INTERNATIONAL STANDARD

ISO/IEC 13211-2

First edition
2000-06-01

Technologies de l'information — Langages de programmation — Prolog — Partie 2: Modules

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

Information technology — Programming languages — Prolog —

Part 2:
Modules

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

Page

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.*

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*.

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

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,

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

_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.