INTERNATIONAL STANDARD

ISO/IEC 13211-1

First edition
1995-06-01

Information technology — Programming languages — Prolog —

Part 1: General core

Technologies de l'information — Langages de programmation — Prolog —

Partie 1: Noyau général

Reference number
ISO/IEC 13211-1:1995(E)

Contents

Annex

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC 13211 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Annex A of this part of ISO/IEC 13211 is for information only.

Introduction

This is the first International Standard for Prolog, Part 1 (General Core). It was produced on 20 April 1995.

There is no other International Standard for Prolog.

Prolog (Programming in Logic) combines the concepts of logical and algorithmic programming, and is recognized not just as an important tool in AI (Artificial Intelligence) and expert systems, but as a general purpose high-level programming language with some unique properties.

The language originates from work in the early 1970s by Robert A. Kowalski while at Edinburgh University (and ever since at Imperial College, London) and Alain Colmerauer at the University of Aix-Marseilles in France. Their efforts led in 1972 to the use of formal logic as the basis for a programming language. Kowalski's research provided the theoretical framework, while Colmerauer's gave rise to the programming language Prolog. Colmerauer and his team then built the first interpreter, and David Warren at the AI Department, University of Edinburgh, produced the first compiler.

The crucial features of Prolog are unification and backtracking. Unification shows how two arbitrary structures can be made equal, and Prolog processors employ a search strategy which tries to find a solution to a problem by backtracking to other paths if any one particular search comes to a dead end.

Prolog is good for windowing and multimedia because of the ease of building complex data structures dynamically, and also because the concept of backing out of an operation is built into the language.

Prolog is taught in more UK university computing degrees than any other programming language.

This part of ISO/IEC 13211 defines the general core features of Prolog, and part 2 will define modules.

This page intentionally left blank

Information technology — Programming languages — Prolog — Part 1: General core

1 Scope

ISO/IEC 13211 is designed to promote the applicability and portability of Prolog text and data among a variety of data processing systems.

This part of ISO/IEC 13211 specifies:

a) The representation of Prolog text,

b) The syntax and constraints of the Prolog language,

c) The semantic rules for interpreting Prolog text,

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

e) The representation of output produced by Prolog, and

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

NOTE — This part of ISO/IEC 13211 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 part of ISO/IEC 13211 is intended for use by implementors and knowledgeable programmers, and is not a tutorial.

1.1 Notes

Notes in this part of ISO/IEC 13211 have no effect on the language, Prolog text or Prolog processors that are defined as conforming to this part of ISO/IEC 13211. Reasons for including a note include:

a) Cross references to other clauses and subclauses of this part of ISO/IEC 13211 in order to help readers find their way around,

b) Warnings when a built-in predicate as defined in this part of ISO/IEC 13211 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 part of ISO/IEC 13211. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 13211 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/IEC 646 : 1991, *Information technology — ISO 7-bit coded character set for information interchange.*

ISO 2382-15 : 1985, *Data processing — Vocabulary — Part 15: Programming languages.*

ISO 8859-1 : 1987, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1.*

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

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

ISO/IEC 10967-1 : 1994, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic.*

BS 6154 : 1981, *Method of defining — Syntactic meta-language.*

3 Definitions

This terminology for Prolog has a format modelled on that of ISO 2382.

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 words or phrases defined in separate entries directly following each other (or separated only by a punctuation sign), * (an asterisk) separates them.

Words and phrases not defined in this glossary are assumed to have the meaning given in ISO 2382-15; if they do not appear in ISO 2382-15, then they are assumed to have their usual meaning.

For the purposes of ISO/IEC 13211, the following definitions apply:

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

3.4 algorithm, Herbrand: See 3.85 — **Herbrand algorithm**.

3.5 alias: 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*.

3.6 anonymous variable: A *variable* (represented in a *term* or *Prolog text* by _) which differs from every other *variable* (and anonymous variable) (see 6.1.2, 6.4.3).

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

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

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

3.10 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.

3.11 associativity (of an *operator***):** Property of being non-associative, right-associative, or left-associative (see 6.3.4, table 4).

3.12 atom: A basic object, denoted by an *identifier* (see 6.1.2 b, 7.1.4).

3.13 atom, null: See 3.117 — **null atom**.

3.14 atom, one-char: See 3.119 — **one-char atom**.

3.15 atomic term: An *atom* or a *number*.

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

3.17 backtrack, to: To return to the *choicepoint* of the current *goal* in order to attempt to *re-execute* it (see 7.7.8).

3.18 bias, exponent: See 3.68 — **exponent bias**.

3.19 body: A *goal*, distinguished by its context as part of a *rule* (see 3.154).

3.20 bootstrapped (built-in predicate): Defined as a special case of a more general *built-in predicate* (see 8.1.5).

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

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

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

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

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

3.26 character: A member of C — an *implementation defined* character set (see 6.5, 7.1.4.1).

3.27 character, quoted: See 3.144 — **quoted character**.

3.28 character, unquoted: See 3.194 — **unquoted character**.

3.29 character-conversion mapping: A *mapping* 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 3.46, 7.4.2.5, 8.14.5).

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

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

3.32 clause: A *fact* or a *rule*. It has two parts: a *head*, and a *body*.

NOTE — In ISO/IEC International Standards "clause" has the meaning: one of the numbered paragraphs of a standard. In this part of ISO/IEC 13211, the context distinguishes the two meanings.

3.33 clause-term: A *read-term* `T`. in *Prolog text* where `T` does not have *principal functor* `(:-)/1` (see 6.2.1.2).

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

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

3.36 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 $\sigma_1 \circ \sigma_2$. When the composition acts on a *term* t, it is denoted by $t\sigma_1\sigma_2$, with the meaning $((t\sigma_1)\sigma_2)$.

3.37 compound term: A *functor* of *arity* `N`, `N` positive, together with a sequence of `N` *arguments* (see 6.1.2 e, 7.1.5).

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

3.39 conforming processor: A *processor* which conforms to all the compliance clauses (see 5.1) for *processors* in this part of ISO/IEC 13211.

3.40 conforming Prolog data: Sequences of *characters* and *bytes* that conform to all the compliance clauses for *Prolog data* in this part of ISO/IEC 13211 (see 5, 6.2.2).

3.41 conforming Prolog text: A sequence of *characters* that conforms to all the compliance clauses for *Prolog text* in this part of ISO/IEC 13211 (see 5, 6.2).

3.42 construct, control: See 3.45 — **control construct**.

3.43 constructor, list: See 3.100 — **list constructor**.

3.44 contain, to: A *term* `T1` contains another *term* `T2` if either `T1` and `T2` are *identical terms*, or `T1` is a *compound term*, one of whose *arguments* contains `T2`.

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

3.46 $Conv_C$**:** The *character-conversion mapping* on C (the set of *characters*) which specifies that, in some *Prolog text* units and *sources*, some *characters* are *converted* to other *characters* (see 3.29, 7.4.2.5, 8.14.5).

The initial value of $Conv_C$ shall be $identity\_mapping_C$.

NOTES

1 A directive or goal `char_conversion(In, Out)` (7.4.2.5, 8.14.5) replaces $Conv_C$ by $update\_mapping_C$(`In, Out,` $Conv_C$).

2 Any unquoted character `C` that is part of a read-term which is input by `read_term/3` (8.14.1) or as Prolog text is replaced by $apply\_mapping_C$(`C,` $Conv_C$).

3 $Conv_C$ can be inspected by calling `current_char_conversion/2` (8.14.6).

4 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 Prolog data and Prolog text to be the same regardless of the encoding.

3.47 convert (from type A to type B): An operation whose *signature* is

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

which converts a value of *type* A to *type* B. It shall be an error if the conversion cannot be made.

For example, see converting a *term* to a *clause* and vice versa (7.6), character-conversion (3.29, 7.4.2.5, 8.14.5), and converting a *floating point value* to an *integer value* and vice versa (9.1.6).

3.48 copy, renamed (of a term): See 3.150 — **renamed copy (of a term)**.

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

3.50 cut: A *control construct* whose effect is to remove all *choicepoints* back to a deeper execution state defined by its cutparent (see 7.7.2, 7.8.4).

3.51 data, conforming Prolog: See 3.40 — **conforming Prolog data**.

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

3.53 database, complete: See 3.35 — **complete database**.

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

3.55 data type, arithmetic: See 3.8 — **arithmetic data type**.

3.56 denormalized value: A *floating point value* of type F providing less than the full precision allowed by F (see F_D, 7.1.3).

3.57 directive: A *term* `D` which affects the meaning of *Prolog text* (see 7.4.2), and is denoted in that *Prolog text* by a *directive-term* `:-(D)`.

3.58 directive-term: A *read-term* `T.` in *Prolog text* where `T` has *principal functor* `(:-)/1` (see 6.2.1.1).

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

3.60 effect, side: See 3.157 — **side effect**.

3.61 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.

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

3.63 error: A special circumstance which causes the normal process of *execution* to be interrupted (see 7.12).

3.64 evaluable functor: The *principal functor* of an *expression* (see 7.9, 9).

3.65 evaluate: To reduce an *expression* to its value. (see 7.9, 8.6.1, 9).

3.66 exceptional value: A non-numeric value of an *expression*: **float_overflow**, **int_overflow**, **underflow**, **zero_divisor**, or **undefined** (see 7.9).

NOTE — It is an `evaluation_error(E)` when the value of an *expression* is an exceptional value.

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

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

3.69 expression: An *atomic term* or a *compound term* which may be *evaluated* to produce a value (see 8.6.1, 9).

3.70 extension: A facility provided by the *processor* that is not specified in this part of ISO/IEC 13211 but that would not cause any ambiguity or contradiction if added to this part of ISO/IEC 13211.

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

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

NOTE — A fact can be represented in *Prolog text* by a *term* whose *principal functor* is neither `(:-)/1` nor `(:-)/2`.

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

3.74 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*.

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

3.76 floating point value: A member of the set F (see 6.1.2 d, 7.1.3).

3.77 functor: An *identifier* together with an *arity*.

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

3.79 function, rounding: See 3.153 — **rounding function**.

3.80 functor, principal: See 3.134 — **principal functor**.

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

3.82 ground term: An *atomic term* 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.

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

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

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

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

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

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

3.89 iff: If and only if.

3.90 implementation defined: Defined partly by this part of ISO/IEC 13211, and partly by the documentation accompanying a *processor* (see 5).

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

NOTE — This part of ISO/IEC 13211 does not require an implementation dependent feature to be defined in the accompanying *processor* documentation.

3.92 implementation specific: *Undefined* by this part of ISO/IEC 13211 but supported by a *conforming processor*.

NOTE — This part of ISO/IEC 13211 does not require an implementation specific feature to be supported by a conforming processor, but it preserves the syntax and semantics of a strictly conforming Prolog text which does not use it, for example, defining a term order on variables, or defining unification for terms which are *STO* (3.165).

3.93 indicator, predicate: See 3.131 — **predicate indicator**.

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

3.95 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 $t\sigma$.

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

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

3.97 integer value: A member of the set I (see 6.1.2 c, 7.1.2).

3.98 level, top: See 3.185 — **top level**.

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

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

3.100 list constructor: The *principal functor* `'.'/2` used for constructing *lists*.

3.101 list, empty: See 3.62 — **empty list**.

3.102 list, non-empty: See 3.114 — **non-empty list**.

3.103 list, partial: See 3.125 — **partial list**.

3.104 list, read-options: See 3.147 — **read-options list**.

3.105 list, write-options: See 3.207 — **write-options list**.

3.106 mapping: A *data type* M_T where T is a *data type* (see 4.3).

3.107 mode, input/output: See 3.94 — **input/output mode**.

3.108 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).

3.109 name (of *atom*)**:** A sequence of *characters* which distinguishes an *atom* from any different *atom* (see 6.1.2 b).

3.110 name, file: See 3.74 — **file name**.

3.111 name, functor: See 3.78 — **functor name**.

3.112 name, predicate: See 3.132 — **predicate name**.

3.113 named variable: A *variable* which is not an *anonymous variable* (see 6.1.2 a, 6.4.3).

3.114 non-empty list: A *compound term* whose *principal functor* is the *list constructor* and whose second *argument* is a *list*.

3.115 normalized value: A *floating point value* of type F' providing the full precision allowed by F (see 7.1.3).

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

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

3.118 number: An *integer value* or *floating point value*.

3.119 one-char atom: An *atom* whose name is a single *character*.

3.120 operand (of a *compound term* **or** *predication*)**:** An *argument* of a *compound term* (*predication*) whose *functor name* (*predicate name*) is an *operator*.

3.121 operand (of an operation): A value supplied to an operation defined by a *signature* and one or more *axioms*.

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

3.123 operator, predefined: See 3.128 — **predefined operator**.

3.124 options, stream: See 3.167 — **stream-options**.

3.125 partial list: A variable, or a *compound term* whose *principal functor* is the *list constructor* and whose second argument is a partial list.

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

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

3.126 position, stream: See 3.168 — **stream position.**

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

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

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

3.130 predicate, built-in: See 3.21 — **built-in predicate.**

3.131 predicate indicator: A *compound term* `A/N`, where `A` is an *atom* and `N` is a non-negative integer, denoting one particular *procedure* (see 7.1.6.6).

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

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

3.134 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 an *atomic term* is `C/0` if the *atomic term* is `C`.

3.135 private (of a *procedure*): A private *procedure* is one whose *clauses* cannot be inspected during *execution.* (see 7.5.3).

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

3.137 procedure, user-defined: See 3.195 — **user-defined procedure.**

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

3.139 processor, conforming: See 3.39 — **conforming processor.**

3.140 Prolog data: A sequence of *read-terms* (see 6.2.2).

3.141 Prolog text: A sequence of *read-terms* denoting *directives* and *clauses* (see 6.2, 7.4).

3.142 public (of a *procedure*): A public *procedure* is one whose *clauses* can be inspected during *execution*, for example by calling the *built-in predicate* `clause/2` (see 7.5.3, 8.8.1).

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

NOTE — This part of ISO/IEC 13211 does not define or require a *processor* to support the concept of *top level.*

3.144 quoted character: A *character* in *Prolog text* or *Prolog data* which is a `single quoted character` or a `double quoted character` or a `back quoted character` (see 6.4.2.1).

NOTE — For example, `'a''b\'c'` contains 5 quoted characters (1) a, (2) ', (3) b, (4) ' (a meta escape sequence), (5) c.

3.145 $\mathcal{R}$**:** The set of real numbers (see 4.1.1).

3.146 read-option: A *compound term* with *uninstantiated* * *arguments* which amplifies the results produced by the *built-in predicate* `read_term/3` (8.14.1) and the *bootstrapped* * *built-in predicates* based on it (see 7.10.3).

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

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

3.149 re-execute, to: To re-execute a *goal* is to attempt to *satisfy* it again (see 7.7.6, 7.7.8).

3.150 renamed copy: (of a *term*) A special *variant* of a *term* (see 7.1.6.2).

3.151 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.*

3.152 rounding: Computing a representable final value (for an operation) which is close to the exact (but unrepresentable) value for that operation (see 9.1.3.1, 9.1.4.1).

3.153 rounding function: A function with *signature*: $rnd : \mathcal{R} \rightarrow X$ (where X is a discrete subset of $\mathcal{R}$) which maps each member 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.

3.154 rule: A *clause* whose *body* is not the *goal* `true`. 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*.

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

3.156 sequence, collating: See 3.34 — **collating sequence**.

3.157 side effect: A non-logical effect of an *activator* during *execution* (see 7.7.9).

3.158 signature: A specification of an *operation* which defines its name, and the *type* of its *operands(s)* and value.

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

For example, the signature:

$$add_I : I \times I \rightarrow I \cup \{\textbf{int\_overflow}\}$$

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

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

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

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

3.162 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).

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

3.164 static (of a *procedure*): A static *procedure* is one whose *clauses* cannot be altered (see 7.5.2).

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

3.166 stream: A connection to a *source* or *sink* (see 7.10.2).

3.167 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).

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

3.169 stream, target: See 7.10.2.5 — **Target stream**.

3.170 stream-term: An *implementation dependent* * *ground term* which identifies a *stream* inside *Prolog text* (see 7.10.2.1).

3.171 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, Z → b }`
Term	T	`foo(X, Y, Z)`
New term	Tσ	`foo(a, 3 + Z, b)`

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

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

3.174 target stream: See 7.10.2.5 — **Target stream**.

3.175 term: An *atomic term*, a *compound term* or a *variable* (see 7.1).

3.176 term, atomic: See 3.15 — **atomic term**.

3.177 term, callable: See 3.24 — **callable term**.

3.178 term, compound: See 3.37 — **compound term**.

3.179 term, ground: See 3.82 — **ground term**.

3.180 terms, identical: See 3.87 — **identical terms**.

3.181 term-precedes: A binary relation on the set of terms which defines a total ordering of terms (see 7.2).

3.182 term, stream: See 3.170 — **stream-term**.

3.183 text, conforming Prolog: See 3.41 — **conforming Prolog text**.

3.184 text, Prolog: See 3.141 — **Prolog text**.

3.185 top level: A process whereby a Prolog *processor* repeatedly inputs and *executes* * *queries*.

NOTE — This part of ISO/IEC 13211 does not define or require a *processor* to support the concept of *top level*.

3.186 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).

3.187 type, data: See 3.54 — **data type**.

3.188 undefined: A feature is undefined if this part of ISO/IEC 13211 (1) states it is undefined, or (2) makes no requirements concerning its *execution*.

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

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

3.191 unifier, most general: See 3.108 — **most general unifier**.

3.192 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).

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

3.194 unquoted character: A *character* in *Prolog text* or *Prolog data* which is not a *quoted character* (see 6.4.2.1).

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

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

3.197 value, denormalized: See 3.56 — **denormalized value**.

3.198 value, exceptional: See 3.66 — **exceptional value**.

3.199 value, normalized: See 3.115 — **normalized value**.

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

3.201 variable, anonymous: See 3.6 — **anonymous variable**.

3.202 variable, named: See 3.113 — **named variable**.

3.203 variable set (of a *term*): See 7.1.1.1 — **Variable set of a term**.

3.204 variant (of a *term*): See 7.1.6.1 — **Variants of a term**.

Table 1 — The basic sets

Symbol	Mathematical Type
$\mathcal{R}$	real numbers
$\mathcal{Z}$	integers
$\mathcal{B}$	Boolean (= {**true**, **false**})

3.205 witness (of a set of *variables*): See 7.1.1.2 — **Witness of a variable set**.

3.206 write-option: A *ground term* that controls the output produced by the *built-in predicate* `write_term/3` (8.14.2) and its *bootstrapped* * *built-in predicates* (see 7.10.4, 7.1.4.2).

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

3.208 $\mathcal{Z}$**:** The set of mathematical integers (see 4.1.1).

4 Symbols and abbreviations

The following symbols and abbreviations are used in this part of ISO/IEC 13211.

4.1 Notation

4.1.1 Basic mathematical types

Table 1 defines the notation for the basic mathematical types.

4.1.2 Mathematical and set operators

Table 2 defines the basic operations which have their conventional (exact) mathematical meaning.

The following notation also has its conventional (exact) mathematical meaning:

x^y, $log_x y$, on $\mathcal{R}$

The following set operations also have their conventional mathematical meaning:

$\in$ (member), $\notin$ (not member), $=$ (equality), $\subset$ (subset), $\cup$ (union), $\rightarrow$ (mapping), $\times$ (cartesian product)

Table 2 — Basic mathematical operations

Operator	Signature	Meaning
$\Longleftrightarrow$	$\mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$	equivalence
$\Rightarrow$	$\mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$	implication
$\wedge$	$\mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$	conjunction
$\vee$	$\mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$	disjunction
$\neg$	$\mathcal{B} \rightarrow \mathcal{B}$	negation
$<$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{B}$	less
$\leq$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{B}$	less or equal
$=$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{B}$	equal
$\neq$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{B}$	not equal
$\geq$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{B}$	greater or equal
$>$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{B}$	greater
$+$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$	addition
$-$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$	subtraction
$*$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$	multiplication
$/$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$	division

4.1.3 Other functions

4.1.3.1 Substitution composition

Signature: $\circ$: $Substitution \times Substitution \rightarrow Substitution$

Axiom: $f \circ g = h$ where $h(x) = f(g(x))$

4.1.3.2 $|x|$ – abs x

Signature: $|\ |$: $\mathcal{R} \rightarrow \mathcal{R}$

Axiom: $|x|$ = if $x \geq 0$ then x else $-x$

4.1.3.3 $\lfloor x \rfloor$ – floor x

The notation $\lfloor x \rfloor$ designates the largest integer not greater than x.

Signature: $\lfloor\ \rfloor$: $\mathcal{R} \rightarrow \mathcal{Z}$

Axiom: $\lfloor x \rfloor = n$ where $(x - 1 < n) \wedge (n \leq x)$

4.1.3.4 $tr(x)$ – truncate x

The notation $tr(x)$ designates the integer part of x (truncated towards zero).

Signature: $tr : \mathcal{R} \rightarrow \mathcal{Z}$

Axiom: $tr(x)$ = if $x \geq 0$ then $\lfloor x \rfloor$ else $-\lfloor -x \rfloor$

4.1.3.5 $\sqrt{x}$ – square root

Signature: $\sqrt{} : \mathcal{R} \rightarrow \mathcal{R} \cup \{\textbf{undefined}\}$

Axiom: If $x > 0$ then $\sqrt{x}$ is the positive square root of x else **undefined**

4.1.3.6 Maximum of real set

Signature: $max : \mathcal{R}\text{–set} \rightarrow \mathcal{R}$

Axiom: $max(S) = x$ if $x \in S \wedge x \geq y$ for all $y \in S$

4.1.3.7 Minimum of real set

Signature: $min : \mathcal{R}\text{–set} \rightarrow \mathcal{R}$

Axiom: $min(S) = x$ if $x \in S \wedge x \leq y$ for all $y \in S$

4.2 Abstract data type: stack

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

$push_D : D \times S_D \rightarrow S_D$
$top_D : S_D \rightarrow D \cup \{\textbf{error}\}$
$pop_D : S_D \rightarrow S_D \cup \{\textbf{error}\}$
$newstack_D : \rightarrow S_D$
$isempty_D : S_D \rightarrow Boolean$

For all $d \in D$, $s \in S_D$, the following axioms shall apply:

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

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

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

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

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

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

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

4.3 Abstract data type: mapping

The following functions are specified for a mapping M_T where T is a data type:

$identity\_mapping_T : \rightarrow M_T$
$mapping_T : T \times T \times M_T \rightarrow M_T$
$apply\_mapping_T : T \times M_T \rightarrow T$
$update\_mapping_T : T \times T \times M_T \rightarrow M_T$

For all a, a', b, $b' \in T$, $m \in M_T$, the following axioms shall apply:

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

$apply\_mapping_T(a, mapping_T(a', b, m))$
$= b$ if $a = a'$
$= apply\_mapping_T(a, m)$ if $a \neq a'$

$update\_mapping_T(a, b, identity\_mapping_T)$
$= identity\_mapping_T$ if $a = b$
$= mapping_T(a, b, identity\_mapping_T)$ if $a \neq b$

$update\_mapping_T(a, b, mapping_T(a', b', m))$
$= mapping_T(a', b', update\_mapping_T(a, b, m))$
if $a \neq a'$
$= mapping_T(a, b, m)$
if $a = a'$ and $a \neq b$
$= m$
if $a = a'$ and $a = b$

NOTE — $Conv_C$ (3.46) is a mapping.

5 Compliance

5.1 Prolog processor

A conforming Prolog processor shall:

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

1) the requirements of this part of ISO/IEC 13211, and

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

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

1) the requirements of this part of ISO/IEC 13211, and

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

c) Reject any Prolog text or read-term whose syntax fails to conform to:

1) the requirements of this part of ISO/IEC 13211, and

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

d) Specify all permitted variations from this part of ISO/IEC 13211 in the manner prescribed by this part of ISO/IEC 13211, and

e) Offer a strictly conforming mode which shall reject the use of an implementation specific feature in Prolog text or while executing a goal.

5.2 Prolog text

Conforming Prolog text shall use only the constructs specified in this part of ISO/IEC 13211, and the implementation defined and implementation specific features supported by the processor.

Strictly conforming Prolog text shall use only the constructs specified in this part of ISO/IEC 13211, and the implementation defined features supported by the processor.

5.3 Prolog goal

A conforming Prolog goal is one whose execution is defined by the constructs specified in this part of ISO/IEC 13211, and the implementation defined and implementation specific features supported by the processor.

A strictly conforming Prolog goal is one whose execution is defined by the constructs specified in this part of ISO/IEC 13211, and the implementation defined features supported by the processor.

5.4 Documentation

A conforming Prolog processor shall be accompanied by documentation that completes the definition of every implementation defined and implementation specific feature specified in this part of ISO/IEC 13211.

5.5 Extensions

A processor may support, as an implementation specific feature, any construct that is implicitly or explicitly undefined in the part of ISO/IEC 13211.

5.5.1 Syntax

A processor may support one or more additional characters in PCS (6.5) and additional syntax rules as an implementation specific feature iff:

a) any sequence of tokens that conforms to the syntax of Prolog text and data defined in subclause 6.2 shall have the abstract syntax defined in that subclause,

b) any sequence of tokens that conforms to the syntax of a term defined in subclause 6.3 shall have the abstract syntax defined in that subclause,

c) any sequence of characters that conforms to the syntax of Prolog tokens defined in subclause 6.4 shall be parsed to those Prolog tokens.

NOTE — The presence of an infix and a postfix operator with the same priority is also an allowable extension as an implementation specific feature as long as, like any other syntax extension, it does not change the meaning of Prolog text which conforms to the standard.

5.5.2 Predefined operators

A processor may support one or more additional predefined operators (table 7) as an implementation specific feature.

5.5.3 Character-conversion mapping

A processor may support some other initial value of $Convert$, the character-conversion mapping (3.46), as an implementation specific feature.

5.5.4 Types

A processor may support one or more additional types (7.1) as an implementation specific feature iff, for every additional type T supported by a processor:

a) No term with type T shall also have a type T' where T and T' are different.

b) For every two terms t and t' with types T and T' respectively, t $term\_precedes$ t' (7.2) shall depend only on T and T' unless $T = T'$.

c) The processor shall define in its accompanying documentation the effect of converting a term of type T to a clause (7.6), and vice versa.

d) The processor shall define in its accompanying documentation, the abstract and token syntax of every term of type T.

e) The processor shall define in its accompanying documentation, the effect of evaluating as an expression a term of type T (7.9).

f) The processor shall define in its accompanying documentation, the effect of writing a term of type T (7.10).

5.5.5 Directives

A processor may support one or more additional directive indicators (7.4.2) as an implementation specific feature.

5.5.6 Side effects

A processor may support one or more additional side effects (7.7.9) as an implementation specific feature.

5.5.7 Control constructs

A processor may support one or more additional control constructs (7.8) as an implementation specific feature.

5.5.8 Flags

A processor may support one or more additional flags (7.11) as an implementation specific feature.

5.5.9 Built-in predicates

A processor may support one or more additional built-in predicates (8) as an implementation specific feature.

When a processor supports additional built-in predicates as an implementation specific feature, it may also support as an implementation specific feature one or more additional forms of `Error_term` (7.12.1).

NOTE — The additional forms of `Error_term` may include for example `>=(N)`, `between(N,M)` and `one_of(List)` as valid domains.

5.5.10 Evaluable functors

A processor may support one or more additional evaluable functors (9) as an implementation specific feature. A processor may support the value of an expression being a value of an additional type instead of an exceptional value.

NOTE — A program that makes no use of extensions should not rely on catching errors from procedures that evaluate their arguments (such as `is/2`, 8.6.1) unless it is executed in strictly conforming mode (5.1 e).

5.5.11 Reserved atoms

A processor may reserve some atoms for use in extensions. The effect of executing a goal whose execution causes a variable to be instantiated to a reserved atom or to a compound term whose functor name is a reserved atom is implementation defined.

Table 3 — 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 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. Subclause 6.2 defines how terms form Prolog text and data, subclause 6.3 defines how tokens are combined to form terms, and subclause 6.4 defines how sequences of characters form 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 part of ISO/IEC 13211 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).

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 table 3.

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 associated with the non-terminal symbol defined by the syntax rule is specified in terms of the abstract elements of the symbols forming its definition.

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

Each term and operator is associated with a priority, i.e. an integer between 0 and 1201. An atomic 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 to 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 4).

Condition — One or more additional conditions 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 members are called terms. The enumerable set of terms is defined as the union of disjoint sets which shall include 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 member of V, and

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

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

b) A is a set of atoms such that the name n of $a \in A$ is defined by:

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

2) n is the two characters {} for the empty curly brackets.

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

Letter digit token — The initial `small letter char` followed by each `alphanumeric char`.

Graphic token — Each `graphic token char`.

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

Semicolon token — The character `;`.

Cut token — The character `!`.

The characters of the name of an atom are numbered from one upwards.

c) I is a set of integers (see 7.1.2) such that $i \in I$ is defined for each form of `integer token` (6.4.4) by:

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 an 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 character`.

d) F is a set of floating point values (see 7.1.3) and $f \in F$ is defined for each `float number` by rounding (see 9.1.4.1) 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 — If `float number` has no exponent then zero, else 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.

NOTES

1 A quoted token that contains no `single quoted character` is the null atom.

2 The middle dot (·) denotes associative concatenation of the directives and clauses.

6.2 Prolog text and data

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

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

6.2.1 Prolog text

Prolog text is a sequence of directive-terms and clause-terms.

```
          prolog text = p text ;
Abstract: pt            pt
```

```
          p text = directive term, p text ;
Abstract: d · t    d             t
```

```
          p text = clause term, p text ;
Abstract: c · t    c             l
```

```
          p text = ;
Abstract: nil
```

6.2.1.1 Directives

```
          directive term = term, end ;
Abstract: dt             dt
Priority:                1201
Condition: The principal functor of dt is (:-)/1
```

```
          directive = directive term ;
Abstract: d         :-(d)
```

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

6.2.1.2 Clauses

```
          clause term = term, end ;
Abstract: c           c
Priority:             1201
Condition: The principal functor of c is not (:-)/1
```

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

6.2.2 Prolog data

A Prolog read-term can be read as data by calling the built-in predicate `read_term/3` (8.14.1).

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

NOTE — Any layout text before the term is regarded as part of the first token of the term. A read-term ends with the end token.

6.3 Terms

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

6.3.1 Atomic terms

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 =  name,  integer ;
Abstract: −n      a      n
Priority: 0
Condition: a is −
```

```
          term =  name,  float number ;
Abstract: −r      a      r
Priority: 0
Condition: a is −
```

A term which is the name - followed directly by a numeric constant 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 (3.120) of an operator. The priority of a term consisting of an operator is therefore given the priority 1201 rather than the normal 0.

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

```
          atom = open list, close list ;
Abstract: [ ]
```

```
          atom = open curly, close curly ;
Abstract: {}
```

An atom is a name, or `[]` (the empty list) , or {} (the empty curly brackets).

NOTE — An atom which is an operator needs to be bracketed in order to denote a term of priority 0.

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 in list notation (6.3.5), and sometimes it can be expressed as a double quoted list (6.3.7). When the principal functor is `{}/1` it can also be expressed as a curly bracketed term (6.3.6).

Functional notation is a subset of the Prolog syntax in which all compound terms can be expressed.

A compound term written in functional notation has the form `f(A1,...,An)` where each argument `Ai` is an `arg` and they are separated by , (comma).

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

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

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

6.3.3.1 Arguments

An argument (represented by `arg` 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 not greater than 999. When an argument is an arbitrary term, its priority shall be less than the priority

Table 4 — 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

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.

```
        arg =  atom ;
Abstract: a    a
Condition: a is an operator
```

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

NOTE — This concept of an "argument" ensures that both the terms `f(x,y)` and `f(:-, ;, [:-, :-|:-])` are syntactically valid whatever operator definitions are currently defined. Comma is not an atom, and the following 'terms' have syntax errors: `f(,,a)`, `[a,,|v]`, and `[a,b|,]`; but the following terms are syntactically valid: `f(',',a)`, `[a,','|v]`, and `[a,b|',']`.

6.3.4 Compound terms – operator notation

Operator notation can be used for inputting or outputting a compound term whose functor symbol is an operator defined in the operator table (see 6.3.4.4, table 7).

An operator is an atom defined by its specifier and priority.

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

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

A lower priority means stronger operator binding.

The specifier of an operator (defined by table 4) is a mnemonic that defines the class (prefix, infix or postfix) and the associativity (right-, left- or non-associative) of the operator.

An operand (3.120) with the same (or smaller) priority as a right-associative operator which follows that operator need not be bracketed.

Table 5 — Valid and invalid terms

Invalid term	Valid term
`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)`

Table 6 — Equivalent terms

Unbracketed term	Equivalent bracketed term
`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`

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

1 The examples of terms in tables 5 and 6 assume that each atom `fx`, `fy`, `xfx`, `xfy`, `yfx`, `xf` and `yf` is an operator with the corresponding specifier and same priority.

2 Table 5 shows some invalid terms and how they need to be bracketed to be valid.

3 Table 6 shows equivalent bracketed and unbracketed terms. 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 (3.120) 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 condition "the first token of a is not `open ct`" 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 assigns an unambiguous reading to terms such as `fy t1 yf` where the operators have the same priority.

6.3.4.3 Operators

An operator is an atom (6.3.1.3).

A comma (6.4.8) shall be equivalent to the atom `','` when `','` is an operator.

	`op =`	`atom ;`
Abstract:	a	a
Priority:	n	n
Specifier:	s	s
Condition:	a is an operator	

	`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.

Table 7 — The operator table

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

NOTES

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

2 A comma token is treated as synonymous with the operator ',' as it is defined in the initial operator table.

3 The third argument of op/3 (8.14.3) may be any atom except ',' so the priority of the comma operator cannot be changed.

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 The operator table

The operator table defines which atoms shall be regarded as operators when (1) a sequence of tokens is parsed as a read-term by the built-in predicate read_term/3 (8.14.1), or (2) Prolog text is prepared for execution (7.4), or (3) output by the built-in predicate (8.14.2).

Table 7 defines the predefined operators, that is, those operators defined in the initial state of the operator table.

NOTES

1 The predicate indicators whose predicate names are operators are: (a) (=)/2 (Prolog unify), (\=)/2 (not Prolog unifiable), (b) Term comparison, (c) (=..)/2 (univ), (d) Arithmetic evaluation, (e) Arithmetic comparison, (f) (\+)/1 (not provable).

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

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

4 The operator table may be altered during execution, see op/3 (8.14.3).

6.3.5 Compound terms – list notation

List notation can be used for inputting or outputting a compound term with principal functor '.'/2 (dot).

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

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

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

          items =  arg ;
Abstract: .(t,[])  t
```

NOTE — For the syntax of an empty list, see 6.3.1.3.

6.3.5.1 Examples

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

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 bracketed term

A term with principal functor `'{}'/1` can also be expressed by enclosing its argument in curly brackets.

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

NOTE — For the syntax of an empty curly brackets, see 6.3.1.3.

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 Terms – double quoted list notation

A double quoted list is either an atom (6.3.1.3) or a list (6.3.5).

If the Prolog flag `double_quotes` has a value `chars`, a `double quoted list` token *dql* containing L `double quoted characters` is a list l with L elements, where the N-th element of the list is the one-char atom whose name is the N-th `double quoted character` of *dql*.

If the Prolog flag `double_quotes` has a value `codes`, a `double quoted list` token *dql* containing L `double quoted characters` is a list l with L elements, where the N-th element of the list is the collating sequence integer of the N-th `double quoted character` of *dql*.

If the Prolog flag `double_quotes` has a value `atom`, a `double quoted list` token *dql* containing L `double quoted characters` is an atom l whose name is the concatenation of L characters, where the N-th character is the character denoted by the N-th `double quoted character` of *dql*.

	`term = double quoted list ;`	
Abstract:	l	*ccl*
Priority:	0	

6.3.7.1 Examples

The following examples show terms expressed in double quoted list notation. They assume that the goal has been input as a term using the built-in predicate `read_term/3` (8.14.1) and is then executed without changing the value of the flag `double_quotes`.

```
( current_prolog_flag(double_quotes, chars),
  atom_chars('jim', "jim")
; current_prolog_flag(double_quotes, codes),
  atom_codes('jim', "jim")
; current_prolog_flag(double_quotes, atom),
  'jim' == "jim"
).
    Succeeds.

( current_prolog_flag(double_quotes, chars),
  [] == ""
; current_prolog_flag(double_quotes, codes),
  [] == ""
; current_prolog_flag(double_quotes, atom),
  '' == ""
).
    Succeeds.
```

6.4 Tokens

Lexically, the syntax of Prolog terms (6.3) shall be a sequence of tokens. This subclause defines how characters are combined to form tokens, and the tokens to form terms and read-terms.

```
term (* 6.4 *)
   = { token (* 6.4 *) } ;

read term (* 6.4 *)
   = term (* 6.4 *) , end (* 6.4 *) ;

token (* 6.4 *)
   = name (* 6.4 *)
   | variable (* 6.4 *)
   | integer (* 6.4 *)
   | float number (* 6.4 *)
   | double quoted list (* 6.4 *)
   | open (* 6.4 *)
   | open ct (* 6.4 *)
   | close (* 6.4 *)
   | open list (* 6.4 *)
   | close list (* 6.4 *)
   | open curly (* 6.4 *)
   | close curly (* 6.4 *)
   | ht sep (* 6.4 *)
   | comma (* 6.4 *)
   ;

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 *) ;
double quoted list (* 6.4 *)
   = [ layout text sequence (* 6.4.1 *) ] ,
     double quoted 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 (a) part of a meta escape sequence, or (b) two adjacent quote characters, for example `'ab''cd''e'`, or `"f""""g"`, or `''''`.

3 Not every sequence of tokens forms a valid term. Additional requirements are made in subclause 6.3.

6.4.1 Layout text

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

a) Is . (dot) a graphic token or an end token?

b) Is an atom followed by an open token the functor of a compound term (6.3.3) or a 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 *)
   = letter digit token (* 6.4.2 *)
   | graphic token (* 6.4.2 *)
   | quoted token (* 6.4.2 *)
   | semicolon token (* 6.4.2 *)
   | cut token (* 6.4.2 *) ;

letter digit token (* 6.4.2 *)
   = small letter char (* 6.5.2 *),
     { alphanumeric 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 . is followed by a `layout char` or `single line comment`.

```
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 character (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 does not contain a single quoted character is the null atom.

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

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

```
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 character (* 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 *) ;

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

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

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.

6.4.2.1 Quoted characters

```
single quoted character (* 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 character (* 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 character (* 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 *)
   | alphanumeric 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 character is a single quoted character or a double quoted character or a back quoted character.

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

A quoted character which consists of a graphic char, or an alphanumeric 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 backspace char (* 6.4.2.1 *)
   | symbolic carriage return char (* 6.4.2.1
   | symbolic form feed char (* 6.4.2.1 *)
   | symbolic horizontal tab char (* 6.4.2.1 *)
   | symbolic new line char (* 6.4.2.1 *)
   | symbolic vertical tab char (* 6.4.2.1 *) ;

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

```
symbolic vertical tab char (* 6.4.2.1 *)
   = "v" ;
```

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 *) ;

symbolic hexadecimal char (* 6.4.2.1 *)
   = "x" ;
```

NOTES

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

2 \ 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.

3 The representations of the symbolic control characters are those recommended by the International Standard for C (ISO/IEC 9899).

4 A back quoted string (6.4.7) contains back quoted characters, but this part of ISO/IEC 13211 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 *),
     alphanumeric char (* 6.5.2 *),
     { alphanumeric char (* 6.5.2 *) }
   | capital letter char (* 6.5.2 *),
     { alphanumeric 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 character (* 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" ;
```

NOTE — A `float number token` is unsigned. Negative floating point values are defined by the term syntax (6.3.1.2).

6.4.6 Double quoted lists

A double quoted list token denotes a term which depends on the value of the Prolog flag `double_quotes` (7.11.2.5) at the time the read-term or Prolog text is input.

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

A double quoted list token *DQ* containing one or more continuation escape sequences shall be equivalent to the double quoted list token which would be obtained by removing the continuation escape sequences from *DQ*.

```
double quoted 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 character (* 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 the 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 character (* 6.4.2.1 *)
   | continuation escape sequence (* 6.4.2 *) ;
```

NOTE — This part of ISO/IEC 13211 does not define a token (or term) based on a back quoted string.

It would be a valid extension of this part of ISO/IEC 13211 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 be followed by 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 equivalent to the 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), so a `layout char` is necessary to separate an `end char` from a `bracketed comment`.

6.5 Processor character set

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

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

```
char (* 6.5 *)
   = graphic char (* 6.5.1 *)
```

```
   | alphanumeric 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 input 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 ISO 8859-1, 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

```
alphanumeric 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 alphanumeric character denotes itself in a quoted character.

NOTE — The alphanumeric 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 *)
   | horizontal tab char (* 6.5.4 *)
   | new line char (* 6.5.4 *) ;
```

```
space char (* 6.5.4 *) = " " ;
horizontal tab char (* 6.5.4 *)
    = implementation dependent ;
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.

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 double quoted 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 double quoted 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.

NOTE — These requirements on the collating sequence are satisfied by both ASCII and EBCDIC.

7 Language concepts and semantics

This clause defines the semantic concepts of Prolog:

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

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

c) Subclause 7.3 defines unification in Prolog,

d) Subclause 7.4 defines the meaning of Prolog text,

e) Subclause 7.5 defines the database,

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

g) Subclause 7.7 defines the execution of a goal,

h) Subclause 7.8 defines the control constructs of Prolog,

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

j) Subclause 7.10 defines input/output concepts,

k) Subclause 7.11 defines flags,

l) Subclause 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 an atomic term.

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

NOTE — Prolog is not a typed language, and an argument of a compound term or predication can be any term whatsoever. Nonetheless, some predications can be satisfied only when the arguments possess particular properties, and some evaluable functors are defined only when the operands (3.121) possess some particular property. Note also that although the control constructs, built-in predicates and evaluable functors are defined for all arguments and operands (3.120), it is often an error unless an argument has a particular sort of value.

It is therefore convenient when defining Prolog to classify a term as belonging to one of several disjoint types.

7.1.1 Variable

A variable is a member of a set V (see 6.1.2 a). 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 an atomic term 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 an atomic term 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 a member of a set I (see 6.1.2 c) 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 When *bounded* is **false**, expressions with an integer value will not have a value **int_overflow**, but might produce a resource error (7.12, 7.12.2 h) because of exhaustion of resources.

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

3 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 9.1.3.

4 The abstract syntax of an integer number is defined in 6.3.1.1 and 6.3.1.2. The token syntax of a (positive) integer token is defined in 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:

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

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, inputting 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 members of C (characters) (7.1.4.1) and members of CC (character codes).

7.1.3 Floating point

A floating point value is a member of a set F (see 6.1.2 d) 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:

$r \geq 2$
$\wedge\ p \geq 2$
$\wedge\ p - 2 \leq -emin \leq r^p - 1$
$\wedge\ p \leq emax \leq r^p - 1$

These parameters should also satisfy:

r is even
$\wedge\ r^{p-1} \geq 10^6$
$\wedge\ (emin - 1) \leq -2 * (p - 1)$
$\wedge\ emax > 2 * (p - 1)$
$\wedge\ -2 \leq (emin - 1) + emax \leq 2$

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

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

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

$$F = F_N \cup F_D \quad \text{if } denorm = \textbf{true}$$
$$= F_N \quad \text{if } denorm = \textbf{false}$$

The members of F_N are called *normalized* floating point values because of the constraint $r^{p-1} \leq i \leq r^p - 1$. The members 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 part of ISO/IEC 13211 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 described in the rationale (annex A) of ISO/IEC 10967-1 – Language Independent Arithmetic (LIA). The constraints on the parameters are also justified and explained there.

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

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

4 The abstract syntax of a floating point number is defined in 6.3.1.1 and 6.3.1.2. The token syntax of a (positive) float number token is defined in 6.4.5.

7.1.3.1 Additional floating point constants and sets

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

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

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

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

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

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

$$\begin{aligned} F^* = F \\ \cup \{\ \pm i * r^{e-p} \\ \mid i, e \in \mathcal{Z},\ r^{p-1} \le i \le r^p - 1,\ e > emax\} \end{aligned}$$

NOTES

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

7.1.4 Atom

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

7.1.4.1 Characters and one-char atoms

C, a set of characters, is an implementation defined subset of PCS, the processor character set (6.5),

Any member of C is represented in a Prolog term by a one-char atom whose name is that member of C.

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

7.1.4.2 Boolean

Bool is a subset of A.

Bool = { `true` , `false` }

When an argument of an option (see for example, 7.10) is `Bool`, a member of *Bool* shall be provided, and omitting to specify such an option shall be equivalent to providing that option with argument `false`.

7.1.5 Compound term

A compound term is a member of a set CT (see 6.1.2 e) 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.

Arguments are numbered from 1.

NOTE — The syntax of a compound term is defined in 6.1.2 e, 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 variant 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` occur in any structure created during the execution of a goal (7.7).

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` is an empty list, or

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

c) The heads 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* `L2_element` during the creation of the sorted list (see 7.2 especially 7.2.1).

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).

7.1.6.6 Predicate indicator

`PI` is a predicate indicator if it is a compound term `'/'(A, N)` where `A` is an atom and `N` is a non-negative integer.

The predicate indicator `'/'(A, N)` indicates the procedure whose identifier is `A` and whose arity is `N`.

NOTE — In Prolog text and this part of ISO/IEC 13211 a predicate indicator `'/'(A, N)` is normally written as `A/N` or `(A)/N` depending on whether or not `A` is an operator.

7.1.6.7 Predicate indicator sequence

`PI_sequence` is a predicate indicator sequence if it is a compound term `','(PI_1, PI_n)` where `PI_1` is a predicate indicator, and `PI_n` is a predicate indicator or a predicate indicator sequence.

The predicate indicator sequence `','(A/N, PI_n)` indicates the procedure whose identifier is `A` and whose arity is `N`, together with all the procedures indicated by `PI_n`.

NOTE — A predicate indicator sequence `','(P1/A1, ','(P2/A2, P3/A3))` is normally written as `P1/A1, P2/A2, P3/A3`.

7.1.6.8 Predicate indicator list

`PI_list` is a predicate indicator list if it is a compound term `'.'(PI_1, PI_n)` where `PI_1` is a predicate indicator, and `PI_n` is an empty list or a predicate indicator list.

The predicate indicator list `'.'(A/N, PI_n)` indicates the procedure whose identifier is `A` and whose arity is `N`, and, if `PI_n` is not the empty list, all the procedures indicated by `PI_n`.

NOTE — A predicate indicator list `'.'(P1/A1, '.'(P2/A2, []))` is normally written as `[P1/A1, P2/A2]`.

7.2 Term order

An ordering *term_precedes* (3.181) defines whether or not a term `X` term-precedes a term `Y`.

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 during the creation of a sorted list (7.1.6.5, 8.10.3.1 j) the ordering shall remain constant.

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

7.2.2 Floating point

If X and Y are floating point values 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 of the name of X (6.1.2 b) is less than the value in the collating sequence of the first character of the name of Y, or

c) the value in the collating sequence of the first character of the name of X is equal to the value in the collating sequence of the first character of the name of Y, and XT *term_precedes* YT where XT is the atom whose name is obtained by deleting the first character of the name of X, and YT is the atom whose name is obtained by deleting the first character of the name 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 Ith argument of X and Yi is the Ith 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 feature 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 to be *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 yields 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 sole purpose to define the concepts (*NSTO*, *STO*) presented in 7.3.3. 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 atomic terms, or

2) $f = g$ where f is an atomic term and g is a compound term, or f is a compound term and g is an atomic term, 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 an atomic term, 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 other step is applicable, 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 8 show the operation of the algorithm. The final two examples show that the result of the algorithm is not necessarily unique.

Table 8 — Unification examples

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

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

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

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

A Prolog text (including goals) is *NSTO* if and only if all unifications during its execution are *NSTO*. It is *STO* otherwise.

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 built-in predicate `(=)/2` (8.2.1) and also when unification is invoked implicitly in this part of ISO/IEC 13211.

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:

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

b) all unifications which are *STO* are made using the built-in 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 part of ISO/IEC 13211 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 A standard-conforming processor might consistently succeed, loop, or fail for a unification that is formally undefined by this part of ISO/IEC 13211.

3 Most implementations do not include the occurs-check test for efficiency reasons, and are compatible with this definition of unification. In the undefined cases, their 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*.

4 *STO* and *NSTO* are decidable properties for a single unification. However processors are not required to include such a test.

5 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.

6 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 part of ISO/IEC 13211. 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 built-in 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 seeing which clause heads are unifiable with a goal. But in the latter case, care is needed, for example consider the user-defined procedure `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 directives and user-defined procedures in a textual form.

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 part of ISO/IEC 13211 leaves undefined:

a) The mechanisms for converting clause-terms and directive-terms of Prolog text into procedures of the database,

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

c) The action to be taken if the read-terms forming Prolog text do not conform to the requirements of this part of ISO/IEC 13211.

NOTE — This part of ISO/IEC 13211 does not define a built-in predicate `consult/1`, nor any similar built-in predicate.

7.4.2 Directives

The characters of a directive-term in Prolog text (6.2.1.1) shall satisfy the same constraints as those required to input a read-term during a successful execution of the built-in predicate `read_term/3` (8.14.1). The principal functor shall be `(:-)/1`, and its argument shall be a directive.

A directive in Prolog text (6.2.1.1) specifies:

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

b) the format and syntax 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.

A processor shall support correctly any directive whose directive indicator is specified in subclause 7.4.2.x.

NOTE — The usage and semantics of directives may be altered in Part 2 (Modules) of ISO/IEC 13211.

7.4.2.1 `dynamic/1`

A directive `dynamic(PI)` where `PI` is a predicate indicator, a predicate indicator sequence, or a predicate indicator list specifies that each user-defined procedure indicated by `PI` is dynamic.

No procedure indicated by `PI` shall be a control construct or built-in predicate.

The first directive `dynamic(PI)` that specifies a user-defined procedure `P` to be dynamic shall precede all clauses for `P`. Further, if `P` is defined to be a dynamic procedure in one Prolog text, then a directive `dynamic(PI)` indicating `P` shall occur in every Prolog text which contains clauses for `P`.

NOTE — More than one directive `dynamic(PI)` may specify a user-defined procedure `P` to be dynamic in a Prolog text.

7.4.2.2 `multifile/1`

A directive `multifile(PI)` where `PI` is a predicate indicator, a predicate indicator sequence, or a predicate indicator list specifies that the clauses for each user-defined procedure indicated by `PI` may be read-terms of more than one Prolog text.

No procedure indicated by `PI` shall be a control construct or built-in predicate.

Each Prolog text that contains clauses for the user-defined procedure `P` shall contain a directive `multifile(PI)` indicating the procedure `P`. The first directive `multifile(PI)` indicating procedure `P` shall precede all clauses for the procedure `P`.

NOTE — More than one directive `multifile(PI)` may specify a user-defined procedure `P` to be multifile.

7.4.2.3 `discontiguous/1`

A directive `discontiguous(PI)` where `PI` is a predicate indicator, a predicate indicator sequence, or a predicate indicator list specifies that each user-defined procedure indicated by `PI` may be defined by clauses which are not consecutive read-terms of the Prolog text.

No procedure indicated by `discontiguous(PI)` shall be a control construct or built-in predicate.

If Prolog text contains a directive `discontiguous(PI)`, then that directive may occur any number of times in that Prolog text. The first directive `discontiguous(PI)` indicating procedure `P` shall precede all clauses for the procedure `P`.

NOTE — More than one directive `discontiguous(PI)` may specify the clauses of the user-defined procedure `P` to be discontiguous.

7.4.2.4 `op/3`

A directive `op(Priority, Op_specifier, Operator)` enables the operator table (see 6.3.4.4 and table 7) to be altered.

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` (8.14.3), and the operator table shall be altered in the same way.

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

7.4.2.5 `char_conversion/2`

A directive `char_conversion(In_char, Out_char)` enables $Conv_C$, the character-conversion mapping (3.46), to be altered.

The arguments `In_char` and `Out_char` shall satisfy the same constraints as those required for a successful execution of the built-in predicate `char_conversion/2` (8.14.5), and $Conv_C$ shall be altered in the same way.

It shall be implementation defined whether or not the character-conversion mapping defined in a directive `char_conversion(In_char, Out_char)` shall affect $Conv_C$ in other Prolog texts or during execution.

7.4.2.6 `initialization/1`

A directive `initialization(T)` converts the term `T` to a goal `G` and includes it 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 `include/1`

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

7.4.2.8 `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 multiple directives `ensure_loaded(P_text)` exist for the same Prolog text, that Prolog text is included in the Prolog text prepared for execution only once. The position where it is included is implementation defined.

7.4.2.9 `set_prolog_flag/2`

A directive `set_prolog_flag(Flag, Value)` enables the value associated with a Prolog flag to be altered.

The arguments `Flag` and `Value` shall satisfy the same constraints as those required for a successful execution of the built-in predicate `set_prolog_flag/2` (8.17.1), and `Value` shall be associated with flag `Flag` in the same way.

It shall be implementation defined whether or not a directive `set_prolog_flag(Flag, Value)` shall affect the values associated with flags in other Prolog texts or during execution.

7.4.3 Clauses

A clause-term in Prolog text (6.2.1.2) enables a clause of a user-defined procedure to be added to the database.

The characters of a clause-term shall satisfy the same constraints as those required to read a read-term during a successful execution of the built-in predicate `read_term/3` (8.14.1).

A clause `Clause` of a clause-term `Clause.` shall satisfy the same constraints as those required for a successful execution of the built-in predicate `assertz(Clause)` (8.9.2), except that no error shall occur because `Clause` refers to a static procedure, and `Clause` shall be converted to a clause c and added to the database in the same way.

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

If no clauses are defined for a procedure indicated by a directive with directive indicator `dynamic/1` (7.4.2.1), `multifile/1` (7.4.2.2), or `discontiguous/1` (7.4.2.3), then the procedure shall exist but have no clauses.

All the clauses for a user-defined procedure `P` shall be read-terms of a single Prolog text unless there is a directive `multifile(UP)` where `UP` indicates `P` in each Prolog text in which there are clauses for `P`.

All the clauses for a user-defined procedure `P` shall be consecutive read-terms of a single Prolog text unless there is a directive `discontiguous(UP)` directive indicating `P` in that 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 is identified by a unique predicate indicator (3.131).

Built-in predicates and control constructs are provided by the processor. They have properties which are defined by the clauses in this part of ISO/IEC 13211. In particular, they cannot be altered or deleted during execution (see 7.5.2).

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

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

NOTES

1 There is a difference between a procedure which does not exist, and one which exists but has no clauses, for example see 7.7.7, 8.9.4.

2 A procedure may have no clauses if (1) it is specified in a directive but no clauses are defined for it, or (2) it is dynamic and all clauses have been retracted.

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 with directive indicator `dynamic/1` in Prolog text overrides the default, and (2) asserting a clause of a non-existent procedure shall create a dynamic procedure.

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

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.

7.5.3 Private and public procedures

Each procedure is either public or private. A dynamic procedure shall be public. Each built-in predicate and

control construct shall be private, and a static user-defined procedure shall be private by default.

A clause of a public procedure can be inspected, a clause of a private procedure cannot be inspected.

NOTE — An additional directive `public/1` that specifies some user-defined procedures to be public would be an extension.

7.5.4 A logical database update

Any change in the database that occurs as the result of executing a goal (for example, when the activator of a subgoal is a call of `assertz/1` or `retract/1`) shall affect only an activation whose execution begins afterwards. The change shall not affect any activation that is currently being executed.

NOTE — Thus the database is frozen during the execution of a goal, and the list of clauses defining a predication is fixed at the moment of its execution (see 7.7.7 e).

7.6 Converting a term to a clause, and a clause to a term

Prolog provides the ability to convert Prolog 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 built-in 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 ISO/IEC 13211 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.

Table 9 — 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

7.6.2 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 9 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 9 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.

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 9 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 subclause 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 subclause 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 (7.7.9) may occur 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 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 defined by the current number of components of S.

decsglstk — A stack of decorated subgoals which defines a sequence of activators that 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 DS is a structured data type with components:

activator — A predication prepared for execution which must be executed successfully in order to satisfy 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*.

Table 10 — The execution stack after initialization

S_index	Decorated subgoal stack	Substitution	BI
1	((goal, 0), $newstack_{DS}$)	{}	*nil*
	$newstack_{ES}$		

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, $newstack_{ES}$)

ES_i = (S_i, $currentgoal_i$, $subst_i$, BI_i)

$currstate$ = $top(S)$ = ES_N

$currentgoal$ = ($decoratedsubgoal_J$, ... $decoratedsubgoal_1$, $newstack_{DS}$)

$currdecsgl$ = $decoratedsubgoal_J$

$decoratedsubgoal_j$ = ($activator_j$, $cutparent_j$)

curract is the activator of *currdecsgl*.

2 The concept of a stack is defined in 4.2.

7.7.3 Initialization

The method by which a user delivers a goal to the Prolog processor shall be implementation defined.

A goal is prepared for execution by converting it into an activator.

Table 10 shows the execution stack after it has been initialized and is ready to execute `goal` after it has been converted into an activator.

Execution can then begin (7.7.7).

Table 11 — The goal succeeds

S_index	Decorated subgoal stack	Substitution	BI
N	($newstack_{DS}$)	Σ	*nil*
...			
1	((goal, 0), $newstack_{DS}$)	{}	...
	$newstack_{ES}$		

Table 12 — The goal fails

S_index	Decorated subgoal stack	Substitution	BI
	$newstack_{ES}$		

NOTE — A processor may support the concept of a query, that is a goal given as interactive input to the top level. But this part of ISO/IEC 13211 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 part of ISO/IEC 13211 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 11. 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 12.

7.7.6 Re-executing a goal

After satisfying an initial goal, execution may continue by trying to satisfy it again.

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 indicator corresponds 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 `unknown` (7.11.2.4):

`error` — There shall be an error
`existence_error(procedure, PF)`
where `PF` is the predicate indicator 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.

7.7.8 Backtracking

The processor backtracks (1) if a goal has failed, or (2) 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, 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 7.7.12 b.

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 are all re-executed in this semantic model of Prolog so that S shows more clearly the history of the execution. However they all fail immediately when they are re-executed.

3 The control construct if-then-else is re-executed (after the *if* fails) so that the *else* can be executed.

4 Step 7.7.8 e happens after the *either* branch of a disjunction `';'(either, or)` 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 by executing the built-in predicates `abolish/1`, `asserta/1`, `assertz/1`, `retract/1`.

b) Changes to the operator table (see 6.3.4.4) by executing the built-in predicate `op/3`,

c) Changes to the values associated with Prolog flags (7.11) by executing the built-in predicate `set_prolog_flag/2`,

d) Changes to $Conv_C$, the character-conversion mapping by executing the built-in predicate `char_conversion/2` (8.14.5),

e) Input/output, for example, stream selection and control, character, byte, and term input/output (8.11, 8.12, 8.13, 8.14).

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 and *curract* are unifiable, then it is selected for execution and continue execution at 7.7.10 e,

d) Else BI is replaced by a value $up(CT)$ and continue execution at 7.7.10 a.

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 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 *currstate* 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 BI has the value $up([\,])$ when (a) all the clauses of p have been examined to see if their head and *curract* are unifiable, or (b) p has no clauses at all.

2 *choicepoint* will be re-executed if backtracking becomes necessary (7.7.8).

3 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*.

4 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).

Table 13 — Before executing a rule `p(X, Y)`

S_index	Decorated subgoal stack	Substitution	BI
N	((`p(X, Y)`, CP), ...)	Σ	$up([P_1\|P_T])$
...			

7.7.10.1 Example – A user-defined rule

If the first clause P_1 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 13 shows the execution stack ready to execute a *curract* `p(X, Y)` using this clause.

The actions to execute this subgoal:

a) c' = `( p(MM, WW), ','(m(MM), f(WW)) )`

b) *MGU* = {X → MM, Y → WW}

c) Applying *MGU* to the c' body, `','(m(MM), f(WW))`

d) Make a copy CCS of *currstate*,
CCS = ((`p(X, Y)`, CP), ...), Σ, $up([P_1|P_T])$

e) Apply *MGU* to CCS

f) Replace the activator by the body of c'

g) Set BI to *nil*

h) Set *cutparent* of subgoal to the current *choicepoint* so that now
CCS = ((`','(m(MM), f(WW))`, $N-1$), ...),
{ X → MM, Y → WW } ∘ Σ, *nil*

i) Push CCS on to S.

Table 14 shows the execution stack after executing the subgoal `p(X, Y)` using the clause `( p(M, W), ','(m(M), f(W)) )`.

7.7.10.2 Example – A user-defined fact

If the first clause M_1 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 15 shows the execution stack ready to execute a *curract* `m(X)` using this clause.

The actions to execute this subgoal:

Table 14 — After executing a rule `p(X, Y)`

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((',' (m(MM), f(WW)), $N-1$), ...)	{X → MM, Y → WW} ∘ Σ	*nil*
N	((p(X, Y), CP), ...)	Σ	$up([P_1\|P_T])$
...			

Table 15 — Before executing a fact `m(pete)`

S_index	Decorated subgoal stack	Substitution	BI
N	((m(X), CP), (f(W), CP), ...)	Σ	$up([M_1\|M_T])$
...			

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, *nil*

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*

i) Push CCS on to S.

Table 16 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:

Table 16 — 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	$up([M_1\|M_T])$
...			

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 callable term representing 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 predicate is designated as re-executable. Then its 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.

The format and notation of the definition of each control construct is consistent with that used for built-in predicates (8.1) except that a mode `goal` indicates that the argument is a goal rather than a term.

NOTES

1 A control construct is static.

2 The control constructs are defined formally in subclause A.5.1.

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*, i.e. (`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 have been replaced by *true* and deleted so that the decorated subgoal stack becomes empty (see 7.7.4).

7.8.1.2 Template and modes

`true`

Table 17 — Before executing `true`

S_index	Decorated subgoal stack	Substitution	BI
N	((`true`, $N-2$), (`f(W)`, CP), ...)	Σ	*ctrl*
...			

Table 18 — After executing `true`

S_index	Decorated subgoal stack	Substitution	BI
N	((`f(W)`, CP), ...)	Σ	*nil*
...			

7.8.1.3 Errors

None.

7.8.1.4 Examples

Tables 17 and 18 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 19 — 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 20 — After executing fail

S_index	Decorated subgoal stack	Substitution	BI
N	((f(Y), CP), ...)	Σ	$up([F_1\|F_T])$
...			

7.8.2.2 Template and modes

```
fail
```

7.8.2.3 Errors

None.

7.8.2.4 Examples

Tables 19 and 20 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 (7.12.2 a),

e) Else if the term `G` is a number, there shall be a type error (7.12.2 b),

f) Else convert the term `G` to a goal $goal$ (7.6.2).

g) Let NN be the S_index of the $choicepoint$ of $currstate$.

h) Push ($goal$, NN) on to $currentgoal$ of CCS.

i) Push CCS on to S.

j) Continue execution at 7.7.7.

k) Pop $currstate$ from S.

l) Continue execution at 7.7.8.

`call(G)` is re-executable. On backtracking, continue at 7.8.3.1 k.

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 Template and modes

```
call(+callable_term)
```

7.8.3.3 Errors

a) `G` is a variable
— `instantiation_error`.

b) `G` is neither a variable nor a callable term
— `type_error(callable, G)`.

c) `G` cannot be converted to a goal
— `type_error(callable, G)`.

Table 21 — Before executing `call(G)`

S_index	Decorated subgoal stack	Substitution	BI
N	((`call(G)`, *CP*), ...)	Σ	*ctrl*
...			

Table 22 — After executing `call(G)`

S_index	Decorated subgoal stack	Substitution	BI
N + 1	((`G`, *N* − 1), ...)	Σ	*nil*
N	((`call(G)`, *CP*), ...)	Σ	*ctrl*
...			

7.8.3.4 Examples

Tables 21 and 22 show the execution stack before and after executing the control construct `call(G)`.

The examples defined in this subclause 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 characters representing a variable,
   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.
   [This behaviour arises because the
   argument of call/1 is converted to a goal
   before it is executed, and 'Z' becomes the
   goal 'call(Z)', and is executed as 'call(!)'
   which is equivalent to true.]

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) Set *BI* of *CCS* to *nil*.

c) Replace the *curract*, `!`, of *CCS* by `true`.

d) Push *CCS* on to *S*.

e) Continue execution at 7.7.7.

f) Make a copy *cut* of *cutparent* of *currstate*.

g) Pop *currstate* from *S*.

h) If *cut* = *S_index* of *top*(*S*) then *top*(*S*) becomes the new *currstate*, and continue execution by backtracking at 7.7.8.

i) Else continue execution at 7.8.4.1 g.

`!` is re-executable. On backtracking, continue at 7.8.4.1 f.

Table 23 — Before executing cut

S_index	Decorated subgoal stack	Substitution	BI
N	((!, CP), ...)	Σ	$ctrl$
...			

Table 24 — After executing cut

S_index	Decorated subgoal stack	Substitution	BI
$N+1$	((true, CP), ...)	Σ	nil
N	((!, CP), ...)	Σ	$ctrl$
...			

NOTES

1 Executing a cut has the effect that:

a) A cut always succeeds, but

b) No attempts are made to re-execute 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 25 — 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 26 — After re-executing cut

S_index	Decorated subgoal stack	Substitution	BI
CP	((p(X, Y), CP), ...)	σ	$up([P_1\|P_T])$
...			

7.8.4.2 Template and modes

!

7.8.4.3 Errors

None.

7.8.4.4 Examples

Tables 23 and 24 show the execution stack before and after executing the control construct !.

Tables 25 and 26 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.

twice(_), (!, fail; write('No ')).
   Outputs "C ",
   Fails.

twice(X), call(X), write('Forwards '), fail.
   Outputs "C Forwards Moss Forwards ",
   Fails.

goal(X), call(X), write('Forwards '), fail.
   Outputs "C Forwards Three Forwards ",
   Fails.

twice(_), \+(\+(!)),
     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 conjunction of two activators *first* and *second* 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 7.8.5.1 f.

NOTES

1 Step 7.8.5.1 d 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.

2 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 Template and modes

```
','(goal, goal)
```

NOTE — `','` is a predefined infix operator.

7.8.5.3 Errors

None.

7.8.5.4 Examples

Tables 27 and 28 show the execution stack before and after executing the control construct `','(First, second)`.

```
','(X=1, var(X)).
   Fails.

','(var(X), X=1).
   Succeeds, unifying X with 1.

','(X = true, call(X)).
   Succeeds, unifying X with true.
```

7.8.6 (;)/2 – disjunction

A disjunction control construct whose first activator is an if-then control construct (7.8.7) shall be an if-then-else control construct, see 7.8.8.

Table 27 — Before executing a conjunction

S_index	Decorated subgoal stack	Substitution	BI
N	((',' (First, Second), *CP*), ...)	Σ	*ctrl*
...			

Table 28 — 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*
...			

7.8.6.1 Description

';'(Either, Or) is true iff Either is true or Or is true.

Procedurally, a control construct disjunction of two activators *either* and *or*, denoted by ';'(Either, Or), shall be executed as follows:

a) Make two copies *CCS*1 and *CCS*2 of *currstate*.

b) Set *BI* of *CCS*1 and *CCS*2 to *nil*.

c) Replace the current activator *curract* of *CCG*2 by *or*.

d) Push *CCG*2 on to *S*.

e) Replace the current activator *curract* of *CCG*1 by *either*.

f) Push *CCG*1 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 7.8.6.1 h.

Table 29 — Before executing a disjunction

S_index	Decorated subgoal stack	Substitution	BI
N	((';' (Either, Or), *CP*), ...)	Σ	*ctrl*
...			

NOTES

1 Step 7.8.6.1 f makes *CCG*1 the new *currstate* and the execution state *CCG*2 is now a *choicepoint* of execution state *CCG*1.

2 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 Template and modes

';'(goal, goal)

NOTE — ';' is a predefined infix operator.

7.8.6.3 Errors

None.

7.8.6.4 Examples

Tables 29 and 30 show the execution stack before and after executing the control construct ';'(Either, Or).

```
';'(true, fail).
   Succeeds.

';'((!, fail), true).
   Fails.
   [Equivalent to (!, fail).]

';'(!, call(3)).
   Succeeds.
   [Equivalent to !.]

';'((X = 1, !), X = 2).
   Succeeds, unifying X with 1.

','(';'(X=1, X=2), ';'(true, !)).
   Succeeds, unifying X with 1.
   On re-execution, succeeds, unifying X with 1.
   On re-execution, fails.
```

Table 30 — 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*
...			

7.8.7 (->)/2 – if-then

An if-then control construct which is the first activator of a disjunction control construct (7.8.6) shall be part of an if-then-else control construct, see 7.8.8.

7.8.7.1 Description

`'->'(If, Then)` is true iff (1) `If` is true, and (2) `Then` is true for the first solution of `If`.

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) Let NN be the *S_index* of the *choicepoint* of *currstate*.

e) Push (*then*, CP) on to *currentgoal* of CCS.

f) Push (`!`, NN) on to *currentgoal* of CCS.

g) Push (*if*, NN) 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.

`'->'(If, Then)` is re-executable. On backtracking, continue at 7.8.7.1 j.

Table 31 — Before executing an if-then

S_index	Decorated subgoal stack	Substitution	BI
N	(('->'(If, Then), CP), ...)	Σ	*ctrl*
...			

NOTES

1 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 Template and modes

```
'->'(goal, goal)
```

NOTE — `'->'` is a predefined infix operator.

7.8.7.3 Errors

None.

7.8.7.4 Examples

Tables 31 and 32 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.
```

Table 32 — 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*
...			

```
'->'(';'(X=1, X=2), true).
   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 (1a) `If` is true, and (1b) `Then` is true for the first solution of `If`, 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) Let N be the *S_index* of *currstate*.

e) Let NN be the *S_index* of the *choicepoint* of *currstate*.

f) Push (*then*, CP) on to *currentgoal* of CCS.

g) Push (!, NN) on to *currentgoal* of CCS.

h) Push (*if*, N) on to *currentgoal* of CCS.

i) Push CCS on to S.

j) Continue execution at 7.7.7.

k) Make a copy CCS of *currstate*.

l) Set BI of CCS to *nil*.

m) Pop *currdecsgl* (`';'('->'(If, Then), Else)`, CP) from *currentgoal* of CCS.

n) Push (*else*, CP) on to *currentgoal* of CCS.

o) Push (!, NN) on to *currentgoal* of CCS.

p) Push CCS on to S.

q) Continue execution at 7.7.7.

`';'('->'(If, Then), Else)` is re-executable. On backtracking, continue at 7.8.8.1 k.

The cut (7.8.8.1 o) prevents an if-then-else from being re-executed a second time.

NOTES

1 Executing an if-then-else has the effect that:

a) If *if* should fail, then the if-then-else will be re-executed, 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.

2 Re-executing an if-then-else has the effect that:

a) 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 33 — Before executing an if-then-else

S_index	Decorated subgoal stack	Substitution	BI
N	((';'('->'(If, Then), Else), *CP*), ...)	Σ	*ctrl*
...			

Table 34 — 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 Template and modes

```
';'('->'(goal, goal), goal)
```

NOTE — ';' and '->' are predefined infix operators so that
```
(If -> Then ; Else)
```
is parsed as
```
';'('->'(If, Then), Else)
```
.

7.8.8.3 Errors

None.

7.8.8.4 Examples

Tables 33 and 34 show the execution stack before and after executing the control construct `';'('->'(If, Then), Else)`.

Table 35 shows what happens after `(';'('->'(If, Then), Else)` is re-executed because `If` failed.

Table 35 — 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*
...			

```
';'('->'(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.

';'('->'(fail, X = 1), X = 2).
   Succeeds, unifying X with 2.

';'('->'(true, ';'(X=1, X=2)), true).
   Succeeds, unifying X with 1.
   On re-execution, succeeds, unifying X with 2.

';'('->'(';'(X=1, X=2), true), true).
   Succeeds, unifying X with 1.

';'('->'(!,fail), true), true).
   Succeeds.
```

7.8.9 catch/3

The `catch` and `throw` (7.8.10) control constructs enable execution to continue 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 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 alternative 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 (1) `call(G)` is true, or (2) the call of `G` is interrupted by a call of `throw/1` whose argument unifies with `C`, and `call(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 7.8.9.1 f.

7.8.9.2 Template and modes

```
catch(?callable_term, ?term, ?term)
```

Table 36 — Before executing `catch(G, C, R)`

S_*index*	Decorated subgoal stack	Substitution	BI
N	((`catch(G, C, R)`, CP), ...)	Σ	*ctrl*
...			

Table 37 — 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*
...			

7.8.9.3 Errors

a) `G` is a variable
— `instantiation_error`.

b) `G` is neither a variable nor a callable term
— `type_error(callable, G)`.

7.8.9.4 Examples

Tables 36 and 37 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)).

bar(X) :-
   X = Y, throw(Y).

coo(X) :-
   throw(X).

car(X) :-
   X = 1, throw(X).

g :-
   catch(p, B, write(h2)),
   coo(c).

p.
p :-
   throw(b).
```

```
catch(foo(5), test(Y), true).
   Succeeds, unifying Y with 10.

catch(bar(3), Z, true).
   Succeeds, unifying Z with 3.

catch(true, _, 3).
   Succeeds.

catch(true, C, write(demoen)), throw(bla).
   system_error.

catch(car(X), Y, true).
   Succeeds, unifying Y with 1.

catch(number_chars(X, ['1', 'a', '0']),
      error(syntax_error(_), _), fail).
   Fails -- number_chars raises a syntax error.

catch(g, C, write(h1)).
   Succeeds, unifying C with c and writing h1.
   On re-execution, fails.

catch(coo(X), Y, true).
   Succeeds, unifying Y with
     error(instantiation_error, Imp_def)
   where 'Imp_def' is an implementation defined
   term.
   ['throw(X)' is causes a goal
   throw(error(instantiation_error, Imp_def))
   to be executed].
```

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 renamed copy CA of *curract*, and a copy CP of *cutparent*.

b) Pop *currstate* from S.

c) It shall be a system error (7.12.2 j) 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 7.8.10.1 f.

e) Else replace CP by the *cutparent* for the new *curract*, and continue at 7.8.10.1 b.

f) Apply *MGU* to *currentgoal*.

Table 38 — 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*
...			

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), like a cut, pops execution states from S until a particular condition is satisfied. 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 Template and modes

```
throw(+nonvar)
```

7.8.10.3 Errors

a) `B` is a variable
— `instantiation_error`.

b) `B` does not unify with the `C` argument of any call of `catch/3`
— `system_error`.

7.8.10.4 Examples

Tables 38 and 39 show the execution stack before and after executing the control construct `throw(B)`, assuming μ is the substitution which resulted from unifying `B` and `C`.

See also 7.8.9.4.

Table 39 — 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 subclause defines the evaluation of a Prolog term as an expression.

7.9.1 Description

Procedurally, a Prolog term `T` is evaluated as an expression as follows:

a) If `T` is an integer or floating point value with value R, then proceeds to 7.9.1 e,

b) If `T` is a compound term, then evaluates, in an implementation dependent order, each argument A_i of `T` as an expression giving a value V_i,

c) Selects the operation F corresponding to the evaluable functor of `T` and the types of V_i,

d) Computes the value R of the operation F with operands (3.121) V_i,

e) The value of the expression is R.

NOTES

1 An error occurs if `T` is an atom or variable.

2 The built-in predicates for arithmetic evaluation (8.6) and arithmetic comparison (8.7) evaluate terms as expressions.

3 The evaluable functors supported by this part of ISO/IEC 13211 are defined in subclause 9.

7.9.2 Errors

The following errors may occur during the evaluation of an expression `E`:

a) `E` is a variable
— `instantiation_error`.

b) `E` is a compound term and an argument of `E` is a variable
— `instantiation_error`.

c) `E` is an atom or compound term and the principal functor `F/N` of `E` is not an evaluable functor
— `type_error(evaluable, F/N)`.

d) The value of an expression is **float_overflow**
— `evaluation_error(float_overflow)`.

e) The value of an expression is **int_overflow**
— `evaluation_error(int_overflow)`.

f) The value of an expression is **underflow**
— `evaluation_error(underflow)`.

g) The value of an expression is **zero_divisor**
— `evaluation_error(zero_divisor)`.

h) The value of an expression is **undefined**
— `evaluation_error(undefined)`.

7.10 Input/output

7.10.1 Sources and sinks

A source/sink (3.161) is a fundamental notion. A program can output results to a sink or input Prolog data from a source.

A source/sink always has a beginning, but has an end only if it is finite.

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/4` (8.11.5). All subsequent references to the source/sink are made by referring to a stream-term (7.10.2) or alias (7.10.2.2).

The effect of opening a source/sink more than once is undefined in this part of ISO/IEC 13211.

7.10.1.1 Input/output modes

An input/output mode is an atom which defines in a call of `open/4` the input/output operations that may be performed on a source/sink. A processor shall support the input/output modes:

`read` — Input. The source/sink is a source. If it is a file, it shall already exist and input shall start at the beginning of that source.

`write` — Output. The source/sink is a sink. If the sink already exists then it shall be emptied, and output shall start at the beginning of that sink, else an empty sink shall be created.

`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 input/output mode is `write`, the initial contents are lost.

2 A processor may support additional input/output modes, such as a mode for both inputting and outputting.

7.10.2 Streams

A stream provides a logical view of a source/sink.

7.10.2.1 Stream-term

A stream-term identifies a stream during a call of an input/output built-in predicate. It is an implementation dependent ground term which is created as a result of opening a source/sink by a call of `open/4` (8.11.5). A stream-term shall not be an atom.

A standard-conforming program shall make no assumptions about the form of the stream-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-term to represent different source/sinks at different times.

NOTE — A stream-term 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-term as an input argument also accept a stream alias as that argument. However, built-in predicates which (can) return a stream-term 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-term.

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-term for these streams shall be implementation dependent.

NOTES

1 Table 40 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 8.11.6).

7.10.2.4 Current streams

During execution there shall be a current input stream and a current output stream. By default, the current input and output streams shall be 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 the current input stream is closed, the standard input stream shall become the current input stream. When the current output stream is closed, the standard output stream shall become the current output stream.

NOTE — The standard input and output streams cannot be closed, and so the current input and output streams are always open streams.

7.10.2.5 Target stream

The input/output built-in predicates defined in subclauses 8.12, 8.13, and 8.14 shall input from or output to a target stream which is:

a) the stream associated with `S_or_a` when a built-in predicate has an argument `S_or_a` whose mode is `@stream_or_alias`,

b) the current input stream when an input built-in predicate has no explicit stream or alias argument,

c) the current output stream when an output built-in predicate has no explicit stream or alias argument,

The target stream is identified in the error terms for these built-in predicates as TS which denotes:

a) `S_or_a` when a built-in predicate has an argument `S_or_a` whose mode is `@stream_or_alias`,

b) `current_input_stream` when an input built-in predicate has no explicit stream or alias argument,

c) `current_output_stream` when an output built-in predicate has no explicit stream or alias argument.

7.10.2.6 Text streams

It shall be implementation defined whether record-based streams, non-record-based streams, or both are supported.

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 where each line is a possibly empty sequence of characters followed by an implementation dependent new line character (6.5, 6.5.4).

A processor may add or remove space characters at the ends of lines in order to conform to the conventions for representing text streams in the operating system. Any such alterations to the stream shall be implementation defined.

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.

The effect of outputting a control character (6.4.2.1) to a text stream shall be implementation defined.

NOTES

1 When a stream is connected to a record-based stream, each record is regarded as a line during Prolog execution.

2 `get_char/2` inputs data from a text stream and returns a one-char atom denoting a character. `get_code/2` inputs data from a text stream and returns a character code.

7.10.2.7 Binary streams

A binary stream is a sequence of bytes (7.1.2.1).

If bytes are output to a sink via a binary stream, and then input from that sink via a binary stream, then the bytes input shall be identical to those output, except that an implementation defined number of zero-valued bytes may be appended to the end of the data input.

NOTE — `get_byte/1` inputs data from a binary stream and returns a byte.

7.10.2.8 Stream positions

The stream position of a stream identifies an absolute position of the source/sink to which the stream is connected and defines where in the source/sink the next input or output will take place. It shall be implementation defined whether or not the stream position of a particular source/sink can be arbitrarily changed during execution of a Prolog goal. If it can, then:

a) A stream position is an implementation dependent ground term.

b) At any time, the stream can be repositioned by calling `set_stream_position/2` (8.11.9).

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, further output shall overwrite the existing contents of the sink.

When an input stream is repositioned, the contents of the stream shall be unaltered, and can be re-input.

7.10.2.9 End position of a stream

When all a stream `S` has been input (for example by `get_byte/2` or `read_term/3`) `S` has a stream position end-of-stream. At this stream position a goal to input more data shall return a specific value to indicate that end of stream has been reached. When one of these terminating values has been input, 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 immediately. When it is necessary to be certain that output has been delivered, this can be done by executing the built-in predicate `flush_output/1` (8.11.7).

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 (8.11.6.1 b).

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` (8.11.5).

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` (7.1.4.2) is `true` then it shall be possible to reposition the stream, else if `Bool` is `false` 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 input 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 a Permission Error (7.12.2 e) signifying that no more input exists in this stream.

`eof_code` — The result of input 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 input from it. This is likely to be useful when inputting from a source 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.

A processor may support one or more additional stream-options as an implementation specific feature.

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 (8.11.5.3) when `reposition(true)` is specified for a particular source/sink and repositioning it is not possible.

7.10.2.12 Options on stream closure

A close-option modifies the behaviour of `close/2` (8.11.6) 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 (7.12.2 h) or System Error condition (7.12.2 j) is satisfied, there shall be no error; instead the stream is closed and the goal succeeds.

A processor may support one or more additional close-options as an implementation specific feature.

NOTE — A `force(true)` close-option closes the stream but data and results may be lost, and the stream may be left in an inconsistent state. The purpose of `force/1` option is to allow an error handling routine to do its best to reclaim resources.

7.10.2.13 Stream properties

The properties of streams can be found using the built-in predicate `stream_property(Stream, Property)` (8.11.8). 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 input/output mode (7.10.1.1) which was specified when the source/sink was opened.

Table 40 — 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)

`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 `not`.

`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 40 defines the properties of the standard streams.

A processor may support one or more additional stream properties as an implementation specific feature.

7.10.3 Read-options list

A read-options list is a list of read-options which affects `read_term/3` (8.14.1) and its bootstrapped built-in predicates. The read-options supported shall include:

`variables(Vars)` — After inputting a term, `Vars` shall be a list of the variables in the term input, in left-to-right traversal order.

`variable_names(VN_list)` — After inputting a term, `VN_list` shall be unified with a list of elements where: (1) each element is a term `A = V`, and (2) `V` is a named variable of the term, and (3) `A` is an atom whose name is the characters of `V`.

`singletons(VN_list)` — After inputting a term, `VN_list` shall be unified with a list of elements where: (1) each element is a term `A = V`, and (2) `V` is a named variable which occurs only once in the term, and (3) `A` is an atom whose name is the characters of `V`.

A processor may support one or more additional read-options as an implementation specific feature.

NOTES

1 Anonymous variables (6.4.3) are included in a list `Vars`. Anonymous variables are not included in a list `VN_list`.

2 The process of inputting a term and the effect of a syntax error are defined in 7.4.3 and 8.14.1.

7.10.4 Write-options list

A write-options list is a list of write-options which affects `write_term/3` (8.14.2) and its bootstrapped built-in predicates. The write-options supported shall include:

`quoted(Bool)` — Iff `Bool` (7.1.4.2) is `true` each atom and functor is quoted if this would be necessary for the term to be input by `read_term/3`.

`ignore_ops(Bool)` — Iff `Bool` (7.1.4.2) 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)` — Iff `Bool` (7.1.4.2) is `true` a term of the form `'$VAR'(N)`, where `N` is an integer, is output as a variable name consisting of a capital letter possibly followed by an integer. The capital letter is the `(i+1)`th letter of the alphabet (see the syntax rule for `capital letter char`, 6.5.2), 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
 ...
```

A processor may support one or more additional write-options as an implementation specific feature.

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 there is a write-option `numbervars(true)`. This write-option is provided so that the built-in predicates `write/1` and `writeq/1` (8.14.2) are compatible with existing practice, but this write-option is more useful when the processor provides the built-in predicate `numbervars/3` as an extension.

7.10.5 Writing a term

When a term `Term` is output using `write_term/3` (8.14.2) the action which is taken is defined by the rules below:

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_term/3`, 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 input 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 (6.1.2 b, 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 subclause 7.10.4 is output,

f) Else 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) Else if `Term` has the form `'.'(Head,Tail)`, and there is an effective write-option `ignore_ops(false)`, 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, `Tail` is output by recursively applying these rules, and finally, `]` (close list char) is output.

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 operator 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 operator 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 — A processor may output the floating point value 1.5 as "1.5" or "1.5E+00" or "0.15e1".

Table 41 — Flags defining I parameters

Parameter	Flag
bounded	bounded
minint	min_integer
maxint	max_integer

Table 42 — Further flags for I

Feature	Flag
rnd_I	integer_rounding_function

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 (7.12.2 c). The range of values associated with some flags can be extended with additional implementation specific values.

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 a flag whose value is 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 41 identifies the parameters which define the integer type I (see 7.1.2) with the corresponding flags.

Table 42 identifies the ISO/IEC 10967-1 – Language Independent Arithmetic (LIA) integer rounding function (see 9.1.3.1) with the flag whose value indicates 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 before execution begins, 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 (3.121) 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 value of this flag is 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 value of this flag is 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` (9.1.3.1). A value `down` indicates that the rounding function is $\lfloor x \rfloor$, and a value `toward_zero` indicates that it is $tr(x)$.

7.11.2 Other flags

7.11.2.1 Flag: char_conversion

Possible values: `on, off`

Default value: `on`

Changeable: Yes

Description: If the value is `on`, (1) unquoted characters in Prolog texts being prepared for execution are converted according to the mapping $Conv_C$ (3.46) defined by previous executions of the directive `char_conversion/2` (7.4.2.5), and (2) unquoted characters in Prolog read-terms are converted according to the mapping $Conv_C$ defined by previous executions of the built-in predicate `char_conversion/2` (8.14.5).

If the value is `off`, unquoted characters in Prolog texts and read-terms are not converted.

NOTE — It is implementation defined whether or not $Conv_C$ during execution is affected by $Conv_C$ created while Prolog text is prepared for execution (see 7.4.2.5).

7.11.2.2 Flag: debug

Possible values: `on, off`

Default value: `off`

Changeable: Yes

Description: When the value is `off`, procedures have the meaning defined by this part of ISO/IEC 13211; when the value is `on`, the effect of executing any goal 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: unknown

Possible values: `error, fail, warning`

Default value: `error`

Changeable: Yes

Description: Defines the effect of attempting to execute a procedure which does not exist in the complete database (see 7.5, 7.7.7 b).

7.11.2.5 Flag: double_quotes

Possible values: `chars, codes, atom`

Default value: implementation defined

Changeable: Yes

Description: This flag determines the abstract syntax of a double quoted list token appearing in a Prolog text or in a term input by `read_term/3` (8.14.1). When value is `chars`, a double quoted list is input as a list of one-char atoms; when value is `codes`, a double quoted list is input as a list of character codes; when value is `atom`, a double quoted list is input as an atom.

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 part of ISO/IEC 13211 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 7.7.7 b, 7.11.2.4).

b) The processor is too small, or execution requires too many resources (see 7.12.2 h).

c) Execution cannot be completed because of some event outside the Prolog processor, for example a disc crash or interrupt (see 7.12.2 j).

d) The value of an evaluable functor is one of the exceptional values (9.1.2).

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 part of ISO/IEC 13211 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, and an instantiated argument or component is required. 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,
   atomic,
   byte,
   callable,
   character,
   compound,
   evaluable,
   in_byte,
   in_character,
   integer,
   list,
   number,
   predicate_indicator,
   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 procedure is defined. It has the form `domain_error(ValidDomain, Culprit)` where

```
ValidDomain ∈ {
   character_code_list,
   close_option,
   flag_value,
   io_mode,
   non_empty_list,
   not_less_than_zero,
   operator_priority,
   operator_specifier,
   prolog_flag,
   read_option,
   source_sink,
   stream,
   stream_option,
   stream_or_alias,
   stream_position,
   stream_property,
   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 ∈ {
   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, PermissionType, Culprit)` where

```
Operation ∈ {
   access,
   create,
   input,
   modify,
   open,
   output,
   reposition
},
```

and

```
PermissionType ∈ {
   binary_stream
   flag,
   operator,
   past_end_of_stream,
   private_procedure,
   static_procedure,
```

```
    source_sink,
    stream,
    text_stream
},
```

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,
    in_character_code,
    max_arity,
    max_integer,
    min_integer
}.
```

g) There shall be a Evaluation Error when the operands (3.121) of an evaluable functor are such that the operation has an exceptional value (9.1.2). It has the form `evaluation_error(Error)` where

```
Error ∈ {
    float_overflow,
    int_overflow,
    undefined,
    underflow,
    zero_divisor
}.
```

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 atom.

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(imp_dep_atom)` where `imp_dep_atom` denotes an implementation dependent atom.

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 Type Error occurs when a value does not belong to one the types defined in this part of ISO/IEC 13211 and a Domain Error occurs when the value is not a member of an implementation defined or implementation dependent set.

2 Most errors defined in this part of ISO/IEC 13211 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.

3 A Resource Error may happen for example when a calculation on unbounded integers has a value which is too large.

4 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 7.7.12.

2 The built-in predicates described in subclause 8.x are defined formally in subclause A.5.x.

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 (7.12.2 h) because, for example, the program has opened too many streams, or a file or disk is full. The use of these built-in predicates may also cause a System Error (7.12.2 j) because the operating system is reporting a problem.

The precise reason for such errors, and the ways they can be circumvented is not specified in this part of ISO/IEC 13211.

8.1 The format of built-in predicate definitions

These subclauses define the format of the definitions of built-in predicates.

8.1.1 Description

The description of the built-in predicate assumes that no error condition is satisfied, and is in two parts: (1) the logical condition for the built-in predicate to be true, and (2) a procedural description of 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" subclause includes a note that the predicate name is a predefined operator (see 6.3.4.4, table 7).

8.1.2.1 Type of an argument

The type of each argument is defined by one of the following atoms:

`atom` — an atom (3.12),

`atom_or_atom_list` — an atom or a list of atoms,

`atomic` — an atomic term (3.15),

`byte` — a byte (7.1.2.1),

`callable_term` — as terminology,

`character` — a one-char atom,

`character_code` — a character code (7.1.2.2),

`character_code_list` — a list of character codes (7.1.2.2),

`character_list` — a list of one-char atoms,

`clause` — as terminology,

`close_options` — a list of close options (8.11.6),

`compound_term` — as terminology,

`evaluable` — an expression (3.69),

`flag` — an atom associated with a Prolog flag (see 7.11),

`head` — as terminology,

`in_byte` — a byte or the integer `-1`,

`in_character` — a one-char atom or the atom `end_of_file`,

`in_character_code` — a character code or the integer `-1`,

`integer` — an integer,

`io_mode` — an input/output mode (7.10.1.1),

`list` — as terminology,

`nonvar` — an atomic term or compound term,

`number` — as terminology,

`operator_specifier` — one of the atoms: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `fx`, `fy`,

`predicate_indicator` — as terminology,

`read_options_list` — a read-options list (7.10.3),

`source_sink` — as terminology,

`stream` — as terminology,

`stream_options` — a list of stream options (7.10.2.11),

`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 (7.10.4, 7.1.4.2).

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 goal succeeds.

NOTE — When the argument is an atomic term, 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 associated error terms for the built-in predicate.

NOTES

1 When the type of an argument is `term`, the argument can be any term and no error is associated with this argument.

2 The effect of an error condition being satisfied is defined in subclause 7.12.

3 When a built-in predicate has a single mode and template, an argument whose type is not `term` and whose mode is `+` or `@` is always associated with two error conditions: an instantiation error when the argument is a variable, and a type error when the argument is neither a variable nor of the correct type.

4 When a built-in predicate has a single mode and template, an argument whose mode is `?`, and type is not `term` is always associated with an error condition: a type error when the argument is neither a variable nor of the correct type.

5 When a built-in predicate has a single mode and template, an argument whose mode is `-` is always associated with an error condition: a type error when the argument is not a variable.

6 When a built-in predicate has more than one mode and template, an argument whose mode is either `-` or `+` is always associated with an error condition: a type error when the argument is neither a variable nor of the correct type.

8.1.4 Examples

An example is normally a predication executing the built-in predicate 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.

8.1.5 Bootstrapped built-in predicates

Sometimes several built-in predicates have similar functionality. In such cases, one or more bootstrapped built-in predicates are defined as special cases of a more general built-in predicate.

The description of a bootstrapped built-in predicate states how it relates to the general built-in predicate, usually followed by a definition in Prolog that defines the logical and procedural behaviour of the bootstrapped built-in predicate when no error conditions are satisfied.

The error conditions and examples for a bootstrapped built-in predicate are included in the appropriate clauses of the general built-in predicate.

8.2 Term unification

These built-in 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), the goal 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 the goal succeeds,

c) Else if the two terms `X` and `Y` are *NSTO* and not unifiable, the goal fails.

NOTE — This built-in predicate can 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 the goal succeeds.

b) Else if `X` and `Y` are not unifiable, the goal 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 built-in predicate can be implemented much less efficiently than `(=)/2` (8.2.1). 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*, the goal is undefined,

b) Else if the two terms X and Y are *NSTO* and unifiable, the goal fails,

c) Else if the two terms X and Y are *NSTO* and not unifiable, the goal succeeds.

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.

\=(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 built-in predicates test the type associated with a term as defined in 7.1.

A goal executing any of these built-in predicates simply succeeds or fails; there is no side effect, unification, or error.

8.3.1 var/1

8.3.1.1 Description

var(X) is true iff X is a member 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 a member 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 a member 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 float/1

8.3.4.1 Description

`float(X)` is true iff `X` is a member of the set F (7.1.3).

8.3.4.2 Template and modes

```
float(@term)
```

8.3.4.3 Errors

None.

8.3.4.4 Examples

```
float(3.3).
   Succeeds.

float(-3.3).
   Succeeds.

float(3).
   Fails.

float(atom).
   Fails.

float(X).
   Fails.
```

8.3.5 atomic/1

8.3.5.1 Description

`atomic(X)` is true if `X` is a member of the set A or I or F (7.1.4, 7.1.2, 7.1.3) and is false if `X` is a member of the set V or CT (7.1.1, 7.1.5).

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 a member 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([]).
   Fails.

compound([a]).
   Succeeds.
```

8.3.7 nonvar/1

8.3.7.1 Description

`nonvar(X)` is true iff `X` is not a member 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 a member of the set I or F (7.1.2, 7.1.3) and is false if `X` is a member of the set V, A or CT (7.1.1, 7.1.4, 7.1.5).

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 built-in predicates test the ordering of two terms as defined in 7.2.

A goal executing any of these built-in predicates simply succeeds or fails; there is no side effect, unification, or error.

8.4.1 (@=<)/2 – term less than or equal, (==)/2 – term identical, (\==)/2 – term not identical, (@<)/2 – term less than, (@>)/2 – term greater than, (@>=)/2 – term greater than or equal

8.4.1.1 Description

`'@=<'(X, Y)` is true iff `X` *term_precedes* `Y` (7.2), or `X` and `Y` are identical terms (3.87).

Procedurally, `'@=<'(X, Y)` is executed as follows:

a) If `X` and `Y` are identical, the goal succeeds.

b) Else if `X` *term_precedes* `Y`, the goal succeeds,

c) Else the goal fails.

8.4.1.2 Template and modes

```
'@=<'(@term, @term)
'=='(@term, @term)
'\\=='(@term, @term)
'@<'(@term, @term)
'@>'(@term, @term)
'@>='(@term, @term)
```

NOTE — `@=<`, `==`, `\==`, `@<`, `@>`, and `@>=` are predefined infix operators (see 6.3.4.4).

8.4.1.3 Errors

None.

8.4.1.4 Examples

```
'@=<'(1.0, 1).
   Succeeds.

'@<'(1.0, 1).
   Succeeds.

'\\=='(1, 1).
   Fails.

'@=<'(aardvark, zebra).
   Succeeds.

'@=<'(short, short).
   Succeeds.

'@=<'(short, shorter).
   Succeeds.

'@>='(short, shorter).
   Fails.

'@<'(foo(a, b), north(a)).
   Fails.

'@>'(foo(b), foo(a)).
   Succeeds.

'@<'(foo(a, X), foo(b, Y)).
   Succeeds.

'@<'(foo(X, a), foo(Y, b)).
   Implementation dependent.

'@=<'(X, X).
   Succeeds.

'=='(X, X).
   Succeeds.

'@=<'(X, Y).
   Implementation dependent.

'=='(X, Y).
   Fails.

\==(_, _).
   Succeeds.

'=='(_, _).
   Fails.

'@=<'(_, _).
   Implementation dependent.

'@=<'(foo(X, a), foo(Y, b)).
   Implementation dependent.
```

8.4.1.5 Bootstrapped built-in predicates

The built-in predicates `(==)/2` (term identical), `(\\==)/2` (term not identical), `(@<)/2` (term less than), `(@>)/2` (term greater than), and `(@>=)/2` (term greater than or equal) also test the identity and term-precedence of their arguments:

The goals `'=='(X, Y)`, `'\\=='(X, Y)`, `'@<'(X, Y)`,

`'@>'(X, Y)`, and `'@>='(X, Y)` are defined as follows:

a) `'=='(X, Y)` is true iff `X` and `Y` are identical terms (3.87),

b) `'\\=='(X, Y)` is true iff `X` and `Y` are not identical terms,

c) `'@<'(X, Y)` is true iff `X` *term_precedes* `Y` (7.2),

d) `'@>'(X, Y)` is true iff `Y` *term_precedes* `X`,

e) `'@>='(X, Y)` is true iff `Y` *term_precedes* `X`, or `X` and `Y` are identical terms.

8.5 Term creation and decomposition

These built-in 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 an atomic term equal to `Name` and `Arity` is `0`.

Procedurally, `functor(Term, Name, Arity)` is executed as follows:

a) If `Term` is an atomic term, then proceeds to 8.5.1.1 d,

b) If `Term` is a compound term, then proceeds to 8.5.1.1 f,

c) If `Term` is a variable, then proceeds to 8.5.1.1 h,

d) If `Name` unifies with `Term`, and `Arity` unifies with `0`, the goal succeeds.

e) Else the goal fails.

f) If `Name` unifies with the identifier of the functor of `Term`, and `Arity` unifies with the arity of the functor of `Term`, the goal succeeds,

g) Else the goal fails.

h) If `Name` is an atomic term and `Arity` is `0`, then unifies `Term` with `Name`, and the goal succeeds,

i) Else if `Name` is an atom and `Arity` is an integer greater than zero, then instantiates `Term` with a term that has functor with identifier `Name` and arity `Arity`, and `Arity` distinct fresh variables, and the goal succeeds,

j) Else the goal fails.

8.5.1.2 Template and modes

```
functor(-nonvar, +atomic, +integer)
functor(+nonvar, ?atomic, ?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` is neither a variable nor an atomic term
— `type_error(atomic, Name)`.

d) `Term` is a variable and `Arity` is neither a variable nor an integer
— `type_error(integer, Arity)`.

e) `Term` is a variable, `Name` is a constant but not an atom, and `Arity` is greater than zero
— `type_error(atom, Name)`.

f) `Term` is a variable and `Arity` is an integer greater than the implementation defined integer `max_arity`
— `representation_error(max_arity)`.

g) `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(mats(A, B), A, B).
   Succeeds, unifying A with 'mats',
   and B with 2).
```

```
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([_|_], '.', 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).

functor(F, 1.5, 1).
   type_error(atom, 1.5).

functor(F, foo(a), 1).
   type_error(atomic, foo(a)).

current_prolog_flag(max_arity, A),
      X is A + 1,
      functor(T, foo, X).
   representation_error(max_arity).

Minus_1 is 0 - 1,
      functor(F, foo, Minus_1).
   domain_error(not_less_than_zero, -1).
```

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) If `Arg` unifies with the `N`-th argument (7.1.5) of compound term `Term`, then the goal succeeds,

b) Else the goal fails.

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) `N` is neither a variable nor an integer
— `type_error(integer, N)`.

d) `Term` is neither a variable nor a compound term
— `type_error(compound, Term)`.

e) `N` is an integer less than zero
— `domain_error(not_less_than_zero, 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 an atomic term 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 an atomic term, then proceeds to 8.5.3.1 d,

b) If `Term` is a compound term, then proceeds to 8.5.3.1 f,

c) If `Term` is a variable, then proceeds to 8.5.3.1 h,

d) If `List` unifies with a list whose only element is `Term`, then the goal succeeds.

e) Else the goal fails.

f) If `List` unifies with a list whose head is the functor name of `Term` and whose tail is a list of the arguments of `Term`, then the goal succeeds,

g) Else the goal fails.

h) If `List` is a list whose only element is an atomic term, then instantiates `Term` with the single element of `List`, and the goal succeeds,

i) Else if `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 instantiates `Term` with `CT`, and the goal succeeds,

j) Else the goal 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` is a variable and `List` is a partial list
— `instantiation_error`.

b) `List` is neither a partial list nor a list
— `type_error(list, List)`.

c) `Term` is a variable and `List` is a list whose head is a variable
— `instantiation_error`.

d) `List` is a list whose head `H` is neither an atom nor a variable, and whose tail is not the empty list
— `type_error(atom, H)`.

e) `List` is a list whose head `H` is a compound term, and whose tail is the empty list
— `type_error(atomic, H)`.

f) `Term` is a variable and `List` is the empty list
— `domain_error(non_empty_list, List)`.

g) `Term` is a variable and the tail of `List` has a length greater than the implementation defined integer `max_arity`
— `representation_error(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).

'=..'(X, [1.1, foo]).
   type_error(atom, 1.1).

'=..'(X, [a(b), 1]).
   type_error(atom, a(b)).

'=..'(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`, then the goal succeeds,

c) Else the goal fails.

NOTE — If the variable sets of `Term_1` and `Term_2` are disjoint, then even if the goal succeeds, `Term_1` will be unaltered, and the variable sets of both argumcnts 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

```
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.
```

NOTE — No unifications take place in the examples above unless explicitly described.

8.6 Arithmetic evaluation

This built-in predicatc causes an expression to be evaluated (7.9) and a value 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 value of evaluating `Expression` as an expression is `Result`.

Procedurally, `'is'(Result, Expression)` is executed as follows:

a) Evaluates `Expression` as an expression (7.9) to produce a value `C`,

b) If `Result` unifies with `C`, then the goal succeeds,

c) Else the goal fails.

8.6.1.2 Template and modes

```
is(?term, @evaluable)
```

NOTE — `is` is a predefined infix operator (see 6.3.4.4).

8.6.1.3 Errors

a) `Expression` is a variable
— `instantiation_error`.

NOTE — The evaluation of `Expression` may also result in errors (see 7.9.2).

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'(3, 3).
   Succeeds.

'is'(3, 3.0).
   Fails.

'is'(foo, 77).
   Fails.

'is'(77, N).
   instantiation_error.
```

8.7 Arithmetic comparison

These built-in predicates cause two expressions to be evaluated (7.9) and their values to be compared.

Each arithmetic comparison built-in predicate corresponds to an operation which depends on the types of the values

which are obtained by evaluating the argument(s) of the built-in predicate.

The following table identifies the integer or floating point operations corresponding to each built-in 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}

The following operations are specified:

eq_F: $F \times F \rightarrow Boolean$
eq_I: $I \times I \rightarrow Boolean$
eq_{FI}: $F \times I \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
eq_{IF}: $I \times F \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
neq_F: $F \times F \rightarrow Boolean$
neq_I: $I \times I \rightarrow Boolean$
neq_{FI}: $F \times I \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
neq_{IF}: $I \times F \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
lss_F: $F \times F \rightarrow Boolean$
lss_I: $I \times I \rightarrow Boolean$
lss_{FI}: $F \times I \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
lss_{IF}: $I \times F \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
leq_F: $F \times F \rightarrow Boolean$
leq_I: $I \times I \rightarrow Boolean$
leq_{FI}: $F \times I \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
leq_{IF}: $I \times F \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
gtr_F: $F \times F \rightarrow Boolean$
gtr_I: $I \times I \rightarrow Boolean$
gtr_{FI}: $F \times I \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
gtr_{IF}: $I \times F \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
geq_F: $F \times F \rightarrow Boolean$
geq_I: $I \times I \rightarrow Boolean$
geq_{FI}: $F \times I \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$
geq_{IF}: $I \times F \rightarrow Boolean \cup \{\textbf{float\_overflow}\}$

For all x, $y \in F$, and m, $n \in I$ the following axioms shall apply:

$eq_F(x,y) = \textbf{true} \iff x = y$

$eq_I(m,n) = \textbf{true} \iff m = n$

$eq_{FI}(x,n) = eq_F(x, float_{I\rightarrow F}(n))$ if $float_{I\rightarrow F}(n) \in F$
$= \textbf{float\_overflow}$ if $float_{I\rightarrow F}(n) \notin F$

$eq_{IF}(n,y) = eq_{FI}(y,n)$

$neq_F(x,y) = \textbf{true} \iff x \neq y$

$neq_I(m,n) = \textbf{true} \iff m \neq n$

$neq_{FI}(x,n) = neq_F(x, float_{I\rightarrow F}(n))$ if $float_{I\rightarrow F}(n) \in F$
$= \textbf{float\_overflow}$ if $float_{I\rightarrow F}(n) \notin F$

$neq_{IF}(n,y) = neq_{FI}(y,n)$

$lss_F(x,y) = \textbf{true} \iff x < y$

$lss_I(m,n) = \textbf{true} \iff m < n$

$lss_{FI}(x,n) = lss_F(x, float_{I\rightarrow F}(n))$ if $float_{I\rightarrow F}(n) \in F$
$= \textbf{float\_overflow}$ if $float_{I\rightarrow F}(n) \notin F$

$lss_{IF}(n,y) = geq_{FI}(y,n)$

$leq_F(x,y) = \textbf{true} \iff x \leq y$

$leq_I(m,n) = \textbf{true} \iff m \leq n$

$leq_{FI}(x,n) = leq_F(x, float_{I\rightarrow F}(n))$ if $float_{I\rightarrow F}(n) \in F$
$= \textbf{float\_overflow}$ if $float_{I\rightarrow F}(n) \notin F$

$leq_{IF}(n,y) = gtr_{FI}(y,n)$

$gtr_F(x,y) = \textbf{true} \iff x > y$

$gtr_I(m,n) = \textbf{true} \iff m > n$

$gtr_{FI}(x,n) = gtr_F(x, float_{I\rightarrow F}(n))$ if $float_{I\rightarrow F}(n) \in F$
$= \textbf{float\_overflow}$ if $float_{I\rightarrow F}(n) \notin F$

$gtr_{IF}(n,y) = leq_{FI}(y,n)$

$geq_F(x,y) = \textbf{true} \iff x \geq y$

$geq_I(m,n) = \textbf{true} \iff m \geq n$

$geq_{FI}(x,n) = geq_F(x, float_{I\rightarrow F}(n))$ if $float_{I\rightarrow F}(n) \in F$
$= \textbf{float\_overflow}$ if $float_{I\rightarrow F}(n) \notin F$

$geq_{IF}(n,y) = lss_{FI}(y,n)$

NOTES

1 The arithmetic evaluable functors are defined in 9.1.1.

2 An Evaluation error (**float_overflow**) occurs when an operand (3.121) which is a large integer cannot be converted to a floating point value (see $float_{I \to F}$, 9.1.6).

8.7.1 (=:=)/2 – arithmetic equal, (=\=)/2 – arithmetic not equal, (<)/2 – arithmetic less than, (=<)/2 – arithmetic less than or equal, (>)/2 – arithmetic greater than, (>=)/2 – arithmetic greater than or equal

8.7.1.1 Description

The following requirements are true for all P where

$P \in \{$ `=:=`, `=\=`, `<`, `=<`, `>`, `>=` $\}$

`'`P`'(E1, E2)` is true iff evaluating `E1` and `E2` as expressions and performing the corresponding arithmetic operation on their values is **true**.

Procedurally, `'`P`'(E1, E2)` is executed as follows:

a) Evaluates `E1` and `E2` as an expression (7.9) to produce values `EV1` and `EV2`,

b) If the result of applying the arithmetic operation P to values `EV1` and `EV2` is **true**, then the goal succeeds,

c) Else the goal fails.

8.7.1.2 Template and modes

```
'=:='(@evaluable, @evaluable)
'=\='(@evaluable, @evaluable)
'<'(@evaluable, @evaluable)
'=<'(@evaluable, @evaluable)
'>'(@evaluable, @evaluable)
'>='(@evaluable, @evaluable)
```

NOTE — `=:=`, `=\=`, `<`, `=<`, `>`, and `>=` are predefined infix operators (see 6.3.4.4).

8.7.1.3 Errors

a) `E1` is a variable
— `instantiation_error`.

b) `E2` is a variable
— `instantiation_error`.

NOTE — The evaluation of `E1` and `E2` may result in errors (7.9.2).

8.7.1.4 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 built-in predicates enable the contents of the database (7.5) to be inspected during execution.

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).
legs(A, 7) :- A, call(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 public, 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 public 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` (7.6.3), and whose body can be converted to a term `B` (7.6.4), and

2) `H` unifies with `Head`, and

3) `B` unifies with `Body`.

b) If a non-empty list is found, then proceeds to 8.8.1.1 d,

c) Else the goal fails.

d) Chooses the first element of the list L, and the goal succeeds.

e) If all the elements of the list L have been chosen, then the goal fails,

f) Else chooses the first element of the list L which has not already been chosen, and the goal succeeds.

`clause(Head, Body)` is re-executable. On backtracking, continue at 8.8.1.1 e.

NOTE — The process of converting a clause to a term (7.6.3, 7.6.4) produces a renamed copy of the term `H :- B` corresponding to the clause.

8.8.1.2 Template and modes

```
clause(+head, ?callable_term)
```

8.8.1.3 Errors

a) `Head` is a variable
— `instantiation_error`.

b) `Head` is neither a variable nor a predication
— `type_error(callable, Head)`.

c) The predicate indicator `Pred` of `Head` is that of a private procedure
— `permission_error(access, private_procedure, Pred)`.

d) `Body` is neither a variable nor a callable term
— `type_error(callable, Body)`.

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, 6), Body).
   Succeeds, unifying Body with insect(I).

clause(legs(C, 7), Body).
   Succeeds, unifying Body with (call(C), call(C)).

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,
      private_procedure, elk/1).

clause(atom(_), Body).
   permission_error(access,
      private_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 Set_{AN} of all the terms `A/N` such that (1) the database contains a user-defined procedure whose predicate has identifier `A` and arity `N`, and (2) `A/N` unifies with `PI`,

b) If a non-empty set is found, then proceeds to 8.8.2.1 d,

c) Else the goal fails.

d) Chooses a member of Set_{AN} and the goal succeeds.

e) If all the members of Set_{AN} have been chosen, then the goal fails,

f) Else chooses a member of Set_{AN} which has not already been chosen, and the goal succeeds.

`current_predicate(PI)` is re-executable. On backtracking, continue at 8.8.2.1 e.

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 it has no clauses.

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

a) `PI` is neither a variable nor a predicate indicator
— `type_error(predicate_indicator, PI)`.

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).
   type_error(predicate_indicator, 4).
```

8.9 Clause creation and destruction

These built-in predicates enable the database (7.5) to be altered during execution.

NOTE — This part of ISO/IEC 13211 requires a "logical database update", see 7.5.4.

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 8.9.1.1 c,

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`,

g) The goal succeeds.

8.9.1.2 Template and modes

```
asserta(@clause)
```

8.9.1.3 Errors

a) `Head` is a variable
— `instantiation_error`.

b) `Head` is neither a variable nor can be converted to a predication
— `type_error(callable, Head)`.

c) `Body` cannot be converted to a goal
— `type_error(callable, Body)`.

d) The predicate indicator `Pred` of `Head` is that of a static procedure
— `permission_error(modify, static_procedure, Pred)`.

8.9.1.4 Examples

The examples defined in this subclause 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(callable, 4).

asserta( (atom(_) :- true) ).
   permission_error(modify,
      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 8.9.2.1 c,

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`,

g) The goal succeeds.

8.9.2.2 Template and modes

```
assertz(@clause)
```

8.9.2.3 Errors

a) `Head` is a variable
— `instantiation_error`.

b) `Head` is neither a variable nor can be converted to a predication
— `type_error(callable, Head)`.

c) `Body` cannot be converted to a goal
— `type_error(callable, Body).`

d) The predicate indicator `Pred` of `Head` is that of a static procedure
— `permission_error(modify, static_procedure, Pred).`

8.9.2.4 Examples

The examples defined in this subclause 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(callable, 4).

assertz( (atom(_) :- true) ).
   permission_error(modify,
      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 `Clause` which unifies with `Head :- Body`.

Procedurally, `retract(Clause)` is executed as follows:

a) If `Clause` unifies with `':-'(Head, Body)`, then proceeds to 8.9.3.1 c,

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` (7.6.3), and whose body can be converted to a term `B` (7.6.4), and

2) `H` unifies with `Head`, and

3) `B` unifies with `Body`.

d) If a non-empty list is found, then proceeds to 8.9.3.1 f,

e) Else the goal fails.

f) Chooses the first element of the list L, removes the clause corresponding to it from the database, and the goal succeeds.

g) If all the elements of the list L have been chosen, then the goal 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 goal succeeds.

`retract(Clause)` is re-executable. On backtracking, continue at 8.9.3.1 g.

8.9.3.2 Template and modes

```
retract(+clause)
```

8.9.3.3 Errors

a) `Head` is a variable
— `instantiation_error`.

b) `Head` is neither a variable nor can be converted to a predication
— `type_error(callable, Head)`.

c) The predicate indicator `Pred` of `Head` is that of a static procedure
— `permission_error(access, static_procedure, Pred)`.

8.9.3.4 Examples

The examples defined in this subclause 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(spider, 8) :- true'.
   On re-execution, fails.

retract( (legs(X, Y) :- Z) ).
   Fails.
   [legs/2 has no clauses.]

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) )).
   Undefined
   [ An attempt to unify two terms:
     :-(foo(A), (A,       call(A))) and
     :-(foo(X), (call(X), call(X)))
     when examining the clause
      'foo(X) :- call(X), call(X)'].

retract(( foo(C) :- A -> B )).
   Succeeds, unifying A and B with call(C),
   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,
      static_procedure, atom/1).
```

8.9.4 abolish/1

8.9.4.1 Description

`abolish(Pred)` is true.

Procedurally, `abolish(Pred)` is executed as follows:

a) If the database contains a dynamic procedure whose predicate indicator is `Pred`, then proceeds to 8.9.4.1 c,

b) Else the goal succeeds.

c) Removes from the database the procedure specified by the predicate indicator `A/N` and all its clauses, and the goal succeeds.

NOTE — `abolish(Pred)` leaves the database in the same state as if the procedures identified by `Pred` 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 neither a variable nor a predicate indicator
— `type_error(predicate_indicator, Pred)`.

d) `Pred` is a term `Name/Arity` and `Arity` is neither a variable nor an integer
— `type_error(integer, Arity)`.

e) `Pred` is a term `Name/Arity` and `Name` is neither a variable nor an atom
— `type_error(atom, Name)`.

f) `Pred` is a term `Name/Arity` and `Arity` is an integer less than zero
— `domain_error(not_less_than_zero, Arity)`.

g) `Pred` is a term `Name/Arity` and `Arity` is an integer greater than the implementation defined integer `max_arity`
— `representation_error(max_arity)`.

h) The predicate indicator `Pred` is that of a static 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 indicator
   foo/2.

abolish(foo/_).
   instantiation_error.

abolish(foo).
   type_error(predicate_indicator, foo).

abolish(foo(_)).
   type_error(predicate_indicator, foo(_)).

abolish(abolish/1).
   permission_error(modify,
      static_procedure, abolish/1).
```

8.10 All solutions

These built-in predicates create a list of all the solutions of a goal.

8.10.1 findall/3

8.10.1.1 Description

`findall(Template, Goal, Instances)` is true iff `Instances` unifies with the list of values to which a variable `X` not occurring in `Template` or `Goal` would be instantiated by successive re-executions of

```
call(Goal), X=Template
```

after systematic replacement of all variables in `X` by new variables.

Procedurally, `findall(Template, Goal, Instances)` is executed as follows:

a) Creates an empty list L,

b) Executes `call(Goal)`,

c) If it fails, then proceeds to 8.10.1.1 g,

d) Else if it succeeds, appends the list $[CL]$ to L where CL is a renamed copy (7.1.6.2) of `Template`,

e) Re-executes `call(Goal)`,

f) Proceeds to 8.10.1.1 c,

g) Unifies L with `Instances`,

h) If the unification succeeds, the goal succeeds,

i) Else the goal 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 neither a variable nor a callable term
— `type_error(callable, Goal)`.

c) `Instances` is neither a partial list nor a list
— `type_error(list, Instances)`.

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, (X=1;X=2), [X, Y]).
   Succeeds, unifying X with 1, and Y with 2.

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:

— `Instances` is a non-empty list of `Template` such that `call(G)` is true where `G` is the iterated-goal term (7.1.6.3) of `Goal`, and

— Each element of `Instances` corresponds to an instance of `Witness` where `Witness` is a witness (7.1.1.2) of the free variables set (7.1.1.4) of `Goal` with respect to `Template`, and

— The elements of `Instances` are in order of solution of the iterated-goal term (7.1.6.3) of `Goal`.

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 the goal fails,

e) Else proceeds to step 8.10.2.1 f.

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 `E` is an element of `S_next` iff `E` is not an element of `WT_list`,

j) Replaces `S` by `S_next`,

k) Unifies `Witness` with each `WW` defined in 8.10.2.1 g,

l) If `T_list` unifies with `Instances`, then the goal succeeds,

m) Else proceeds to step 8.10.2.1 d.

`bagof(Template, Goal, Instances)` is re-executable. On backtracking, continue at 8.10.2.1 d.

NOTES

1 Step 8.10.2.1 f 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 goal can succeed only once.

3 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) The iterated-goal term G of Goal is a variable
— instantiation_error.

b) The iterated-goal term G of Goal is neither a variable nor a callable term
— type_error(callable, G).

c) Instances is neither a partial list nor a list
— type_error(list, Instances).

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, (X=Y ; X=Z), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y, Z].

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 there is no definition for the
   procedure (^)/2, and that the value associated
   with flag 'unknown' 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, 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).
```

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. Each list is a sorted list, but the order in which each list is found is undefined.

8.10.3.1 Description

setof(Template, Goal, Instances) is true iff

— Instance_list is a non-empty list of Template such that call(G) is true where G is the iterated-goal term (7.1.6.3) of Goal, and

— Each element of Instance_list corresponds to an instance of Witness where Witness 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 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) Executes the goal findall(Witness+Template, G, S),

d) If S is the empty list, the goal fails.

e) Else proceeds to step 8.10.3.1 f.

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 E is an element of S_next iff E is not an element WT_list,

j) Let ST_list be the sorted list (7.1.6.5) of T_list,

k) Replaces S by S_next,

l) Unifies Witness with each WW defined in 8.10.3.1 g,

m) If ST_list unifies with Instances, then the goal succeeds,

n) Else proceeds to step 8.10.3.1 d.

setof(Template, Goal, Instances) is re-executable. On backtracking, continue at 8.10.3.1 d.

8.10.3.2 Template and modes

```
setof(?term, +callable_term, ?list)
```

8.10.3.3 Errors

a) The iterated-goal term G of Goal is a variable
— instantiation_error.

b) The iterated-goal term G of Goal is neither a variable nor a callable term
— type_error(callable, G).

c) Instances is neither a partial list nor a list
— type_error(list, Instances).

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 there is no definition for the
   procedure (^)/2, and that the value associated
   with flag 'unknown' 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}.
   Implementation dependent.
   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.
   [If the previous example succeeds,
       unifying L with [f(U,b),f(V,c)],
     then this example fails.
   If the previous example succeeds,
       unifying L with [f(V,c),f(U,b)],
     then this example succeeds,
       unifying U with a, and V with a).]

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]].

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 _.

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 built-in predicates link an external source/sink with a Prolog stream, its stream-term and stream alias. They enable the source/sink to be opened and closed, and its properties found during execution.

NOTE — Some of these built-in predicates may cause a Resource Error (7.12.2 h) because, for example, the program has opened too many streams, or a file or disk is full. Some of these built-in predicates may also cause a System Error (7.12.2 j) 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 are undefined in this part of ISO/IEC 13211.

8.11.1 current_input/1

8.11.1.1 Description

`current_input(Stream)` is true iff the stream-term `Stream` identifies the current input stream (7.10.2.4).

Procedurally, `current_input(Stream)` is executed as follows:

a) Unifies `Stream` with the stream-term of the current input stream,

b) The goal succeeds.

8.11.1.2 Template and modes

```
current_input(?stream)
```

8.11.1.3 Errors

a) `Stream` is neither a variable nor a stream
— `domain_error(stream, Stream)`.

8.11.2 current_output/1

8.11.2.1 Description

`current_output(Stream)` is true iff the stream-term `Stream` identifies the current output stream (7.10.2.4).

Procedurally, `current_output(Stream)` is executed as follows:

a) Unifies `Stream` with the stream-term of the current output stream,

b) The goal succeeds.

8.11.2.2 Template and modes

```
current_output(?stream)
```

8.11.2.3 Errors

a) `Stream` is neither a variable nor a stream
— `domain_error(stream, Stream)`.

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) Sets the current input stream to be the stream associated with stream-term or alias `S_or_a`,

b) The goal 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-term 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) Sets the current output stream to be the stream associated with stream-term or alias `S_or_a`,

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-term 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/4, open/3

8.11.5.1 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 input/output mode `Mode` and the list of stream-options `Options` (7.10.2.11).

b) Instantiates `Stream` with the stream-term which is to be associated with this stream,

c) The goal succeeds.

8.11.5.2 Template and modes

```
open(@source_sink, @io_mode, -stream,
  @stream_options)
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) `Options` is a partial list or a list with an element `E` which is a variable
— `instantiation_error`.

d) Mode is neither a variable nor an atom
— type_error(atom, Mode).

e) Options is neither a partial list nor a list
— type_error(list, Options).

f) Stream is not a variable
— type_error(variable, Stream).

g) Source_sink is neither a variable nor a source/sink
— domain_error(source_sink, Source_sink).

h) Mode is an atom but not an input/output mode
— domain_error(io_mode, Mode).

i) An element E of the Options list is neither a variable nor a stream-option
— domain_error(stream_option, E).

j) The source/sink specified by Source_sink does not exist
— existence_error(source_sink, Source_sink).

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.

8.11.5.4 Examples

```
open('/user/roger/data', read, D, [type(binary)]).
   Succeeds.
   [It opens the binary file '/user/roger/data'
   for input, and unifies D with a
   stream-term for the stream.]

open('/user/scowen', write, D, [alias(editor)]).
   Succeeds.
   [It opens the text file '/user/scowen' for
   output, unifies D with a stream-term for the
   stream, and associates the alias 'editor'
   with the stream.]

open('/user/dave/data', read, DD, []).
   Succeeds.
   [It opens the text file '/user/dave/data'
   for input, and unifies DD with a
   stream-term for the stream.]
```

8.11.5.5 Bootstrapped built-in predicate

The built-in predicate open/3 provides similar functionality to open/4 except that a goal open(Source_sink, Mode, Stream) opens the source/sink Source_sink with an empty list of stream-options.

```
open(Source_sink, Mode, Stream) :-
   open(Source_sink, Mode, Stream, []).
```

8.11.6 close/2, close/1

This built-in predicate closes the stream associated with stream-term or alias S_or_a if it is open. The behaviour of this built-in predicate may be modified by specifying a list of close-options (7.10.2.12) in the Options parameter.

8.11.6.1 Description

close(S_or_a, Options) is true.

Procedurally, close(S_or_a, Options) is executed as follows:

a) If there is a close-option force(true), ignores any Resource Error condition (7.12.2 h) or System Error condition (7.12.2 j) that may be satisfied, and proceeds to 8.11.6.1 c,

b) Any output which is currently buffered by the processor for the stream associated with S_or_a is sent to that stream (7.10.2.10),

c) If the stream-term or alias S_or_a is the standard input stream or the standard output stream, then proceeds to 8.11.6.1 i,

d) If the stream associated with S_or_a is not the current input stream, then proceeds to 8.11.6.1 f,

e) The current input stream becomes the standard input stream user_input,

f) If the stream associated with S_or_a is not the current output stream, then proceeds to 8.11.6.1 h,

g) The current output stream becomes the standard output stream user_output,

h) Closes the stream associated with S_or_a and deletes any alias associated with that stream,

i) The goal succeeds.

8.11.6.2 Template and modes

```
close(@stream_or_alias, @close_options)
close(@stream_or_alias)
```

8.11.6.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Options` is a partial list or a list with an element `E` which is a variable
— `instantiation_error`.

c) `Options` is neither a partial list nor a list
— `type_error(list, Options)`.

d) `S_or_a` is neither a variable nor a stream-term or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) An element `E` of the `Options` list is neither a variable nor a close-option
— `domain_error(close_option, E)`.

f) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

8.11.6.4 Bootstrapped built-in predicate

The built-in predicate `close/1` provides similar functionality to `close/2` except that a goal `close(S_or_a)` closes, with an empty list of close-options, the stream associated with stream-term or alias `S_or_a` if it is open.

```
close(S_or_a) :-
   close(S_or_a, []).
```

8.11.7 flush_output/1, flush_output/0

NOTE — Flushing an output stream is explained in 7.10.2.10.

8.11.7.1 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-term or alias `S_or_a` is sent to that stream,

b) The goal succeeds.

8.11.7.2 Template and modes

```
flush_output(@stream_or_alias)
flush_output
```

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-term 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)`.

8.11.7.4 Bootstrapped built-in predicates

The built-in predicate `flush_output/0` provides similar functionality to `flush_output/1` except that a goal `flush_output` flushes the current output stream.

```
flush_output :-
   current_output(S),
   flush_output(S).
```

8.11.8 stream_property/2, at_end_of_stream/0, at_end_of_stream/1

8.11.8.1 Description

`stream_property(Stream, Property)` is true iff the stream associated with the stream-term `Stream` has stream property (7.10.2.13) `Property`.

Procedurally, `stream_property(Stream, Property)` is executed as follows:

a) Creates a set Set_{SP} of all terms `(S,P)` such that `S` is a currently open stream which has property `P`,

b) If Set_{SP} is empty, the goal fails,

c) Else, chooses a member `(SS,PP)` of Set_{SP} and removes it from the set,

d) Unifies `SS` with `Stream`, and `PP` with `Property`,

e) If the unification succeeds, the goal succeeds,

f) Else proceeds to 8.11.8.1 b.

stream_property(Stream, Property) is re-executable. On backtracking, continue at 8.11.8.1 b.

The order in which properties are found by stream_property/2 is implementation dependent.

8.11.8.2 Template and modes

```
stream_property(?stream, ?stream_property)
at_end_of_stream
at_end_of_stream(@stream_or_alias)
```

8.11.8.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) Stream is neither a variable nor a stream-term
— domain_error(stream, Stream).

c) Property is neither a variable nor a stream property
— domain_error(stream_property, Property).

d) S_or_a is neither a variable nor a stream-term or alias
— domain_error(stream_or_alias, S_or_a).

e) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

8.11.8.4 Examples

```
stream_property(S, file_name(F)).
    Succeeds, unifying S with a stream-term
       and F with the name of the file to which
       it is connected.
    On re-execution, succeeds in turn with
       each stream that is connected to a file.

stream_property(S, output).
    Succeeds, unifying S with a stream-term
       which is open for output.
    On re-execution, succeeds in turn
       with each stream that is open for output.
```

8.11.8.5 Bootstrapped built-in predicates

The built-in predicates at_end_of_stream/0 and at_end_of_stream/1 examine the single stream-property end_of_stream/1.

A goal 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).

A goal at_end_of_stream(S_or_a) is true iff the stream associated with stream-term or alias S_or_a has a stream position end-of-stream or past-end-of-stream.

```
at_end_of_stream :-
    current_input(S),
    stream_property(S, end_of_stream(E)),
    !,
    (E = at ; E = past).

at_end_of_stream (S_or_a) :-
    ( atom(S_or_a) ->
      stream_property(S, alias(S_or_a))
    ; S = S_or_a
    ),
    stream_property(S, end_of_stream(E)),
    !,
    (E = at ; E = past).
```

8.11.9 set_stream_position/2

8.11.9.1 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-term or alias S_or_a to Position,

b) The goal Succeeds.

NOTE — Normally, Position will previously have been returned as a position/1 stream property of the stream.

8.11.9.2 Template and modes

```
set_stream_position(@stream_or_alias,
  @stream_position)
```

8.11.9.3 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-term 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 character code to be input from and output to a text stream.

8.12.1 get_char/2, get_char/1, get_code/1, get_code/2

8.12.1.1 Description

`get_char(S_or_a, Char)` is true iff `Char` unifies with the next character to be input from the target stream (7.10.2.5).

Procedurally, `get_char(S_or_a, Char)` is executed as follows:

a) If the stream position of the target stream is past-end-of-stream, then proceeds to 8.12.1.1 j,

b) Else if the stream position of the target stream is end-of-stream, then proceeds to 8.12.1.1 g,

c) Else let `C` be the next character to be input from the target stream,

d) Changes the stream position of the target stream to take account of the character which has been input,

e) If `Char` unifies with a one-char atom whose name is `C`, the goal succeeds,

f) Else the goal fails.

g) Sets the stream position so that it is past-end-of-stream,

h) If the atom `end_of_file` unifies with `Char`, the goal succeeds,

i) Else the goal fails.

j) Performs the action specified in subclause 7.10.2.11 appropriate to the value of `A` where the target stream has stream property `eof_action(A)`.

8.12.1.2 Template and modes

```
get_char(?in_character)
get_char(@stream_or_alias, ?in_character)
get_code(?in_character_code)
get_code(@stream_or_alias, ?in_character_code)
```

8.12.1.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Char` is neither a variable nor an in-character
— `type_error(in_character, Char)`.

c) `Code` is neither a variable nor an integer
— `type_error(integer, Code)`.

d) `S_or_a` is neither a variable nor a stream-term or alias
— `domain_error(stream_or_alias, S_or_a)`.

e) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

f) `S_or_a` is an output stream
— `permission_error(input, stream, S_or_a)`.

g) The target stream is associated with a binary stream
— `permission_error(input, binary_stream, TS)`.

h) The target stream has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `permission_error(input, past_end_of_stream, TS)`.

i) The entity input from the stream is not a character (7.1.4.1)
— `representation_error(character)`.

j) `Code` is an integer but not an in-character code (7.1.2.2)
— `representation_error(in_character_code)`.

8.12.1.4 Examples

```
get_char(Char).
   If the contents of current input stream are
      qwerty ...
   Succeeds, unifying Char with 'q' and
   the current input stream is left as
      werty ...

get_code(Code).
   If the contents of current input stream are
      qwerty ...
   Succeeds, unifying Code with 0'q and
   the current input stream is left as
      werty ...

get_char(st_i, Char).
   If the contents of the stream associated
   with st_i are
      qwerty ...
   Succeeds, unifying Char with 'q' and
```

```
   st_i is left as
      werty ...

get_code(st_i, Code).
   If the contents of the stream associated
   with st_i are
      qwerty ...
   Succeeds, unifying Code with 0'q and
   st_i is left as
      werty ...

get_char(st_i, Char).
   If the contents of the stream associated
   with st_i are
      'qwerty' ...
   Succeeds, unifying Char with '''' (the
   atom containing just a single
   quote) and st_i is left as
      qwerty' ...

get_code(st_i, Code).
   If the contents of the stream associated
   with st_i are
      'qwerty' ...
   Succeeds, unifying Code with 0''' and
   st_i is left as
      qwerty' ...

get_char(st_i, p).
   If the contents of the stream associated
   with st_i are
      qwerty ...
   Fails.  The stream associated
   with st_i is left as
      werty ...

get_code(st_i, 0'p).
   If the contents of the stream associated
   with st_i are
      qwerty ...
   Fails.  The stream associated
   with st_i is left as
      werty ...

get_char(st_i, Char).
   If the stream position of the
      stream associated with st_i is end-of-stream
   Succeeds, unifying Char with end_of_file,
   and sets stream position of st_i to
   past-end-of-stream.

get_code(st_i, Code).
   If the stream position of the
      stream associated with st_i is end-of-stream
   Succeeds, unifying Code with -1,
   and sets stream position of st_i to
   past-end-of-stream.

get_char(user_output, X).
   permission_error(input, stream, user_output).

get_code(user_output, X).
   permission_error(input, stream, user_output).
```

8.12.1.5 Bootstrapped built-in predicates

The built-in predicates `get_char/1`, `get_code/1`, and `get_code/2` all provide similar functionality to `get_char/2`.

Goals `get_char(Char)` unifies `Char` with a one-char atom whose name is the character which has been input, and `get_code(Code)` and `get_code(S_or_a, Code)` unify `Code` with the character code of the character which has been input.

```
get_char(Char) :-
   current_input(S), get_char(S, Char).

get_code(Code) :-
   current_input(S),
   get_char(S, Char),
   ( Char = end_of_file ->
     Code = -1
   ; char_code(Char, Code)
   ).

get_code(S, Code) :-
   get_char(S, Char),
   ( Char = end_of_file ->
     Code = -1
   ; char_code(Char, Code)
   ).
```

NOTE — The built-in predicate `char_code/2` is defined in 8.16.6.

8.12.2 peek_char/2, peek_char/1, peek_code/1, peek_code/2

8.12.2.1 Description

`peek_char(S_or_a, Char)` is true iff `Char` unifies with the next character to be input from the target stream (7.10.2.5).

Procedurally, `peek_char(S_or_a, Char)` is executed as follows:

a) If the stream position of the target stream is past-end-of-stream, then proceeds to 8.12.2.1 h,

b) Else if the stream position of the target stream is end-of-stream, then proceeds to 8.12.2.1 f,

c) Else let `C` be the next character to be input from the target stream,

d) If `Char` unifies with a one-char atom whose name is `C`, the goal succeeds,

e) Else the goal fails.

f) If the atom `end_of_file` unifies with `Char`, the goal succeeds,

g) Else the goal fails.

h) Performs the action specified in subclause 7.10.2.11 appropriate to the value of `A` where the target stream has stream property `eof_action(A)`.

NOTE — `peek_char(S_or_a, Char)` leaves unaltered the stream position of the target stream.

8.12.2.2 Template and modes

```
peek_char(?in_character)
peek_char(@stream_or_alias, ?in_character)
peek_code(?in_character_code)
peek_code(@stream_or_alias, ?in_character_code)
```

8.12.2.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Char` is neither a variable nor an in-character
— `type_error(in_character, Char)`.

c) `Code` is neither a variable nor an integer
— `type_error(integer, Code)`.

d) `S_or_a` is neither a variable nor a stream-term or alias
— `domain_error(stream_or_alias, S_or_a)`.

e) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

f) `S_or_a` is an output stream
— `permission_error(input, stream, S_or_a)`.

g) The target stream is associated with a binary stream
— `permission_error(input, binary_stream, TS)`.

h) The target stream has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `permission_error(input, past_end_of_stream, TS)`.

i) The next entity to be input from the stream is not a character (7.1.4.1)
— `representation_error(character)`.

j) `Code` is an integer but not an in-character code (7.1.2.2)
— `representation_error(in_character_code)`.

8.12.2.4 Examples

```
peek_char(Char).
   If the contents of current input stream are
      qwerty ...
   Succeeds, unifying Char with 'q' and
   the current input stream is left as
      qwerty ...

peek_code(Code).
   If the contents of current input stream are
      qwerty ...
   Succeeds, unifying Code with 0'q and
   the current input stream is left as
      qwerty ...

peek_char(st_i, Char).
   If the contents of the stream associated
   with st_i are
      qwerty ...
   Succeeds, unifying Char with 'q' and
   st_i is left as
      qwerty ...

peek_code(st_i, Code).
   If the contents of the stream associated
   with st_i are
      qwerty ...
   Succeeds, unifying Code with 0'q and
   st_i is left as
      qwerty ...

peek_char(st_i, Char).
   If the contents of the stream associated
   with st_i are
      'qwerty' ...
   Succeeds, unifying Char with '''' (the
   atom containing just a single
   quote) and st_i is left as
      'qwerty' ...

peek_code(st_i, Code).
   If the contents of the stream associated
   with st_i are
      'qwerty' ...
   Succeeds, unifying Code with 0''' and
   st_i is left as
      'qwerty' ...

peek_char(st_i, p).
   If the contents of the stream associated
   with st_i are
      qwerty ...
   Fails.  The stream associated
   with st_i is left as
      qwerty ...

peek_code(st_i, 0'p).
   If the contents of the stream associated
   with st_i are
      qwerty ...
   Fails.  The stream associated
   with st_i is left as
      qwerty ...

peek_char(st_i, Char).
   If the stream position of the stream
      associated with st_i is end-of-stream
   Succeeds, unifying Char with end_of_file, and
   sets stream position of st_i to end-of-stream.

peek_code(st_i, Code).
   If the stream position of the
      stream associated with st_i is end-of-stream
   Succeeds, unifying Code with -1, and
   sets stream position of st_i to end-of-stream.

peek_char(s, Char).
   If the stream position of the stream
      associated with s is past-end-of-stream,
      and s has stream property eof_action(error)
   permission_error(input, past_end_of_stream, s).
```

```
peek_char(user_output, X).
   permission_error(input, stream, user_output).

peek_code(user_output, X).
   permission_error(input, stream, user_output).
```

8.12.2.5 Bootstrapped built-in predicates

The built-in predicates `peek_char/1`, `peek_code/1`, and `peek_code/2` all provide similar functionality to `peek_char/2`.

Goals `peek_char(Char)` unifies `Char` with a one-char atom whose name is the next character to be input, and `peek_code(Code)` and `peek_code(S_or_a, Code)` unify `Code` with the character code of the next character.

```
peek_char(Char) :-
   current_input(S),
   peek_char(S, Char).

peek_code(Code) :-
   current_input(S),
   peek_char(S, Char),
   ( Char = end_of_file ->
     Code = -1
   ; char_code(Char, Code)
   ).

peek_code(S, Code) :-
   peek_char(S, Char),
   ( Char = end_of_file ->
     Code = -1
   ; char_code(Char, Code)
   ).
```

NOTE — The built-in predicate `char_code/2` is defined in 8.16.6.

8.12.3 put_char/2, put_char/1, put_code/1, put_code/2, nl/0, nl/1

8.12.3.1 Description

`put_char(S_or_a, Char)` is true.

Procedurally, `put_char(S_or_a, Char)` is executed as follows:

a) Outputs the character `C` which is the name of the one-char atom `Char` to the target stream (7.10.2.5).

b) Changes the stream position of the target stream to take account of the character which has been output,

c) The goal succeeds.

8.12.3.2 Template and modes

```
put_char(+character)
put_char(@stream_or_alias, +character)
put_code(+character_code)
put_code(@stream_or_alias, +character_code)
nl
nl(@stream_or_alias)
```

8.12.3.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Char` is a variable
— `instantiation_error`.

c) `Code` is a variable
— `instantiation_error`.

d) `Char` is neither a variable nor a one-char atom
— `type_error(character, Char)`.

e) `Code` is neither a variable nor an integer
— `type_error(integer, Code)`.

f) `S_or_a` is neither a variable nor a stream-term or alias
— `domain_error(stream_or_alias, S_or_a)`.

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)`.

i) The target stream is associated with a binary stream
— `permission_error(output, binary_stream, TS)`.

j) `Char` is neither a variable nor a character (7.1.4.1)
— `representation_error(character)`.

k) `Code` is an integer but not a character code (7.1.2.2)
— `representation_error(character_code)`.

8.12.3.4 Examples

```
put_char(t).
   If the contents of current output stream are
      ... qwer
   Succeeds, and the current output stream
   is left as
      ... qwert

put_char(st_o, 'A').
   If the contents of the stream associated
   with st_o are
```

```
      ... qwer
   Succeeds, and the stream associated with st_o
   is left as
      ... qwerA

put_code(0't).
   If the contents of current output stream are
      ... qwer
   Succeeds, and the current output stream
   is left as
      ... qwert

put_code(st_o, 0't).
   If the contents of the stream associated
   with st_o are
      ... qwer
   Succeeds, and the stream associated with st_o
   is left as
      ... qwert

nl, put_char(a).
   If the contents of current output stream are
      ... qwer
   Succeeds, and the current output stream
   is left as
      ... qwer
      a

nl(st_o), put_char(st_o, a).
   If the contents of the stream associated
   with st_o are
      ... qwer
   Succeeds, and the stream associated with st_o
   is left as
      ... qwer
      a

put_char(my_file, C).
   instantiation_error.

put_char(st_o, 'ty').
   type_error(character, ty).

put_code(my_file, C).
   instantiation_error.

put_code(st_o, 'ty').
   type_error(integer, ty).

nl(Str).
   instantiation_error.

nl(user_input).
   permission_error(output, stream, user_input).
```

8.12.3.5 Bootstrapped built-in predicates

The built-in predicates `put_char/1`, `put_code/1`, `put_code/2`, `nl/0`, and `nl/1` all provide similar functionality to `put_char/2`.

A goal `put_char(Char)` outputs the character which is the name of `Char`, `put_code(Code)` and `put_code(S_or_a, Code)` output the character whose character code is `Code`, and `nl` and `nl(S_or_a)` output the implementation dependent new line character (6.5.4).

```
put_char(Char) :-
   current_output(S),
   put_char(S, Char).

put_code(Code) :-
   current_output(S),
   char_code(Char, Code),
   put_char(S, Char).

put_code(S, Code) :-
   char_code(Char, Code),
   put_char(S, Char).

nl :-
   current_output(S),
   put_char(S, '\n').

nl(S) :-
   put_char(S, '\n').
```

NOTE — The built-in predicates `nl/0` and `nl/1` terminate the current line or record. The built-in predicate `char_code/2` is defined in 8.16.6.

8.13 Byte input/output

These built-in predicates enable a single byte to be input from and output to a binary stream.

8.13.1 get_byte/2, get_byte/1

8.13.1.1 Description

`get_byte(S_or_a, Byte)` is true iff `Byte` unifies with the next byte to be input from the target stream (7.10.2.5).

Procedurally, `get_byte(S_or_a, Byte)` is executed as follows:

a) If the stream position of the target stream is past-end-of-stream, then proceeds to 8.13.1.1 k,

b) If the target stream has stream property `eof_action(A)` and its stream position is past-end-of-stream, then performs the action appropriate to the value of `A` specified in subclause 7.10.2.11.

c) Else if the stream position of the target stream is end-of-stream, then proceeds to 8.13.1.1 h,

d) Else let `B` be the next byte to be input from the target stream,

e) Changes the stream position of the target stream to take account of the byte which has been input,

f) If `B` unifies with `Byte`, the goal succeeds,

g) Else the goal fails.

h) Sets the stream position so that it is past-end-of-stream,

i) If the integer value -1 unifies with Byte, the goal succeeds,

j) Else the goal fails.

k) Performs the action specified in subclause 7.10.2.11 appropriate to the value of A where the target stream has stream property eof_action(A).

8.13.1.2 Template and modes

```
get_byte(?in_byte)
get_byte(@stream_or_alias, ?in_byte)
```

8.13.1.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) Byte is neither a variable nor an in-byte
— type_error(in_byte, Byte).

c) S_or_a is neither a variable nor a stream-term 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).

f) The target stream is associated with a text stream
— permission_error(input, text_stream, *TS*).

g) The target stream has stream properties end_of_stream(past) and eof_action(error) (7.10.2.9, 7.10.2.11, 7.10.2.13)
— permission_error(input, past_end_of_stream, *TS*).

8.13.1.4 Examples

```
get_byte(Byte).
   If the contents of the current input stream are
      [113,119,101,114, ...]
   Byte is unified with 113 and
   the current input stream is left as
      [119,101,114, ...]

get_byte(st_i, Byte).
   If the contents of the stream associated
   with st_i are
      [113,119,101,114, ...]
   Byte is unified with 113 and
   st_i is left as
      [119,101,114, ...]

get_byte(st_i, 114).
   If the contents of the stream associated
   with st_i are
      [113,119,101,114,116,121 ...]
   Fails.  The stream associated
   with st_i is left as
      [119,101,114,116,121 ...]

get_byte(st_i, Byte).
   Stream position of st_i is end-of-stream.
   Byte is unified with -1 and
   stream position of st_i is past-end-of-stream.

get_byte(user_output, X).
   permission_error(input, stream, user_output).
```

8.13.1.5 Bootstrapped built-in predicate

The built-in predicate get_byte/1 provides similar functionality to get_byte/2.

```
get_byte(Byte) :-
   current_input(S),
   get_byte(S, Byte).
```

8.13.2 peek_byte/2, peek_byte/1

8.13.2.1 Description

peek_byte(S_or_a, Byte) is true iff Byte unifies with the next byte to be input from the target stream (7.10.2.5).

Procedurally, peek_byte(S_or_a, Byte) is executed as follows:

a) If the stream position of the target stream is past-end-of-stream, then proceeds to 8.13.2.1 h,

b) Else if the stream position of the target stream is end-of-stream, then proceeds to 8.13.2.1 f,

c) Else let B be the next byte to be input from the target stream,

d) If B unifies with Byte, the goal succeeds,

e) Else the goal fails.

f) If the integer value -1 unifies with Byte, the goal succeeds,

g) Else the goal fails.

h) Performs the action specified in subclause 7.10.2.11 appropriate to the value of A where the target stream has stream property eof_action(A).

NOTE — peek_byte(S_or_a, Byte) leaves unaltered the stream position of the target stream.

8.13.2.2 Template and modes

```
peek_byte(?in_byte)
peek_byte(@stream_or_alias, ?in_byte)
```

8.13.2.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) Byte is neither a variable nor an in-byte
— type_error(in_byte, Byte).

c) S_or_a is neither a variable nor a stream-term 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).

f) The target stream is associated with a text stream
— permission_error(input, text_stream, *TS*).

g) The target stream has stream properties end_of_stream(past) and eof_action(error) (7.10.2.9, 7.10.2.11, 7.10.2.13)
— permission_error(input, past_end_of_stream, *TS*).

8.13.2.4 Examples

```
peek_byte(Byte).
   If the contents of current input stream are
      [113,119,101,114, ...]
   Byte is unified with 113 and
   the current input stream is left as
      [113,119,101,114, ...]

peek_byte(st_i, Byte).
   If the contents of the stream associated
   with st_i are
      [113,119,101,114, ...]
   Byte is unified with 113 and
   st_i is left as
      [113,119,101,114, ...]

peek_byte(st_i, 114).
   If the contents of the stream associated
   with st_i are
      [113,119,101,114, ...]
   Fails.  The stream associated
   with st_i is left as
      [113,119,101,114, ...]

peek_byte(st_i, Byte).
   Stream position of st_i is end-of-stream.
   Byte is unified with -1 and
   stream position of st_i is end-of-stream.

peek_byte(user_output, X).
   permission_error(input, stream, user_output).
```

8.13.2.5 Bootstrapped built-in predicate

The built-in predicate peek_byte/1 provides similar functionality to peek_byte/2.

```
peek_byte(Byte) :-
   current_input(S),
   peek_byte(S, Byte).
```

8.13.3 put_byte/2, put_byte/1

8.13.3.1 Description

put_byte(S_or_a, Byte) is true.

Procedurally, put_byte(S_or_a, Byte) is executed as follows:

a) Outputs the byte Byte to the target stream (7.10.2.5).

b) Changes the stream position of the target stream to take account of the byte which has been output,

c) The goal succeeds.

8.13.3.2 Template and modes

```
put_byte(+byte)
put_byte(@stream_or_alias, +byte)
```

8.13.3.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) Byte is a variable
— instantiation_error.

c) Byte is neither a variable nor a byte
— type_error(byte, Byte).

d) S_or_a is neither a variable nor a stream-term or alias
— domain_error(stream_or_alias, S_or_a).

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) The target stream is associated with a text stream
— permission_error(output, text_stream, *TS*).

8.13.3.4 Examples

```
put_byte(84).
   If the current output stream contains
      [..., 113,119,101,114]
   Succeeds, and leaves that stream
      [..., 113,119,101,114,116]

put_byte(st_o, 84).
   If the stream associated with st_o contains
      [..., 113,119,101,114]
   Succeeds, and leaves that stream
      [..., 113,119,101,114,116]

put_byte(my_file, C).
   instantiation_error.

put_byte(user_output, 'ty').
   type_error(byte, ty).
```

8.13.3.5 Bootstrapped built-in predicate

The built-in predicate `put_byte/1` provides similar functionality to `put_byte/2`.

```
put_byte(Byte) :-
   current_output(S),
   put_byte(S, Byte).
```

8.14 Term input/output

These built-in predicates enable a Prolog term to be input from or output to a text 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/3, read_term/2, read/1, read/2

8.14.1.1 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 target stream (7.10.2.5).

Procedurally, `read_term(S_or_a, Term, Options)` is executed as follows:

a) Sets `C_Seq` to an empty sequence of characters,

b) Inputs a character `C` from the target stream,

c) Changes the stream position of the target stream to take account of the character which has been input,

d) If the value associated with the flag `char_conversion` (7.11.2.1) is `off`, or `C` is a quoted character (6.4.2.1), then sets `C_next` to `C`, and proceeds to 8.14.1.1 f,

e) Else sets `C_next` to $apply\_mapping_C(C, Conv_C)$ (4.3) where $Conv_C$ (3.46) is the character-conversion mapping,

f) Appends `C_next` to `C_Seq`,

g) Attempts to parse `C_Seq` as a sequence of tokens (6.4),

h) If `C_Seq` is too short, then proceeds to 8.14.1.1 b,

i) If `C_next` represents an end token (6.4.8), then proceeds to 8.14.1.1 k,

j) Else proceeds to 8.14.1.1 b,

k) Parses `C_Seq` as a read-term (6.4) `T.`,

l) If `T` unifies with `Term`, then instantiates the arguments of the read-options (7.10.3) `Options`, and the goal succeeds,

m) Else the goal fails.

NOTES

1 The two steps 8.14.1.1 d and 8.14.1.1 e ensure that whether or not a character is quoted depends only on the characters of the target stream. It is independent of the mapping $Conv_C$, or the value associated with the flag `char_conversion`.

2 The number of characters which are input is undefined when an error occurs during `read_term/3`.

8.14.1.2 Template and modes

```
read_term(@stream_or_alias, ?term,
  +read_options_list)
```

8.14.1.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Options` is a partial list or a list with an element `E` which is a variable
— `instantiation_error`.

c) `S_or_a` is neither a variable nor a stream-term or alias
— `domain_error(stream_or_alias, S_or_a)`.

d) `Options` is neither a partial list 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).

f) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

g) S_or_a is an output stream
— permission_error(input, stream, S_or_a).

h) The target stream is associated with a binary stream
— permission_error(input, binary_stream, *TS*).

i) The target stream has stream properties end_of_stream(past) and eof_action(error) (7.10.2.9, 7.10.2.11, 7.10.2.13)
— permission_error(input, past_end_of_stream, *TS*).

j) The read-term Term breaches an implementation defined limit specified by Flag where Flag is the flag (7.11) max_arity, max_integer, or min_integer
— representation_error(Flag).

k) One or more characters were input, but they cannot be parsed as a sequence of tokens
— syntax_error(imp_dep_atom).

l) The sequence of tokens cannot be parsed as a term using the current set of operator definitions
— syntax_error(imp_dep_atom).

8.14.1.4 Examples

```
read(T).
   current input stream is
term1. term2. ...
   Succeeds, unifying T with term1.
   The current input stream is left as
 term2. ...

read(st_o, term1).
   If the contents of the stream associated
   with st_o are
term1. term2. ...
   Succeeds, and the stream associated with st_o
   is left as
 term2. ...

read_term(st_o, T, [variables(VL),
      variable_names(VN), singletons(VS)]).
   If the contents of the stream associated
   with st_o are
foo(A+Roger, A+_).  term2. ...
   Succeeds, unifying T with foo(X1+X2, X1+X3),
   VL with [X1, X2, X3],
   VN with ['A' = X1, 'Roger' = X2],
   and VS with ['Roger' = X2].
   The stream associated with st_o
   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(imp_dep_atom) where 'imp_dep_atom'
   is an implementation dependent atom.
   The current input stream is left as
 term2. ...

read(T).
   current input stream is
3.1
   syntax_error(imp_dep_atom) where 'imp_dep_atom'
   is an implementation dependent atom.
   The current input stream is left with
   position past-end-of-stream.
```

8.14.1.5 Bootstrapped built-in predicates

The built-in predicates read_term/2, read/1, and read/2 all provide similar functionality to read_term/3.

Goals read_term(Term, Options), read(Term), and read(S_or_a, Term) all input characters and attempt to parse them as a term which unifies with Term.

Goals read(Term) and read(S_or_a, Term) input terms using an empty read-options list.

A goal read_term(Term, Options) instantiates the arguments of the read-options Options.

```
read_term(Term, Options) :-
   current_input(S),
   read_term(S, Term, Options).

read(Term) :-
   current_input(S),
   read_term(S, Term, []).

read(S, Term) :-
   read_term(S, Term, []).
```

8.14.2 write_term/3, write_term/2, write/1, write/2, writeq/1, writeq/2, write_canonical/1, write_canonical/2

8.14.2.1 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 target stream (7.10.2.5) in a form which is defined by the write-options list (7.10.4, 7.1.4.2) `Options` and rules for writing a term (7.10.5),

b) Changes the stream position of the target stream to take account of the characters which have been output,

c) The goal succeeds.

8.14.2.2 Template and modes

```
write_term(@stream_or_alias, @term,
  @write_options_list)
```

8.14.2.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Options` is a partial list or a list with an element `E` which is a variable
— `instantiation_error`.

c) `Options` is neither a partial list nor a list
— `type_error(list, Options)`.

d) `S_or_a` is neither a variable nor a stream-term or alias
— `domain_error(stream_or_alias, S_or_a)`.

e) An element `E` of the `Options` list is neither a variable nor a valid write-option
— `domain_error(write_option, E)`.

f) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

g) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

h) The target stream is associated with a binary stream
— `permission_error(output, binary_stream, TS)`.

8.14.2.4 Examples

```
write_term(S, [1,2,3], []).
   Succeeds, outputting the characters
[1,2,3]
   to the stream associated with S.

write_canonical([1,2,3]).
   Succeeds, outputting the characters
.(1,.(2,.(3,[ ])))
   to the current output stream.

write_term(S, '1<2', []).
   Succeeds, outputting the characters
1<2
   to the stream associated with S.

writeq(S, '1<2').
   Succeeds, outputting the characters
'1<2'
   to the stream associated with S.

writeq('$VAR'(0)).
    Succeeds, outputting the character
A
    to the current output stream.

write_term(S, '$VAR'(1), [numbervars(false)]).
    Succeeds, outputting the characters
$VAR(1)
    to the stream associated with S.

write_term(S, '$VAR'(51), [numbervars(true)]).
    Succeeds, outputting the characters
Z1
    to the stream associated with S.
```

8.14.2.5 Bootstrapped built-in predicates

The built-in predicates `write_term/2`, `write/1`, `write/2`, `writeq/1`, `writeq/2`, `write_canonical/1`, and `write_canonical/2` all provide similar functionality to `write_term/3`.

Goals `write(Term)` and `write(S_or_a, Term)` output `Term` in a form which is defined by a write-options list `[quoted(false), ignore_ops(false), numbervars(true)]`.

Goals `writeq(Term)` and `writeq(S_or_a, Term)` output `Term` in a form which is defined by a write-options list `[quoted(true), ignore_ops(false), numbervars(true)]`.

Goals `write_canonical(S_or_a, Term)` and `write_canonical(Term)` output `Term` in a form which is defined by a write-options list `[quoted(true), ignore_ops(true), numbervars(false)]`.

```
write_term(Term, Options) :-
   current_output(S),
   write_term(S, Term, Options).

write(Term) :-
   current_output(S),
   write_term(S, Term, [numbervars(true)]).

write(S, Term) :-
   write_term(S, Term, [numbervars(true)]).

writeq(Term) :-
   current_output(S),
   write_term(S, Term,
      [quoted(true), numbervars(true)]).

writeq(S, Term) :-
   write_term(S, Term,
      [quoted(true), numbervars(true)]).
```

```
write_canonical(T) :-
    current_output(S),
    write_term(S, Term,
        [quoted(true), ignore_ops(true)]).

write_canonical(S, Term) :-
    write_term(S, Term,
        [quoted(true), ignore_ops(true)]).
```

8.14.3 op/3

A goal `op(Priority, Op_specifier, Operator)` enables the operator table (see 6.3.4.4 and table 7) to be altered.

8.14.3.1 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 a member `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 8.14.3.1 f,

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`, then proceeds to 8.14.3.1 h,

g) `Op` is made an operator with specifier `Op_specifier` and priority `Priority`,

h) If *Ops* is non-empty, then proceeds to 8.14.3.1 c,

i) Else, the goal 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.

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 from the operator table.

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, operators can be removed from the operator table and their priority or specifier can be changed. However, it is an error to attempt to change the `','` operator from its initial status, see 6.3.4.3.

8.14.3.2 Template and modes

```
op(+integer, +operator_specifier,
  @atom_or_atom_list)
```

8.14.3.3 Errors

a) `Priority` is a variable
— `instantiation_error`.

b) `Op_specifier` is a variable
— `instantiation_error`.

c) `Operator` is a partial list or a list with an element `E` which is a variable
— `instantiation_error`.

d) `Priority` is neither a variable nor an integer
— `type_error(integer, Priority)`.

e) `Op_specifier` is neither a variable nor an atom
— `type_error(atom, Op_specifier)`.

f) `Operator` is neither a partial list nor a list nor an atom
— `type_error(list, Operator)`.

g) An element `E` of the `Operator` list is neither a variable nor an atom
— `type_error(atom, E)`.

h) `Priority` is not between 0 and 1200 inclusive
— `domain_error(operator_priority, Priority)`.

i) `Op_specifier` is not a valid operator specifier
— `domain_error(operator_specifier, Op_specifier)`.

j) `Operator` is `','`
— `permission_error(modify, operator, ',')`.

k) An element of the `Operator` list is `','`
— `permission_error(modify, operator, ',').`

l) `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).`

8.14.3.4 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.]
```

8.14.4 current_op/3

8.14.4.1 Description

`current_op(Priority, Op_specifier, Operator)` is true iff `Operator` is an operator with properties defined by specifier `Op_specifier` and priority `Priority`.

Procedurally, `current_op(Priority, Op_specifier, Operator)` is executed as follows:

a) Searches the current operator definitions and creates a set Set_{Op} 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, then proceeds to 8.14.4.1 d,

c) Else the goal fails.

d) Chooses a member of Set_{Op} and the goal succeeds.

e) If all the members of Set_{Op} have been chosen, then the goal fails,

f) Else chooses a member of Set_{Op} which has not already been chosen, and the goal succeeds.

`current_op(Priority, Op_specifier, Operator)` is re-executable. On backtracking, continue at 8.14.4.1 e.

The order in which operators are found by `current_op/3` is implementation dependent.

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 procedure 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 otherwise found by `current_op/3`.

8.14.4.2 Template and modes

```
current_op(?integer, ?operator_specifier,
   ?atom)
```

8.14.4.3 Errors

a) `Priority` is neither a variable nor an operator priority
— `domain_error(operator_priority, Priority).`

b) `Op_specifier` is neither a variable nor an operator specifier
— `domain_error(operator_specifier, Op_specifier).`

c) `Operator` is neither a variable nor an atom
— `type_error(atom, Operator).`

8.14.4.4 Examples

```
current_op(P, xfy, OP).
   If the operator table has 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 ','.
   On re-execution, succeeds unifying
      P with  200, and OP with '^'.
   [The order of solutions is
      implementation dependent.]
```

8.14.5 char_conversion/2

A goal `char_conversion(In_char, Out_char)` enables $Conv_C$, the character-conversion mapping (3.46), to be altered.

8.14.5.1 Description

`char_conversion(In_char, Out_char)` is true.

Procedurally, `char_conversion(In_char, Out_char)` is executed as follows:

a) Replaces $Conv_C$, the character-conversion mapping (3.46), with the conversion $update\_mapping_C(IC, OC, Conv_C)$ (4.3) where IC is the character of the name of `In_char`, and OC is the character of the name of `Out_char`.

b) The goal succeeds.

NOTES

1 See also character-conversion directives (7.4.2.5).

2 The one-char atoms `In_char` and `Out_char` should be quoted in order to ensure that their characters have not been converted by a character-conversion directive when the Prolog text is prepared for execution.

3 $Conv_C$ affects only characters input by term input (8.14). When it is necessary to convert characters input by character input/output built-in predicates (8.12), it will be necessary to program the conversion explicitly using `current_char_conversion/2` (8.14.6).

4 When `In_char` and `Out_char` are the same, the effect on $Conv_C$ is to remove any conversion of a character `In_char`.

8.14.5.2 Template and modes

```
char_conversion(+character, +character)
```

8.14.5.3 Errors

a) `In_char` is a variable
— `instantiation_error`.

b) `Out_char` is a variable
— `instantiation_error`.

c) `In_char` is neither a variable nor a one-char atom (7.1.4.1)
— `representation_error(character)`.

d) `Out_char` is neither a variable nor a one-char atom (7.1.4.1)
— `representation_error(character)`.

8.14.5.4 Examples

`char_conversion('&', ',')`
Replaces $Conv_C$ by $update\_mapping_C$(&, ',', $Conv_C$).
Succeeds.

`char_conversion('''', '\'')`
Replaces $Conv_C$ by $update\_mapping_C$(', ', $Conv_C$) where ' is a character in an extended character set equivalent to the single quote.
Succeeds.

`char_conversion('a', a)`
Replaces $Conv_C$ by $update\_mapping_C$(**a**, a, $Conv_C$) where **a** is a character in an extended character set equivalent to the small letter character a.
Succeeds.

After these three goals, when the valuc associated with flag `char_conversion` is on, all occurrences of &, ', and **a** as unquoted characters input by term input built-in predicates are converted to ,, ', and a respectively. For example, the three characters **a&a** are converted to the characters a,a. However (1) the characters '**aaa**' represent an atom '**aaa**' because they are enclosed by the single quotes, and (2) the characters '**a&a**' form an atom 'a,a'.

`char_conversion('&', '&')`
Replaces $Conv_C$ by $update\_mapping_C$(&, &, $Conv_C$) thus removing the conversion from & to ','.
Succeeds.

8.14.6 current_char_conversion/2

8.14.6.1 Description

`current_char_conversion(In_char, Out_char)` is true iff (1) $apply\_mapping_C(IC, Conv_C)$ equals OC

where (a) $Conv_C$ is the character-conversion mapping (3.46), (b) IC is the character of the name of atom `In_char`, and (c) OC is the character of the name of atom `Out_char`, and (2) `In_char` is not equal to `Out_char`.

Procedurally, `current_char_conversion(In_char, Out_char)` is executed as follows:

a) Creates a set Set_{Conv} of all the conversions (`In` → `Out`) in $Conv_C$ such that:

1) IC is the character of the name of atom `In`,

2) `In` unifies with `In_char`,

3) OC is the character of the name of atom `Out`,

4) `Out`, unifies with `Out_char`,

5) $apply\_mapping_C(IC,\ Conv_C)$ equals OC, and

6) `In` does not unify with `Out`,

b) If a non-empty set is found, then proceeds to 8.14.6.1 d,

c) Else the goal fails.

d) Chooses a member of Set_{Conv} which has not already been chosen, unifies `In` with `In_char`, and `Out` with `Out_char`, and the goal succeeds.

e) If all the members of Set_{Conv} have been chosen, then the goal fails,

f) Else proceeds to 8.14.6.1 d.

`current_char_conversion(In_char, Out_char)` is re-executable. On backtracking, continue at 8.14.6.1 e.

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 $Conv_C$ 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 which has been removed by `char_conversion(C, C)` is not otherwise found by `current_char_conversion/2`.

8.14.6.2 Template and modes

```
current_char_conversion(?character,
  ?character)
```

8.14.6.3 Errors

a) `In_char` is neither a variable nor a one-char atom — `type_error(character, In_char)`.

b) `Out_char` is neither a variable nor a one-char atom — `type_error(character, Out_char)`.

8.14.6.4 Examples

Assume $Conv_C$ is
$update\_mapping_C$(**a**, a,
$update\_mapping_C$(*a*, a,
$identity\_mapping_C$)).

`current_char_conversion(C, a)`
Succeeds, unifying C with **a**.
On re-execution, succeeds, unifying C with *a*.
[The order of solutions is
implementation dependent.]

8.15 Logic and control

These built-in predicates are simply derived from the control constructs (7.8) and provide additional facilities for affecting the control flow during execution.

8.15.1 (\+)/1 – not provable

8.15.1.1 Description

`'\\+'(Term)` is true *iff* `call(Term)` is false.

Procedurally, `'\\+'(Term)` is executed as follows:

a) Executes `call(Term)`,

b) If it succeeds, the goal fails,

c) Else if it fails, the goal succeeds.

NOTE — A built-in predicate with the same meaning as `(\+)/1` is implemented in many existing processors with a name `(not)/1`. This name is misleading because the built-in predicate gives *negation by failure* rather than true negation.

8.15.1.2 Template and modes

```
'\\+'(@callable_term)
```

NOTE — `\+` is a predefined infix operator (see 6.3.4.4).

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)`.

8.15.1.4 Examples

```
'\\+'(true).
   Fails.

\+(!).
   Fails, the cut has no effect.

'\\+'((!, fail)).
   Succeeds, the cut has no effect.

(X=1; X=2), \+((!, fail)).
   Succeeds, unifying X with 1.
   On re-execution, succeeds unifying X with 2.

'\\+'(4 = 5).
   Succeeds.

\+(3).
   type_error(callable, 3).

'\\+'(X).
   instantiation_error.

\+(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 goal succeeds,

c) Else if it fails, the goal 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)`.

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` is executed as follows:

a) The goal succeeds.

`repeat` is re-executable. On re-execution, continue at 8.15.3.1 a above.

8.15.3.2 Template and modes

```
repeat
```

8.15.3.3 Errors

None.

8.15.3.4 Examples

```
repeat, write('hello '), fail.
   Outputs
hello hello hello hello hello ...
   indefinitely.

repeat, !, fail.
   Fails, equivalent to (!, fail).
```

8.16 Atomic term processing

These built-in predicates enable atomic terms 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, to convert a single character to and from the corresponding character code, and to convert a number to and from a list of characters.

NOTE — The characters of the name of an atom and their numbering are defined in 6.1.2 b.

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 of the name of the atom `Atom`.

Procedurally, `atom_length(Atom, Length)` is executed as follows:

a) If `Length` is a variable, then instantiates `Length` with an integer equal to the number of characters of the name of the atom `Atom`, and the goal succeeds,

b) Else if `Length` is an integer, and `Length` unifies with the number of characters of the name of the atom `Atom`, then the goal succeeds,

c) Else the goal fails.

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)`.

d) `Length` is an integer that is less than zero
— `domain_error(not_less_than_zero, 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 characters of the name of the atom `Atom_12` are the result of concatenating the characters of the name of the atom `Atom_2` to the characters of the name of the atom `Atom_1`.

Procedurally, `atom_concat(Atom_1, Atom_2, Atom_12)` is executed as follows:

a) Creates the sorted list $List_{ac}$ containing as elements all the terms `ac(A1, A2, A3)` such that

1) `A1` is an atom which unifies with `Atom_1`, and

2) `A2` is an atom which unifies with `Atom_2`, and

3) `A3` is an atom which unifies with `Atom_12`, and

4) the characters of the name of `A3` are the result of concatenating the characters of the name of `A2` to the characters of the name of `A1`,

b) If a non-empty list is found, then proceeds to 8.16.2.1 d,

c) Else the goal fails.

d) Chooses the first element, `ac(AA1, AA2, AA3)`, of $List_{ac}$,

e) The goal succeeds, unifying `Atom_1` with `AA1`, unifying `Atom_2` with `AA2`, and unifying `Atom_12` with `AA3`.

f) If all the elements of $List_{ac}$ have been chosen, then the goal fails,

g) Else chooses the first element of $List_{ac}$, `ac(AA1, AA2, AA3)`, which has not already been chosen, and proceeds to step 8.16.2.1 e.

`atom_concat(Atom_1, Atom_2, Atom_12)` is re-executable. On re-execution, continue at 8.16.2.1 f above.

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

```
atom_concat('hello', ' world', S3).
   Succeeds, unifying S3 with 'hello world'.

atom_concat(T, ' world', 'small world').
   Succeeds, unifying T with 'small'.

atom_concat('hello', ' world', 'small world').
   Fails.

atom_concat(T1, T2, 'hello').
   Succeeds, unifying T1 with '',
      and T2 with 'hello'.
   On re-execution, succeeds,
       unifying T1 with 'h', and T2 with 'ello'.
   [...]

atom_concat(small, V2, V4).
   instantiation_error.
```

8.16.3 sub_atom/5

8.16.3.1 Description

`sub_atom(Atom, Before, Length, After, Sub_atom)` is true iff atom `Atom` can be broken into three pieces, `AtomL`, `Sub_atom` and `AtomR` such that `Before` is the number of characters of the name of `AtomL`, `Length` is the number of characters of the name of `Sub_atom` and `After` is the number of characters of the name of `AtomR`.

Procedurally, `sub_atom(Atom, Before, Length, After, Sub_atom)` is executed as follows:

a) Creates the sorted list $List_{sa}$ containing as elements all the terms `sa(L1, L2, L3, A2)` such that

1) there is an atom `A1` whose name has `L1` characters, and

2) there is an atom `A2` whose name has `L2` characters, and

3) there is an atom `A3` whose name has `L3` characters, and

4) `Sub_atom` unifies with `A2`, and

5) `Before` unifies with `L1`, and

6) `Length` unifies with `L2`, and

7) `After` unifies with `L3`, and

8) `Atom` is the atom whose name is the result of concatenating the characters of the name of `A3` to the characters of the name of the atom `A12`, where `A12` is the atom whose name results from concatenating the characters of the name of `A2` to the characters of the name of the atom `A1`,

b) If a non-empty list is found, then proceeds to 8.16.3.1 d,

c) Else the goal fails.

d) Chooses the first element, `sa(LL1, LL2, LL3, AA2)`, of $List_{sa}$,

e) The goal succeeds, unifying `Before` with `LL1`, unifying `Length` with `LL2`, unifying `After` with `LL3`, and unifying `Sub_atom` with `AA2`.

f) If all the elements of $List_{sa}$ have been chosen, then the goal fails,

g) Else chooses the first element of $List_{sa}$, `sa(LL1, LL2, LL3, AA2)`, which has not already been chosen, and proceeds to step 8.16.3.1 e.

`sub_atom(Atom, Before, Length, After, Sub_atom)` is re-executable. On re-execution, continue at 8.16.3.1 f above.

8.16.3.2 Template and modes

`sub_atom(+atom, ?integer, ?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) `Before` is neither a variable nor an integer
— `type_error(integer, Before)`.

e) `Length` is neither a variable nor an integer
— `type_error(integer, Length)`.

f) `After` is neither a variable nor an integer
— `type_error(integer, Length)`.

g) `Before` is an integer that is less than zero
— `domain_error(not_less_than_zero, Before)`.

h) `Length` is an integer that is less than zero
— `domain_error(not_less_than_zero, Length)`.

i) `After` is an integer that is less than zero
— `domain_error(not_less_than_zero, After)`.

8.16.3.4 Examples

```
sub_atom(abracadabra, 0, 5, _, S2).
   Succeeds, unifying S2 to 'abrac'.

sub_atom(abracadabra, _, 5, 0, S2).
   Succeeds, unifying S2 to 'dabra'.

sub_atom(abracadabra, 3, L, 3, S2).
   Succeeds, unifying L to 5
   and S2 to 'acada'.

sub_atom(abracadabra, B, 2, A, ab).
   Succeeds, unifying B to 0 and A to 9.
   On re-execution, succeeds,
   unifying B to 7 and A to 2.

sub_atom('Banana', 3, 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 0,
      and Length with 0, and Sub_atom with ''.
   On re-execution, succeeds,
      unifying Start with 0, and Length with 1,
      and Sub_atom with 'a'.
   On re-execution, succeeds,
      unifying Start with 0, and Length with 2,
      and Sub_atom with 'ab'.
   On re-execution, 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 'b'.
   On re-execution, succeeds,
      unifying Start with 2, 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 one-char atoms whose names are the successive characters of the name of atom `Atom`.

Procedurally, `atom_chars(Atom, List)` is executed as follows:

a) If `Atom` is a variable, then instantiates `Atom` with the atom whose name (see 6.1.2 b) has the same sequence of characters as the elements of `List`, and the goal succeeds,

b) Else if `List` is a variable, then instantiates `List` with a list of one-char atoms identical to the sequence of characters of the name of `Atom`, and the goal succeeds,

c) Else if `List` is a list of one-char atoms, and `Atom` is the atom whose name has the same sequence of characters, then the goal succeeds,

d) Else the goal fails.

8.16.4.2 Template and modes

```
atom_chars(+atom, ?character_list)
atom_chars(-atom, +character_list)
```

8.16.4.3 Errors

a) `Atom` is a variable and `List` is a partial list or a list with an element which is a variable
— `instantiation_error`.

b) `Atom` is neither a variable nor an atom
— `type_error(atom, Atom)`.

c) `Atom` is a variable and `List` is neither a list nor a partial list
— `type_error(list, List)`.

d) `Atom` is a variable and an element `E` of the list `List` is neither a variable nor a one-char atom
— `type_error(character, E)`.

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 the name of atom `Atom`, and the value of each element is the character code for the corresponding character of the name.

Procedurally, `atom_codes(Atom, List)` is executed as follows:

a) If `Atom` is a variable, then instantiates `Atom` with the atom whose name (see 6.1.2 b) is a sequence of characters such that the character code (7.1.2.2) of the Nth character is the Nth element of `List`, and the goal succeeds,

b) Else if `List` is a variable, then instantiates `List` with a list of character codes such that the Nth element of `List` is the character code of the Nth character of the name of `Atom`, and the goal succeeds,

c) Else if `List` is a list of character codes, and `Atom` is an atom whose name is a sequence of characters such that the character code of the Nth character is the Nth element of `List`, then the goal succeeds,

d) Else the goal fails.

8.16.5.2 Template and modes

```
atom_codes(+atom, ?character_code_list)
atom_codes(-atom, +character_code_list)
```

8.16.5.3 Errors

a) `Atom` is a variable and `List` is a partial list or a list with an element which is a variable
— `instantiation_error`.

b) `Atom` is neither a variable nor an atom
— `type_error(atom, Atom)`.

c) `Atom` is a variable and `List` is neither a list nor a partial list
— `type_error(list, List)`.

d) `Atom` is a variable and an element `E` of the list `List` is neither a variable nor a character code
— `representation_error(character_code)`.

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 one-char atom `Char` is `Code`.

Procedurally, `char_code(Char, Code)` is executed as follows:

a) If `Char` is a variable, then instantiates `Char` with the atom whose name (see 6.1.2 b) is a character corresponding to the character code (7.1.2.2) `Code` and the goal succeeds,

b) Else if `Char` is a one-char atom whose name has a character code which unifies with `Code`, then the goal succeeds,

c) Else the goal fails.

8.16.6.2 Template and modes

```
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 one-char atom
— `type_error(character, Char)`.

c) `Code` is neither a variable nor an integer
— `type_error(integer, Code)`.

d) `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 one-char atoms corresponding to a character sequence of `Number` which could be output (7.10.5 b, 7.10.5 c).

Procedurally, `number_chars(Number, List)` is executed as follows:

a) If `List` is not a list of one-char atoms, then proceeds to 8.16.7.1 e,

b) Else parses the list of the characters of the name of the one-char atoms according to the syntax rules for numbers and negative numbers (6.3.1.1, 6.3.1.2) to give a value N,

c) If `Number` unifies with N, then the goal succeeds,

d) Else the goal fails.

e) Let `LC` be a list of one-char atoms whose names correspond to the sequence of characters which would be output by `write_canonical(Number)` (see 7.10.5 b, 7.10.5 c, 8.14.2),

f) If `LC` unifies with `List`, then the goal succeeds,

g) Else the goal fails.

NOTES

1 The sequence of one-char atoms 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=['0', '.', '1'],
   number_chars(X,C), number_chars(X,C).
```

8.16.7.2 Template and modes

```
number_chars(+number, ?character_list)
number_chars(-number, +character_list)
```

8.16.7.3 Errors

a) `Number` is a variable and `List` is a partial list or a list with an element which is a variable
— `instantiation_error`.

b) `Number` is neither a variable nor a number
— `type_error(number, Number)`.

c) `Number` is a variable and `List` is neither a list nor a partial list
— `type_error(list, List)`.

d) An element `E` of the list `List` is not a one-char atom
— `type_error(character, E)`.

e) `List` is a list of one-char atoms but is not parsable as a number
— `syntax_error(imp_dep_atom)`.

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', ' ']).
   syntax_error(imp_dep_atom) where 'imp_dep_atom'
   is an implementation dependent atom.

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 (7.10.5 b, 7.10.5 c).

Procedurally, `number_codes(Number, List)` is executed as follows:

a) If `List` is not a list of character codes, then proceeds to 8.16.8.1 e,

b) Else parses the list of characters corresponding to those character codes according to the syntax rules for numbers and negative numbers (6.3.1.1, 6.3.1.2) to give a value N,

c) If `Number` unifies with N, then the goal succeeds,

d) Else the goal fails.

e) Let `LC` be a list of character codes corresponding to the sequence of characters which would be output by `write_canonical(Number)` (see 7.10.5 b, 7.10.5 c, 8.14.2),

f) If `LC` unifies with `List`, then the goal succeeds,

g) Else the goal 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, ?character_code_list)
number_codes(-number, +character_code_list)
```

8.16.8.3 Errors

a) `Number` is a variable and `List` is a partial list or a list with an element which is a variable
— `instantiation_error`.

b) `Number` is neither a variable nor a number
— `type_error(number, Number)`.

c) `Number` is a variable and `List` is neither a list nor a partial list
— `type_error(list, List)`.

d) An element `E` of the list `List` is not a character code (7.1.2.2)
— `representation_error(character_code)`.

e) `List` is a list of character codes but is not parsable as a number
— `syntax_error(imp_dep_atom)`.

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 character codes,
   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

A goal `set_prolog_flag(Flag, Value)` enables the value associated with a Prolog flag to be altered.

8.17.1.1 Description

`set_prolog_flag(Flag, Value)` is true.

Procedurally, `set_prolog_flag(Flag, Value)` is executed as follows:

a) Associates `Value` with the flag `Flag` (7.11), where `Value` is a value that is within the implementation defined range of values for `Flag`,

b) The goal succeeds.

8.17.1.2 Template and modes

```
set_prolog_flag(+flag, @nonvar)
```

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 an invalid flag for the processor
— `domain_error(prolog_flag, Flag)`.

e) `Value` is inappropriate for `Flag`
— `domain_error(flag_value, Flag + Value)`.

f) `Value` is appropriate for `Flag` but flag `Flag` is not modifiable
— `permission_error(modify, flag, Flag)`.

8.17.1.4 Examples

```
set_prolog_flag(unknown, fail).
   Succeeds, associating the value fail
   with flag unknown.

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 Set_{cpf} 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, then proceeds to 8.17.2.1 d,

c) Else the goal fails.

d) Chooses a member of Set_{cpf} and the goal succeeds.

e) If all the members of Set_{cpf} have been chosen, then the goal fails,

f) Else chooses a member of Set_{cpf} which has not already been chosen, and the goal succeeds.

`current_prolog_flag(Flag, Value)` is re-executable. On re-execution, continue at 8.17.2.1 e 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 part of ISO/IEC 13211 or implementation specific.

8.17.2.2 Template and modes

```
current_prolog_flag(?flag, ?term)
```

8.17.2.3 Errors

a) `Flag` is neither a variable nor an atom
— `type_error(atom, Flag)`.

b) `Flag` is an atom but an invalid flag for the processor
— `domain_error(prolog_flag, 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 built-in 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 built-in 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 subclause 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 of ISO/IEC 10967-1 – Language Independent Arithmetic (LIA). They conform to a subset of its requirements.

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
`sign/1`	$sign_I$, $sign_F$
`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 Exceptional values

An exceptional value is **float_overflow**, **int_overflow**, **underflow**, **zero_divisor**, or **undefined**.

NOTE — It is an `evaluation_error(E)` if the value of an expression is an exceptional value (see 7.9.2).

9.1.3 Integer operations and axioms

The following operations are specified:

$$add_I : I \times I \to I \cup \{\textbf{int\_overflow}\}$$
$$sub_I : I \times I \to I \cup \{\textbf{int\_overflow}\}$$
$$mul_I : I \times I \to I \cup \{\textbf{int\_overflow}\}$$
$$intdiv_I : I \times I \to I \cup \{\textbf{int\_overflow}, \textbf{zero\_divisor}\}$$
$$rem_I : I \times I \to I \cup \{\textbf{zero\_divisor}\}$$
$$mod_I : I \times I \to I \cup \{\textbf{zero\_divisor}\}$$
$$neg_I : I \to I \cup \{\textbf{int\_overflow}\}$$
$$abs_I : I \to I \cup \{\textbf{int\_overflow}\}$$
$$sign_I : I \to I$$

The behaviour of the integer operations are defined in terms of a rounding function $rnd_I(x)$ (see 9.1.3.1).

For all x, $y \in I$, the following axioms shall apply:

$$add_I(x,y) = x + y \quad \text{if } x+y \in I$$
$$= \textbf{int\_overflow} \quad \text{if } x+y \notin I$$

$$sub_I(x,y) = x - y \quad \text{if } x-y \in I$$
$$= \textbf{int\_overflow} \quad \text{if } x-y \notin I$$

$$mul_I(x,y) = x * y \quad \text{if } x*y \in I$$
$$= \textbf{int\_overflow} \quad \text{if } x*y \notin I$$

$$intdiv_I(x,y) = rnd_I(x/y) \quad \text{if } y \neq 0 \wedge rnd_I(x/y) \in I$$
$$= \textbf{int\_overflow} \quad \text{if } y \neq 0 \wedge rnd_I(x/y) \notin I$$
$$= \textbf{zero\_divisor} \quad \text{if } y = 0$$

$$rem_I(x,y) = x - (rnd_I(x/y) * y) \quad \text{if } y \neq 0$$
$$= \textbf{zero\_divisor} \quad \text{if } y = 0$$

$$mod_I(x,y) = x - (\lfloor x/y \rfloor * y) \quad \text{if } y \neq 0$$
$$= \textbf{zero\_divisor} \quad \text{if } y = 0$$

$$neg_I(x) = -x \quad \text{if } -x \in I$$
$$= \textbf{int\_overflow} \quad \text{if } -x \notin I$$

$$abs_I(x) = |x| \quad \text{if } |x| \in I$$
$$= \textbf{int\_overflow} \quad \text{if } |x| \notin I$$

$$sign_I(x) = 1 \quad \text{if } x > 0$$
$$= 0 \quad \text{if } x = 0$$
$$= -1 \quad \text{if } x < 0$$

9.1.3.1 Integer division rounding function

An integer division rounding function shall be implementation defined:

$$rnd_I : \mathcal{R} \rightarrow \mathcal{Z}$$

For $x \in \mathcal{R}$, the following axiom shall apply, either

$$rnd_I(x) = \lfloor x \rfloor$$

or

$$rnd_I(x) = tr(x)$$

NOTE — The notations $\lfloor x \rfloor$ and $tr(x)$ are defined in 4.1.3.3 and 4.1.3.4. The flag `integer_rounding_function` (7.11.1.4) makes the implementation defined choice of rounding function accessible to a goal.

9.1.4 Floating point operations and axioms

The following operations are specified:

$$\begin{aligned}
&add_F : F \times F \rightarrow F \cup \{\textbf{float_overflow}, \textbf{underflow}\}\\
&sub_F : F \times F \rightarrow F \cup \{\textbf{float_overflow}, \textbf{underflow}\}\\
&mul_F : F \times F \rightarrow F \cup \{\textbf{float_overflow}, \textbf{underflow}\}\\
&div_F : F \times F\\
&\quad \rightarrow F \cup \{\textbf{float_overflow}, \textbf{underflow}, \textbf{zero_divisor}\}\\
&neg_F : F \rightarrow F\\
&abs_F : F \rightarrow F\\
&sign_F : F \rightarrow F\\
&intpart_F : F \rightarrow F\\
&fractpart_F : F \rightarrow F
\end{aligned}$$

The behaviour of the floating point operations are defined in terms of a rounding function $rnd_F(x)$ (see 9.1.4.1), a floating point result function $result_F(x, round)$ (see 9.1.4.2), and an approximate-addition function $add^*_F(x, y)$ (see 9.1.4.3).

For all x, $y \in F$, $n \in I$ the following axioms shall apply:

$$\begin{aligned}
add_F(x,y) &= result_F(add^*_F(x,y),\ rnd_F)\\
sub_F(x,y) &= add_F(x,-y)\\
mul_F(x,y) &= result_F(x*y,\ rnd_F)\\
div_F(x,y) &= result_F(x/y,\ rnd_F) && \text{if } y \neq 0\\
&= \textbf{zero_divisor} && \text{if } y = 0\\
neg_F(x) &= -x\\
abs_F(x) &= |x|\\
sign_F(x) &= 1 && \text{if } x > 0\\
&= 0 && \text{if } x = 0\\
&= -1 && \text{if } x < 0\\
intpart_F(x) &= sign_F(x) * \lfloor |x| \rfloor\\
fractpart_F(x) &= x - intpart_F(x)
\end{aligned}$$

9.1.4.1 Floating point rounding function

A floating point rounding function shall be implementation defined:

$$rnd_F : \mathcal{R} \rightarrow F^*$$

For all $x \in \mathcal{R}$, $i \in \mathcal{Z}$, the following axiom shall apply:

$$rnd_F(-x) = -rnd_F(x)$$

For all $x \in \mathcal{R}$, $i \in \mathcal{Z}$, such that $|x| \geq fmin_N$ and $|x * r^i| \geq fmin_N$, the following axiom shall apply:

$$rnd_F(x * r^i) = rnd_F(x) * r^i$$

NOTE — This rule means that the rounding function does not depend on the exponent part of the floating point value except when denormalization occurs.

9.1.4.2 Floating point result function

A floating point result function shall be implementation defined:

$$result_F : \mathcal{R} \times (\mathcal{R} \rightarrow F^*) \rightarrow F \cup \{\textbf{float\_overflow}, \textbf{underflow}\}$$

For all $x \in \mathcal{R}$ and any rounding function $round \in (\mathcal{R} \rightarrow F^*)$, the following axioms shall apply:

$$\begin{aligned}
&result_F(x, round)\\
&\quad = round(x)\\
&\qquad \text{if } x = 0 \vee fmin_N \leq |x| \leq fmax\\
&\quad = round(x)\\
&\qquad \text{if } |x| > fmax \wedge |round(x)| = fmax\\
&\quad = \textbf{float_overflow}\\
&\qquad \text{if } |x| > fmax \wedge |round(x)| \neq fmax\\
&\quad = round(x) \text{ or } \textbf{underflow}\\
&\qquad \text{if } 0 < |x| < fmin_N \wedge |round(x)| \leq fmin_N
\end{aligned}$$

It shall be implementation defined whether a processor chooses $round(x)$ or **underflow** when $0 < |x| < fmin_N$.

9.1.4.3 Floating point approximate-addition function

A floating point approximate-addition function shall be implementation defined:

$$add^*_F : F \times F \rightarrow \mathcal{R}$$

For all u, v, x, $y \in F$, $i \in \mathcal{Z}$, the following axioms shall apply:

$$add^*_F(u, v) = add^*_F(v, u)$$

$$add^*_F(-u, -v) = -add^*_F(u, v)$$

$$x \le (u + v) \le y \;\Rightarrow\; x \le (add^*_F(u, v)) \le y$$

$$u \le v \;\Rightarrow\; add^*_F(u, x) \le add^*_F(v, x)$$

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.

NOTE — The five axioms for the approximate-addition ensure:

a) add^*_F is commutative,

b) add^*_F is sign symmetric,

c) $add^*_F(u, v)$ is in the same "basic interval" as $u + v$, and is exact if $u + v$ is exactly representable (a "basic interval" is the range between two adjacent values of F),

d) add^*_F is monotonic,

e) add^*_F does not depend on the exponents of its arguments, only their differences.

9.1.5 Mixed mode operations and axioms

These operations convert the integer operand or operands (3.121) to floating point and then use the appropriate floating point operation.

The following operations are specified:

$$add_{FI} : F \times I \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}\}$$
$$add_{IF} : I \times F \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}\}$$
$$sub_{FI} : F \times I \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}\}$$
$$sub_{IF} : I \times F \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}\}$$
$$mul_{FI} : F \times I \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}\}$$
$$mul_{IF} : I \times F \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}\}$$
$$div_{FI} : F \times I \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}, \textbf{zero\_divisor}\}$$
$$div_{IF} : I \times F \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}, \textbf{zero\_divisor}\}$$
$$div_{II} : I \times I \to F \cup \{\textbf{float\_overflow}, \textbf{underflow}, \textbf{zero\_divisor}\}$$

For all x, $y \in F$, m, $n \in I$, the following axioms shall apply:

$$add_{FI}(x, n) = add_F(x, float_{I\to F}(n)) \text{ if } float_{I\to F}(n) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F$$

$$add_{IF}(n, x) = add_F(float_{I\to F}(n), x) \text{ if } float_{I\to F}(n) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F$$

$$sub_{FI}(x, n) = sub_F(x, float_{I\to F}(n)) \text{ if } float_{I\to F}(n) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F$$

$$sub_{IF}(n, x) = sub_F(float_{I\to F}(n), x) \text{ if } float_{I\to F}(n) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F$$

$$mul_{FI}(x, n) = mul_F(x, float_{I\to F}(n)) \text{ if } float_{I\to F}(n) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F$$

$$mul_{IF}(n, x) = mul_F(float_{I\to F}(n), x) \text{ if } float_{I\to F}(n) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F$$

$$div_{FI}(x, n) = div_F(x, float_{I\to F}(n)) \text{ if } float_{I\to F}(n) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F$$

$$div_{IF}(n, x) = div_F(float_{I\to F}(n), x) \text{ if } float_{I\to F}(n) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F$$

$$div_{II}(n, m) = div_F(float_{I\to F}(n), float_{I\to F}(m)) \text{ if } float_{I\to F}(n) \in F \wedge float_{I\to F}(m) \in F$$
$$= \textbf{float\_overflow} \text{ if } float_{I\to F}(n) \notin F \vee float_{I\to F}(m) \notin F$$

NOTE — A floating point value is never implicitly converted to an integer. The programmer must state which conversion is

to be applied, see 9.1.6.1.

9.1.6 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 behaviour of the type conversion operations are defined in terms of a rounding function $rnd_F(x)$ (see 9.1.4.1), a floating point result function $result_F(x, round)$ (see 9.1.4.2), and floating point to integer rounding functions 9.1.6.1.

$$float_{I \to F} : I \to F \cup \{\textbf{float\_overflow}\}$$
$$float_{F \to F} : F \to F$$
$$floor_{F \to I} : F \to I \cup \{\textbf{int\_overflow}\}$$
$$truncate_{F \to I} : F \to I \cup \{\textbf{int\_overflow}\}$$
$$round_{F \to I} : F \to I \cup \{\textbf{int\_overflow}\}$$
$$ceiling_{F \to I} : F \to I \cup \{\textbf{int\_overflow}\}$$

For all $x \in F$, $n \in I$, the following axioms shall apply:

$$float_{I \to F}(n) = result_F(n,\ rnd_F)$$

$$float_{F \to F}(x) = x$$

$$floor_{F \to I}(x) = floor_{\mathcal{R} \to \mathcal{Z}}(x) \quad \text{if } floor_{\mathcal{R} \to \mathcal{Z}}(x) \in I$$
$$= \textbf{int\_overflow} \quad \text{if } floor_{\mathcal{R} \to \mathcal{Z}}(x) \notin I$$

$$truncate_{F \to I}(x) = truncate_{\mathcal{R} \to \mathcal{Z}}(x) \quad \text{if } truncate_{\mathcal{R} \to \mathcal{Z}}(x) \in I$$
$$= \textbf{int\_overflow} \quad \text{if } truncate_{\mathcal{R} \to \mathcal{Z}}(x) \notin I$$

$$round_{F \to I}(x) = round_{\mathcal{R} \to \mathcal{Z}}(x) \quad \text{if } round_{\mathcal{R} \to \mathcal{Z}}(x) \in I$$
$$= \textbf{int\_overflow} \quad \text{if } round_{\mathcal{R} \to \mathcal{Z}}(x) \notin I$$

$$ceiling_{F \to I}(x) = ceiling_{\mathcal{R} \to \mathcal{Z}}(x) \quad \text{if } ceiling_{\mathcal{R} \to \mathcal{Z}}(x) \in I$$
$$= \textbf{int\_overflow} \quad \text{if } ceiling_{\mathcal{R} \to \mathcal{Z}}(x) \notin I$$

9.1.6.1 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 $x \in \mathcal{R}$, $n \in \mathcal{Z}$, the following axioms shall apply:

$$floor_{\mathcal{R} \to \mathcal{Z}}(x) = \lfloor x \rfloor$$

$$truncate_{\mathcal{R} \to \mathcal{Z}}(x) = \lfloor x \rfloor \quad \text{if } x \geq 0$$
$$= -\lfloor |x| \rfloor \quad \text{if } x < 0$$

$$round_{\mathcal{R} \to \mathcal{Z}}(x) = \lfloor x + 1/2 \rfloor$$

$$ceiling_{\mathcal{R} \to \mathcal{Z}}(x) = -\lfloor -x \rfloor$$

9.1.7 Examples

```
'+'(7, 35).
   Evaluates to the value 42.

'+'(0, 3+11).
   Evaluates to the value 14.

'+'(0, 3.2+11).
   Evaluates to a value
   approximately equal to 14.2000.

'+'(77, N).
   instantiation_error.

'+'(foo, 77).
   type_error(number, foo).

'-'(7).
   Evaluates to the value -7.

'-'(3-11).
   Evaluates to the value 8.

'-'(3.2-11).
   Evaluates to a value
   approximately equal to 7.8000.

'-'(N).
   instantiation_error.

'-'(foo).
   type_error(number, foo).

'-'(7, 35).
   Evaluates to the value -28.

'-'(20, 3+11).
   Evaluates to the value 6.

'-'(0, 3.2+11).
   Evaluates to a value
   approximately equal to -14.2000.

'-'(77, N).
   instantiation_error.

'-'(foo, 77).
   type_error(number, foo).

'*'(7, 35).
   Evaluates to the value 245.

'*'(0, 3+11).
   Evaluates to the value 0.

'*'(1.5, 3.2+11).
   Evaluates to a value
   approximately equal to 21.3000.
```

```
'*'(77, N).
   instantiation_error.

'*'(foo, 77).
   type_error(number, foo).

'/'(7, 35).
   Evaluates to the value 0.

'/'(7.0, 35).
   Evaluates to a value
   approximately equal to 0.2000.

'/'(140, 3+11).
   Evaluates to the value 10.

'/'(20.164, 3.2+11).
   Evaluates to a value
   approximately equal to 14.2000.

'/'(7, -3).
   Evaluates to an implementation defined value.

'/'(-7, 3).
   Evaluates to an implementation defined value.

'/'(77, N).
   instantiation_error.

'/'(foo, 77).
   type_error(number, foo).

'/'(3, 0).
   evaluation_error(zero_divisor).

mod(7, 3).
   Evaluates to the value 1.

mod(0, 3+11).
   Evaluates to the value 0.

mod(7, -2).
   Evaluates to the value -1.

mod(77, N).
   instantiation_error.

mod(foo, 77).
   type_error(number, foo).

mod(7.5, 2).
   type_error(integer, 7.5).

mod(7, 0).
   evaluation_error(zero_divisor).

floor(7.4).
   Evaluates to the value 7.

floor(-0.4).
   Evaluates to the value -1.

round(7.5).
   Evaluates to the value 8.

round(7.6).
   Evaluates to the value 8.

round(-0.6).
   Evaluates to the value -1.

round(N).
   instantiation_error.

ceiling(-0.5).
   Evaluates to the value 0.

truncate(-0.5).
   Evaluates to the value 0.

truncate(foo).
   type_error(number, foo).

float(7).
   Evaluates to the value 7.0.

float(7.3).
   Evaluates to a value
   approximately equal to 7.3.

float(5 / 3).
   Evaluates to the value 1.0.

float(N).
   instantiation_error.

float(foo).
   type_error(number, foo).

abs(7).
   Evaluates to the value 7.

abs(3-11).
   Evaluates to the value 8.

abs(3.2-11.0).
   Evaluates to a value
   approximately equal to 7.8000.

abs(N).
   instantiation_error.

abs(foo).
   type_error(number, foo).

current_prolog_flag(max_integer, MI),
  X is '+'(MI, 1).
   evaluation_error(int_overflow).

current_prolog_flag(max_integer, MI),
  X is '-'('+'(MI, 1), 1).
   evaluation_error(int_overflow).

current_prolog_flag(max_integer, MI),
  X is '-'(-1, MI).
   evaluation_error(int_overflow).

current_prolog_flag(max_integer, MI),
  X is '*'(MI, 2).
   evaluation_error(int_overflow).

current_prolog_flag(max_integer, MI),
  R is float(MI) * 2,
  X is floor(R).
   evaluation_error(int_overflow).
```

9.2 The format of other evaluable functor definitions

These subclauses define the format of the definitions of other evaluable functors.

9.2.1 Description

The description assumes that no error condition is satisfied, and is a mathematical description of the value of evaluating as an expression a term with that evaluable functor.

9.2.2 Template and modes

A specification for the type of the values when the arguments of the evaluable functor are evaluated as an expression, and the type of its value. The cases form a mutually exclusive set.

Notation for the structure and type of the arguments and value:

a) int-exp — integer expression,

b) integer — integer value,

c) float-exp — floating point expression,

d) float — floating point value

When appropriate, a "Template and modes" subclause includes a note that the evaluable functor is a predefined operator (see 6.3.4.4, table 7).

9.2.2.1 Examples

```
sin(float-exp) = float

'<<'(int-exp, int-exp) = integer
```

9.2.3 Errors

A list of the error conditions and associated error term when a term with that evaluable functor is evaluated as an expression.

NOTE — The effect of an error condition being satisfied is defined in clause 7.12.

9.2.4 Examples

An example is normally a term with that evaluable functor as principal functor, e.g.

```
functor(Argument, argument)
```

and its value or the error term that will occur.

Sometimes, an example will be a goal. In this case the format is the same as that for examples of built-in predicates (8.1.4).

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 has the value of `VX` raised to the power of `VY`. If `VX` and `VY` are both zero, the value 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
— `evaluation_error(undefined)`.

d) `VX` is zero and `VY` is negative
— `evaluation_error(undefined)`.

e) The magnitude of the `VX` raised to the power of `VY` is too large
— `evaluation_error(float_overflow)`.

f) The magnitude of the `VX` raised to the power of `VY` is too small and not zero
— `evaluation_error(underflow)`.

9.3.1.4 Examples

```
'**'(5, 3).
   Evaluates to a value
   approximately equal to 125.0000.

'**'(-5.0, 3).
   Evaluates to a value
   approximately equal to -125.0000.

'**'(5, -1).
   Evaluates to a value
   approximately equal to 0.2000.

'**'(77, N).
   instantiation_error.

'**'(foo, 2).
   type_error(number, foo).

'**'(5, 3.0).
   Evaluates to a value
   approximately equal to 125.0000.

'**'(0.0, 0).
   Evaluates to 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 has the value of 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) `X` is not a variable and `VX` is not a number
— `type_error(number, VX)`.

NOTE — The value of `sin(X)` has little or no significance if `VX` has a large magnitude.

9.3.2.4 Examples

```
sin(0.0).
   Evaluates to the value 0.0.

sin(N).
   instantiation_error.

sin(0).
   Evaluates to the value 0.0.

sin(foo).
   type_error(number, foo).

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.
```

9.3.3 cos/1

9.3.3.1 Description

`cos(X)` evaluates the expression `X` with value `VX` and has the value of 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) `X` is not a variable and `VX` is not a number
— `type_error(number, VX)`.

NOTE — The value of `cos(X)` has little or no significance if `VX` has a large magnitude.

9.3.3.4 Examples

```
cos(0.0).
   Evaluates to the value 1.0.

cos(N).
   instantiation_error.

cos(0).
   Evaluates to the value 1.0.

cos(foo).
   type_error(number, foo).

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.
```

9.3.4 atan/1

9.3.4.1 Description

`atan(X)` evaluates the expression `X` with value `VX` and has the value of the principal value of the arc tangent of `VX`, that is, the value 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) X is not a variable and VX is not a number
— type_error(number, VX).

9.3.4.4 Examples

```
atan(0.0).
   Evaluates to the value 0.0.

PI is atan(1.0) * 4.
   Succeeds, unifying PI with a value
   approximately equal to 3.14159.

atan(N).
   instantiation_error.

atan(0).
   Evaluates to the value 0.0.

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 has 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) X is not a variable and VX is not a number
— type_error(number, VX).

c) The magnitude of the exponential function of VX is too large
— evaluation_error(float_overflow).

d) The magnitude of the exponential function of VX is too small and not zero
— evaluation_error(underflow).

9.3.5.4 Examples

```
exp(0.0).
   Evaluates to the value 1.0.

exp(1.0).
   Evaluates to a value
   approximately equal to 2.7818.

exp(N).
   instantiation_error.

exp(0).
   Evaluates to the value 1.0.

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 has 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) X is not a variable and VX is not a number
— type_error(number, VX).

c) VX is zero or negative
— evaluation_error(undefined).

9.3.6.4 Examples

```
log(1.0).
   Evaluates to the value 0.0.

log(2.7818).
   Evaluates to a value
   approximately equal to 1.0000.

log(N).
   instantiation_error.

log(0).
   evaluation_error(undefined).

log(foo).
   type_error(number, foo).

log(0.0).
   evaluation_error(undefined).
```

9.3.7 sqrt/1

9.3.7.1 Description

`sqrt(X)` evaluates the expression `X` with value `VX` and has the value $\sqrt{(VX)}$.

9.3.7.2 Template and modes

```
sqrt(float-exp) = float
sqrt(int-exp) = float
```

9.3.7.3 Errors

a) `X` is a variable
— `instantiation_error`.

b) `X` is not a variable and `VX` is not a number
— `type_error(number, VX)`.

c) `VX` is negative
— `evaluation_error(undefined)`.

9.3.7.4 Examples

```
sqrt(0.0).
   Evaluates to the value 0.0.

sqrt(1).
   Evaluates to the value 1.0.

sqrt(1.21).
   Evaluates to a value
   approximately equal to 1.1000.

sqrt(N).
   instantiation_error.

sqrt( 1.0).
   evaluation_error(undefined).

sqrt(foo).
   type_error(number, foo).
```

9.4 Bitwise functors

The operands (3.121) and value of these evaluable functors are integers which are treated as a binary sequences of bits. The value is implementation defined when an operand or value is negative because the representation of a negative integer is implementation defined.

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 has the value of `VN` right-shifted `VS` bit positions.

The value 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 value 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) `N` is not a variable and `VN` is not an integer
— `type_error(integer, VN)`.

d) `S` is not a variable and `VS` is not an integer
— `type_error(integer, VS)`.

9.4.1.4 Examples

```
'>>'(16, 2).
   Evaluates to the value 4.

'>>'(19, 2).
   Evaluates to the value 4.

'>>'(-16, 2).
   Evaluates to an implementation defined value.

'>>'(77, N).
   instantiation_error.

'>>'(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 has the value of `VN` left-shifted `VS` bit positions, where the `VS` least significant bit positions of the result are zero.

The value 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) N is not a variable and VN is not an integer
— type_error(integer, VN).

d) S is not a variable and VS is not an integer
— type_error(integer, VS).

9.4.2.4 Examples

```
'<<'(16, 2).
   Evaluates to the value 64.

'<<'(19, 2).
   Evaluates to the value 76.

'<<'(-16, 2).
   Evaluates to an implementation defined value.

'<<'(77, N).
   instantiation_error.

'<<'(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 has the value such that each bit is set iff each of the corresponding bits in VB1 and VB2 is set.

The value 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) B1 is not a variable and VB1 is not an integer
— type_error(integer, VB1).

d) B2 is not a variable and VB2 is not an integer
— type_error(integer, VB2).

9.4.3.4 Examples

```
'/\\'(10, 12).
   Evaluates to the value 8.

/\(10, 12).
   Evaluates to the value 8.

'/\\'(17 * 256 + 125, 255).
   Evaluates to the value 125.

/\(-10, 12).
   Evaluates to an implementation defined value.

'/\\'(77, N).
   instantiation_error.

'/\\'(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 has the value such that each bit is set iff at least one of the corresponding bits in VB1 and VB2 is set.

The value 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) B1 is not a variable and VB1 is not an integer
— type_error(integer, VB1).

d) B2 is not a variable and VB2 is not an integer
— type_error(integer, VB2).

9.4.4.4 Examples

```
'\\/'(10, 12).
   Evaluates to the value 14.

\/(10, 12).
   Evaluates to the value 14.

'\\/'(125, 255).
   Evaluates to the value 255.

\/(-10, 12).
   Evaluates to an implementation defined value.

'\\/'(77, N).
   instantiation_error.

'\\/'(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 has the value such that each bit is set iff the corresponding bit in VB1 is not set.

The value 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) B1 is not a variable and VB1 is not an integer
— type_error(integer, VB1).

9.4.5.4 Examples

```
'\\'( '\\'(10)).
   Evaluates to the value 10.

\( \(10)).
   Evaluates to the value 10.

\(10).
   Evaluates to an implementation defined value.

'\\'(N).
   instantiation_error.

'\\'(2.5).
   type_error(integer, 2.5).
```

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 part of ISO/IEC 13211. Many features implicit in the clauses defining the informal semantics (7.7), control constructs (7.8), and built-in predicates (8) are explicitly described here.

NOTES

1 The following built-in predicates are not specified formally (aspects of the system environment have not been formalized) :

`char_conversion/2`, `close/2`, `current_char_conversion/2`, `current_op/3`, `flush_output/1`, `op/3`, `open/4`, `read_term/3`, `set_stream_position/2`, `stream_property/2`, `write_term/3`.

2 This formal specification does not provide description of the character sets and syntax of Prolog texts.

3 Unless explicitly stated, there is no semantics for undefined or implementation defined or dependent features.

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 clause 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, which can be read as `:-` of standard Prolog), " **,**" (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, arguments of **special-pred** (A.3.1).

— 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. In some cases an extension of the syntax will be allowed, such that clauses may have variables as predication. In that case it will be assumed that these variables are instantiated by goals only. The formal specification is written in such a way that proof trees which use such clauses can be built with such instances only.

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

There is no formal specification without comments expressed in the natural language. This specification respects this rule. However a strong discipline has been used in order to limit the need of long comprehensive comments.

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 and a goal, 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.

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**$(\cdots)$

does not contain any anonymous variable in its arguments. This is because such variable is usually intended to be existentially quantified inside the negation (it is implicitly universally quantified outside of the clause, hence inside 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.1.5 References

More information on the specification method may be found in the following documents:

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

A. Ed-Dbali, P. Deransart: Software Formal Specifications by Logic Programming: The example of Standard Prolog. LNAI 636, Springer Verlag, LPSS'92, September 1992.

P. Deransart, J. Maluszynski: A Grammatical View of Logic Programming, The MIT Press, 1993. (NSTO properties).

S. Renault, P. Deransart: Design of Formal Specifications by Logic Normal Programs: Merging Formal Text and Good Comments. Int. Journal of Software Engeneering and Knowledge Engineering, V4, 3 (1994) 369-390.

Information about a runnable specification (intended to be compatible with most existing Prolog processors) is available on request by E-Mail to:
AbdelAli.Ed-Dbali@lifo.univ-orleans.fr.

Some PhD theses have been devoted to aspects of standard Prolog, e.g. by Gilles Richard, Sophie Renault (validation), AbdelAli Ed-Dbali (runnable specification), Jean-Louis Bouquard, Bruno Dumant, Michel Téguia (NSTO properties).

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 of a standard conforming processor 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 built-in predicate are visited.

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

c) effects of the built-in predicates.

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 — The semantics of a program conforming to this part of ISO/IEC 13211.

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.

Any concept which is not defined in this informal description refers to concepts defined in the body document.

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 the singleton `true` 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.

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)).

It is important to observe that the algorithm works also if the clauses of the program have variables as predication in their body, if each variable is instantiated by a goal before it is selected.

This observation will be used to define some "bootstrapped" built-in predicates, where a variable may occur in the place of a predication (see for example the definition of the disjunction in the clause A.2.3.4). However this is not permitted in standard Prolog.

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 same order as the 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, different from the singleton `true` 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 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.

NOTE — 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. But they correspond to different semantics when built-in predicates are considered.

A.2.1.4 The visited search-tree

Given a predication-choice function, i.e. a search-tree, the computations of a database and goal are 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).
p(X, Y) :- s(X).

q(a) :- true.
q(b) :- true.
q(c) :- true.

r(b, b1) :- true.
r(c, c1) :- true.

s(d) :- true.
```

Figure A.1 shows the 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
```

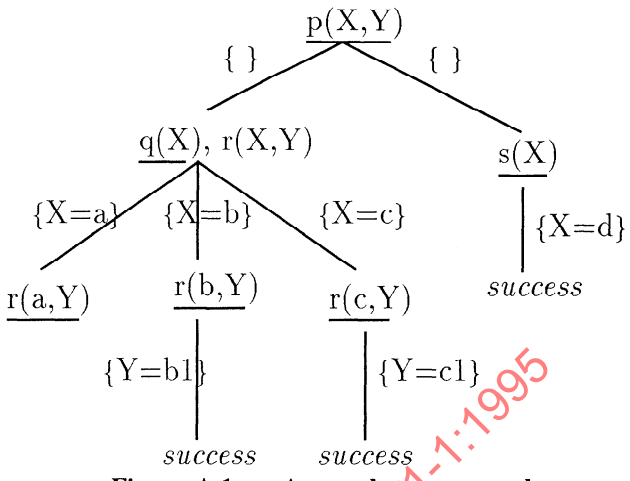

Figure A.1 — A search-tree example

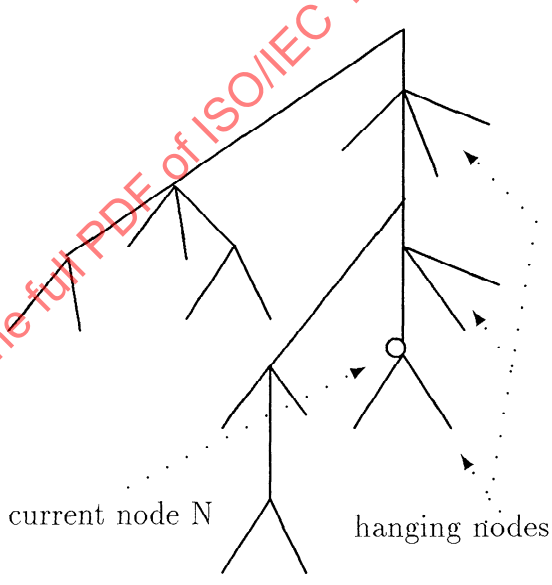

Figure A.2 — A visited search-tree

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.

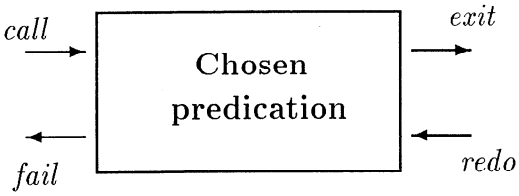

Figure A.3 — Byrd's trace model

Observe now that there is no way to visit the search-tree beyond the first (i.e. left-most) infinite branch with the standard 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 correspond to **backtracking**.

A.2.1.8 An analogy with Byrd's box model

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

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

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

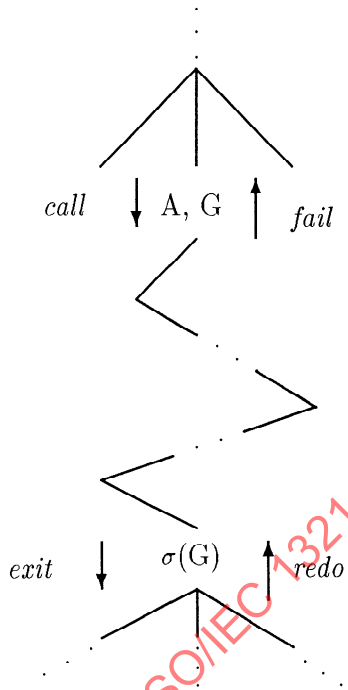

Figure A.4 — Byrd's model: a search-tree point of view

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 part of ISO/IEC 13211 "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 exhaustive 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.

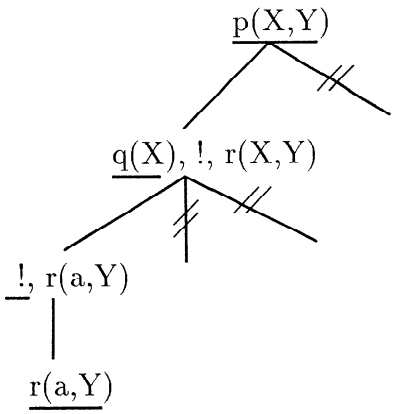

Figure A.5 — A search-tree example showing the effect of cut

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) :- true.
q(b) :- true.
q(c) :- true.

r(b, b1) :- true.
r(c, c1) :- true.

s(d) :- true.
```

Figure A.5 shows that the search-tree corresponding to the goal `p(X, Y)` has one failed branch only.

NOTE — Cuts sometimes increase the number of success branches. This may be understood by the use of the cut to specify negation by failure (see the bootstrapped (\+)/2 definition). The composition of two negations may increase the number of successes.

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

A.2.3 Search-tree for kernel Prolog

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

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

In **kernel Prolog** (as in this part of ISO/IEC 13211) 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 part of ISO/IEC 13211, 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:

If the principal functor of G_1 is not (->)/2 then 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, and a variable is not allowed as a predication in this part of ISO/IEC 13211):

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

if the principal functor of G_1 is not `(->)/2`.

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 (;)/2 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 (but after the head). 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.2.3.7).

In the formal semantic the scope of a cut is represented by flagging cut (`!(flag)`) where the `flag` denotes the parent node of the node in which the instance of the clause in which the cut occurs has been used. When a cut flagged by N is chosen all the ancestor choice points until N (inclusive) are made deterministic.

NOTE — Due to the well-formedness of goals, there is no way to execute in this formal specification an unflagged cut. Hence if a cut occurs inside the arguments of some disjunction, the arguments of the bootstrapped definition contain this already flagged cut.

A.2.3.6 Call

`call` is a control construct which permits the use of a variable as a predication and limits 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 body (see **is-an-extended-body** A.3.1), the scope of a cut in this goal is limited to this goal. It is sometime said that call is *not transparent* or *opaque* to cut, otherwise it would be *transparent* and its scope would extend to all predications to its left in the body of the clause and its parent.

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.

Notice finally that this part of ISO/IEC 13211 does not define how the computations continue after a success branch has been obtained, i.e. how the visit of the search-tree is continued, nor how the answer substitutions are displayed.

A.2.3.7 If-then

The **conditional construct** `'->'(Cond, Then)` is defined out of the context of a disjunction (i.e. not the first argument of (;)/2) as follows:

if `Cond` succeeds then cuts the choice points issued from `Cond` only and executes `Then`. `Cond` is opaque to cut.

if `Cond` fails then fails.

It can be defined by the following pseudo-clause:

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

`'->'(Cond, Then)` not being the first argument of a predication `(;)/2`.

A.2.3.8 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 cuts the choice points issued from `Cond` only and executes `Then` ignoring `Else`.

If `Cond` fails then executes `Else`.

It could be defined by the following pseudo-clause:

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

A.2.4 Database and database update view

In this part of ISO/IEC 13211 the relationships between clauses (issued from Prolog terms in a Prolog text) stored in the database and a term defining a clause is defined by means of "conversion" (7.6). Its purpose is to define also what is a "well-formed goal". In this formal definition an abstract syntax is assumed and given for the database, clauses and terms, which in particular defines what is a "well-formed goal". This Abstract database contains the user-defined procedures only, but in contains implicitly all the control constructs and built-in predicates.

It is also 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) with principal functor `:-/2`. Moreover it is assumed that a predicate is defined only once.

NOTE — A predicate is thus uniquely determined by its predicate indicator, i.e. its name and arity.

The semantics described so far assumes that the database remains unchanged during the the execution of a goal. Standard Prolog contains five built-in predicates which may modify the database: asserta/1, assertz/1, retract/1, abolish/1, 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, retract/1 removes the first clause which unifies with the argument and abolish/1 which removes completely a procedure. retract/1 is resatisfiable and removes 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`.

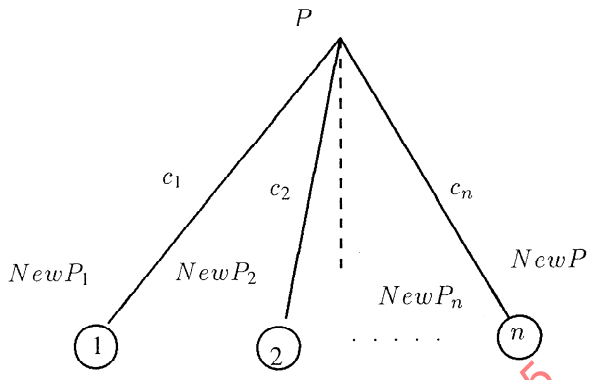

Figure A.6 — Standard database update view

Notice that if all clauses of a predicate have been removed retract/1 just fails and all informations about this predicate remain, except that the packet of clauses is empty. Only abolish/1 leaves the database as this predicate had never existed.

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

If there is no modification of the database all the $NewP_i$ and P are the same and all the children are visited and expanded. 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 update the 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 built-in predicate. The mechanism of the exception handling can be informally described as follows.

The built-in predicate 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**).

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 predicate 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 built-in predicate errors. This case is exactly as though a user error is raised by calling `throw(`*Error_term, impl_def*`)`.

If the user for any reason omits to specify an appropriate catcher, the result is a system error (see 7.8.10.3b). However in the formal specification there is a catcher at the root (see A.4.1.1 and A.4.1.43) in order to propagate eventual error to previous steps of execution (errors occurring during the execution of findall/3 for example).

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 built-in predicate 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 part of ISO/IEC 13211 it is required that an environment is defined at least by the values of the flags

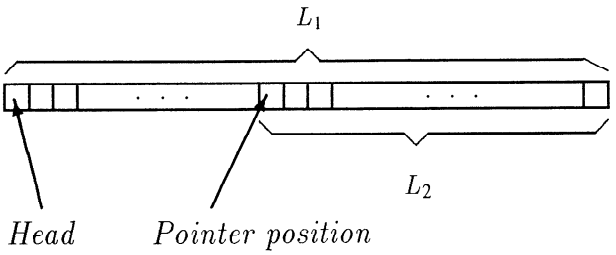

Figure A.7 — $L_1 - L_2$: Difference list of characters

and the standard 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 text streams respectively. It is denoted *env(PF, IF, OF, IFL, OFL)* (**D-is-an-environment** A.3.7).

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

To denote a stream, we use *stream(N, L1 - L2)*, where N is the abstract name of the stream (not represented in the formal specification) 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 or implementation defined 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.4.1.41) which is implementation dependent in the comparison of variables, or renaming.

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

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) :- true.
p(b) :- true.

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

and the goal: `goal`.

Its semantics contains all the partially visited search-trees depicted in Figure A.8 (→ denotes the node N to be executed and ← 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) visited 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)

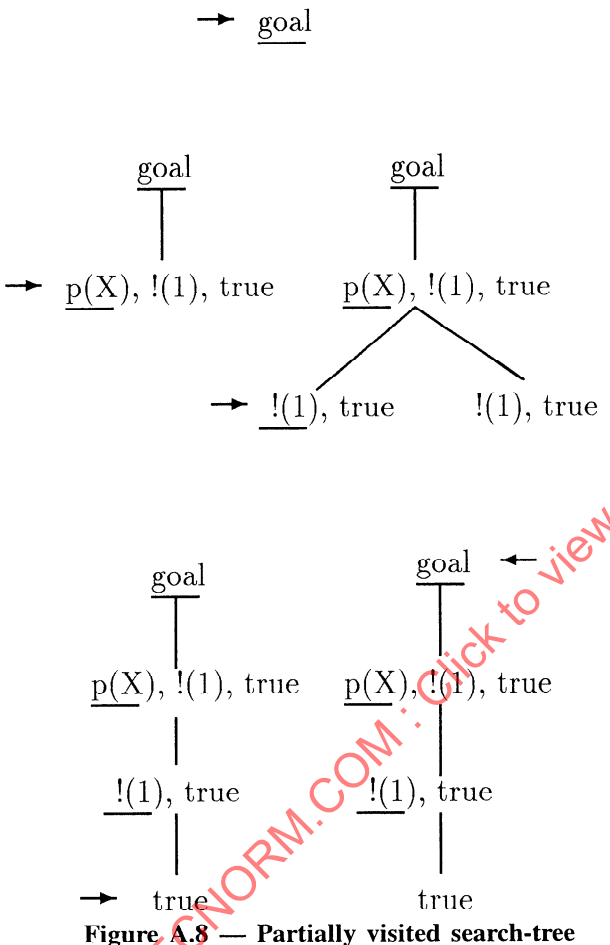

Figure A.8 — Partially visited search-tree

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 in proof-trees rooted by **treatment**(*F1, N, F2*) N is a hanging node of $F1$ on the slice, but N may have children in $F2$.

The different clauses of **treatment** together with the clauses of **treatbip** deal with success, built-in predicate not in error, error case, special predicate and failure. All possible cases (depending of the kind of built-in predicates called "substitution- or boot-bip") are covered, ensuring the completeness of the formal definition for all well-formed programs and goals built with user-defined predicates, control constructs and built-in predicates.

The addition of a new node is made by **expand**(A.4.1.18) in which **buildchild** (A.4.1.25) 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 built-in predicates like bagof/3 or atom_concat/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.4.1.32) 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 specified. 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.4.1.37) 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*)) ⇐
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 built-in predicate 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 `(:-)/2` 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-head(*H*),
  D-is-a-body(*B*).

D-is-a-head(*H*) ⇐
  D-is-a-predication(*H*).

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.

The syntax of a clause is now extended as follows:

D-is-an-extended-clause(*C*) — *iff* *C* is a clause extended by other data structures.

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

D-is-an-extended-body(*B*) — *iff* *B* is a body extended by other data structures.

D-is-an-extended-body(*func(',', G1.G2.nil)*) ⇐
  D-is-an-extended-body(*G1*),
  D-is-an-extended-body(*G2*).

D-is-an-extended-body(*func(';', G1.G2.nil)*) ⇐
  D-is-an-extended-body(*G1*),
  D-is-an-extended-body(*G2*).

D-is-an-extended-body(*func('->', G1.G2.nil)*) ⇐
  D-is-a-term(*G1*),
  D-is-an-extended-body(*G2*).

D-is-an-extended-body(*B*) ⇐
D-is-an-extended-predication(*B*).

D-is-an-extended-body(*G*) — *iff* *G* is a predication extended by other data structures.

D-is-an-extended-predication(*G*) ⇐
D-is-a-predication(*G*).

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

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

D-is-an-extended-predication(*X*) ⇐
L-var(*X*).

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

D-is-a-special-pred(*special-pred(undefined-action, E.nil)*).

D-is-a-special-pred(*special-pred(forward-error, E.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(undefined-action, nil)*, *special-pred(forward-error, 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-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(*X*) ⇐
D-is-a-number(*X*).

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

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

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

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

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 (static procedures are private or public (7.5.3)).

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

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

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

D-is-a-callable-term(*T*) — *iff* *T* is a callable term as it is defined in 3.24.

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, G, V*) — *iff* *L* is an abstract list of terms and *G* is a goal and *V* a term which contains all the variables (each one occurs exactly once) in *G* not occurring in *L*.

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

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

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

L-syntax-error-in-code-list(*List*) — *iff* *List* is a list of codes but not parsable as a number.

L-syntax-error-in-char-list(*List*) — *iff* *List* is a list of characters but not parsable as a number.

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

D-is-a-goal(*G*) ⇐
  D-is-a-body(*G*).

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*).

L-is-a-character-code(*I*) — *iff* *I* is an integer such that there exists a character C whose *character_code* 7.1.2.2 value is *I*.

L-is-an-in-character-code(*I*) — *iff* *I* is an integer such that there exists a character C whose *character_code* 7.1.2.2 value is *I* *or* *I* is the integer -1.

D-is-a-byte(*C*) — *iff* *C* is an integer between 0 and 255 as defined in 7.1.2.1.

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

D-is-an-in-byte(*C*) — *iff* *C* is an integer between -1 and 255.

D-is-an-in-byte(*B*) ⇐
  D-is-a-byte(*B*).

D-is-an-in-byte(*func(-1, nil)*).

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-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 corresponding to a well formed clause **then** if its principal functor is `(:-)/2`, 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*) ⇐
  D-name(*A*, *Name*),
  D-arity(*A*, *Arity*),
  not **D-equal**(*Name/Arity*, `:-/2`),
  D-fact-to-clause(*A*, *C*).

D-term-to-body(*T*, *B*) — **if** T is a term corresponding to a well formed body **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 `(:-)/2`, 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*)) ⇐
  D-name(*B*, *Name*),
  D-arity(*B*, *Arity*),
  not **D-equal**(*Name/Arity*, `:-/2`).

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;

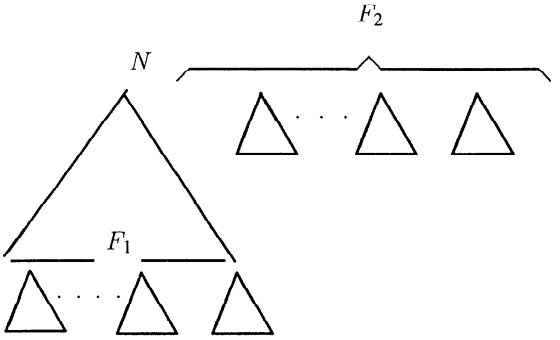

Figure A.9 — The non-empty forest: $for(N, F_1, F_2)$

— 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 current flags and all available streams for the current computation;

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

— L is a list of nodes (dewey numbers) containing the active ancestor nodes at this step of resolution (i.e. the `catch` goals which could be chosen if `throw` is called). The 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, P*) ⇐
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(Nl, F1, F2)* **is** Nl ⇐
D-label of node N **in** $F1$ **is** Nl.

D-label of node N **in** *for(Nl, 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-var-in-list(*L*) — **if** L is a list **then** it contains a variable.

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

D-var-in-list(*func(., X.L.nil)*) ⇐
  not **L-var**(*X*),
  D-var-in-list(*L*).

NOTE — References: **L-var** A.3.1

D-bad-element-in-char-list(*L,E*) — **if** L is a non empty list **then** an element E of L is neither a variable nor a one-char atom.

D-bad-element-in-char-list(*func(., E.L.nil), E*) ⇐
  not **L-var**(*E*),
  not **D-is-a-char**(*E*).

D-bad-element-in-char-list(*func(., X.L.nil), E*) ⇐
  L-var(*X*),
  D-bad-element-in-char-list(*L, E*).

D-bad-element-in-char-list(*func(., X.L.nil), E*) ⇐
  D-is-a-char(*X*),
  D-bad-element-in-char-list(*L, E*).

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

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

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*).

L-concat-list(*A, L*) — **if** A is an atom **then** L is the list of couples (A_1, A_2) such that the concatenation of A_1 and A_2 gives A.

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 a list **then** A is an element of L and $L1$ is L where this element is deleted.

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

D-one-delete(*func(., A.L.nil), B, func(., A.L1.nil)*) ⇐
  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 code (see `char_code/2` built-in predicate).

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` built-in predicate).

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` built-in predicate).

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` built-in predicate).

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` built-in predicate).

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 unifiers (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 unifiers (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 $T1$ (3.95)).

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, T, X, Y*) — *iff* F is a search tree, T a term and X and Y are any-terms such that Y is identical to X except all its variables which do not occur in T 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).

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,

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* ∈ {`bounded`,
`max_integer`,
`min_integer`,
`integer_rounding_function`,
`char_conversion`,
`debug`,
`max_arity`,
`unknown`,
`double_quotes`}

D-is-a-modifiable-flag(*Flag*) — *iff* $Flag$ is a modifiable flag (its value can be updated by `set_prolog_flag/2` built-in predicate).

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

with *flag-name* ∈ {`char_conversion`,
`debug`,
`unknown`,
`double_quotes`}

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,_,_func([],nil).nil)*).

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)*),
not **D-equal**(*V, func([], nil)*),
D-transform-list(*V1,V*),
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 **??**.

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

L-binary-stream(*BS*) — *iff* BS is a binary stream.

L-text-stream(*TS*) — *iff* TS is a text stream.

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*).

D-is-an-in-char(*Char*) — *iff* C is the abstract representation of a concrete character or of `end_of_file`.

D-is-an-in-char(*Char*) ⇐
D-is-a-char(*Char*).

D-is-an-in-char(*func(`end_of_file`, nil)*).

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 3.167.

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(float, _.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-byte-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(at_end_of_stream, nil),
func(at_end_of_stream, _.nil),
func(get_char, _.nil),
func(get_char, _._.nil),
func(get_code, _.nil),
func(get_code, _._.nil),
func(peek_char, _.nil),
func(peek_char, _._.nil),
func(peek_code, _.nil),
func(peek_code, _._.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-byte-input-output-bip(B).

with $B \in$ {func(get_byte, _.nil),
func(get_byte, _._.nil),
func(peek_byte, _.nil),
func(peek_byte, _._.nil),
func(put_byte, _.nil),
func(put_byte, _._.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(\+, _.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.

D-boot-bip(*B*)

with $B \in$ {func(->, _._.nil),
func(\+, _.nil),
func(number, _.nil),
func(is, _._.nil),
func(=:=, _._.nil),
func(=\=, _._.nil),
func(<, _._.nil),
func(=<, _._.nil),
func(>, _._.nil),
func(>=, _._.nil),
func(once, _.nil),
func(setof, _._._.nil),
func(get_char , _.nil),
func(get_code , _.nil),
func(get_byte , _.nil),
func(peek_char , _.nil),
func(peek_code , _.nil),
func(peek_byte , _.nil),
func(put_char, _.nil),
func(put_code, _.nil),
func(put_byte, _.nil),
func(at_end_of_stream, 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)}

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-backtrack-bip(*B*) — **if** *B* is the built-in predicate `atom_concat/3` **then** its third argument is ground. **or** *B* is the built-in predicate `current_prolog_flag/2`.

D-is-a-backtrack-bip(*func*(`atom_concat`, *A1.A2.A3.nil*)) ⇐
D-is-an-atom(*A3*).

D-is-a-backtrack-bip(*func*(`current_prolog_flag`, _._*nil*)).

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

D-is-a-subst-bip(*B*) — **if** *B* is a predication **then** it is the predication of a class of built-in predicates which does 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-name(*B*, *Func*),
D-arity(*B*, *Arity*),
not **D-equal**(*Func*, *func*(`atom_concat`, *nil*)),
not **D-equal**(*Arity*, *func*(3, *nil*)).

D-is-an-evaluable-expression(*Exp*) — **if** *Exp* is an expression whose principal functor is an evaluable one.

D-is-an-evaluable-expression(*Func*)

with $Func \in$ {func(+, _._.nil),
func(-, _._.nil),
func(*, _._.nil),
func(//, _._.nil),
func(/, _._.nil),
func(rem, _._.nil),
func(mod, _._.nil),
func(**, _._.nil),
func(>>, _._.nil),
func(<<, _._.nil),
func(/\, _._.nil),
func(\/, _._.nil),
func(-, _.nil),
func(abs, _.nil),
func(sign, _.nil),
func(float_integer_part, _.nil),
func(float_fractional_part, _.nil),
func(float, _.nil),
func(floor, _.nil),
func(truncate, _.nil),
func(round, _.nil),
func(ceiling, _.nil),
func(sin, _.nil),
func(cos, _.nil),
func(atan, _.nil),
func(exp, _.nil),
func(log, _.nil),
func(sqrt, _.nil),
func(\, _.nil)}

D-packet(*DB*, *Env*, *A*, *Q*) — **if** *DB* is a database and *Env* an environment and *A* is a predication **then**

— *Q* is the list of clauses defining the procedure corresponding to *A*;

— or all clauses of *DB* if *A* corresponds to the following re-executable built-in predicates: `clause/2`, `current_predicate/1`, `retract/1`;

— or a list of pairs of atoms (A_1, A_2) such that the concatenation of A_1 and A_2 gives the 3rd argument of *A* (if *A* corresponds to the re-executable `atom_concat/3` built-in predicate);

— or a list of the prolog flags in *Env* (if *A* corresponds to the re-executable built-in predicate `current_prolog_flag/2`).

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*),
not **D-is-a-backtrack-bip**(*A*).

D-packet(*_, _, func*(`atom_concat`, *A1.A2.A3.nil), L*) ⇐
D-is-an-atom(*A3*),
L-concat-list(*A3, L*).

D-packet(*_, Env, func*(`current_prolog_flag`, *_._.nil), PF*) ⇐
D-equal(*Env, env(PF, _, _, _, _)*).

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

with $SP \in$ {special-pred(inactivate, _.nil),
special-pred(undefined-action, _.nil),
special-pred(forward-error, _.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, **D-equal** A.3.1, **corresponding-pred-definition** A.4.1.52, **exist-corresponding-pred-definition** A.4.1.53, **L-concat-list** A.3.4

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

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

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

NOTE — References: **streamname** A.4.1.61, **D-member** A.3.4

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

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

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

NOTE — References: **streamname** A.4.1.61, **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, func(`catch`, *G.X.special-pred(undefined-action, X.nil).nil),*
P, nil, E, empsubs, nil, partial)),
buildforest(*for(N, vid, vid), nil, F*),
L-var(*X*),
L-not-occur-in(*X, G*).

NOTES

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

2 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 *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*),
  is-a-catch-node(*K, F*),
  D-last-child(*K, F, M*),
  not **completely-visited-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*),
  D-choice of node *N* **in** *F1* **is** *Q*,
  not **D-equal**(*Q, nil*),
  expand(*F1, N, 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*),
  D-choice of node *N* **in** *F1* **is** *nil*,
  erasepack(*F1, N, F2*).

NOTE — References: **success-node** A.4.1.16, **erasepack** A.4.1.24, **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.4.1.32, **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.

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*(*undefined-action, E.nil*), *F1*).

treat-special-pred(*F1, N, special-pred*(*forward-error, E.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, Exp.V.nil), F2*) ⇐
not **error**(*F1, special-pred(value, Exp.V.nil)*), **expand**(*F1, N, F2*).

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

in-error(_, *special-pred(value, Exp.V.nil), instantiation-error*) ⇐
L-var(*Exp*).

in-error(_, *special-pred(value, Exp.V.nil), type-error(evaluable, func(/, Func.Arity.nil))*) ⇐
not **L-var**(*Exp*),
not **D-is-a-number**(*Exp*),
not **D-is-an-evaluable-expression**(*Exp*),
D-name(*Exp, Func*),
D-arity(*Exp, Arity*).

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

treat-special-pred(*F1, N, special-pred(operation-value, Exp.V.nil), F2*) ⇐
not **error**(*F1, special-pred(operation-value, Exp.V.nil)*),
L-value(*Exp, Value*),
D-label of node *N* **in** *F1* **is** *Nl*,
D-equal(*Nl, nd(I, G, P, Q, E, _, L, _)*),
D-number-of-child(*I, F1, J*),
erase(G, G1),
D-equal(*G2, (V = Value ', ' G1)*),
predication-choice(G2, A1),
D-packet(*P, E, A1, Q1*),
D-equal(*Nl1, nd(J.I, G2, P, Q1, E, empsubs, L, partial)*),
addchild(F1, Nl, Nl1, nil, F2).

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

in-error(_, *special-pred(operation-value, Exp.V.nil), T*) ⇐
L-error-in-expression(*Exp, T*).

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, Comp.nil), F2*) ⇐
L-arithmetic-comparison(*Comp*),
D-label of node *N* **in** *F1* **is** *Nl*,
D-equal(*Nl, nd(N, G, P, _, E, S, L, _)*),
final-resolution-step(*G, empsubs, P, E, G1, Q*),
D-equal(*Nl1, nd(zero.N, G1, P, Q, E, empsubs, L, partial)*),
addchild(*F1, Nl, Nl1, nil, F2*).

treat-special-pred(*F1, N, special-pred(simple-comparison, Comp.nil), F2*) ⇐
not **L-arithmetic-comparison**(*Comp*),
erasepack(*F1, N, F2*).

NOTE — References: **treat-inactivate** A.4.1.64, **expand** A.4.1.18, **error** A.4.1.14, **in-error** A.4.1.15, **L-value** A.3.6, **treat-bip** A.4.1.32, **L-sorted** A.3.4, **L-error-in-expression** A.3.6, **L-arithmetic-comparison** A.3.6

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

if *F*1 is a PVST up to node *N* and the chosen predication in the goal of *N* is a user defined predicate with non empty list of choice or a boot-strapped built-in predicate **then** *F*2 is the new PVST obtained after one step of resolution (So the node *N* in *F*2 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*)),
fail-or-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, E.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, Comp.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.4.1.23, **fail-or-undefined-pred-treatment** A.4.1.19, **add-value-child** A.4.1.21, **add-compare-child** A.4.1.22, **buildchild** A.4.1.25, **addchild** A.4.1.26

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

if *F1* is a PVST up to node *N*, and *A* is the predicate `fail` or is an undefined predication **then** *F2* is the extension of *F1* after execution of `fail` or according to the value of the flag `unknown` (7.11.2.4).

fail-or-undefined-pred-treatment(*F1, N, func*(`fail`, *nil*), *F2*) ⇐
  erasepack(*F1, N, F2*).

fail-or-undefined-pred-treatment(*F1, N, A, F2*) ⇐
  not **D-equal**(*A, func*(`fail`, *nil*)),
  D-name(*A, Func*),
  D-arity(*A, Arity*),
  D-database of node *N* **in** *F1* **is** *DB*,
  exist-corresponding-pred-definition(*func*(/, *Func.Arity.nil*), *DB*),
  erasepack(*F1, N, F2*).

fail-or-undefined-pred-treatment(*F1, N, A, F2*) ⇐
  not **D-equal**(*A, func*(`fail`, *nil*)),
  D-name(*A, Func*),
  D-arity(*A, Arity*),
  D-database of node *N* **in** *F1* **is** *DB*,
  not **exist-corresponding-pred-definition**(*func*(/, *Func.Arity.nil*), *DB*),
  undefined-pred-treatment(*F1, N, A, F2*).

NOTE — References: **erasepack** A.4.1.24, **D-equal** A.3.1, **D-name** A.3.1, **D-arity** A.3.1, **D-database of node** _ **in** _ **is** _ A.3.3.4, **exist-corresponding-pred-definition** A.4.1.53, **undefined-pred-treatment** A.4.1.20

A.4.1.20 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 `unknown` (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*(`unknown`, *nil*), *func*(*fail, nil*), *PF*, _, _, _),
  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*(`unknown`, *nil*), *func*(*error, nil*), *PF*, _, _, _),
  D-name(*A, Func*),
  D-arity(*A, Arity*),
  treat-bip(*F1, N, func*(`throw`, *existence_error*(*procedure, func*(/, *Func.Arity.nil*).*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*(`unknown`, *nil*), *func*(*warning, nil*), *PF*, _, _, _),
  D-label-of-node-in-is(*N,F1,Nl*),
  D-equal(*Nl, nd*(*N, G, P*, _, *E, S, L*, _)),
  erase(*G, G2*),
  D-equal(*G1*, (`write`(*output_warning_stream,unknown_procedure_message*) `,` `fail` `,` *G2*)),
  predication-choice(*G1, A1*),
  D-packet(*P, E, A1, Q*),
  D-equal(*Nl1, nd*(*zero.N, G1, P, Q, E, empsubs, L, partial*)),
  addchild(*F1, Nl, Nl1, nil, F2*).

NOTE — References: **D-environment of node** _ **in** _ **is** _ A.3.3.4, **D-equal** A.3.1, **corresponding-flag-and-value** A.4.1.75, **D-name** A.3.1, **D-arity** A.3.1, **erasepack** A.4.1.24, **treat-bip** A.4.1.32

A.4.1.21 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*)),
  D-is-a-number(*Num*),
  erase(*G, G1*),
  D-equal(*G2*, (`number`(*Num*) `,` *Num* = *V* `,` *G1*)),
  predication-choice(*G2, A1*),
  D-packet(*P, E, 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, Exp.nil*).*V.nil*)),
  erase(*G, G1*),
  L-var(*V1*),
  L-rename(*F1,V1,V11*),
  D-equal(*G2*, (*special-pred*(*value, Exp.V11.nil*) `,` *special-pred*(*operation-value, func*(*Op, V11.nil*).*V.nil*) `,` *G1*)),
  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*, _)),