1 A stream may be associated with more than one alias.

2 All built-in predicates which have a stream identifier as an input argument also accept a stream alias as that argument. However, built-in predicates which (can) return a stream identifier do not return or allow a stream alias. For example, a goal `current_input(some_alias)` can never succeed because `current_input/1` unifies its argument with a stream identifier.

7.10.2.3 Standard streams

Two streams are predefined and open during the execution of every goal: the standard input stream has the alias `user_input` and the standard output stream has the alias `user_output`.

The stream identifier for these streams shall be implementation dependent.

NOTES

1 Table 38 defines the properties of the standard streams.

2 A goal which attempts to close either standard stream succeeds, but does not close the stream (see 8.11.8).

7.10.2.4 Current streams

During execution there is a current input stream and a current output stream. When an input predicate does not have an explicit stream identifier argument, it shall read from the current input stream. Similarly, when an output predicate does not have an explicit stream identifier argument, it shall write to the current output stream.

By default, the current input and output streams are the same as the standard input and output streams, but the built-in predicates `set_input/1` and `set_output/1` can be used to change them.

When a stream which is the current input stream is closed, the standard input stream automatically becomes the current input stream. Since the standard input stream cannot be closed, this guarantees that the current input stream always refers to an open stream. Similarly for output.

7.10.2.5 Text streams

A text stream is a sequence of characters where each character is a member of C (7.1.4.1). A text stream is also regarded as a sequence of lines (7.10.2.7) where each line is a sequence of characters and terminated by a new line character (6.5.4).

`get_char/1` reads data from a text stream and returns a character. `get_code/1` reads data from a text stream and returns a character code.

A processor may add or remove space characters at the ends of lines in order to conform to the conventions for representing text files in the operating system. Any such alterations to the stream shall be implementation defined.

The effect of outputting a control character (6.4.2.1) to a text stream shall be implementation defined.

7.10.2.6 Binary streams

A binary stream is a sequence of bytes (7.1.2.1).

`get_char/1` and `get_code/1` read data from a binary stream and return a byte.

If data is output to a sink via a binary stream, and then read from that sink via a binary stream, then the data read shall be identical to that written, except that an implementation defined number of zero-valued bytes may be appended to the end of the data read back.

7.10.2.7 Lines and Records

Text streams are divided into lines.

It shall be implementation defined whether record-based streams, non-record-based streams, or both are supported.

Each line consists of a possibly empty sequence of characters followed by an implementation dependent new line character (6.5, 6.5.4).

It shall be implementation defined whether the last line in a text stream is followed by a new line character. If so, closing a stream which is a sink shall cause a new line character to be output if the stream does not already end with one.

NOTE — When a stream is connected to a record-based file, each record is regarded as a line during Prolog execution.

7.10.2.8 Stream positions

It shall be implementation defined whether or not any stream has a stream position. If it does, then:

a) A stream position is an implementation dependent ground term which identifies an absolute position of the source/sink to which the stream is connected.

b) At any time, the stream can be repositioned by calling `set_stream_position/2` (8.11.14).

A standard-conforming program shall make no assumption about the form of a stream position term, except that:

a) It is a ground term.

b) It uniquely identifies a particular position in source/sink to which the stream is connected during the time that the stream is open.

When an output stream is repositioned, it shall be implementation defined whether the remaining contents of the sink are discarded or will be overwritten, or will be appended to any output at that position.

When an input stream is repositioned, the contents of the stream shall be unaltered, and can be re-read.

7.10.2.9 End position of a stream

When all the characters in a stream `S` have been read (for example by `get_code`, `get_char`, or `read`) `S` has a stream position end-of-stream. At this stream position a goal to read more data returns a specific value to indicate that end of stream has been reached: `get_code` returns -1; `get_char` and `read` each return the atom `end_of_file`. When one of these terminating values has been read, the stream has a stream position past-end-of-stream.

When a stream has stream property `reposition(true)`, the terms `P` denoting stream positions end-of-stream and past-end-of-stream in stream property `position(P)` shall be implementation defined.

NOTE — A stream need not have an end, in which case its stream position is never end-of-stream or past-end-of-stream.

7.10.2.10 Flushing an output stream

Output to a stream may not be sent to the sink connected to that stream before completion of the goal which performs the output.

When it is necessary to be certain that output has been delivered, this can be done by executing the built-in predicate `flush_output/0` (8.11.9) or `flush_output/1` (8.11.10).

NOTES

1 Output is normally buffered, and `flush_output` will be necessary when, for example, the program has output a question which a user is required to answer.

2 A stream is always flushed when it is closed.

7.10.2.11 Options on stream creation

A stream-options list is a list of stream-options which define properties of a stream created with `open/4` (8.11.6).

The stream-options supported shall include:

`type(T)` — Specifies whether the stream is a text stream or a binary stream. `T` shall be:

`text` — the stream is a text stream, or

`binary` — the stream is a binary stream.

When no `type(T)` stream-option is specified, the stream shall be a text stream.

`reposition(Bool)` — If `Bool` is `true` then it shall be possible to reposition the stream, else if `Bool`

is `false` or no reposition argument is specified, it shall be implementation defined whether or not it is possible to reposition the stream.

`alias(A)` — Specifies that the atom `A` is to be an alias for the stream.

`eof_action(Action)` — The effect of attempting to read from a stream whose stream position is past-end-of-stream shall be specified by the value of the atom `Action`:

`error` — There shall be an Existence Error (7.12.2d) signifying that no more input exists in this stream.

`eof_code` — The result of a read shall be as if the stream position is end-of-stream (7.10.2.9).

`reset` — The stream position shall be reset so that it is not past-end-of-stream, and another attempt is made to read from it. This is likely to be useful when reading from a device such as a terminal. There may also be an implementation dependent operation to reset the source to which the stream is attached.

It shall be implementation defined which `eof_action` is the default.

If the stream-options list contains contradictory stream-options, the rightmost stream-option is the one which applies.

NOTES

1 It depends on the particular source/sink whether or not repositioning is possible, for example, it is impossible when the source/sink is a terminal.

2 It is an error when `reposition(true)` is specified for a particular source/sink and it is not possible.

7.10.2.12 Options on stream closure

A close-option modifies the behaviour of `close/2` (8.11.8) if an error condition is satisfied while trying to close a stream.

The close-options supported shall include:

`force(false)` — This is the default. If an error condition is satisfied, the stream is not closed.

`force(true)` — If a Resource Error condition (7.12.2h) or System Error condition (7.12.2j) is satisfied, there shall be no error; instead the stream is closed and the goal succeeds.

Table 38 — Properties of the standard streams

`user_input`	`user_output`
`mode(read)`	`mode(append)`
`input`	`output`
`alias(user_input)`	`alias(user_output)`
`eof_action(reset)`	`eof_action(reset)`
`reposition(false)`	`reposition(false)`
`type(text)`	`type(text)`

7.10.2.13 Stream properties

The properties of streams can be accessed via the predicate `stream_property(Stream, Property)` (8.11.11). The stream properties supported shall include:

`file_name(F)` — When the stream is connected to a source/sink which is a file, `F` shall be an implementation defined term which identifies the file which is the source/sink for the stream.

`mode(M)` — `M` is unified with the I/O mode (7.10.1.1 which was specified when the source/sink was opened.

`input` — This stream is connected to a source.

`output` — This stream is connected to a sink.

`alias(A)` — If the stream has an alias, then `A` shall be that alias.

`position(P)` — If the stream has a reposition property, `P` shall be the current stream position (7.10.2.8) of the stream.

`end_of_stream(E)` — If the stream position is end-of-stream then `E` is unified with `at` else if the stream position is past-end-of-stream then `E` is unified with `past` else `E` is unified with `no`.

`eof_action(A)` — If a stream-option (7.10.2.11) `eof_action(Action)` was specified when the stream was opened, then `A` is unified with `Action`, else `A` is unified with the implementation defined action which is associated with that stream.

`reposition(Bool)` — If repositioning is possible on this stream then `Bool` is unified with `true` else `Bool` is unified with `false`.

`type(T)` — The value of `T` defines whether the stream is a text stream (`T == text`) or a binary stream (`T == binary`).

Table 38 defines the properties of the standard streams.

7.10.3 Read-options list

A read-options list is a list of read-options which affects read_term/3 (8.14.2) and its related predicates. The read-options supported shall include:

variables(Vars) — After reading a term, Vars shall be a list of the variables in the term read, in left-to-right traversal order.

variable_names(VN_list) — After reading a term, VN_list shall be unified with a list of elements where: (1) each element is a term V = A, and (2) V is a named variable of the term, and (3) A is an atom whose characters are the characters of V.

singletons(VN_list) — After reading a term, VN_list shall be unified with a list of elements where: (1) each element is a term V = A, and (2) V is a named variable which occurs only once in the term, and (3) A is an atom whose characters are the characters of V.

If the read-options list contains contradictory read-options, the rightmost read-option is the one which applies.

NOTES

1 Anonymous variables (6.4.3) are included in a list Vars.

2 Anonymous variables are not included in a list VN_list.

7.10.4 Reading a term

The built-in predicate read_term/3 (8.14.2) reads a sequence of characters from a text stream and parses them as a read-term (6.2.2). There shall be a Syntax Error (7.12.2i) if the characters which are read do not conform to this syntax.

The final character read shall be an end token, i.e. an unquoted '.' character.

If the value associated with the flag char_conversion (7.11.2.1) is on, and the character-conversion relation (see 8.14.15, 8.14.16) includes (In → Out), then an unquoted character (6.4.2.1) In read from the text stream will be replaced by a character Out.

7.10.5 Write-options list

A write-options list is a list of write-options which affects write_term/3 (8.14.6) and its related predicates. The write-options supported shall include:

quoted(Bool) — When Bool is true each atom and functor is quoted if this would be necessary for the term to be read as data by read/1.

ignore_ops(Bool) — When Bool is true each compound term is output in functional notation (6.3.3). Neither operator (6.3.4.3) notation nor list notation (6.3.5) is used when this write-option is in force.

numbervars(Bool) — When Bool is true a term of the form '$VAR'(N), where N is an integer, is output as a variable name consisting of an capital letter possibly followed by an integer. The capital letter is the (i+1)th letter of the alphabet, and the integer is j, where

```
i = N mod 26
j = N // 26
```

The integer j is omitted if it is zero. For example,

```
'$VAR'(0) is written as A
'$VAR'(1) is written as B
 ...
'$VAR'(25) is written as Z
'$VAR'(26) is written as A1
'$VAR'(27) is written as B1
 ...
```

If the write-options list contains contradictory write-options, the rightmost write-option is the one which applies.

NOTE — The current operators do not affect output when this write-option is effective.

7.10.6 Writing a term

When a term Term is output using write_term/3 (8.14.6) the action which is taken is defined by the first rule of those below which is applicable:

a) If Term is a variable, a character sequence representing that variable is output. The sequence begins with _ (underscore) and the remaining characters are implementation dependent. The same character sequence is used for each occurrence of a particular variable in Term. A different character sequence is used for each distinct variable in Term.

b) If Term is an integer with value N_1, a character sequence representing N_1 shall be output. The first character shall be - if the value of N_1 is negative. The other characters shall be a sequence of decimal digit chars (6.5.2). The first decimal digit char shall be 0 iff the value of Term is zero.

c) If Term is a float with value F_1, a character sequence representing F_1 shall be output. The first character shall be - if the value of F_1 is negative. The other characters shall be an implementation dependent

sequence of characters which conform to the syntax for floating point numbers (6.4.5).

If there is an effective write-option `quoted(true)`, then the characters output shall be such that if they form a number with value F_2 in a term input by `read/1`, then
$F_1 = F_2$

d) If `Term` is an atom then if (1) there is an effective write-option `quoted(true)` and (2) the sequence of characters forming the atom could not be read as a valid atom without quoting, then `Term` is output as a quoted token, else `Term` is output as the sequence of characters defined by the syntax for the atom (6.1.2b, 6.4.2).

e) If `Term` has the form `'$VAR'(N)` for some positive integer N, and there is an effective write-option `numbervars(true)`, a variable name as defined in clause 7.10.5 is output.

f) If `Term` has a principal functor which is not a current operator, or if there is an effective write-option `ignore_ops(true)`, then the term is output in canonical form, that is:

1) The atom of the principal functor is output.

2) `(` (open char) is output.

3) Each argument of the term is output by recursively applying these rules.

4) `,` (comma char) is output between each successive pair of arguments.

5) `)` (close char) is output.

g) If `Term` has the form `'.'(Head,Tail)` then `Term` is output using list notation, that is:

1) `[` (open list char) is output.

2) `Head` is output by recursively applying these rules.

3) If `Tail` has the form `'.'(H,T)` then `,` (comma char) is output, set `Head:=H`, `Tail:=T`, and goto (2).

4) If `Tail` is `[]` then a closing bracket `]` (close list char) is output,

5) Else a `|` (head tail separator char) is output, and then `Tail` is output by recursively applying these rules.

h) If `Term` has a principal functor which is an operator, and there is an effective write-option `ignore_ops(false)`, then the term is output in expression form, that is:

1) The atom of the principal functor is output in front of its argument (prefix operator), between its arguments (infix operator), or after its argument (postfix operator). In all cases, a space is output to separate an operator from its argument(s) if any ambiguity could otherwise arise.

2) Each argument of the term is output by recursively applying these rules. When an argument is itself to be output in expression form, it is preceded by `(` (open char) and followed by `)` (close char) if: (i) the principal functor is an operator whose priority is so high that the term could not be re-input correctly with same set of current operators, or (ii) the argument is an atom which is a current operator.

NOTE — For example, a processor may write the floating point number 1.5 as ”1.5” or ”1.5E+00” or ”0.15e1”.

7.11 Flags

A flag is an atom which is associated with a value that is either implementation defined or defined by the user.

Each flag has a permitted range of values; any other value is a Domain Error (7.12.2c). The range of values associated with some flags can be extended with additional implementation defined values.

A processor can include additional implementation defined flags.

The definition of each flag indicates whether or not its value is changeable during execution.

NOTE — A built-in predicate `current_prolog_flag(Flag, Value)` (8.17.2) enables a program to discover all the flags supported by a processor and their current values.

A built-in predicate `set_prolog_flag(Flag, Value)` (8.17.1) enables a program to change the current value of those flags whose values are changeable. It shall be a Domain Error if there is an attempt to change the value of a flag whose value is not changeable.

7.11.1 Flags defining integer type I

The properties of the arithmetic type I which are provided by the processor are available to the program as values associated with various flags.

Table 39 — Flags defining I parameters

Parameter	Flag
$bounded$	bounded
$minint$	min_integer
$maxint$	max_integer

Table 40 — Further flags for I

Feature	Flag
rnd_I	integer_rounding_function

Table 39 identifies the parameters which define the integer type I (see 7.1.2) with the corresponding flags.

Table 40 identifies the LCAS integer rounding function (see 9.1.2.1) with the flag whose value indicate the precise methods adopted by the processor.

NOTE — The value of these flags is fixed and implementation defined. But it might be possible to set the values of some flags at the start of the program, for example, integer_rounding_function. This possibility would be an extension.

7.11.1.1 Flag: bounded

Possible value: true, false

Default value: implementation defined

Changeable: No

Description: If the value of this flag is true, integer arithmetic is performed correctly only if the operands and mathematically correct result all lie in the closed interval (min_integer, max_integer).

If the value of this flag is false, integer arithmetic is always performed correctly (except when there is a system_error), and a goal current_prolog_flag(max_integer, N) or current_prolog_flag(min_integer, N) will fail.

7.11.1.2 Flag: max_integer

Possible value: The default value only

Default value: implementation defined

Changeable: No

Description: If the value of flag bounded is true then the largest integer such that integer arithmetic is performed correctly if the operands and mathematically correct result all lie in the closed interval (min_integer, max_integer).

7.11.1.3 Flag: min_integer

Possible value: The default value only

Default value: implementation defined

Changeable: No

Description: If the value of flag bounded is true then the smallest integer such that integer arithmetic is performed correctly if the operands and mathematically correct result all lie in the closed interval (min_integer, max_integer).

NOTE — The possible values are required to be -M or -(M+1) where M is the value of the flag max_integer.

7.11.1.4 Flag: integer_rounding_function

Possible values: down, toward_zero

Default value: implementation defined

Changeable: No

Description: The value of this flag determines the precise definition of integer division // /2 and integer remainder rem/2 (9.1.2.1).

7.11.2 Other flags

7.11.2.1 Flag: char_conversion

Possible values: on, off

Default: on

Changeable: Yes

Description: When the value is on, unquoted characters read as data are transformed as defined by the user defined predicate char_conversion/2 (8.14.15); when the value is off, characters read as data are not changed.

7.11.2.2 Flag: debug

Possible values: on, off

Default: `off`

Changeable: Yes

Description: When the value is `off`, predicates have the meaning defined by this draft International Standard; when the value is `on`, the effect of calling any predicate shall be implementation defined.

7.11.2.3 Flag: max_arity

Possible values: The default value only

Default value: implementation defined

Changeable: No

Description: The maximum arity allowed for any compound term, or `unbounded` when the processor has no limit for the number of arguments for a compound term.

7.11.2.4 Flag: undefined_predicate

Possible values: `error, fail, warning`

Default: `error`

Changeable: Yes

Description: Defines the effect of attempting to execute a procedure with no clauses defining it (see 7.7.7b).

7.12 Errors

An error is a special circumstance which causes the normal process of execution to be interrrupted.

The error conditions for each control construct and built-in predicate are specified in the clauses defining them.

Other error conditions are defined in this draft International Standard where it states: “It shall be an error if ...”.

When more than one error condition is satisfied, the error that is reported by the Prolog processor is implementation dependent.

NOTE — Errors may also occur if:

a) There is an attempt to execute a goal for which there is no procedure (see 7.7.7b, 7.11.2.4).

b) The processor is too small, or execution requires too many resources (see 7.12.2h).

c) Execution cannot be completed because of some event outside the Prolog processor, for example a disc crash or interrupt (see 7.12.2j).

d) The result of an evaluable functor is one of the exceptional values (7.9.4).

7.12.1 The effect of an error

When an error occurs, the current goal shall be replaced by a goal `throw(error(Error_term, Imp_def)` where:

`Error_term` — is a term that supplies information about the error, and

`Imp_def` — is an implementation defined term.

NOTE — This draft International Standard defines features for continuing execution in a manner specified by the user, see the control construct `catch/3` (7.8.9).

7.12.2 Error classification

Errors are classified according to the form of `Error_term`:

a) There shall be an Instantiation Error when an argument or one of its components is a variable. It has the form `instantiation_error`.

b) There shall be a Type Error when the type of an argument or one of its components is incorrect, but not a variable. It has the form `type_error(ValidType, Culprit)` where

```
ValidType ∈ {
  atom,
  body,
  callable,
  character,
  compound,
  constant,
  integer,
  list,
  number,
  variable
}.
```

and `Culprit` is the argument or one of its components which caused the error.

c) There shall be a Domain Error when the type of an argument is correct but the value is outside the domain for which the predicate is defined. It has the form `domain_error(ValidDomain, Culprit)` where

```
ValidDomain ∈ {
  character_code_list,
  character_list,
  close_option,
  flag_value,
  io_mode,
  not_less_than_zero,
  operator_priority,
  operator_specifier,
  prolog_flag,
  read_option,
  source_sink,
  stream_or_alias,
  stream_option,
  stream_position,
  write_option
}.
```

and `Culprit` is the argument or one of its components which caused the error.

d) There shall be an Existence Error when the object on which an operation is to be performed does not exist. It has the form `existence_error(ObjectType, Culprit)` where

```
ObjectType ∈ {
  operator,
  past_end_of_stream,
  procedure,
  static_procedure,
  source_sink,
  stream
},
```

and `Culprit` is the argument or one of its components which caused the error.

e) There shall be a Permission Error when it is not permitted to perform a specific operation. It has the form `permission_error(Operation, ObjectType, Culprit)` where

```
Operation ∈ {
  access_clause,
  create,
  input,
  modify,
  open,
  output,
  reposition
},
```

and `ObjectType` is defined in 7.12.2d, and `Culprit` is the argument or one of its components which caused the error.

f) There shall be a Representation Error when an implementation defined limit has been breached. It has the form `representation_error(Flag)` where

```
Flag ∈ {
  character,
  character_code,
  exceeded_max_arity,
  flag
}.
```

g) There shall be a Calculation Error when the operands of an evaluable functor result in the operation producing an exceptionable value (7.9.4). It has the form `calculation_error(Error)` where

```
Error ∈ {
  overflow,
  underflow,
  zero_divide,
  undefined,
}.
```

h) There shall be a Resource Error at any stage of execution when the processor has insufficient resources to complete execution. It has the form `resource_error(Resource)` where `Resource` is an implementation dependent term.

i) There shall be a Syntax Error when a sequence of characters which are being input as a read-term do not conform to the syntax. It has the form `syntax_error`.

j) There may be a System Error at any stage of execution. The conditions in which there shall be a system error, and the action taken by a processor after a system error are implementation dependent. It has the form `system_error`.

NOTES

1 A goal which causes an Instantiation Error might not be erroneous if it were possible to delay execution until the variables become instantiated.

2 A Type Error occurs when a value does not belong to one the types defined in the draft International Standard and a Domain Error occurs when the value is not an element of an implementation defined or implementation dependent set.

3 Most errors defined in this draft International Standard occur because the arguments of the goal fail to satisfy a particular condition; they are thus detected before execution of the goal begins, and no side effects will have taken place. The exceptional cases are: Syntax Errors, Resource Errors, and System Errors.

4 A Resource Error may happen for example when a calculation on unbounded integers has a result which is too large.

5 A System Error may happen for example (a) in interactions with the operating system (for example, a disc crash or interrupt), or (b) when a goal `throw(T)` has been executed and there is no active goal `catch/3`.

---

8 Built-in predicates

A built-in predicate is a procedure which is provided automatically by a standard-conforming processor.

NOTES

1 A built-in predicate is static, and its execution is described in clause 7.7.12.

2 Each built-in predicate described in clause 8.x.y is defined formally in clause A.5.x.y.

3 The use of any built-in predicate (and particularly those concerned with input/output – 8.11, 8.12, 8.13, 8.14) may cause a Resource Error (7.12.2h) because, for example, the program has opened too many streams, or a file or disk is full. The use of these predicates may also cause a System Error (7.12.2j) because the operating system is reporting a problem.

The precise reason for such errors, and the ways they can be circumvented cannot be specified in this draft International Standard.

8.1 The format of predicate definitions

These clauses define the format of the definitions of built-in predicates.

8.1.1 Description

The description of the built-in predicate is in two parts: firstly logically defining the condition under which it is true, and secondly procedurally describing what happens as a goal is executed and whether the goal succeeds or fails.

Most built-in predicates are not re-executable; the description mentions the exceptional cases explicitly.

8.1.2 Template and modes

A specification for both the type of arguments and which of them shall be instantiated for the built-in predicate to be satisfied. The cases form a mutually exclusive set.

When appropriate, a “Template and modes” clause includes a note that the predicate is a predefined operator (see 6.3.4.4, table 5).

8.1.2.1 Type of an argument

The type of each argument is defined by one of the following atoms:

`atom` — as terminology,

`atom_or_atom_list` — an atom or a list of atoms,

`body` — as terminology,

`callable_term` — as terminology,

`character` — as terminology,

`character_code` — a character code (7.1.2.2),

`character_list` — a list of characters,

`clause` — as terminology,

`close_options` — a list of close options (8.11.8),

`compound_term` — as terminology,

`constant` — as terminology,

`flag` — an atom whose identifier is that of a Prolog flag (see 7.11),

`head` — as terminology,

`integer` — an integer,

`io_mode` — an I/O mode (7.10.1.1),

`io_options` — a list of I/O options (7.10.2.11),

`operator_specifier` — one of the atoms: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `fx`, `fy`

`list` — as terminology,

`nonvar` — a constant or compound term,

`number` — as terminology,

`predicate_indicator` — as terminology,

`read_options_list` — a read-options list (7.10.3),

`source_sink` — as terminology,

`stream` — as terminology,

`stream_or_alias` — a stream or an alias (7.10.2.2),

`stream_position` — a stream position (7.10.2.8),

`stream_property` — a stream property (7.10.2.13),

`term` — as terminology,

`write_options_list` — a write-options list (7.10.5).

8.1.2.2 Mode of an argument

The mode of each argument defines whether or not an argument shall be instantiated when the built-in predicate is executed. The mode is one of the following atoms:

`+` — the argument shall be instantiated,

`?` — the argument shall be instantiated or a variable,

`@` — the argument shall remain unaltered,

`-` — the argument shall be a variable that will be instantiated iff the predicate succeeds.

NOTES

1 When the type of an argument is `term`, the argument can be any term and there will be no error.

2 When the argument is a constant, there is no difference between the modes `+` and `@`. The mode `@` is therefore used only when the argument may be a compound term.

8.1.3 Errors

A list of the error conditions and and associated error term for the built-in predicate.

NOTE — The effect of an error condition being satisfied is defined in clause 7.12.

8.1.4 Examples

An example is normally the built-in predicate used as a goal, together with a statement saying whether the goal succeeds or fails or there is an error. The statement also describes any side effect and unification that occurs.

Sometimes the examples start by defining an environment in which it is assumed the goal appears.

8.2 Term unification

These predicates are concerned with the unification of two terms as defined in 7.3.

8.2.1 =/2 – Prolog unify

8.2.1.1 Description

If `X` and `Y` are $NSTO$ (7.3.3) then `'='(X, Y)` is true iff `X` and `Y` are unifiable (7.3).

Procedurally, `'='(X, Y)` is executed as follows:

a) If the two terms `X` and `Y` are STO (7.3.3), is undefined,

b) Else if the two terms `X` and `Y` are $NSTO$ and unifiable, computes and applies a most general unifier of `X` and `Y`, and succeeds,

c) Else if the two terms `X` and `Y` are $NSTO$ and not unifiable, fails.

NOTE — When unification involves infinite trees either as arguments or as a partial result of some top-down unification algorithm, the behaviour is undefined. A standard-conforming processor might consistently succeed or fail for a unification that is formally undefined by this draft International Standard.

This predicate can apparently be implemented much more efficiently than `unify_with_occurs_check(X, Y)` and in practice it is easy for programmers to avoid accidental use of the undefined cases.

8.2.1.2 Template and modes

```
'='(?term, ?term)
```

NOTE — `=` is a predefined infix operator (see 6.3.4.4).

8.2.1.3 Errors

None.

8.2.1.4 Examples

```
'='(1, 1).
   Succeeds.

'='(X, 1).
   Succeeds, unifying X with 1.

'='(X, Y).
   Succeeds, unifying X with Y.

'='(_, _).
   Succeeds.

'='(X, Y), '='(X, abc).
   Succeeds, unifying X with abc, and Y with abc.

'='(f(X, def), f(def, Y)).
   Succeeds, unifying X with def, and Y with def.
```

```
'='(1, 2).
  Fails.

'='(1, 1.0).
  Fails.

'='( g(X),
    f(f(X)) ).
  Fails.

'='( f(X,  1),
    f(a(X) ) ).
  Fails.

'='( f(X,   Y,   X  ),
    f(a(X), a(Y), Y, 2) ).
  Fails.

'='( X,
    a(X) ).
  Undefined.

'='( f(X,   1),
    f(a(X), 2) ).
  Undefined.

'='( f(1, X,   1),
    f(2, a(X), 2) ).
  Undefined.

'='( f(1, X  ),
    f(2, a(X)) ).
  Undefined.

'='( f(X,   Y,   X, 1),
    f(a(X), a(Y), Y, 2) ).
  Undefined.
```

8.2.2 unify_with_occurs_check/2 – unify

`unify_with_occurs_check(X, Y)` attempts to compute and apply a most general unifier of the two terms `X` and `Y`.

8.2.2.1 Description

`unify_with_occurs_check(X, Y)` is true iff `X` and `Y` are unifiable (7.3).

Procedurally, `unify_with_occurs_check(X, Y)` is executed as follows:

a) If `X` and `Y` are unifiable, computes and applies a most general unifier of `X` and `Y`, and succeeds.

b) Else if `X` and `Y` are not unifiable, fails.

NOTE — For any arguments `unify_with_occurs_check(X, Y)` always succeeds or fails; there is never an error or an undefined result.

This predicate can apparently be implemented much less efficiently than `=/2`, and in practice it is easy for programmers to avoid accidental use of the undefined cases.

8.2.2.2 Template and modes

```
unify_with_occurs_check(?term, ?term)
```

8.2.2.3 Errors

None.

8.2.2.4 Examples

```
unify_with_occurs_check(1, 1).
  Succeeds.

unify_with_occurs_check(X, 1).
  Succeeds, unifying X with 1.

unify_with_occurs_check(X, Y).
  Succeeds, unifying X with Y.

unify_with_occurs_check(_, _).
  Succeeds.

unify_with_occurs_check(X, Y),
     unify_with_occurs_check(X, abc).
  Succeeds, unifying X with abc, and Y with abc.

unify_with_occurs_check(f(X, def), f(def, Y)).
  Succeeds, unifying X with def, and Y with def.

unify_with_occurs_check(1, 2).
  Fails.

unify_with_occurs_check(1, 1.0).
  Fails.

unify_with_occurs_check( g(X),
                    f(f(X)) ).
  Fails.

unify_with_occurs_check( f(X,  1),
                    f(a(X) ) ).
  Fails.

unify_with_occurs_check( f(X,   Y,   X  ),
                    f(a(X), a(Y), Y, 2) ).
  Fails.

unify_with_occurs_check( X,
                    a(X) ).
  Fails.

unify_with_occurs_check( f(X,   1),
                    f(a(X), 2) ).
  Fails.

unify_with_occurs_check( f(1, X,   1),
                    f(2, a(X), 2) ).
  Fails.

unify_with_occurs_check( f(1, X),
                    f(2, a(X)) ).
  Fails.
```

```
unify_with_occurs_check( f(X,    Y,    X, 1),
                         f(a(X), a(Y), Y, 2) ).
   Fails.
```

8.2.3 \=/2 – not Prolog unifiable

8.2.3.1 Description

If X and Y are $NSTO$ (7.3.3) then \=(X, Y) is true iff X and Y are not unifiable (7.3).

Procedurally, \=(X, Y) is executed as follows:

a) If the two terms X and Y are STO, is undefined,

b) Else if the two terms X and Y are $NSTO$ and unifiable, fails,

c) Else if the two terms X and Y are $NSTO$ and not unifiable, succeeds.

NOTE — The predicate cannot be used to generate solutions.

8.2.3.2 Template and modes

'\\='(@term, @term)

NOTES

1 \= is a predefined infix operator (see 6.3.4.4).

2 The quoted atom '\\=' is identical to the unquoted atom \= (see 6.4.2.1).

8.2.3.3 Errors

None.

8.2.3.4 Examples

```
'\\='(1, 1).
   Fails.

\=(X, 1).
   Fails.

'\\='(X, Y).
   Fails.

\=(_, _).
   Fails.

'\\='(X, Y), '\\='(X, abc).
   Fails.

\=(f(X, def), f(def, Y)).
   Fails.

'\\='(1, 2).
   Succeeds.

\=(1, 1.0).
   Succeeds.

'\\='( g(X),
       f(f(X)) ).
   Succeeds.

\=( f(X,  1),
    f(a(X) ) ).
   Succeeds.

'\\='( f(X,    Y,    X   ),
       f(a(X), a(Y), Y, 2) ).
   Succeeds.

\=( X,
    a(X) ).
   Undefined.

'\\='( f(X,    1),
       f(a(X), 2) ).
   Undefined.

'\\='( f(1, X,    1),
       f(2, a(X), 2) ).
   Undefined.

\=( f(2, X),
    f(2, a(X)) ).
   Undefined.

'\\='( f(X,    Y,    X, 1),
       f(a(X), a(Y), Y, 2) ).
   Undefined.
```

8.3 Type testing

These predicates test the type associated with a term as defined in 7.1.

Each of these predicates simply succeeds or fails; there is no side effect, substitution, or error.

8.3.1 var/1

8.3.1.1 Description

var(X) is true iff X is an element of the set V (7.1.1).

8.3.1.2 Template and modes

var(@term)

8.3.1.3 Errors

None.

8.3.1.4 Examples

```
var(foo).
   Fails.

var(Foo).
   Succeeds.

foo=Foo, var(Foo).
   Fails.

var(_).
   Succeeds.
```

8.3.2 atom/1

8.3.2.1 Description

`atom(X)` is true iff `X` is an element of the set A (7.1.4).

8.3.2.2 Template and modes

`atom(@term)`

8.3.2.3 Errors

None.

8.3.2.4 Examples

```
atom(atom).
   Succeeds.

atom('string').
   Succeeds.

atom(a(b)).
   Fails.

atom(Var).
   Fails.

atom([]).
   Succeeds.

atom(6).
   Fails.

atom(3.3).
   Fails.
```

8.3.3 integer/1

8.3.3.1 Description

`integer(X)` is true iff `X` is an element of the set I (7.1.2).

8.3.3.2 Template and modes

`integer(@term)`

8.3.3.3 Errors

None.

8.3.3.4 Examples

```
integer(3).
   Succeeds.

integer(-3).
   Succeeds.

integer(3.3).
   Fails.

integer(X).
   Fails.

integer(atom).
   Fails.
```

8.3.4 real/1

8.3.4.1 Description

`real(X)` is true iff `X` is an element of the set F (7.1.3).

8.3.4.2 Template and modes

`real(@term)`

8.3.4.3 Errors

None.

8.3.4.4 Examples

```
real(3.3).
   Succeeds.

real(-3.3).
   Succeeds.
```

```
real(3).
   Fails.

real(atom).
   Fails.

real(X).
   Fails.
```

8.3.5 atomic/1

8.3.5.1 Description

`atomic(X)` is true iff `X` is an element of the set A (7.1.4), or the set I (7.1.2), or the set F (7.1.3).

8.3.5.2 Template and modes

`atomic(@term)`

8.3.5.3 Errors

None.

8.3.5.4 Examples

```
atomic(atom).
   Succeeds.

atomic(a(b)).
   Fails.

atomic(Var).
   Fails.

atomic(6).
   Succeeds.

atomic(3.3).
   Succeeds.
```

8.3.6 compound/1

8.3.6.1 Description

`compound(X)` is true iff `X` is an element of the set CT (7.1.5).

8.3.6.2 Template and modes

`compound(@term)`

8.3.6.3 Errors

None.

8.3.6.4 Examples

```
compound(33.3).
   Fails.

compound(-33.3).
   Fails.

compound(-a).
   Succeeds.

compound(_).
   Fails.

compound(a).
   Fails.

compound(a(b)).
   Succeeds.

compound([a]).
   Succeeds.
```

8.3.7 nonvar/1

8.3.7.1 Description

`nonvar(X)` is true iff `X` is not an element of the set V (7.1.1).

8.3.7.2 Template and modes

`nonvar(@term)`

8.3.7.3 Errors

None.

8.3.7.4 Examples

```
nonvar(33.3).
   Succeeds.

nonvar(foo).
   Succeeds.

nonvar(Foo).
   Fails.

foo = Foo, nonvar(Foo).
   Succeeds.

nonvar(_).
   Fails.
```

```
nonvar(a(b)).
   Succeeds.
```

8.3.8 number/1

8.3.8.1 Description

`number(X)` is true iff `X` is an element of the set I (7.1.2), or the set F (7.1.3).

8.3.8.2 Template and modes

`number(@term)`

8.3.8.3 Errors

None.

8.3.8.4 Examples

```
number(3).
   Succeeds.

number(3.3).
   Succeeds.

number(-3).
   Succeeds.

number(a).
   Fails.

number(X).
   Fails.
```

8.4 Term comparison

These predicates test the ordering of two terms as defined in 7.2.

Each of these predicates simply succeeds or fails; there is no side effect, substitution, or error.

8.4.1 ==/2 – identical

8.4.1.1 Description

`'=='(X, Y)` is true iff `X` and `Y` are identical terms (3.67).

Procedurally, `'=='(X, Y)` is executed as follows:

a) If `X` and `Y` are identical, the predicate succeeds.

b) Else the predicate fails.

8.4.1.2 Template and modes

`'=='(@term, @term)`

NOTE — `'=='` is a predefined infix operator (see 6.3.4.4).

8.4.1.3 Errors

None.

8.4.1.4 Examples

```
'=='(1, 1).
   Succeeds.

'=='(X, X).
   Succeeds.

'=='(1, 2).
   Fails.

'=='(X, 1).
   Fails.

'=='(X, Y).
   Fails.

'=='(_, 1).
   Fails.

'=='(_, _).
   Fails.

'=='(X, a(X)).
   Fails.
```

8.4.2 \==/2 – not identical

8.4.2.1 Description

`'\\=='(X, Y)` is true iff `X` and `Y` are not identical terms (3.67).

Procedurally, `'\\=='(X, Y)` is executed as follows:

a) If `X` and `Y` are identical, the predicate fails.

b) Else the predicate succeeds.

8.4.2.2 Template and modes

`'\\=='(@term, @term)`

NOTE — '\\==' is a predefined infix operator (see 6.3.4.4).

8.4.2.3 Errors

None.

8.4.2.4 Examples

```
'\\=='(1, 1).
   Fails.

\==(1, 2).
   Succeeds.

'\\=='(X, 1).
   Succeeds.

\==(_, _).
   Succeeds.

'\\=='(X, a(X)).
   Succeeds.
```

8.4.3 @¡/2 – term less than

8.4.3.1 Description

'@<'(X, Y) is true iff X *term_precedes* Y (7.2).

Procedurally, '@<'(X, Y) is executed as follows:

a) If X *term_precedes* Y, the predicate succeeds,

b) Else the predicate fails.

8.4.3.2 Template and modes

'@<'(@term, @term)

NOTE — '@<' is a predefined infix operator (see 6.3.4.4).

8.4.3.3 Errors

None.

8.4.3.4 Examples

```
'@<'(1.0, 1).
   Succeeds.

'@<'(aardvark, zebra).
   Succeeds.

'@<'(short, short).
   Fails.

'@<'(short, shorter).
   Succeeds.

'@<'(foo(b), foo(a)).
   Fails.

'@<'(foo(a, b), north(a)).
   Fails.

'@<'(X, X).
   Fails.

'@<'(foo(a, X), foo(b, Y)).
   Succeeds.

'@<'(X, Y).
   Implementation dependent.

'@<'(_, _).
   Implementation dependent.

'@<'(foo(X, a), foo(Y, b)).
   Implementation dependent.
```

8.4.4 @=¡/2 – term less than or equal

8.4.4.1 Description

'@=<'(X, Y) is true iff X *term_precedes* Y (7.2), or X and Y are identical terms (3.67).

Procedurally, '@=<'(X, Y) is executed as follows:

a) If X and Y are identical, the predicate succeeds.

b) Else if X *term_precedes* Y, the predicate succeeds,

c) Else the predicate fails.

8.4.4.2 Template and modes

'@=<'(@term, @term)

NOTE — '@=<' is a predefined infix operator (see 6.3.4.4).

8.4.4.3 Errors

None.

8.4.4.4 Examples

```
'@=<'(1.0, 1).
   Succeeds.

'@=<'(aardvark, zebra).
   Succeeds.
```

```
'@=<'(short, short).
  Succeeds.

'@=<'(short, shorter).
  Succeeds.

'@=<'(foo(b), foo(a)).
  Fails.

'@=<'(X, X).
  Succeeds.

'@=<'(foo(a, X), foo(b, Y)).
  Succeeds.

'@=<'(X, Y).
  Implementation dependent.

'@=<'(_, _).
  Implementation dependent.

'@=<'(foo(X, a), foo(Y, b)).
  Implementation dependent.
```

8.4.5 @¿/2 – term greater than

8.4.5.1 Description

`'@>'(X, Y)` is true iff `Y` $term\_precedes$ `X`. (7.2).

Procedurally, `'@>'(X, Y)` is executed as follows:

a) If `Y` $term\_precedes$ `X`, the predicate succeeds,

b) Else the predicate fails.

8.4.5.2 Template and modes

`'@>'(@term, @term)`

NOTE — `'@>'` is a predefined infix operator (see 6.3.4.4).

8.4.5.3 Errors

None.

8.4.5.4 Examples

```
'@>'(1.0, 1).
  Fails.

'@>'(aardvark, zebra).
  Fails.

'@>'(short, short).
  Fails.

'@>'(short, shorter).
  Fails.

'@>'(foo(b), foo(a)).
  Succeeds.

'@>'(X, X).
  Fails.

'@>'(foo(a, X), foo(b, Y)).
  Fails.

'@>'(X, Y).
  Implementation dependent.

'@>'(_, _).
  Implementation dependent.

'@>'(foo(X, a), foo(Y, b)).
  Implementation dependent.
```

8.4.6 @¿=/2 – term greater than or equal

8.4.6.1 Description

`'@>='(X, Y)` is true iff `Y` $term\_precedes$ `X` (7.2), or `X` and `Y` are identical terms (3.67).

Procedurally, `'@>='(X, Y)` is executed as follows:

a) If `X` and `Y` are identical, the predicate succeeds.

b) Else if `Y` $term\_precedes$ `X`, the predicate succeeds,

c) Else the predicate succeeds.

8.4.6.2 Template and modes

`'@>='(@term, @term)`

NOTE — `'@>='` is a predefined infix operator (see 6.3.4.4).

8.4.6.3 Errors

None.

8.4.6.4 Examples

```
'@>='(1.0, 1).
  Fails.

'@>='(aardvark, zebra).
  Fails.

'@>='(short, short).
  Succeeds.

'@>='(short, shorter).
  Fails.
```

```
'@>='(foo(b), foo(a)).
   Succeeds.

'@>='(X, X).
   Succeeds.

'@>='(foo(a, X), foo(b, Y)).
   Fails.

'@>='(X, Y).
   Implementation dependent.

'@>='(_, _).
   Implementation dependent.

'@>='(foo(X, a), foo(Y, b)).
   Implementation dependent.
```

8.5 Term creation and decomposition

These predicates enable a term to be assembled from its component parts, or split into its component parts, or copied.

8.5.1 functor/3

8.5.1.1 Description

`functor(Term, Name, Arity)` is true iff:

— `Term` is a compound term with a functor whose identifier is `Name` and arity `Arity`, or

— `Term` is a constant equal to `Name` and `Arity` is `0`.

Procedurally, When `functor(Term, Name, Arity)` succeeds, either:

a) `Term` is a constant, and `Name` is unified with `Term`, and `Arity` is unified with `0`, or

b) `Term` is a compound term, and `Name` is unified with the identifier of the functor of `Term`, and `Arity` is unified with the arity of the functor of `Term`, or

c) `Term` is a variable and `Name` is a constant and `Arity` is `0`, and `Term` is unified with `Name`, or

d) `Term` is a variable and `Name` is an atom and `Arity` is an integer greater than zero, and `Term` is unified with a term that has functor with identifier `Name` and arity `Arity`, and `Arity` distinct uninstantiated arguments.

8.5.1.2 Template and modes

```
functor(-nonvar, +constant, +integer)
functor(@nonvar, ?constant, ?integer)
```

8.5.1.3 Errors

a) `Term` and `Name` are both variables
— `instantiation_error`.

b) `Term` and `Arity` are both variables
— `instantiation_error`.

c) `Term` is a variable and `Name` neither a variable nor a constant
— `type_error(constant, Name)`.

d) `Term` is a variable and `Arity` neither a variable nor an integer
— `type_error(integer, Arity)`.

e) `Term` is a variable and `Arity` is an integer greater than the implementation defined constant `max_arity`
—
`representation_error(exceeded_max_arity)`.

f) `Term` is a variable and `Arity` is an integer that is less than zero
— `domain_error(not_less_than_zero, Arity)`.

8.5.1.4 Examples

```
functor(foo(a, b, c), foo, 3).
   Succeeds.

functor(foo(a, b, c), X, Y).
   Succeeds, unifying X with foo, and Y with 3.

functor(X, foo, 3).
   Succeeds, unifying X with foo(_, _, _).

functor(X, foo, 0).
   Succeeds, unifying X with foo.

functor(foo(a), foo, 2).
   Fails.

functor(foo(a), fo, 1).
   Fails.

functor(1, X, Y).
   Succeeds, unifying X with 1, and Y with 0.

functor(X, 1.1, 0).
   Succeeds, unifying X with 1.1.

functor(F, foo(a), 1).
   Fails.

functor([_|_], '.', 2).
```

```
   Succeeds.

functor([], [], 0).
   Succeeds.

functor(X, Y, 3).
   instantiation_error.

functor(X, foo, N).
   instantiation_error.

functor(X, foo, a).
   type_error(integer, a).

current_prolog_flag(max_arity, A),
     X is A + 1,
     functor(T, foo, X).
   representation_error(exceeded_max_arity).

Minus_1 is 0 - 1,
     functor(F, foo, Minus_1).
   domain_error(not_less_than_zero).
```

8.5.2 arg/3

8.5.2.1 Description

`arg(N, Term, Arg)` is true iff the `N-th` argument of `Term` is `Arg`.

Procedurally, `arg(N, Term, Arg)` is executed as follows:

a) unifies `Arg` with the `N`-th argument of compound term `Term` and succeeds, or

b) fails.

Arguments are numbered from 1.

8.5.2.2 Template and modes

```
arg(+integer, +compound_term, ?term)
```

8.5.2.3 Errors

a) `N` is a variable
— `instantiation_error`.

b) `Term` is a variable
— `instantiation_error`.

c) `Term` is neither a variable nor a compound term
— `type_error(compound, Term)`.

d) `N` is neither a variable nor an integer
— `type_error(integer, N)`.

8.5.2.4 Examples

```
arg(1, foo(a, b), a).
   Succeeds.

arg(1, foo(a, b), X).
   Succeeds, unifying X with a.

arg(1, foo(X, b), a).
   Succeeds, unifying X with a.

arg(1, foo(X, b), Y).
   Succeeds, unifying X with Y.

arg(1, foo(a, b), b).
   Fails.

arg(0, foo(a, b), foo).
   Fails.

arg(3, foo(3, 4), N).
   Fails.

arg(X, foo(a, b), a).
   instantiation_error.

arg(1, X, a).
   instantiation_error.

arg(0, atom, A).
   type_error(compound, atom).

arg(0, 3, A).
   type_error(compound, 3).

arg(1, foo(X), u(X)).
   Undefined.
```

8.5.3 =../2 – univ

8.5.3.1 Description

`'=..'(Term, List)` is true iff:

— `Term` is a constant and `List` is the list whose only element is `Term`, or

— `Term` is a compound term and `List` is the list whose head is the functor name of `Term` and whose tail is a list of the arguments of `Term`.

Procedurally, `'=..'(Term, List)` is executed as follows:

a) If `Term` is a constant then unifies `List` with a list whose only element is `Term`, or

b) If `Term` is a compound term then unifies `List` with a list whose head is the functor name of `Term` and whose tail is a list of the arguments of `Term`, or

c) If `Term` is a variable and `List` is a list whose

only element is a constant then unifies Term with the single element of List, or

d) If Term is a variable and List is a list and there exists a compound term CT such that the functor name of CT is the head of List and a list of the arguments of CT is the tail of List then unifies Term with CT and the predicate succeeds, or

e) The predicate fails.

8.5.3.2 Template and modes

```
'=..'(+nonvar, ?list)
'=..'(-nonvar, +list)
```

NOTE — '=..' is a predefined infix operator (see 6.3.4.4).

8.5.3.3 Errors

a) Term and List are variables
— instantiation_error.

b) Term is a variable and List is a partial list
— instantiation_error.

c) Term is a variable and List is neither a variable nor a partial list nor a list
— type_error(list, List).

d) Term is a variable and the head of List is a variable
— instantiation_error.

e) Term is a variable and the head of list List is a number and the tail of List is not the empty list
— type_error(atom, List).

f) Term is a variable and the head of list List is a compound term
— type_error(atom, List).

g) Term is a variable and the tail of List has a length greater than the implementation defined constant max_arity
—
representation_error(exceeded_max_arity).

8.5.3.4 Examples

```
'=..'(foo(a, b), [foo, a, b]).
   Succeeds.

'=..'(X, [foo, a, b]).
   Succeeds, unifying X with foo(a, b).

'=..'(foo(a, b), L).
   Succeeds, unifying L with [foo, a, b].

'=..'(foo(X, b), [foo, a, Y]).
   Succeeds, unifying X with a, and Y with b.

'=..'(1, [1]).
   Succeeds.

'=..'(foo(a, b), [foo, b, a]).
   Fails.

'=..'(X, Y).
   instantiation_error.

'=..'(X, [foo, a | Y]).
   instantiation_error.

'=..'(X, [foo|bar]).
   type_error(list, [foo|bar]).

'=..'(X, [Foo, bar]).
   instantiation_error.

'=..'(X, [3, 1]).
   type_error(atom, [3, 1]).

'=..'(X, [1.1, foo]).
   type_error(atom, [1.1, foo]).

'=..'(X, [a(b), 1]).
   type_error(atom, [a(b), 1]).

'=..'(X, 4).
   type_error(list, 4).

'=..'(f(X), [f, u(X)]).
   Undefined.
```

8.5.4 copy_term/2

8.5.4.1 Description

copy_term(Term_1, Term_2) is true iff Term_2 unifies with a term T which is a renamed copy (7.1.6.2) of Term_1.

Procedurally, copy_term(Term_1, Term_2) is executed as follows:

a) Let T be a renamed copy (7.1.6.2) of Term_1.

b) If Term_2 unifies with T, the predicate succeeds,

c) Else the predicate fails.

NOTE — If the variable sets of Term_1 and Term_2 are disjoint, then even if the predicate succeeds, Term_1 will be unaltered, and the variable sets of both arguments will remain disjoint.

8.5.4.2 Template and modes

```
copy_term(?term, ?term)
```

8.5.4.3 Errors

None.

8.5.4.4 Examples

NOTE — No unifications take place in the examples above unless explicitly described.

```
copy_term(X, Y).
   Succeeds.

copy_term(X, 3).
   Succeeds.

copy_term(_, a).
   Succeeds.

copy_term(a+X, X+b).
   Succeeds, unifying X with a.

copy_term(_, _).
   Succeeds.

copy_term(X+X+Y, A+B+B).
   Succeeds, unifying A with B.

copy_term(a, b).
   Fails.

copy_term(a+X, X+b),
     copy_term(a+X, X+b).
   Fails.

copy_term(demoen(X, X), demoen(Y, f(Y))).
   Undefined.
```

8.6 Arithmetic evaluation

This predicate causes an expression to be evaluated, and a result (defined by clause 9) to be unified with a term.

8.6.1 is/2 – evaluate expression

8.6.1.1 Description

`'is'(Result, Expression)` is true iff the result of evaluating `Expression` as an expression is `Result`.

Procedurally, `'is'(Result, Expression)` evaluates `Expression` and unifies `Result` with the resulting value.

8.6.1.2 Template and modes

```
'is'(?nonvar, +nonvar)
```

NOTE — `'is'` is a predefined infix operator (see 6.3.4.4).

8.6.1.3 Errors

a) `Expression` is a variable
— `instantiation_error`.

8.6.1.4 Examples

```
'is'(Result, 3+11.0).
   Succeeds, unifying Result with 14.0.

X = 1+2, Y is X * 3.
   Succeeds, unifying X with 1+2, and Y with 9.

'is'(foo, 77).
   Fails.

'is'(77, N).
   instantiation_error.
```

8.7 Arithmetic comparison

These predicates cause two expressions to be evaluated, and their results (defined by clause 9) to be compared.

8.7.1 Arithmetic comparison predicates and operations

Each arithmetic comparison predicate corresponds to an operation which depends on the types of the values which are obtained by evaluating the argument(s) of the predicate.

The following table identifies the integer or floating point operations corresponding to each predicate:

Predicate indicator	Operation
`=:= /2`	eq_I, eq_F, eq_{FI}, eq_{IF}
`=\= /2`	neq_I, neq_F, neq_{FI}, neq_{IF}
`< /2`	lss_I, lss_F, lss_{FI}, lss_{IF}
`=< /2`	leq_I, leq_F, leq_{FI}, leq_{IF}
`> /2`	gtr_I, gtr_F, gtr_{FI}, gtr_{IF}
`>= /2`	geq_I, geq_F, geq_{FI}, geq_{IF}

NOTE — The arithmetic evaluable functors are defined in 9.1.1.

8.7.2 Arithmetic comparison operations

The following operations are specified:

eq_F: $F \times F \to Boolean$
eq_I: $I \times I \to Boolean$
eq_{FI}: $F \times I \to Boolean \cup \{\mathbf{overflow}\}$
eq_{IF}: $I \times F \to Boolean \cup \{\mathbf{overflow}\}$
neq_F: $F \times F \to Boolean$
neq_I: $I \times I \to Boolean$
neq_{FI}: $F \times I \to Boolean \cup \{\mathbf{overflow}\}$
neq_{IF}: $I \times F \to Boolean \cup \{\mathbf{overflow}\}$
lss_F: $F \times F \to Boolean$
lss_I: $I \times I \to Boolean$
lss_{FI}: $F \times I \to Boolean \cup \{\mathbf{overflow}\}$
lss_{IF}: $I \times F \to Boolean \cup \{\mathbf{overflow}\}$
leq_F: $F \times F \to Boolean$
leq_I: $I \times I \to Boolean$
leq_{FI}: $F \times I \to Boolean \cup \{\mathbf{overflow}\}$
leq_{IF}: $I \times F \to Boolean \cup \{\mathbf{overflow}\}$
gtr_F: $F \times F \to Boolean$
gtr_I: $I \times I \to Boolean$
gtr_{FI}: $F \times I \to Boolean \cup \{\mathbf{overflow}\}$
gtr_{IF}: $I \times F \to Boolean \cup \{\mathbf{overflow}\}$
geq_F: $F \times F \to Boolean$
geq_I: $I \times I \to Boolean$
geq_{FI}: $F \times I \to Boolean \cup \{\mathbf{overflow}\}$
geq_{IF}: $I \times F \to Boolean \cup \{\mathbf{overflow}\}$

For all values x and y in F, and m and n in I the following axioms shall apply:

$$eq_F(x,y) = \mathbf{true} \iff x = y$$

$$eq_I(m,n) = \mathbf{true} \iff m = n$$

$$eq_{FI}(x,n) = eq_F(x, float_{I\to F}(n))$$

$$eq_{IF}(n,y) = eq_F(float_{I\to F}(n), y)$$

$$neq_F(x,y) = \mathbf{true} \iff x \neq y$$

$$neq_I(m,n) = \mathbf{true} \iff m \neq n$$

$$neq_{FI}(x,n) = neq_F(x, float_{I\to F}(n))$$

$$neq_{IF}(n,y) = neq_F(float_{I\to F}(n), y)$$

$$lss_F(x,y) = \mathbf{true} \iff x < y$$

$$lss_I(m,n) = \mathbf{true} \iff m < n$$

$$lss_{FI}(x,n) = lss_F(x, float_{I\to F}(n))$$

$$lss_{IF}(n,y) = lss_F(float_{I\to F}(n), y)$$

$$leq_F(x,y) = \mathbf{true} \iff x \leq y$$

$$leq_I(m,n) = \mathbf{true} \iff m \leq n$$

$$leq_{FI}(x,n) = leq_F(x, float_{I\to F}(n))$$

$$leq_{IF}(n,y) = leq_F(float_{I\to F}(n), y)$$

$$gtr_F(x,y) = \mathbf{true} \iff x > y$$

$$gtr_I(m,n) = \mathbf{true} \iff m > n$$

$$gtr_{FI}(x,n) = gtr_F(x, float_{I\to F}(n))$$

$$gtr_{IF}(n,y) = gtr_F(float_{I\to F}(n), y)$$

$$geq_F(x,y) = \mathbf{true} \iff x \geq y$$

$$geq_I(m,n) = \mathbf{true} \iff m \geq n$$

$$geq_{FI}(x,n) = geq_F(x, float_{I\to F}(n))$$

$$geq_{IF}(n,y) = geq_F(float_{I\to F}(n), y)$$

NOTE — It shall be an overflow error if an operand which is a large integer cannot be converted to an approximate floating point value (see $float_{I\to F}$, 9.1.5).

8.7.3 The arithmetic comparison predicates

The following requirements are true for all P where

$P \in \{$ `=:=`, `=\=`, `<`, `=<`, `>`, `>=` $\}$

8.7.3.1 Description

`'P'(E1, E2)` is true iff evaluating `E1` and `E2` as expressions and performing the corresponding arithmetic operation on the results is **true**.

Procedurally, `'P'(E1, E2)` is executed as follows:

a) Evaluates the two expressions `E1` and `E2`,

b) If the result of applying the arithmetic operation P to the resulting values is **true**, the predicate succeeds,

c) Else the predicate fails.

8.7.3.2 Template and modes

`'P'(+nonvar, +nonvar)`

NOTE — P is a predefined infix operator (see 6.3.4.4).

8.7.3.3 Errors

a) `E1` is a variable
— `instantiation_error`.

b) `E2` is a variable
— `instantiation_error`.

8.7.3.4 Examples

```
'=:='(0, 1).
   Fails.

'=\\='(0, 1).
   Succeeds.

'<'(0, 1).
   Succeeds.

'>'(0, 1).
   Fails.

'>='(0, 1).
   Fails.

'=<'(0, 1).
   Succeeds.

'=:='(1.0, 1).
   Succeeds.

=\=(1.0, 1).
   Fails.

'<'(1.0, 1).
   Fails.

'>'(1.0, 1).
   Fails.

'>='(1.0, 1).
   Succeeds.

'=<'(1.0, 1).
   Succeeds.

'=:='(3*2, 7-1).
   Succeeds.

'=\\='(3*2, 7-1).
   Fails.

'<'(3*2, 7-1).
   Fails.

'>'(3*2, 7-1).
   Fails.

'>='(3*2, 7-1).
   Succeeds.

'=<'(3*2, 7-1).
   Succeeds.

'=:='(X, 5).
   instantiation_error.

=\=(X, 5).
   instantiation_error.

'<'(X, 5).
   instantiation_error.

'>'(X, 5).
   instantiation_error.

'>='(X, 5).
   instantiation_error.

'=<'(X, 5).
   instantiation_error.
```

8.8 Clause retrieval and information

These predicates enable the contents of the database (7.5) to be inspected during resolution.

The examples provided for these built-in predicates assume the database has been created from the following Prolog text:

```
:- dynamic(cat/0).
cat.

:- dynamic(dog/0).
dog :- true.

elk(X) :- moose(X).

:- dynamic(legs/2).
legs(A, 6) :- insect(A).

:- dynamic(insect/1).
insect(ant).
insect(bee).
```

8.8.1 clause/2

8.8.1.1 Description

`clause(Head, Body)` is true iff:

— The predicate of `Head` is dynamic, and

— There is a clause in the database which corresponds to a term `H :- B` which unifies with `Head :- Body`.

Procedurally, `clause(Head, Body)` is executed as follows:

a) Searches sequentially through each dynamic user-defined procedure in the database and creates a list L of all the terms `clause(H, B)` such that

1) the database contains a clause whose head can be converted to a term `H`, and whose body coan be converted to a term `B`, and

2) `H` unifies with `Head`, and

3) `B` unifies with `Body`.

b) If a non-empty list is found, proceeds to 8.8.1.1d,

c) Else the predicate fails.

d) Chooses the first element of the list L, and the predicate succeeds.

e) If all the elements of the list L have been chosen, then the predicate fails,

f) Else chooses the first element of the list L which has not already been chosen, and the predicate succeeds.

`clause/2` is re-executable. On backtracking, continue at 8.8.1.1e.

8.8.1.2 Template and modes

```
clause(+head, ?body)
```

8.8.1.3 Errors

a) `Head` is a variable
— `instantiation_error`.

b) `Head` is not a predication
— `type_error(callable, Head)`.

c) The predicate indicator `Pred` of `Head` is not that of a dynamic procedure
— `permission_error(access_clause, static_procedure, Pred)`.

8.8.1.4 Examples

These examples assume the database has been created from the Prolog text defined at the beginning of 8.8.

```
clause(cat, true).
   Succeeds.

clause(dog, true).
   Succeeds.

clause(legs(I, N), Body).
   Succeeds, unifying N with 6,
   and Body with insect(I).

clause(insect(I), T).
   Succeeds, unifying I with ant, and T with true.
   On re-execution,
   succeeds, unifying I with bee, and T with true.

clause(x, Body).
   Fails.

clause(_, B).
   instantiation_error.

clause(4, X).
   type_error(callable, 4).

clause(elk(N), Body).
   permission_error(access_clause,
      static_procedure, elk/1).

clause(atom(_), Body).
   permission_error(access_clause,
      static_procedure, atom/1).

clause(legs(A, 6), insect(f(A))).
   Undefined.
```

8.8.2 current_predicate/1

8.8.2.1 Description

`current_predicate(PI)` is true iff `PI` is a predicate indicator for one of the user-defined procedures in the database.

Procedurally, `current_predicate(PI)` is executed as follows:

a) Searches the database and creates a set S of all the terms `A/N)` such that (1) the database contains a user-defined procedure whose predicate has identifier `A` and arity `N`, (2) `A/N` unifies with `PI`,

b) If a non-empty set is found, proceeds to 8.8.2.1d,

c) Else the predicate fails.

d) Chooses an element of the set S and the predicate succeeds.

e) If all the elements of the set S have been chosen, then the predicate fails,

f) Else chooses an element of the set S which has not already been chosen, and the predicate succeeds.

`current_predicate(PI)` is re-executable. On backtracking, continue at 8.8.2.1e.

The order in which predicate indicators are found by `current_predicate(PI)` is implementation dependent.

NOTE — All user-defined procedures are found, whether static or dynamic.

A user-defined procedure is also found even when all its clauses have been retracted.

A user-defined procedure is not found if it has been abolished.

8.8.2.2 Template and modes

```
current_predicate(?predicate_indicator)
```

8.8.2.3 Errors

None.

8.8.2.4 Examples

These examples assume the database has been created from the Prolog text defined at the beginning of 8.8.

```
current_predicate(dog/0).
   Succeeds.

current_predicate(current_predicate/1).
   Fails.

current_predicate(elk/Arity).
   Succeeds, unifying Arity with 1.

current_predicate(foo/A).
   Fails.

current_predicate(Name/1).
   Succeeds, unifying Name with elk.
   On re-execution, succeeds,
   unifying Name with insect.
   [The order of solutions is
     implementation dependent]

current_predicate(4).
   Fails.
```

8.9 Clause creation and destruction

These predicates enable the database (7.5) to be altered during resolution.

8.9.1 asserta/1

8.9.1.1 Description

`asserta(Clause)` is true.

Procedurally, `asserta(Clause)` is executed as follows:

a) If `Clause` unifies with `':-'(Head, Body)` then proceeds to 8.9.1.1c,

b) Else unifies `Head` with `Clause` and `true` with `Body`,

c) Converts (7.6.1) the term `Head` to a head `H`,

d) Converts (7.6.2) the term `Body` to a goal `G`,

e) Constructs the clause with head `H` and body `B`,

f) Adds that clause before all existing clauses of the procedure whose predicate is equal to the functor of `Head`.

8.9.1.2 Template and modes

```
asserta(@clause)
```

8.9.1.3 Errors

a) `Head` is a variable
— `instantiation_error`.

b) `Head` cannot be converted to a predication
— `type_error(callable, Head)`.

c) `Body` cannot be converted to a goal
— `type_error(body, Body)`.

d) The predicate indicator `Pred` of `Head` is not that of a dynamic procedure
— `permission_error(modify, static_procedure, Pred)`.

8.9.1.4 Examples

The examples defined in this clause assume the database has been created from the following Prolog text:

```
:- dynamic(legs/2).
legs(A, 6) :- insect(A).

:- dynamic(insect/1).
insect(ant).
insect(bee).

asserta(legs(octopus, 8)).
   Succeeds.

asserta( (legs(A, 4) :- animal(A)) ).
   Succeeds.

asserta( (foo(X) :- X, call(X)) ).
   Succeeds.

asserta(_).
   instantiation_error.

asserta(4).
   type_error(callable, 4).

asserta( (foo :- 4) ).
   type_error(body, 4).

asserta( (atom(_) :- true) ).
   permission_error(modify_clause,
     static_procedure, atom/1).
```

After these examples the database could have been created from the following Prolog text:

```
:- dynamic(legs/2).
legs(A, 4) :- animal(A).
legs(octopus, 8).
legs(A, 6) :- insect(A).

:- dynamic(insect/1).
insect(ant).
insect(bee).

:- dynamic(foo/1).
foo(X) :- call(X), call(X).
```

8.9.2 assertz/1

8.9.2.1 Description

`assertz(Clause)` is true.

Procedurally, `assertz(Clause)` is executed as follows:

a) If `Clause` unifies with `':-'(Head, Body)` then proceeds to 8.9.2.1c,

b) Else unifies `Head` with `Clause` and `true` with `Body`,

c) Converts (7.6.1) the term `Head` to a head `H`,

d) Converts (7.6.2) the term `Body` to a goal `G`,

e) Constructs the clause with head `H` and body `B`,

f) Adds that clause after all existing clauses of the procedure whose predicate is equal to the functor of `Head`.

8.9.2.2 Template and modes

```
assertz(@clause)
```

8.9.2.3 Errors

a) `Head` is a variable
— `instantiation_error`.

b) `Head` cannot be converted to a predication
— `type_error(callable, Head)`.

c) `Body` cannot be converted to a goal
— `type_error(body, Body)`.

d) The predicate indicator `Pred` of `Head` is not that of a dynamic procedure
— `permission_error(modify, static_procedure, Pred)`.

8.9.2.4 Examples

The examples defined in this clause assume the database has been created from the following Prolog text:

```
:- dynamic(legs/2).
legs(A, 4) :- animal(A).
legs(octopus, 8).
legs(A, 6) :- insect(A).

:- dynamic(insect/1).
insect(ant).
insect(bee).

:- dynamic(foo/1).
foo(X) :- call(X), call(X).
```

```
assertz(legs(spider, 8)).
   Succeeds.

assertz( (legs(B, 2) :- bird(B)) ).
   Succeeds.

assertz( (foo(X) :- X -> call(X)) ).
   Succeeds.

assertz(_).
   instantiation_error.

assertz(4).
   type_error(callable, 4).

assertz( (foo :- 4) ).
   type_error(body, 4).

assertz( (atom(_) :- true) ).
   permission_error(modify_clause,
      static_procedure, atom/1).
```

After these examples the database could have been created from the following Prolog text:

```
:- dynamic(legs/2).
legs(A, 4) :- animal(A).
legs(octopus, 8).
legs(A, 6) :- insect(A).
legs(spider, 8).
legs(B, 2) :- bird(B).

:- dynamic(insect/1).
insect(ant).
insect(bee).

:- dynamic(foo/1).
foo(X) :- call(X), call(X).
foo(X) :- call(X) -> call(X).
```

8.9.3 retract/1

8.9.3.1 Description

`retract(Clause)` is true iff the database contains at least one dynamic procedure with a clause which unifies with `Head :- Body`.

Procedurally, `retract(Clause)` is executed as follows:

a) If `Clause` unifies with `':-'(Head, Body)` then proceeds to 8.9.3.1c,

b) Else unifies `Head` with `Clause` and `true` with `Body`,

c) Searches sequentially through each dynamic user-defined procedure in the database and creates a list L of all the terms `clause(H, B)` such that

1) the database contains a clause whose head can be converted to a term `H`, and whose body coan be converted to a term `B`, and

2) `H` unifies with `Head`, and

3) `B` unifies with `Body`.

d) If a non-empty list is found, proceeds to 8.9.3.1f,

e) Else the predicate fails.

f) Chooses the first element of the list L, removes the clause corresponding to it from the database, and the predicate succeeds.

g) If all the elements of the list L have been chosen, then the predicate fails,

h) Else chooses the first element of the list L which has not already been chosen, removes the clause, if it exists, corresponding to it from the database and the predicate succeeds.

`retract/1` is re-executable. On backtracking, continue at 8.9.3.1g.

8.9.3.2 Template and modes

```
retract(+clause)
```

8.9.3.3 Errors

a) `Head` is a variable
— `instantiation_error`.

b) `Head` is not a predication
— `type_error(callable, Head)`.

c) The predicate indicator `Pred` of `Head` is not that of a dynamic procedure
— `permission_error(modify, static_procedure, Pred)`.

8.9.3.4 Examples

The examples defined in this clause assume the database has been created from the following Prolog text:

```
:- dynamic(legs/2).
legs(A, 4) :- animal(A).
legs(octopus, 8).
legs(A, 6) :- insect(A).
legs(spider, 8).
legs(B, 2) :- bird(B).

:- dynamic(insect/1).
insect(ant).
insect(bee).

:- dynamic(foo/1).
foo(X) :- call(X), call(X).
foo(X) :- call(X) -> call(X).


retract(legs(octopus, 8)).
   Succeeds, retracting the clause
     'legs(octopus, 8)'.

retract(legs(spider, 6)).
   Fails.

retract( (legs(X, 2) :- T) ).
   Succeeds, unifying T with bird(X),
     and retracting the clause
        'legs(B, 2) :- bird(B)'.

retract( (legs(X, Y) :- Z) ).
   Succeeds, unifying Y with 4,
     and Z with animal(X),
     noting the list of clauses to be retracted
        = [ (legs(A, 4) :- animal(A)),
            (legs(A, 6) :- insect(A)),
            (legs(spider, 8) :- true) ],
     and retracting the clause
        'legs(A, 4) :- animal(A)'.
   On re-execution, succeeds,
     unifying Y with 6, and Z with insect(X),
     and retracting the clause
       'legs(A, 6) :- insect(A)'.
   On re-execution, succeeds, unifying Y with 8,
     and X with spider, and Z with true,
     and retracting the clause
       'legs(A, 8) :- animal(A)'.
   On re-execution, fails.

retract(insect(I)), write(I),
     retract(insect(bee)), fail.
   'retract(insect(I))' succeeds,
     unifying I with 'ant',
     noting the list of clauses to be retracted
        = [insect(ant), insect(bee)],
     and retracting the clause 'insect(ant)'.
```

```
   'write(ant)' succeeds, outputting 'ant'.
   'retract(insect(bee))' succeeds,
     noting the list of clauses to be retracted
       = [insect(bee)],
     and retracting the clause 'insect(bee)'.
   'fail' fails.
   On re-execution, 'retract(insect(bee))' fails.
   On re-execution, 'write(ant)' fails.
   On re-execution, 'retract(insect(I))' succeeds,
      unifying I with 'bee',
      noting the list of clauses to be retracted
        = [insect(bee)],
      [the clause 'insect(bee)' has already
        been retracted.]
   'write(bee)' succeeds, outputting 'bee'.
   'retract(insect(bee))' fails.
   On re-execution, 'write(bee)' fails.
   On re-execution, 'retract(insect(I))' fails.
   Fails.

retract(( foo(A) :- A, call(A) )).
   Succeeds, retracting the clause
     'foo(X) :- call(X), call(X)'.

retract(( foo(A) :- A -> B )).
   Succeeds, unifying A with B,
   and retracting the clause
     'foo(X) :- call(X) -> call(X)'.

retract( (X :- in_eec(Y)) ).
   instantiation_error.

retract( (4 :- X) ).
   type_error(callable, 4).

retract( (atom(X) :- X == '[]') ).
   permission_error(modify_clause,
     static_procedure, atom/1).
```

After these examples, the database is empty.

8.9.4 abolish/1

8.9.4.1 Description

`abolish(Pred)` is true.

Procedurally, `abolish(Pred)` removes from the database all clauses for the dynamic procedure specified by the predicate indicator `Pred` leaving the database in the same state as if the procedure had never existed.

8.9.4.2 Template and modes

```
abolish(@predicate_indicator)
```

8.9.4.3 Errors

a) `Pred` is a variable
— `instantiation_error`.

b) `Pred` is a term `Name/Arity` and either `Name` or `Arity` is a variable
— `instantiation_error`.

c) `Pred` is a term `Name/Arity` and `Arity` is neither a variable nor an integer
— `type_error(integer, Arity)`.

d) `Pred` is a term `Name/Arity` and `Name` is neither a variable nor an atom
— `type_error(atom, Name)`.

e) The procedure specified by `Pred` is not that of a dynamic procedure
— `permission_error(modify, static_procedure, Pred)`.

8.9.4.4 Examples

```
abolish(foo/2).
   Succeeds, also undefines foo/2 if there exists
   a dynamic procedure with predicate foo/2.

abolish(foo/_).
   instantiation_error.

abolish(abolish/1).
   permission_error(modify_clause,
      static_procedure, abolish/1).
```

8.10 All solutions

These predicates create a list of all the solutions of a goal.

8.10.1 findall/3

8.10.1.1 Description

`findall(Term, Goal, Bag)` is true iff `Bag` unifies with the list of values to which a variable `X` not occurring in `Term` or `Goal` would be instantiated by successive resatisfaction of

```
call(Goal), X=Term
```

after systematic replacement of all variables in `X` by new variables.

Procedurally, `findall(Term, Goal, Bag)` is executed as follows:

a) Creates an empty list L,

b) Executes `call(G)`,

c) If it fails, proceeds to 8.10.1.1g,

d) Else if it succeeds, appends a renamed copy (refrenamedcopyofaterm) of `Term` to L,

e) Re-executes `call(G)`,

f) Proceeds to 8.10.1.1c,

g) Unifies L with `Bag`,

h) If the unification succeeds, the predicate succeeds,

i) Else the predicate fails.

8.10.1.2 Template and modes

```
findall(@term, @callable_term, ?list)
```

8.10.1.3 Errors

a) `Goal` is a variable
— `instantiation_error`.

b) `Goal` is not a callable term
— `type_error(callable, Goal)`.

8.10.1.4 Examples

```
findall(X, (X=1; X=2), S).
  Succeeds, unifying S with [1, 2].

findall(X+Y, (X=1), S).
  Succeeds, unifying S with [1+_].

findall(X, fail, L).
  Succeeds, unifying S with [].

findall(X, (X=1; X=1), S).
  Succeeds, unifying S with [1, 1].

findall(X, (X=2; X=1), [1, 2]).
  Fails.

findall(X, Goal, S).
  instantiation_error.

findall(X, 4, S).
  type_error(callable, 4).
```

8.10.2 bagof/3

`bagof/3` assembles as a list the solutions of a goal for each different instantiation of the free variables in that goal. The elements of each list are in order of solution, but the order in which each list is found is undefined.

8.10.2.1 Description

`bagof(Template, Goal, Instances)` is true iff:

— `G` is the iterated-goal term (7.1.6.3) of `Goal`, and

— `FV` is a witness (7.1.1.2) of the free variables set (7.1.1.4) of `Goal` with respect to `Template`, and

— `Instances` is a non-empty list of `Template` such that `G` is true, and

— Each element of `Instances` corresponds to a single binding of `FV`, and

— The elements of `Instances` are in order of solution.

Procedurally, `bagof(Template, Goal, Instances)` is executed as follows:

a) Let `Witness` be a witness (7.1.1.2) of the free variables set (7.1.1.4) of `Goal` with respect to `Template`,

b) Let `G` be the iterated-goal term (7.1.6.3) of `Goal`,

c) Executes the goal `findall(Witness+Template, G, S)`,

d) If `S` is the empty list, then fails,

e) Else proceeds to step 8.10.2.1f.

f) Chooses any element, `W+T`, of `S`.

g) Let `WT_list` be the largest proper sublist (7.1.6.4) of `S` such that, for each element `WW+TT` of `WT_list`, `WW` is a variant (7.1.6.1) of `W`,

h) Let `T_list` be the list such that, for each element `WW+TT` of `WT_list`, there is a corresponding element `TT` of `T_list`,

i) Let `S_next` be the largest proper sublist of `S` such that `WW+TT` is an element of `S_next` iff `WW+TT` is not an element `WT_list`,

j) Replaces `S` by `S_next`,

k) If `T_list` unifies with `Instances`, unifies `Witness` with each `WW` defined in 8.10.2.1g, and succeeds,

l) Else proceeds to step 8.10.2.1d.

`bagof/3` is re-executable. On backtracking, continue at 8.10.2.1d.

NOTES

1 Step 8.10.2.1f does not define which element of those eligible will be chosen. The order of solutions for `bagof/3` is thus undefined.

2 If the free variables set of `Goal` with respect to `Template` is empty, and `Iterated_Goal` succeeds, then the predicate can succeed only once.

3 This definition implies that the variables of `Template` and the variables in the existential variables set (7.1.1.3) of `Goal` remain uninstantiated after each success of `bagof(Template, Goal, Instances)`.

8.10.2.2 Template and modes

```
bagof(@term, +callable_term, ?list)
```

8.10.2.3 Errors

a) `G` is a variable
— `instantiation_error`.

b) `G` is not a callable term
— `type_error(callable, G)`.

8.10.2.4 Examples

```
bagof(X, (X=1 ; X=2), S).
   Free variables set: {}.
   Succeeds, unifying S with [1,2].

bagof(X, (X=1 ; X=2), X).
   Free variables set: {}.
   Succeeds, unifying X with [1,2].

bagof(X, fail, S).
   Free variables set: {}.
   Fails.

bagof(1, (Y=1 ; Y=2), L).
   Free variables set: {Y}.
   Succeeds, unifying L with [1],
     and Y with 1.
   On re-execution, succeeds, unifying L with [1],
     and Y with 2.
   [The order of solutions is undefined]

bagof(f(X, Y), (X=a ; Y=b), L).
   Free variables set: {}.
   Succeeds, unifying L with [f(a, _), f(_, b)].

bagof(X, Y^((X=1, Y=1) ; (X=2, Y=2)), S).
   Free variables set: {}.
   Succeeds, unifying S with [1, 2].

bagof(X, Y^((X=1 ; Y=1) ; (X=2, Y=2)), S).
   Free variables set: {}.
   Succeeds, unifying S with [1, _, 2].

bagof(X, (Y^(X=1 ; Y=2) ; X=3), S).
   Free variables set: {Y}.
   Warning: the procedure ^/2 is undefined.
   Succeeds, unifying S with [3], and Y with _.
   [Assuming the value associated with the flag
   'undefined_predicate' is 'warning'.]

bagof(X, (X=Y ; X=Z ; Y=1), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y, Z].
   On re-execution, succeeds, unifying S with [_],
     and Y with 1.

bagof(X, (X=Y ; X=Z), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y, Z].

bagof(X, a(X, Y), L).
   Clauses of a/2:
     a(1, f(_)).
     a(2, f(_)).
   Free variables set: {Y}.
   Succeeds, unifying L with [1, 2],
     and Y with f(_).

bagof(X, b(X, Y), L).
   Clauses of b/2:
     b(1, 1).
     b(1, 1).
     b(1, 2).
     b(2, 1).
     b(2, 2).
     b(2, 2).
   Free variables set: {Y}.
   Succeeds, unifying L with [1,1,2],
     and Y with 1.
   On re-execution, succeeds,
     unifying L with [1,2,2], and Y with 2.
   [The order of solutions is undefined]

bagof(X, Y^Z, L).
   instantiation_error.

bagof(X, 1, L).
   type_error(callable, 1).
```

The following fully worked examples explain bagof/3 in greater detail.

```
** Example: bagof(f(X,Y), (X=a;Y=b), L).

   Template = f(X,Y)
   Goal = (X=a;Y=b)
   Instances = L

   Iterated-goal term = (X=a;Y=b)
   Free variables set of
     Goal with respect to Template: {}
   step c -- findall(w+f(X,Y), (X=a;Y=b), S).
     S = [w+f(a,_), w+f(_,b)]
   step f -- W+T = w+f(_,b)
   step g -- WT_list = [w+f(a,_), w+f(_,b)]
   step h -- T_list = [f(a,_), f(_,b)]
   step i -- S_next = []
   Succeeds, unifying L with [f(a,_),f(_,b)].

   On re-execution,
   step d ---  Fails.

** Example: bagof(X,Y^((X=1;Y=1);(X=2,Y=2)),B).

   Template = X
```

```
   Goal = Y^((X=1;Y=1);(X=2,Y=2))
   Instances = B

   Iterated-goal term = ((X=1;Y=1);(X=2,Y=2))
   Free variables set of
      Goal with respect to Template: {}
   step c -- findall(w+X, ((X=1;Y=1);(X=2,Y=2)),
        S).
      S = [w+1, w+_, w+2]
   step f -- W+T = w+_
   step g -- WT_list = [w+1, w+_, w+2]
   step h -- T_list = [1, _, 2]
   step i -- S_next = []
   Succeeds, unifying B with [1, _, 2]

   On re-execution,
   step d ---  Fails.


** Example: bagof(X,(Y^(X=1;Y=2);X=3),C).

   Template = X
   Goal = (Y^(X=1;Y=2);X=3)
   Instances = C

   Iterated-goal term = (Y^(X=1;Y=2);X=3)
   Free variables set of
      Goal with respect to Template: {Y}
   step c -- findall(w(Y)+X, (Y^(X=1;Y=2);X=3),
        S).
      S = [w(_)+3]
   step f -- W+T = w(_)+3
   step g -- WT_list = [w(_)+3]
   step h -- T_list = [3]
   step i -- S_next = []
   Succeeds, unifying C with [3], and Y with _.

   On re-execution,
   step d ---  Fails.

   Note -- This assumes the first alternative
      fails because the procedure ^/2 has
      no defining clauses in the database,
      and the value associated with flag
      'undefined_predicate' is 'fail'.

** Example: bagof(X, (X=Y ; X=Z ; Y=1), D).

   Template = X
   Goal = (X=Y ; X=Z ; Y=1)
   Instances = D

   Iterated-goal term = (X=Y ; X=Z ; Y=1)
   Free variables set of
      Goal with respect to Template: {Y, Z}
   step c -- findall(w(Y,Z)+X, (X=Y ; X=Z ; Y=1),
        S).
   S = [w(X1,_)+X1, w(_,X2)+X2, w(1,_)+X3]
   step f -- W+T = w(_,X2)+X2
   step g -- WT_list = [w(X1,_)+X1, w(_,X2)+X2]
   step h -- T_list = [X1, X2]
   step i -- S_next = [w(1,_)+X3]
   Succeeds, unifying D with [X1, X2],
      and Y with X1, and Z with X2.

   On re-execution,
   step f -- W+T = w(1,_)+X3
   step g -- WT_list = [w(1,_)+X3]
   step h -- T_list = [X3]
   step i -- S_next = []
   Succeeds, unifying D with [X3], and Y with 1.

   On re-execution,
   step d ---  Fails.
```

8.10.3 setof/3

`setof/3` assembles as a list the solutions of a goal for each different instantiation of the free variables in that goal. The elements of each list are distinct and ordered, but the order in which each list is found is undefined.

8.10.3.1 Description

`setof(Template, Goal, Instances)` is true iff

— `G` is the iterated-goal term (7.1.6.3) of `Goal`, and

— `FV` is a witness (7.1.1.2) of the free variables set (7.1.1.4) of `Goal` with respect to `Template`, and

— `Instance_list` is a non-empty list of `Template` such that `G` is true, and

— Each element of `Instance_list` corresponds to a single binding of `FV`, and

— `Instances` is the sorted list (7.1.6.5) of `Instance_list`.

Procedurally, `setof(Template, Goal, Instances)` is executed as follows:

a) Let `Witness` be a witness of the free variables set (7.1.1.4) of `Goal` with respect to `Template`,

b) Let `G` be the iterated-goal term (7.1.6.3) of `Goal`,

c)
Execute the goal `findall(Witness+Template, G, S)`,

d) If `S` is the empty list, the predicate fails.

e) Else proceed to step 8.10.3.1f.

f) Choose any element, `W+T`, of `S`.

g) Let `WT_list` be the largest proper sublist (7.1.6.4) of `S` such that, for each element `WW+TT` of `WT_list`, `WW` is a variant (7.1.6.1) of `W`,

h) Let `T_list` be a list such that, for each element `WW+TT` of `WT_list`, there is a corresponding element `TT` of `T_list`,

i) Let SS be the largest proper sublist of S such that WW+TT is an element of S_next iff WW+TT is not an element WT_list,

j) Let S_next be the sorted list (7.1.6.5) of S,

k) Replace S by S_next,

l) If T_list unifies with Instances, the predicate succeeds and unifies Witness with each WW defined in 8.10.3.1g,

m) Else proceed to step 8.10.3.1d.

setof/3 is re-executable. On backtracking, continue at 8.10.3.1d.

8.10.3.2 Template and modes

```
setof(@term, +callable_term, ?list)
```

8.10.3.3 Errors

a) G is a variable
— instantiation_error.

b) G is not a callable term
— type_error(callable, G).

8.10.3.4 Examples

```
setof(X, (X=1; X=2), S).
   Free variables set: {}.
   Succeeds, unifying S with [1,2].

setof(X, (X=1; X=2), X).
   Free variables set: {}.
   Succeeds, unifying X with [1,2].

setof(X, (X=2; X=1), S).
   Free variables set: {}.
   Succeeds, unifying S with [1,2].

setof(X, (X=2; X=2), S).
   Free variables set: {}.
   Succeeds, unifying S with [2].

setof(X, (X=Y; X=Z), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y, Z] or [Z, Y].
   [The solution is implementation dependent.]

setof(X, fail, S).
   Free variables set: {}.
   Fails.

setof(1, (Y=2 ; Y=1), L).
   Free variables set: {Y}.
   Succeeds, unifying L with [1], and
     Y with 1.
   On re-execution, succeeds,
     unifying L with [1], and Y with 2.
   [The order of solutions is undefined]

setof(f(X,Y), (X=a ; Y=b), L).
   Free variables set: {}.
   Succeeds, unifying L with [f(_,b),f(a,_)].

setof(X, Y^((X=1, Y=1) ; (X=2, Y=2)), S).
   Free variables set: {}.
   Succeeds, unifying S with [1,2].

setof(X, Y^((X=1 ; Y=1) ; (X=2, Y=2)), S).
   Free variables set: {}.
   Succeeds, unifying S with [_,1,2].

setof(X, (Y^(X=1 ; Y=2) ; X=3), S).
   Free variables set: {Y}.
   Warning: the procedure ^/2 is undefined.
   Succeeds, unifying S with [3], and Y with _.
   [Assuming the value associated with the flag
   'undefined_predicate' is 'warning'.]

setof(X, (X=Y ; X=Z ; Y=1), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y,Z] or [Z,Y].
   On re-execution, succeeds, unifying S with [_],
     and Y with 1.

setof(X, a(X, Y), L).
   Clauses of a/2:
     a(1, f(_)).
     a(2, f(_)).
   Free variables set: {Y}.
   Succeeds, unifying L with [1, 2],
     and Y with f(_).

The following examples assume that member/2
   is defined with the following clauses:
     member(X, [X | _]).
     member(X, [_ | L]) :-
        member(X, L).

setof(X, member(X,[f(U,b),f(V,c)]), L).
   Free variables set: {U, V}.
   Succeeds, unifying L with [f(U,b),f(V,c)] or
     with [f(V,c),f(U,b)].

setof(X, member(X,[f(U,b),f(V,c)]),
     [f(a,c),f(a,b)]).
   Free variables set: {U, V}.
   Implementation dependent.

setof(X, member(X,[f(b,U),f(c,V)]),
     [f(b,a),f(c,a)]).
   Free variables set: {U, V}.
   Succeeds, unifying U with a, and V with a.

setof(X, member(X,[V,U,f(U),f(V)]), L).
   Free variables set: {U, V}.
   Succeeds, unifying L with [U,V,f(U),f(V)] or
     with [V,U,f(V),f(U)].

setof(X, member(X,[V,U,f(U),f(V)]),
     [a,b,f(a),f(b)]).
   Free variables set: {U, V}.
   Implementation dependent.
   Succeeds, unifying U with a, and V with B;
     or, unifying U with b, and V with a.

setof(X, member(X,[V,U,f(U),f(V)]),
     [a,b,f(b),f(a)]).
```

```
   Free variables set: {U, V}.
   Fails.

setof(X,
      (exists(U,V)^member(X,[V,U,f(U),f(V)])),
      [a,b,f(b),f(a)]).
   Free variables set: {}.
   Succeeds.

The following examples assume that b/2 is defined
   with the following clauses:
      b(1, 1).
      b(1, 1).
      b(1, 2).
      b(2, 1).
      b(2, 2).
      b(2, 2).

setof(X, b(X, Y), L).
   Free variables set: {Y}.
   Succeeds, unifying L with [1, 2], and Y with 1.
   On re-execution, succeeds,
      unifying L with [1, 2], and Y with 2.
   [The order of solutions is undefined]

setof(X-Xs,Y^setof(Y,b(X,Y),Xs),L).
   Free variables set: {}.
   Succeeds, unifying L with [1-[1,2],2-[1,2]].
   [Each list is independently ordered]

setof(X-Xs,setof(Y,b(X,Y),Xs),L).
   Free variables set: {Y}.
   Succeeds, unifying L with [1-[1,2],2-[1,2]],
      and Y with _.
   [Each list is independently ordered]

setof(X-Xs,bagof(Y,d(X,Y),Xs),L).
   Clauses of d/3:
      d(1,1).
      d(1,2).
      d(1,1).
      d(2,2).
      d(2,1).
      d(2,2).
   Free variables set: {Y}.
   Succeeds,
      unifying L with [1-[1,2,1],2-[2,1,2]],
      and Y with _.
```

8.11 Stream selection and control

These predicates link an external source/sink with a Prolog stream, its stream identifier and stream alias. They enable the source/sink to be opened and closed, and its properties found during execution.

NOTE — The use of these predicates may cause a Resource Error (7.12.2h) because, for example, the program has opened too many streams, or a file or disk is full. The use of these predicates may also cause a System Error (7.12.2j) because the operating system is reporting a problem.

The precise reasons for such errors, the side effects which have occurred, and the way they can be circumvented cannot be specified in this draft International Standard.

8.11.1 current_input/1

8.11.1.1 Description

`current_input(Stream)` is true iff the stream identifier `Stream` identifies the current input stream (7.10.2.4).

Procedurally, `current_input(Stream)` unifies `Stream` with the stream identifier of the current input stream.

8.11.1.2 Template and modes

`current_input(?stream)`

8.11.1.3 Errors

None.

8.11.2 current_output/1

8.11.2.1 Description

`current_output(Stream)` is true iff the stream identifier `Stream` identifies the current output stream (7.10.2.4).

Procedurally, `current_output(Stream)` unifies `Stream` with the stream identifier of the current output stream.

8.11.2.2 Template and modes

`current_output(?stream)`

8.11.2.3 Errors

None.

8.11.3 set_input/1

8.11.3.1 Description

`set_input(S_or_a)` is true.

Procedurally, `set_input(S_or_a)` is executed as follows:

a) If S_or_a is not a stream identifier or alias for an input stream which is currently open, then there shall be an error,

b) Else set the stream associated with stream identifier or alias S_or_a to be the current input stream, and succeeds.

8.11.3.2 Template and modes

```
set_input(@stream_or_alias)
```

8.11.3.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an output stream
— permission_error(input, stream, S_or_a).

8.11.4 set_output/1

8.11.4.1 Description

set_output(S_or_a) is true.

Procedurally, set_output(S_or_a) is executed as follows:

a) If S_or_a is not a stream identifier or alias for an output stream which is currently open, then there shall be an error,

b) Else set the stream associated with stream identifier or alias S_or_a to be the current output stream, and succeeds.

8.11.4.2 Template and modes

```
set_output(@stream_or_alias)
```

8.11.4.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

8.11.5 open/3

8.11.5.1 Description

open(Source_sink, Mode, Stream) is true iff open(Source_sink, Mode, Stream, []).

8.11.5.2 Template and modes

```
open(@source_sink, @io_mode, -stream)
```

8.11.5.3 Errors

a) Source_sink is a variable
— instantiation_error.

b) Mode is a variable
— instantiation_error.

c) Source_sink is neither a variable nor a source/sink
— domain_error(source_sink, Source_sink).

d) Mode is neither a variable nor an atom
— type_error(atom, Mode).

e) Stream is not a variable
— type_error(variable, Stream).

f) Mode is an atom but not an I/O mode
— domain_error(io_mode, Mode).

g) The source/sink specified by Source_sink cannot be opened
— permission_error(open, source/sink, Source_sink).

8.11.5.4 Examples

```
open('/user/dave/data', read, DD).
   Succeeds.
   [It opens the text file '/user/dave/data'
   for input, and unifies DD with a
   stream identifier for the stream.]
```

8.11.6 open/4

8.11.6.1 Description

`open(Source_sink, Mode, Stream, Options)` is true.

Procedurally, `open(Source_sink, Mode, Stream, Options)` is executed as follows:

a) Opens the source/sink `Source_sink` for input or output as indicated by I/O mode `Mode` and the list of stream-options `Options`.

b) Unifies `Stream` with the stream identifier which is to be associated with this stream,

c) The predicate succeeds.

8.11.6.2 Template and modes

```
open(@source_sink, @io_mode, -stream,
  @io_options)
```

8.11.6.3 Errors

a) `Source_sink` is a variable
— `instantiation_error`.

b) `Mode` is a variable
— `instantiation_error`.

c) `Options` is a variable
— `instantiation_error`.

d) `Options` is a list with an element `E` which is a variable
— `instantiation_error`.

e) `Source_sink` is neither a variable nor a source/sink
— `domain_error(source_sink, Source_sink)`.

f) `Mode` is neither a variable nor an atom
— `type_error(atom, Mode)`.

g) `Options` is neither a variable nor a list
— `type_error(list, Options)`.

h) An element `E` of the `Options` list is neither a variable nor a stream-option
— `domain_error(stream_option, E)`.

i) `Stream` is not a variable
— `type_error(variable, Stream)`.

j) `Mode` is an atom but not an I/O mode
— `domain_error(io_mode, Mode)`.

k) The source/sink specified by `Source_sink` cannot be opened
— `permission_error(open, source_sink, Source_sink)`.

l) An element `E` of the `Options` list is `alias(A)` and `A` is already associated with an open stream
— `permission_error(open, source_sink, alias(A))`.

m) An element `E` of the `Options` list is `reposition(true)` and it is not possible to reposition this stream
— `permission_error(open, source_sink, reposition(true))`.

NOTE — A permission error when `Mode` is `write` or `append` means that `Source_sink` does not specify a sink that can be created, for example, a specified disk or directory does not exist. If `Mode` is `read` then it is also possible that the file specification is valid but the file does not exist.

8.11.6.4 Examples

```
open('/user/dave/data', read, DD, []).
   Succeeds.
   [It opens the text file '/user/dave/data'
   for input, and unifies DD with a
   stream identifier for the stream.]
```

8.11.7 close/1

8.11.7.1 Description

`close(S_or_a)` is true iff
`close(S_or_a, []).`

8.11.7.2 Template and modes

```
close(@stream_or_alias)
```

8.11.7.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

8.11.8 close/2

This built-in predicate closes the stream associated with stream identifier or alias S_or_a if it is open. The behaviour of this predicate may be modified by specifying a list of close-options (7.10.2.12) in the Options parameter.

8.11.8.1 Description

close(S_or_a, Options) is true.

Procedurally, close(S_or_a, Options) is executed as follows:

a) If S_or_a is an atom which is not the alias of a currently open stream, then the predicate succeeds,

b) Else if S_or_a is a valid stream representation but does not represent a currently open stream, then the predicate succeeds,

c) Else, any output which is currently buffered by the processor for the stream associated with S_or_a is sent to that stream,

d) If the stream identifier or alias S_or_a is the standard input stream or the standard output stream, then the predicate succeeds,

e) Else if the stream associated with S_or_a is not the current input stream then proceeds to 8.11.8.1g,

f) The current input stream becomes the standard input stream user_input,

g) If the stream associated with S_or_a is not the current output stream then proceeds to 8.11.8.1i,

h) The current output stream becomes the standard output stream user_output,

i) Closes the stream associated with S_or_a and deletes any alias associated with that stream,

j) The predicate succeeds.

The above implies that when a stream Stream has already been closed, a subsequent call close(S_or_a) simply succeeds.

8.11.8.2 Template and modes

close(@stream_or_alias, @close_options)

8.11.8.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) Options is a variable
— instantiation_error.

c) Options is a list with an element E which is a variable
— instantiation_error.

d) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

e) Options is neither a variable nor a list
— type_error(list, Options).

f) An element E of the Options list is neither a variable nor a close-option
— domain_error(close_option, E).

8.11.9 flush_output/0

NOTE — Flushing an output stream is explained in clause 7.10.2.10.

8.11.9.1 Description

flush_output is true.

Procedurally, flush_output is executed as follows:

a) Any output which is currently buffered by the processor for the current output stream is sent to that stream,

b) The predicate succeeds.

8.11.9.2 Template and modes

flush_output

8.11.9.3 Errors

None.

8.11.10 flush_output/1

NOTE — Flushing an output stream is explained in clause 7.10.2.10.

8.11.10.1 Description

`flush_output(S_or_a)` is true.

Procedurally, `flush_output(S_or_a)` is executed as follows:

a) Any output which is currently buffered by the processor for the stream associated with stream identifier or alias `S_or_a` is sent to that stream,

b) The predicate succeeds.

8.11.10.2 Template and modes

```
flush_output(@stream_or_alias)
```

8.11.10.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

d) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

8.11.11 stream_property/2

8.11.11.1 Description

`stream_property(Stream, Property)` is true iff the stream identified by the stream identifier `Stream` has stream property (7.10.2.13) `Property`.

Procedurally, `stream_property(Stream, Property)` is executed as follows:

a) Computes SP, the set of all pairs `(S,P)` such that `S` is a currently open stream which has property `P`,

b) If SP is empty, the predicate fails,

c) Else, chooses one pair `(S,P)` in SP and removes it from the set,

d) Unifies `S` with `Stream` and `P` with `Property`,

e) If the unification succeeds, the predicate succeeds,

f) Else, proceeds to 8.11.11.1b.

`stream_property(Stream, Property)` is re-executable. On backtracking, continue at 8.11.11.1b.

NOTE — When used in non-determinate ways, stream_property shall exhibit logical semantics for state changes. For example:

```
:- stream_property(S, P),
  write(S:P),
  nl,
  close(S),
  fail.
```

shall show all the properties that all open streams had before this goal was run. Note that this example may call `close(S)` several times for each stream S, but this does not cause any problem since close simply succeeds if called on a stream which is already closed.

8.11.11.2 Template and modes

```
stream_property(?stream,
?stream_property)
```

8.11.11.3 Errors

None.

8.11.11.4 Examples

```
stream_property(S, file_name(F))
   If S is instantiated, succeeds,
      unifying F with the name of the file
      to which it is connected,
   Else succeeds, unifying S with a
      stream identifier and F with the name
      of the file to which it is connected;
      on re-execution, succeeds in turn with
      each stream which is connected to a file.

stream_property(S, output)
   If S is instantiated, succeeds iff output
      is permitted on this stream,
```

```
Else succeeds, unifying S with a
   stream identifier which is open for
   output; on re-execution, succeeds in turn
   with each stream which is open for output.
```

8.11.12 at_end_of_stream/0

8.11.12.1 Description

`at_end_of_stream` is true iff the current input stream has a stream position end-of-stream or past-end-of-stream (7.10.2.9, 7.10.2.13).

8.11.12.2 Template and modes

```
at_end_of_stream
```

8.11.12.3 Errors

None.

8.11.13 at_end_of_stream/1

8.11.13.1 Description

`at_end_of_stream(S_or_a)` is true iff the stream associated with stream identifier or alias `S_or_a` has a stream position end-of-stream or past-end-of-stream (7.10.2.9, 7.10.2.13).

8.11.13.2 Template and modes

```
at_end_of_stream(@stream_or_alias)
```

8.11.13.3 Errors

None.

8.11.14 set_stream_position/2

8.11.14.1 Description

`set_stream_position(S_or_a, Position)` is true.

Procedurally, `set_stream_position( S_or_a, Position)` is executed as follows:

a) Sets the stream position of the stream associated with stream identifier or alias `S_or_a` to `Position`,

b) Succeeds.

NOTE — Normally, `Position` will previously have been returned as a `position/1` stream property of the stream.

8.11.14.2 Template and modes

```
set_stream_position(@stream_or_alias,
  @stream_position)
```

8.11.14.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Position` is a variable
— `instantiation_error`.

c) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

d) `Position` is neither a variable nor a stream position
— `domain_error(stream_position, Position)`.

e) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

f) `S_or_a` has stream property `reposition(false)`
— `permission_error(reposition, stream, S_or_a)`.

8.12 Character input/output

These built-in predicates enable a single character or byte to be input and output from a stream.

8.12.1 get_char/1

8.12.1.1 Description

`get_char(Char)` is true iff `(current_input(S), get_char(S, Char))`.

8.12.1.2 Template and modes

```
get_char(?character)
```

8.12.1.3 Errors

a) The current input stream has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `existence_error(past_end_of_stream, current_input_stream)`.

8.12.1.4 Examples

```
get_char(Char).
   current input stream
      qwerty ...
   Char is unified with the atom 'q' and
   the current input stream becomes
      werty ...
```

8.12.2 get_char/2

8.12.2.1 Description

`get_char(S_or_a, Char)` is true iff

— The stream associated with stream identifier or alias `S_or_a` is a text stream, and `Char` unifies with the next character to be read from `S_or_a`, or

— The stream associated with stream identifier or alias `S_or_a` is a binary stream, and `Char` unifies with the next byte to be read from `S_or_a`.

Procedurally, `get_char(S_or_a, Char)` is executed as follows:

a) If the stream associated with `S_or_a` has stream properties `end_of_stream(past)` and `eof_action(A)` then performs the action appropriate to the value of `A` specified in clause 7.10.2.11.

b) If the stream position is end-of-stream, proceeds to 8.12.2.1h,

c) If the stream associated with `S_or_a` is a text stream, reads the next character `C` from the stream associated with `S_or_a`,

d) If the stream associated with `S_or_a` is a binary stream, reads the next byte `C` from the stream associated with `S_or_a`,

e) Advances the stream position of the stream associated with `S_or_a` by one character,

f) If `C` unifies with `Char`, the predicate succeeds,

g) Else the predicate fails.

h) Sets the stream position so that it is past-end-of-stream,

i) If the atom `end_of_file` unifies with `C`, the predicate succeeds,

j) Else the predicate fails.

8.12.2.2 Template and modes

```
get_char(@stream_or_alias, ?character)
```

8.12.2.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

d) `S_or_a` has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `existence_error(past_end_of_stream, S_or_a)`.

e) `S_or_a` is an output stream
— `permission_error(input, stream, S_or_a)`.

8.12.2.4 Examples

```
get_char(Stream, Char).
   The contents of Stream are
      qwerty ...
   Char is unified with 'q' and
   Stream is left as
      werty ...

get_char(Stream, Char).
   The contents of Stream are
      'qwerty' ...
   Char is unified with '\'' (the
   atom containing just a single
   quote) and Stream is left as
      qwerty' ...
```

```
get_char(my_file, '\13\').
   The contents of my_file are
     \13\10\newline ...
   Succeeds and my_file is left as
     \10\newline ...

get_char(Stream, p).
   The contents of Stream are
     qwerty ...
   Fails and
   Stream is left as
     werty ...

get_char(user_output, X).
   The contents of user_output are
     qwerty ...
   permission_error(input, stream, user_output).
   user_output is left as
     qwerty ...

get_char(S, Char).
   Stream position of S is end-of-stream.
   Char is unified with end_of_file and
   Stream position of S is past-end-of-stream.
```

8.12.3 put_char/1

8.12.3.1 Description

`put_char(Char)` is true iff
`(current_output(S), put_char(S, Char))`.

8.12.3.2 Template and modes

```
put_char(@character)
```

8.12.3.3 Errors

a) `Char` is a variable
— `instantiation_error`.

b) `Char` is neither a variable nor a character
— `type_error(character, Char)`.

c) `Char` is neither a variable nor a character (7.1.4.1)
— `representation_error(character)`.

8.12.3.4 Examples

```
put_char(t).
   current output stream
     ... qwer
   Succeeds and leaves that stream
     ... qwert
```

8.12.4 put_char/2

8.12.4.1 Description

Procedurally, `put_char(S_or_a, Char)` is executed as follows:

a) Outputs the character `Char` to the stream associated with stream identifier or alias `S_or_a`.

b) Changes the stream position on the stream associated with `S_or_a` to take account of the character which has been output,

c) The predicate succeeds.

8.12.4.2 Template and modes

```
put_char(@stream_or_alias, @character)
```

8.12.4.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Char` is a variable
— `instantiation_error`.

c) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

d) `Char` is neither a variable nor a character
— `type_error(character, Char)`.

e) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

f) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

g) `Char` is neither a variable nor a character (7.1.4.1)
— `representation_error(character)`.

8.12.4.4 Examples

```
put_char(Stream,t).
   If the stream associated with Stream contains
     ... qwer
   Succeeds and leaves that stream
     ... qwert

put_char(my_file, C).
   instantiation_error.

put_char(Str, 'ty').
```

```
   type_error(character, ty).

put_char(Str, 'A').
   If the stream associated with Stream contains
      ... qwer
   Succeeds and leaves that stream
      ... qwerA
```

8.12.5 nl/0

8.12.5.1 Description

`nl` is true iff
`(current_output(S), nl(S))`.

8.12.5.2 Template and modes

```
nl
```

8.12.5.3 Errors

None.

8.12.5.4 Examples

```
nl, put_char(a).
   Current output stream
      ... qwer
   Succeeds and leaves that stream
      ... qwer
      a
```

8.12.6 nl/1

8.12.6.1 Description

`nl(S_or_a)` is true.

Procedurally, `nl(S_or_a)` is executed as follows:

a) Outputs a new line character to the stream associated with `S_or_a`,

b) Succeeds.

NOTES

1 This built-in predicate terminates the current line or record.

2 `nl(S_or_a)` is equivalent to `put_char(S_or_a, '\n')`.

8.12.6.2 Template and modes

```
nl(@stream_or_alias)
```

8.12.6.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

d) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

8.12.6.4 Examples

```
nl(st), put_char(st, a).
   If the stream associated with st contains
      ... qwer
   Succeeds and leaves that stream
      ... qwer
      a

nl(Str).
   instantiation_error.

nl([my_file]).
   domain_error(stream_or_alias, [my_file]).
```

8.13 Character code input/output

These built-in predicates enable a single byte to be input and output from a stream.

8.13.1 get_code/1

8.13.1.1 Description

`get_code(Code)` is true iff
`(current_input(S), get_code(S, Code))`.

8.13.1.2 Template and modes

```
get_code(?character_code)
```

8.13.1.3 Errors

a) The current input stream has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `existence_error(past_end_of_stream, current_input_stream)`

8.13.1.4 Examples

```
get_code(Code).
   current input stream
     qwerty ...
   Code is unified with 0'q and
   the current input stream becomes
     werty ...
```

8.13.2 get_code/2

8.13.2.1 Description

If the stream associated with stream identifier or alias `S_or_a` is a text stream then `get_code(S_or_a, Int)` is true iff `Int` unifies with the character code (7.1.2.2) corresponding to the next character to be read from `S_or_a`, else if `S_or_a` is a binary stream `get_code(S_or_a, Int)` is true iff `Int` unifies with the next byte to be read from `S_or_a`.

Procedurally, `get_code(S_or_a, Int)` is executed as follows:

a) If the stream associated with `S_or_a` has stream properties `end_of_stream(past)` and `eof_action(A)` then performs the action appropriate to the value of `A` specified in clause 7.10.2.11.

b) If there is no more data in the stream, proceeds to 8.13.2.1g,

c) Else reads the next character with character code `I`, from the stream associated with `S_or_a`,

d) Advances the stream position of the stream associated with `S_or_a` by one character,

e) If `I` unifies with `Int`, the predicate succeeds,

f) Else the predicate fails.

g) Sets the stream position so that it is past-end-of-stream,

h) If the integer -1 unifies with `Int`, the predicate succeeds,

i) Else the predicate fails.

8.13.2.2 Template and modes

```
get_code(@stream_or_alias,
?character_code)
```

8.13.2.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

d) `S_or_a` has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `existence_error(past_end_of_stream, S_or_a)`

e) `S_or_a` is an output stream
— `permission_error(input, stream, S_or_a)`.

8.13.2.4 Examples

```
get_code(Stream, Code).
   The contents of Stream are
     qwerty ...
   Code is unified with 0'q and
   Stream is left as
     werty ...

get_code(Stream, Code).
   The contents of Stream are
     'qwerty' ...
   Code is unified with 0''' and
   Stream is left as
     qwerty' ...

get_code(my_file, 0'\13\).
   The contents of my_file are
     0'\13\0'\10\\newline ...
   Succeeds and my_file is left as
     0'\10\newline ...

get_code(Stream, 0'p).
   The contents of Stream are
     qwerty ...
   Fails and
   Stream is left as
     werty ...

get_code(user_output, X).
   The contents of Stream are
```

```
     qwerty ...
   permission_error(input, stream, user_output).
   Stream is left as
     qwerty ...

get_code(S, Code).
   If stream position of the
     stream associated with S is end-of-stream
   then
     Succeeds, unifying Code with 1, and
     Sets stream position of S to
       past-end-of-stream.
```

8.13.3 put_code/1

8.13.3.1 Description

`put_code(Code)` is true iff
`(current_output(S), put_code(S, Code))`.

8.13.3.2 Template and modes

```
put_code(@character_code)
```

8.13.3.3 Errors

a) `Code` is a variable
— `instantiation_error`.

b) `Code` is neither a variable nor an integer
— `type_error(integer, Code)`.

c) `Code` is neither a variable nor a character code (7.1.2.2)
— `representation_error(character_code)`.

8.13.3.4 Examples

```
put_code(0't).
   current output stream
     ... qwer
   Succeeds and leaves that stream
     ... qwert
```

8.13.4 put_code/2

8.13.4.1 Description

Procedurally, `put_code(S_or_a, Code)` is executed as follows:

a) Outputs the character `Code` to the stream associated with stream identifier or alias `S_or_a`.

b) Changes the stream position on the stream associated with `S_or_a` to take account of the character which has been output,

c) The predicate succeeds.

8.13.4.2 Template and modes

```
put_code(@stream_or_alias,
@character_code)
```

8.13.4.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Code` is a variable
— `instantiation_error`.

c) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

d) `Code` is neither a variable nor an integer
— `type_error(integer, Code)`.

e) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

f) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

g) `Code` is neither a variable nor a character code (7.1.2.2)
— `representation_error(character_code)`

8.13.4.4 Examples

```
put_code(Stream, 0't).
   If the stream associated with Stream contains
     ... qwer
   Succeeds and leaves that stream
     ... qwert

put_code(my_file, C).
   instantiation_error.

put_code(Str, 'ty').
   type_error(integer, 'ab').

put_code(Str, 65).
   If the stream associated with Stream contains
     ... qwer
   Succeeds and leaves that stream
     ... qwerA
```

8.14 Term input/output

These predicates enable a Prolog term to be input from or output to a stream. The syntax of such terms can also be altered by changing the operators, and making some characters equivalent to one another.

8.14.1 read_term/2

8.14.1.1 Description

`read_term(Term, Options)` is true iff
`(current_input(S), read_term(S, Term, Options))`.

8.14.1.2 Template and modes

`read_term(?term, +read_options_list)`

8.14.1.3 Errors

a) One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
— `syntax_error`.

b) `Options` is a variable
— `instantiation_error`.

c) `Options` is a list with an element `E` which is a variable
— `instantiation_error`.

d) `Options` is neither a variable nor a list
— `type_error(list, Options)`.

e) An element `E` of the `Options` list is neither a variable nor a valid read-option
— `domain_error(read_option, E)`.

8.14.2 read_term/3

8.14.2.1 Description

`read_term(S_or_a, Term, Options)` is true iff `Term` unifies with `T`, where `T.` is a read-term which has been constructed by inputting and parsing characters from the stream associated with stream identifier or alias `S_or_a` (see 7.10.4).

The read-options (7.10.3) specified in `Options` will be instantiated to provide additional information about the term which is read.

NOTE — The effect of this predicate may be modified by calling the built-in predicate `char_conversion/2` (8.14.15), and if the value associated with the flag `char_conversion` (7.11.2.1) is `on`.

8.14.2.2 Template and modes

`read_term(@stream_or_alias, ?term, +read_options_list)`

8.14.2.3 Errors

a) One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
— `syntax_error`.

b) `S_or_a` is a variable
— `instantiation_error`.

c) `Options` is a variable
— `instantiation_error`.

d) `Options` is a list with an element `E` which is a variable
— `instantiation_error`.

e) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

f) `Options` is neither a variable nor a list
— `type_error(list, Options)`.

g) An element `E` of the `Options` list is neither a variable nor a valid read-option
— `domain_error(read_option, E)`.

h) An element `E` of the `Options` list is `alias(A)` and `A` is already associated with an open stream
— `domain_error(read_option, E)`.

i) An element `E` of the `Options` list is `reposition(true)` and it is not possible to reposition this stream
— `permission_error(reposition, E)`.

j) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

k) `S_or_a` is an output stream
— `permission_error(input, stream, S_or_a)`.

8.14.3 read/1

8.14.3.1 Description

`read(Term)` is true iff
`(current_input(S), read_term(S, Term, [])).`

8.14.3.2 Template and modes

```
read(?term)
```

8.14.3.3 Errors

a) One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
— `syntax_error`.

8.14.3.4 Examples

```
read(T).
   current input stream is
term1. term2. ...
   Succeeds, unifying T with term1.
   The current input stream is left as
 term2. ...

read(term1).
   current input stream is
term1. term2. ...
   Succeeds.
   current input stream is left as
 term2. ...

read(T).
   current input stream is
3.1.  term2. ...
   Succeeds, unifying T with 3.1.
   The current input stream is left as
  term2. ...

read(4.1).
   current input stream is
3.1.  term2. ...
   Fails.
   The current input stream is left as
  term2 ...

read(T).
   current input stream is
foo 123. term2. ...
   and foo is not a current prefix operator
   syntax_error.
   The current input stream is left as
 term2. ...

read(T).
   current input stream is
3.1
   syntax_error.
   The current input stream is left with
   position past-end-of-stream.
```

8.14.4 read/2

8.14.4.1 Description

`read(S_or_a, Term)` is true iff
`read_term(S_or_a, Term, []).`

8.14.4.2 Template and modes

```
read(@stream_or_alias, ?term)
```

8.14.4.3 Errors

a) One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
— `syntax_error`.

b) `S_or_a` is a variable
— `instantiation_error`.

c) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a).`

d) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a).`

e) `S_or_a` is an output stream
— `permission_error(input, stream, S_or_a).`

8.14.5 write_term/2

8.14.5.1 Description

`write_term(Term, Options)` is true iff
`(current_output(S), write_term(S, Term, Options)).`

8.14.5.2 Template and modes

```
write_term(@term, @write_options_list)
```

8.14.5.3 Errors

a) Options is a variable
— instantiation_error.

b) Options is a list with an element E which is a variable
— instantiation_error.

c) Options is neither a variable nor a list
— type_error(list, Options).

d) An element E of the Options list is neither a variable nor a valid write-option
— domain_error(write_option, E).

e) Options is neither a variable nor a list
— type_error(list, Options).

f) An element E of the Options list is neither a variable nor a valid write-option
— domain_error(write_option, E).

g) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

h) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

8.14.6 write_term/3

8.14.6.1 Description

write_term(S_or_a, Term, Options) is true.

Procedurally, write_term(S_or_a, Term, Options) is executed as follows:

a) Outputs Term to the stream associated with stream identifier or alias S_or_a in a form which is defined by the write-options list (7.10.5) Options,

b) Succeeds.

8.14.6.2 Template and modes

```
write_term(@stream_or_alias, @term,
  @write_options_list)
```

8.14.6.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) Options is a variable
— instantiation_error.

c) Options is a list with an element E which is a variable
— instantiation_error.

d) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

8.14.6.4 Examples

```
write_term(S, [1,2,3], []).
   Succeeds, outputting the characters
[1,2,3]
   to the stream associated with S.

write_term(S, [1,2,3], [ignore_ops(true)]).
   Succeeds, outputting the characters
.(1,.(2,.(3,[ ])))
   to the stream associated with S.

write_term(S, '1<2', []).
   Succeeds, outputting the characters
1<2
   to the stream associated with S.

write_term(S, '1<2', [quoted(true)]).
   Succeeds, outputting the characters
'1<2'
   to the stream associated with S.

write_term(S, '$VAR'(0), [numbervars(true)]).
   Succeeds, outputting the character
A
   to the stream associated with S.

write_term(S, '$VAR'(1), [numbervars(true)]).
   Succeeds, outputting the character
B
   to the stream associated with S.

write_term(S, '$VAR'(25), [numbervars(true)]).
   Succeeds, outputting the characters
Z
   to the stream associated with S.

write_term(S, '$VAR'(26), [numbervars(true)]).
   Succeeds, outputting the character
A1
   to the stream associated with S.

write_term(S, '$VAR'(51), [numbervars(true)]).
   Succeeds, outputting the characters
Z1
   to the stream associated with S.

write_term(S, '$VAR'(52), [numbervars(true)]).
   Succeeds, outputting the characters
A2
   to the stream associated with S.
```

8.14.7 write/1

8.14.7.1 Description

write(Term) is true iff
  (current_output(S), write(S, Term)).

8.14.7.2 Template and modes

write(@term)

8.14.7.3 Errors

None.

8.14.8 write/2

8.14.8.1 Description

write(S_or_a, Term) is true iff
  write_term(S_or_a, Term, [numbervars(true)]).

8.14.8.2 Template and modes

write(@stream_or_alias, @term)

8.14.8.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

8.14.8.4 Examples

```
write(out, [1,2,3]).
  Succeeds, outputting the characters
[1,2,3]
  to the stream associated with the alias 'out'.

write(out, 1<2).
  Succeeds, outputting the characters
1<2
  to the stream associated with the alias 'out'.

write(out, '1<2').
  Succeeds, outputting the characters
1<2
  to the stream associated with the alias 'out'.

write(out, '$VAR'(0)<'$VAR'(1)).
  Succeeds, outputting the characters
A<B
  to the stream associated with the alias 'out'.
```

8.14.9 writeq/1

8.14.9.1 Description

writeq(Term) is true iff
  (current_output(S), writeq(S, Term)).

8.14.9.2 Template and modes

writeq(@term)

8.14.9.3 Errors

None.

8.14.10 writeq/2

8.14.10.1 Description

writeq(S_or_a, Term) is true iff
  write_term(S_or_a, Term,
    [quoted(true), numbervars(true)]).

8.14.10.2 Template and modes

writeq(@stream_or_alias, @term)

8.14.10.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

8.14.10.4 Examples

```
writeq(out, [1,2,3]).
   Succeeds, outputting the characters
[1,2,3]
   to the stream associated with the alias 'out'.

writeq(out, 1<2).
   Succeeds, outputting the characters
1<2
   to the stream associated with the alias 'out'.

writeq(out, '1<2').
   Succeeds, outputting the characters
'1<2'
   to the stream associated with the alias 'out'.

writeq(out, '$VAR'(0)<'$VAR'(1)).
   Succeeds, outputting the characters
A<B
   to the stream associated with the alias 'out'.
```

8.14.11 write_canonical/1

8.14.11.1 Description

write_canonical(T) is true iff
(current_output(S), write_canonical(S, T)).

8.14.11.2 Template and modes

```
write_canonical(@term)
```

8.14.11.3 Errors

None.

8.14.12 write_canonical/2

8.14.12.1 Description

write_canonical(S_or_a, Term) is true iff
write_term(S_or_a, Term,
[quoted(true), ignore_ops(true)]).

8.14.12.2 Template and modes

```
write_canonical(@stream_or_alias, @term)
```

8.14.12.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

8.14.12.4 Examples

```
write_canonical(out, [1,2,3]).
   Succeeds, outputting the characters
'.'(1,'.'(2,'.'(3,[])))
   to the stream associated with the alias 'out'.

write_canonical(out, 1<2).
   Succeeds, outputting the characters
<(1,2)
   to the stream associated with the alias 'out'.

write_canonical(out, '1<2').
   Succeeds, outputting the characters
'1<2'
   to the stream associated with the alias 'out'.

write_canonical(out, '$VAR'(0)<'$VAR'(1)).
   Succeeds, outputting the characters
<('$VAR'(0),'$VAR'(1))
   to the stream associated with the alias 'out'.
```

8.14.13 op/3

This predicate enables the predefined operators (see 6.3.4.4 and table 5) to be altered during execution.

8.14.13.1 Description

op(Priority, Op_specifier, Operator) is true.

Procedurally, op(Priority, Op_specifier, Operator) is executed as follows:

a) If Operator is an atom, creates the set Ops containing just that one atom,

b) Else if Operator is a list of atoms, creates the set Ops consisting of all the atoms in the list,

c) Chooses an element Op in the set Ops and removes it from the set,

d) If Op is not currently an operator with the same operator class (prefix, infix or postfix) as Op_specifier, then proceeds to 8.14.13.1f,

e) The operator property of Op with the same class as Op_specifier is removed, so that Op is no longer an operator of that class,

f) If Priority=0, proceeds to 8.14.13.1h,

g) Op is made an operator with specifier Op_specifier and priority Priority,

h) If Ops is non-empty, proceeds to 8.14.13.1c,

i) Else, the predicate succeeds.

In the event of an error being detected in an Operator list argument, it is undefined which, if any, of the atoms in the list is made an operator before the exception is raised.

NOTES

1 Operator notation is defined in 6.3.4. See also operator directives (7.4.2.4).

2 A Priority of zero can be used to remove an operator property from an atom.

3 It does not matter if the same atom appears more than once in an Operator list; this is not an error and the duplicates simply have no effect.

4 In general, the predefined operators can be removed, or their priority can be changed. However, it is an error to attempt to change the meaning of the , operator from its predefined status, see 6.3.4.3.

8.14.13.2 Template and modes

```
op(@integer, @operator_specifier,
  @atom_or_atom_list)
```

8.14.13.3 Errors

a) Priority is a variable
— instantiation_error.

b) Op_specifier is a variable
— instantiation_error.

c) Operator is a variable
— instantiation_error.

d) Operator is a list with an element E which is a variable
— instantiation_error.

e) Priority is neither a variable nor an integer
— type_error(integer, Priority).

f) Op_specifier is neither a variable nor an atom
— type_error(atom, Op_specifier).

g) Operator is neither a variable nor an atom nor a list
— type_error(list, Operator).

h) An element E of the Operator list is neither a variable nor an atom
— type_error(atom, E).

i) Priority is not between 0 and 1200 inclusive
— domain_error(operator_priority, Priority).

j) Op_specifier is not a valid operator specifier
— domain_error(operator_specifier, Op_specifier).

k) Operator is ','
— permission_error(modify, operator, Operator).

l) An element E of the Operator list is ','
— permission_error(modify, operator, E).

m) Op_specifier is a specifier such that Operator would have an invalid set of specifiers (see 6.3.4.3)
— permission_error(create, operator, Operator).

8.14.13.4 Examples

```
op(30, xfy, ++).
   Succeeds, making ++ a right associative
     infix operator with priority 30.

op(0, yfx, ++).
   Succeeds, making ++ no longer an
     infix operator.

op(max, xfy, ++).
   type_error(integer, max).

op(-30, xfy, ++).
   domain_error(operator_priority, -30).

op(1201, xfy, ++).
   domain_error(operator_priority, 1201).
```

```
op(30, XFY, ++).
   instantiation_error.

op(30, yfy, ++).
   domain_error(operator_specifier, yfy).

op(30, xfy, 0).
   type_error(list, 0).

op(30, xfy, ++), op(40, xfx, ++).
   Succeeds, making ++ a non-associative
     infix operator with priority 40.

op(30, xfy, ++), op(50, yf, ++).
   permission_error(create, operator, ++).
   [There cannot be an infix and a
     postfix operator with the same name.]
```

8.14.14 current_op/3

8.14.14.1 Description

`current_op(Priority, Op_specifier, Operator)` is true iff `Operator` is an operator with properties defined by specifier `Op_specifier` and precedence `Priority`.

Procedurally, `current_op(Priority, Op_specifier, Operator)` is executed as follows:

a) Searches the current operator definitions and creates a set S of all the triples `(P,Spec,Op)` such that there is an operator:

1) whose name, `Op`, unifies with `Operator`,

2) whose specifier, `Spec`, unifies with `Op_specifier`, and

3) whose priority, `P`, unifies with `Priority`,

b) If a non-empty set is found, proceeds to 8.14.14.1d,

c) Else the predicate fails.

d) Chooses an element of the set S and the predicate succeeds.

e) If all the elements of the set S have been chosen, then the predicate fails,

f) Else chooses an element of the set S which has not already been chosen, and the predicate succeeds.

`current_op(Priority, Op_specifier, Operator)` is re-executable. On backtracking, continue at 8.14.14.1e.

The order in which operators are found by `current_op/3` is implementation dependent.

When the operator `,` (comma) is a member of the set S it is represented by the atom `','`.

NOTES

1 The definition above implies that if a program calls `current_op/3` and then modifies an operator definition by calling `op/3`, and then backtracks into the call to `current_op/3`, then the changes are guaranteed not to affect that `current_op/3` goal. That is, `current_op/3` behaves as if it were implemented as a dynamic predicate whose clauses are retracted and asserted when `op/3` is called.

2 An operator `Old_op` which has been removed by `op(0, Op_specifier, Old_op)` is not found by `current_op/3`.

8.14.14.2 Template and modes

```
current_op(?integer, ?operator_specifier,
  ?atom)
```

8.14.14.3 Errors

None.

8.14.14.4 Examples

```
current_op(P, xfy, OP).
   If the predefined operators have not been
     altered, then
   Succeeds, unifying P with 1100,
     and OP with ';'.
   On re-execution, succeeds unifying
     P with 1050, and OP with '->'.
   On re-execution, succeeds unifying
     P with 1000, and OP with ','.
    [The order of solutions is
     implementation dependent.]
```

8.14.15 char_conversion/2

8.14.15.1 Description

`char_conversion(Input_char, Output_char)` is true.

Procedurally, `char_conversion(Input_char, Output_char)` is executed as follows:

a) If `Input_char` is equal to `Output_char`, the predicate succeeds.

b) Else update the character-conversion relation with the conversion (`Input_char` → `Output_char`), and the predicate succeeds.

NOTES

1 See also char-conversion directives (7.4.2.5).

2 A character `Input_char` and `Output_char` should be quoted in order to ensure that they have not been converted by a character-conversion directive when the Prolog text is read.

3 The character-conversion relation affects only characters read by term input (8.14). When it is necessary to convert characters read by character input/output built-in predicates (8.12), it will be necessary to program the conversion explicitly using `current_char_conversion/2` (8.14.16).

8.14.15.2 Template and modes

```
char_conversion(@character, @character)
```

8.14.15.3 Errors

a) `Input_char` is a variable
— `instantiation_error`.

b) `Output_char` is a variable
— `instantiation_error`.

c) `Input_char` is neither a variable nor a character
— `type_error(character, Input_char)`.

d) `Output_char` is neither a variable nor a character
— `type_error(character, Output_char)`.

8.14.15.4 Examples

`char_conversion('&', ',')`
Updates the char-conversion relation with (`&` → `','`) .
Succeeds.

`char_conversion('''', '\'')`
Updates the char-conversion relation with (**'** → `'`) where **'** is a character in an extended character set equivalent to the single quote.
Succeeds.

`char_conversion('`**a**`', a)`
Updates the char-conversion relation with (**a** → `a`) where **a** is a character in an extended character set equivalent to the small letter character `a`.
Succeeds.

After these three goals, when the value associated with flag `char_conversion` is on, all occurrences of `&`, **'**, and **a** as unquoted characters read by term input predicates are converted to `,`, `\`, and `a` respectively, for example the three characters **aaa** are converted to the characters `aaa`. However the characters `'`**aaa**`'` represent an atom **aaa** because they are enclosed by the single quotes.

`char_conversion('`**a**`', '`**a**`')`
Updates the char-conversion relation by removing the conversion (**a** → `a`).
Succeeds.

8.14.16 current_char_conversion/2

8.14.16.1 Description

`current_char_conversion(Input_char, Output_char)` is true iff the character-conversion relation contains the conversion (`Input_char` → `Output_char`).

Procedurally, `current_char_conversion(Input_char, Output_char)` is executed as follows:

a) Creates a set S of all the conversions (`In` → `Out`) in the the current character-conversion relation such that:

1) `In` unifies with `Input_char`, and

2) `Out`, unifies with `Output_char`,

b) If a non-empty set is found, proceeds to 8.14.16.1d,

c) Else the predicate fails.

d) Chooses an element of the set S which has not already been chosen, unifies `In` with `Input_char`, and `Out` with `Output_char`, and the predicate succeeds.

e) If all the elements of the set S have been chosen, then the predicate fails,

f) Else proceeds to 8.14.16.1d.

`current_char_conversion(Input_char, Output_char)` is re-executable. On backtracking, continue at 8.14.16.1e.

The order in which character-conversions are found by `current_char_conversion/2` is implementation dependent.

NOTES

1 The definition above implies that if a program calls current_char_conversion/2 and then modifies the character-conversion relation by calling char_conversion/2, and then backtracks into the call to current_char_conversion/2, then the changes are guaranteed not to affect that current_char_conversion/2 goal.

2 A character-conversion (C → CC) which has been removed by char_conversion(C, C) is not found by current_char_conversion/2.

8.14.16.2 Template and modes

```
current_char_conversion(?character,
  ?character)
```

8.14.16.3 Errors

None.

8.14.16.4 Examples

current_char_conversion(C, a)
  Assume the char-conversion relation is
    (**a** → a, *a* → a),
  Succeeds, unifying C with **a**.
  On re-execution, succeeds, unifying C with *a*.

8.15 Logic and control

These predicates are simply derived from the control constructs (7.8) and provide additional facilities for affecting the control flow during resolution.

8.15.1 fail_if/1

8.15.1.1 Description

fail_if(Term) is true *iff* call(Term) is false.

Procedurally, fail_if(Term) is executed as follows:

a) Executes call(Term),

b) If it succeeds, the predicate fails,

c) Else if it fails, the predicate succeeds.

NOTE — A predicate with the same meaning as fail_if/1 is implemented in many existing processors with a name not/1 which is misleading because the predicate gives *negation by failure* rather than true negation. Other processors implement this feature with a predicate \+/1.

8.15.1.2 Template and modes

```
fail_if(@callable_term)
```

8.15.1.3 Errors

a) Term is a variable
— instantiation_error.

b) Term is neither a variable nor a callable term
— type_error(callable, Term).

NOTE — Errors produced by the execution of the goal fail_if(Term) are regarded as errors in Term.

8.15.1.4 Examples

```
fail_if(true).
  Fails.

fail_if(!).
  Fails, the cut has no effect.

fail_if((!, fail)).
  Succeeds, the cut has no effect.

(X=1; X=2), fail_if((!, fail)).
  Succeeds, unifying X with 1.
  On re-execution, succeeds unifying X with 2.

fail_if(4 = 5).
  Succeeds.

fail_if(3).
  type_error(callable, 3).

fail_if(X).
  instantiation_error.

fail_if(X = f(X)).
  Undefined.
```

8.15.2 once/1

8.15.2.1 Description

once(Term) is true *iff* call(Term) is true.

Procedurally, once(Term) is executed as follows:

a) Executes call(Term),

b) If it succeeds, the predicate succeeds,

c) Else if it fails, the predicate fails.

NOTE — once(Term) behaves as call(Goal), but is not re-executable.

8.15.2.2 Template and modes

```
once(+callable_term)
```

8.15.2.3 Errors

a) `Term` is a variable
— `instantiation_error`.

b) `Term` is neither a variable nor a callable term
— `type_error(callable, Term)`.

NOTE — Errors produced by the execution of the goal `once(Term)` are regarded as errors in `Term`.

8.15.2.4 Examples

```
once(!).
   Succeeds (the same as true).

once(!), (X=1; X=2).
   Succeeds, unifying X with 1.
   On re-execution, succeeds unifying X with 2.

once(repeat).
   Succeeds (the same as true).

once(fail).
   Fails.

once(X = f(X)).
   Undefined.
```

8.15.3 repeat/0

8.15.3.1 Description

`repeat` is true.

Procedurally, `repeat` succeeds.

`repeat` is re-executable.

8.15.3.2 Template and modes

```
repeat
```

8.15.3.3 Errors

None.

8.15.3.4 Examples

```
repeat, write("hello "), fail.
   Writes
hello hello hello hello hello ...
   indefinitely.

repeat, !, fail.
   Fails, equivalent to (!, fail).
```

8.16 Constant processing

These predicates enable constants to be processed as a sequence of characters (7.1.4.1) and character codes (7.1.2.2). Facilities exist to split and join atoms, and to convert a single character to and from the corresponding character code, and to convert a number to and from a list of characters.

NOTES

1 The characters forming an atom are defined in 6.1.2b.

2 These predicates assume the characters of an atom can be numbered: clause 6.1.2b defines that the characters of a non-empty atom are numbered from one upwards.

8.16.1 atom_length/2

8.16.1.1 Description

`atom_length(Atom, Length)` is true iff integer `Length` equals the number of characters in the atom `Atom`.

Procedurally, `atom_length(Atom, Length)` unifies `Length` with the number of characters in `Atom`.

8.16.1.2 Template and modes

```
atom_length(+atom, ?integer)
```

8.16.1.3 Errors

a) `Atom` is a variable
— `instantiation_error`.

b) `Atom` is neither a variable nor an atom
— `type_error(atom, Atom)`.

c) `Length` is neither a variable nor an integer
— `type_error(integer, Length)`.

8.16.1.4 Examples

```
atom_length('enchanted evening', N).
```

```
   Succeeds, unifying N with 17.

atom_length('enchanted\
 evening', N).
   Succeeds, unifying N with 17.

atom_length('', N).
   Succeeds, unifying N with 0.

atom_length('scarlet', 5).
   Fails.

atom_length(Atom, 4).
   instantiation_error.

atom_length(1.23, 4).
   type_error(atom, 1.23).

atom_length(atom, '4').
   type_error(integer, '4').
```

8.16.2 atom_concat/3

8.16.2.1 Description

`atom_concat(Atom_1, Atom_2, Atom_12)` is true iff the atom `Atom_12` is the atom formed by concatenating the characters of the atom `Atom_2` to the characters of the atom `Atom_1`.

Procedurally, `atom_concat(Atom_1, Atom_2, Atom_12)` unifies `Atom_12` with the concatenation of `Atom_1` and `Atom_2`.

`atom_concat(Atom_1, Atom_2, Atom_12)` is re-executable when only `Atom_12` is instantiated. On re-execution successive values for `Atom_1` and `Atom_2` are generated.

8.16.2.2 Template and modes

```
atom_concat(?atom, ?atom, +atom)
atom_concat(+atom, +atom, -atom)
```

8.16.2.3 Errors

a) `Atom_1` and `Atom_12` are variables
— `instantiation_error`.

b) `Atom_2` and `Atom_12` are variables
— `instantiation_error`.

c) `Atom_1` is neither a variable nor an atom
— `type_error(atom, Atom_1)`.

d) `Atom_2` is neither a variable nor an atom
— `type_error(atom, Atom_2)`.

e) `Atom_12` is neither a variable nor an atom
— `type_error(atom, Atom_12)`.

8.16.2.4 Examples

In the examples below,

```
  S1 = 'hello',
  S2 = ' world',
  S4 = 'small world'.

atom_concat(S1, S2, S3).
   Succeeds, unifying S3 with 'hello world'.

atom_concat(T, S2, S4).
   Succeeds, unifying T with 'small'.

atom_concat(S1, S2, S4).
   Fails.

atom_concat(T1, T2, S1).
   Succeeds, unifying T1 with '',
      and T2 with 'hello'.
   On re-execution, succeeds,
       unifying T1 with 'h', and T2 with 'ello'.

atom_concat(small, S2, S4).
   instantiation_error.
```

8.16.3 sub_atom/4

8.16.3.1 Description

`sub_atom(Atom, Start, Length, Sub_atom)` is true iff atom `Sub_atom` is the atom with `Length` characters starting at the `Start`-th character of atom `Atom`.

Procedurally, `sub_atom(Atom, Start, Length, Sub_atom)` unifies `Sub_atom` with an atom `Atom` which has `Length` characters identical with the `Length` characters of atom `Atom` that start with the `Start`-th character of `Atom`.

`sub_atom(Atom, Start, Length, Sub_atom)` is re-executable. On re-execution all possible values for `Start`, `Length` and `Sub_atom` are generated.

8.16.3.2 Template and modes

```
sub_atom(+atom, ?integer, ?integer, ?atom)
```

8.16.3.3 Errors

a) `Atom` is a variable
— `instantiation_error`.

b) Atom is neither a variable nor an atom
— type_error(atom, Atom).

c) Sub_atom is neither a variable nor an atom
— type_error(atom, Sub_atom).

d) Start is neither a variable nor an integer
— type_error(integer, Start).

e) Length is neither a variable nor an integer
— type_error(integer, Length).

8.16.3.4 Examples

```
sub_atom('Banana', 4, 2, S2).
   Succeeds, unifying S2 with 'an'.

sub_atom('charity', _, 3, S2).
   Succeeds, unifying S2 with 'cha'.
   On re-execution, succeeds,
      unifying S2 with 'har'.
   On re-execution, succeeds,
      unifying S2 with 'ari'.
   On re-execution, succeeds,
      unifying S2 with 'rit'.
   On re-execution, succeeds,
      unifying S2 with 'ity'.

sub_atom('ab', Start, Length, Sub_atom).
   Succeeds, unifying Start with 1,
      and Length with 0, and Sub_atom with ''.
   On re-execution, succeeds,
      unifying Start with 1, and Length with 1,
      and Sub_atom with 'a'.
   On re-execution, succeeds,
      unifying Start with 1, and Length with 2,
      and Sub_atom with 'ab'.
   On re-execution, succeeds,
      unifying Start with 2, and Length with 0,
      and Sub_atom with ''.
   On re-execution, succeeds,
      unifying Start with 2, and Length with 1,
      and Sub_atom with 'b'.
   On re-execution, succeeds,
      unifying Start with 3, and Length with 0,
      and Sub_atom with ''.
```

8.16.4 atom_chars/2

8.16.4.1 Description

atom_chars(Atom, List) is true iff List is a list whose elements are the characters corresponding to the successive characters of atom Atom.

Procedurally, atom_chars(Atom, List) is executed as follows:

a) If Atom is an atom then List is unified with a list of characters which shall be identical to the sequence of characters which form the abstract syntax of Atom (see 6.1.2b),

b) Else if List is a list of characters, then Atom is unified with the atom whose abstract syntax has the same sequence of characters,

c) Else the predicate fails.

8.16.4.2 Template and modes

```
atom_chars(+atom, +list)
atom_chars(+atom, -list)
atom_chars(-atom, +list)
```

8.16.4.3 Errors

a) Atom and List are variables
— instantiation_error.

b) Atom is neither a variable nor an atom
— type_error(atom, Atom).

c) List is neither a variable nor a list nor a partial list
— type_error(list, List).

8.16.4.4 Examples

```
atom_chars('', L).
   Succeeds, unifying L with [].

atom_chars([], L).
   Succeeds, unifying L with ['[', ']'].

atom_chars('''', L).
   Succeeds, unifying L with ['''].

atom_chars('ant', L).
   Succeeds, unifying L with
      ['a', 'n', 't'].

atom_chars(Str, ['s', 'o', 'p']).
   Succeeds, unifying Str with 'sop'.

atom_chars('North', ['N' | X]).
   Succeeds, unifying X with
      ['o', 'r', 't', 'h'].

atom_chars('soap', ['s', 'o', 'p']).
   Fails.

atom_chars(X, Y).
   instantiation_error.
```

8.16.5 atom_codes/2

8.16.5.1 Description

atom_codes(Atom, List) is true iff List is a list whose elements correspond to the successive characters of atom Atom, and the value of each element is the character code for the corresponding character.

Procedurally, atom_codes(Atom, List) is executed as follows:

a) If Atom is an atom then List is unified with a list of character codes (7.1.2.2) corresponding to a sequence of characters which shall be identical to the sequence of characters which form the abstract syntax of Atom (see 6.1.2b),

b) Else if List is a list of character codes, then Atom is unified with the atom whose abstract syntax has the sequence of characters corresponding to the same list of character codes,

c) Else the predicate fails.

8.16.5.2 Template and modes

```
atom_codes(+atom, +list)
atom_codes(+atom, -list)
atom_codes(-atom, +list)
```

8.16.5.3 Errors

a) Atom and List are variables
— instantiation_error.

b) Atom is neither a variable nor an atom
— type_error(atom, Atom).

c) List is neither a variable nor a list nor a partial list
— type_error(list, List).

8.16.5.4 Examples

```
atom_codes('', L).
   Succeeds, unifying L with [].

atom_codes([], L).
   Succeeds, unifying L with [0'[, 0']].

atom_codes('''', L).
   Succeeds, unifying L with [0''''].

atom_codes('ant', L).
   Succeeds, unifying L with
   [0'a, 0'n, 0't].

atom_codes(Str, [0's, 0'o, 0'p]).
   Succeeds, unifying Str with 'sop'.

atom_codes('North', [0'N | X]).
   Succeeds, unifying X with
   [0'o, 0'r, 0't, 0'h].

atom_codes('soap', [0's, 0'o, 0'p]).
   Fails.

atom_codes(X, Y).
   instantiation_error.
```

8.16.6 char_code/2

8.16.6.1 Description

char_code(Char, Code) is true iff the character code (7.1.2.2) for the character Char is Code.

Procedurally, char_code(Char, Code) unifies Code with character code for the character Char.

8.16.6.2 Template and modes

```
char_code(+character, +character_code)
char_code(+character, -character_code)
char_code(-character, +character_code)
```

8.16.6.3 Errors

a) Char and Code are variables
— instantiation_error.

b) Char is neither a variable nor a character (7.1.4.1)
— representation_error(character).

c) Code is neither a variable nor a character code (7.1.2.2)
— representation_error(character_code).

8.16.6.4 Examples

```
char_code('a', Code).
   Succeeds, unifying Code with the
      character code for the character 'a'.

char_code(Str, 99).
   Succeeds, unifying Str with the character
      whose character code is 99.

char_code(Str, 0'c).
   Succeeds, unifying Str with the character 'c'.

char_code(Str, 163).
   If there is an extended character whose
```

```
        character code is 163 then
      Succeeds, unifying Str with that
        extended character,
   else
      representation_error(character_code).

char_code('b', 84).
   Succeeds iff the character 'b' has the
     character code 84.

char_code('ab', Int).
   type_error(character, ab).

char_code(C, I).
   instantiation_error.
```

8.16.7 number_chars/2

8.16.7.1 Description

`number_chars(Number, List)` is true iff `List` is a list whose elements are the characters corresponding to a character sequence of `Number` which could be output (7.10.6b, 7.10.6c).

Procedurally, `number_chars(Number, List)` is executed as follows:

a) If `List` is a list of characters, then that sequence of characters is parsed according to the syntax rules for numbers and negative numbers (6.3.1.1, 6.3.1.2). If the parse is successful, `Number` is unified with the resulting value, else the predicate fails.

b) Else if `Number` is an integer or float, then `List` is unified with a list of characters which shall be identical to the sequence of characters which would be output by `write_canonical(Number)` (see 7.10.6b, 7.10.6c, 8.14.11),

c) Else the predicate fails.

NOTES

1 The sequence of characters ensures that, for every number `X`, the following goal is true:

```
number_chars(X,C), number_chars(Y,C), X == Y.
```

2 This definition ensures that, the following goal is true:

```
C=['.', '1'],
   number_chars(X,C), number_chars(X,C).
```

8.16.7.2 Template and modes

```
number_chars(+number, +list)
number_chars(+number, -list)
number_chars(-number, +list)
```

8.16.7.3 Errors

a) `Number` and `List` are variables
— `instantiation_error`.

b) `Number` is neither a variable nor a number
— `type_error(number, Number)`.

c) `List` is neither a variable nor a list of characters
— `domain_error(character_list, List)`.

d) `List` is not parsable as a number
— `syntax_error`.

8.16.7.4 Examples

```
number_chars(33, L).
   Succeeds, unifying L with ['3', '3'].

number_chars(33, ['3', '3']).
   Succeeds.

number_chars(33.0, L).
   Succeeds, unifying L with an
   implementation dependent list of characters,
   e.g. ['3', '.', '3', 'E', +, '0', '1'].

number_chars(X,
     ['3', '.', '3', 'E', +, '0']).
   Succeeds, unifying X with a value
     approximately equal to 3.3.

number_chars(3.3,
     ['3', '.', '3', 'E', +, '0']).
   Implementation dependent: may succeed or fail.

number_chars(A, [-, '2', '5']).
   Succeeds, unifying A with -25.

number_chars(A, ['\n', ' ', '3']).
   [The new line and space characters are
     not significant.]
   Succeeds, unifying A with 3.

number_chars(A, ['3', ' ']).
   Fails.

number_chars(A, ['0', x, f])
   Succeeds, unifying A with 15.

number_chars(A, ['0', ''', a])
   Succeeds, unifying A with the
   collating sequence integer for the
   character 'a'.

number_chars(A, ['4', '.', '2']).
   Succeeds, unifying A with 4.2.

number_chars(A,
     ['4', '2', '.', '0', 'e', '-', '1']).
   Succeeds, unifying A with 4.2.
```

8.16.8 number_codes/2

8.16.8.1 Description

number_codes(Number, List) is true iff List is a list whose elements are the character codes corresponding to a character sequence of Number which could be output (7.10.6b, 7.10.6c).

Procedurally, number_codes(Number, List) is executed as follows:

a) If List is a list of character codes, then the sequence of characters corresponding to those character codes is parsed according to the syntax rules for numbers and negative numbers (6.3.1.1, 6.3.1.2). If the parse is successful, Number is unified with the resulting value, else the predicate fails.

b) Else if Number is an integer or float, then List is unified with a list of character codes corresponding to a sequence of characters which shall be identical to the sequence of characters which would be output by write_canonical(Number) (see 7.10.6b, 7.10.6c, 8.14.11),

c) Else the predicate fails.

NOTE — The sequence of character codes representing the characters of a number shall be such that for every value X, the following goal is true:

```
number_codes(X, C), number_codes(Y, C), X==Y.
```

8.16.8.2 Template and modes

```
number_codes(+number, +list)
number_codes(+number, -list)
number_codes(-number, +list)
```

8.16.8.3 Errors

a) Number and List are variables
— instantiation_error.

b) Number is neither a variable nor a number
— type_error(number, Number).

c) List is neither a variable nor a list of character codes
— domain_error(character_code_list, List).

d) List is not parsable as a number
— syntax_error.

8.16.8.4 Examples

```
number_codes(33, L).
   Succeeds, unifying L with [0'3, 0'3].

number_codes(33, [0'3, 0'3]).
   Succeeds.

number_codes(33.0, L).
   Succeeds, unifying L with an
   implementation dependent list of characters,
   e.g. [0'3, 0'., 0'3, 0'E, 0'+, 0'0, 0'1].

number_codes(33.0,
     [0'3, 0'., 0'3, 0'E, 0'+, 0'0, 0'1]).
   Implementation dependent: may succeed or fail.

number_codes(A, [0'-, 0'2, 0'5]).
   Succeeds, unifying A with -25.

number_codes(A, [0' , 0'3]).
   [The space character is not significant.]
   Succeeds, unifying A with 3.

number_codes(A, [0'0, 0'x, 0'f])
   Succeeds, unifying A with 15.

number_codes(A, [0'0, 0''', 0'a])
   Succeeds, unifying A with the
     collating sequence integer for the
     character 'a'.

number_codes(A, [0'4, 0'., 0'2]).
   Succeeds, unifying A with 4.2.

number_codes(A,
     [0'4, 0'2, 0'., 0'0, 0'e, 0'-, 0'1]).
   Succeeds, unifying A with 4.2.
```

8.17 Implementation defined hooks

These built-in predicates enable a program to find the current value of any flag (7.11), and to change the current value of some flags.

8.17.1 set_prolog_flag/2

8.17.1.1 Description

set_prolog_flag(Flag, Value) is true *iff*:

a) Flag is a flag, and

b) Value is a value that is within the implementation defined range of values for Flag.

Procedurally, set_prolog_flag(Flag, Value) is executed as follows:

a) If Flag is a flag (7.11), and Value is a value that is within the implementation defined range of values for

Flag, proceeds to 8.17.1.1c,

b) Else the predicate fails.

c) Associates the value Value with the flag Flag, and the predicate succeeds.

8.17.1.2 Template and modes

```
set_prolog_flag(@flag, @term)
```

8.17.1.3 Errors

a) Flag is a variable
— instantiation_error.

b) Value is a variable
— instantiation_error.

c) Flag is neither a variable nor an atom
— type_error(atom, Flag).

d) Flag is an atom but is invalid in the processor
— domain_error(prolog_flag, Flag).

e) Value is inappropriate for Flag
— domain_error(flag_value, Flag + Value).

8.17.1.4 Examples

```
set_prolog_flag(undefined_predicate, fail).
   Succeeds, associating the value fail
   with flag undefined_predicate.

set_prolog_flag(X, off).
   instantiation_error.

set_prolog_flag(5, decimals).
   type_error(atom, 5).

set_prolog_flag(date, 'July 1988').
   domain_error(flag, date).

set_prolog_flag(debug, trace).
   domain_error(flag_value, debug+trace).
```

8.17.2 current_prolog_flag/2

8.17.2.1 Description

current_prolog_flag(Flag, Value) is true iff Flag is a flag supported by the processor, and Value is the value currently associated with it.

Procedurally, current_prolog_flag(Flag, Value) is executed as follows:

a) Searches the current flags supported by the processor and creates a set S of all the terms flag(F, V) such that (1) there is a flag F which unifies with Flag, and (2) the value V currently associated with F unifies with Value,

b) If a non-empty set is found, proceeds to 8.17.2.1d,

c) Else the predicate fails.

d) Chooses an element of the set S and the predicate succeeds.

e) If all the elements of the set S have been chosen, then the predicate fails,

f) Else chooses an element of the set S which has not already been chosen, and the predicate succeeds.

current_prolog_flag(Flag, Value) is re-executable. On re-execution, continue at 8.17.2.1e above.

The order in which flags are found by current_prolog_flag(Flag, Value) is implementation dependent.

NOTE — All flags are found, whether defined by this draft International Standard or implementation defined.

8.17.2.2 Template and modes

```
current_prolog_flag(?flag, ?term)
```

8.17.2.3 Errors

a) Flag is not a variable or an atom
— type_error(atom, Flag).

8.17.2.4 Examples

```
current_prolog_flag(debug, off).
   Succeeds iff the value currently associated
   with the flag 'debug' is 'off'.

current_prolog_flag(F, V).
   Succeeds, unifying 'F' with one of the
   flags supported by the processor, and 'V'
   with the value currently associated with
   the flag 'F'.
   On re-execution, successively unifies 'F'
   and 'V' with each other flag supported by
   the processor and its associated value.

current_prolog_flag(5, _).
   type_error(atom, 5).
```

8.17.3 halt/0

8.17.3.1 Description

Procedurally, `halt` is executed as follows:

a) Exits from the processor,

b) Returns to whatever system invoked Prolog.

Any other effect of `halt/0` is implementation defined.

NOTE — This predicate neither succeeds nor fails.

8.17.3.2 Template and modes

```
halt
```

8.17.3.3 Errors

None.

8.17.3.4 Examples

```
halt.
   Implementation defined.
```

8.17.4 halt/1

8.17.4.1 Description

Procedurally, `halt(X)` is executed as follows:

a) Exits from the processor,

b) Returns to whatever system invoked Prolog passing the value of `X` as a message.

Any other effect of `halt/1` is implementation defined.

NOTE — This predicate neither succeeds nor fails.

8.17.4.2 Template and modes

```
halt(@integer)
```

8.17.4.3 Errors

a) `X` is a variable
— `instantiation_error`.

b) `X` is neither a variable nor an integer
— `type_error(integer, X)`.

8.17.4.4 Examples

```
halt(1).
   Implementation defined.

halt(a).
   type_error(integer, a).
```

---

9 Evaluable functors

This clause defines the evaluable functors which shall be implemented by a standard-conforming Prolog processor.

9.1 The simple arithmetic functors

The basic arithmetic functions are defined mathematically in the style, and conforming with the requirements, of IS10967-1.

9.1.1 Evaluable functors and operations

Each evaluable functor corresponds to one or more operations according to the types of the values which are obtained by evaluating the argument(s) of the functor.

The following table identifies the integer or floating point operations corresponding to each functor:

Evaluable functor	Operation
`'+'/2`	add_I, add_F, add_{FI}, add_{IF}
`'-'/2`	sub_I, sub_F, sub_{FI}, sub_{IF}
`'*'/2`	mul_I, mul_F, mul_{FI}, mul_{IF}
`'//'/2`	$intdiv_I$
`'/'/2`	div_F, div_{II}, div_{FI}, div_{IF}
`rem/2`	rem_I
`mod/2`	mod_I
`'-'/1`	neg_I, neg_F
`abs/1`	abs_I, abs_F
`sqrt/1`	$sqrt_I$, $sqrt_F$
`sign/1`	$sign_I$, $sign_F$
`float_truncate/2`	$trunc_F$
`float_round/2`	$round_F$
`float_integer_part/1`	$intpart_F$
`float_fractional_part/1`	$fractpart_F$
`float/1`	$float_{I\to F}$, $float_{F\to F}$
`floor/1`	$floor_{F\to I}$
`truncate/1`	$truncate_{F\to I}$

`round/1`	$round_{F \to I}$
`ceiling/1`	$ceiling_{F \to I}$

NOTE — `'+'`, `'-'`, `'*'`, `'//'`, `'/'`, `'rem'`, `'mod'` are infix predefined operators (see 6.3.4.4).

9.1.2 Integer operations and axioms

The following operations are specified:

$$
\begin{aligned}
&add_I : I \times I \to I \cup \{\mathbf{overflow}\}\\
&sub_I : I \times I \to I \cup \{\mathbf{overflow}\}\\
&mul_I : I \times I \to I \cup \{\mathbf{overflow}\}\\
&intdiv_I : I \times I \to I \cup \{\mathbf{overflow}, \mathbf{zero\_divisor}\}\\
&rem_I : I \times I \to I \cup \{\mathbf{zero\_divisor}\}\\
&mod_I : I \times I \to I \cup \{\mathbf{zero\_divisor}, \mathbf{undefined}\}\\
&neg_I : I \to I \cup \{\mathbf{overflow}\}\\
&abs_I : I \to I \cup \{\mathbf{overflow}\}\\
&sign_I : I \to I
\end{aligned}
$$

The behaviour of the integer operations are defined in terms of a rounding function $rnd_I(x)$ (see 9.1.2.1).

For all values x and y in I, the following axioms shall apply:

$$
\begin{aligned}
add_I(x,y) &= x + y \quad \text{if } x + y \in I\\
&= \mathbf{overflow} \quad \text{if } x + y \notin I\\
\\
sub_I(x,y) &= x - y \quad \text{if } x - y \in I\\
&= \mathbf{overflow} \quad \text{if } x - y \notin I\\
\\
mul_I(x,y) &= x * y \quad \text{if } x * y \in I\\
&= \mathbf{overflow} \quad \text{if } x * y \notin I\\
\\
intdiv_I(x,y) &= rnd_I(x/y)\\
&\quad \text{if } y \neq 0 \text{ and } rnd_I(x/y) \in I\\
&= \mathbf{overflow}\\
&\quad \text{if } y \neq 0 \text{ and } rnd_I(x/y) \notin I\\
&= \mathbf{zero\_divisor}\\
&\quad \text{if } y = 0\\
\\
rem_I(x,y) &= x - (rnd_I(x/y) * y)\\
&\quad \text{if } y \neq 0\\
&= \mathbf{zero\_divisor}\\
&\quad \text{if } y = 0\\
\\
mod_I(x,y) &= x - (\lfloor x/y \rfloor * y)\\
&\quad \text{if } y \neq 0\\
&= \mathbf{zero\_divisor}\\
&\quad \text{if } y = 0\\
\\
neg_I(x) &= -x \quad \text{if } -x \in I\\
&= \mathbf{overflow} \quad \text{if } -x \notin I\\
\\
abs_I(x) &= |x| \quad \text{if } |x| \in I\\
&= \mathbf{overflow} \quad \text{if } |x| \notin I\\
\\
sign_I(x) &= 1\\
&\quad \text{if } x \geq 0\\
&= -1\\
&\quad \text{if } x < 0
\end{aligned}
$$

NOTE — IS 10967 (LIA) permits mod_I to have one or both definitions of mod_I^1 and mod_I^2:

$$
\begin{aligned}
mod_I^1(x,y) &= x - (\lfloor x/y \rfloor * y) && \text{if } y \neq 0\\
&= \mathbf{zero\_divisor} && \text{if } y = 0\\
\\
mod_I^2(x,y) &= x - (\lfloor x/y \rfloor * y) && \text{if } y > 0\\
&= \mathbf{undefined} && \text{if } y < 0\\
&= \mathbf{zero\_divisor} && \text{if } y = 0
\end{aligned}
$$

For mod_I a processor would produce either $x - (\lfloor x/y \rfloor * y)$ or **undefined** when y is negative. The choice would be implementation defined, and independent of the argument values.

9.1.2.1 Integer division rounding function

An integer division rounding function is defined:

$$rnd_I : \mathcal{R} \to \mathcal{Z}$$

For $x \in \mathcal{R}$, the following axioms shall apply:

$rnd_I(x) \leq x$	(round toward minus infinity), or
$\lvert rnd_I(x) \rvert \leq \lvert x \rvert$	(round toward zero)

NOTE — A processor can provide one or both of these rounding functions. This will result in different versions of the $intdiv_I$ and rem_I operations for each rounding function provided.

9.1.3 Floating point operations and axioms

The following operations are specified:

$$
\begin{aligned}
&add_F : F \times F \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\}\\
&sub_F : F \times F \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\}\\
&mul_F : F \times F \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\}\\
&div_F : F \times F\\
&\quad \to F \cup \{\mathbf{overflow}, \mathbf{underflow}, \mathbf{zero\_divisor}\}\\
&neg_F : F \to F\\
&abs_F : F \to F\\
&sqrt_F : F \to F \cup \{\mathbf{undefined}\}\\
&sign_F : F \to F\\
&trunc_F : F \times I \to F \cup \{\mathbf{undefined}\}\\
&round_F : F \times I \to F \cup \{\mathbf{overflow}, \mathbf{undefined}\}\\
&intpart_F : F \to F\\
&fractpart_F : F \to F
\end{aligned}
$$

The behaviour of the floating point operations are defined in terms of a rounding function $rnd_F(x)$ (see 9.1.3.1),

a range checking function $chk_F(x,y)$ (see 9.1.3.2), an approximate-addition function $add^*_F(x,y)$ (see 9.1.3.3), an exponent function $e_F(x)$ (see 9.1.3.4), and a reduce-accuracy-and-round function $rn_F(x,n)$ (see 9.1.3.5).

For all values x and y in F, and n in I the following axioms shall apply:

$$
\begin{aligned}
add_F(x,y) &= chk_F(add^*_F(x,y),\ rnd_F(add^*_F(x,y))) \\
sub_F(x,y) &= add_F(x,-y) \\
mul_F(x,y) &= chk_F(x*y,\ rnd_F(x*y)) \\
div_F(x,y) &= chk_F(x/y,\ rnd_F(x/y)) \\
&\quad \text{if } y \neq 0 \\
&= \mathbf{zero\_divisor} \\
&\quad \text{if } y = 0 \\
neg_F(x) &= -x \\
abs_F(x) &= |x| \\
sqrt_F(x) &= chk_F(\sqrt{x},\ rnd_F(\sqrt{x})) \\
&\quad \text{if } x \geq 0 \\
&= \mathbf{undefined} \\
&\quad \text{if } x < 0 \\
sign_F(x) &= 1 \quad \text{if } x \geq 0 \\
&= -1 \quad \text{if } x < 0 \\
trunc_F(x,n) &= \lfloor x/r^{e_F(x)-n} \rfloor * r^{e_F(x)-n} \\
&\quad \text{if } n > 0 \text{ and } x \geq 0 \\
&= -trunc_F(-x,n) \\
&\quad \text{if } n > 0 \text{ and } x < 0 \\
&= \mathbf{undefined} \\
&\quad \text{if } n \leq 0 \\
round_F(x,n) &= rn_F(x,n) \\
&\quad \text{if } n > 0 \text{ and } |rn_F(x,n)| \leq fmax \\
&= \mathbf{overflow} \\
&\quad \text{if } n > 0 \text{ and } |rn_F(x,n)| > fmax \\
&= \mathbf{undefined} \\
&\quad \text{if } n \leq 0 \\
intpart_F(x) &= sign_F(x) * \lfloor |x| \rfloor \\
fractpart_F(x) &= x - intpart_F(x)
\end{aligned}
$$

9.1.3.1 Floating point rounding function

A floating point rounding function is defined:

$$rnd_F : \mathcal{R} \rightarrow F^*$$

For $x \in \mathcal{R}$ and $i \in \mathcal{Z}$, such that $|x| \geq fmin_N$ and $|x * r^i| \geq fmin_N$, the following axiom shall apply:

$$rnd_F(x * r^i) = rnd_F(x) * r^i$$

NOTES

1 This rule means that the rounding function does not depend on the exponent part of the floating point number except when denormalization occurs.

2 An implementation can provide more than one floating point rounding function. Each rounding function shall produce a distinct family of floating point operations.

9.1.3.2 Floating point range checking function

A floating point range checking function is defined:

$$chk_F : \mathcal{R} \times F^* \rightarrow F \cup \{\mathbf{overflow}, \mathbf{underflow}\}$$

There are five alternatives for the range checking function chk_F. They differ in their treatment of underflow:

chk_F^D	(denormalize on underflow (gradual underflow))
chk_F^{Za}	(force underflow to 0, check after rounding)
chk_F^{Na}	(notify on underflow, check after rounding)
chk_F^{Zb}	(force underflow to 0, check before rounding)
chk_F^{Nb}	(notify on underflow, check before rounding)

For each of these, the following axiom shall apply:

$$
\begin{aligned}
chk_F(x,y) &= 0 && \text{if } x = 0 \\
&= \mathbf{overflow} && \text{if } |y| > fmax \text{ and } x \neq 0
\end{aligned}
$$

The checking functions differ when $x \neq 0$ and $|y| \leq fmax$, and the following axioms shall apply:

$$
\begin{aligned}
chk_F^D\ (x,y) &= y \\
chk_F^{Za}(x,y) &= y \quad \text{if } |y| \geq fmin_N \\
&= 0 \quad \text{if } |y| < fmin_N \\
chk_F^{Na}(x,y) &= y \quad \text{if } |y| \geq fmin_N \\
&= \mathbf{underflow} \quad \text{if } |y| < fmin_N \\
chk_F^{Zb}(x,y) &= y \quad \text{if } |x| \geq fmin_N \\
&= 0 \quad \text{if } |x| < fmin_N \\
chk_F^{Nb}(x,y) &= y \quad \text{if } |x| \geq fmin_N \\
&= \mathbf{underflow} \quad \text{if } |x| < fmin_N
\end{aligned}
$$

chk_F^D shall only be provided when F is a denormalized floating point type (i.e., $denorm = \mathbf{true}$).

A processor shall provide at least one of chk_F^D, chk_F^{Za}, or chk_F^{Zb}. One of these shall be the default.

If chk_F^D is provided, chk_F^{Na} or chk_F^{Nb} should also be provided. If chk_F^{Za} is provided, chk_F^{Na} should also be provided. If chk_F^{Zb} is provided, chk_F^{Nb} should also be provided.

Each choice of range checking function shall produce a distinct family of floating point operations.

NOTES

1 The function $chk_F(x, y)$ determines the final result of a floating point operation based on a before-rounding value x and an after-rounding value y.

2 chk_F^D is the only checking function that produces denormalized values.

9.1.3.3 Floating point approximate-addition function

A floating point approximate-addition function is defined:

$$add_F^* : F \times F \to \mathcal{R}$$

For all values u, v, x, and y in F, and i in $\mathcal{Z}$, the following axioms shall apply:

$$add_F^*(u, v) = add_F^*(v, u)$$

$$x \le u + v \le y \;\Rightarrow\; x \le rnd_F(add_F^*(u, v)) \le y$$

$$u \le v \;\Rightarrow\; add_F^*(u, x) \le add_F^*(v, x)$$

If u, v, $u * r^i$, and $v * r^i$ are all in F_N,
$$add_F^*(u * r^i, v * r^i) = add_F^*(u, v) * r^i$$

The approximate-addition function should satisfy

$$add_F^*(x, y) = x + y$$

which trivially satisfies the above axioms.

More than one approximate addition function can be provided.

If $add_F^*(x, y)$ is not always equal to $x + y$, then chk_F^{Zb} and chk_F^{Nb} shall not be provided.

If multiple rounding or range checking functions are provided, every combination of rounding function, range checking function, and operation shall be supported. The choice of approximate addition function may be coupled to the choice of rounding function.

9.1.3.4 Floating point exponent function

A floating point exponent function is defined:

$$e_F : F \to \mathcal{Z}$$

For all values x in F, the following axiom shall apply:

$$\begin{aligned} e_F(x) &= \lfloor log_r|x| \rfloor + 1 && \text{if } |x| \ge fmin_N \\ &= emin && \text{if } |x| < fmin_N \end{aligned}$$

9.1.3.5 Floating point reduce-accuracy-and-round function

A floating point reduce-accuracy-and-round function is defined:

$$rn_F : F \times \mathcal{Z} \to F^*$$

For all values x in F, and n in $\mathcal{Z}$, the following axiom shall apply:

$$\begin{aligned} &rn_F(x, n) \\ &\quad = sign_F(x) * \lfloor |x|/r^{e_F(x)-n} + 1/2 \rfloor * r^{e_F(x)-n} \end{aligned}$$

9.1.4 Mixed mode operations and axioms

These operations convert the integer operand or operands to floating point and then use the appropriate floating point operation.

The following operations are specified:

$$\begin{aligned} &add_{FI} : F \times I \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\} \\ &add_{IF} : I \times F \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\} \\ &sub_{FI} : F \times I \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\} \\ &sub_{IF} : I \times F \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\} \\ &mul_{FI} : F \times I \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\} \\ &mul_{IF} : I \times F \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\} \\ &div_{FI} : F \times I \\ &\quad \to F \cup \{\mathbf{overflow}, \mathbf{underflow}, \mathbf{zero\_divisor}\} \\ &div_{IF} : I \times F \\ &\quad \to F \cup \{\mathbf{overflow}, \mathbf{underflow}, \mathbf{zero\_divisor}\} \\ &div_{II} : I \times I \\ &\quad \to F \cup \{\mathbf{overflow}, \mathbf{underflow}, \mathbf{zero\_divisor}\} \\ &sqrt_I : I \to F \cup \{\mathbf{overflow}, \mathbf{undefined}\} \\ &exponent_I : I \to I \cup \{\mathbf{overflow}, \mathbf{undefined}\} \\ &fraction_I : I \to F \cup \{\mathbf{overflow}, \mathbf{undefined}\} \\ &scale_I : I \times I \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\} \end{aligned}$$

For all values x and y in F, and m and n in I, the following axioms shall apply:

$$add_{FI}(x, n) = add_F(x, float_{I \to F}(n))$$

$$add_{IF}(n, y) = add_F(float_{I \to F}(n), y)$$

$$sub_{FI}(x,n) = sub_F(x, float_{I\to F}(n))$$
$$sub_{IF}(n,y) = sub_F(float_{I\to F}(n), y)$$
$$mul_{FI}(x,n) = mul_F(x, float_{I\to F}(n))$$
$$mul_{IF}(n,y) = mul_F(float_{I\to F}(n), y)$$
$$div_{FI}(x,n) = div_F(x, float_{I\to F}(n))$$
$$div_{IF}(n,y) = div_F(float_{I\to F}(n), y)$$
$$div_{II}(n,m) = div_F(float_{I\to F}(n), float_{I\to F}(m))$$
$$sqrt_I(n) = sqrt_F(float_{I\to F}(n))$$
$$exponent_I(n) = exponent_F(float_{I\to F}(n))$$
$$fraction_I(n) = fraction_F(float_{I\to F}(n))$$
$$scale_I(n,m) = scale_F(float_{I\to F}(n), m)$$

NOTE — A floating point value is never implicitly converted to an integer. The programmer must state which conversion is to applied, see 9.1.5.1.

9.1.5 Type conversion operations

The following functions are specified to convert a value from integer type I to floating point type F, and vice versa. The relevant rounding functions are defined in 9.1.3.2 and 9.1.5.1.

$float_{I\to F} : I \to F \cup \{\mathbf{overflow}\}$
$float_{F\to F} : F \to F$
$floor_{F\to I} : F \to I \cup \{\mathbf{overflow}\}$
$truncate_{F\to I} : F \to I \cup \{\mathbf{overflow}\}$
$round_{F\to I} : F \to I \cup \{\mathbf{overflow}\}$
$ceiling_{F\to I} : F \to I \cup \{\mathbf{overflow}\}$

For all values x in F, and n in I, the following axioms shall apply:

$$float_{I\to F}(n) = chk_F(n,\ rnd_F(n))$$

$$float_{F\to F}(x) = x$$

$$floor_{F\to I}(x) = floor_{\mathcal{R}\to\mathcal{Z}}(x) \quad \text{if } floor_{\mathcal{R}\to\mathcal{Z}}(x) \in I$$
$$= \mathbf{overflow} \quad \text{if } floor_{\mathcal{R}\to\mathcal{Z}}(x) \notin I$$

$$truncate_{F\to I}(x) = truncate_{\mathcal{R}\to\mathcal{Z}}(x) \quad \text{if } truncate_{\mathcal{R}\to\mathcal{Z}}(x) \in I$$
$$= \mathbf{overflow} \quad \text{if } truncate_{\mathcal{R}\to\mathcal{Z}}(x) \notin I$$

$$round_{F\to I}(x) = round_{\mathcal{R}\to\mathcal{Z}}(x) \quad \text{if } round_{\mathcal{R}\to\mathcal{Z}}(x) \in I$$
$$= \mathbf{overflow} \quad \text{if } round_{\mathcal{R}\to\mathcal{Z}}(x) \notin I$$

$$ceiling_{F\to I}(x) = ceiling_{\mathcal{R}\to\mathcal{Z}}(x) \quad \text{if } ceiling_{\mathcal{R}\to\mathcal{Z}}(x) \in I$$
$$= \mathbf{overflow} \quad \text{if } ceiling_{\mathcal{R}\to\mathcal{Z}}(x) \notin I$$

9.1.5.1 Floating point to integer rounding functions

The following rounding functions are specified:

$floor_{\mathcal{R}\to\mathcal{Z}} : \mathcal{R} \to \mathcal{Z}$
$truncate_{\mathcal{R}\to\mathcal{Z}} : \mathcal{R} \to \mathcal{Z}$
$round_{\mathcal{R}\to\mathcal{Z}} : \mathcal{R} \to \mathcal{Z}$
$ceiling_{\mathcal{R}\to\mathcal{Z}} : \mathcal{R} \to \mathcal{Z}$

For all values x in $\mathcal{R}$, and n in $\mathcal{Z}$, the following axioms shall apply:

$$floor_{\mathcal{R}\to\mathcal{Z}}(x) = \lfloor x \rfloor$$

$$truncate_{\mathcal{R}\to\mathcal{Z}}(x) = \lfloor x \rfloor \quad \text{if } x \geq 0$$
$$= -\lfloor |x| \rfloor \quad \text{if } x < 0$$

$$round_{\mathcal{R}\to\mathcal{Z}}(x) = \lfloor x + 1/2 \rfloor$$

$$ceiling_{\mathcal{R}\to\mathcal{Z}}(x) = -\lfloor -x \rfloor$$

9.1.6 Exceptional values

It shall be an error when the result of a simple arithmetic functor is an exceptional value (see 7.9.4).

NOTE — A conforming implementation of IS 10967 (LIA) must notify the user when any arithmetic operation returns an exceptional value, or when resource exhaustion occurs. Such notification may be alteration of the control flow of the program such that execution can be continued only by explicit action within the program. The notification process should include identification of the kind of violation causing the notification, and the operation responsible for the violation.

The process called "notification" in IS 10967 (LIA) is implemented as an error in this draft International Standard.

9.1.7 Examples

```
X is '+'(7, 35).
   Succeeds, unifying X with 42.

X is '+'(0, 3+11).
   Succeeds, unifying X with 14.

X is '+'(0, 3.2+11).
   Succeeds, unifying X with a value
   approximately equal to 14.2000.
```

```
X is '+'(77, N).
   instantiation_error.

X is '+'(foo, 77).
   type_error(number, foo).

current_prolog_flag(max_integer, MI),
  X is '+'(MI, 1).
   calculation_error(overflow).

X is '-'(7).
   Succeeds, unifying X with -7.

X is '-'(3-11).
   Succeeds, unifying X with 8.

X is '-'(3.2-11).
   Succeeds, unifying X with a value
   approximately equal to 7.8000.

X is '-'(N).
   instantiation_error.

X is '-'(foo).
   type_error(number, foo).

X is '-'(7, 35).
   Succeeds, unifying X with -28.

X is '-'(20, 3+11).
   Succeeds, unifying X with 6.

X is '-'(0, 3.2+11).
   Succeeds, unifying X with a value
   approximately equal to -14.2000.

X is '-'(77, N).
   instantiation_error.

X is '-'(foo, 77).
   type_error(number, foo).

read_prolog_flag(max_integer, MI),
  X is '-'(-1, MI).
   calculation_error(overflow).

X is '*'(7, 35).
   Succeeds, unifying X with 245.

X is '*'(0, 3+11).
   Succeeds, unifying X with 0.

X is '*'(1.5, 3.2+11).
   Succeeds, unifying X with a value
   approximately equal to 21.3000.

X is '*'(77, N).
   instantiation_error.

X is '*'(foo, 77).
   type_error(number, foo).

read_prolog_flag(max_integer, MI),
  X is '*'(MI, 2).
   calculation_error(overflow).

X is '/'(7, 35).
   Succeeds, unifying X with 0.

X is '/'(7.0, 35).
   Succeeds, unifying X with a value
   approximately equal to 0.2000.

X is '/'(140, 3+11).
   Succeeds, unifying X with 10.

X is '/'(20.164, 3.2+11).
   Succeeds, unifying X with a value
   approximately equal to 14.2000.

X is '/'(7, -3).
   Succeeds, unifying X with an
   implementation defined value.

X is '/'(-7, 3).
   Succeeds, unifying X with an
   implementation defined value.

X is '/'(77, N).
   instantiation_error.

X is '/'(foo, 77).
   type_error(number, foo).

X is '/'(3, 0).
   calculation_error(zero_divide).

X is mod(7, 3).
   Succeeds, unifying X with 1.

X is mod(0, 3+11).
   Succeeds, unifying X with 0.

X is mod(77, N).
   instantiation_error.

X is mod(foo, 77).
   type_error(number, foo).

X is mod(7.5, 2).
   type_error(7.5, integer).

X is mod(7, 0).
   calculation_error(zero_divide).

X is mod(7, -2).
   calculation_error(undefined).

X is floor(7.4).
   Succeeds, unifying X with 7.

X is floor(-0.4).
   Succeeds, unifying X with -1.

X is round(7.5).
   Succeeds, unifying X with 8.

X is round(7.6).
   Succeeds, unifying X with 8.

X is round(-0.6).
   Succeeds, unifying X with -1.

X is round(N).
   instantiation_error.

X is ceiling(-0.5).
   Succeeds, unifying X with 0.

X is truncate(-0.5).
   Succeeds, unifying X with 0.
```

```
X is truncate(foo).
   type_error(number, foo).

current_prolog_flag(max_integer, MI),
  R is float(MI) * 2,
  X is floor(R).
   calculation_error(overflow).

X is float(7).
   Succeeds, unifying X with 7.0.

X is float(7.3).
   Succeeds, unifying X with 7.3.

X is float(5 / 3).
   Succeeds, unifying X with 1.0.

X is float(N).
   instantiation_error.

X is float(foo).
   type_error(number, foo).

X is abs(7).
   Succeeds, unifying X with 7.

X is abs(3-11).
   Succeeds, unifying X with 8.

X is abs(3.2-11.0).
   Succeeds, unifying X with a value
   approximately equal to 7.8000.

X is abs(N).
   instantiation_error.

X is abs(foo).
   type_error(number, foo).

X is sqrt(0.0).
   Succeeds, unifying X with 0.0.

X is sqrt(4.0).
   Succeeds, unifying X with a value
   approximately equal to 2.0000.

X is sqrt(0).
   Succeeds, unifying X with 0.0.

X is sqrt(1.0).
   Succeeds, unifying X with 1.0.

X is sqrt(N).
   instantiation_error.

X is sqrt(foo).
   type_error(number, foo).

X is sqrt(-1.0).
   calculation_error(undefined).
```

9.2 The format of other evaluable functor definitions

9.2.1 Description

A mathematical description of the result of the evaluable functor.

A evaluable functor is not re-executable.

9.2.2 Template and modes

A specification for the type of the expressions which are the arguments of the evaluable functor. The cases form a mutually exclusive set.

Notation for the structure and type of the arguments and result:

a) int-exp — integer expression,

b) integer — integer value,

c) float-exp — floating point expression,

d) float — floating point value,

A note reminds readers when a functor name of an evaluable functor is a predefined operator, for example `'+'` is an infix predefined operator (see 6.3.4.4).

9.2.2.1 Examples

```
sin(float-exp) = float

'<<'(int-exp, int-exp) = integer
```

9.2.3 Errors

A list of the circumstances that will cause an error when the evaluable functor is executed, together with the sort of error that is caused.

9.2.4 Examples

An example is normally the evaluable functor used in a goal

```
X is functor(Argument, argument)
```

together with a statement saying whether the goal succeeds and unifies a value with `X`, fails or produces an error.

9.3 Other arithmetic functors

9.3.1 ** /2 – power

9.3.1.1 Description

'**'(X, Y) evaluates the expressions X and Y with values VX and VY and produces as result the value of VX raised to the power of VY. If VX and VY are both zero, the result is 1.0.

9.3.1.2 Template and modes

```
'**'(int-exp, int-exp) = float
'**'(float-exp, int-exp) = float
'**'(int-exp, float-exp) = float
'**'(float-exp, float-exp) = float
```

NOTE — '**' is an infix predefined operator (see 6.3.4.4).

9.3.1.3 Errors

a) X is a variable
— instantiation_error.

b) Y is a variable
— instantiation_error.

c) VX is negative and Y is not an integer expression
— calculation_error(undefined).

d) VX is zero and VY is negative
— calculation_error(undefined).

e) The result is too large
— calculation_error(overflow).

9.3.1.4 Examples

```
X is '**'(5, 3).
   Succeeds, unifying X with a value
   approximately equal to 125.0000.

X is '**'(-5.0, 3).
   Succeeds, unifying X with a value
   approximately equal to -125.0000.

X is '**'(5, -1).
   Succeeds, unifying X with a value
   approximately equal to 0.2000.

X is '**'(77, N).
   instantiation_error.

X is '**'(foo, 2).
   type_error(number, foo).

X is '**'(5, 3.0).
   Succeeds, unifying X with a value
   approximately equal to 125.0000.

X is '**'(0.0, 0).
   Succeeds, unifying X with a value
   approximately equal to 1.0.
```

9.3.2 sin/1

9.3.2.1 Description

sin(X) evaluates the expression X with value VX and produces as result the sine of VX (measured in radians).

9.3.2.2 Template and modes

```
sin(float-exp) = float
sin(int-exp) = float
```

9.3.2.3 Errors

a) X is a variable
— instantiation_error.

b) VX is neither a variable nor a real
— type_error(real, VX).

NOTE — The result has little or no significance if VX has a large magnitude.

9.3.2.4 Examples

```
X is sin(0.0).
   Succeeds, unifying X with 0.0.

PI is atan(1.0) * 4,
  X is sin(PI / 2.0).
   Succeeds, unifying X and PI with values
   approximately equal to 1.0000 and 3.14159.

X is sin(N).
   instantiation_error.

X is sin(0).
   Succeeds, unifying X with 0.0.

X is sin(foo).
   type_error(number, foo).
```

9.3.3 cos/1

9.3.3.1 Description

cos(X) evaluates the expression X with value VX and produces as result the cosine of VX (measured in radians).

9.3.3.2 Template and modes

```
cos(float-exp) = float
cos(int-exp) = float
```

9.3.3.3 Errors

a) X is a variable
— instantiation_error.

b) VX is neither a variable nor a real
— type_error(real, VX).

NOTE — The result has little or no significance if VX has a large magnitude.

9.3.3.4 Examples

```
X is cos(0.0).
   Succeeds, unifying X with 1.0.

PI is atan(1.0) * 4,
  X is cos(PI / 2.0).
   Succeeds, unifying X and PI with values
   approximately equal to 0.0000 and 3.14159.

X is cos(N).
   instantiation_error.

X is cos(0).
   Succeeds, unifying X with 1.0.

X is cos(foo).
   type_error(number, foo).
```

9.3.4 atan/1

9.3.4.1 Description

atan(X) evaluates the expression X with value VX and produces as result the principal value of the arc tangent of VX, that is, the result R satisfies
$-\pi/2 \leq R \leq \pi/2$

9.3.4.2 Template and modes

```
atan(float-exp) = float
atan(int-exp) = float
```

9.3.4.3 Errors

a) X is a variable
— instantiation_error.

b) VX is neither a variable nor a real
— type_error(real, VX).

9.3.4.4 Examples

```
X is atan(0.0).
   Succeeds, unifying X with 0.0.

PI is atan(1.0) * 4.
   Succeeds, unifying PI with a value
   approximately equal to 3.14159.

X is atan(N).
   instantiation_error.

X is atan(0).
   Succeeds, unifying X with 0.0.

X is atan(foo).
   type_error(number, foo).
```

9.3.5 exp/1

9.3.5.1 Description

exp(X) evaluates the expression X with value VX and produces as result the value of the exponential function of VX.

9.3.5.2 Template and modes

```
exp(float-exp) = float
exp(int-exp) = float
```

9.3.5.3 Errors

a) X is a variable
— instantiation_error.

b) VX is neither a variable nor a real
— type_error(real, VX).

c) Failure during calculation
— calculation_error(overflow).

9.3.5.4 Examples

```
X is exp(0.0).
   Succeeds, unifying X with 0.0.

X is exp(1.0).
   Succeeds, unifying X with a value
   approximately equal to 2.7818.

X is exp(N).
   instantiation_error.
```

```
X is exp(0).
   Succeeds, unifying X with 0.0.

X is exp(foo).
   type_error(number, foo).
```

9.3.6 log/1

9.3.6.1 Description

`log(X)` evaluates the expression `X` with value `VX` and produces as result the value of the natural logarithm of `VX`.

9.3.6.2 Template and modes

```
log(float-exp) = float
log(int-exp) = float
```

9.3.6.3 Errors

a) `X` is a variable
— `instantiation_error`.

b) `VX` is neither a variable nor a real
— `type_error(real, VX)`.

c) `VX` is zero or negative
— `calculation_error(undefined)`.

9.3.6.4 Examples

```
X is log(1.0).
   Succeeds, unifying X with 0.0.

X is log(2.7818).
   Succeeds, unifying X with a value
   approximately equal to 1.0000.

X is log(N).
   instantiation_error.

X is log(0).
   calculation\_error(undefined).

X is log(foo).
   type_error(number, foo).

X is log(0.0).
   Range error.
```

9.4 Logical functors

9.4.1 ¿¿ /2 – bitwise right shift

9.4.1.1 Description

`'>>'(N, S)` evaluates the expressions `N` and `S` with values `VN` and `VS` and produces as result the value of `VN` right-shifted `VS` bit positions.

The result shall be implementation defined depending on whether the shift is logical (fill with zeros) or arithmetic (fill with a copy of the sign bit).

The result shall be implementation defined if `VS` is negative, or `VS` is larger than the bit size of an integer.

9.4.1.2 Template and modes

```
'>>'(int-exp, int-exp) = integer
```

NOTE — `'>>'` is an infix predefined operator (see 6.3.4.4).

9.4.1.3 Errors

a) `N` is a variable
— `instantiation_error`.

b) `S` is a variable
— `instantiation_error`.

c) `VN` is neither a variable nor an integer
— `type_error(integer, VN)`.

d) `VS` is neither a variable nor an integer
— `type_error(integer, VS)`.

9.4.1.4 Examples

```
X is '>>'(16, 2).
   Succeeds, unifying X with 4.

X is '>>'(19, 2).
   Succeeds, unifying X with 4.

X is '>>'(-16, 2).
   Succeeds, unifying X with an
   implementation defined value.

X is '>>'(77, N).
   instantiation_error.

X is '>>'(foo, 2).
   type_error(integer, foo).
```

9.4.2 ¡¡ /2 – bitwise left shift

9.4.2.1 Description

'<<'(N, S) evaluates the expressions N and S with values VN and VS and produces as result the value VN left-shifted VS bit positions, where the VS least significant bit positions of the result are zero.

The result shall be implementation defined if VS is negative, or VS is larger than the bit size of an integer.

9.4.2.2 Template and modes

```
'<<'(int-exp, int-exp) = integer
```

NOTE — '<<' is an infix predefined operator (see 6.3.4.4).

9.4.2.3 Errors

a) N is a variable
— instantiation_error.

b) S is a variable
— instantiation_error.

c) VN is neither a variable nor an integer
— type_error(integer, VN).

d) VS is neither a variable nor an integer
— type_error(integer, VS).

9.4.2.4 Examples

```
X is '<<'(16, 2).
   Succeeds, unifying X with 64.

X is '<<'(19, 2).
   Succeeds, unifying X with 76.

X is '>>'(-16, 2).
   Succeeds, unifying X with an
   implementation defined value.

X is '<<'(77, N).
   instantiation_error.

X is '<<'(foo, 2).
   type_error(integer, foo).
```

9.4.3 /\ /2 – bitwise and

9.4.3.1 Description

'/\\'(B1, B2) evaluates the expressions B1 and B2 with values VB1 and VB2 and produces as result the value such that each bit is set iff each of the corresponding bits in VB1 and VB2 is set.

The result shall be implementation defined if VB1 or VB2 is negative.

9.4.3.2 Template and modes

```
'/\\'(int-exp, int-exp) = integer
```

NOTE — '/\\' is an infix predefined operator (see 6.3.4.4).

9.4.3.3 Errors

a) B1 is a variable
— instantiation_error.

b) B2 is a variable
— instantiation_error.

c) VB1 is neither a variable nor an integer
— type_error(integer, VB1).

d) VB2 is neither a variable nor an integer
— type_error(integer, VB2).

9.4.3.4 Examples

```
X is '/\\'(10, 12).
   Succeeds, unifying X with 8.

X is /\(10, 12).
   Succeeds, unifying X with 8.

X is '/\\'(17 * 256 + 125, 255).
   Succeeds, unifying X with 125.

X is /\(-10, 12).
   Succeeds, unifying X with an
   implementation defined value.

X is '/\\'(77, N).
   instantiation_error.

X is '/\\'(foo, 2).
   type_error(integer, foo).
```

9.4.4 \/ /2 – bitwise or

9.4.4.1 Description

'\\/'(B1, B2) evaluates the expressions B1 and B2 with values VB1 and VB2 and produces as result the value such that each bit is set iff at least one of the corresponding bits in VB1 and VB2 is set.

The result shall be implementation defined if VB1 or VB2 is negative.

9.4.4.2 Template and modes

```
'\\/'(int-exp, int-exp) = integer
```

NOTE — '\\/' is an infix predefined operator (see 6.3.4.4).

9.4.4.3 Errors

a) B1 is a variable
— instantiation_error.

b) B2 is a variable
— instantiation_error.

c) VB1 is neither a variable nor an integer
— type_error(integer, VB1).

d) VB2 is neither a variable nor an integer
— type_error(integer, VB2).

9.4.4.4 Examples

```
X is '\\/'(10, 12).
   Succeeds, unifying X with 14.

X is \/(10, 12).
   Succeeds, unifying X with 14.

X is '\\/'(125, 255).
   Succeeds, unifying X with 255.

X is \/(-10, 12).
   Succeeds, unifying X with an
   implementation defined value.

X is '\\/'(77, N).
   instantiation_error.

X is '\\/'(foo, 2).
   type_error(integer, foo).
```

9.4.5 \ /1 – bitwise complement

9.4.5.1 Description

'\\'(B1) evaluates the expression B1 with value VB1 and produces as result the value such that each bit is set iff the corresponding bit in VB1 is not set.

The result shall be implementation defined.

9.4.5.2 Template and modes

```
'\\'(int-exp) = integer
```

NOTE — '\\' is a prefix predefined operator (see 6.3.4.4).

9.4.5.3 Errors

a) B1 is a variable
— instantiation_error.

b) VB1 is neither a variable nor an integer
— type_error(integer, VB1).

9.4.5.4 Examples

```
X is '\\'( '\\'(10)).
   Succeeds, unifying X with 10.

X is \( \(10)).
   Succeeds, unifying X with 10.

X is \(10).
   Succeeds, unifying X with an
   implementation defined value.

X is '\\'(N).
   instantiation_error.

X is '\\'( 2).
   type_error(integer, foo).
```

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).

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).
```

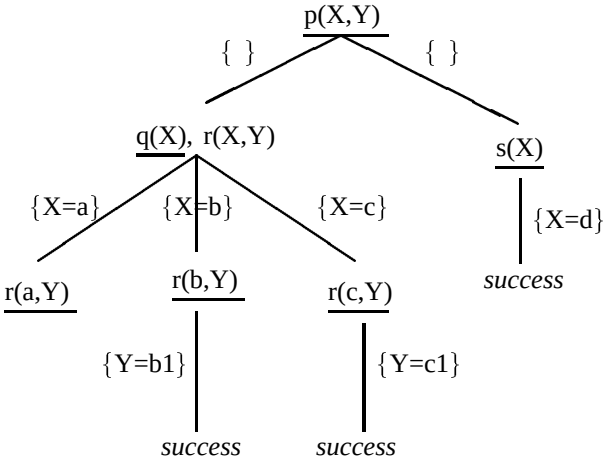

Figure A.1 — A search-tree example

```
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.

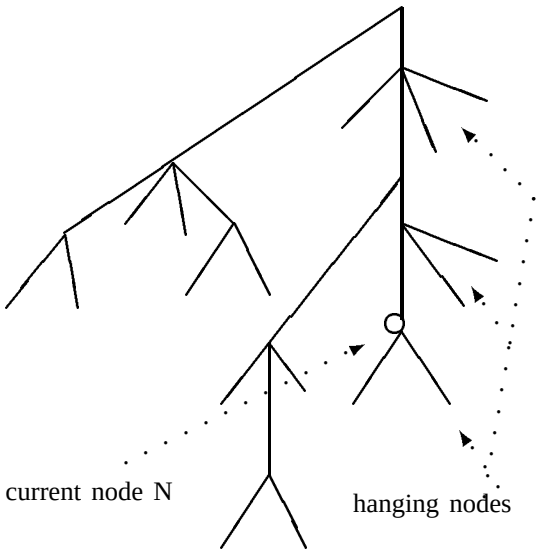

Figure A.2 — A visited search-tree

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**.

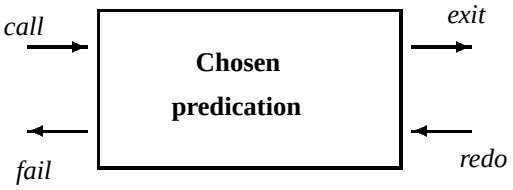

Figure A.3 — Byrd's trace model

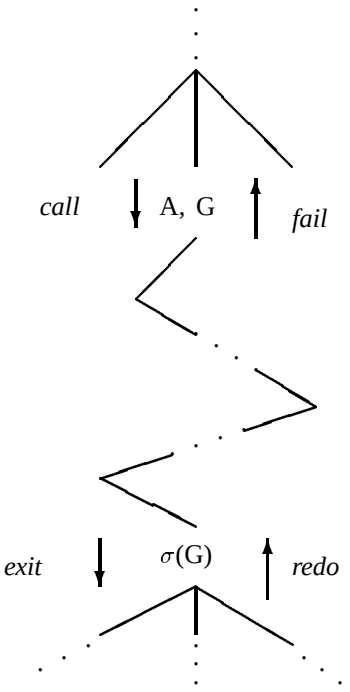

Figure A.4 — Byrd's model: a search-tree point of view

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 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 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.

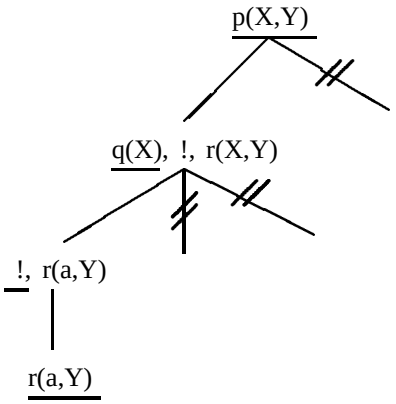

Figure A.5 — A search-tree example showing the effect of cut

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

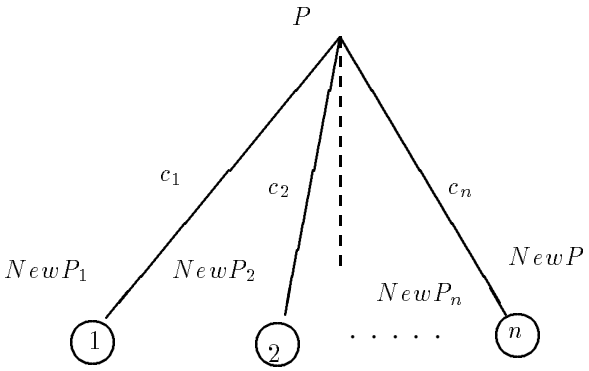

Figure A.6 — Standard database update view

(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. 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.

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

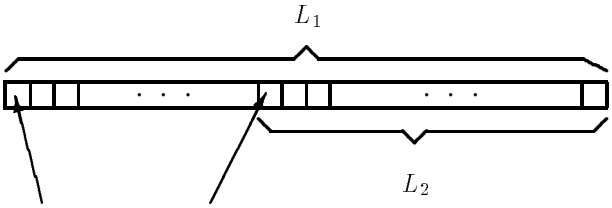

Figure A.7 — $L_1 - L_2$: Difference list of characters

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, 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

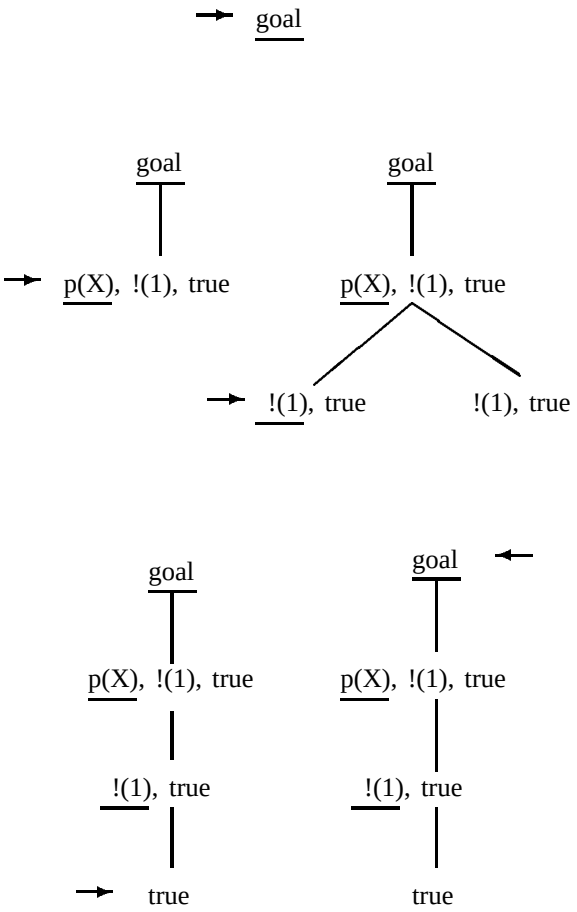

Figure A.8 — Partially visited search-tree

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* DB 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 6.1.2b.

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

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

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*(func(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);

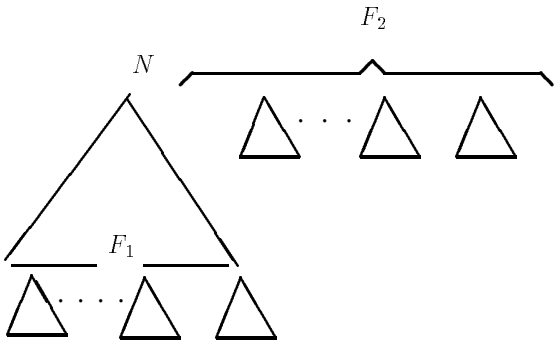

Figure A.9 — The non-empty forest: $for(N, F_1, F_2)$

— 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 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)* ⇐
D-is-a-logic-control-bip*(B)*.

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

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

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

D-is-an-input-output-bip*(B)* ⇐
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))* ⇐
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 bootstrapped 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(*`&`*, 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,` *Op1*`, Exp2).`

with *Op1* ∈ { `=:=` ,`=\=` , `<` , `>` , `=<` , `=>` , }.

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), emp-subs)* ⇐
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), emp-subs)* ⇐
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,

---