



INTERNATIONAL STANDARD ISO/IEC 13211-1:1995
TECHNICAL CORRIGENDUM 2

Published 2012-02-15

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОМИССИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

Information technology — Programming languages — Prolog —

Part 1:

General core

TECHNICAL CORRIGENDUM 2

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

Partie 1: Noyau général

RECTIFICATIF TECHNIQUE 2

Technical Corrigendum 2 to ISO/IEC 13211-1:1995 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Information technology - Programming languages - Prolog - Part 1: General Core TECHNICAL CORRIGENDUM 2

Allow bar character | as infix operator, forbid '{ }' and '[]' as operators.

6.3.4.3 Operators

Add prior to syntax rules:

A bar (6.4) shall be equivalent to the atom ' | ' when ' | ' is an operator.

Add the syntax rule:

`op = bar ;`

Abstract: |

Priority: *n* *n*

Specifier: *s* *s*

Condition: ' | ' is an operator

Add at the end of 6.3.4.3 before NOTES:

There shall not be an operator ' { } ' or ' [] '.

An operator ' | ' shall be only an infix operator with priority greater than or equal to 1001.

Add to note 1

Bar is also a solo character (6.5.3), and a token (6.4) but not an atom.

Replace note 3

3 The third argument of `op/3` (8.14.3) may be any atom except ' , ' so the priority of the comma operator cannot be changed.

by

3 The third argument of `op/3` (8.14.3) may be any atom except ' , ' , ' [] ' , and ' { } ' so the priority of the comma operator cannot be changed, and so empty lists and curly bracket pairs cannot be declared as operators.

6.3.4.4

Add in Table 7 - The operator table:

Priority Specifier Operator(s)

400 yfx div

200 fy +

6.4 Tokens

Add as the last syntax rule:

```
bar (* 6.4 *)
  = [ layout text sequence (* 6.4.1 *) ] ,
    bar token (* 6.4.8 *) ;
```

6.4.8 Other tokens

Add as the last syntax rule:

```
bar token (* 6.4.8 *)
  = bar char (* 6.5.3 *) ;
```

6.5.3 Solo characters

Add alternative for solo char:

```
| bar char (* 6.5.3 *)
```

Add as the last syntax rule:

```
bar char (* 6.5.3 *) = "|" ;
```

Add the new subclause into the place indicated by its number:

7.1.1.5 Witness variable list of a term

The witness variable list of a term T is a list of variables and a witness of the variable set (7.1.1.2) of T . The variables appear according to their first occurrence in left-to-right traversal of T .

NOTES

1 For example, $[X, Y]$ is the witness variable list of each of the terms $f(X, Y)$, $X+Y+X+Y$, $X+Y+X$, and $X*Y+X*Y$.

2 The concept of a witness variable list of a term is required when defining `term_variables/2` (8.5.5).

Add the new subclause into the place indicated by its number:

7.1.6.9 List prefix of a term

LP is a list prefix of a term P if:

a) LP is an empty list, or

b) P is a compound term whose principal functor is the list constructor and the heads of LP and P are identical, and the tail of LP is a list prefix of the tail of P .

NOTE — For example, $[]$, $[1]$, and $[1, 2]$ are all list prefixes of $[1, 2, 3]$, $[1, 2|X]$, and $[1, 2|nonlist]$.

Correct example for call/1.

7.8.3.4 example no. 6

For program

```
b(X) :-
    Y = (write(X), X),
    call(Y).
replace
b(3).
    Outputs '3', then
    type_error(callable, 3).
by
b(3).
    type_error(callable, (write(3), 3)).
```

Adjust Template and Modes of catch/3, remove error conditions. In this manner all errors of the goal are caught by catch/3.

7.8.9 catch/3

Replace

7.8.9.2 Template and modes

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

7.8.9.3 Errors

a) G is a variable
— `instantiation_error`.
b) G is neither a variable nor a callable term
— `type_error(callable, G)`
by

7.8.9.2 Template and modes

```
catch(goal, ?term, goal)
```

7.8.9.3 Errors

None.

7.9.1 Description (Evaluating an expression)

Replace 7.9.1 Note 1

1 An error occurs if *T* is an atom or variable.

by

1 An error occurs if *T* is a variable or if there is no operation *F* in step 7.9.1 c).

7.9.2 Errors (Evaluating an expression)

Replace error condition i and j which both were added in Technical Corrigendum 1.

i) The value of an argument *Culprit* is not a member of the set *I*

— `type_error(integer, Culprit)`.

j) The value of an argument *Culprit* is not a member of the set *F*

— `type_error(float, Culprit)`.

by

i) *E* is a compound term with no corresponding operator in step 7.9.1 c but there is an operator corresponding to the same principal functor with different types such that

a) the *i*-th argument of the corresponding operator has type *Type*, and

b) the value *Culprit* of the *i*-th argument of *E* has a different type

— `type_error(Type, Culprit)`.

Add new error class, new types, and new domain.

7.12.2 Error classification

Remove in subclause b variable from the enumerated set ValidType and add pair to the set ValidType. Add in subclause c order to the set ValidDomain.

Add additional error class:

k) There shall be an Uninstantiation Error when an argument or one of its components is not a variable, and a variable or a component as variable is required. It has the form `uninstantiation_error(Culprit)` where *Culprit* is the argument or one of its components which caused the error.

8.1.3 Errors (The format of built-in predicate definitions)

Replace in Note 5

5 When a built-in predicate has a single mode and template, an argument whose mode is – is always associated with an error condition: a type error when the argument is not a variable.

the words

a type error

by

an un instantiation error

Add testing built-in predicate subsumes_term/2.

Add the new subclauses into the place indicated by their number:

8.2.4 subsumes_term/2

This built-in predicate provides a test for syntactic one-sided unification.

8.2.4.1 Description

`subsumes_term(General, Specific)` is true iff there is a substitution θ such that

a) `General $\theta$` and `Specific $\theta$` are identical, and

b) `Specific $\theta$` and `Specific` are identical.

Procedurally, `subsumes_term(General, Specific)` simply succeeds or fails accordingly. There is no side effect or unification.

8.2.4.2 Template and modes

`subsumes_term(@term, @term)`

8.2.4.3 Errors

None.

8.2.4.4 Examples

```
subsumes_term(a, a).  
Succeeds.
```

```
subsumes_term(f(X,Y), f(Z,Z)).  
Succeeds.
```

```
subsumes_term(f(Z,Z), f(X,Y)).  
Fails.
```

```
subsumes_term(g(X), g(f(X))).  
Fails.
```

```
subsumes_term(X, f(X)).  
Fails.
```

```
subsumes_term(X, Y), subsumes_term(Y, f(X)).  
Succeeds.
```

NOTES

1 The final two examples show that `subsumes_term/2` is not transitive. A transitive definition corresponding to the term-lattice partial order is `term_instance/2` (3.95).

```
term_instance(Term, Instance) :-
    copy_term(Term, Copy),
    subsumes_term(Copy, Instance).
```

```
term_instance(g(X), g(f(X))).
Succeeds.
```

2 Many existing processors implement a built-in predicate `subsumes/2` which unifies the arguments. This often leads to erroneous programs. The following definition is mentioned only for backwards compatibility.

```
subsumes(General, Specific) :-
    subsumes_term(General, Specific),
    General = Specific.
```

Add testing built-in predicates `callable/1`, `ground/1`, `acyclic_term/1`.

Add the new subclauses into the place indicated by their number:

8.3.9 callable/1

8.3.9.1 Description

`callable(Term)` is true iff `Term` is a callable term (3.24).

NOTE — Not every callable term can be converted to the body of a clause, for example `(1, 2)`.

8.3.9.2 Template and modes

```
callable(@term)
```

8.3.9.3 Errors

None.

8.3.9.4 Examples

```
callable(a).
Succeeds.
```

```
callable(3).
Fails.
```

```
callable(X).
Fails.
```

```
callable((1, 2)).
Succeeds.
```

8.3.10 ground/1

8.3.10.1 Description

`ground(Term)` is true iff `Term` is a ground term (3.82).

8.3.10.2 Template and modes

`ground(@term)`

8.3.10.3 Errors

None.

8.3.10.4 Examples

```
ground(3).  
Succeeds.  
  
ground(a(1, _)).  
Fails.
```

8.3.11 `acyclic_term/1`

8.3.11.1 Description

`acyclic_term(Term)` is true iff `Term` is acyclic, that is, it is a variable or a term instantiated (3.96) with respect to the substitution of a set of equations not subject to occurs check (7.3.3).

8.3.11.2 Template and modes

`acyclic_term(@term)`

8.3.11.3 Errors

None.

8.3.11.4 Examples

```
acyclic_term(a(1, _)).  
Succeeds.  
  
X = f(X), acyclic_term(X).  
Undefined.  
[STO 7.3.3, does not succeed in many implementations,  
but fails, produces an error, or loops]
```

Add built-in predicates `compare/3`, `sort/2`, `keysort/2` based on term order.

8.4 Term comparison, 8.4.1

Move the two paragraphs from subclause 8.4 to subclause 8.4.1. Add into subclause 8.4:

These built-in predicates compare and sort terms based on the ordering of terms (7.2).

Add the new subclauses into the place indicated by their number:

8.4.2 compare/3 – three-way comparison

8.4.2.1 Description

`compare(Order, X, Y)` is true iff `Order` unifies with `R` which is one of the following atoms:
 '=' iff `X` and `Y` are identical terms (3.87), '<' iff `X term_precedes Y` (7.2), and '>' iff `Y term_precedes X`.

Procedurally, `compare(Order, X, Y)` is executed as follows:

- a) If `X` and `Y` are identical, then let `R` be the atom '=' and proceeds to 8.4.2.1 d.
- b) Else if `X term_precedes Y` (7.3), then let `R` be the atom '<' and proceeds to 8.4.2.1 d.
- c) Else let `R` be the atom '>'.
- d) If `R` unifies with `Order`, then the goal succeeds.
- e) Else the goal fails.

8.4.2.2 Template and modes

```
compare(-atom, ?term, ?term)
compare(+atom, @term, @term)
```

8.4.2.3 Errors

- a) `Order` is neither a variable nor an atom
 — `type_error(atom, Order)`.
- b) `Order` is an atom but not <, =, or >
 — `domain_error(order, Order)`.

8.4.2.4 Examples

```
compare(Order, 3, 5).
    Succeeds, unifying Order with (<).

compare(Order, d, d).
    Succeeds, unifying Order with (=).

compare(Order, Order, <).
    Succeeds, unifying Order with (<).

compare(<, <, <).
    Fails.

compare(1+2, 3, 3.0).
```