Contents

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

Introduction

This Final Committee Draft defines syntax and semantics of modules in ISO Prolog.

Modules in Prolog serve to partition the name space and support encapsulation for the purposes of constructing large systems out of smaller components. The proposed 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 meta-arguments, that is arguments that are subject to module qualification.

1 Scope

This Final Committee Draft 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 Final Committee Draft.

This Final Committee Draft 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 Final Committee Draft 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 references

The following standards contain provisions which, through the reference of this text, constitute provisions of ISO/IEC 13211. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 13211 are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

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

3 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: A *procedure* is *accessible* if it can be *activated*; in the case of a *dynamic * procedure* modified; or in the case of a non *private * procedure* inspected with *module name qualification* from any *module* which is currently *loaded*.

3.2 activated goal **: A *goal* has been *activated* when it is called for execution.

3.3 calling context: The set of *visible procedures*, denoted by a *module name*, and used as a context for *activation* of a *metapredicate*.

3.4 database, visible: See 3.50 – *visible database*.

3.5 defining module: The *module* in whose *module body* a *procedure* is defined explicitly and entirely.

3.6 export: To make a *procedure* of an *exporting module* available for *import* by other *modules*.

3.7 exported procedure: A *procedure* that is made available by a *module* for *import* by other *modules*.

3.8 import: To make *procedures * exported* by a *module * visible* in an *importing module*.

3.9 import, selective: The *importation* into a *module* of only certain explicitly named *procedures * exported* by a *module* (see 7.2.3.2).

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

NOTE — The interface of a module shall be loaded before any body of the module (see 7.2.1).

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

3.12 lookup module: 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 7.1.1.3).

3.13 meta-argument: An argument in a *metaprocedure* which is context sensitive, and therefore will be subject to *module name qualification* when the *procedure* is activated.

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

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

3.16 metapredicate mode indicator: A compound term each of whose arguments is `':'`, or `'*'` (see 7.1.1.4).

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

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

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

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

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

3.22 module directive: A *term* `D` which affects the meaning of *module text* (7.2.2), and is denoted in that *module text* by a *directive-term* `:- (D).`.

3.23 module, existing: A *module* whose *interface* has been *prepared for execution* (see 7.2.1).

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

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

3.26 module, importing: A *module* into which *procedures* are *imported*, used as a context to search for a *procedure*.

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

3.28 module name qualification: The *qualification* of a *meta-argument* with the *module name* of the *calling module*.

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

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

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

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

3.33 preparation for execution: *Implementation dependent* handling of a *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*.

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

3.35 procedure, visible: See 3.49 - *visible procedure*.

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

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

3.38 qualification: The textual replacement (7.4.2) of a *term* `T` by the *term* `M:T` where `M` is a *module name*.

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

3.40 qualified clause: A *term* whose associated unqualified *term* (7.1.1.3) is a *clause*.

3.41 qualified identifier: A *compound term*, used to denote a *module qualified* object, with *principal functor* `(:)/2` where the first *argument* is a *module name* and the second *argument* is an *identifier*.

3.42 qualified metapredicate mode indicator: A *qualified term* whose first *argument* is an *atom* denoting a *module name* and whose second *argument* is a *metapredicate mode indicator*.

3.43 qualified predicate indicator: A *compound term*, used to denote a *module qualified* object, with *principal functor* `(:)/2` where the first *argument* is a *module name* and the second *argument* is a *predicate indicator*.

NOTE — If `P` is an operator the qualified indicator shall be denoted by `M:(P)/N`. If `M` is an operator, the qualified predicate indicator shall be denoted by `(M):P/N`.

3.44 qualified predicate name: The *qualified identifier* of a *predicate*.

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

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

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

3.48 re-export, selective: The *re-exportation* by an *importing module* of certain named *procedures * exported* from a *module* (see 7.2.2.3).

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

3.50 visible database (of a *module* `M`): The set of *procedures * accessible* without *module name qualification* from within `M`.

4 Symbols and abbreviations

5 Compliance

5.1 Prolog processor

A conforming processor shall:

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

1) the requirements of this Final Committee Draft, 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 Final Committee Draft 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 Final Committee Draft and ISO/IEC 13211, and

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

d) Specify all permitted variations from this Final Committee Draft and ISO/IEC 13211 in the manner prescribed by this Final Committee Draft 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.

5.2 Module text

Conforming module text shall use only the constructs specified in this Final Committee Draft 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 Final Committee Draft and ISO/IEC 13211-1, and the implementation defined features specified by this standard.

5.3 Prolog goal

A conforming Prolog goal is one whose execution is defined by the constructs specified in this Final Committee

Draft 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 Final Committee Draft and ISO/IEC 13211-1, and the implementation defined features specified by this standard.

5.4 Prolog modules

5.4.1 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**.

5.4.2 The user module

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

5.5 Documentation

A conforming Prolog processor shall be accompanied by documentation that completes the definition of every implementation defined and implementation specific feature specified in this Final Committee Draft and ISO/IEC 13211-1.

5.6 Extensions

A processor may support as an implementation specific feature, any construct that is implicitly or explicitly undefined in the Final Committee Draft and ISO/IEC 13211-1.

5.6.1 Modules

A Prolog processor may supply additional implementation-specific or user-defined modules whose exported procedures are visible within every loaded module without explicit import.

5.6.1.1 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.

5.6.1.2 Inaccessible Procedures

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

6 Syntax

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

Clause 6.1 defines the syntax of module text. Clause 6.2 defines the role of the operator ‘:’.

6.1 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
```
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
```

Clause 7.2.2 defines the module directives and the module interface directives. Clause 7.2.3 defines new directives that can appear in the body of a module and their meanings.

Table 1 — The 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`	`- \`

6.2 Terms

6.2.1 Operators

The operator table defines which atoms will be regarded as operators 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 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 7.2.3.4.

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 identifier ':' is an atom qualifier. This means that a predicate name can be a compound term provided that the functor is ':'.

3 ':' is neither a control construct nor a built-in predicate. When it appears in a goal it serves to determine the calling context.

7 Language concepts and semantics

This clause defines the semantic concepts of Prolog with modules.

a) Subclause 7.1 defines the lookup module and unqualified term associated with a term,

b) Subclause 7.2 defines the division of module text into Prolog modules,

c) Subclause 7.2.4 defines the relationship between clauses in module text and in the complete database,

d) Subclause 7.3 defines the complete database and its relation to Prolog modules,

e) Subclause 7.4 defines metapredicates and the process of name qualification,

f) Subclause 7.5 defines the process of converting terms to clauses and vice versa in the context of modules,

g) Subclause 7.6 defines the process of executing a goal in the presence of module qualification,

h) Subclause 7.7 defines the process of executing a control construct in the presence of module qualification.

i) Subclause 7.8 defines predicate properties,

j) Subclause 7.9 defines errors in addition to those required by ISO/IEC 13211-1.

7.1 Related terms

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

7.1.1 Qualified and unqualified terms

7.1.1.1 Qualified terms

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

7.1.1.2 Unqualified terms

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

7.1.1.3 Lookup module and unqualified terms associated with a term and module

Given a module `M` and a term `T`, the associated lookup module `LM = lm(M,T)` and associated unqualified term `UT = ut(M,T)` of the pair `(M,T)` are defined as follows:

a) If the principal functor of `T` is not (:)/2 then `lm(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 `lm(M,T)` is the lookup module `lm(MM,TT)`, and `ut(M,T)` is the unqualified term `ut(MM,TT)`.

NOTE — The lookup module `LM` determines the visible database to be the visible database of `LM`.

7.1.1.4 Metapredicate mode indicators

A metapredicate mode indicator is a compound term `M_Name(Modes)` each of whose arguments is ':' or '*'. An argument whose position corresponds to a ':' is a meta-argument. Arguments corresponding to '*' are not meta-arguments.

7.2 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 context 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 the built-in metapredicates (7.4.1) and the control constructs `call/1` and `catch/3` may be module qualified for the purpose of setting the calling context.

7.2.0.1 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`.

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`.

7.2.1 Module interface

A module interface in module text specifies the name of the module, the operators, character conversions and Prolog flags 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)`. (7.2.2.1)

b) The last directive is a directive `end_module(Name)`. (7.2.2.9)

c) Each other element of the sequence is a module interface directive. (7.2.2.2 through 7.2.2.8)

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

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`.

7.2.2 Module directives

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

7.2.2.1 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`.

7.2.2.2 Module interface directive export/1

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

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

NOTES

1 A predicate indicator that denotes a metapredicate may appear in `EL` in which case the corresponding metapredicate procedure is exported.

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

7.2.2.3 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 indicated by `PI`, and that `MM` makes these procedures available for import into other modules by importation from `MM`.

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

No procedure indicated by `PI` in a `reexport(M,PI)` directive in `MM` shall be the subject in `MM` of a selective `reexport(N,PI)` directive from a module `N` distinct from `M`. Neither shall it be the subject in `MM` of a selective import directive (7.2.3.2) `import(N,PI)` from a module `N` distinct from `M`.

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

7.2.2.4 Module interface directive reexport/1

A module interface directive `reexport(REM)` in the module interface of a module `M`, where `REM` is an atom, a sequence of atoms, or a list of atoms specifies that the module `M` imports all the procedures exported by the modules indicated by `REM` and that `M` makes these procedures available for import into other modules by importation from `MM`.

No procedure indicated by `REM` shall be a control construct or a built-in predicate.

7.2.2.5 Module interface directive metapredicate/1

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

NOTE — The inclusion of a metapredicate mode indicator in the argument of a module interface directive does not thereby export the indicated metapredicate. Any procedure exported by a module shall be the subject of either an export or reexport directive.

7.2.2.6 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 operator table 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.

7.2.2.7 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.

7.2.2.8 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.

7.2.2.9 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.)

7.2.3 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)` (7.2.3.1).

b) The last element of the sequence is a directive `end_body(Name)`(7.2.3.4).

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

If a procedure with 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`.

7.2.3.1 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`.

7.2.3.2 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 indicated by `PI`.

A procedure indicated by `PI` in a `import(M,PI)` directive shall be a procedure exported by the module `M`.

No procedure indicated by `PI` in a `import(M,PI)` directive in `MM` shall be the subject in `MM` of a selective import directive `import(N,PI)` from a module `N` distinct from `M`. Nor shall it be the subject in `M` of a selective reexport directive (7.2.2.3) `reexport(N,PI)` from a module `N` distinct from `M`.

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

7.2.3.3 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 indicated by `MI`. Such procedures shall be visible in `MM` without name qualification.

NOTES

1 More than one directive `import(MI, PI)` in the bodies of a module `MM` may specify the importation of exported procedures from a given module `M`. Subsequent imports of the same module `M` into `MM` in a module body of `MM` shall have no effect.

2 More than one directive `import(MI)` in the bodies of a module `MM` may specify the importation of exported predicates from a given module `M`. Subsequent imports of the same module `M` into `MM` in a module body of `MM` shall have no effect.

3 A module `M` shall not import a procedure whose predicate indicator is that of a procedure defined in `M`.

4 A module `M` shall not define a procedure whose predicate indicator is that of a procedure that `M` imports.

5 A module `M` shall not import a procedure with a given predicate indicator from two different modules.

7.2.3.4 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 flags to a new initial implementation defined 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 only affect the preparation for execution of all 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 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.

7.2.4 Clauses

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

A clause `Clause` of a clause-term `Clause.` in the body of a module `M` shall be an unqualified term which is a clause term whose head is an unqualifed term and shall satisfy the same constraints as those required for a successful execution of the built-in predicate `M:assertz(Clause)` (8.4.2), except that no error shall occur because `Clause` refers to a static procedure. The `Clause` shall be converted to a clause `h:- t` and added to the module `M`.

The predicate indicator `P/N` of the head of `Clause` 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 `M` 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.

7.2.4.1 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.

```
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).
```

7.3 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 (3.43) where the module qualification of the predicate indicator is the defining module of the procedure.

7.3.1 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 lookup context.

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

7.3.2 Examples

The following examples use the complete database defined in 7.2.4.1.

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
```

7.4 Metapredicates

Metapredicates are procedures one or more of whose arguments are meta-arguments. When the metapredicate is activated these arguments will be unified to terms that require module qualification. The calling context can be set explicitly by using the infix operator `':'`.

7.4.1 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(*,:,*)`.

7.4.2 Module name expansion

An argument `X` of a metapredicate `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. A 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`.

The meta-arguments in an unqualified term which represents a metapredicate goal `MP` in the calling context of a module `CM` shall be module qualified with `CM`. If the term is module qualified then the meta-arguments shall be module qualified with the associated lookup module of the pair `(CM,MP)`.

7.4.2.1 Module qualifying an argument list

An argument list `L` is converted in the calling context of module `M` to a module qualified argument list `MQL` as follows:

a) If `L` is the empty list then `MQL` is the empty list.

b) If `L` is the list whose head is `H` and whose tail is `T` then

1) If `H` is a meta-argument then `MQL` is the list whose head is `M:H` and whose tail is the list `MQT` obtained by converting `T` in the calling context of module `M` to a module qualified list,

2) Else `H` is not a meta-argument and `MQL` is the list whose head is `H` and whose tail is the list `MQT` obtained by converting `T` in the calling context module `M` to a module qualified list.

7.4.2.2 Module qualifying a control construct

The control constructs `','/2`, `';'/2`, `'->'/2` , `call/1` and `catch/3` require that some of their arguments be module qualified with a module `M` during conversion for visibility (7.5.2.2.) This is done as follows:

a) If an argument of `','/2`, `';'/2`, or `'->'/2` is already module qualified no qualification is done.

b) Each argument `X` of `','/2`, `';'/2`, or `'->'/2` that is not already module qualified is replaced by `M:X`.

c) If the argument of `call/1` is already module qualified no qualification is done.

d) If the argument `X` of `call/1` is not module qualified it is replaced by `M:X`.

e) If the first argument of `catch/3` is already module qualified no further module qualification of this argument is done.

f) If the first argument `X` of `catch/3` is not module qualified it is replaced by `M:X`.

g) The second argument of `catch/3` is not subject to module qualification.

h) If the third argument of `catch/3` is already module qualified no further module qualification of this argument is done.

i) If the third argument `X` of `catch/3` is not module qualified it is replaced by `M:X`.

7.4.2.3 Module qualifying a term

A term `MP` shall be converted in the calling context of module `M` to a module qualified term `MQP` as follows.

a) If `MP` is an unqualified term with principal functor `P` and argument list `L` then `MQP` is the term whose principal functor is `P` and whose argument list is the list `MQL` obtained by converting `L` in the calling context of module `M` to a module qualified list.

b) Else if `MP` is a qualified term with principal functor `(:)/2` with first argument `MM` and second argument `TT` then `MQP` is the term whose principal functor is `(:)/2` with first argument `MM` and second argument `MQTT` the term obtained by module qualifying `TT` in the calling context of module `MM` to a module qualified term.

7.4.3 Examples

7.4.3.1 Examples: Module qualification

These examples on module qualification assume that the complete database has been created from the following module text.

```
:- module(foo).
    :-export(p/2).
    :-metapredicate(p(*,:)).
:- end_module(foo).

:- module(bar).
:- end_module(bar).

:- body(bar).
    :-import(foo).
:- end_body(bar).

If p(X,Y) is called in the context of module bar
then the corresponding module qualified term is
p(X,bar:Y).

If foo:p(X,Y) is called in the context of a module m
then the corresponding module qualified term is
foo:p(X,foo:Y).
```

7.4.3.2 Examples: export and import

These examples of importation, exportation and metapredicates assume that the complete database has been created from the following module text.

```
:- 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).


baz:q(X).
   succeeds,
   unifying X with 1
   and writing bar:1
   on re-execution succeeds
   unifying X with 1
   and writing foo:2.

bar:q(X).
   succeeds,
   unifying X with 1
   and writing bar:1
   on re-execution succeeds
   unifying X with 1
   and writing foo:2.

foo:p(3).
   succeeds,
   writing foo:3.
bar:p(3).
   succeeds,
   writing bar:3.
```

7.4.3.3 Examples: metapredicates

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):-
       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).
```

7.5 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.

7.5.1 Converting a term to the head of a clause

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

a) The associated unqualified term (7.1.1.2) `UT` of `(M,T)` is converted to a predication `H` as in 7.6.1 of ISO/IEC 13211-1:

b) The lookup module `MM` for the predication is the lookup module of `(M,T)`.

7.5.2 Converting a term an activated goal

In the calling context of a module `M` with defining module `DM` a term `T` is converted to an activated goal `G` in three steps.

a) The term `T` is first converted for control constructs (7.5.2.1) in the calling context of `M` to a body `BG` with module qualifications;

b) The body with module qualifications `BG` is converted for visibility of built-ins (7.5.2.2) in the calling context of `M` to a body `VBG` qualified for visibility;

c) Terms in the body qualified for visibility `VBG` that denote metapredicates are module qualified (7.5.2.3) in the calling context of the defining module `DM`.

7.5.2.1 Conversion of a term to a module qualified body

A term `T` shall be converted for control constructs to a body with module qualifications `BG` in the context of a calling module `M` with defining module `DM` as follows:

a) If `T` is an unqualified term and `M` is equal to `DM` then `T` is converted for control constructs as follows:

1) If `T` is a variable then `BG` is the control construct call whose argument is `T`.

2) If `T` is a term whose principal functor is one of the control constructs call, catch, throw, ! , true or fail then `BG` is the same control construct and the arguments if, if any, of `BG` and `T` are identical.

3) If `T` is a term whose principal functor is one of the control one of the control constructs `(,)/2` or `(;)/2` or `(->)/2` then `BG` is the corresponding control construct and the arguments of `T` shall be converted for control constructs with calling context and defining module `M`.

4) If `T` is an atom or compound term whose principal functor `FT` does not appear in table 8 of ISO/IEC 13211-1 then `BG` is a predication whose predicate indicator is `FT`, and the arguments, if any, of `T` and `BG` are identical.

b) Else if `T` is an unqualified term and `M` is not equal to `DM` then the qualified term `M:T` is converted for control constructs, with calling context and defining module `M`.

c) Else if `T` is a qualified term the associated unqualified `ut(M,T)` is converted for control constructs in the calling context of the defining module `lm(M,T)` with defining module `lm(M,T)` to the predication `UG` and `BG` is `lm(M,T):UG`.

7.5.2.2 Conversion of a module qualified body for visibility

A module qualified body `BG` shall be converted for the visibility of control constructs in the calling context of a module `M` to a body qualified for visibility `VBG` as follows:

a) If `BG` is an unqualified term then the conversion proceeds as follows:

1) If the principal functor of `BG` is one of the control constructs `','/2`, `';'/2`, or `'->'/2` then `VBG` is the corresponding control construct and each argument of `BG` is converted for visibility in the calling context of `M`,

2) Else `BG` and `VBG` are identical.

b) Else if `BG` is a module qualified term with principal functor `(:)/2`, first argument `MM` and second argument `UBG` the conversion proceeds as follows:

1) If the principal functor of `UBG` is one of the control constructs `','/2`, `';'/2`, or `'->'/2` then `VBG` is the corresponding control construct, the arguments of `UBG` are module qualified with `MM` as in 7.4.2.2 and converted for visibility of control constructs.

2) If the principal functor of `UBG` is the control construct `call/1` or `catch/3` then `VBG` is the same control construct and the arguments of `UBG` are module qualified according to 7.4.2.2,

3) Else `BG` and `VBG` are identical.

7.5.2.3 Converting a body qualified for visibility to an activated goal

A body `VBG` qualified for visibility can be converted to an activated goal `AG` in the calling context of a module `M` with defining module `DM` as follows.

a) If VBG is a term whose principal functor is one of the control constructs ','/2, ';'/2, or '->'/2 then each argument of VBG is converted to an activated goal in the in the calling context of M.

b) If VBG is an unqualified term denoting a metapredicate then AG is the result of module qualifying (7.4.2.3) the arguments of VBG with DM.

c) If VBG is a qualified term denoting a metapredicate then AG1 is the unqualified term obtained by module qualifying the arguments of the unqualified term VBG1 associated to (M,VBG1) with the lookup module lm(M,VBG1), if the metapredicate denoted by VBG1 is one of the metapredicate built-ins (7.4.1) then AG is AG1 else AG is lm(M,VBG1):AG1,

d) Else VBG and BG are identical.

7.5.3 Converting a term to the body of a clause

It is implementation defined as to whether all of the steps in conversion of a term to an activated goal take place when a term is converted to a goal. Converting a term to a goal shall convert the term for control constructs (7.5.2.1) but may also convert for visibility (7.5.2.2). Conversion to an activated goal (7.5.2.3) must be completed (if not done at conversion to a goal time) by 7.6.4 e.

7.5.4 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 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.

c) 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.

d) 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.

NOTE — A fully activated goal is not subject to further module qualifcation of its arguments.

7.5.5 Examples

The following examples are provided to illustrate the three stages of converting a term to a fully activated goal.

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

Term -  baz(X),  baz/1 not a metapredicate.
Converted for control constructs -  m:baz(X).
Converted for visibility -      m:baz(X).
Fully activated goal -  m:baz(X).

Term  - metabaz(X),  metabaz a metapredicate
Converted for control constructs:-  m:metabaz(X).
Converted for visibility   -  m:metabaz(X).
Fully activated goal -  m:metabaz(m:X).

Term -  X
Converted for control constructs -  m:call(X)
Converted for visibility   -   call(m:X).
Fully activated goal     -     call(m:X).

Term - '->'(bar:a(X), b(Y)).
Converted for control constructs
          -  '->'(bar:a(X), m:b(Y))
Converted for visibility
          -  '->'(bar:a(X), m:b(Y))
Fully activated goal
          -  '->'(bar:a(X), m:b(Y))

Term -  ','(setof(X,G,S)), write(S)).
Converted for control constructs
          - ','(m:setof(X,G,S)), m:write(S)).
Converted for visibility
          - ','(m:setof(X,G,S)), write(S)).
Fully activated goal
          - ','(setof(X,m:G,S)), write(S)).

Term -  true.
Converted for control constructs  - true.
Converted for visibility - true.
Fully activated goal -  true.
```

7.6 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.

7.6.1 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 $\mathtt{ES}_{i+1}$ is $\mathtt{ES}_i$.

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

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

`contextmodule` – An atom identifying the module in which the `subgoal` 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

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,)

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

`S_` `index`	Decorated Subgoal Stack,	Substi- tution	BI
1	((`m:goal`, user, 0), $\mathtt{newstack}_{DS}$) , $\mathtt{newstack}_{ES}$	{}	`nil`

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.

7.6.2 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. Execution of a metapredicate requires that all arguments of type ':' be module qualified (7.4.2) with the module name of the calling context prior to execution (7.6.4 e).

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`.

7.6.2.1 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.

7.6.2.2 A goal fails

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

7.6.2.3 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 7.6.5.

7.6.3 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.

7.6.3.1 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 defined 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 7.6.3.1 f.

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 (7.2.3) 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.

7.6.3.2 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 lookup module `LT` associated to `(m,T)`.

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

c) If the principal functor of `UT` is a user-defined procedure `q` (not a control construct or built-in predicate) then the visible database (7.3.1) of `LT` is searched for a procedure `q`. If no such procedure exists the search fails.

7.6.4 Selecting a clause for execution

Execution proceeds in a succession of steps.

a) Using the visible database given by the lookup module `contextmodule` of the current decorated sub-goal `currdecsgl`, the processor searches the complete database (7.6.3) 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 7.6.4 a, 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 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) Set `contextmodule` in the current decorated sub-goal to the lookup module associated to the pair `(contextmodule, curract)` and set `curract` to the associated unqualified term.

e) Ensure that `curract` has been converted to an activated goal by module qualifying (7.4.2.3) its meta-arguments.

f) 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 (7.7).

g) If `p` is a built-in predicate `BP` then `BI` is set to `bip` and continue execution at 7.6.7.

h) 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 7.6.6

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

7.6.5 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 7.6.6.

c) If `BI` is `bip` then `p` is a built-in predicate, continue execution at 7.6.7.

d) If `BI` is `ctrl` the effect of re-executing it is defined in 7.7.

e) If `BI` is `nil` then the new `curract` has not been executed, continue execution at 7.6.4.

7.6.6 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 7.6.6.1,

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 7.6.6 e,

d) Else `BI` is replaced by a value `(DM, up(CT))` and continue execution at 7.6.6 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 7.6.4.

7.6.6.1 Executing a user-defined procedure with no more clauses

When a user-defined procedure has been selected for execution 7.6.4 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 7.6.5.

7.6.7 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 required for module name expansion (7.6.4 e).

7.7 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.

7.7.1 call/1

7.7.1.1 Description

`call(G)` is true in the calling context of module `CM` iff `G` represents a goal which is true.

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 lookup module associated to `(CM,G)` (7.5.3).

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 7.6.4.

k) Pop `currstate` from `S`.

l) Continue execution at 7.6.5.

`call(G)` is re-executable. On backtracking, continue at 7.7.1.1 k.

NOTE — The built-in predicates `once/1` and `\+/1` contain an implicit `call`, their behaviour in the presence of modules is modified accordingly.

7.7.1.2 Template and modes

`call(+callable_term)`.

7.7.1.3 Errors

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

b) The lookup module of `(CM,G)` cannot be determined (7.1.1).
– `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)`.

7.7.1.4 Examples

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

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

7.7.2 catch/3

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

7.7.2.1 Description

`catch(G,C,R)` is true in the calling context of module `CM` iff (1) `call(G)` is true , or (2) the call of `G` is interrupted by a call of `throw/1` whose argument unifies with `C`, and `call(R)` is true.

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 7.6.4.

f) Pop `currstate` from `S`.

g) Continue execution at 7.6.5.

`catch(G,C,R)` is re-executable. On backtracking, continue at 7.7.2.1 f.

7.7.2.2 Template and modes

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

7.7.2.3 Errors

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

b) The lookup module of `(CM,G)` cannot be determined (7.1.1).
– `instantiation_error`.

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

7.7.3 throw/1

7.7.3.1 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 7.7.2).

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 7.7.3.1 b .

e) Apply `MGU` to `currentgoal`.

f) Replace `curract` by `call(R)`, where `R` is the third argument of the control construct `catch/3` from 7.7.3.1 d.

g) Set `BI` to `nil`.

h) Continue execution at 7.6.4.

7.7.3.2 Template and modes

`throw(+nonvar)`

7.7.3.3 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`.

7.8 Predicate properties

The properties of proceduress can be found using the built-in predicate `predicate_property(Callable, Property)`, where `Callable` is the meta-argument term `Module:Goal` 8.2.2. 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 `DefiningModule` defines the procedure.

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

7.9 Errors

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

7.9.1 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.)

8 Built-in predicates

8.1 The format of built-in predicate definitions

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

8.1.1 Type of an argument

The following additional argument types are required:

`metapredicate_mode_indicator` – as terminology.

`predicate_property` – a procedure property (7.8).

`prototype` – as terminology.

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

8.2 Module predicates

The examples provided for these built-in predicates assume the complete database has been created from the module text given in the first example of 7.4.3.2.

8.2.1 current_module/1

8.2.1.1 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 8.2.1.1 d,

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 8.2.1.1 e.

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.

8.2.1.2 Template and Modes

`current_module(?atom)`

8.2.1.3 Errors

a) Module is neither a variable nor an atom

— `type_error(atom, Module)`.

8.2.1.4 Examples

```
current_module(foo).
    succeeds.

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

8.2.2 predicate_property/2

8.2.2.1 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 lookup module `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 8.2.2.1 f,

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 8.2.2.1 g.

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

8.2.2.2 Template and modes

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

8.2.2.3 Errors

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

b) The lookup module of `(M,Prototype)` cannot be determined (7.1.1)
– `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)`.

8.2.2.4 Examples

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

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

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

bar:predicate_property(X:foo:p(Y), exported).
   instantiation_error.
```

8.3 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).
```

8.3.1 clause/2

8.3.1.1 Description

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

– The associated unqualified term of `(M,Head)` is `HH`, (7.1.1.3),

– The procedure of `HH` is public, and

– There is a clause in the the lookup module `DM` associated with `(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 lookup module `DM` associated with `(M, Head)` (7.1.1.3) 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 to a term `H` and whose body can be converted 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 8.3.1.1 f,

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 8.3.1.1 g.

8.3.1.2 Template and modes

`clause(+term, ?callable_term)`

8.3.1.3 Errors

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

b) The lookup module of `(M,Head)` cannot be determined (7.1.1.3)
– `instantiation_error`.

c) `Head` is a qualified term and either the associated unqualified term or lookup module is a variable
– `instantiation_error`.

d) `Head` is neither a variable nor a predication
– `type_error(callable, Head)`.

e) `Head` cannot be converted to a predication.
– `type_error(callable, Head)`.

f) The predicate indicator `Pred` of the associated unqualified term of `Head` is that of a private procedure
– `permission_error(access, private_procedure, Pred)`.

g) The predicate indicator `Pred` of the associated unqualified term of `Head` is that of a procedure imported or re-exported by `DM`
– `permission_error(access, implicit, Pred )`.

h) `Body` is neither a variable nor a callable term
– `type_error(callable, Body)`.

i) The module identified by `DM` does not exist
– `existence_error(module, DM)`.

8.3.1.4 Examples

The examples amplify those of ISO/IEC 13211-1 by illustrating the effect of the module structure.

```
insects:clause(legs(X) , A).
   succeeds unifying X with 6
   and A with true.

insects:clause(body_type(X), true).
   succeeds unifying X with segmented.

animals:clause(limbs(X) , B).
   succeeds unifying B with insects:legs(X)
   on re-execution unifies B with mammals:legs(X).

clause(insects:legs(X) , A).
   succeeds unifying X with 6
   and A with true.

animals:clause(elk(X), B).
    permission_error(access, implicit, elk).

animals:predicate_property(elk(_), defined_in(M)),
M:clause(elk(Y), B).
    succeeds,  M is unified with mammals,
    B is unified with moose(Y).

animals:clause(mammals:elk(X), B).
   succeeds, B is unified with
   moose(X).

clause(insects:M:legs(X), A).
   instantiation_error.
```

8.3.2 current_predicate/1

8.3.2.1 Description

`current_predicate(PI)` is true in the calling context of a module `M`, iff `PI` is a predicate indicator for one of the user-defined procedures in the visible database of `M`.

Procedurally, `current_predicate(PI)` is executed as follows:

a) Searches the visible database of `M` and creates a set Set_{AN} of terms `A/N` such that (1) the visible database contains a user-defined procedure whose predicate has identifier `A` and arity `N`, and (2) `A/N` identifies with `PI`.

b) If a non-empty set is found, then proceeds to 8.3.2.1 d,

c) Else the goal fails.

d) Chooses a member of Set_{AN} and the goal succeeds.

e) If all members of Set_{AN} have been chosen, then the goal fails,

f) Else chooses a member of Set_{AN} which has not already been chosen, and the goal succeeds.

`current_predicate(PI)` is re-executable. On backtracking continue at 8.3.2.1 e.

The order in which predicate indicators are found by `current_predicate(PI)` is implementation dependent.

8.3.2.2 Template and modes

`current_predicate(?predicate_indicator)`

8.3.2.3 Errors

a) `PI` is neither a variable nor a predicate indicator – `type_error(predicate_indicator, PI)`.

8.3.2.4 Examples

```
insects:current_predicate(legs/1).
    Succeeds.

animals:current_predicate(ant/X).
    Succeeds unifying X with 0.

animals:current_predicate(legs/1).
    Fails.
```

8.4 Database access and modification

This clause describes the interaction of the predicates `asserta/1`, `assertz/1`, `retract/1` and `abolish/1` with the module system.

8.4.1 asserta/1

8.4.1.1 Description

`asserta(Clause)` is true.

Procedurally, `asserta(Clause)` is executed in the calling context of a module `M` as follows:

a) Extracts the unqualified term `C` and associated lookup module `CM` from `(M, Clause)` (7.1.1.3).

b) If `C` unifies with `’:-’(Head, Body)` proceeds to 8.4.1.1 d,

c) Else unifies `Head` with `C` and `true` with `Body`.

d) Converts (7.5.1) the term `Head` to a head `H` with lookup module `DM` in calling context `CM`.

e) Converts (7.5.3) the term `Body` to a body `B` in calling context `CM` with defining module `DM`.

f) Constructs the clause with head `H` and body `B`.

g) Adds the clause to the selected module `DM` before all existing clauses of the procedure in `DM` whose predicate is equal to the functor of `Head`.

h) The goal succeeds.

8.4.1.2 Template and modes

`asserta(@qualified_or_unqualified_clause)`

8.4.1.3 Errors

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

b) `DM` is a variable
– `instantiation_error`.

c) The lookup module of `(M,Clause)` cannot be determined (7.1.1.3)
– `instantiation_error`.

d) `Head` cannot be converted to a predication
– `type_error(callable, Head)`.

e) `Body` cannot be converted to a goal
– `type_error(callable, Body)`.

f) The predicate indicator `Pred` of `Head` is that of a static procedure
– `permission_error(modify, static_procedure, Pred)`.

g) The procedure identified by `Pred` is imported or re-exported by the module `DM`
– `permission_error(modify, implicit, Pred)`.

h) The module identified by `DM` does not exist
– `existence_error(module, DM)`.

8.4.1.4 Examples

```
mammals:asserta((moose(fred)).
    succeeds adding moose(fred) to the
    module mammals.


animals:asserta((elk(X) :- new_moose(X))).
     permission_error(modify, implicit, elk).

animals:predicate_property(elk(_), defined_in(M)),
   M:asserta(elk(joe)).
     succeeds adding elk(joe) to
     the module mammals,
     M is unified with mammals.

nomodule:asserta(foo(3)).
    existence_error(module, nomodule).

asserta(mammals:elk(anna)).
   succeeds adding elk(anna) to
   the module mammals.

mammals:asserta(animals:horns(X) :- moose(X)).
   succeeds adding horns(X) :-  mammals:moose(X)
   to the module animals.

asserta(M:mammals:elk(joe)).
   type_error(instantiation_error).
```

After these examples the complete database could have been created from the following module text.

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

:- body(mammals).

   :- dynamic(cat/0).
   cat.

   :- dynamic(dog/0).
   dog :- true.

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

   :- dynamic(moose/1).
   :- moose(fred).
   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).

 horns(X) :-  mammals:moose(X).


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

:- end_body(animals).
```

8.4.2 assertz/1

8.4.2.1 Description

`assertz(Clause)` is true.

Procedurally, `assertz(Clause)` is executed in the calling context of module `M` as follows:

a) Extracts the unqualified term `C` and associated lookup module `LM` from `(M, Clause)` (7.1.1.3).

b) If `C` unifies with `':-'(Head, Body)` proceeds to 8.4.1.1 d,

c) Else unifies `Head` with `C` and `true` with `Body`.

d) Converts (7.5.1) the term `Head` to a head `H` and lookup module `DM` in calling context `LM`.

e) Converts (7.5.3) the term `Body` to a body `B` in calling context `LM` with defining module `DM`.

f) Constructs the clause with head `H` and body `B`.

g) Adds the clause to the selected module `DM` after all existing clauses of the procedure in `DM` whose predicate is equal to the functor of `Head`.

h) The goal succeeds.

8.4.2.2 Template and modes

`assertz(@qualified_or_unqualified_clause)`

8.4.2.3 Errors

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

b) `DM` is a variable
– `instantiation_error`.

c) The lookup module of `(M, Clause)` cannot be determined (7.1.1)
– `instantiation_error`.

d) `Head` cannot be converted to a predication
– `type_error(callable, Head)`.

e) `Body` cannot be converted to a goal
– `type_error(callable, Body)`.

f) The predicate indicator `Pred` of `Head` is that of a static procedure
– `permission_error(modify, static_procedure, Pred)`.

g) The procedure identified by `Pred` is imported or re-exported by the module `DM`
– `permission_error(modify, implicit, Pred)`.

h) The module identified by `DM` does not exist
– `existence_error(module, DM)`.

8.4.3 retract/1

8.4.3.1 Description

`retract(Clause)` is true in the calling context of a module `M` iff:

– The associated unqualified term of `(M,Clause)` is `C` with lookup module `DM` (7.1.1.3),

– The complete database contains at least one dynamic procedure with defining module `DM` and with a clause `Head :- Body` which unifies with `C`.

Procedurally `retract(Clause)` is executed in the calling context of a module `M` as follows:

a) Determines the lookup module `DM` associated with `(M, Clause)` (7.1.1.3) to be searched for the clauses.

b) Determines the unqualified term `C` and lookup module `L1` associated with `(M, Clause)`.

c) If `C` unifies with `':-'(HH, BB)` proceeds to 8.4.3.1 g,

d) Else unifies `C` with `HH` and `true` with `BB`.

e) Determines the unqualified term `Head` and lookup module `DM` associated with `(L1, HH)`.

f) Determines the unqualifed term `Body` and the lookup module `BM` associated to `(L1,BB`.

g) Chooses the module `DM` as the defining module to search.

h) Searches sequentially through each dynamic user-defined open procedure in `DM` and creates a list `L` of all the terms `clause(H,B)` such that: (1) the module `DM` contains a clause whose head can be converted to a term `HH` and whose body can be converted with context module `BM` and defining module `DM` to a goal `B`, (2) `H` unifies with `Head`, and (3) `B` unifies with `Body`.

i) If a non-empty list is found, then proceeds to 8.4.3.1 k,

j) Else the goal fails.

k) Chooses the first element of the list `L`, removes the clause corresponding to it from the defining module `DM`, and the goal succeeds.

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

m) Else chooses the first element of the list `L` which has not already been chosen, removes the clause, if it exists, corresponding to it from the defining module `DM` and the goal succeeds.

`retract/1` is re-executable. On backtracking, continue at 8.4.3.1 l.

8.4.3.2 Template and modes

`retract(+qualified_or_unqualified_clause)`

8.4.3.3 Errors

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

b) `DM` is a variable
– `instantiation_error`.

c) The lookup module of `(M,Clause)` cannot be determined (7.1.1)
– `instantiation_error`.

d) `Head` is not a predication
– `type_error(callable, Head)`.

e) `Body` cannot be converted to a goal
– `type_error(callable, Body)`.

f) The predicate indicator `Pred` of `Head` is that of a private procedure
– `permission_error(modify, static_procedure, Pred)`.

g) The procedure identified by `Pred` is imported or re-exported by the module `DM`
–`permission_error(modify, implicit, Pred)`.

h) The module identified by `DM` does not exist
– `existence_error(module, DM)`.

8.4.3.4 Examples

The following examples assume that the complete database has been created from the module text in subclause (8.4.1.4)

```
mammals:retract(cat).
   succeeds.


animals:predicate_property(ant, defined_in(M)),
    M:retract(ant).
    succeeds.

retract(animals:dog).
   succeeds.

retract(M:animals:cat).
   type_error(instantiation_error).

retract(nomodule:foo(bar)).
   existence_error(module, nomodule).
```

After these examples the complete database could have been created from the following module text:

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

:- body(mammals).

   :- dynamic(cat/0).


   :- dynamic(dog/0).

   :- 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).

   :- dynamic(ant/0).


   :- 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).
```

8.4.4 abolish/1

8.4.4.1 Description

`abolish(Pred)` is true.

Procedurally, `abolish(Pred)` is executed in the calling context of a module `M` as follows:

a) Determines the lookup module `DM` of `(M, Pred)`.

b) Determines the unqualified term `PI` of `(M,Pred)`.

c) If the module `DM` defines a dynamic procedure whose predicate indicator is `PI`, then proceeds to 8.4.4.1 e,

d) Else the goal succeeds.

e) Removes from the module `DM` the procedure specified by `PI` and all its clauses, and the goal succeeds.

8.4.4.2 Template and modes

`abolish(@predicate_indicator)`

8.4.4.3 Errors

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

b) `DM` is a variable
– `instantiation_error`.

c) The lookup module `DM` of `(M,Pred)` cannot be determined (7.1.1).
–`instantiation_error`.

d) `PI` is a term `Name/Arity` and at least one of `Name`, or `Arity` is a variable,
– `instantiation_error.`

e) `PI` is neither a term nor a predicate indicator
–`type_error(predicate_indicator, PI)`.

f) `PI` is a term `Name/Arity` and `Arity` is neither a variable nor an integer
– `type_error(integer, Arity).`

g) `PI` is a term `Name/Arity` and `Name` is neither a variable nor an atom
– `type_error(atom, Name).`

h) `PI` is a term `Name/Arity` and `Arity` is an integer less than zero
– `domain_error(not_less_than_zero, Arity).`

i) `PI` is a term `Name/Arity` and `Arity` is an integer greater than the implementation defined integer `max_arity`
– `representation_error(max_arity).`

j) The predicate indicator `PI` is that of a prodedure which is static
– `permission_error(modify, static_procedure, Pred).`

k) `PI` is a term `Name/Arity` and the procedure identified by `Name` is imported or re-exported by `DM`
– `permission_error(modify, implicit, Name).`

l) The module identified by `DM` does not exist
– `existence_error(module, DM)`.

8.4.4.4 Examples

```
insects:abolish(bee/0).
       succeeds removing insects:bee
       from the complete database.


animals:abolish(dog/0).
     permission_error(modify, implicit, dog/0).

insects:abolish(X:mammal:legs/2)
     instantiation_error.
```