G3 New Work Item Proposal

February 2004

PROPOSAL FOR A NEW WORK ITEM

Date of presentation of proposal: 2006-04-26	Proposer: Jonathan Hodgson WG17
Secretariat: ANSI	**ISO/IEC JTC 1 N** ISO/IEC JTC 1/SC 22 N 4049

A proposal for a new work itemshall be submitted to the secretariat of the ISO/IEC joint technical committee concerned with a copy to the ISO Central Secretariat.

Presentation of the proposal - to be completed by the proposer.

Title Definite Clause Grammar Rules
Scope(and field of application) Programming Language Prolog
Purpose and justification - To codify existing practice.
Programme of work If the proposed new work item is approved, which of the following document(s) is (are) expected to be developed? ____ a single International Standard ___ more than one International Standard (expected number: .....) ____ a multi-part International Standard consisting of ..... parts ____ an amendment or amendments to the following International Standard(s) ..... __X__ a technical report , type ..3..... And which standard development track is recommended for the approved new work item? _X___a. Default Timeframe ____b. Accelerated Timeframe ____c. Extended Timeframe
Relevant documents to be considered A draft report is attached
Co-operation and liaison
Preparatory work offered with target date(s)
Signature: J.P.E. Hodgson
Will the service of a maintenance agency or registration authority be required? ..... - If yes, have you identified a potential candidate? .....Paul Moura has agreed to be the editor.....

- If yes, indicate name ..... Are there any known requirements for coding? ...NO..... -If yes, please specify on a separate page Does the proposed standard concern known patented items? ....NO..... - If yes, please provide full information in an annex

Comments and recommendations of the JTC 1 or SC 22 Secretariat - attach a separate page as an annex, if necessary

Comments with respect to the proposal in general, and recommendations thereon: It is proposed to assign this new item to JTC 1/SC 22

Voting on the proposal - Each P-member of the ISO/IEC joint technical committee has an obligation to vote within the time limits laid down (normally three months after the date of circulation).

Date of circulation: 2006-04-26	Closing date for voting: 2006-07-26	Signature of Secretary: Lisa Rajchel

NEW WORK ITEM PROPOSAL - PROJECT ACCEPTANCE CRITERIA		
Criterion	**Validity**	**Explanation**
A. Business Requirement		
A.1 Market Requirement	Essential ___ Desirable __X_ Supportive ___	Most implementations of Prolog supply some version of this
B. Related Work		
B.1 Completion/Maintenance of current standards	Yes _X__ No___	It is desirable that an accceptable format for DCGs be set forth.
B.2 Commitment to other organisation	Yes ___ No__X_	
B.3 Other Source of standards	Yes ___ No_X__	
C. Technical Status		
C.1 Mature Technology	Yes _X__ No___	Most implementations of Prolog supply some version of th

C.2 Prospective Technology	Yes ___ No___X	
C.3 Models/Tools	Yes _X__ No___	A reference model will be supplied.
D. Conformity Assessment and Interoperability		
D.1 Conformity Assessment	Yes ___ No__X_	
D.2 Interoperability	Yes _ No__X	
E. Adaptability to Culture, Language, Human Functioning and Context of use		
E.1 Cultural and Linguistic Capability	Yes ___ No__X_	
E.2 Adaptability to Human Functioning and Context of Use	Yes _X__ No___	
F. Other Justification		

Notes to Proforma

A. Business Relevance. That which identifies market place relevance in terms of what problem is being solved and or need being addressed.

A.1 Market Requirement. When submitting a NP, the proposer shall identify the nature of the Market Requirement, assessing the extent to which it is essential, desirable or merely supportive of some other project.

A.2 Technical Regulation. If a Regulatory requirement is deemed to exist - e.g. for an area of public concern e.g. Information Security, Data protection, potentially leading to regulatory/public interest action based on the use of this voluntary international standard - the proposer shall identify this here.

B. Related Work. Aspects of the relationship of this NP to other areas of standardisation work shall be identified in this section.

B.1 Competition/Maintenance. If this NP is concerned with completing or maintaining

existing standards, those concerned shall be identified here.

B.2 External Commitment. Groups, bodies, or for a external to JTC 1 to which a commitment has been made by JTC for Co-operation and or collaboration on this NP shall be identified here.

B.3 External Std/Specification. If other activities creating standards or specifications in this topic area are known to exist or be planned, and which might be available to JTC 1 as PAS, they shall be identified here.

C. Technical Status. The proposer shall indicate here an assessment of the extent to which the proposed standard is supported by current technology.

C.1 Mature Technology. Indicate here the extent to which the technology is reasonably stable and ripe for standardisation.

C.2 Prospective Technology. If the NP is anticipatory in nature based on expected or forecasted need, this shall be indicated here.

C.3 Models/Tools. If the NP relates to the creation of supportive reference models or tools, this shall be indicated here.

D. Conformity Assessment and Interoperability Any other aspects of background information justifying this NP shall be indicated here.

D.1 Indicate here if Conformity Assessment is relevant to your project. If so, indicate how it is addressed in your project plan.

D.2 Indicate here if Interoperability is relevant to your project. If so, indicate how it is addressed in your project plan

E. Adaptability to Culture, Language, Human Functioning and Context of Use
NOTE: The following criteria do not mandate any feature for adaptability to culture, language, human functioning or context of use. The following criteria require that if any features are provided for adapting to culture, language, human functioning or context of use by the new Work Item proposal, then the proposer is required to identify these features.

E.1 Cultural and Linguistic Adaptability. Indicate here if cultural and natural language adaptability is applicable to your project. If so, indicate how it is addressed in your project plan.

ISO/IEC TR 19764 (Guidelines, methodology, and reference criteria for cultural and linguistic adaptability in information technology products) now defines it in a simplified way: ability for a product, while keeping its portability and interoperability properties, to:
- be internationalized, that is, be adapted to the special characteristics of natural

languages and the commonly accepted rules for their se, or of cultures in a given geographical region;
- take into account the usual needs of any category of users, with the exception of specific needs related to physical constraints
Examples of characteristics of natural languages are: national characters and associated elements (such as hyphens, dashes, and punctuation marks), writing systems, correct transformation of characters, dates and measures, sorting and searching rules, coding of national entities (such as country and currency codes), presentation of telephone numbers and keyboard layouts. Related terms are localization, jurisdiction and multilingualism.

E.2 Adaptability to Human Functioning and Context of Use. Indicate here whether the proposed standard takes into account diverse human functioning and diverse contexts of use. If so, indicate how it is addressed in your project plan.
NOTE:
1. Human functioning is defined by the World Health Organization at http://www3.who.int/icf/beginners/bg.pdf as: << In ICF (International Classification of Functioning, Disability and Health), the term functioning refers to all body functions, activities and participation. >>
2. Content of use is defined in ISO 9241-11:1998 (Ergonomic requirements for office work with visual display terminals (VDTs) Part 11: Guidance on usability) as: << Users, tasks, equipment (hardware, software and materials), and the physical and societal environments in which a product is used.>>
3. Guidance for Standard Developers to address the needs of older persons and persons with disabilities).

F. Other Justification Any other aspects of background information justifying this NP shall be indicated here.

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

Editor: Paulo Moura
pmoura@di.ubi.pt

March 25, 2006

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.

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

Draft document comments

- Paulo Moura: *Portuguese comments on ISO/IEC DTR 13211–3: 2004: Grammar Rules in Prolog*, IPQ CT 167 WG17, 2005
- Klaus Daessler: *German comments on ISO/IEC DTR 13211–3: 2004: Grammar Rules in Prolog*, DIN NI22 WG17, 2005

Contributors

This list needs to be completed; so far I've only included people present at the ISO meeting collocated with the ICLP'05, 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 Klimpf (UK)
- Jonathan Hodgson (USA)
- Jose Morales (Spain)
- Katsuhiko Nakamura (Japan)
- Klaus Daessler (Germany)
- Manuel Carro (Spain)
- Mats Carlsson (Sweden)
- Paulo Moura (Portugal)
- Richard O'Keefe (NZ)
- Roger Scowen (UK)
- Tony Dodd (UK)

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 on ISO/IEC 13211–1:

3.209 body (of a grammar-rule): The second argument of a grammar-rule. A grammar-body-sequence, or a grammar-body-alternatives, or a grammar-body-choice, or a grammar-body-element.

3.210 clause-term: A read-term `T.` in Prolog text where `T` does not have principal functor `(:-)/1` nor principal functor `(-->)/2`.

3.211 definite clause grammar: A set of grammar rules.

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

3.213 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.

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

3.215 grammar-body-element: A grammar-body-cut, or a grammar-body-goal, or a grammar-body-nonterminal, or a grammar-body-terminals.

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

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

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

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

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

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

3.222 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.

3.223 new variable with respect to a term T: A variable that is not an element of the variable set of `T`.

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

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

3.226 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.

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

3.228 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

The source code in this section uses variables named `S0`, `S1`, ..., `S` to represent the list of terminals used as arguments when parsing grammar rules or when converting grammar rules into clauses. In this notation, the variables , `S1`, ..., `S` can be regarded as a sequence of states, with `S0` represents 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 parsing `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 `c` is `(-->)/2`	
	`grammar rule =`	`grammar rule term`
Abstract:	*g*	*g*

NOTE — Section 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.

6.3 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:

7.13 Grammar rules

7.13.1 Terminals and non-terminals

In the context of a grammar rule, *terminals* represent words of some language and *non-terminals* represent categories of words (see, respectively, sections 3.18 and 3.16). Terminals are represented by Prolog terms enclosed in Prolog lists in order to distinguish them from non-terminals (string notation may be used in alternative to lists when terminals are characters; see section 6.3.7 of ISO/IEC 13211–1). Non-terminals are represented by Prolog callable terms.

7.13.2 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 7.13.3):

```
NonTerminal --> GRBody.

NonTerminal, PushBackList --> GRBody.
```

The control constructs that may be used on a grammar rule body are described later, in section 7.13.6. An empty grammar rule body is represented by an empty list of terminals:

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

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

7.13.3 Push-back lists

A *push-back list* is a proper list of terminals on the left-hand side of a grammar rule (see 3.222). A push-back list contains terminals that would be *asserted* in the input terminal list after the terminals consumed by the successful application of the grammar rule.

7.13.3.1 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].
```

Procedurally, these grammar rules can be interpreted as, respectively, consuming, and then restoring, one or two terminals.

7.13.4 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, in the presence of a mechanism for encapsulating Prolog code, such as a module system or an object-oriented extension, a non-terminal indicator may be used in predicate directives without the need to know the details of the expansion of grammar rules into Prolog clauses.

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`. 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.

7.13.5 Calling Prolog goals from grammar rules

In the body of grammar rules, curly brackets enclose a sequence of Prolog goals that are called when the grammar rule is used during parsing.

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.

7.13.5.1 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.

7.13.6 Control constructs supported by grammar rules

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

The following Prolog control constructs specified on the ISO/IEC 13211–1 Prolog standard must not be recognized as control constructs when used in the body of grammar rules: `true/0`, `fail/0`, `repeat/0`, `call/1`, `once/1`, `catch/3`, and `throw/1`.

A Prolog implementation may support additional control constructs. Examples include *soft-cuts* and control constructs that enable the use of grammar rules stored on a encapsulation unit such as a module or an object. When the Prolog implementation offers a strictly conforming mode (see 5.1e), this mode shall reject these additional control constructs.

7.13.7 Parsing with 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 parse, it is recommended, but not mandatory, that the error term specified on clause 7.7.7b of ISO/IEC 13211–1 when the flag `unknown` is set to `error` would be:

```
existence_error(grammar_rule, GRI)
```

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

NOTES

1 Implementers should consider reporting errors at the same abstraction level as grammar rules whenever practical.

2 Parsing with grammar rules is defined on sections 8.18.1 and 12.2. In brief, grammar rules can be expanded into Prolog clauses, which allows us to map parsing a grammar rule body into proving a goal given a set of predicate clauses. See section 7.7 of ISO/IEC 13211–1 for details.

8 Built-in predicates

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

8.18 Grammar rule built-in predicates

8.18.1 phrase/3, phrase/2

8.18.1.1 Description

`phrase(GRBody, Input, Rest)` is true iff the grammar rule body `GRBody` can successfully parse, accordingly to the currently defined grammar rules, the list of terminals `Input` unifying `Rest` with the list of the remaining terminals.

Procedurally, `phrase(GRBody, Input, Rest)` is executed by calling the Prolog goal corresponding to the expansion of the grammar rule body `GRBody`, given the terminal lists `Input` and `Rest`, accordingly to the logical expansion of grammar rules described in section 11.

8.18.1.2 Template and modes

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

8.18.1.3 Errors

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

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

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

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

8.18.1.4 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, Input) :-
    phrase(GRBody, Input, []).
```

8.18.1.5 Examples

These examples assume that the following grammar rules have been loaded into the Prolog interactive session:

```
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
```

8.18.2 expand_term/2

8.18.2.1 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 section 10.1.1.

8.18.2.2 Template and modes

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

8.18.2.3 Errors

None.

8.18.2.4 Examples

These examples assume that the following clauses for the `term_expansion/2` predicate have been loaded into the Prolog interactive session:

```
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; as such the example below illustrates just a 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 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 recommendation.

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`, which is described below. 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

Example clause for `term_expansion/2`:

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

11 Logical expansion of grammar rules

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

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

11.1 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 represents 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 $\equiv$ is used to link a expansion rule with its resulting Prolog term or with another expansion rule.

11.2 Expanding a grammar rule

Grammar rules with a push-back list:

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

where:

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

Grammar rule with no push-back list:

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

where:

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

11.3 Expanding a grammar rule non-terminal

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

where:

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

(see section 12.4 for the definition of the auxiliary predicate `append/3`)

11.4 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 12.4 for the definition of the auxiliary predicate `append/3`)

11.5 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)`

12 Reference implementations

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

12.1 Grammar-rule translator

This section provides a reference implementation for a translator of grammar rules into plain Prolog clauses. 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 solution, only compliance with the logical expansion, as specified in section 11.

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.

```
% 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).
```

12.1.1 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, So, S)) :-
    !,
    dcg_body(GRBody, S0, S, _).    % check that GRBody is valid
```

12.2 phrase/3

This section provides a reference implementation in plain Prolog of the built-in predicates `phrase/3`. It includes the necessary clauses for error handling, as specified in section 8.18.1.3. For the reference implementation of `phrase/2` see section 8.18.1.4.

```
% error handling:

phrase(GRBody, Input, Rest) :-
    var(GRBody),
    throw(error(instantiation_error, phrase(GRBody, Input, Rest))).

phrase(GRBody, Input, Rest) :-
    \+ callable(GRBody),
    throw(error(type_error(callable, GRBody), phrase(GRBody, Input, Rest))).

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

phrase(GRBody, Input, Rest) :-
    nonvar(Rest),
    \+ is_list(Rest),
    throw(error(type_error(list, Rest), phrase(GRBody, Input, Rest))).
```

```
phrase(GRBody, Input, Rest) :-
    dcg_body(GRBody, S0, S, Goal),
    Input = S0, Rest = S,
    call(Goal).
```

The predicate `dcg_body/4` is part of the grammar rule translator reference implementation, defined in the previous section. An alternative solution is to define clauses implementing a meta-interpreter for grammar rules. Thus, we may replace the last clause above with the following ones:

```
phrase((GRBody1, GRBody2), Input, Rest) :-
    !,
    phrase(GRBody1, Input, Aux),
    phrase(GRBody2, Aux, Rest).

phrase((GRBody1; GRBody2), Input, Rest) :-
    !,
    (    phrase(GRBody1, Input, Rest)
    ;    phrase(GRBody2, Input, Rest)
    ).

phrase((GRBody1 -> GRBody2), Input, Rest) :-
    !,
    phrase(GRBody1, Input, Aux),
    phrase(GRBody2, Aux, Rest).

phrase(\+ GRBody, Input, Rest) :-
    !,
    \+ phrase(GRBody, Input, Rest), Input = Rest.

phrase({}, Input, Rest) :-
    !,
    Input = Rest.

phrase({Goal}, Input, Rest) :-
    !,
    call(Goal), Input = Rest.

phrase([], Input, Rest) :-
    !,
    Input = Rest.

phrase([Head| Tail], Input, Rest) :-
    !,
    append([Head| Tail], Rest, Input).

phrase(GRHead, Input, Rest) :-
```

```
    \+ (GRHead --> _),
    current_prolog_flag(unknown, Value),
    (    Value == fail ->
         fail
    ;    Value == warning ->
         % implementation-defined warning
    ;    functor(GRHead, NonTerminal, Arity),
         throw(error(
            existence_error(grammar_rule, NonTerminal//Arity),
            phrase(GRHead, Input, Rest)))
    ).

phrase(GRHead, Input, Rest) :-
    (GRHead --> GRBody),
    phrase(GRBody, Input, Rest).
```

Note that, although this alternative does not support cuts in grammar rule bodies, it makes it simple to report existence errors at the same abstraction level as grammar rules.

12.3 `expand_term/2`

This section provides a reference implementation in plain Prolog of the built-in predicate `expand_term/2`. For the sole purpose of clarity, is assumed that the conversion of a grammar rule into a Prolog clause is performed by a predicate named `dcg_rule/2`.

```
expand_term(Term, Expansion) :-
    (    var(Term) ->
         Expansion = Term
    ;    current_predicate(term_expansion/2),
         term_expansion(Term, Expand) ->
         Expansion = Expand
    ;    Term = (_ --> _) ->
         dcg_rule(Term, Clause),
         Expansion = Clause
    ;    Expansion = Term
    ).
```

Note that the call to `term_expansion/2` is protected by a call to the built-in predicate `current_predicate/1` in order to prevent an exception being generated if the user abolishes the `term_expansion/2` predicate.

12.4 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).
```

13 Test-cases for the reference implementations

13.1 Built-in predicates and user-defined hook predicates

```
% user-defined hook predicates:

gr_pred_test(term_expansion(_, _), [(dynamic), multifile]).


% built-in predicates:

gr_pred_test(expand_term(_, _), [built_in, static]).
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.
```

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

Tests needed!

13.3 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 11.

```
% 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.
```