# ISO/IEC DTR 13211–3:2006
# Definite clause grammar rules

Editor: Klaus.Daessler
klaus.daessler@mathint.com

November 20, 2012

## Introduction

This technical report (TR) is an optional part of the International Standard for Prolog, ISO/IEC 13211. Prolog manufacturers wishing to implement Definite Clause Grammar rules in a portable way should do so in compliance with this technical report.

Grammar rules provide convenient functionality for parsing and processing text in a variety of languages. They have been implemented by many Prolog processors.

This TR is an extension to the ISO/IEC 13211–1 Prolog standard, adopting a similar structure. In particular, this TR adds new sections and clauses to, or modifies existing clauses on ISO/IEC 13211–1.

### Previous editors and draft documents

- Paulo Moura: *ISO/IEC DTR 13211– 3:2006 Grammar rules in Prolog*, ISO, 2006-10
- Roger Scowen: *N171 — ISO/IEC DTR 13211–3:2004 Grammar rules in Prolog*, ISO, 2004-05
- Tony Dodd: *DCGs in ISO Prolog — A Proposal*, BSI, 1992

## Contributors

*This list needs to be completed; so far we have only included people present at the ISO meetings collocated with the ICLP (2005, 2006, and 2007), Richard O'Keefe, and the authors of the two drafts cited.*

- Bart Demoen (Belgium)
- Jan Wielemaker, (Netherlands)
- Joachim Schimpf (UK)
- Jonathan Hodgson (USA)
- Jose Morales (Spain)
- Katsuhiko Nakamura (Japan)
- Klaus Daessler (Germany)
- Manuel Carro (Spain)
- Mats Carlsson (Sweden)
- Paulo Moura (Portugal)
- Pierre Deransart (France)
- Peter Szabo (Hungary)
- Peter Szeredi (Hungary)
- Richard O'Keefe (NZ)
- Roger Scowen (UK)
- Tony Dodd (UK)
- Ulrich Neumerkel (Austria)
- Victor Santos Costa (Portugal)
- Manuel Hermenegildo (Spain)
- Mike Covington (USA)
- David Warren (USA)

# 1 Scope

This TR is designed to promote the applicability and portability of Prolog grammar rules in data processing systems that support standard Prolog as defined in ISO/IEC 13211–1:1995 and, if supported by the processor, in ISO/IEC 13211–2:2000, and the 2 Corrigenda of 13211-1. As such, this TR specifies:

a) The representation, syntax, and constraints of Prolog grammar rules

b) A logical expansion of grammar rules into Prolog clauses

c) A set of built-in predicates for parsing with and expanding grammar rules

NOTE 1 — This part of ISO/IEC 13211 will supplement ISO/IEC 13211–1:1995 including its corrigenda, and ISO/IEC 13211–2:2000.

NOTE 2 — The limitations, expressed in clause 1, Scope, of ISO/IEC 13211–1:1995 apply to this TR.

# 2 Normative references

The following TR contains provisions which, through reference in this text, constitute provisions of this TR as Part of ISO/IEC 13211.

- ISO/IEC 13211-1:1995
- ISO/IEC 13211-2:2000
- Corrigendum 1 of 13211-1:2006
- Corrigendum 2 of 13211-1:2012

# 3 Definitions

*For the purposes of this TR, the following definitions are added to the ones specified in ISO/IEC 13211–1:*

**3.1 body (of a grammar-rule):** The second argument of a grammar-rule. A grammar-body-sequence, or a grammar-body-alternative, or a grammar-body-not, or a grammar-body-element.

**3.2 clause-term:** A read-term `T.` in Prolog text where `T` does not have principal functor `(:-)/1` nor principal functor `(-->)/2.` (This definition replaces clause 3.33 of ISO/IEC 13211–1).

**3.3 definite clause grammar:** A sequence of grammar-rules.

**3.4 comprehensive terminal-sequence:** see terminal-sequence, comprehensive.

**3.5 expansion (of a grammar-rule):** The preparation for execution (cf. ISO/IEC 13211–1, section 7.5.1) of a grammar rule.

**3.6 generating (wrt. a definite clause grammar):** Producing legal terminal-sequences, completely parseable by that grammar, obeying right-hand-contexts, if any.

**3.7 grammar-body-alternative:** A compound term with principal functor `(;)/2` with each argument being a body (of a grammar-rule).

**3.8 grammar-body-element:** A cut (the atom `!`), or a grammar-body-goal, or a non-terminal, or a terminal-sequence.

**3.9 grammar-body-goal:** A compound term with principal functor `{}/1` whose argument is a goal.

**3.10 grammar-body-not:** A compound term with principal functor `(\+)/1` whose argument is a body (of a grammar rule.)

**3.11 grammar-body-sequence:** A compound term with principal functor `(',')/2` and each argument being a body (of a grammar-rule).

**3.12 grammar-body-terminals:** A terminal-sequence.

**3.13 grammar-rule:** A compound term with principal functor `(-->)/2`.

**3.14 grammar-rule-term:** A read-term `T.` where `T` is a grammar-rule.

**3.15 head (of a grammar-rule):** The first argument of a grammar-rule. Either a non-terminal (of a grammar), or a compound term whose principal functor is `(,)/2`, where the first argument is a non-terminal (of a grammar), and the second argument is a latent right context.

**3.16 new variable with respect to a term T:** A variable that is not a member of the variable set of `T`.

**3.17 non-terminal (of a grammar):** An atom or compound term that denotes a non-terminal symbol of the grammar.

**3.18 non-terminal indicator:** A compound term `A//N` where `A` is an atom and `N` is a non-negative integer, denoting one particular non-terminal (cf 7.14.4) .

**3.19 parsing (wrt. a definite clause grammar):** Successively accepting or consuming legal terminal-sequences, assigning them to corresponding non-terminals and obeying right-hand-contexts, if any.

**3.20 remaining terminal-sequence:** See terminal-sequence, remaining.

**3.21 right-hand-context:** A terminal-sequence occurring as optional second argument of a grammar-rule-head, constraining parsing resp. generating during completing this grammar rule application.

**3.22 steadfastness of a goal wrt. an argument** Goal `G` is steadfast in argument `n` of its argument list, if for any term `T` that is the `n`th argument in the goal, and the goal `Gf` that results by replacing `T` by a fresh variable `Vf` the results of `G` and `(Gf, Vf=T)` are the same.

**3.23 terminal (of a grammar):** Any Prolog term that denotes a terminal symbol of the grammar.

**3.24 terminal-sequence:** A list (cf. ISO/IEC 13211–1, section 6.3.5) whose first argument, if any, is a terminal (of a grammar), and the second argument is a terminal-sequence, if any.

**3.25 terminal-sequence, comprehensive:** Terminal sequence containing a (possibly empty) prefix accepted by `phrase/3` (cf 8.1.1) .

**3.26 terminal-sequence, remaining:** Rest of comprehensive terminal-sequence without the leading terminal-sequence covered by a grammar rule body.

**3.27 variable, new with respect to a term T:** See *new variable with respect to a term `T`*.

# 4 Symbols and abbreviations

NOTE — No changes from the ISO/IEC 13211–1 Prolog standard.

# 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 TR, and
   2. the requirements of ISO/IEC 13211–1, and
   3. 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 TR, and
   2. the requirements of ISO/IEC 13211–1, and
   3. 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 TR, and
   2. the requirements of ISO/IEC 13211–1, and
   3. the implementation defined and implementation specific features of the Prolog processor,

d) Specify all permitted variations from this TR in the manner prescribed by this TR and by the ISO/IEC 13211–1, 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.

NOTE — This extends corresponding section of ISO/IEC 13211–1.

## 5.2 Prolog text

NOTE — No changes from the ISO/IEC 13211–1 Prolog standard.

## 5.3 Prolog goal

NOTE — No changes from the ISO/IEC 13211–1 Prolog standard.

## 5.4 Documentation

*The corresponding section on the ISO/IEC 13211–1 Prolog standard is modified as follows:*

A conforming Prolog processor shall be accompanied by documentation that completes the definition of every implementation defined and implementation specific feature specified in this TR and on the ISO/IEC 13211–1 Prolog standard.

## 5.5 Extensions

*The corresponding section on the ISO/IEC 13211–1 Prolog standard is modified as follows:*

A processor may support, as an implementation specific feature, any construct that is implicitly or explicitly undefined in this TR or in the ISO/IEC 13211–1 Prolog standard.

### 5.5.2 Predefined operators

*Please see section 6.3 for the new predefined operators that this TR adds to the ISO/IEC 13211–1 Prolog standard.*

# 6 Syntax

## 6.1 Notation

### 6.1.1 Backus Naur Form

*No changes from the ISO/IEC 13211–1 Prolog standard.*

### 6.1.2 Abstract term syntax

*The text near the end of this section on the ISO/IEC 13211–1 Prolog standard is modified as follows:*

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

a) `d.t` where `d` is the abstract syntax for a directive, and `t` is Prolog text, or

b) `g.t` where `g` is the abstract syntax for a grammar rule, and `t` is Prolog text, or

c) `c.t` where `c` is the abstract syntax for a clause, and `t` is Prolog text, or

d) `nil`, the empty list.

*The following section extends, with the specified number, the corresponding ISO/IEC 13211–1 section.*

### 6.1.3 Variable names convention for terminal-sequences

This TR uses variables named `S0`, `S1`, ..., `S` to represent the terminal-sequences used as arguments when processing grammar rules or when expanding grammar rules into clauses. In this notation, the variables `S0`, `S1`, ..., `S` can be regarded as a sequence of states, with `S0` representing the initial state and the variable `S` representing the final state. Thus, if the variable `Si` represents the terminal-sequence in a given state, the variable `Si+1` will represent the remaining terminal-sequence after processing `Si` with a grammar rule.

## 6.2 Prolog text and data

*The first paragraph of this section on ISO/IEC 13211–1 is modified as follows:*

Prolog text is a sequence of read-terms which denote (1) directives, (2) grammar rules, and (3) clauses of user-defined procedures.

### 6.2.1 Prolog text

*The corresponding section on the ISO/IEC 13211–1 is modified as follows:*

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

| | | | |
|---|---|---|---|
| | `prolog text =` | `p text` | |
| Abstract: | *pt* | *pt* | |
| | `p text =` | `directive term ,` | `p text` |
| Abstract: | *d.t* | *d* | *t* |
| | `p text =` | `grammar rule term ,` | `p text` |
| Abstract: | *g.t* | *g* | *t* |
| | `p text =` | `clause term ,` | `p text` |
| Abstract: | *c.t* | *c* | *t* |
| | `p text =` | `;` | |
| Abstract: | *nil* | | |

#### 6.2.1.1 Directives

*Syntactically, there are no changes w.r.t. ISO/IEC 13211–1 Prolog standard, with exception of the operator syntax (cf 6.3); for the semantics compare 7.4.2. Whenever directives are applicable to non-terminals, the respective non-terminal indicators (cf 7.14.4) , as arguments of directives, shall be used like predicate indicators.*

#### 6.2.1.2 Clauses

*The corresponding section on the ISO/IEC 13211–1 is modified as follows:*

| | | |
|---|---|---|
| | `clause term =` | `term, end` |
| Abstract: | *c* | *c* |
| Priority: | 1201 | |
| Condition: | The principal functor of `c` is `(:-)/1` | |
| Condition: | The principal functor of `c` is `(-->)/2` | |

NOTE — Subclauses 7.5 and 7.6 define how clauses become part of the database.

*The following subclause extends, with the specified number, the corresponding ISO/IEC 13211–1 subclause:*

#### 6.2.1.3 Grammar rules

| | | |
|---|---|---|
| | `grammar rule term =` | `term, end` |
| Abstract: | *gt* | *gt* |
| Priority: | 1201 | |
| Condition: | The principal functor of `gt` is `(-->)/2` | |
| | `grammar rule =` | `grammar rule term` |
| Abstract: | *g* | *g* |

NOTE — Clause 11 of this TR defines how a grammar rule in Prolog text is expanded into an equivalent clause when Prolog text is prepared for execution.

#### 6.2.1.4 Right-hand contexts

| | | |
|---|---|---|
| | `right hand context term =` | `term` |
| Abstract: | *rc* | *rc* |
| Priority: | 1201 | |
| Condition: | none | |
| | `right hand context =` | `right hand context term` |
| Abstract: | *r* | *r* |

### 6.3 Terms

Extend the operator table of subclause 6.3.4.4 of ISO/IEC 13211–1 as follows:

| `Priority` | `Specifier` | `Operator(s)` |
|---|---|---|
| 1105 | `xfy` | `'\|'` |

NOTE — The operator `(-->)/2`, specified in subclause 6.3.4.4 of the ISO/IEC 13211–1 Prolog standard, is used as the principal functor of grammar rules.

## 7 Language concepts and semantics

*The following subclause extends, with the specified number, the corresponding ISO/IEC 13211–1 subclause:*

## 7.4 Prolog text

### 7.4.2 Directives

A non-terminal indicator may appear anywhere that a predicate indicator can appear in the following directives: `dynamic/1`, `multifile/1`, and `discontiguous/1` as specified in subclause 7.4.2 of the ISO/IEC 13211–1 Prolog standard.

The directives
`op/3`, `char_conversion/1`, `initialization/1`, `include/1`, `ensure_loaded/1`, and `set_prolog_flag/2`
shall not have non-terminal indicators as arguments.

If the Prolog processor supports Prolog Modules as defined in ISO/IEC 13211-2:2000, a non-terminal indicator may appear anywhere that a predicate indicator can in the Module interface directives `export/1, reexport/1, reexport/2` and the Module directives `import/1, import/2` of subclause 6.24 of ISO/IEC 13211-2:2000.

## 7.13 Predicate properties

*The following subclause extends subclauses 6.8 and 7.2.2 of ISO/IEC 13211–2 Prolog Modules:*
The following optional property is added to the list of predicate properties of subclause 6.8 of ISO/IEC 13211–2:

- `expanded_from(non_terminal, A//N)` — The predicate results from the expansion of a grammar rule for the non-terminal `A//N`

NOTE — the `expanded_from/2` property name was chosen in order to account for other possible, implementation-specific expansions.

## 7.14 Grammar rules

### 7.14.1 terminals and non-terminals

Zero, one or more terminals are represented by terms directly contained in lists in order to distinguish them from non-terminals (string notation may be used as an alternative to lists when terminals are characters and the flag "double_quotes" has value "chars"; see subclauses 6.3.7 and 6.4.6 of ISO/IEC 13211–1). Non-terminals are represented by callable terms.

#### 7.14.1.1 Example

A simple grammar consisting of 9 grammar rules, parsing or generating terminal sequences of the form

```
[the, dog, runs]
[the, dog, barks]
```

```
[the, dog, bites]
[the, nice, cat, barks]
```

is given:

```
sentence --> nominal_phrase, verbal_phrase.
verbal_phrase --> verb.
nominal_phrase --> article, noun.
article --> [the].
noun --> [dog].
noun --> [nice, cat].
verb --> [runs].
verb --> [barks].
verb --> [bites].
```

Here the symbols `sentence, verbal_phrase, verb` etc. denote non-terminals, whereas `[runs], [nice, cat]` etc. denote terminals.

NOTE — In the context of a grammar rule, *terminals* represent tokens of some language, and *non-terminals* represent sequences of tokens (see, respectively, definitions 3.17 and 3.23).

### 7.14.2 Format of grammar rules

A grammar rule has the format:

```
GRHead --> GRBody.
```

A grammar rule is interpreted as stating that its head, `GRHead`, can be rewritten by its body, `GRBody`. The head and the body of grammar rules are constructed from *non-terminals* and *terminals*. The head of a grammar rule is a non-terminal or the conjunction of a non-terminal and, following, a terminal-sequence (a *right-hand-context*, see 7.14.3):

```
NonTerminal --> GRBody.

NonTerminal, RightHandContext --> GRBody.
```

The control constructs that may be used on a grammar rule body are described in section 7.15. An empty grammar rule body is represented by an empty list:

```
GRHead --> [].
```

The empty list cannot be omitted, i.e. there is no `(-->)/1` form for grammar rules.

### 7.14.3 Right-hand-contexts

#### 7.14.3.1 Description

A *right-hand-context* is a terminal-sequence (see 3.24), as an optional second argument of the head of a grammar rule (see 3.15). A right-hand-context contains terminals that are prefixed to the remaining terminal-sequence after successful application of the grammar rule.

#### 7.14.3.2 Syntax

```
RightHandContext   =  TerminalSequence
```

#### 7.14.3.3 Examples

Assume we need rules to *look-ahead* one or two tokens that would be consumed next. This could be accomplished by the following grammar rules:

```
look_ahead(X), [X]      --> [X].

look_ahead(X, Y), [X,Y] --> [X,Y].
```

When used for parsing, procedurally, these grammar rules can be interpreted as, respectively, consuming, and then restoring, one or two terminals.

Another example may be a small grammar rule with right-hand-context:

```
phrase1, [word] --> phrase2, phrase3.
```

After preparation for execution this occurs in the database as follows.

```
phrase1(S0, S):-
    phrase2(S0, S1),
    phrase3(S1, S2),
    S = [word | S2].
```

NOTES

1 In case of parsing, as soon as `phrase2` and `phrase3` successfully parsed the comprehensive terminal-sequence (input list), the terminal `word` is prefixed to the remaining terminal-sequence. `word` is then the first terminal to be consumed in further parsing after `phrase1`. Thus the path of further parsing is

constrained by the right-hand-context.

2 Sometimes the concepts *comprehensive terminal-sequence* resp. *remaining terminal-sequence* are named *input list* resp. *output list*. This is misleading, because it only considers the case of parsing using a grammar. There a terminal list shall be parsed wrt. non-terminals, and a rest will remain after a parsing step. The inverse case, generating sentences by expanding grammars, where the comprehensive terminal-sequence is the real output list, is ignored by such wording.

3 There are exotic cases, where the remaining terminal-sequence is not part of the comprehensive terminal-sequence, e.g. with the following grammar rule...but in fact there is a trailing sequence as the following example shows.

```
nt,[word] --> [].
```

which is expanded by preparation for execution to:

```
nt(S0, [word|S]) :-
    S0 = S.
```

This non-terminal `nt` represents an empty terminal sequence (cf. 7.15.1), but constrains further parsing to take place with `word` as next token. The comprehensive terminal-sequence is identical with the remaining terminal-sequence for that non-terminal.

4 It should be noted that where the right-hand context is non empty, as in the cases above then `phrase/2` (cf 8.1.1.3) cannot succeed as the right-hand context is pushed back into the terminal sequence.

5 Some processors allow a cut in the right-hand context; e.g.

```
a, !, [word] --> b.
```

Moving this cut to the end of the grammar body, c.f. `a, [word] --> b, !.` has the identical effect on execution. Thus this TR does not permit a cut in the right hand context.

### 7.14.4 Non-terminal indicator

A *non-terminal indicator* is a compound term with the format `//(A, N)` where `A` is an atom and `N` is a non-negative integer.

The non-terminal indicator `//(A, N)` indicates the grammar rule non-terminal whose functor is `A` and whose arity is `N`.

NOTES

1 In Prolog text, including ISO/IEC 13211–1 and this TR, a non-terminal indicator `//(A, N)` is normally written as `A//N`.

2 The concept of non-terminal indicator is similar to the concept of *predicate indicator* defined in subclauses 3.131 and 7.1.6.6 of the ISO/IEC 13211–1 Prolog. Non-terminal indicators may be used in exception terms thrown when processing or using grammar rules. In addition, non-terminal indicators may appear at some places, where a predicate indicator as defined in ISO/IEC 13211–1 can appear. See 7.4.2. Furthermore non-terminal indicators may be used as a predicate property (cf. subclause 7.13). In particular, using non-terminal indicators in predicate directives allows the details of the expansion of grammar rules into Prolog clauses to be abstracted.

#### 7.14.4.1 Examples

For example, given the following grammar rule:

```
sentence --> noun_phrase, verb_phrase.
```

The corresponding non-terminal indicator for the grammar rule left-hand side non-terminal is `sentence//0`.

Example:

```
:- export(sentence//0).
```

So the grammar rules of M for the non-terminal `sentence//0` can be used outside module M.

### 7.14.5 Prolog goals in grammar rules

#### 7.14.5.1 Description

In the body of grammar rules, curly brackets enclose a non-empty sequence of Prolog goals that are executed when the grammar rule, prepared for execution, is processed.

`{}/1` is a functor of a `grammar body goal` {prolog_goal} (cf. Definition 3.9). After expansion `prolog_goal` is unchanged and is handled as a ordinary prolog goal according to subclause 7.7 of ISO/IEC 13211-1 Prolog.

NOTE — The ISO/IEC 13211–1 Prolog standard defines, in subclause 6.3.6, a *curly bracketed term* as a compound term with principal functor `'{}'/1`, whose argument may also be expressed by enclosing its argument in curly brackets.

#### 7.14.5.2 Examples

Consider, for example, the following grammar rule:

```
digit(D) --> [C], {0'0 =< C, C =< 0'9, D is C - 0'0}.
```

This rule, which assumes that C is a string consisting of a single character, recognizes a single terminal as the code of a character representing a digit when the corresponding numeric value can be unified with the non-terminal argument.

## 7.15 Grammar control constructs

This subclause describes the meaning of special non-terminals which are part of grammar rules: `[]//0, (',')//2, (;)//2, (->)//2 in an if-then-else, ('|')//2, {}//1, phrase//1, !//0, (\+)//1 and call//1`.
After preparation for execution of Grammar Rules, named "Grammar Rule expansion", or "expansion" for short, these non-terminals result in control constructs resp. built-in predicates of ISO/IEC 13211-1 Prolog.
Because the expansion is not simply a replacement, but must satisfy some constraints resp. conditions hold, this process may appear somewhat difficult.
The expansion is understood best by describing it wrt. the built-in predicates `phrase/3` and `phrase/2` (see 8.1.1). Execution of these predicates serves two goals: Firstly the final Grammar Rule Expansion, when this has not taken place earlier, i.e. preparation for execution of its Grammar Body and arguments; thereafter the execution of the resulting Prolog goals.
From the parsing viewpoint `phrase(NT, S)` is true when the terminal sequence `S` is covered by the non-terminal `NT`. Or, from the generation viewpoint it is true when `NT` generates `S`. The Grammar Body non-terminals are described here informally. Section 11 provides a reference implementation that further defines the semantics of expansion. In particular this subclause explicates the linkages between the terminal sequences upon expansion of the control constructs.

### 7.15.1 `[]//0` – empty terminal sequence

The expansion result of the grammar control construct `empty terminal sequence` (a terminal sequence without contents) unifies the actual remaining terminal sequence with the actual comprehensive terminal sequence, i.e. has no effect on parsing resp. generating during execution.

### 7.15.2 `(',')//2` – concatenation

In the body of a grammar rule `(',')//2` is the principal functor of a `grammar-body-sequence`(cf. Definition 3.12) with a first grammar body GBFirst and a second grammar body GBSecond. Each of them is then subject to subsequent separate expansion - GBFirst first, and then GBSecond. After being completely expanded GBFirst and GBSecond are arguments of a conjunction wrt. subclause 7.8.5 of ISO/IEC 13211-1 Prolog.

For a formal description of expansion of concatenation see subclause 11, third clause of `dcg_cbody/4`.

If contained directly in the head of a grammar rule, `(',')//2` is the main functor of a term consisting of a grammar rule body and a right-hand-context; cf 7.14.3.

### 7.15.3 `('|')//2` – alternative

The operator `('|')` used as a non-terminal `'|'/2` has the same behaviour as `(;)/2`. See section 7.15.4

For a formal description of expansion this form of alternative see section 11, sixth clause of `dcg_cbody/4`.

### 7.15.4 `(;)//2` – alternative

In the body of a grammar rule `(;)//2` is the principal functor of a `grammar-body-alternative`. (cf. definition 3.7) with a first grammar body GBFirst and a second grammar body GBSecond. Each of them is then subject to subsequent separate expansion - GBFirst first, and then GBSecond. After being completely expanded GBFirst and GBSecond shall be arguments of a disjunction `(;)/2` wrt. subclause 7.8.6 of ISO/IEC 13211-1 Prolog.
For a formal description of expansion of an alternative see section 11, fifth clause of `dcg_cbody/4`.

NOTE — The effect of comma and semicolon, `(',')//2`, `(;)//2`, may be understood best by application of `write_canonical/1` (see subclause 8.14.2.5 of ISO/IEC 13211–1) on a grammar rule, containing them:

```
?-write_canonical((sentence --> subject, verb, object;
                                object, verb, subject)).

-->(sentence, ;(',','(subject, ','(verb, object)),
               (',','(object, ','(verb, subject))))
yes
```

This leads to the following Prolog clause after preparation for execution :

```
sentence(S0, S) :-
   ( subject(S0, S1),
     verb(S1, S2),
     object(S2, S)
   ; object(S0, S3),
     verb(S3, S4),
     subject(S4, S)
   ).
```

### 7.15.5 `(;)//2` – if-then-else

NOTE — `(;)//2` serves two different functions depending on whether or not the first argument is a compound term with functor `(->)/2`. See 7.15.4 for the use of `(;)//2` for `alternative`, when the first argument of `';'(_,_)` does not immediately contain a term with functor `'->'(_,_)`.

The grammar control construct `(;)/2` may have as arguments a grammar body GBFirst and a second grammar body GBSecond, where the principal functor of GBFirst is `(->)/2` with arguments GBIf and GBThen. Each of GBIf, GBThen, and GBSecond is subject to separate expansion to GBIfExpanded, GBThenExpanded and GBSecondExpanded, respectively. The result is a Prolog disjunction with the if-then control construct `(->)(GBIfExpanded, GBThenExpanded)` as first, and `GBSecondExpanded` as second argument.

For a formal description of expansion of if-then-else see section 11, fourth clause of `dcg_cbody/4`.

There shall be no use of `(->)/2` in grammar rules except in the first argument of an alternative as described above. Further a conforming proceesr shall not permit the use of `(->)/2` in any other way as an implementation specific extension.

### 7.15.6 {}//1 – extra conditions, external goal

The argument `G` of {}`//1` is not subject to expansion, i.e. the Prolog goals inside the braces remain unexpanded.
For a formal description of expansion of extra conditions see section 11, ninth clause of `dcg_cbody/4`.

### 7.15.7 `phrase//1`

Expanding the non-terminal `phrase//1` with argument `G` shall result in the expansion `phrase/3` with first argument `G`.

For a formal description of expansion of `phrase//1` see section 11, seventh clause of `dcg_cbody/4`.

### 7.15.8 `!//0` – cut

In the body of a grammar rule `!//0` is functor of a `grammar-body-cut`. After expansion the grammar-body-cut becomes the functor `!/0` as in subclause 7.8.4 of ISO/IEC 13211-1 Prolog. For a formal description of expansion of `!//0` see section 11, eleventh clause of `dcg_cbody/4`.

Implementations conforming to this DTR shall not define or use a predicate `!/2`

### 7.15.9 `(\+)//1`

In the body of a grammar rule `(\+)//1` is functor of a `grammar-body-not` (cf. Definition 3.10) After expansion of `(\+)//1` the functor `(\+)/1` is applied to the expanded argument of the `grammar-body-not`. If the resulting goal succeeds the expanded rule does not change the comprehensive terminal sequence.

For a formal description of expansion of `(\+)//1` see section 11, tenth clause of `dcg_cbody/4`.

Implementations conforming to this DTR shall not define or use a predicate `(\+)//3`.

NOTE — The effect of `(\+)//1` can be seen in the following example.

The grammar rule

```
a --> \+ b.
```

may be expanded to:

```
a(S0, S) :- \+ b(S0, _), S0 = S.
```

### 7.15.10 call//1

Expanding, i.e. preparing for execution of the non-terminal

```
call//1
```

shall result in the expansion

```
call/3
```

which is a legal goal for `call/3` which is required by this DTR and defined in 8.15.4 of ISO/IEC 13211-1:1995/Cor.2:2012(E).

For a formal description of expansion of `call//1` see section 11, ninth clause of `dcg_cbody/4`.

NOTE — Consider the following example for the correspondence for grammar rules between `call//1` and `call/3`:

```
atom_charsdiff(Atom, Xs0, Xs):-
    atom_chars(Atom, Chars),
    append(Chars, Xs, Xs0).

atomchars(Atom) --> call(atom_charsdiff(Atom)).
```

```
at_eos_pred([ ], [ ]).

at_eos --> call(at_eos_pred).
```

A Prolog processor may support additional control constructs. Examples include *soft-cuts* and control constructs that enable the use of grammar rules stored on encapsulation units other than modules, such as objects. These additional control constructs must be treated as non-terminals by a Prolog processor working on a strictly conforming mode (see 5.1e).

### 7.16 Executing procedures expanded from grammar rules

If a grammar rule to be prepared for execution has a non-terminal indicator N//A, and N is the name of the predicate indicator N/A', with A' == A + 2, of a built-in predicate, existing in the complete database, the result of expansion and the behaviour of the prepared grammar rule on execution is implementation dependent. This does not hold for the built-in predicates defined in 7.15.

When the database does not contain a grammar rule with non-terminal indicator N//A during execution of a non-terminal with non-terminal indicator N//A , the error term as specified in clause 7.7.7b of ISO/IEC 13211–1 when the flag `unknown` is set to `error` shall be:

```
existence_error(procedure, N//A)
```

NOTES

1 Prolog Processors shall report errors resulting from execution of grammar rules at the same abstraction level as grammar rules whenever possible.

2 Parsing resp. generating of texts with grammar rules is defined in section 8.1.1. Grammar rules are expanded into Prolog clauses during preparation for execution, which maps the parsing or generating with a grammar rule body into executing a goal given a sequence of predicate clauses. See subclause 7.7 of ISO/IEC 13211–1 for details.

## 8 Built-in predicates

### 8.1 Grammar rule built-in predicates

#### 8.1.1 phrase/3, phrase/2

##### 8.1.1.1 Description

`phrase(GRBody, S0, S)` is true iff the comprehensive terminal sequence `S0` unifies either with the concatenation of the terminal sequence of `GRBody` with

the remaining terminal sequence `S`, or with the concatenation of a terminal sequence resulting from generation by the non-terminal of `GRBody` w.r.t. the current Grammar rules with the remaining terminal sequence `S`.
If the non-terminal of `GRBody` is followed by a right-hand-context (cf. Definition 3.2), then the right-hand-context shall be prefixed to the remaining terminal sequence after having been parsed resp. generated wrt. the non-terminal of `GRBody`.
Procedurally, `phrase(GRBody, S0, S)` is executed by calling the Prolog goal corresponding to the expansion of the grammar rule body `GRBody`, given the terminal-sequences `S0` and `S`, according to the logical expansion of grammar rules described in subclause 11. See in particular the clauses for `dcg_rule/2`.

`phrase(GRBody, S0, S)` shall be steadfast in its third argument S (cf. Definition 3.22).

#### 8.1.1.2 Template and modes

```
phrase(+terminal-sequence, ?terminal-sequence, ?terminal-sequence)
```

#### 8.1.1.3 Bootstrapped built-in predicates

The built-in predicate `phrase/2` provides similar functionality to `phrase/3`. The goal `phrase(GRBody, S0)` is true when all terminals in the terminal-sequence S0 are consumed and recognized resp. generated:

```
phrase(GRBody, S0) :-
    phrase(GRBody, S0, []).
```

#### 8.1.1.4 Errors

a) `GRBody` is a variable
— `instantiation_error`

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

The following two errors are implementation defined if applied to `phrase/3` , i.e. no error checking is required on S0 and S by this TR for `phrase/3`. If, however, a Prolog processor offers them, their form and consequence must be the following:

c) `S0` is not a terminal-sequence
— `type_error(terminal-sequence, S0)`

For `phrase/2` error clause `c` is required.

d) `S` is not a terminal-sequence
— `type_error(terminal-sequence, S)`

NOTE — This relaxation is allowed because handling these errors could overburden a Prolog Processor.

#### 8.1.1.5 Examples

These examples assume that the following grammar rules has been correctly prepared for execution and are part of the complete database:

```
determiner --> [the].
determiner --> [a].

noun --> [boy].
noun --> [girl].

verb --> [likes].
verb --> [scares].

noun_phrase --> determiner, noun.
noun_phrase --> noun.

verb_phrase --> verb.
verb_phrase --> verb, noun_phrase.

sentence --> noun_phrase, verb_phrase.
```

Some example calls of `phrase/2` and `phrase/3`:

```
| ?- phrase([the], [the]).

yes

| ?- phrase(sentence, [the, girl, likes, the, boy]).

yes
 | ?- phrase(sentence, [the, girl, likes, the, boy, today]).

no

| ?- phrase(sentence, [the, girl, likes]).
 no

| ?- phrase(sentence, Sentence).

Sentence = [the, girl, likes, the, boy]
yes

| ?- phrase(noun_phrase, [the, girl, scares, the, boy], Rest).

Rest = [scares, the, boy]
yes
```

### 8.1.2 expand_term/2

#### 8.1.2.1 Description

`expand_term(Term, Expansion)` is true iff:

— `Expansion` unifies with the expansion of `Term`.

Procedurally, `expand_term(Term, Expansion)` is executed as follows:

a) If `Term` is a variable, unifies `Expansion` with `Term`

b) Else if the goal `term_expansion(Term, Expand)` is true then `Expansion` is unified with `Expand`

c) Else if the principal functor of `Term` is `(-->)/2` then it is assumed that it represents a grammar rule and `Expansion` is unified with its expansion into a Prolog clause

d) Else if the principal functor of `Term` is not `(-->)/2` then `Expansion` is unified with `Term`

e) Else the goal fails

NOTE — The predicate `term_expansion/2` is described in subclause 10.1.1.

#### 8.1.2.2 Template and modes

```
expand_term(?term, ?term)
```

#### 8.1.2.3 Errors

None.

#### 8.1.2.4 Examples

These examples assume that the following clauses for the `term_expansion/2` predicate have been prepared for execution:

```
term_expansion(succ(A, B), pred(B, A)).
term_expansion(0, zero).
term_expansion(1, one).
```

Some example calls of `expand_term/2`:

```
?- expand_term(Term, Expansion).

Term = Expansion
yes
```

```
 ?- expand_term(succ(1, 2), Expansion).

Expansion = pred(2, 1)
yes

?- expand_term(1, one).

yes

?- expand_term(odd(1), Expansion).

Expansion = odd(1)
yes
```

The next query returns an implementation-dependent Prolog clause; therefore the example below illustrates one possible answer:

```
?- expand_term((noun_phrase --> noun), Expansion)

Expansion = noun_phrase(A, B) :- noun(A, B)
yes
```

NOTES

1 Despite the fact that `expand_term/2` may be used to retrieve the translation of a grammar rule to a Prolog clause, users should not rely on a specific translation of a grammar rule, which is implementation-dependent.

2 Users may use alternate grammar rule translators by defining suitable clauses for `term_expansion/2`. Prolog implementers may use this mechanism to ensure backward compatibility with code written for older translators that are not compliant with this TR.

3. Some Prolog systems provide support for term expansion mechanisms, based on `term_expansion/2` and `expand_term/2` predicates, that may be used when compiling Prolog source files. The specification of such mechanisms — in particular how term expansion is performed during the compilation of Prolog source code — is outside the scope of this technical report.

# 9 Evaluable functors

NOTE — No changes from the ISO/IEC 13211–1 Prolog standard.

# 10 User-defined predicates

## 10.1 Grammar rule user-defined predicates

### 10.1.1 term_expansion/2

#### 10.1.1.1 Description

`term_expansion(Term, Expansion)` is a user-defined, dynamic, and multifile predicate, which may be used for the rewriting of terms. The predicate is automatically called by the built-in predicate `expand_term/2`, see 8.1.2. This predicate exists even if it has no clauses.

#### 10.1.1.2 Template and modes

```
term_expansion(?term, ?term)
```

#### 10.1.1.3 Errors

None.

#### 10.1.1.4 Examples

```
   term_expansion(next(Previous, Next), previous(Next, Previous)).
```

# 11 Logical Expansion: Prolog Definition of phrase/3

```
% To avoid name clashes: if phrase/3 exists already on the processor,
% the word phrase may be replaced by iso-phrase or else at two places.
% Missing prerequisite definitions as (append/3) shall be defined by
% the "Prolog Prologue" (ISO/IEC JTC1 SC22 WG17 N235)

phrase(GRBody, S0, S) :-
   dcg_body(GRBody, S0, S, Goal),
   call(Goal).

% dcg_rule(DCGrule, Expansion).
% Translates a DCG rule into a Prolog rule, when no error condition applies.

dcg_rule((NonTerminal, Terminals --> GRBody), (Head :- Body)) :-
   dcg_non_terminal(NonTerminal, S0, S, Head),
   dcg_body(GRBody, S0, S1, Goal1),
   dcg_terminals(Terminals, S, S1, Goal2),
   Body = (Goal1, Goal2).

dcg_rule((NonTerminal-->GRBody), (Head :- Body)) :-
```

```
   NonTerminal \= (_,_),
   dcg_non_terminal(NonTerminal, S0, S, Head),
   dcg_body(GRBody, S0, S, Body).

% translates a grammar goal non-terminal:

dcg_non_terminal(NonTerminal, S0, S, Goal) :-
   NonTerminal =.. NonTerminalUniv,
   append(NonTerminalUniv, [S0, S], GoalUniv),
   Goal =.. GoalUniv.

% translates a terminal-sequence:

dcg_terminals(Terminals, S0, S, S0 = List) :-
   append(Terminals, S, List).

% translates a grammar rule body:

dcg_body(Var, S0, S, Body) :-
   var(Var),
   Body = phrase(Var, S0, S).

dcg_body(GRBody, S0, S, Body) :-
   nonvar(GRBody),
   dcg_constr(GRBody),
   dcg_cbody(GRBody, S0, S, Body).

dcg_body(NonTerminal, S0, S, Goal) :-
   nonvar(NonTerminal),
   \+ dcg_constr(NonTerminal),
   NonTerminal \= (_->_),
   dcg_non_terminal(NonTerminal, S0, S, Goal).

dcg_constr([]).
dcg_constr([_|_]).
dcg_constr((_,_)).
dcg_constr((_;_)).
dcg_constr((_'|'_)).
dcg_constr(call(_)).
dcg_constr({_}).
dcg_constr(\+_).
dcg_constr(!).

dcg_cbody([], S0, S, (S0=S)).

dcg_cbody([T|Ts], S0, S, Goal) :-
```

```
   dcg_terminals([T|Ts], S0, S, Goal).

dcg_cbody(( GRFirst, GRSecond ), S0, S, ( First, Second )) :-
   dcg_body(GRFirst, S0, S1, First),
   dcg_body(GRSecond, S1, S, Second).

dcg_cbody(( GREither ; GROr ), S0, S, ( Either ; Or )) :-
   subsumes_term((_->_),GREither),
   dcg_cbody_if_then(GREither, S0, S, Either),
   dcg_body(GROr, S0, S, Or).

dcg_cbody(( GREither ; GROr ), S0, S, ( Either ; Or )) :-
   \+ subsumes_term((_->_),GREither),
   dcg_body(GREither, S0, S, Either),
   dcg_body(GROr, S0, S, Or).

dcg_cbody(( GREither '|' GROr ), S0, S, ( Either ; Or )) :-
   dcg_body(GREither, S0, S, Either),
   dcg_body(GROr, S0, S, Or).

dcg_cbody(phrase(Cont), S0, S, phrase(Cont, S0, S)).

dcg_cbody(call(Cont), S0, S, call(Cont, S0, S)).

dcg_cbody({Goal}, S0, S, (Goal, S0 = S)).

dcg_cbody(\+ GRBody, S0, S, (\+ Goal, S0 = S)) :-
   dcg_body(GRBody, S0, _, Goal).

dcg_cbody(!, S0, S, (!, S0 = S)).

dcg_cbody_if_then(( GRIf -> GRThen ), S0, S, ( If -> Then )) :-
   dcg_body(GRIf, S0, S1, If),
   dcg_body(GRThen, S1, S, Then).
```