



**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 sole 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  $\theta$  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).
```

```

type_error(atom, 1+2).

compare(>=, 3, 3.0).
domain_error(order, >=).

```

### 8.4.3 sort/2

#### 8.4.3.1 Description

`sort(List, Sorted)` is true iff `Sorted` unifies with the sorted list of `List` (7.1.6.5).

Procedurally, `sort(List, Sorted)` is executed as follows:

- a) Let `SL` be the sorted list of list `List` (7.1.6.5).
- b) If `SL` unifies with `Sorted`, then the goal succeeds.
- c) Else the goal fails.

NOTE — The following definition defines the logical and procedural behaviour of `sort/2` when no error conditions are satisfied and assumes that `member/2` is defined as in 8.10.3.4.

```

sort([], []).
sort(List, Sorted) :-
    setof(X, member(X, List), Sorted). /* 8.10.3, 8.10.3.4 */

```

#### 8.4.3.2 Template and modes

```

sort(@list, -list)
sort(+list, +list)

```

#### 8.4.3.3 Errors

- a) `List` is a partial list  
— `instantiation_error`.
- b) `List` is neither a partial list nor a list  
— `type_error(list, List)`.
- c) `Sorted` is neither a partial list nor a list  
— `type_error(list, Sorted)`.

#### 8.4.3.4 Examples

```

sort([1, 1], Sorted).
Succeeds unifies Sorted with [1].

sort([1+Y, z, a, V, 1, 2, V, 1, 7.0, 8.0, 1+Y, 1+2,
      8.0, -a, -X, a], Sorted).
Succeeds, unifying Sorted with
[V, 7.0, 8.0, 1, 2, a, z, -X, -a, 1+Y, 1+2]

sort([X, 1], [1, 1]).
Succeeds, unifying X with 1.

```

```
sort([1, 1], [1, 1]).
  Fails.
```

```
sort([V], V).
  Undefined.
  [STO 7.3.3, corresponds to the goal [V] = V. In many
  implementations this goal succeeds and violates
  the mode sort(@list, -list).]
```

```
sort([f(U), U, U, f(V), f(U), V], L).
  Succeeds unifying L with [U, V, f(U), f(V)] or
  [V, U, f(V), f(U)].
  [The solution is implementation dependent.]
```

## 8.4.4 keysort/2

### 8.4.4.1 Description

`keysort(Pairs, Sorted)` is true iff `Pairs` is a list of compound terms with principal functor `(-)/2` and `Sorted` unifies with a permutation `KVs` of `Pairs` such that the `Key` entries of the elements `Key-Value` of `KVs` are in weakly increasing term order (7.2). Elements with an identical `Key` appear in the same relative sequence as in `Pairs`.

Procedurally, `keysort(Pairs, Sorted)` is executed as follows:

- Let `Ts` be the sorted list (7.1.6.5) containing as elements terms `t(Key, P, Value)` for each element `Key-Value` of `Pairs` with `P` such that `Key-Value` is the `P`-th element in `Pairs`.
- Let `KVs` be the list with elements `Key-Value` occurring in the same sequence as elements `t(Key, _, Value)` in `Ts`.
- If `KVs` unifies with `Sorted`, then the goal succeeds.
- Else the goal fails.

NOTE — The following definition defines the logical and procedural behaviour of `keysort/2` when no error conditions are satisfied. The auxiliary predicate `numbered_from/2` is not needed in many existing processors because `Ps` happens to be a sorted list of variables.

```
keysort(Pairs, Sorted) :-
  pairs_ts_ps(Pairs, Ts, Ps),
  numbered_from(Ps, 1),
  sort(Ts, STs),          /* 8.4.3 */
  pairs_ts_ps(Sorted, STs, _).

pairs_ts_ps([], [], []).
pairs_ts_ps([Key-Value|Pairs], [t(Key,P,Value)|Ts], [P|Ps]) :-
  pairs_ts_ps(Pairs, Ts, Ps).

numbered_from([], _).
numbered_from([I0|Is], I0) :-
  I1 is I0 + 1,
  numbered_from(Is, I1).
```

### 8.4.4.2 Template and modes

```
keysort(@list, -list)
keysort(+list, +list)
```

### 8.4.4.3 Errors

- a) Pairs is a partial list  
— instantiation\_error.
- b) Pairs is neither a partial list nor a list  
— type\_error(list, Pairs).
- c) Sorted is neither a partial list nor a list  
— type\_error(list, Sorted).
- d) An element of a list prefix of Pairs is a variable  
— instantiation\_error.
- e) An element E of a list prefix of Pairs is neither a variable nor a compound term with principal functor (-)/2  
— type\_error(pair, E).
- f) An element E of a list prefix of Sorted is neither a variable nor a compound term with principal functor (-)/2  
— type\_error(pair, E).

### 8.4.4.4 Examples

```
keysort([1-1, 1-1], Sorted).
    Succeeds unifying Sorted with [1-1, 1-1].

keysort([2-99, 1-a, 3-f(_), 1-z, 1-a, 2-44], Sorted).
    Succeeds unifying Sorted with [1-a, 1-z, 1-a,
    2-99, 2-44, 3-f(_)].

keysort([X-1, 1-1], [2-1, 1-1]).
    Succeeds unifying X with 2.

Pairs = [1-2|Pairs], keysort(Pairs, Sorted).
    Undefined.
    [STO 7.3.3.. type_error(list, [1-2, 1-2, ...]) or
    loops in many implementations.]

keysort([V-V], V).
    Undefined.
    [STO 7.3.3, corresponds to the goal [V-V] = V.
    In many implementations this goal succeeds
    and violates the mode keysort(@list, -list).]
```

---

*Add built-in predicate term\_variables/2.*

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

### 8.5.5 term\_variables/2

### 8.5.5.1 Description

`term_variables(Term, Vars)` is true iff `Vars` unifies with the witness variable list of `Term` (7.1.1.5).

Procedurally, `term_variables(Term, Vars)` is executed as follows:

- a) Let `TVars` be the witness variable list of `Term` (7.1.1.5).
- b) If `Vars` unifies with `TVars`, then the goal succeeds.
- c) Else the goal fails.

NOTE — The order of variables in `Vars` ensures that, for every term `T`, the following goals are true:

```
term_variables(T, Vs1), term_variables(T, Vs2), Vs1 == Vs2.
```

```
term_variables(T, Vs1), term_variables(Vs1, Vs2), Vs1 == Vs2.
```

### 8.5.5.2 Template and modes

```
term_variables(@term, -list)
term_variables(?term, ?list)
```

### 8.5.5.3 Errors

- a) `Vars` is neither a partial list nor a list  
— `type_error(list, Vars)`.

### 8.5.5.4 Examples

```
term_variables(t, Vars).
Succeeds, unifying Vars with [].
```

```
term_variables(A+B*C/D, Vars).
Succeeds, unifying Vars with [A, B, C, D].
```

```
term_variables(t, [_ , _|a]).
type_error(list, [_ , _|a]).
```

```
S=B+T, T=A*B, term_variables(S, Vars).
Succeeds, unifying Vars with [B, A], T with A*B,
and S with B+A*B.
```

```
T=A*B, S=B+T, term_variables(S, Vars).
Same answer as above example.
```

```
term_variables(A+B+B, [B|Vars]).
Succeeds, unifying A with B and Vars with [B].
```

```
term_variables(X+Vars, Vars), Vars = [_ , _].
Undefined.
[STO 7.3.3, corresponds to the goal [X, Vars] = Vars.]
```

### 8.9.3.3 Errors (retract/1)

Replace in error condition *c*

— `permission_error(access, static_procedure, Pred)`.

by

— `permission_error(modify, static_procedure, Pred)`.

Add built-in predicate `retractall/1`.

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

## 8.9.5 retractall/1

### 8.9.5.1 Description

`retractall(Head)` is true.

Procedurally, `retractall(Head)` is executed as follows:

a) Searches sequentially through each dynamic user-defined procedure in the database and removes all clauses whose head unifies with `Head`, and the goal succeeds.

NOTES

1 The dynamic predicate remains known to the system as a dynamic predicate even when all of its clauses are removed.

2 Many existing processors define `retractall/1` as follows.

```
retractall(Head) :-
    retract((Head :- _)),
    fail.
retractall(_).
```

### 8.9.5.2 Template and modes

`retractall(@callable_term)`

### 8.9.5.3 Errors

a) `Head` is a variable

— `instantiation_error`.

b) `Head` is neither a variable nor a callable term

— `type_error(callable, Head)`.

c) The predicate indicator `Pred` of `Head` is that of a static procedure

— `permission_error(modify, static_procedure, Pred)`.

### 8.9.5.4 Examples

The examples defined in this subclause assume the database has been created from the following Prolog text:

```

:- dynamic(insect/1).
insect(ant).
insect(bee).

retractall(insect(bee)).
    Succeeds, retracting the clause 'insect(bee)'.

retractall(insect(_)).
    Succeeds, retracting all the clauses of predicate insect/1.

retractall(insect(spider)).
    Succeeds.

retractall(mammal(_)).
    Succeeds.

retractall(3).
    type_error(callable, 3).

retractall(retractall(_)).
    permission_error(modify, static_procedure, retractall/1).

```

### 8.11.5.3 Errors (open/4, open/3)

*Replace error condition f*

f) Stream is not a variable

— `type_error(variable, Stream)`.

*by*

f) Stream is not a variable

— `uninstantiation_error(Stream)`.

### 8.14.3.3 Errors (op/3)

*Replace*

l) `Op_specifier` is a specifier such that `Operator` would have an invalid set of specifiers (see 6.3.4.3).

— `permission_error(create, operator, Operator)`.

*by*

l) `Operator` is an atom, `Priority` is a priority, and `Op_specifier` is a specifier such that `Operator` would have an invalid set of priorities and specifiers (see 6.3.4.3).

— `permission_error(create, operator, Operator)`.

*Add additional error:*

m) `Operator` is a list, `Priority` is a priority, and `Op_specifier` is a specifier such that an element `Op` of the list `Operator` would have an invalid set of priorities and specifiers (see 6.3.4.3).

— `permission_error(create, operator, Op)`.

### 8.14.3.4

Add the following examples:

```
op(500, xfy, {}).
  permission_error(create, operator, {}).

op(500, xfy, [{}]).
  permission_error(create, operator, {}).

op(1000, xfy, '|').
  permission_error(create, operator, '|').

op(1000, xfy, ['|']).
  permission_error(create, operator, '|').

op(1150, fx, '|').
  permission_error(create, operator, '|').

op(1105, xfy, '|').
  Succeeds, making | a right associative
  infix operator with priority 1105.

op(0, xfy, '|').
  Succeeds, making | no longer an infix operator.
```

---

Add built-in predicate `call/2..8` and `false/0`.

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

### 8.15.4 `call/2..8`

These built-in predicates provide support for higher-order programming.

NOTE — A built-in predicate `apply/2` was implemented in some processors. Most uses can be directly replaced by `call/2..8`.

#### 8.15.4.1 Description

`call(Closure, Arg1, ...)` is true iff `call(Goal)` is true where `Goal` is constructed by appending `Arg1, ...` additional arguments to the arguments (if any) of `Closure`.

Procedurally, a goal of predicate `call/N` with  $N \geq 2$  is executed as follows:

- a) Let `call(p(X1, ..., XM), Y2, ..., YN)` be the goal to be executed,  $M \geq 0$ ,
- b) Execute `call(p(X1, ..., XM, Y2, ..., YN))` instead.

#### 8.15.4.2 Template and modes

`call(+callable_term, ?term, ...)`

#### 8.15.4.3 Errors

- a) `Closure` is a variable  
— `instantiation_error`.

b) Closure is neither a variable nor a callable term

— `type_error(callable, Closure)`.

c) The number of arguments in the resulting goal exceeds the implementation defined maximum arity (7.11.2.3)

— `representation_error(max_arity)`.

d) `call/N` is called with  $N \geq 9$  and it shall be implementation dependent whether this error condition is satisfied

— `existence_error(procedure, call/N)`.

e) Goal cannot be converted to a goal

— `type_error(callable, Goal)`.

NOTE — A standard-conforming processor may implement `call/N` in one of the following ways because error condition d is implementation dependent (3.91).

1) Implement only the seven built-in predicates `call/2` up to `call/8`.

2) Implement `call/2..N` up to any  $N$  that is within  $8..max\_arity$  (7.11.2.3). Produce existence errors for larger arities below `max_arity`.

3) Implement `call/9` and above only for certain execution modes.

#### 8.15.4.4 Examples

```
call(integer, 3).
Succeeds.
```

```
call(functor(F,c), 0).
Succeeds, unifying F with c.
```

```
call(call(call(atom_concat, pro), log), Atom).
Succeeds, unifying Atom with prolog.
```

```
call(;; X = 1, Y = 2).
Succeeds, unifying X with 1. On re-execution,
succeeds, unifying Y with 2.
```

```
call(;; (true->fail), X=1).
Fails.
```

The following examples assume that `maplist/2` is defined with the following clauses:

```
maplist(_Cont, []).
maplist(Cont, [E|Es]) :-
    call(Cont, E),
    maplist(Cont, Es).
```

```
maplist(>(3), [1, 2]).
Succeeds.
```

```
maplist(>(3), [1, 2, 3]).
Fails.
```

```
maplist(=(X), Xs).
```

Succeeds,  
unifying Xs with [].  
On re-execution, succeeds,  
unifying Xs with [X].  
On re-execution, succeeds,  
unifying Xs with [X, X].  
On re-execution, succeeds,  
unifying Xs with [X, X, X].  
Ad infinitum.

## 8.15.5 false/0

### 8.15.5.1 Description

false is false.

### 8.15.5.2 Template and modes

false

### 8.15.5.3 Errors

None.

### 8.15.5.4 Examples

false.  
Fails.

---

*Correct error conditions of atom\_chars/2, atom\_codes/2, number\_chars/2, number\_codes/2.*

## 8.16.4.3 Errors (atom\_chars/2)

*Replace error condition a, c, and d. Add error condition e.*

a) Atom is a variable and List is a partial list.

— instantiation\_error.

c) List is neither a partial list nor a list

— type\_error(list, List).

d) Atom is a variable and an element of a list prefix of List is a variable.

— instantiation\_error.

e) An element E of a list prefix of List is neither a variable nor a one-char atom

— type\_error(character, E).

## 8.16.5.3 Errors (atom\_codes/2)

*Replace error condition a, c, and d. Add error condition e and f.*

- a) Atom is a variable and List is a partial list.  
— instantiation\_error.
- c) List is neither a partial list nor a list  
— type\_error(list, List).
- d) Atom is a variable and an element of a list prefix of List is a variable.  
— instantiation\_error.
- e) An element E of a list prefix of List is neither a variable nor an integer  
— type\_error(integer, E).
- f) An element of a list prefix of List is neither a variable nor a character code  
— representation\_error(character\_code).

### 8.16.7.3 Errors (number\_chars/2)

Replace error condition a, c, and d. Add error condition f.

- a) Number is a variable and List is a partial list.  
— instantiation\_error.
- c) List is neither a partial list nor a list  
— type\_error(list, List).
- d) Number is a variable and an element of a list prefix of List is a variable.  
— instantiation\_error.
- f) An element E of a list prefix of List is neither a variable nor a one-char atom  
— type\_error(character, E).

### 8.16.8.3 Errors (number\_codes/2)

Replace error conditions a, c, and d. Add error condition f and g.

- a) Number is a variable and List is a partial list.  
— instantiation\_error.
- c) List is neither a partial list nor a list  
— type\_error(list, List).
- d) Number is a variable and an element of a list prefix of List is a variable.  
— instantiation\_error.
- f) An element E of a list prefix of List is neither a variable nor an integer  
— type\_error(integer, E).
- g) An element of a list prefix of List is neither a variable nor a character code  
— representation\_error(character\_code).

---

Add evaluable functors (+)/1 (unary plus) and (div)/2 (flooring integer division) to simple arithmetic functors (9.1). Add operators corresponding to (-)/1 and (//)/2 (integer division).

## 9.1.1

Add to table:

Evaluable functor Operation

(div)/2            *intfloordiv<sub>I</sub>*

(+)/1              *pos<sub>I</sub>, pos<sub>F</sub>*

Add '*div*' to enumeration in Note. Add to Note:

'+', '-' are prefix predefined operators.

### 9.1.3

Add specifications:

*intfloordiv<sub>I</sub>* :  $I \times I \rightarrow I \cup \{\text{int\_overflow, zero\_divisor}\}$

*pos<sub>I</sub>* :  $I \rightarrow I$

Add as axioms:

*intfloordiv<sub>I</sub>*(*x*,*y*) =  $\lfloor x/y \rfloor$

if  $y \neq 0 \wedge \lfloor x/y \rfloor \in I$

= **int\_overflow**

if  $y \neq 0 \wedge \lfloor x/y \rfloor \notin I$

= **zero\_divisor**

if  $y = 0$

*pos<sub>I</sub>* (*x*) = *x*

### 9.1.4

Add specification:

*pos<sub>F</sub>* :  $F \rightarrow F$

Add as axiom:

*pos<sub>F</sub>* (*x*) = *x*

---

Add evaluable functors *max*/2, *min*/2, (^)/2, *asin*/1, *acos*/1, *atan2*/2, *tan*/1. Add evaluable atom *pi*/0.

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

## 9.3.8 max/2 – maximum

### 9.3.8.1 Description

*max*(*X*, *Y*) evaluates the expressions *X* and *Y* with values *VX* and *VY* and has the value of the maximum of *VX* and *VY*. If *VX* and *VY* have the same type then the value *R* satisfies  $R \in \{VX, VY\}$ .

If *VX* and *VY* have different types then let *VI* and *VF* be the values of type integer and float. The

value  $R$  shall satisfy

$R \in \{VI, \text{float}(VI), VF, \text{undefined}\}$

and the value shall be implementation dependent.

NOTE — The possible values of  $\text{float}(VI)$  include the exceptional value  $\text{float\_overflow} \notin F$  (9.1.6).

### 9.3.8.2 Template and modes

```
max(float-exp, float-exp) = float
max(float-exp, int-exp) = number
max(int-exp, float-exp) = number
max(int-exp, int-exp) = integer
```

### 9.3.8.3 Errors

a)  $X$  is a variable

— `instantiation_error`.

b)  $Y$  is a variable

— `instantiation_error`.

c)  $VX$  and  $VY$  have different type and it shall be implementation dependent whether this error condition is satisfied

— `evaluation_error(undefined)`.

d)  $VX$  and  $VY$  have different type and one of them is an integer  $VI$  with

$\text{float}_{I \rightarrow F}(VI) = \text{float\_overflow}$  (9.1.6) and it shall be implementation dependent whether this error condition is satisfied

— `evaluation_error(float_overflow)`.

### 9.3.8.4 Examples

```
max(2, 3).
```

Evaluates to 3.

```
max(2.0, 3).
```

Evaluates to 3, 3.0, or `evaluation_error(undefined)`.

[The result is implementation dependent.]

```
max(2, 3.0).
```

Evaluates to 3.0 or `evaluation_error(undefined)`.

[The result is implementation dependent.]

```
max(0, 0.0).
```

Evaluates to 0, 0.0, or `evaluation_error(undefined)`.

[The result is implementation dependent.]

## 9.3.9 min/2 – minimum

### 9.3.9.1 Description

`min(X, Y)` evaluates the expressions `X` and `Y` with values `VX` and `VY` and has the value of the minimum of `VX` and `VY`. If `VX` and `VY` have the same type then the value `R` satisfies  $R \in \{VX, VY\}$ .

If `VX` and `VY` have different types then let `VI` and `VF` be the values of type integer and float. The value `R` shall satisfy

$R \in \{VI, \text{float}(VI), VF, \text{undefined}\}$

and the value shall be implementation dependent.

NOTE — The possible values of `float(VI)` include the exceptional value `float_overflow`  $\notin F$  (9.1.6).

### 9.3.9.2 Template and modes

```
min(float-exp, float-exp) = float
min(float-exp, int-exp) = number
min(int-exp, float-exp) = number
min(int-exp, int-exp) = integer
```

### 9.3.9.3 Errors

a) `X` is a variable

— `instantiation_error`.

b) `Y` is a variable

— `instantiation_error`.

c) `VX` and `VY` have different type and it shall be implementation dependent whether this error condition is satisfied

— `evaluation_error(undefined)`.

d) `VX` and `VY` have different type and one of them is an integer `VI` with

$\text{float}_{I \rightarrow F}(VI) = \text{float\_overflow}$  (9.1.6) and it shall be implementation dependent whether this error condition is satisfied

— `evaluation_error(float_overflow)`.

### 9.3.9.4 Examples

```
min(2, 3).
```

Evaluates to 2.

```
min(2, 3.0).
```

Evaluates to 2, 2.0, or `evaluation_error(undefined)`.

[The result is implementation dependent.]

```
min(2.0, 3).
```

Evaluates to 2.0 or `evaluation_error(undefined)`.

[The result is implementation dependent.]

```
min(0, 0.0).
```

Evaluates to 0, 0.0, or `evaluation_error(undefined)`.

[The result is implementation dependent.]

### 9.3.10 (^)/2 – integer power