



BSI Standards Publication

Information technology — Programming languages — Prolog

Part 1: General core (ISO/IEC 13211-1: 1995)

National foreword

This British Standard is the UK implementation of ISO/IEC 13211-1:1995, incorporating corrigenda November 2007, February 2012 and July 2017.

ISO/IEC technical corrigenda November 2007, February 2012 and July 2017 have been appended to the end of this document in their entirety.

The UK participation in its preparation was entrusted to Technical Committee IST/5/-/17, Prolog.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2017
Published by BSI Standards Limited 2017

ISBN 978 0 580 51091 5

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 15 October 1995.

Amendments/corrigenda issued since publication

Date	Text affected
31 March 2012	Implementation of ISO/IEC corrigenda November 2007 and February 2012
31 December 2017	Implementation of ISO/IEC corrigenda July 2017

**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	Page
Foreword	viii
Introduction	ix
1 Scope	1
1.1 Notes	1
2 Normative references	1
3 Definitions	2
4 Symbols and abbreviations	10
4.1 Notation	10
4.1.1 Basic mathematical types	10
4.1.2 Mathematical and set operators	10
4.1.3 Other functions	10
4.2 Abstract data type: stack	11
4.3 Abstract data type: mapping	11
5 Compliance	11
5.1 Prolog processor	11
5.2 Prolog text	12
5.3 Prolog goal	12
5.4 Documentation	12
5.5 Extensions	12
5.5.1 Syntax	12
5.5.2 Predefined operators	12
5.5.3 Character-conversion mapping	12
5.5.4 Types	12
5.5.5 Directives	13
5.5.6 Side effects	13
5.5.7 Control constructs	13
5.5.8 Flags	13
5.5.9 Built-in predicates	13

© ISO/IEC 1995

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case Postale 56 • CH-1211 Genève 20 • Switzerland
Printed in Switzerland

5.5.10	Evaluable functors	13
5.5.11	Reserved atoms	13
6	Syntax	13
6.1	Notation	13
6.1.1	Backus Naur Form	13
6.1.2	Abstract term syntax	14
6.2	Prolog text and data	15
6.2.1	Prolog text	15
6.2.2	Prolog data	15
6.3	Terms	15
6.3.1	Atomic terms	16
6.3.2	Variables	16
6.3.3	Compound terms – functional notation	16
6.3.4	Compound terms – operator notation	17
6.3.5	Compound terms – list notation	19
6.3.6	Compound terms – curly bracketed term	20
6.3.7	Terms – double quoted list notation	20
6.4	Tokens	20
6.4.1	Layout text	21
6.4.2	Names	21
6.4.3	Variables	23
6.4.4	Integer numbers	23
6.4.5	Floating point numbers	23
6.4.6	Double quoted lists	24
6.4.7	Back quoted strings	24
6.4.8	Other tokens	24
6.5	Processor character set	24
6.5.1	Graphic characters	25
6.5.2	Alphanumeric characters	25
6.5.3	Solo characters	25
6.5.4	Layout characters	25
6.5.5	Meta characters	26
6.6	Collating sequence	26
7	Language concepts and semantics	26
7.1	Types	27
7.1.1	Variable	27
7.1.2	Integer	27
7.1.3	Floating point	28
7.1.4	Atom	29
7.1.5	Compound term	29
7.1.6	Related terms	29
7.2	Term order	30
7.2.1	Variable	31
7.2.2	Floating point	31
7.2.3	Integer	31
7.2.4	Atom	31
7.2.5	Compound	31
7.3	Unification	31
7.3.1	The mathematical definition	31
7.3.2	Herbrand algorithm	31
7.3.3	Subject to occurs-check (<i>STO</i>) and not subject to occurs-check (<i>NSTO</i>)	33
7.3.4	Normal unification in Prolog	33
7.4	Prolog text	33
7.4.1	Undefined features	34

7.4.2	Directives	34
7.4.3	Clauses	35
7.5	Database	36
7.5.1	Preparing a Prolog text for execution	36
7.5.2	Static and dynamic procedures	36
7.5.3	Private and public procedures	36
7.5.4	A logical database update	37
7.6	Converting a term to a clause, and a clause to a term	37
7.6.1	Converting a term to the head of a clause	37
7.6.2	Converting a term to the body of a clause	37
7.6.3	Converting the head of a clause to a term	37
7.6.4	Converting the body of a clause to a term	38
7.7	Executing a Prolog goal	38
7.7.1	Execution	38
7.7.2	Data types for the execution model	38
7.7.3	Initialization	39
7.7.4	A goal succeeds	39
7.7.5	A goal fails	39
7.7.6	Re-executing a goal	39
7.7.7	Selecting a clause for execution	40
7.7.8	Backtracking	40
7.7.9	Side effects	40
7.7.10	Executing a user-defined procedure	40
7.7.11	Executing a user-defined procedure with no more clauses	42
7.7.12	Executing a built-in predicate	42
7.8	Control constructs	43
7.8.1	true/0	43
7.8.2	fail/0	43
7.8.3	call/1	44
7.8.4	!/0 – cut	45
7.8.5	(';')/2 – conjunction	47
7.8.6	(;)/2 – disjunction	47
7.8.7	(->)/2 – if-then	49
7.8.8	(;)/2 – if-then-else	50
7.8.9	catch/3	51
7.8.10	throw/1	53
7.9	Evaluating an expression	54
7.9.1	Description	54
7.9.2	Errors	54
7.10	Input/output	54
7.10.1	Sources and sinks	54
7.10.2	Streams	55
7.10.3	Read-options list	58
7.10.4	Write-options list	58
7.10.5	Writing a term	59
7.11	Flags	60
7.11.1	Flags defining integer type <i>I</i>	60
7.11.2	Other flags	61
7.12	Errors	61
7.12.1	The effect of an error	62
7.12.2	Error classification	62
8	Built-in predicates	63
8.1	The format of built-in predicate definitions	63
8.1.1	Description	63
8.1.2	Template and modes	64
8.1.3	Errors	64

8.1.4	Examples	65
8.1.5	Bootstrapped built-in predicates	65
8.2	Term unification	65
8.2.1	(=)/2 – Prolog unify	65
8.2.2	unify_with_occurs_check/2 – unify	66
8.2.3	(\=)/2 – not Prolog unifiable	67
8.3	Type testing	67
8.3.1	var/1	67
8.3.2	atom/1	68
8.3.3	integer/1	68
8.3.4	float/1	68
8.3.5	atomic/1	68
8.3.6	compound/1	69
8.3.7	nonvar/1	69
8.3.8	number/1	69
8.4	Term comparison	70
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	70
8.5	Term creation and decomposition	71
8.5.1	functor/3	71
8.5.2	arg/3	72
8.5.3	(=..)/2 – univ	72
8.5.4	copy_term/2	73
8.6	Arithmetic evaluation	74
8.6.1	(is)/2 – evaluate expression	74
8.7	Arithmetic comparison	74
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	76
8.8	Clause retrieval and information	77
8.8.1	clause/2	77
8.8.2	current_predicate/1	78
8.9	Clause creation and destruction	78
8.9.1	asserta/1	78
8.9.2	assertz/1	79
8.9.3	retract/1	80
8.9.4	abolish/1	81
8.10	All solutions	82
8.10.1	findall/3	82
8.10.2	bagof/3	83
8.10.3	setof/3	84
8.11	Stream selection and control	86
8.11.1	current_input/1	86
8.11.2	current_output/1	86
8.11.3	set_input/1	87
8.11.4	set_output/1	87
8.11.5	open/4, open/3	87
8.11.6	close/2, close/1	88
8.11.7	flush_output/1, flush_output/0	89
8.11.8	stream_property/2, at_end_of_stream/0, at_end_of_stream/1	89
8.11.9	set_stream_position/2	90
8.12	Character input/output	91
8.12.1	get_char/2, get_char/1, get_code/1, get_code/2	91
8.12.2	peek_char/2, peek_char/1, peek_code/1, peek_code/2	92

8.12.3	put_char/2, put_char/1, put_code/1, put_code/2, nl/0, nl/1	94
8.13	Byte input/output	95
8.13.1	get_byte/2, get_byte/1	95
8.13.2	peek_byte/2, peek_byte/1	96
8.13.3	put_byte/2, put_byte/1	97
8.14	Term input/output	98
8.14.1	read_term/3, read_term/2, read/1, read/2	98
8.14.2	write_term/3, write_term/2, write/1, write/2, writeq/1, writeq/2, write_canonical/1, write_canonical/2	99
8.14.3	op/3	101
8.14.4	current_op/3	102
8.14.5	char_conversion/2	103
8.14.6	current_char_conversion/2	103
8.15	Logic and control	104
8.15.1	(\+)/1 – not provable	104
8.15.2	once/1	105
8.15.3	repeat/0	105
8.16	Atomic term processing	105
8.16.1	atom_length/2	106
8.16.2	atom_concat/3	106
8.16.3	sub_atom/5	107
8.16.4	atom_chars/2	108
8.16.5	atom_codes/2	109
8.16.6	char_code/2	109
8.16.7	number_chars/2	110
8.16.8	number_codes/2	111
8.17	Implementation defined hooks	112
8.17.1	set_prolog_flag/2	112
8.17.2	current_prolog_flag/2	112
8.17.3	halt/0	113
8.17.4	halt/1	113
9	Evaluable functors	114
9.1	The simple arithmetic functors	114
9.1.1	Evaluable functors and operations	114
9.1.2	Exceptional values	114
9.1.3	Integer operations and axioms	114
9.1.4	Floating point operations and axioms	115
9.1.5	Mixed mode operations and axioms	116
9.1.6	Type conversion operations	117
9.1.7	Examples	117
9.2	The format of other evaluable functor definitions	119
9.2.1	Description	119
9.2.2	Template and modes	119
9.2.3	Errors	119
9.2.4	Examples	119
9.3	Other arithmetic functors	119
9.3.1	(**)/2 – power	119
9.3.2	sin/1	120
9.3.3	cos/1	120
9.3.4	atan/1	120
9.3.5	exp/1	121
9.3.6	log/1	121
9.3.7	sqrt/1	122
9.4	Bitwise functors	122
9.4.1	(>>)/2 – bitwise right shift	122
9.4.2	(<<)/2 – bitwise left shift	122

9.4.3	(/)/2 – bitwise and	123
9.4.4	(\)/2 – bitwise or	123
9.4.5	(\)/1 – bitwise complement	124

Annex

A	Formal semantics	125
A.1	Introduction	125
A.1.1	Specification language: syntax	125
A.1.2	Specification language: semantics	126
A.1.3	Comments in the formal specification	126
A.1.4	About the style of the Formal Specification	127
A.1.5	References	127
A.2	An informal description	127
A.2.1	Search-tree for “pure” Prolog	128
A.2.2	Search tree for “pure” Prolog with cut	131
A.2.3	Search-tree for kernel Prolog	132
A.2.4	Database and database update view	134
A.2.5	Exception handling	135
A.2.6	Environments	135
A.2.7	The semantics of a standard program	136
A.2.8	Getting acquainted with the formal specification	136
A.2.9	Built-in predicates	137
A.2.10	Relationships with the informal semantics of 7.7 and 7.8	138
A.3	Data structures	138
A.3.1	Abstract databases and terms	138
A.3.2	Predicate indicator	143
A.3.3	Forest	143
A.3.4	Abstract lists, atoms, characters and lists	146
A.3.5	Substitutions and unification	148
A.3.6	Arithmetic	149
A.3.7	Difference lists and environments	149
A.3.8	Built-in predicates and packets	150
A.3.9	Input and output	154
A.4	The Formal Semantics	155
A.4.1	The kernel	155
A.5	Control constructs and built-in predicates	170
A.5.1	Control constructs	170
A.5.2	Term unification	171
A.5.3	Type testing	172
A.5.4	Term comparison	172
A.5.5	Term creation and decomposition	173
A.5.6	Arithmetic evaluation - (is)/2	174
A.5.7	Arithmetic comparison	174
A.5.8	Clause retrieval and information	174
A.5.9	Clause creation and destruction	175
A.5.10	All solutions	178
A.5.11	Stream selection and control	180
A.5.12	Character input/output	183
A.5.13	Byte input/output	189
A.5.14	Term input/output	192
A.5.15	Logic and control	194
A.5.16	Atomic term processing	195
A.5.17	Implementation defined hooks	198

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.

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 $\mathbb{Z}$ or $\mathbb{R}$.

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* T_1 contains another *term* T_2 if either T_1 and T_2 are *identical terms*, or T_1 is a *compound term*, one of whose *arguments* contains T_2 .

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

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

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 $\mathcal{R}$,

$$\begin{aligned} x \in X &\Rightarrow rnd(x) = x \\ x < y &\Rightarrow rnd(x) \leq rnd(y) \end{aligned}$$

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:

$$add_I : I \times I \rightarrow I \cup \{\text{int_overflow}\}$$

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	$\{ X \rightarrow a, Y \rightarrow 3+Z, Z \rightarrow b \}$
Term	T	$f \circ \circ (X, Y, Z)$
New term	$T\sigma$	$f \circ \circ (a, 3 + Z, b)$

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:

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

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
$\iff$	$\mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$	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: $\circ : \text{Substitution} \times \text{Substitution} \rightarrow \text{Substitution}$

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

4.1.3.2 $|x|$ – abs x

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

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

4.1.3.3 $\lfloor x \rfloor$ – floor x

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

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

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

4.1.3.5 $\sqrt{x}$ – square root

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

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:

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

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

$top_D(push_D(d, s)) = d$
 $top_D(newstack_D) = \text{error}$
 $pop_D(push_D(d, s)) = s$
 $pop_D(newstack_D) = \text{error}$
 $isempty_D(newstack_D) = \text{true}$
 $isempty_D(push_D(d, s)) = \text{false}$

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:

$apply_mapping_T(a, identity_mapping_T) = a$
 $apply_mapping_T(a, mapping_T(a', b, m))$
 $\quad = b$ if $a = a'$
 $\quad = apply_mapping_T(a, m)$ if $a \neq a'$
 $update_mapping_T(a, b, identity_mapping_T)$
 $\quad = identity_mapping_T$ if $a = b$
 $\quad = mapping_T(a, b, identity_mapping_T)$ if $a \neq b$
 $update_mapping_T(a, b, mapping_T(a', b', m))$
 $\quad = mapping_T(a', b', update_mapping_T(a, b, m))$
 $\quad$ if $a \neq a'$
 $\quad = mapping_T(a, b, m)$
 $\quad$ if $a = a'$ and $a \neq b$
 $\quad = m$
 $\quad$ 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 *Conv_C*, 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 $f(x_1, \dots, x_n)$ where:

- 1) f is the functor name of the compound term, and
- 2) n is the arity of the compound term, and
- 3) $x_1, \dots, x_n$ 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 ($\cdot$) 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: $d \cdot t$ d t

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

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 $f(A_1, \dots, A_n)$ where each argument A_i 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

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

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 1term 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	<code>:- --></code>
1200	fx	<code>:- ?-</code>
1100	xfy	<code>;</code>
1050	xfy	<code>-></code>
1000	xfy	<code>', '</code>
900	fy	<code>\+</code>
700	xfx	<code>= \=</code>
700	xfx	<code>== \== @< @=< @> @>=</code>
700	xfx	<code>=..</code>
700	xfx	<code>is == \= < =< > >=</code>
500	yfx	<code>+ - /\ \/</code>
400	yfx	<code>* / // rem mod << >></code>
200	xfx	<code>**</code>
200	xfy	<code>^</code>
200	fy	<code>- \</code>

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\nb' 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:

- If T is an atomic term then S_V is the empty set,
- Else if T is a variable then S_V is the set $\{T\}$,
- 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:

- If T is a variable or an atomic term then EV is the empty set,
- Else if T unifies with $\wedge(v, G)$ then EV is the union of the variable set (7.1.1.1) of v and the existential variables set of the term G ,
- Else EV is the empty set.

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

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 $Z \wedge (A+X+Y+Z)$ 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 $\mathbb{Z}$ 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,

$$I = \mathbb{Z}$$

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:

$$\begin{aligned} r &\geq 2 \\ \wedge p &\geq 2 \\ \wedge p - 2 &\leq -emin \leq r^p - 1 \\ \wedge p &\leq emax \leq r^p - 1 \end{aligned}$$

These parameters should also satisfy:

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

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

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

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

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

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:

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

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

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

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

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

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

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 .

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

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 x_s .

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 T_2 is a renamed copy of a term T_1 if:

- T_2 is a variant of T_1 , and
- None of the variables in the variable set of T_2 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 $\wedge(., Goal)$ 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 $\wedge(X, foo(X))$.
- 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:

- x is the null atom and y is not the null atom, or
- 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
- 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:

- The arity of x is less than the arity of y , or
- 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
- 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 , x_i is the i th argument of x and y_i is the i th argument of y then ' $=$ ' (x_i , y_i), and

2) if x_N is the N th argument of x and y_N the N th argument of y and x_N *term_precedes* y_N .

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) $f(\dots) = g(\dots)$ where f and g are different functors, or

4) $f(a_1, a_2, \dots, a_N) = f(b_1, b_2, \dots, b_M)$ 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 $f(a_1, a_2, \dots, a_N) = f(b_1, b_2, \dots, b_N)$ 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

$$(v_1 = t_1, v_2 = t_2, \dots, v_N = t_N)$$

defines an *MGU*

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

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)	success (<i>unifiable</i>)
(7.3.2 h)	$MGU = \{ \}$
	$X = Y$
(7.3.2 h)	success (<i>unifiable</i>)
	$MGU = \{X \rightarrow Y\}$ or
	$MGU = \{Y \rightarrow X\}$
	$3 = 4$
(7.3.2 a1)	failure (<i>not unifiable</i>)
	$3 = f(X)$
(7.3.2 a2)	failure (<i>not unifiable</i>)
	$f(X) = g(X)$
(7.3.2 a3)	failure (<i>not unifiable</i>)
	$f(X) = f(g(X), 1)$
(7.3.2 a4)	failure (<i>not unifiable</i>)
	$f(X) = f(X)$
(7.3.2 d)	$X = X$
(7.3.2 b)	
(7.3.2 h)	success (<i>unifiable</i>)
	$MGU = \{ \}$
	$f(X, Y) = f(g(Y), a)$
(7.3.2 d)	$X = g(Y), Y = a$
(7.3.2 f)	$X = g(a), Y = a$
(7.3.2 h)	success (<i>unifiable</i>)
	$MGU = \{X \rightarrow g(a), Y \rightarrow a\}$
	$f(X, X, X) = f(Y, g(Y), a)$
(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 (<i>not unifiable</i>)
	$f(X, X, X) = f(Y, g(Y), a)$
(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 (<i>not unifiable, positive occurs-check</i>)

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(P)` 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 `ConvC`, 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 `ConvC` 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 `ConvC` 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(CClause)` (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:

- 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.
- 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
<code>(', ')/2</code>	Conjunction
<code>(;)/2</code>	Disjunction
<code>(->)/2</code>	If-then
<code>!/0</code>	Cut
<code>call/1</code>	Call
<code>true/0</code>	True
<code>fail/0</code>	Fail
<code>catch/3</code>	Catch
<code>throw/1</code>	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:

- If T is a variable then G is the control construct `call (7.8.3)`, whose argument is T .
- 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.
- 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 :

- If N is zero then T is the atom P .
- 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 *$cutparent$* .