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

Editor: Jonathan Hodgson
Email: jpehodgson@verizon.net

November 10, 2015

## 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 in 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 subclauses to, or modifies existing subclauses of ISO/IEC 13211–1.

### Previous editors and draft documents

- Klaus Däßssler: *ISO/IEC 13211 – 3: 2015 Grammar rules in Prolog*, ISO, 2010-2015
- 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 drafts cited above.*

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

# 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 two Corrigenda of 13211-1: ISO/IEC 13211-1 Technical Corrigendum 1:2007-11, and ISO/IEC 13211-1 Technical Corrigendum 2:2012-02. 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 grammar rules

d) A reference implementation.

NOTE — The scope, expressed in clause 1, Scope, of ISO/IEC 13211–1:1995 applies 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 alternative:** A compound term with principal functor `(;)/2` or with principal functor `('|')/2` with each argument being a body (of a grammar-rule).

**3.2 body (of a grammar-rule):** See `grammar-rule-body`

**3.3 clause-term:** A read-term `T.` in Prolog text where `T` has neither principal functor `(:-)/1` nor principal functor `(-->)/2.` (This definition replaces subclause 3.33 of ISO/IEC 13211–1).

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

**3.5 cover, a terminal-sequence by a non-terminal (resp. a body):** A terminal sequence is covered by a non-terminal (resp. a body) if the non-terminal (resp. the body) generates the terminal sequence. Alternatively if the non-terminal (resp. body) parses the terminal sequence.

**3.6 definite clause grammar:** A definite clause grammar is a set of definite clause non-terminal definitions.

**3.7 definite clause non-terminal definition:** A definite clause non-terminal definition is a sequence of grammar-rules.

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

**3.9 generating (wrt a non-terminal and a definite clause grammar):** Producing a terminal-sequence of that definite clause grammar, obeying semi-contexts, if any.

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

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

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

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

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

**3.15 grammar-rule-body:** A compound term which forms, or is in the form of, the second argument of a grammar-rule. A grammar-body-sequence, or an alternative, or a grammar-body-not, or a grammar-body-element, or a grammar-goal.

**3.16 grammar-rule-head:** 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 semicontext (cf Definition 3.22).

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

**3.18 non-terminal (of a grammar-rule):** A callable term (cf. ISO/IEC 13211–1, Definitions 3.25), i.e., an atom or a compound term, that denotes a non terminal symbol of a grammar rule.

**3.19 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 7.13.4).

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

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

**3.22 semicontext:** A terminal-sequence occurring optionally after the non-terminal of a grammar-rule-head, constraining parsing (respectively generation) by this grammar rule.

**3.23 steadfastness of a goal wrt. an argument** Goal `G` is steadfast with respect to argument `n` of its sequence of arguments, if for any term `T` that is the `n`th argument in the goal, and the goal `Gnw` that results by replacing `T` by a new variable `Vnw` the execution (cf. ISO/IEC 13211–1, subclause 7.7.1) of `G` and `(Gnw, Vnw=T)` is the same.

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

**3.25 terminal-sequence:** A list (cf. ISO/IEC 13211–1, subclauses 3.99, 6.3.5 and 6.3.1.3) whose first argument, if any, is a terminal (of a grammar), and the second argument, if any, is a terminal-sequence.

**3.26 terminal-sequence, comprehensive:** A terminal sequence containing a prefix, and the prefix covered (cf. Definition 3.5) by a grammar-rule-body, i.e. accepted resp. generated by `phrase/3` (cf 8.18.1) .

**3.27 terminal-sequence, remaining:** The rest of a comprehensive terminal-sequence without the leading terminal-sequence covered (cf. Definition 3.5) by a grammar-rule-body.

**3.28 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 the corresponding subclause 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 subclause in 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 in ISO/IEC 13211–1 Prolog.

## 5.5 Extensions

*The corresponding subclause in 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.

A Prolog processor may support additional grammar control constructs, beyond the required ones by this standard (cf. 7.14). These additional control constructs must be treated as non-terminals by a Prolog processor working in a strictly conforming mode (see 5.1e).

NOTE — Examples for additional grammar control constructs include *soft-cuts* and control constructs that enable the use of grammar rules stored on encapsulation units other than modules, such as objects.

### 5.5.2 Predefined operators

*See subclause 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 subclause in 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 subclause extends, with the specified number, the corresponding ISO/IEC 13211–1 subclause.*

### 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 parsing `Si` with a grammar rule.

## 6.2 Prolog text and data

*The first paragraph of this subclause 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 subclause in 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 the exception of the operator syntax (cf 6.3). For the semantic changes see 7.4.2 of this TR. Whenever directives are applicable to non-terminals, the non-terminal indicators (cf 7.13.4), as arguments of these directives, shall be used like predicate indicators for the predicates, resulting from expanding these non-terminals.*

NOTE — The directives `dynamic/1, multifile/1 and discontiguous/1` are applicable to non-terminal indicators.

#### 6.2.1.2 Clauses

*The corresponding subclause in 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 not `(:-)/1` | |
| Condition: | The principal functor of `c` is not `(-->)/2` | |

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

*The following subclause modifies, 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 — Subclause 10 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 Semicontexts

| | `semicontext term =` | `term` |
|---|---|---|
| Abstract: | *sc* | *sc* |
| Priority: | 1201 | |
| Condition: | semicontext term is a list | |

| | `semicontext =` | `semicontext term` |
|---|---|---|
| Abstract: | *s* | *s* |

NOTE — Subclause 10 of this TR, `dcg_rule/4`, first clause, defines how a semicontext in a grammar rule is expanded when Prolog text is prepared for execution.

## 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 (7.13.4) 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.

### 7.4.4 Grammar rules

A grammar rule term in Prolog text (6.2.1.3) enables suitable clauses to be added to the database.

The non-terminal indicator NT//N of the non-terminal of the grammar-rule-head shall not be the non-terminal-indicator of a grammar control construct. Further the predicate indicator NT/M where M is N + 2, and NT//N is the non-terminal indicator of the non-terminal of the grammar-rule-head shall not

be the predicate indicator of a built-in predicate or a control construct.

During preparation for execution the Prolog processor converts grammar rule terms to Prolog procedures of the database. Section 10 of this TR defines a correspondence between grammar rule terms and suitable clauses of a procedure in the database.

## 7.5 Database

### 7.5.1 Preparing a Prolog text for execution

If a Prolog grammar rule with non-terminal indicator NT//N is prepared for execution, and a Prolog clause with predicate-indicator NT/M, where M is N + 2, is already part of the extended database, or vice versa, then the effect of this preparation for execution shall be implementation dependent.

## 7.13 Grammar rules

### 7.13.1 Terminals and non-terminals

In grammar rule bodies, one or more terminals are represented by terms directly contained in lists in order to distinguish them from non-terminals. The empty terminal sequence (empty list) is possible. Non-terminals are represented by callable terms.

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

#### 7.13.1.1 Example

A simple grammar consisting of 11 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 by:

```
sentence --> noun_phrase, verb_phrase.
verb_phrase --> verb.
noun_phrase --> article, noun.
noun_phrase --> article, adjective, noun.
article --> [the].
```

```
adjective --> [nice].
noun --> [dog].
noun --> [cat].
verb --> [runs].
verb --> [barks].
verb --> [bites].
```

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

### 7.13.2 Format of grammar rules

A grammar rule has the format:

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

where `GRHead`, the grammar-rule-head (cf. Definition 3.15), can be rewritten by `GRBody`, its grammar-rule-body (cf. Definition 3.14). The head and the body of grammar rules are constructed from *terminals* and *non-terminals*, including special non-terminals, the *grammar control constructs*. The grammar-rule-head is a non-terminal, or a non-terminal, followed by a terminal-sequence (a *semi-context*, see 7.13.3):

```
NonTerminal --> GRBody.

NonTerminal, Semicontext --> GRBody.
```

If `NonTerminal` is a grammar control construct its effect shall be implementation dependent.
The effect of a `Semicontext` which is not a terminal-sequence shall be implementation dependent.

The control constructs that may be used in a body are described in subclause 7.14. An empty body is represented by an empty terminal sequence:

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

NOTE — There is no `(-->)/1` form for grammar rules.

### 7.13.3 Semicontext

#### 7.13.3.1 Description

A *semicontext* is a terminal-sequence (see 3.25), which follows, separated by a comma, the non-terminal of the head of a grammar rule (see 3.16). The terminals of the semicontext make up a prefix of the remaining terminal-sequence.

#### 7.13.3.2 Examples

Assume we need rules to *look-ahead* one or two tokens that would later be next consumed. 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 semicontext:

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

After preparation for execution this may occur in the database as follows.

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

NOTES

1 In case of parsing with `phrase1`, as soon as `phrase2` and `phrase3` have 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 further parsing is constrained by the semicontext.

2 The concepts *comprehensive terminal-sequence* resp. *remaining terminal-sequence* are often called *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 there will be a remainder after the parsing step. The inverse case, generating sentences by expanding grammars, where the comprehensive terminal-sequence is the output list, is ignored by such wording.

3 There are cases, where the remaining terminal-sequence is the comprehensive terminal-sequence. See, e.g. the following grammar rule. There maybe a trailing terminal-sequence, however, as the following example shows.

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

which may be expanded by preparation for execution to:

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

This non-terminal `nt` represents an empty terminal sequence (cf. 7.14.1), but constrains further parsing to take place with `word` as next token.

4 It should be noted that `phrase/2` (cf 8.18.1.3) cannot succeed when applied to a grammar rule, whose head contains a non empty semicontext, as in the case above.

5 Some processors allow a cut in the semicontext; e.g.

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

Moving this cut to the end of the grammar body, c.f. `a, [word] --> b, !.` leads to identical execution. Thus this TR does not permit a cut in the semicontext.

### 7.13.4 Non-terminal indicator

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

The non-terminal indicator `//(A, N)` indicates the non-terminal of the head of a grammar rule where `A` is an atom, representing the non-terminal, and `N` is its arity, a non-negative integer

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` or as `(A)//N` depending on whether or not A is an operator (cf. 7.1.6.6 of 13211–1).

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 in a predicate property . In particular, using non-terminal indicators in predicate directives allows the details of the expansion of grammar rules into Prolog clauses to be abstracted.

#### 7.13.4.1 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`.

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

Thus grammar rules for `sentence//0` may be distributed over several files.

## 7.14 Grammar control constructs

This definition of each grammar control construct gives its logical meaning and the procedural effects, if any, of executing it wrt. its arguments after preparing it for execution.

Expansion of grammar control constructs is not simply a replacement by Prolog control constructs. For the expansion of every grammar control construct there is a formal definition in subclause 10.

The correspondence between the following subclauses and the corresponding formal definitions is given by the argument of the clauses of predicate `dcg_constr/1` or, respectively, by the principal functor of the first argument of the clauses of predicate `dcg_cbody/4` in subclause 10

After preparation for execution of Grammar Rules, named "Grammar Rule expansion" or "expansion" for short, the non-terminals, with exception of phrase//1, result in control constructs, respectively built-in predicates of ISO/IEC 13211-1 Prolog. Grammar Rule expansion is defined by using the built-in predicate `phrase/3` (see subclause 8.18.1) and subclause 10.

The following subclauses explicate the linkages between the terminal sequences upon expansion of the control constructs.
The meaning of `phrase/2` is sometimes defined using if and sometimes defined using iff depending on whether or not semicontext makes a difference. For `phrase/3`, only iff is used.

The procedural effects are described using the logical expansion (Section 10).

### 7.14.1 `[]//0` – empty terminal-sequence

#### 7.14.1.1 Description

`phrase([],S0,S)` is true iff `S0=S`.

### 7.14.2 `('.')//2` – terminal sequence

#### 7.14.2.1 Description

`('.')` used as a non-terminal `('.')//2` separates its first argument, the terminal on its left hand side from the second argument, the terminal sequence on its right hand side.

### 7.14.3 (',')//2 – concatenation

#### 7.14.3.1 Description

The grammar body element (A,B) describes the concatenation of A and B.

phrase((A,B), S) is true if S is the concatenation of S1 and S2
where phrase(A, S1) and phrase(B, S2) are true.

More precisely taking semicontext into account,
phrase((A,B), S0,S), is true iff
phrase(A, S0,S1), phrase(B, S1,S) is true.

### 7.14.4 (;)//2 – alternative

#### 7.14.4.1 Description

The grammar body element (A;B) describes the alternative of A and B.

phrase((A;B), S) is true iff ( phrase(A, S) ; phrase(B, S) ) is true.

More precisely, taking semicontext into account, phrase((A;B), S0,S), is true iff ( phrase(A, S0,S) ; phrase(B, S0,S) ) is true.

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 may lead 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)
   ).
```

### 7.14.5 `(;)//2` with `(->)//2` – if-then-else

#### 7.14.5.1 Description

NOTE – `(;)//2` (cf. 7.14.4) serves two different functions depending on whether or not its first argument is a compound term with grammar control construct `(->)//2`. See 7.14.4 for the use of `(;)//2` for `alternative`, when the first argument of `(;)//2` does not immediately contain a grammar control construct `(->)//2`.

#### 7.14.5.2 Description

The grammar body element `( A -> B ; C )` describes one of `( A, B )` or `C`

`phrase(( A -> B ; C ), S)` is true if
if `phrase(A, S,_)` is true
then
`phrase((A, B), S)` is true
for the first solution/answer of `phrase(A, S,_)`
else
`phrase(C, S)` is true.

More precisely taking semicontext into account,
`phrase(( A -> B ; C ), S0,S)` is true iff
`( phrase(A, S0, S1) -> phrase(B, S1, S) ; phrase(C, S0, S) )`
is true.

#### 7.14.5.3 Examples

```
phrase( ( ("1"|"2") -> "3" ; "4" ), "13") succeeds
phrase( ( ("1"|"2") -> "3" ; "4" ), "23") succeeds
phrase( ( ("1"|"2") -> "3" ; "4" ), "4") succeeds
phrase( ( ("1"|"2") -> "3" ; "4" ), Xs) succeeds only once unifying Xs
with "13".
phrase( ( ("1"|"2") -> "3" ; "4" ), [X]) fails
```

### 7.14.6 `('|')//2` – second form of alternative

#### 7.14.6.1 Description

The grammar body element `(A|B)` describes the alternative of `A` and `B`.

`phrase((A|B), S)` is true iff `( phrase(A, S) ; phrase(B, S) )` is true.

More precisely taking semicontext into account, `phrase((A|B), S0,S)`, is true iff `( phrase(A, S0,S) ; phrase(B, S0,S) )` is true.

`'|'` used as a non-terminal `('|')//2` has the same behaviour as `(;)//2`, when used for an alternative. See subclause 7.14.4. The use of `('|')//2` instead of `(;)//2` in an if-then-else, cf 7.14.5, shall be implementation-dependent.

### 7.14.7 {}//1 – grammar-body-goal

#### 7.14.7.1 Description

The grammar body element {G} describes the empty sequence.

More precisely taking semicontext into account,
`phrase({G}, S0, S)` is true iff `( G, S0 = S )` is true.

The non-terminal {G}, with G a Prolog goal, according to ISO/IEC 13211-1:1995, can appear at any place of a non-terminal inside a grammar-rule-body. After expansion the braces are omitted, the goal G is unchanged. On execution G is executed like any Prolog goal.
If G immediately contains a cut ('!'), this is handled like a grammar-body-cut (cf. 7.14.10), i.e. the effect of the cut extends outside the non-terminal {G}.

#### 7.14.7.2 Example

```
phrase({true}, nonlist, S)
```

succeeds, unifying `S` with `nonlist`.

### 7.14.8 call//1

#### 7.14.8.1 Description

The grammar body element `call//1` describes the use of a Prolog predicate that expects two additional arguments.

`phrase(call(G_2), S)` is true iff `call(G_2, S, [])` is true.

More precisely taking semicontext into account, `phrase(call(G_2), S0,S)` is true iff `call(G_2, S0,S)` is true.

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).
```

### 7.14.9 `phrase//1`

#### 7.14.9.1 Description

The grammar body element `phrase(NonTerminal)` describes `NonTerminal`.

`phrase(phrase(NonTerminal), S )` is true iff `phrase(NonTerminal, S,[])` is true.

More precisely taking semicontext into account,
`phrase(phrase(NonTerminal), S0,S)` is true iff
`phrase(NonTerminal,S0,S)` is true.

For a definition of the built-in predicate `phrase/3` see subclause 8.18.1.

### 7.14.10 `!//0` – grammar-body-cut

#### 7.14.10.1 Description

The grammar body `!` describes the empty terminal sequence.

`phrase(!, S)` is true iff `S = []` is true.

`phrase(!, S0,S)` is true iff `S0 = S` is true

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

### 7.14.11 `(\+)//1` – grammar-body-not

The presence of `(\+)//1` in grammar rules shall be implementation dependent.

#### 7.14.11.1 Description

If present in a `(\+)//1` in grammar rules shall be defined as follows:

The grammar body element `\+ GRBody` describes the empty terminal sequence.

`phrase(\+ GRBody, S)` is true iff
`\+ phrase(GRBody ,S,_), S = []` is true.

`phrase(\+ GRBody, S0,S)` is true iff
`( \+ phrase(GRBody ,S0,_), S0 = S )` is true.

Implementations conforming to this TR 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.
```

### 7.14.12 `(->)//2` - if-then

The effect of `(->)//2` in grammar rules except in the first argument of an alternative (cf. 7.14.5) shall be implementation dependent. If present in the processor, it is defined as follows.

#### 7.14.12.1 Description

The grammar body element `A -> B` describes one of `( A, B )`.

`phrase((A -> B), S)` is true iff `phrase((A -> B ; fail), S)` is true.

`phrase((A -> B), S0,S)` is true iff `phrase((A -> B ; fail), S0,S)` is true.

## 7.15 Executing clauses expanded from grammar rules

If a grammar rule to be prepared for execution has a non-terminal indicator NT//N, and NT is the name of the predicate indicator NT/M, with M is N + 2, of a built-in predicate, the result of expansion and the behaviour of the prepared grammar rule on execution is implementation dependent. This does not hold for the required non-terminals expanding to built-in predicates defined in 7.14.

When the database does not contain a procedure, prepared for execution from one or more grammar rules with non-terminal indicator NT//N during execution of a goal, prepared for execution from a non-terminal with non-terminal indicator NT//N, the behaviour of the processor shall be as follows:

If the error handling of the processor is standard conforming as specified in subclause 7.7.7 of ISO/IEC 13211–1, then the error term as specified in subclause 7.7.7b of ISO/IEC 13211–1 when the flag `unknown` is set to `error` shall be:

```
existence_error(procedure, NT/M)
```

If the error handling of the processor supports definite clause grammar errors, then the error term shall be:

```
existence_error(grammar_rule, NT/M)
```

In other cases the behaviour shall be implementation specific.

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 terminal sequences using grammar rules is defined in subclause 8.18.1. Grammar rules are expanded there 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

## 8.18 Grammar rule built-in predicates

### 8.18.1 phrase/3, phrase/2

#### 8.18.1.1 Description

In the absence of semicontexts `phrase(GRBody, S)` is true if S is a phrase covered by the non-terminal or more generally the grammar-rule-body `GRBody`.

In the absence of semicontexts `phrase(GRBody, S0, S))` is true if `P` is a phrase covered by the non-terminal `GRBody` and `append(P,S, S0)` is true.

The description of `phrase(GRBody, S0, S))` in the presence of semicontexts exploits the decomposition of `GRBody` into a compound term composed from non terminals given in subclause 7.14 so that it is enough to provide a recursive description assuming that `GRBody` is a non terminal.

In the presence of semicontexts `phrase(NonTerminal, S0,S)` is true
if there is a grammar rule
`Nonterminal, Semicontext --> GRbody1`
and recursively `phrase(GRbody1, S0,S1)` is true
and `append(Semicontext, S1, S)` is true.
else
there is no semicontext and there is a grammar rule
`Nonterminal --> GRbody1`
and recursively `phrase(GRbody1, S0,S)` is true.

Procedurally `phrase(GrBody, S0, S)` is executed as
`dcg_body(GRBody, S0, S, Goal), call(Goal)`
where `dcg_body/4)` is described in clause 10.

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

Execution of the predicate `phrase/3` serves two goals: Firstly the final expansion(of a grammar rule) (cf. Definition 3.7), when this has not taken place earlier, i.e. preparation for execution of its body and arguments; thereafter, secondly, the execution of the resulting Prolog goals.

NOTE 1 — If the simple grammar of example 7.14.1.1 is prepared for execution.

Then with

```
GRBody: non-terminal:  noun_phrase
S0: comprehensive terminal-sequence: [the, dog, barks]
S:  remaining terminal-sequence: [barks]
```

`phrase(noun_phrase, [the, dog, barks], [barks])` is true.

#### 8.18.1.2 Template and modes

```
phrase(+grammar-rule-body, ?comprehensive-terminal-sequence,
?remaining-terminal-sequence)
```

For definitions of `comprehensive-terminal-sequence and remaining-terminal-sequence` see Definitions 3.25 resp. 3.26, for `grammar-rule-body` see Definition 3.14.

#### 8.18.1.3 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 accepted respectively generated.

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

#### 8.18.1.4 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.

#### 8.18.1.5 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]).
yes

| ?- phrase(sentence, Sentence).

Sentence = [the, boy, likes]
yes

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

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

# 9 Evaluable functors

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

# 10 Logical Expansion

```
:- op(1105,xfy,'|').

% This program uses append/3 as defined in the Prolog prologue.

% Expands 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).

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

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

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 \= ( _ -> _ ),
   NonTerminal \= ( \+ _ ),
   dcg_non_terminal(NonTerminal, S0, S, Goal).

% The following constructs in a grammar rule body
```

```
% are defined in the corresponding subclauses.

dcg_constr([]).        % 7.14.1
dcg_constr([_|_]).     % 7.14.2 - terminal sequence
dcg_constr(( _, _ )).  % 7.14.3 - concatenation
dcg_constr(( _ ; _ )). % 7.14.4 - alternative
                       % 7.14.5 - if-then-else
dcg_constr(( _'|'_ )). % 7.14.6 - alternative
dcg_constr({_}).       % 7.14.7
dcg_constr(call(_)).   % 7.14.8
dcg_constr(phrase(_)). % 7.14.9
dcg_constr(!).         % 7.14.10
% dcg_constr(\+ _).    % 7.14.11 - not (existence implementation dep.)
% dcg_constr((_->_)).  % 7.14.12 - if-then (existence implementation dep.)

% The principal functor of the first argument indicates
% the construct to be expanded.

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_body(GREither, S0, S, Either),
   dcg_body(GROr, S0, S, Or).
dcg_cbody(( GRCond ; GRElse ), S0, S, ( Cond ; Else )) :-
   subsumes_term(( _GRIf -> _GRThen ), GRCond),
   dcg_cbody(GRCond, S0, S, Cond),
   dcg_body(GRElse, S0, S, Else).
dcg_cbody(( GREither '|' GROr ), S0, S, ( Either ; Or )) :-
   dcg_body(GREither, S0, S, Either),
   dcg_body(GROr, S0, S, Or).
dcg_cbody({Goal}, S0, S, ( Goal, S0 = S )).
dcg_cbody(call(Cont), S0, S, call(Cont, S0, S)).
dcg_cbody(phrase(Body), S0, S, phrase(Body, S0, S)).
dcg_cbody(!, S0, S, ( !, S0 = S )).
dcg_cbody(\+ GRBody, S0, S, ( \+ phrase(GRBody,S0,_), S0 = S )).
dcg_cbody(( GRIf -> GRThen ), S0, S, ( If -> Then )) :-
   dcg_body(GRIf, S0, S1, If),
   dcg_body(GRThen, S1, S, Then).
```