INTERNATIONAL STANDARD

ISO/IEC 13211-2

First edition
2000-06-01

Information technology — Programming languages — Prolog —

Part 2: **Modules**

Technologies de l'information — Langages de programmation — Prolog —

Partie 2: Modules

Reference number
ISO/IEC 13211-2:2000(E)

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 734 10 79
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 13211 may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 13211-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

ISO/IEC 13211 consists of the following parts, under the general title *Information technology — Programming languages — Prolog*:

— *Part 1: General core*

— *Part 2: Modules*

Introduction

This is the first International Standard for Prolog, Part 2 (Modules). It was produced on May 1, 2000.

Prolog (Programming in Logic) combines the concepts of logical and algorithmic programming, and is recognized not just as an important tool in AI (Artificial Intelligence) and expert systems, but as a general purpose high-level programming language with some unique properties.

The language originates from work in the early 1970s by Robert A. Kowalski while at Edinburgh University (and ever since at Imperial College, London) and Alain Colmerauer at the University of Aix-Marseilles in France. Their efforts led in 1972 to the use of formal logic as the basis for a programming language. Kowalski's research provided the theoretical framework, while Colmerauer's gave rise to the programming language Prolog. Colmerauer and his team then built the first interpreter, and David Warren at the AI Department, University of Edinburgh, produced the first compiler.

The crucial features of Prolog are unification and backtracking. Unification shows how two arbitrary structures can be made equal, and Prolog processors employ a search strategy which tries to find a solution to a problem by backtracking to other paths if any one particular search comes to a dead end.

Prolog is good for windowing and multimedia because of the ease of building complex data structures dynamically, and also because the concept of backing out of an operation is built into the language. Prolog is also good for interactive web applications because the language lends itself to both the production and analysis of text, allowing for production of HTML 'on the fly'.

This International Standard defines syntax and semantics of modules in ISO Prolog. There is no other International Standard for Prolog modules.

Modules in Prolog serve to partition the name space and support encapsulation for the purposes of constructing large systems out of smaller components. The module system is procedure-based rather than atom-based. This means that each procedure is to be defined in a given name space. The requirements for Prolog modules are rendered more complex by the existence of context sensitive procedures.

Information technology — Programming languages — Prolog — Part 2: Modules

1 Scope

This part of ISO/IEC 13211 is designed to promote the applicability and portability of Prolog modules that contain Prolog text complying with the requirements of the Programming Language Prolog as specified in this part of ISO/IEC 13211.

This part of ISO/IEC 13211 specifies:

a) The representation of Prolog text that constitutes a Prolog module,

b) The constraints that shall be satisfied to prepare Prolog modules for execution, and

c) The requirements, restrictions and limits imposed on a conforming Prolog processor that processes modules.

This part of ISO/IEC 13211 does not specify:

a) The size or number of Prolog modules that will exceed the capacity of any specific data processing system or language processor, or the actions to be taken when the limit is exceeded,

b) The methods of activating the Prolog processor or the set of commands used to control the environment in which Prolog modules are prepared for execution,

c) The mechanisms by which Prolog modules are loaded,

d) The relationship between Prolog modules and the processor-specific file system.

1.1 Notes

Notes in this part of ISO/IEC 13211 have no effect on the language, Prolog text, module text or Prolog processors that are defined as conforming to this part of ISO/IEC 13211. Reasons for including a note include:

a) Cross references to other clauses and subclauses of this part of ISO/IEC 13211 in order to help readers find their way around.

b) Warnings when a built-in predicate as defined in this part of ISO/IEC 13211 has a different meaning in some existing implementations.

2 Normative reference

The following normative document contains provision which, through reference in this text, constitute provisions of this part of ISO/IEC 13211. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 13211 are encouraged to investigate the possibility of applying the most recent edition of the normative document indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 13211-1 : 1995, *Information technology — Programming languages – Prolog Part 1: General core.*

3 Terms and definitions

The terminology for this part of ISO/IEC 13211 has a format modeled on that of ISO 2382.

An entry consists of a phrase (in **bold type**) being defined, followed by its definition. Words and phrases defined in the glossary are printed in *italics* when they are defined in ISO/IEC 13211-1 or other entries of this part of ISO/IEC 13211. When a definition contains two words or phrases defined in separate entries directly following each other (or separated only by a punctuation sign), * (an asterisk) separates them.

Words and phrases not defined in the glossary are assumed to have the meaning given in ISO 2382-15 and ISO/IEC 13211-1; if they do not appear in ISO 2382-15 or ISO/IEC 13211-1, then they are assumed to have their usual meaning.

A double asterisk (**) is used to denote those definitions where there is a change from the meaning given in ISO/IEC 13211-1.

3.1 accessible procedure: See 3.39 – *procedure, accessible.*

3.2 activation, of a procedure: A *procedure* has been *activated* when it is called for execution.

3.3 argument, qualified: A *qualified term* which is an *argument* in a *module name qualified * predication.*

3.4 calling context: The set of *visible procedures*, the *operator* table, the *character conversion mapping* and *Prolog flag* values denoted by a *module name*, and used as a context for *activation* of a *context sensitive procedure.*

3.5 database, visible: The *visible database* of a *module* `M` is the set of *procedures* that can be *activated* without *module name qualification* from within `M`.

3.6 defining module: See 3.23 – *module, defining.*

3.7 export: To make a *procedure* of an *exporting module* available for *import* or *re-export* by other *modules*.

3.8 exported procedure: See 3.41 – *procedure, exported.*

3.9 import: To make *procedures * exported* or *re-exported* by a *module * visible* in an *importing* or *re-exporting module*.

3.10 import, selective: The *importation* into a *module* of only certain explicitly indicated *procedures * exported* or *re-exported* by a *module* (see 6.2.5.2).

3.11 load (a *module*): *Load* the *module interface* of a *module* and correctly prepare all its *bodies*, if any, for *execution*.

NOTE — The interface of a module shall be loaded before any body of the module (see 6.2.3).

3.12 load (a *module interface*): Correctly prepare the *module interface* of the *module* for *execution*.

3.13 lookup module: See 3.29 – *module, lookup*.

3.14 meta-argument: An argument in a *metaprocedure* which is context sensitive.

3.15 metapredicate: A *predicate* denoting a *metaprocedure*.

3.16 metapredicate directive: A *directive* stipulating that a *procedure* is a *metapredicate*.

3.17 metapredicate mode indicator: Either a predicate indicator or a compound term each of whose arguments is `':'`, or `'*'` (see 6.1.1.4).

3.18 metaprocedure: A *procedure* whose actions depend on the *calling context*, and which therefore carries augmented *module* information designating this *calling context*.

3.19 metavariable: A *variable* occurring as an *argument* in a *metaprocedure* which will be subject to *module name qualification* when the *procedure* is activated.

3.20 module: A named collection of *procedures* and *directives* together with provisions to *export* some of the *procedures* and to *import* and *re-export* * *procedures* from other *modules*.

3.21 module body: A *Prolog text* containing the definitions of the *procedures* of a *module* together with *import* and other *directives* local to that *module body*.

3.22 module, calling (of a *procedure*): The *module* in which a corresponding *activator* is *executed*.

3.23 module, defining: The *module* in whose *module body (or bodies)* a *procedure* is defined explicitly and entirely.

3.24 module directive: A *term* `D` which affects the meaning of *module text* (6.2.4), and is denoted in that *module text* by a *directive-term* `:- (D).`.

3.25 module, existing: A *module* whose *interface* has been *prepared for execution* (see 6.2.3).

3.26 module, exporting: A *module* that makes available *procedures* for *import* or *re-export* by other *modules*.

3.27 module interface: A sequence of *read-terms* which specify the *exported* and re-exported procedures and *exported * metapredicates* of a *module*.

3.28 module, importing: A *module* into which *procedures* are *imported*, adding them to the *visible database* of the *module*.

3.29 module, lookup: The *module* where search for *clauses* of a *procedure* takes place.

NOTE — The lookup module defines the visible database of procedures accessible without module name qualification (see 6.1.1.3).

3.30 module name: An *atom* identifying a *module*.

3.31 module name qualification: The *qualification* of a term with a *module name*.

3.32 module, qualifying: See 6.1.1.3 – `Qualifying module, lookup module and defining module`.

3.33 module, re-exporting: A *module* which, by *re-exportation,* imports* certain *procedures* and *exports* these same *procedures*.

3.34 module text: A sequence of *read-terms* denoting *directives*, *module directives* and *clauses*.

3.35 module, user: A *module* with name `user` containing all *user-defined procedures* that are not specified as belonging to a specific *module*.

3.36 predicate **: An *identifier* or *qualified identifier* together with an *arity*.

3.37 predicate name, qualified: The *qualified identifier* of a *predicate*.

3.38 preparation for execution: *Implementation dependent* handling of both *Prolog text* and *module text* by a *processor* which results, if successful, in the processor being ready to execute the prepared *Prolog text* or *module text*.

3.39 procedure, accessible: A *procedure* is *accessible* if it can be *activated* with *module name qualification* from any *module* which is currently *loaded*.

3.40 procedure, context sensitive: A *procedure* is *context sensitive* if the effect of its execution depends on the *calling context* in which it is *activated*.

3.41 procedure, exported: A *procedure* that is made available by a *module* for *import* or *re-export* by other *modules*.

3.42 procedure, visible (in a *module* M): A *procedure* that can be activated from `M` without using *module name qualification*.

3.43 process **: *Execution* activity of a *processor* running *prepared Prolog text* and *module text* to manipulate *conforming Prolog data*, accomplish *side effects* and compute results.

3.44 prototype: A *compound term* where each *argument* is a *variable*.

3.45 prototype, qualified: A *qualified term* whose first *argument* is a *module name* and second *argument* is a *prototype*.

3.46 qualification: The textual replacement (6.4.3) of a *term* `T` by the *term* `M:T` where `M` is a *module name*.

3.47 qualified argument: See 3.3 – *argument, qualified*

3.48 qualified term: See 3.51 – *term, qualified*.

3.49 re-export: To make *procedures * exported* by a *module * visible* in the *re-exporting module*, while at the same time making them available for *import* or *re-export* by other modules from the *re-exporting module*.

3.50 re-export, selective: The *re-exportation* by a *re-exporting * module* of certain indicated *procedures * exported* from another *module* (see 6.2.4.3).

3.51 term, qualified: A *term* whose *principal functor* is `(:)/2`.

3.52 visible procedure (in a *module* `M`): See 3.42 – *procedure, visible*.

3.53 visible database (of a *module* `M`): See 3.5 – *database, visible*.

4 Compliance

4.1 Prolog processor

A conforming processor shall:

a) Correctly prepare for execution Prolog text and module text which conforms to:

1) the requirements of this part of ISO/IEC 13211, including the requirements set out in ISO/IEC 13211-1 General Core, whether or not the text makes explicit use of modules, and

2) the implementation defined and implementation specific features of the Prolog processor,

b) Correctly execute Prolog goals which have been prepared for execution and which conform to:

1) the requirements of this part of ISO/IEC 13211 and ISO/IEC 13211, and

2) the implementation defined and implementation specific features of the Prolog processor,

c) Reject any Prolog text, module text or read-term whose syntax fails to conform to:

1) the requirements of this part of ISO/IEC 13211 and ISO/IEC 13211, and

2) the implementation defined and implementation specific features of the Prolog processor,

d) Specify all permitted variations from this part of ISO/IEC 13211 and ISO/IEC 13211 in the manner prescribed by this part of ISO/IEC 13211 and ISO/IEC 13211, and

e) Offer a strictly conforming mode which shall reject the use of an implementation specific feature in Prolog text, module text or while executing a goal.

4.2 Module text

Conforming module text shall use only the constructs specified in this part of ISO/IEC 13211 and ISO/IEC 13211-1, and the implementation defined and implementation specific features supported by the processor.

Strictly conforming module text shall use only the constructs specified in this part of ISO/IEC 13211 and ISO/IEC 13211-1, and the implementation defined features specified by this part of ISO/IEC 13211.

4.3 Prolog goal

A conforming Prolog goal is one whose execution is defined by the constructs specified in this part of ISO/IEC 13211 and ISO/IEC 13211-1, and the implementation defined and implementation specific features supported by the processor.

A strictly conforming Prolog goal is one whose execution is defined by constructs specified in this part of ISO/IEC 13211 and ISO/IEC 13211-1, and the implementation defined features specified by this part of ISO/IEC 13211.

4.4 Prolog modules

4.4.1 Prolog text without modules

A processor supporting modules shall be able to prepare and execute Prolog text that does not explicitly use modules. Such

text shall be prepared and executed as the body of the required built-in module named **user**.

4.4.2 The module user

A Prolog processor shall support a built-in module `user`. User-defined procedures not defined in any particular module shall belong to the module `user`.

4.5 Documentation

A conforming Prolog processor shall be accompanied by documentation that completes the definition of every implementation defined implementation specific features (if any) specified in this part of ISO/IEC 13211and ISO/IEC 13211-1.

4.5.1 Dynamic Modules

A Prolog processor may support additional implementation specific procedures that support the creation or abolition of modules during execution of a Prolog goal.

4.5.2 Inaccessible Procedures

A Prolog processor may support additional features whose effect is to make certain procedures defined in the body of a module not accessible from outside the module.

5 Syntax

This clause defines the abstract syntax of Prolog text that supports modules. The notation is that of ISO/IEC 13211-1.

Clause 5.1 defines the syntax of module text. Clause 5.2 defines the role of the operator ':'.

5.1 Module text

Module text is a sequence of read-terms which denote (1) module directives, (2) interface directives, (3) directives, and (4) clauses of user-defined procedures.

The syntax of a module directive and of a module interface directive is that of a directive.

```
                module text =  m text ;
Abstract:       mt             mt
```

```
                m text = directive term, m text ;
Abstract:       d · t    d               t
```

```
                m text = clause term, m text ;
Abstract:       c · t    c             t
```

```
                m text = ;
Abstract:       nil
```

Table 1 — The initial operator table

Priority	Specifier	Operator(s)
1200	xfx	:- -->
1200	fx	:- ?-
1100	xfy	;
1050	xfy	->
1000	xfy	,
900	fy	\+
700	xfx	= \=
700	xfx	== \== @< @=< @> @>=
700	xfx	=..
700	xfx	is =:= =\= < =< > >=
600	xfy	:
500	yfx	+ - /\ \/
400	yfx	* / // rem mod << >>
200	xfx	**
200	xfy	^
200	fy	- \

Clause 6.2.4 defines the module directives and the module interface directives. Clause 6.2.5 defines directives in addition to those of ISO/IEC 13211-1 that can appear in a module body and their meanings.

5.2 Terms

5.2.1 Operators

The operator table specific to a module `M` defines which atoms will be regarded as operators in the context of the given module module `M` when (1) a sequence of tokens is parsed as a read-term by the built-in predicate `read_term/3` or (2) Prolog text is prepared for execution or (3) output by the built-in predicates `write_term/3, write_term/2, write/1, write/2, writeq/1, writeq/2`.

The effect of the directives `op/3`, `char_conversion/2` and `set_prolog_flag/2` in modules with multiple bodies is described in 6.2.5.4.

Table 1 defines the predefined operators. The operator `':'` is used for module qualification.

NOTES

1 This table is the same as table 7 of ISO/IEC 13211-1 with the single addition of the operator ':'.

2 When used in a predicate indicator or predicate name ':' is an atom qualifier. This means that a predicate name can be a compound term provided that the functor is ':'.

3 The operator table can be changed both by the use of the module interface directive `op/3` and by the module directive `op/3` in the body of a module.

6 Language concepts and semantics

This clause defines the semantic concepts of Prolog with modules.

a) Subclause 6.1 defines the qualifying module and unqualified term associated with a qualified term,

b) Subclause 6.2 defines the division of module text into Prolog modules,

c) Subclause 6.2.6 defines the relationship between clauses in module text and in the complete database,

d) Subclause 6.3 defines the complete database and its relation to Prolog modules,

e) Subclause 6.4 defines metapredicates and the process of name qualification,

f) Subclause 6.5 defines the process of converting terms to clauses and vice versa in the context of modules,

g) Subclause 6.6 defines the process of executing a goal in the presence of module qualification,

h) Subclause 6.7 defines the process of executing a control construct in the presence of module qualification.

i) Subclause 6.8 defines predicate properties,

j) Subclause 6.9 defines required flags in addition to those required by ISO/IEC 13211-1.

k) Subclause 6.10 defines errors in addition to those required by ISO/IEC 13211-1.

6.1 Related terms

This clause extends the definitions of clause 7.1 of ISO/IEC 13211-1.

6.1.1 Qualified and unqualified terms

6.1.1.1 Qualified terms

A qualified term is a term whose principal functor is `(:)/2`.

6.1.1.2 Unqualified terms

An unqualified term is a term whose principal functor is not `(:)/2`.

6.1.1.3 Qualifying module

Given a module `M` and a term `T`, the associated qualifying module `QM = qm(M:T)` and associated unqualified term `UT = ut(M:T)` of `(M:T)` are defined as follows:

a) If the principal functor of `T` is not (:)/2 then `qm(M:T)` is `M` and `ut(M:T)` is `T`;

b) If the principal functor of `T` is (:)/2 with first argument `MM`, and second argument `TT`, then `qm(M:T)` is the qualifying module of `qm(MM:TT)`, and `ut(M:T)` is the unqualified term `ut(MM:TT)`.

6.1.1.4 Metapredicate mode indicators

A metapredicate mode indicator is either a predicate indicator or a compound term `M_Name(Modes)` each of whose arguments is ':' or '*'.

If the flag `colon_sets_calling_context` 6.9.1 is `true` shall be a compound term each of whose arguments is ':' or '*'. In this case an argument whose position corresponds to a ':' is a meta-argument, and an argument corresponding to '*' shall not be a meta-argument.

6.2 Module text

Module text specifies one or more user-defined modules and the required module `user`. A module consists of a single module interface and zero or more corresponding bodies. The interface shall be prepared for execution before any of the bodies. Bodies may be separated from the interface. If there are multiple bodies, they need not be contiguous.

The heads of clauses in module text shall be implicitly module qualified only by the module body in which they appear, not by explicit qualification of the clause head.

Every procedure that is neither a control construct nor a built-in predicate belongs to some module. Built-in predicates and control constructs are visible everywhere and do not require module qualification, except that if the flag `colon_sets_calling_context` 6.9.1 is `true` the builtin metapredicates (6.4.1) , the context sensitive builtins 6.4.2 and `call/1` and `catch/3` may be module qualified for the purpose of setting the calling context.

6.2.1 Module user

The required module `user` contains all user-defined procedures not defined within a body of a specific module. It has by default an empty module interface. However, module text may contain an explicit interface for module `user`. Any such interface must be loaded before any Prolog text belonging to the module `user`.

NOTE — An explicit interface for module `user` enables procedures to be exported from module `user` to other modules and allows metapredicates to be defined in module `user`.

6.2.2 Procedure Visibility

All procedures defined in a module are accessible from any module by use of explicit module qualification. It shall be an allowable extension to provide a mechanism that hides certain procedures defined in a module `M` so that they cannot be activated, inspected or modified except from within a body of the module `M`.

A module shall not make visible by import or re-export two or more procedures with a given (unqualified) predicate indicator defined in different modules. If a procedure with (unqualified) predicate indicator `PI` from the complete database is visible in `M` no other procedure with the same predicate indicator shall be made visible in `M`.

NOTE — More than one import or re-export directive may make visible a single procedure in a module.

6.2.3 Module interface

A module interface in module text specifies the name of the module, the operators, character conversions and Prolog flag

values that shall be used when the processor begins to prepare for execution the bodies of the module, and the user-defined procedures of a module that are

a) exported from the module,

b) re-exported from the module, and

c) defined to be metapredicates by the module.

A sequence of directives shall form the module interface of the module with name `Name` if :

a) The first directive is a directive `module(Name)`. (6.2.4.1)

b) The last directive is a directive `end_module(Name)`. (6.2.4.9)

c) Each other element of the sequence is a module interface directive. (6.2.4.2 through 6.2.4.8)

The interface for a module `Name` shall be loaded before any body of the module.

6.2.4 Module directives

Module directives are module text which serve to 1) separate module text into the individual modules, and 2) define operators, character conversions and flag values that apply to the preparation for execution of the bodies of the corresponding module.

6.2.4.1 Module directive module/1

The module directive `module(Name)` specifies that the interface text bracketed by the directive and the matching closing interface directive `end_module(Name)` defines the interface to the Prolog module `Name`.

6.2.4.2 Module interface directive export/1

A module interface directive `export(PI)` in the module interface of a module `M`, where `PI` is a predicate indicator, a predicate indicator sequence or a predicate indicator list, specifies that the module `M` makes the procedures designated by `PI` available for import into or re-export by other modules.

A procedure designated by `PI` in a `export(PI)` directive shall be that of a procedure defined in the body (or bodies) of the module `M`.

No procedure designated by `PI` shall be a control construct, a built-in predicate, or an imported procedure.

NOTE — Since control constructs and built-in predicates are visible everywhere they cannot be exported.

6.2.4.3 Module interface directive reexport/2

A directive `reexport(M, PI)` in the interface of a module `MM` where `M` is an atom and `PI` is a predicate indicator, a predicate indicator sequence or a predicate indicator list specifies that the module `MM` imports from the module `M` all the procedures designated by `PI`, and that `MM` makes these procedures available for import or re-export (from `MM`) by other modules.

A procedure designated by `PI` in a `reexport(M,PI)` directive shall be that of a procedure exported or re-exported by the module `M`.

No procedure designated by `PI` shall be a control construct or a built-in predicate.

6.2.4.4 Module interface directive reexport/1

A module interface directive `reexport(PI)` in the module interface of a module `M`, where `PI` is an atom, a sequence of atoms, or a list of atoms specifies that the module `M` imports all the user defined procedures exported or re-exported by the modules designated by `PI` and that `M` makes these procedures available for import into or re-exportation by other modules.

6.2.4.5 Module interface directive metapredicate/1

A module interface directive `metapredicate(MI)` in the module interface of a module `M`, where `MI` is a metapredicate mode indicator, a metapredicate mode indicator sequence, or a metapredicate mode indicator list specifies that the module defines and exports the metaprocedures designated by `MI`.

6.2.4.6 Module interface directive op/3

A module interface directive `op(Priority, Op_specifier, Operator)` in the module interface of a module `M` enables the initial operator table to be altered only for the preparation for execution of all the bodies of the module `M`.

The arguments `Priority`, `Op_specifier`, and `Operator` shall satisfy the same constraints as for the successful execution of the built-in predicate `op/3` (8.14.3 of ISO/IEC 13211-1) and the initial operator table of the module shall be altered in the same way.

Operators defined in a module interface directive `op(Priority, Op_specifier, Operator)` shall not affect the syntax of read terms in Prolog and module texts other than the bodies of the corresponding module.

6.2.4.7 Module interface directive char_conversion/2

A module interface directive `char_conversion(In_char, Out_char)` in the module interface of a module `M` enables the initial character conversion mapping $Conv_C$ (see 3.29 of ISO/IEC 13211-1) to be altered only for the preparation for execution of all the bodies of the module `M`.

The arguments `In_char`, and `Out_char` shall satisfy the same constraints as for the successful execution of the built-in predicate `char_conversion/2` (8.14.5 of ISO/IEC 13211-1) and $Conv_C$ shall be altered in the same way.

Character conversions defined in a module interface directive `char_conversion(In_char, Out_char)` shall not affect the syntax of read terms in Prolog and module texts other than the bodies of the corresponding module.

6.2.4.8 Module interface directive set_prolog_flag/2

A module interface directive `set_prolog_flag(Flag, Value)` in the module interface of a module `M` enables the initial value associated with a Prolog flag to be altered only for the preparation for execution of all the bodies of the module `M`.

The arguments `Flag`, and `Value` shall satisfy the same constraints as for the successful execution of the built-in predicate `set_prolog_flag/2` (8.17.1 of ISO/IEC 13211-1) and the `Value` shall be associated with flag `Flag` in the same way.

Values associated with flags in a module interface directive `set_prolog_flag(Flag, Value)` shall not affect the values associated with flags in Prolog and module texts other than the bodies of the corresponding module.

6.2.4.9 Module directive end_module/1

The module directive `end_module(Name)`where `Name` is an atom that has already appeared as the argument of a module directive `module/1`, specifies the termination of the interface for the module `Name`.

NOTE — Unless otherwise so defined module directives are not Prolog text. Thus `op/3`, `char_conversion/2` and `set_prolog_flag/2` are both module directives and directives (see ISO/IEC 13211-1 7.4.2.4, 7.4.2.5 and 7.4.2.9.)

6.2.5 Module body

A module body belonging to a module is Prolog text which defines user-defined procedures that belong to the module.

A sequence of directives and clauses shall form a body of the module with name `Name` if:

a) The first element of the sequence is a directive `body(Name)` (6.2.5.1).

b) The last element of the sequence is a directive `end_body(Name)`(6.2.5.4).

Directives `import/1` and `import/2` make visible in the importing module procedures defined in an exporting or re-exporting module.

6.2.5.1 Module directive body/1

A module directive `body(Name)` where `Name` is an atom giving the name of a module specifies that the Prolog text bracketed between this directive and the next end module directive `end_body(Name)` belongs to the module `Name`. Such procedures shall be visible in all bodies of `Name` without name qualification.

6.2.5.2 Directive import/2

A directive `import(M, PI)` in a body of a module `MM` where `M` is an atom and `PI` is a predicate indicator, a predicate indicator sequence or a predicate indicator list specifies that the module `MM` imports from the module `M` all the procedures designated by `PI`.

A procedure designated by `PI` in a `import(M,PI)` directive shall be a procedure exported or re-exported by the module `M`.

No procedure designated by `PI` shall be a control construct or a built-in predicate.

6.2.5.3 Directive import/1

A directive `import(MI)` in a body of a module `MM` where `MI` is an atom, a sequence of atoms, or a list of atoms specifies that the module `MM` imports all the procedures exported by the modules designated by `MI`. Such procedures shall be visible in `MM` without name qualification.

6.2.5.4 Module directive end_body/1

The module directive `end_body(Name)` where `Name` is an atom that has already appeared as the argument of a module directive `body/1` specifies the termination of the Prolog text belonging to the particular module body of module `Name`.

The preparation for execution of any module interface shall set the operator table, character conversion mapping $Conv_C$ (see 3.29 of ISO/IEC 13211-1), and Prolog flag values to a new initial state, determined by the module interface directives `op/3`, `char_conversion/2`, and `set_prolog_flag/2` in the interface of `M`. This state shall affect only the preparation for execution of the subsequent bodies of the module `M`. The effect of directives `op/3`, `char_conversion/2`, and `set_prolog_flag/2` in a body of a module `M` shall accumulate during the preparation for execution of the current body and all subsequent bodies of the module `M`.

NOTE — A single module may have more than one body. However module text does not permit the nesting of any module body within the Prolog text of the body of any module other than the `user` module.

6.2.6 Clauses

A clause-term in one of the bodies of a module `M` of module text causes a clause of a user-defined procedure to be added to the module `M`.

A clause `C` of a clause-term (`= C.`) in the body of a module `M` shall be an unqualified term which is a clause term whose head is an unqualified term and shall satisfy the same constraints as those required for a successful execution of the built-in predicate `assertz(C)` (7.4.2) in the context of `M`, except that no error referring to modification of a static procedure shall occur. `C` shall be converted to a clause `h:- t` and added to the module `M`.

The predicate indicator `P/N` of the head of `C` shall not be the predicate indicator of any built-in predicate, or a control construct, and shall not be that of any predicate imported into or reexported by `M`.

NOTE — If the directive `discontiguous/1` is in effect for a predicate defined in the body of a module, then clauses for that predicate may appear in separate bodies of the module. The order in

which the clauses are added to the complete database depends on the order in which the bodies are prepared for execution.

6.2.6.1 Examples

The examples defined in this clause assume the complete database has been created from module text that includes the following:

```
:- module(utilities).
:- export([length/2, reverse/2]).
:- end_module(utilities).
:- body(utilities).
    length(List, Len) :- length1(List, 0, N).
    length1([], N, N).
    length1([H | T], N, L) :-
          N1 is N + 1,length1(T, N1, L).

    reverse(List, Reversed) :-
          reverse1(List, [], Reversed).
    reverse1([], R,R).
    reverse1([H | T], Acc, R) :-
          reverse1(T, [H | Acc], R).
:-end_body(utilities).

:- module(foo).
:- end_module(foo).
:- body(foo).
:-import(utilities).
    p(Y) :- q(X),length(X,Y).

    q([1,2,3,4]).
:- end_body(foo).
```

The examples are executed in the context of the module `user`.

```
foo:p(X).
   succeeds,
   unifying X with 4.
foo:reverse([1,2,3], L).
   succeeds,
   unifying  L with [3,2,1].
utilities:reverse1([1,2,3], [], L).
   succeeds,
   unifying L with [3,2,1].
foo:reverse1([1,2,3], [], L).
   existence_error(procedure, foo:reverse1).
```

6.3 Complete database

The complete database is the database of procedures against which execution of a goal is performed. The procedures in the complete database are:

a) all control constructs,

b) all built-in predicates,

c) all user-defined procedures.

Each user-defined procedure is identified by a unique qualified predicate indicator where the module name qualification of the predicate indicator is the defining module of the procedure.

6.3.1 Visible database

The visible database of a module `M` is the collection of all procedures in the complete database that can be activated from `M` without explicit module qualification and from outside `M` with `M` as calling context.

It includes all built-in predicates and control constructs, all procedures defined in the bodies of `M`, all procedures imported into `M`, and all procedures re-exported by `M`.

NOTE — A procedure visible in a module `M` that is neither a control construct nor a built-in predicate is either (1) completely defined in the bodies of `M` or (2) completely defined in the bodies of some module `MM`, exported from `MM` and imported or reexported into `M`. Furthermore the options (1) and (2) are mutually exclusive.

6.3.2 Examples

The following examples use the complete database defined in 6.2.6.1.

The visible database of `foo` consists of the following procedures:

```
All built-in predicates and control
constructs.

From foo:
    p/1, q/1.

Imported from utilities:
   length/2, reverse/2
```

6.4 Context sensitive predicates

The effect of a context sensitive procedure depends on the calling context (3.40) in which it is activated.

Metapredicates are predicates denoting procedures one or more of whose arguments are meta-arguments. If the flag `colon_sets_calling_context` has the value `true` then activation of the metapredicate will require these arguments to be unified with terms that require module qualification. The effect of certain other built-ins which are not metapredicates is also dependent on the calling context.

When the flag `colon_sets_calling_context` is `true` the calling context can be set explicitly by using the infix operator `':'`. When the flag `colon_sets_calling_context` is `false` some other implementation defined method for explicitly setting the calling context shall be provided.

6.4.1 Metapredicate built-ins

The following built-in predicates are metapredicates listed with their metapredicate mode indicators:

a) The database access and modification built-in predicates `clause(:,*)`, `asserta(:)`, `assertz(:)`, `retract(:)`, `abolish(:)`, and `predicate_property(:,*)`,

b) The logic and control built-in predicates `once(:)`, `\+(:)`, and

c) The all solutions predicates `setof(*,:,*)`, `bagof(*,:,*)`, and `findall(*,:,*)`.

6.4.2 Context sensitive built-ins

The following built-in predicates are context sensitive:

a) Built-ins affecting the operator table, character conversion and Prolog flags: `op/3`, `current_op/3`, `char_conversion/2`, `current_char_conversion/2`, `set_prolog_flag/2`, and `current_prolog_flag/2`;

b) Built-in predicates that read or write terms: `read_term/3`, `write_term/3`, `write_term/2`, `write/1`, `write/2`, `writeq/1`, and `writeq/2`.

6.4.3 Module name expansion

When the Prolog flag `colon_sets_calling_context` is `true` an argument `X` of a metapredicate goal `MP` which occurs at a position corresponding to a ':' in the metapredicate mode indicator of `MP` shall be qualified with the module name of the calling context when `MP` is activated. An unqualified term `X` appearing as a ':' argument in a call of a predicate `MP` in module `M` will be replaced by `(M:X)` in the activation of `MP`.

When the Prolog flag `colon_sets_calling_context` is `true` the meta-arguments in an unqualified term `MP` which represents a metapredicate goal in the calling context of a module `CM` shall be module qualified with `CM`. If the term `MP` is module qualified then the meta-arguments shall be module qualified with the qualifying module of the term.

When the Prolog flag `colon_sets_calling_context` is `false` arguments of a metapredicate goal are not subject to module qualification. An implementation defined method of setting the calling context shall be provided.

6.4.4 Examples: Metapredicates

6.4.4.1 colon_sets_calling_context true

These examples on module qualification assume that the Prolog flag `lon_sets_calling_context` is `true`.

The following example illustrates the use of a metapredicate to obtain context information for debugging purposes.

```
:- module(trace).
    :- exports(#/1).
    :- metapredicate(#(:)).

:- end_module(trace).
:- body(trace).
    :- op(950, fx, #).

    (# Goal) :-
        Goal = Module : G,
        inform_user('CALL', Module, G),
        call(Goal),
        inform_user('EXIT', Module, G).
    (# Goal) :-
        Goal = Module : G,
        inform_user('FAIL', Module, Goal),
        fail.
    inform_user(Port, Module, Goal) :-
       write(Port), write(' '), write(Module),
       write(' calls '), writeq(Goal), nl.
:- end_body(trace).
```

```
:- module(sort_with_errors).
    :- export(sort/2).
:- end_module(sort_with_errors).
:- body(sort_with_errors).
    :- import(trace).
    sort(List, SortedList) :-
       sort(List, SortedList, []).
    sort([], L,L).
    sort([X|L], R0, R) :-
      # split(X,L,L1,L2),
      # sort(L1, R0, R1),
      # sort(L2, [X|R1], R).
    split(_, [], [], []).
    split(X, [Y|L], [Y |L1], L2):-
      Y @< X, !,
      split(X,L, L2, L2).
    split(X, [Y | L], [Y |L1], L2):-
         Y @< X, !,
      split(X,L, L2, L2).
    split(X, [Y | L], [Y |L1], L2):-
      split(X, L, L2, L2).

:- end_body(sort_with_errors).
The goal:
sort([3,2,1], L).
fails, writing
CALL sort_with_errors calls split(3,[2,1],_A,_B)
FAIL sort_with_errors calls split(3,[2,1],_A,_B).
```

6.4.4.2 colon_sets_calling_context false

This example illustrates an alternate mechanism for setting the calling context. Here `@/2` is used to set the calling context. `G @ M` represents a call of the goal `G` in the calling context of the module `M`.

```
:-module tools.
:-meta [interpret/1].
:-end_module tools.

:-begin_module tools.

interpret(Goal) :-
   calling_context(Module),
   inter(Goal, Module, Module, Module).

% inter(
% Goal,
% CallingContextOfCurrentClause,
% LookupContextOfGoal
% CallingContextOfGoal)

inter(true, _, _, _) :- !.
inter((G1,G2), CallingContext, Home, At) :- !,
   inter(G1, CallingContext, Home, At),
   inter(G2, CallingContext, Home, At).
inter((M:G), CallingContext, _, At) :- !,
   inter(G, CallingContext, M, At).
inter((G@M), CallingContext, Home, _) :- !,
   inter(G, CallingContext, Home, M).
inter(calling_context(M), CallingContext, _, _) :-
!,
   M = CallingContext.
inter(G, _, Home, At):-
   functor(G,N,A),
     % next find  defining module
   current_visible(HomeModule, N/A) @ Home,
   current_predicate(N/A) @ HomeModule, !,
     % fails with BIPS
   clause(G, Body) @ HomeModule,
   inter(Body, At, HomeModule, HomeModule).
```

```
inter(G, _, Home, At) :-
   call(Home:G) @ At.

:-end_module tools.

:-module programs.
:-export [mysort/2].
:-end_module programs.

:-begin_module programs.
    % dynamic only for debugging reasons
:-dynamic([app/3,mysort/2,part/4]).

app([],L,L).
app([H|T],L,[H|G]):-
   app(T,L,G).

mysort([],[]).
mysort([G|T],S):-
   part(G,T,L,H),
   mysort(L,LS),
   mysort(H,HS),
   app(LS,[G|HS],S).

part(_,[],[],[]).

part(J,[H|R],[H|L],U):-
   H =< J,
   !,
   part(J,R,L,U).

part(J,[H|R],L,[H|U]):-
   H > J,
   !,
   part(J,R,L,U).
:-end_module programs.

:-begin_module daten.
list([7,2,6,5,1]).
list([9,0,4,8,3]).
:-end_module daten.

/* module "user" */

:-import tools.
:-import programs.
:-import daten.

dosort:- interpret(sort).

sort:-
   list(X),
   write('unsorted: '), write(X), nl,
   mysort(X,Y),
   write('sorted:   '), write(Y), nl,
   fail.
sort.
```

6.5 Converting a term to a clause, and a clause to a term

Prolog provides the ability to convert Prolog data to and from code. However the argument of a goal is a term whereas the complete database contains procedures with the user-defined procedures being formed from clauses. Some procedures convert a term to a clause, while others convert a clause to a corresponding term. This clause defines how the conversion is to be carried out in the presence of modules.

6.5.1 Converting a term to the head of a clause

A term `T` can be converted with `M` as calling context to a predication which is the head `H` of a clause with defining module `MM`:

a) The associated unqualified term (6.1.1.2) `UT` of `(M:T)` is converted to a predication `H` as in 7.6.1 of ISO/IEC 13211-1:

b) The defining module `MM` for the predication is the qualifying module 6.1.1.3 of `(M:T)`.

6.5.2 Converting a module qualified term to a body

In the calling context of a module `M` with given defining module `DM` a term `T` is converted to the body of a clause in a sequence of steps.

a) The term `T` is module qualified with the name of the calling context to give `M:T`;

b) The term `M:T` is simplified (6.5.2.1) to reduce repeated module qualification giving a term `RT`.

c) The simplified term `RT` is converted to a body `BT` in the calling context of `M` with defining module `DM` (6.5.2.2).

d) The body `BT` is further simplified to remove redundant module qualifications (6.5.2.3).

6.5.2.1 Simplifying a module qualified term

A module qualified term `M:T` is simplified to a reduced module qualified term `RT` as follows:

a) If `T` is a variable then `RT` is `M:T`,

b) Else if the principal functor of `T` is `':-/2'` or one of the control constructs `(,)/2`, `(;)/2` or `(->)/2`, with first argument `A` and second argument `B`, the simplified term `RT` is the same functor (respectively, control construct) with arguments `RA` and `RB` obtained by simplifying the qualified terms `M:A` and `M:B` respectively.

c) Else if the principal functor of `T` is `(:)`, first argument `MM`, second argument `TT`, the term `MM:TT` is simplified to give `RT`,

d) Else `RT` is `M:T`.

6.5.2.2 Converting a simplified term to a body

If the Prolog flag `colon_sets_calling_context` has the value `true` then in the calling context of a module `CM` with defining module `DM` a simplified (qualified) term `T` is converted to a goal `G` which is the body of a clause:

a) If `T` is one of the control constructs `(,)/2`, `(;)/2` or `(->)/2`, then each argument of `T` shall be converted to a goal.

b) Else `T` is a term with principal functor `(:)/2` with first argument `M` and second argument `TT`, and `T` shall be converted to a goal `G` as follows:

1) If TT is a variable then G is the control construct call with argument M:TT.

2) Else if TT is a term whose principal functor is one of the control constructs, true, fail, !, or throw/1 then G is the same control construct and the arguments (if any) of G and TT are identical.

3) Else if TT is a term whose principal functor is call/1 or catch/3 then G is M:G1 where G1 is the corresponding control construct and the arguments of G1 and TT are identical.

4) If TT is an atom or compound term whose principal functor FT does not appear in table 9 of ISO/IEC 13211-1 then G is the goal M:G1 where G1 is a predication whose predicate indicator is FT, and the arguments, if any, of G1 and T are identical.

If the Prolog flag colon_sets_calling_context has the value false then in the calling context of a module CM with defining module DM a simplified (qualified) term T is converted to a goal G which is the body of a clause:

a) If T is one of the control constructs (,)/2, (;)/2 or (->)/2, then each argument of T shall be converted to a goal.

b) Else T is a term with principal functor (:)/2 with first argument M and second argument TT, and T shall be converted to a goal G as follows:

1) If TT is a variable then G is the control construct call with argument TT.

2) Else if TT is a term whose principal functor is one of the control constructs, true, fail, !, or throw/1 then G is the same control construct and the arguments (if any) of G and TT are identical.

3) Else if TT is a term whose principal functor is call/1 or catch/3 then G is the same control control construct and the arguments of G and TT are identical.

4) If TT is an atom or compound term whose principal functor FT does not appear in table 9 of ISO/IEC 13211-1 then G is the goal M:G1 where G1 is a predication whose predicate indicator is FT and the arguments, if any, of G1 and T are identical.

NOTE — In this second case additional implementation specific conversions (6.5.2.4c) are required to account for the explicit method of setting the calling context.

6.5.2.3 Removing redundant module qualifications

A body which is a goal G in a defining module DM is reduced to a goal RG without redundant module qualifications as follows:

a) If G is one of the control constructs (,)/2, (;)/2 or (->)/2, then RG is the same control construct and the arguments of RG are obtained from those of G be reducing each argument for redundant module qualifications.

b) If G is a module qualified goal M:G1 and M is the defining module DM then RG is G1,

c) Else RG is identical to RG.

6.5.2.4 Further implementation defined conversions

An implementation may perform additional conversions on a goal, these may include:

a) Removing module qualifications of predications visible in the defining module.

b) If the flag colon_sets_calling_context has the value true performing module qualification of the meta arguments of metapredicates and/or the control constructs call/1 and catch/3.

c) If the flag colon_sets_calling_context has the value false performing conversions required by the implementation specific method of setting the calling context.

6.5.3 Converting the body of a clause to a term

A goal G which is a predication with predicate indicator P/N in the body of a clause of a module M can be converted to a term T:

a) If the principal functor of G is not (:)/2 and if N is zero, then T is the atom P.

b) If G is a control construct which appears in table 9 of ISO/IEC 13211-1, then T is a term with the corresponding principal functor. If the principal functor of T is call/1, catch/3 or throw/1 then the arguments of G and T are identical, else if the principal functor of T is (,)/2 or (;)/2 or (->)/2 then each argument of G shall also be converted to a term.

c) If colon_sets_calling_context is false and G is an instance of the implementation specific construct that sets the calling context then G shall be converted to a term T using to the implementation specific method for conversion.

d) If the principal functor of G is not (:)/2 and N is not zero then T is a renamed copy of TT where TT is the compound term whose principal functor is P/N and the arguments of G and TT are identical.

e) Else if the principal functor of G is (:)/2 with first argument MM and second argument GG then G is converted to the term MM:TT, where TT is obtained by converting GG to a term in the calling context of MM.

The following examples are provided to illustrate the simplification of module qualified terms and the conversion of terms to goals.

```
Defining module = m, context module = foo.
This would arise in a goal such as
foo:asserta(m:bar(X) :- baz(X)).

In the case where the Prolog flag
colon_sets_calling context is true
the corresponding clause asserted into
module m would be
bar(X) :- foo:baz(X).


(i) Case colon_sets_calling_context true.
```

```
Module qualified term --    m:(:-(dm:h, (a,m1:b)))
Simplified term --         :- ( dm:h , (m:a, m1:b))
Clause in dm --           h :- m:a, m1:b.

Module qualified term --    n:('->'(X, throw(B))
Simplified term --         '->'(n:X, n:throw(B))
Body --                  '->'(call(n:X), throw(B)).

Module qualified term --    m:(',(n:a, b))
Simplified term --         ',(n:a, m:b)
Body (defining module m) -- ',(n:a, b).

(ii) Case colon_sets_calling_context false.
@/2 sets the calling context.

Module qualified term --   m:(:- dm:h, (a @ q , m1:b)))
Simplified term --      :- (dm:h, (m:(a @ q), m1:b)))
Clause in dm   --          h :- a @ q, m1:b.

Module qualified term --    n:('->'(X, throw(B))
Simplified term --         '->'( n:X, n:throw(B))
Body --                  '->'( call(X), throw(B)).

Module qualified term --    n:('->'( X @ q, throw(B))
Simplified term --       '->'(n:(X @ q), n:throw(B))
Body --                '->'(call(X) @ q, throw(B)).
```

6.6 Executing a Prolog goal

This clause describes the flow of control through Prolog clauses as a goal is executed in the presence of module qualification. It is based on the stack model in clause 7.7 of ISO/IEC 13211-1.

6.6.1 Data types for the execution model

The execution model of module Prolog is based on an execution stack `S` of execution states `ES`. It is an extension of the model in clause 7.7 of ISO/IEC 13211-1, where the extension adds module information.

`ES` is a structured data type with components:

`S_index` – A value defined by the current number of components of `S`.

`decsglstk` – A stack of decorated subgoals which defines a sequence of activators that might be activated during execution.

`subst` – A substitution which defines the state of the instantiations of the variables.

`BI` – Backtrack information: a value which defines how to re-execute a goal.

The `choicepoint` for the execution state $\texttt{ES}_{i+1}$ is $\texttt{ES}_i$.

A decorated subgoal `DS` is a structured data type with components:

`activator` – A predication `P` prepared for execution which must be executed successfully in order to satisfy the goal.

`contextmodule` – An atom identifying the module in which the `activator` is being called.

`cutparent` – A pointer to a deeper execution state that indicates where control is resumed should a cut be re-executed.

`currstate`, the current execution state is `top(S)`. It contains:

a) An index which identifies its position in `S`, and

Table 2 — The execution stack after initialization with the goal m:goal

S_index	Decorated Subgoal Stack,	Substitution	BI
1	((`m:goal`, user, 0), $\texttt{newstack}_{DS}$) , $\texttt{newstack}_{ES}$	{}	`nil`

b) The current decorated subgoal stack, and

c) The current substitution, and

d) Backtracking information.

`currdecgsgl`, the current decorated subgoal, is `top(decsglstk)` of `currstate`. It contains:

a) The current activator, `curract`, (this may be a qualified term,)

b) The current context module `contextmodule`, which gives the context in which the current decorated subgoal is to be executed, and

c) Its `cutparent`.

`BI` has value:

`nil` – Its initial value, or

`ctrl` – The procedure is a control construct, or

`bip` – The activated procedure is a built-in predicate, or

`(DM, up(CL))` – `CL` is a list of the clauses of a user-defined procedure whose predicate is identical to `curract`, and which are still to be executed, and `DM` is the module in whose body these clauses appear.

6.6.2 Initialization

The method by which a user delivers a goal to the Prolog processor shall be implementation defined.

A goal is prepared for execution by transforming it into an activator. If the flag `colon_sets_calling_context` is true `true` execution of a metapredicate requires that all arguments of type ':' be module qualified (6.4.3) with the module name of the calling context prior to execution (6.6.4f).

The initial value of the calling context is `user`.

Table 2 shows the execution stack after it has been initialized and is ready to execute `m:goal`.

6.6.2.1 A goal succeeds

A goal is satisfied when the decorated subgoal stack of `currstate` is empty. A solution for the goal `m:goal` is represented by the corresponding substitution Σ.

6.6.2.2 A goal fails

Execution fails when the execution stack `S` is empty.

6.6.2.3 Re-executing a goal

After satisfying an initial goal, execution may continue by trying to satisfy it again.

Procedurally,

a) Pop `currstate` from `S`,

b) Continue execution at 6.6.5.

6.6.3 Searching the complete database

This clause describes how, with lookup module `m`, the processor locates a procedure `p` in the complete database whose predicate indicator corresponds to a given (possibly module qualified) activator.

6.6.3.1 Searching the visible database

The procedure in the complete database corresponding to a procedure `p` (whose principal functor is necessarily not `(:)/2`) in the visible database deetermined by a module `m` is located as follows:

a) If the principal functor of `p` is a control construct or built-in predicate then `p` is the required procedure.

b) If there is a user-defined procedure `p` with the same principal functor and arity as `p` defined in `m` then `p` is the required procedure.

c) The selective import, reexport and selective reexport directives of `m` are examined; (1) if there is a directive naming `p` as imported or re-exported from a module `n` then search is carried out in the visible database of `n` for a procedure `p` which is exported by `n`; (2) else if there is a directive naming a module `n` as imported or re-exported then search is carried out in the visible database of `n` for a procedure `p` which is exported by `n`.

d) Else the search fails.

Procedurally the search in the visible database of a module `m` for a user defined procedure `p` is carried out as follows:

a) If there is a user-defined procedure `p` with the same principal functor and arity as `p` defined in `m` then `p` is the required procedure,

b) Else form two sets `Open` and `Closed` each initially empty.

c) Add `m` to the set `Closed`.

d) If there is a selective import directive `import(n,PI)` or a selective reexport directive `reexport(n,PI)` where `PI` includes `p` replace `Open` by the set whose sole member is `n`,

e) Else create a list `S` of all the modules that are the subject of `import/1` or `reexport/1` directives in `m` and replace `Open` by the set `S`.

f) If `Open` is empty the search fails,

g) Else remove a module `n` from `Open` and add it to `Closed`.

h) If there is a user defined procedure `q` with the same principal functor and arity as `q` defined in `n` and exported by `n` then `q` is the required procedure, and the search terminates,

i) Else if there is a `import/2` directive or a `reexport/2` directive in `n` naming `p` as imported from a module `nn` and `nn` is not on `Closed` replace `Open` by the set whose sole element is `nn`,

j) Else create the set `S` of all modules that are the subject of `import/1` or `reexport/1` directives in `n` and add to `Open` the elements of `S` that are on neither `Open` nor `Closed`.

k) Continue at 6.6.3.1f.

NOTES

1 Because a module `m` may not make visible two different procedures from the same database that would have the same unqualified predicate indicator (6.2.2) in `m` no more than one such procedure can be found.

2 Because no more than one procedure can be found the choice of module from the set `Open` does not need to be specified.

3 Since importation is idempotent no module needs to be searched more than once.

4 The provision of an explicit search algorithm in this subclause does not prescribe that this algorithm shall be implemented by a conforming processor rather it specifies only the effect of the algorithm.

6.6.3.2 Searching for a given procedure

The processor locates in the complete database with lookup module `m` a procedure `p` corresponding to a given term `T`.

a) Determine the unqualified term `UT` and qualifying module `LT` associated to `(m:T)`.

b) If the principal functor of `UT` is a control construct or built-in procedure `p` then `p` is the required procedure.

c) If the principal functor of `UT` is a user-defined procedure `p` (not a control construct or built-in predicate) then the visible database (6.3.1) of `LT` is searched for a procedure `p`. If no such procedure exists the search fails.

6.6.4 Selecting a clause for execution

Execution proceeds in a succession of steps.

a) Using the visible database given by the module `contextmodule` of the current decorated subgoal `currdecsgl`, the processor searches the complete database (6.6.3.2) for a procedure `p` whose (possibly module qualified) predicate indicator corresponds with the (possibly qualified) identifier and arity of `curract`.

b) If no procedure is found in step 6.6.4a, then action depends on the value of the flag `unknown`:

`error` – There shall be an error

```
existence_error(procedure, M:PF)
```

where `M` is the lookup module `contextmodule` and `PF` is the predicate indicator of the (possibly qualified) `curract`, or

`warning` – An implementation dependent warning shall be generated and curract replaced by the control construct `fail`, or

`fail` – `curract` shall be replaced by the control construct `fail`.

c) If `curract` identifies a user-defined predicate set `DM` to the module name of the module in whose body the predicate is defined.

d) If the flag `colon_sets_calling_context` is `true` set `contextmodule` in the current decorated subgoal to the qualifying module associated to `(contextmodule:curract)` and set `curract` to the associated unqualified term.

e) If the flag `colon_sets_calling_context` is `false` perform any implementation actions required to set the value of `contextmodule`.

f) If the flag `colon_sets_calling_context` is `true` ensure that any meta-arguments of `curract` have been module qualified (6.4.3).

g) If `p` is a control construct (true, fail, call, cut, conjunction, disjunction, if-then, if-then-else, catch, throw) then `BI` is set to `ctrl` and execution continues according to the rules defined in (6.7).

h) If `p` is a built-in predicate `BP` then `BI` is set to `bip` and continue execution at 6.6.7.

i) If `p` is a user-defined procedure then `DM` is set to the module in which the procedure is defined and `BI` is set to `(DM,up(CL))`, where `CL` is a list of the current clauses of `p` of the procedure; Continue execution at 6.6.6

NOTE — After the execution of these steps `curract` is not module qualified.

6.6.5 Backtracking

A procedure backtracks (1) if a goal has failed, or (2) if the initial goal has been satisfied, and the processor is asked to re-execute it.

Procedurally, backtracking shall be executed as follows:

a) Examine the value of `BI` for the new `currstate`.

b) If `BI` is `(DM, up(CL))` then `p` is a user defined procedure remove the head of `CL` and continue at 6.6.6.

c) If `BI` is `bip` then `p` is a built-in predicate, continue execution at 6.6.7.

d) If `BI` is `ctrl` the effect of re-executing it is defined in 6.7.

e) If `BI` is `nil` then the new `curract` has not been executed, continue execution at 6.6.4.

6.6.6 Executing a user-defined procedure:

Procedurally a user-defined procedure shall be executed as follows:

a) If there are no (more) clauses for `p` then `BI` has the value `(DM, up([]))` and continue execution at 6.6.6.1,

b) Else consider clause `c` where `BI` has the value `(DM, up([c | CT]))` with the calling context `DM`.

c) If the head of `c` and `curract` are unifiable then it is selected for execution, and continue execution at 6.6.6 e,

d) Else `BI` is replaced by a value `(DM, up(CT))` and continue execution at 6.6.6 a.

e) Let `c'` be a renamed copy of the clause `c` of `up([c | _])`.

f) Unify the head of `c'` and `curract` producing a most general unifier `MGU`.

g) Apply the substitution `MGU` to the body of `c'`.

h) Make a copy `CCS` of `currstate`. It contains a copy of the current goal which is called `CCG`.

i) Apply the substitution `MGU` to `CCG`.

j) Replace the current activator of `CCG` by the `MGU` modified body of `c'`.

k) Set `BI` of `CCS` to `nil`.

l) Set the substitution on `CCS` to a composition of the substitution of `currstate` and `MGU`.

m) Set `cutparent` of the new first subgoal of the decorated subgoal stack of `CCS` to the current choice point.

n) Set the `contextmodule` of the new first subgoal of the decorated subgoal stack to `DM`.

o) Push `CCS` on to `S`. It becomes the new `currstate` and the previous `currstate` becomes its `choicepoint`.

p) Continue execution at 6.6.4.

6.6.6.1 Executing a user-defined procedure with no more clauses

When a user-defined procedure has been selected for execution 6.6.4 but has no more clauses, i.e. `BI` has a value `(DM, up([]))`, it shall be executed as follows:

a) Pop `currstate` from `S`.

b) Continue execution at 6.6.5.

6.6.7 Executing a built-in predicate

Procedurally a built-in predicate shall be executed as in section 7.7.12 of ISO/IEC 13211-1.

For the built-in predicates that have meta-arguments, the database access and modification built-in predicates – `clause(:,*)`, `asserta(:)`, `assertz(:)`, `retract(:)`, `abolish(:)`, and `predicate_property(:,*)`, the logic and control built-in predicates `once(:)`, `\+(:)`, and the all solutions predicates `setof(*,:,*)`, `bagof(*,:,*)`, and `findall(*,:,*)`, the current decorated subgoal gives access to the calling context.

For the builtin predicates which are context sensitive (6.4.2) – `op/3`, `current_op/3`, `char_conversion/2`, `current_char_conversion/2`, `set_prolog_flag/2`, `current_prolog_flag/2`, `read_term/3`, `write_term/3`, `write_term/2`, `write/1`, `write/2`, `writeq/1`, and `writeq/2`, the current decorated subgoal gives access to the calling context.

6.7 Executing a control construct

This clause describes the modifications required to the descriptions of the execution model of ISO/IEC 13211-1. For all control constructs not specifically described, the model is unchanged.

6.7.1 call/1

6.7.1.1 Description

`call(G)` is true in the calling context of module `CM` iff `G` represents a goal which is true in the context of `CM`. Procedurally, a control construct call, denoted by `call(G)`, shall be executed as follows:

a) Make a copy `CCS` of `currstate`.

b) Set BI of CCS to nil.

c) Pop currdecsgl (= (call(G), CM, CP)) from currentgoal of CCS.

d) If the term G has as associated unqualified term a variable, there shall be an instantiation error,

e) Else if the term G has as associated unqualified term a number, there shall be a type error,

f) Else in the calling context of the module CM and defining module CM convert the term G to a goal Goal with calling context M, the qualifying module of (CM:G) (6.5.2).

g) Let NN be the choice point of currstate.

h) Push (Goal, M, NN) on to currentgoal of CCS.

i) Push CCS onto S.

j) Continue execution at 6.6.4.

k) Pop currstate from S.

l) Continue execution at 6.6.5.

call(G) is re-executable. On backtracking, continue at 6.7.1.1k.

6.7.1.2 Template and modes

```
call(+callable_term).
```

6.7.1.3 Errors

a) G is a variable
– instantiation_error.

b) The qualifying module of (CM:G) cannot be determined (6.1.1).
– instantiation_error.

c) G is neither a variable nor a callable term
– type_error(callable, G).

d) G cannot be converted to a goal
– type_error(callable, G).

6.7.1.4 Examples

```
call(m:X:foo).

type_error(callable, m:X:foo).
```

6.7.2 catch/3

The catch and throw control constructs enable execution to continue after an error without intervention from the user.

6.7.2.1 Description

catch(G,C,R) is true in the calling context of module CM iff (1) call(G) is true in the context of CM, or (2) the call of G is interrupted by a call of throw/1 whose argument unifies with C, and call(R) is true in the context of CM. Procedurally, a control construct catch, denoted by catch(G,C,R) is executed as follows:

a) Make a copy CCS of currstate.

b) Replace curract of CCS by call(G).

c) Set BI to nil.

d) Push CCS onto S.

e) Continue execution at 6.6.4.

f) Pop currstate from S.

g) Continue execution at 6.6.5.

catch(G,C,R) is re-executable. On backtracking, continue at 6.7.2.1f.

6.7.2.2 Template and modes

```
catch(?callable_term, ?term, ?term)
```

6.7.2.3 Errors

a) G is a variable
– instantiation_error.

b) The qualifying module of (CM:G) cannot be determined (6.1.1).
– instantiation_error.

c) G is neither a variable nor a callable term
– type_error(callable, G).

6.7.3 throw/1

6.7.3.1 Description

throw(B) is a control construct that is neither true nor false. It exists only for its procedural effect of causing the normal flow of control to be transferred back to an existing call of catch/3 (see 6.7.2).

Procedurally, a control construct throw, denoted by throw(B), shall be executed as follows:

a) Make a renamed copy CA of curract, and a copy CP of cutparent.

b) Pop currstate from S.

c) It shall be a system error (7.12.2j of ISO/IEC 13211-1) if S is now empty,

d) Else if (1) the new curract is a call of the control construct catch/3, and (2) the argument of CA unifies with the second argument C of the catch with most general unifier MGU, and (3) the cutparent is less than CP, then continue at 6.7.3.1b.

e) Apply MGU to currentgoal.

f) Replace curract by call(R), where R is the third argument of the control construct catch/3 from 6.7.3.1d.

g) Set BI to nil.

h) Continue execution at 6.6.4.

6.7.3.2 Template and modes

```
throw(+nonvar)
```

6.7.3.3 Errors

a) `B` is a variable
– `instantiation_error`.

b) `B` does not unify with the `C` argument of any call of `catch/3`
– `system_error`.

6.8 Predicate properties

The properties of procedures can be found using the built-in predicate `predicate_property(Callable, Property)`, where `Callable` is the meta-argument term `Module:Goal` (7.2.2). The predicate properties supported shall include:

`static` – The procedure is static.

`dynamic` – The procedure is dynamic.

`public` – The procedure is a public procedure.

`private` – The procedure is a private procedure.

`built_in` – The procedure is a built-in predicate.

`multifile` – The procedure is the subject of a multifile directive.

`exported` - The module `Module` exports the procedure.

`metapredicate(MPMI)` – The procedure is a metapredicate, and `MPMI` is its metapredicate mode indicator.

`imported_from(From)` — The predicate is imported into module `Module` from the module `From`.

`defined_in(DefiningModule)` – The module with the name `DefiningModule` is the defining module of the procedure.

A processor may support one of more additional predicate properties as an implementation specific feature.

6.9 Flags

The following flag is added to those of 7.11 of ISO/IEC 13211-1.

6.9.1 Flag: colon_sets_calling_context

Possible value: `true`, `false`

Default value: Implementation defined

Changeable: No

Description: If the value of this flag is `true` the operator (:) is used to set the calling context of a metapredicate goal. Meta-arguments in a metapredicate goal must be module qualified when the predicate is activated, with the defining module of the procedure in whose body they are found. If the value is `false` some other implementation defined mechanism by which context sensitive predicates can access their calling context must be provided.

6.10 Errors

The following errors are defined in addition to those defined in section 7.12 of ISO/IEC 13211-1.

6.10.1 Error classification

The following types are added to the classification of 7.12.2 of ISO/IEC 13211-1.

a) The list of valid types is extended by the addition of `metapredicate_mode_indicator`. (See 7.12.2 b of ISO/IEC 13211-1.)

b) The list of valid domains is extended by the addition of `predicate_property`. (See 7.12.2 c of ISO/IEC 13211-1.)

c) The list of object types is extended by the addition of `module`. (See 7.12.2 d of ISO/IEC 13211-1.)

d) The list of permission types is extended by the addition of `implicit`. (See 7.12.2 e of ISO/IEC 13211-1.)

7 Built-in predicates

7.1 The format of built-in predicate definitions

The format of the built-in predicate definitions follows that of ISO/IEC 13211-1.

7.1.1 Type of an argument

The following additional argument types are required:

`metapredicate_mode_indicator` – as terminology.

`predicate_property` – a procedure property (6.8).

`prototype` – as terminology.

`qualified_or_unqualified_clause` – a clause or term whose associated unqualified term is a clause.

7.2 Module predicates

The examples provided for these built-in predicates assume the complete database has been created from the following module text. The flag `colon_sets_calling_context` is assumed to have the value `true`.

```
:- module(foo).
    :- export(p/1).
    :- metapredicate(p(:)).
:- end_module(foo).

:- module(bar).
    :- export(q/1).
:- end_module(bar).

:- module(baz).
    :- export(q/1).
:- end_module(baz).

:- body(foo).
    p(X) :- write(X).
:- end_body(foo).

:- body(bar).
    :- import(foo, p/1).
    q(X) :- a(X), p(X)
    q(X) :- a(X), foo:p(2).
    a(1).
:- end_body(bar).
```

```
:- body(baz).
    :- import(bar, q/1).
:- end_body(baz).
```

7.2.1 current_module/1

7.2.1.1 Description

`current_module(Module)` is true iff `Module` unifies with the name of an existing module.

Procedurally `current_module(Module)` is executed as follows:

a) Searches the complete database for all active modules and creates a set `S` of all terms `M` such that there is a module whose identifier unifies with `Module`.

b) If a non-empty set is found, then proceeds to 7.2.1.1d,

c) Else the goal fails.

d) Chooses an element of `S` and the goal succeeds.

e) If all the elements of `S` have been chosen then the goal fails,

f) Else chooses an element of the set `S` which has not already been chosen and the goal succeeds.

`current_module(Module)` is re-executable. On backtracking, continue at 7.2.1.1e.

NOTE — `current_module(M)` succeeds if the interface to `M` has been loaded, whether or not any bodies of `M` may have been prepared for execution.

7.2.1.2 Template and Modes

```
current_module(?atom)
```

7.2.1.3 Errors

a) Module is neither a variable nor an atom

— `type_error(atom, Module)`.

7.2.1.4 Examples

```
current_module(foo).
   succeeds.

current_module(fred:sid).
   type_error(atom, fred:sid).
```

7.2.2 predicate_property/2

7.2.2.1 Description

`predicate_property(Prototype, Property)` is true in the calling context of a module `M` iff the procedure associated with the argument `Prototype` has predicate property `Property`.

Procedurally `predicate_property(Prototype, Property)` is executed as follows:

a) Determines the qualifying module of `MM` of `(M:Prototype)`.

b) Determines the unqualified term `T` with principal functor `P` of arity `N` associated with `(M:Prototype)`. `P/N` is the associated predicate indicator.

c) Searches the complete database and creates a set Set_{PP} of all terms `PP` such that `P/N` identifies a procedure in the visible database of `MM` which has predicate property `PP` and `PP` is unifiable with `Property`.

d) If Set_{PP} is non empty set is proceeds to 7.2.2.1f,

e) Else the predicate fails.

f) Chooses the first element `PPP` of Set_{PP}, unifies `PPP` with `Property` and the predicate succeeds.

g) If all the elements of Set_{PP} have been chosen the predicate fails,

h) Else chooses the first element `PPP` of Set_{PP} that has not already been chosen, unifies `PPP` with `Property` and the predicate succeeds.

`predicate_property(Prototype, Property)` is re-executable. On backtracking, continue at 7.2.2.1g.

The order in which properties are found by `predicate_property/2` is implementation dependent.

7.2.2.2 Template and modes

```
predicate_property(+prototype, ?predicate_property)
```

7.2.2.3 Errors

a) `Prototype` is a variable
– `instantiation_error`.

b) The qualifying module of `(M:Prototype)` cannot be determined (6.1.1)
– `instantiation_error`.

c) `Prototype` is neither a variable nor a callable term
– `type_error(callable, Prototype)`.

d) `Property` is neither a variable nor a predicate property
– `domain_error(predicate_property, Property)`.

e) The module identified by `MM` does not exist
– `existence_error(module, MM)`.

7.2.2.4 Examples

```
Goals attempted in the context of the module
bar.

predicate_property(q(X), exported).
    succeeds, X is not instantiated.

predicate_property(p(X), defined_in(S)).
    succeeds, S is unified with foo,
    X is not instantiated.

predicate_property(foo:p(X), metapredicate(Y)).
    succeeds, Y is unified with p(:),
    X is not instantiated.

predicate_property(X:p(Y), exported).
```

```
  instantiation_error.

Goal attempted in the context of the module
baz.

predicate_property(foo:p(X), metapredicate(Y)).
    succeeds, Y is unified with p(:),
    X is not instantiated.

The following example assumes that the Prolog
flag colon_sets_calling_context has the value true.

bar:predicate_property(p(X), imported_from(Y)).
   succeeds, Y is unified with foo,
   X is not instantiated.
```

7.3 Clause retrieval and information

This clause describes the interaction of the built-in predicate `clause/2` with the module system.

The examples provided for these built-in predicates assume that the complete database has been created from the following module text.

```
:- module(mammals).
  :- export( dog/0, cat/0, elk/1).
:- end_module(mammals).

:- body(mammals).

  :- dynamic(cat/0).
  cat.

  :- dynamic(dog/0).

  dog :- true.

  :- dynamic(elk/1).
  elk(X) :- moose(X).

  :- dynamic(moose/1).

  legs(4).

  :- end_body(mammals).


:- module(insects)
  :- export(ant/0, bee/0).
:- end_module(insects).

:- body(insects).
  :- dynamic(ant/0).
  ant.

  :- dynamic(bee/0).
  bee.

  :- dynamic(legs/1).
  legs(6).

  body_type(segmented).

:- end_body(insects).

:- module(animals).
  :- exports(limbs/1).
:- end_module(animals).

:- body(animals).
  :- import(insects, [ant/0, bee/0]).
  :- import(mammals, [dog/0, cat/0, elk/1]).

  :- dynamic(horns/1).


  limbs(X) :- insects:legs(X).
  limbs(X) :- mammals:legs(X).

:- end_body(animals).
```

7.3.1 clause/2

7.3.1.1 Description

`clause(Head, Body)` is true in the calling context of a module `M` iff:

– The associated unqualified term of `(M:Head)` is `HH`, (6.1.1.3),

– The procedure of `HH` is public, and

– There is a clause in the qualifying module `DM` of `(M:Head)` which corresponds to a term `H:- B` which unifies with `HH :- Body`.

Procedurally, `clause(Head, Body)` is executed in the calling context of a module `M` as follows:

a) Determines the qualifying module `DM` of `(M:Head)` (6.1.1.3) to be searched for the clauses.

b) Determines the unqualified term `HH` associated with `(M:Head)`.

c) Searches sequentially through each public user-defined procedure defined in the chosen module and creates a list `L` of all the terms `clause(H,B)` such that:

1) `DM` contains a clause whose head can be converted with calling context and defining module `DM` to a term `H` and whose body can be converted with calling context and defining module `DM` to a term `B`,

2) `H` unifies with `HH`, and

3) `B` unifies with `Body`.

d) If a non-empty list is found, then proceeds to 7.3.1.1f,

e) Else the goal fails.

f) Chooses the first element of the list `L`, and the goal succeeds.

g) If all the elements of the list `L` have been chosen then the goal fails,

h) Else chooses the first element of `L` that has not already been chosen, and the goal succeeds.

`clause/2` is re-executable. On backtracking, continue at 7.3.1.1g.

7.3.1.2 Template and modes

```
clause(+term, ?callable_term)
```

7.3.1.3 Errors

a) `Head` is a variable
– `instantiation_error`.