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

Editor: Paulo Moura
pmoura@di.ubi.pt

May 28, 2010

Introduction

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

Grammar rules provide convenient and simple functionality for parsing and processing text in a variety of languages. They have been implemented in many Prolog systems. As such, they are deemed an worthy extension to the ISO/IEC 13211 Prolog standard. Parsing and processing text with grammar rules is defined in sections 8.1.1 and 11.2. In brief, grammar rules can be expanded into Prolog clauses, which allows us to map parsing a grammar rule body into executing a goal given a set of predicate clauses. See section 7.7 of ISO/IEC 13211–1 for details.

This TR is written as an extension to the ISO/IEC 13211–1 Prolog standard, adopting a similar structure. Specifically, this TR either adds new sections and clauses to, or modifies the reading of existing clauses on ISO/IEC 13211–1.

This TR provides reference implementations for the specified built-in predicates and for a translator from grammar rules into Prolog clauses. In addition, it includes a comprehensive set of tests to help users and implementers check for compliance of Prolog systems. The source code of these reference implementations may be used without restrictions for any purpose.

This draft may contain in several places informative text, type-set in *italics*. Such informative text is used for editorial comments deemed useful during the development of this draft and may not be included in the final version.

Previous editors and draft documents

- 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 I've only included people present at the ISO meetings collocated with the ICLP (2005, 2006, and 2007) and the authors of the two drafts cited above, and Richard as I have included here some contributions from him that I found on the net.

- 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)
- Péter Szabó (Hungary)
- Péter Szeredi (Hungary)
- Richard O'Keefe (NZ)
- Roger Scowen (UK)
- Tony Dodd (UK)
- Ulrich Neumerkel (Austria)
- Vítor 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. 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

d) References implementations and tests for the specified built-in predicates and for a grammar rule translator

NOTE — This part of ISO/IEC 13211 will supplement ISO/IEC 13211–1:1995.

2 Normative references

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

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 or the first argument of the built-in predicates `phrase/3` and `phrase/2` (8.1.1). A grammar-body-sequence, or a grammar-body-alternatives, or a grammar-body-choice, 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`.

3.3 definite clause grammar: A sequence of grammar rules.

3.4 grammar-body-alternatives: A compound term with principal functor `(;)/2` and each argument being a body (of a grammar-rule).

3.5 grammar-body-choice: A compound term with principal functor `(->)/2`, the first argument is a body (of a grammar-rule), and the second argument is a grammar-body-alternatives.

3.6 grammar-body-cut: The atom `!`.

3.7 grammar-body-element: A grammar-body-cut, or a grammar-body-goal, or a grammar-body-nonterminal, or a sequence of grammar-body-terminals.

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

3.9 grammar-body-nonterminal: A non-terminal (of a grammar).

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

3.11 grammar-body-terminals: A sequence of terminals.

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

3.13 grammar-rule-term: A read-term `T.` in Prolog text where `T` is a grammar-rule.

3.14 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` the first argument is a non-terminal (of a grammar), and the second argument is a sequence of terminals.

3.15 new variable with respect to a term T: A variable that is not an element of the variable set of `T` (see also 7.1.6.2 of ISO/IEC 13211-1).

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

3.17 non-terminal-indicator: A compound term `A//N` where `A` is an atom and `N` is a non-negative integer, denoting one particular grammar-rule non-terminal.

3.18 sequence of terminals: The Prolog atom `[]`, or a compound term whose principal functor is `(.)/2`, the first argument is a terminal (of a grammar), and the second argument is a sequence of terminals.

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

3.20 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 on 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 lists of terminals

This TR uses variables named `S0`, `S1`, ..., `S` to represent the list of terminals 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 S_i represents the initial list of terminals, the variable S_{i+1} will represent the remaining list of terminals 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*		

6.1 Directives

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

6.2 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 not `(:-)/1`	
Condition:	The principal functor of `c` is not `(-->)/2`	

NOTE — Subclauses 7.5 and 7.6 defines how each clause becomes part of the database.

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

6.3 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 — Section 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.

6.3 Terms

NOTE — The operator `-->/2`, specified in section 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 section extends, with the specified number, the corresponding ISO/IEC 13211–1 section:

7.4.2.1 `dynamic/1`

Adds the following note: The argument of this directive may also be a grammar rule non-predicate indicator, a grammar rule non-predicate indicator sequence, or a grammar rule non-predicate indicator list specifying that each user-defined grammar rule non-terminal indicated by the directive argument is multifile.

7.4.2.2 `multifile/1`

Adds the following note: The argument of this directive may also be a grammar rule non-predicate indicator, a grammar rule non-predicate indicator sequence, or a grammar rule non-predicate indicator list specifying that each user-defined grammar rule non-terminal indicated by the directive argument is dynamic.

7.4.2.3 `discontiguous/1`

Adds the following note: The argument of this directive may also be a grammar rule non-predicate indicator, a grammar rule non-predicate indicator sequence,

or a grammar rule non-predicate indicator list specifying that each user-defined grammar rule non-terminal indicated by the directive argument is discontiguous.

7.13 Predicate properties

The following optional property is added to the list of predicate properties:

- `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

Terminals are represented by terms enclosed in lists in order to distinguish them from non-terminals (string notation may be used as an alternative to lists when terminals are characters; see sections 6.3.7 and 6.4.6 of ISO/IEC 13211–1). Non-terminals are represented by callable terms.

NOTE — In the context of a grammar rule, *terminals* represent words or tokens of some language and *non-terminals* represent sequences of words (see, respectively, sections 3.18 and 3.16).

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 *terminals* and *non-terminals*. The head of a grammar rule is a non-terminal or the conjunction of a non-terminal and a list of terminals (a *push-back list*, see 7.14.3):

```
NonTerminal --> GRBody.

NonTerminal, PushBackList --> GRBody.
```

The control constructs that may be used on a grammar rule body are described later, in section 7.14.6. 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 Push-back lists

A *push-back list* is a list of terminals on the left-hand side of a grammar rule (see 3.14). A push-back list contains terminals that in the output list are prefixed to the list obtained from the input list by successful application of the body of the grammar rule.

7.14.3.1 Examples

For example, assume that we need rules to *look-ahead* one or two tokens that would be consumed next. This could be easily accomplished by the following two 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.

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 sections 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 wherever a predicate indicator as defined in ISO/IEC 13211–1 can appear. 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`. Assuming a `public/1` directive for declaring predicate scope, we could write:

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

in order to be possible to use grammar rules for the non-terminal `sentence//0` outside its encapsulation unit.

7.14.5 Prolog goals in grammar rules

In the body of grammar rules, curly brackets enclose a sequence of Prolog goals that are executed when the grammar rule is processed.

NOTE — The ISO/IEC 13211–1 Prolog standard defines, in section 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.

7.14.5.1 Examples

Consider, for example, the following grammar rule:

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

This rule 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.

7.14.6 Control constructs and built-in predicates supported by grammar rules

The following control constructs specified in the ISO/IEC 13211–1 Prolog standard may be used in the body of grammar rules: `','/2`, `';'/2`, `->/2`, and `!/0`.

The following built-in predicates specified in the ISO/IEC 13211–1 Prolog standard may be used in the body of grammar rules: `\+/1`.

The `:/2` control construct specified in the ISO/IEC 13211–2 Prolog standard may be used in the body of grammar rule (see 11.1.1).

The `call/1` Prolog control construct specified in the ISO/IEC 13211–1 Prolog standard (sections 7.8) shall be interpreted as the `call//1` built-in non-terminal (see 8.1.2) when used in the body of grammar rule.

The following Prolog control constructs and built-in predicates derived from control constructs specified in the ISO/IEC 13211–1 Prolog standard (sections 7.8 and 8.15) shall not be recognized as control constructs when used in a grammar rule body: `true/0`, `fail/0`, `repeat/0`, `once/1`, `catch/3`, and `throw/1`. When appearing in the place of a non-terminal, these Prolog control constructs and built-in predicates must be interpreted as non-terminals.

A Prolog implementation 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 implementation working on a strictly conforming mode (see 5.1e).

7.14.7 Executing procedures expanded from grammar rules

When the database does not contain a grammar rule for a non-terminal required for the grammar rule body we are trying to execute, it is recommended, but not mandatory, that the error term specified in clause 7.7.7b of ISO/IEC 13211–1 when the flag `unknown` is set to `error` would be:

```
existence_error(procedure, GRI)
```

where `GRI` is the grammar rule non-terminal indicator for which no grammar rule is available.

NOTE — Implementers should report errors at the same abstraction level as grammar rules whenever practical.

8 Built-in predicates

8.1 Grammar rule built-in predicates and non-terminals

8.1.1 phrase/3, phrase/2

8.1.1.1 Description

`phrase(GRBody, S0, S)` is true iff the grammar rule body `GRBody` parses, according to the currently defined grammar rules, the list of terminals `S0` unifying `S` with the list of the remaining terminals.

When `GRBody` contains a `!` as a subgoal, the effect of `!` shall not extend outside `GRBody`.

Procedurally, `phrase(GRBody, S0, S)` is executed by calling the Prolog goal corresponding to the expansion of the grammar rule body `GRBody` with the terminal lists `S0` and `S`, according to the logical expansion of grammar rules defined in section 10.

Calling the `phrase/2-3` built-in predicates shall be considered the only safe way to use grammar rules from predicate clauses. Programmers are advised against directly calling the implementation-dependent predicates generated by the compilation of the grammar rules.

8.1.1.2 Template and modes

```
phrase(+callable_term, ?list, ?list)
```

8.1.1.3 Errors

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

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

Type-checking of the second and third argument is implementation-dependent. When performed, the implementation shall conform with the following specification:

c) `S0` is neither a partial list nor a list
— `type_error(list, S0)`

d) `S` is neither a partial list nor a list
— `type_error(list, S)`

8.1.1.4 Bootstrapped built-in predicates

The built-in predicate `phrase/2` provides similar functionality to `phrase/3`. The goal `phrase(GRBody, Input)` is true when all tokens in the input list are consumed and recognized:

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

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].

sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.
```

```
noun_phrase --> noun.

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

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

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

yes

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

yes

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

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

8.1.2 call//1

8.1.2.1 Description

`call(Closure)` is true iff the `call(Goal)` is true where `Goal` is a term constructed by extending the arguments of `Closure` with the input list of terminals and the list of the remaining terminals.

The `call//1` built-in non-terminal provides controlled access to the input list of terminals and to the list of the remaining terminals processed by the grammar rule containing the call.

8.1.2.2 Template and modes

`call(+callable_term)`

8.1.2.3 Errors

a) `Closure` is a variable
— `instantiation_error`

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

9 Evaluable functors

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

10 Logical expansion of grammar rules

This section extends, with the specified number, the ISO/IEC 13211–1 Prolog standard:

This section presents a logical view for the expansion of grammar rules into Prolog clauses, starting with a description of the used notation.

10.1 Notation

The terms `S0` and `S` represent, respectively, the input list of terminals and the remaining list of terminals after parsing using a grammar rule. Variables named S_i represent intermediate parsing states, as explained in section 6.1.3.

The term $E_{Type}(T, S_i, S_{i+1})$ denotes an expansion of type *Type* of a term T, given, respectively, the input and output lists of terminals S_i and S_{i+1}

Four types of expansion rules are used, denoted by the terms: E_{rule} (expansion of grammar rules), E_{body} (expansion of grammar rule bodies), $E_{terminals}$ (expansion of grammar rule terminals), and $E_{non\_terminal}$ (expansion of grammar rule non-terminals).

The symbol ≡ is used to link an expansion rule with its resulting Prolog term or with another expansion rule.

10.2 Expanding a grammar rule

Grammar rules with a push-back list:

E_{rule}`((NonTerminal, Terminals --> GRBody), S0, S)` ≡ `Head :- Body`

where:

$E_{non\_terminal}$`(NonTerminal, S0, S)` ≡ `Head`
E_{body}`(GRBody, S0, S1)`, $E_{terminals}$`(Terminals, S, S1)` ≡ Body

Grammar rule with no push-back list:

E_{rule}`((NonTerminal --> GRBody), S0, S)` ≡ `Head :- Body`

where:

$E_{non\_terminal}$`(NonTerminal, S0, S)` ≡ `Head`
E_{body}`(GRBody, S0, S)` ≡ `Body`

10.3 Expanding a grammar rule non-terminal

$E_{non\_terminal}$`(NonTerminal, S0, S)` ≡ `Head`

where:

```
NonTerminal =.. NonTerminalUniv,
append(NonTerminalUniv, [S0, S], HeadUniv),
Head =.. HeadUniv
```

(see section 11.3 for the definition of the auxiliary predicate `append/3`)

10.4 Expanding a terminal list

List of terminals, either a push-back list or a grammar rule body goal:

$E_{terminals}$`([], S0, S)` ≡ `S0 = S`
$E_{terminals}$`([T| Ts], S0, S)` ≡ `S0 = [T| Tail]`

where:

$E_{terminals}$`(Ts, S1, S)` ≡ `Tail`

where `S1` is a new variable with respect to the term `[T| Ts]`.

An alternative definition, given a list of terminals `Terminals` is:

$E_{terminals}$`(Terminals, S0, S)` ≡ `S0 = List`

where:

```
append(Terminals, S, List)
```

(see section 11.3 for the definition of the auxiliary predicate `append/3`)

10.5 Expanding a grammar rule body

Non-instantiated variable on a grammar rule body:

E_{body}`(Var, S0, S)` ≡ `phrase(Var, S0, S)`

If-then-else construct on the body of a grammar rule:

E_{body}`((GRIf -> GRThen; GRElse), S0, S)` ≡ `If -> Then; Else`

where:

E_{body}`(GRIf, S0, S1)` ≡ `If`
E_{body}`(GRThen, S1, S)` ≡ `Then`
E_{body}`(GRElse, S0, S)` ≡ `Else`

If-then construct on the body of a grammar rule:

E_{body}`((GRIf -> GRThen), S0, S)` ≡ `If -> Then`

where:

E_{body}`(GRIf, S0, S1)` ≡ `If`
E_{body}`(GRThen, S1, S)` ≡ `Then`

Disjunction on the body of a grammar rule:

E_{body}`((GREither; GROr), S0, S)` ≡ `Either; Or`

where:

E_{body}`(GREither, S0, S)` ≡ `Either`
E_{body}`(GROr, S0, S)` ≡ `Or`

Conjunction on the body of a grammar rule:

E_{body}`((GRFirst, GRSecond), S0, S)` ≡ `First, Second`

where:

E_{body}`(GRFirst, S0, S1)` ≡ `First`
E_{body}`(GRSecond, S1, S)` ≡ `Second`

Cut on the body of a grammar rule:

E_{body}`(!, S0, S)` ≡ `!, S0 = S`

Curly-bracketed term on the body of a grammar rule:

E_{body}`({}, S0, S)` ≡ `S0 = S`
E_{body}`({Goal}, S0, S)` ≡ `Goal, S0 = S`

when `Goal` is a non-variable term and:

E_{body}`({Goal}, S0, S)` ≡ `call(Goal), S0 = S`

when `Goal` is a Prolog variable.

Negation on the body of a grammar rule:

E_{body}`(\+ Body, S0, S)` ≡ `\+ Goal, S0 = S`

where:

E_{body}(Body, S0, S) ≡ Goal

List of terminals on the body of a grammar rule:

E_{body}(Terminals, S0, S) ≡ $E_{terminals}$(Terminals, S0, S)

Non-terminal on the body of a grammar rule:

E_{body}(NonTerminal, S0, S) ≡ $E_{non\_terminal}$(NonTerminal, S0, S)

11 Reference implementations

The reference implementations provided is this section do not preclude alternative or optimized implementations.

11.1 Grammar-rule translator

This section provides a reference implementation for a translator of grammar rules into Prolog clauses as specified in the ISO/IEC 13211–1 Prolog standard. The main idea is to translate grammar rules into clauses by adding two extra arguments to each grammar rule non-terminal, following the logical expansion of grammar rules, described in the previous section. The first extra argument is used for the input list of terminals. The second extra argument is used for the list of terminals in the input list not consumed by the grammar rule. This is a straight-forward solution. Nevertheless, compliance with this TR does not imply this specific translation, only compliance with the logical expansion, as specified in clause 10.

This translator includes error-checking code that ensures that both the input grammar rule and the resulting clause are valid. In addition, this translator attempts to simplify the resulting clauses by removing redundant calls to `true/0` and by folding unifications. In some cases, the resulting clauses could be further optimized. Other optimizations can be easily plugged in, by modifying or extending the `dcg_simplify/4` predicate. However, implementers must be careful to delay output unifications in the presence of goals with side-effects such as cuts or input/output operations, ensuring the steadfastness of the generated clauses.
Translating a grammar-rule non-terminal may result in a conflict with a built-in predicate. The behavior of the Prolog processor in this case is implementation dependent. However, if the Prolog processor raises an error for this case, it shall the same error generated when an attempt is made to redefine a built-in predicate.

```
% converts a grammar rule into a normal clause:
```

```
dcg_rule(Rule, Clause) :-
    dcg_rule(Rule, S0, S, Expansion),
    dcg_simplify(Expansion, S0, S, Clause).


dcg_rule((RHead --> _), _, _, _) :-
    var(RHead),
    throw(instantiation_error).

dcg_rule((RHead, _ --> _), _, _, _) :-
    var(RHead),
    throw(instantiation_error).

dcg_rule((_, Terminals --> _), _, _, _) :-
    var(Terminals),
    throw(instantiation_error).

dcg_rule((NonTerminal, Terminals --> GRBody), S0, S, (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), S0, S, (Head :- Body)) :-
    !,
    dcg_non_terminal(NonTerminal, S0, S, Head),
    dcg_body(GRBody, S0, S, Body).

dcg_rule(Term, _, _, _) :-
    throw(type_error(grammar_rule, Term)).


% translates a grammar goal non-terminal:

dcg_non_terminal(NonTerminal, _, _, _) :-
    \+ callable(NonTerminal),
    throw(type_error(callable, NonTerminal)).

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


% translates a list of terminals:
```

```
dcg_terminals(Terminals, _, _, _) :-
    \+ is_proper_list(Terminals),
    throw(type_error(list, Terminals)).

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


% translates a grammar rule body:

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

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

dcg_body((GREither; GROr), S0, S, (Either; Or)) :-
    !,
    dcg_body(GREither, S0, S, Either),
    dcg_body(GROr, S0, S, Or).

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

dcg_body(!, S0, S, (!, S0 = S)) :-
    !.

dcg_body({}, S0, S, (S0 = S)) :-
    !.

dcg_body({Goal}, S0, S, (call(Goal), S0 = S)) :-
    var(Goal),
    !.

dcg_body({Goal}, _, _, _) :-
    \+ callable(Goal),
    throw(type_error(callable, Goal)).

dcg_body({Goal}, S0, S, (Goal, S0 = S)) :-
    !.
```

```
dcg_body(\+ GRBody, S0, S, (\+ Goal, S0 = S)) :-
    !,
    dcg_body(GRBody, S0, S, Goal).

dcg_body([], S0, S, (S0=S)) :-
    !.

dcg_body([T| Ts], S0, S, Goal) :-
    !,
    dcg_terminals([T| Ts], S0, S, Goal).

dcg_body(NonTerminal, S0, S, Goal) :-
    dcg_non_terminal(NonTerminal, S0, S, Goal).


% simplifies the resulting clause:

dcg_simplify((Head :- Body), _, _, Clause) :-
    dcg_conjunctions(Body, Flatted),
    dcg_fold_left(Flatted, FoldedLeft),
    dcg_fold_pairs(FoldedLeft, FoldedPairs),
    (    FoldedPairs == true ->
         Clause = Head
    ;    Clause = (Head :- FoldedPairs)
    ).


% removes redundant calls to true/0 and flattens conjunction of goals:

dcg_conjunctions((Goal1 -> Goal2), (SGoal1 -> SGoal2)) :-
    !,
    dcg_conjunctions(Goal1, SGoal1),
    dcg_conjunctions(Goal2, SGoal2).

dcg_conjunctions((Goal1; Goal2), (SGoal1; SGoal2)) :-
    !,
    dcg_conjunctions(Goal1, SGoal1),
    dcg_conjunctions(Goal2, SGoal2).

dcg_conjunctions(((Goal1, Goal2), Goal3), Body) :-
    !,
    dcg_conjunctions((Goal1, (Goal2, Goal3)), Body).

dcg_conjunctions((true, Goal), Body) :-
    !,
```

```
    dcg_conjunctions(Goal, Body).

dcg_conjunctions((Goal, true), Body) :-
    !,
    dcg_conjunctions(Goal, Body).

dcg_conjunctions((Goal1, Goal2), (Goal1, Goal3)) :-
    !,
    dcg_conjunctions(Goal2, Goal3).

dcg_conjunctions(\+ Goal, \+ SGoal) :-
    !,
    dcg_conjunctions(Goal, SGoal).

dcg_conjunctions(Goal, Goal).


% folds left unifications:

dcg_fold_left((Term1 = Term2), true) :-
    !,
    Term1 = Term2.

dcg_fold_left(((Term1 = Term2), Goal), Folded) :-
    !,
    Term1 = Term2,
    dcg_fold_left(Goal, Folded).

dcg_fold_left(Goal, Goal).


% folds pairs of consecutive unifications (T1 = T2, T2 = T3):

dcg_fold_pairs((Goal1 -> Goal2), (SGoal1 -> SGoal2)) :-
    !,
    dcg_fold_pairs(Goal1, SGoal1),
    dcg_fold_pairs(Goal2, SGoal2).

dcg_fold_pairs((Goal1; Goal2), (SGoal1; SGoal2)) :-
    !,
    dcg_fold_pairs(Goal1, SGoal1),
    dcg_fold_pairs(Goal2, SGoal2).

dcg_fold_pairs(((T1 = T2a), (T2b = T3)), (T1 = T3)) :-
    T2a == T2b,
    !.
```

```
dcg_fold_pairs(((T1 = T2a), (T2b = T3), Goal), ((T1 = T3), Goal2)) :-
    T2a == T2b,
    !,
    dcg_fold_pairs(Goal, Goal2).

dcg_fold_pairs((Goal1, Goal2), (Goal1, Goal3)) :-
    !,
    dcg_fold_pairs(Goal2, Goal3).

dcg_fold_pairs(\+ Goal, \+ SGoal) :-
    !,
    dcg_fold_pairs(Goal, SGoal).

dcg_fold_pairs(Goal, Goal).
```

11.1.1 Extended version for Prolog compilers with encapsulation mechanisms

Assuming that the infix operator `:/2` is used for calling predicates inside an encapsulation unit, the following clause would allow translation of grammar rule bodies that explicitly use non-terminals from another encapsulation unit:

```
dcg_body(Unit:GRBody, S0, S, Unit:Goal) :-
    !,
    dcg_body(GRBody, S0, S, Goal).
```

One possible problem with this clause is that any existence errors when executing the goal `Unit:Goal` will most likely be expressed in terms of the expanded predicates and not in terms of the original grammar rule non-terminals. In order to more easily report errors at the same abstraction level as grammar rules, the following alternative clause may be used:

```
dcg_body(Unit:GRBody, S0, S, Unit:phrase(GRBody, S0, S)) :-
    !,
    dcg_body(GRBody, S0, S, _).  % ensure that GRBody is valid
```

11.2 phrase/3

This section provides a reference implementation in Prolog of the built-in predicate `phrase/3`. It includes the necessary clauses for error handling, as specified in section 8.1.1.3. For the reference implementation of `phrase/2` see section 8.1.1.4.

```
phrase(GRBody, S0, S) :-
    var(GRBody),
    throw(error(instantiation_error, phrase(GRBody, S0, S))).
```

```
phrase(GRBody, S0, S) :-
    \+ callable(GRBody),
    throw(error(type_error(callable, GRBody), phrase(GRBody, S0, S))).

phrase(GRBody, S0, S) :-
    nonvar(S0),
    \+ is_list(S0),
    throw(error(type_error(list, S0), phrase(GRBody, S0, S))).

phrase(GRBody, S0, S) :-
    nonvar(S),
    \+ is_list(S),
    throw(error(type_error(list, S), phrase(GRBody, S0, S))).

phrase(GRBody, S0, S) :-
    dcg_body(GRBody, TS0, TS, Goal),
    TS0 = S0, TS = S,
    call(Goal).
```

The predicate `dcg_body/4` is part of the grammar rule translator reference implementation, defined in section 11.1. An alternative, informal implementation of `phrase/3` using a meta-interpreter is presented in the Annex A.

11.3 Auxiliary predicates used on the reference implementations

The following auxiliary predicates are used on the reference implementations:

```
append([], List, List).
append([Head| Tail], List, [Head| Tail2]) :-
    append(Tail, List, Tail2).

callable(Term) :-
    nonvar(Term),
    functor(Term, Functor, _),
    atom(Functor).

is_list([]) :-
    !.
is_list([_| Tail]) :-
    is_list(Tail).

is_proper_list(List) :-
    List == [], !.
is_proper_list([_| Tail]) :-
    nonvar(Tail),
    is_proper_list(Tail).
```

12 Test-cases for the reference implementations

12.1 Built-in predicates and user-defined hook predicates

```
% built-in predicates:
gr_pred_test(phrase(_, _,_), [built_in, static]).
gr_pred_test(phrase(_, _), [built_in, static]).

% simple test predicate:
test_gr_preds :-
    write('Testing existence of built-in predicates'), nl,
    write('and user-defined hook predicates...'), nl, nl,
    gr_pred_test(Pred, ExpectedProps),
    functor(Pred, Functor, Arity),
    write('Testing predicate '), write(Functor/Arity), nl,
    write('  Expected properties: '), write(ExpectedProps), nl,
    findall(Prop, predicate_property(Pred, Prop), ActualProps),
    write('  Actual properties: '), write(ActualProps), nl,
    fail.
test_gr_preds.
```

12.2 `phrase/2-3` built-in predicate tests

Tests needed!

12.3 Grammar-rule translator tests

Know any hard to translate grammar rules? Contribute them!

When checking compliance of a particular grammar rule translator, results of the tests in this section must be compliant with the logical expansion of grammar rules, as specified in section 10.

```
% terminal tests with list notation:
gr_tr_test(101, (p --> []), success).
gr_tr_test(102, (p --> [b]), success).
gr_tr_test(103, (p --> [abc, xyz]), success).
gr_tr_test(104, (p --> [abc | xyz]), error).
gr_tr_test(105, (p --> [[], {}, 3, 3.2, a(b)]), success).
gr_tr_test(106, (p --> [_]), success).

% terminal tests with string notation:
gr_tr_test(151, (p --> "b"), success).
gr_tr_test(152, (p --> "abc", "q"), success).
gr_tr_test(153, (p --> "abc" ; "q"), success).
```

```
% simple non-terminal tests:
gr_tr_test(201, (p --> b), success).
gr_tr_test(202, (p --> 3), error).
gr_tr_test(203, (p(X) --> b(X)), success).

% conjunction tests:
gr_tr_test(301, (p --> b, c), success).
gr_tr_test(311, (p --> true, c), success).
gr_tr_test(312, (p --> fail, c), success).
gr_tr_test(313, (p(X) --> call(X), c), success).

% disjunction tests:
gr_tr_test(351, (p --> b ; c), success).
gr_tr_test(352, (p --> q ; []), success).
gr_tr_test(353, (p --> [a] ; [b]), success).

% if-then-else tests:
gr_tr_test(401, (p --> b -> c), success).
gr_tr_test(411, (p --> b -> c; d), success).
gr_tr_test(421, (p --> b -> c1, c2 ; d), success).
gr_tr_test(422, (p --> b -> c ; d1, d2), success).
gr_tr_test(431, (p --> b1, b2 -> c ; d), success).
gr_tr_test(441, (p --> [x] -> [] ; q), success).

% negation tests:
gr_tr_test(451, (p --> \+ b, c), success).
gr_tr_test(452, (p --> b, \+ c, d), success).

% cut tests:
gr_tr_test(501, (p --> !, [a]), success).
gr_tr_test(502, (p --> b, !, c, d), success).
gr_tr_test(503, (p --> b, !, c ; d), success).
gr_tr_test(504, (p --> [a], !, {fail}), success).
gr_tr_test(505, (p(a), [X] --> !, [X, a], q), success).
gr_tr_test(506, (p --> a, ! ; b), success).

% {}/1 tests:
gr_tr_test(601, (p --> {b}), success).
gr_tr_test(602, (p --> {3}), error).
gr_tr_test(603, (p --> {c,d}), success).
gr_tr_test(604, (p --> '{}'((c,d))), success).
gr_tr_test(605, (p --> {a}, {b}, {c}), success).
gr_tr_test(606, (p --> {q} -> [a] ; [b]), success).
gr_tr_test(607, (p --> {q} -> [] ; b), success).
gr_tr_test(608, (p --> [foo], {write(x)}, [bar]), success).
gr_tr_test(609, (p --> [foo], {write(hello)},{nl}), success).
```

```
gr_tr_test(610, (p --> [foo], {write(hello), nl}), success).

% "metacall" tests:
gr_tr_test(701, (p --> X), success).
gr_tr_test(702, (p --> _), success).

% non-terminals corresponding to "graphic" characters
% or built-in operators/predicates:
gr_tr_test(801, ('[' --> b, c), success).
gr_tr_test(802, ('=' --> b, c), success).

% pushback tests:
gr_tr_test(901, (p, [t] --> b, c), success).
gr_tr_test(902, (p, [t] --> b, [t]), success).
gr_tr_test(903, (p, [t] --> b, [s, t]), success).
gr_tr_test(904, (p, [t] --> b, [s], [t]), success).
gr_tr_test(905, (p(X), [X] --> [X]), success).
gr_tr_test(906, (p(X, Y), [X, Y] --> [X, Y]), success).
gr_tr_test(907, (p(a), [X] --> !, [X, a], q), success).
gr_tr_test(908, (p, [a,b] --> [foo], {write(hello), nl}), success).
gr_tr_test(909, (p, [t1], [t2] --> b, c), error).
gr_tr_test(910, (p, b --> b), error).
gr_tr_test(911, ([t], p --> b), error).
gr_tr_test(911, ([t1], p, [t2] --> b), error).


% simple expand_term/2 test predicate:

test_gr_tr :-
    write('Testing expand_term/2 predicate...'), nl, nl,
    gr_tr_test(N, GR, Result),
    write(N), write(':  '), writeq(GR), write('  ---  '),
    write(Result), write(' expected'), nl,
    (    catch(
             expand_term(GR, Clause),
             Error,
             (write('  error: '), write(Error), nl, fail)) ->
         write('  '), writeq(Clause)
    ;    write('  expansion failed!')
    ),
    nl, nl,
    fail.

test_gr_tr.
```

```
% simple predicate for dumping test grammar rules into a file:
% (restricted to rules whose expansion is expected to succeed)

create_gr_file :-
    write('Creating grammar rules file "gr.pl" ...'),
    open('gr.pl', write, Stream),
    (    gr_tr_test(N, GR, success),
         write(Stream, '% '), write(Stream, N),
         write(Stream, ':'), nl(Stream),
         write_canonical(Stream, GR), write(Stream, '.'),
         nl(Stream), fail
    ;    close(Stream)
    ),
    write(' created.'), nl.
```

A phrase/3 meta-interpreter

Note that this alternative reference implementation makes it simple to report existence errors at the same abstraction level as grammar rules.

```
phrase(GRBody, S0, S) :-
    phrase(GRBody, Cont, S0, S1),
    (   Cont == {} ->
        S = S1
    ;   Cont = !(SBody),
        !,
        phrase(SBody, S1, S)
    ).

phrase(GRBody, _, S0, S) :-
    var(GRBody),
    throw(error(instantiation_error, phrase(GRBody, S0, S))).

phrase(GRBody, _, S0, S) :-
    \+ callable(GRBody),
    throw(error(type_error(callable, GRBody), phrase(GRBody, S0, S))).

phrase(GRBody, _, S0, S) :-
    nonvar(S0),
    \+ is_list(S0),
    throw(error(type_error(list, S0), phrase(GRBody, S0, S))).

phrase(GRBody, _, S0, S) :-
    nonvar(S),
    \+ is_list(S),
```

```
    throw(error(type_error(list, S), phrase(GRBody, S0, S))).

phrase(!, Cont, S0, S) :-
    !,
    Cont = !({}),
    S = S0.

phrase((GRBody1, GRBody2), Cont, S0, S) :-
    !,
    phrase(GRBody1, ContGRBody1, S0, S1),
    (   ContGRBody1 == {} ->
        phrase(GRBody2, Cont, S1, S)
    ;   ContGRBody1 = !(SGRBody1),
        Cont = !((SGRBody1, GRBody2)),
        S = S1
    ).

phrase(\+ GRBody, Cont, S0, S) :-
    !,
    \+ phrase(GRBody, S0, S),
    Cont = {},
    S0 = S.

phrase((GRBody1; GRBody2), Cont, S0, S) :-
    !,
    (    phrase(GRBody1, Cont, S0, S)
    ;    phrase(GRBody2, Cont, S0, S)
    ).

phrase((GRBody1 -> GRBody2), Cont, S0, S) :-
    !,
    phrase(GRBody1, S0, S1),
    phrase(GRBody2, Cont, S1, S).


phrase({}, Cont, S0, S) :-
    !,
    Cont = {},
    S = S0.

phrase({Goal}, Cont, S0, S) :-
    !,
    call(Goal),
    Cont = {},
    S = S0.
```

```
phrase([], Cont, S0, S) :-
    !,
    Cont = {},
    S = S0.

phrase([Head| Tail], Cont, S0, S) :-
    !,
    append([Head| Tail], S, S0),
    Cont = {}.

phrase(GRHead, _, S0, S) :-
    \+ dcg_clause(GRHead, _),
    current_prolog_flag(unknown, Value),
    (    Value == fail ->
         fail
    ;    Value == warning ->
         % implementation-defined warning
    ;    functor(GRHead, NonTerminal, Arity),
         throw(error(
            existence_error(procedure, NonTerminal//Arity),
            phrase(GRHead, S0, S)))
    ).

phrase(GRHead, {}, S0, S) :-
    dcg_clause(GRHead, GRBody),
    phrase(GRBody, ContY, S0, S1),
    (   ContY == {} ->
        S = S1
    ;   ContY = !(SBody),
        !,
        phrase(SBody, S1, S)
    ).
```