ISO/IEC DTR 13211–1:2006
New built-in flags, predicates, and functions proposal

Editor: Paulo Moura
pmoura@di.ubi.pt

July 18, 2008

Introduction

This proposal specifies a set of built-in predicates and flags to be added to Part 1 of the International Standard for Prolog, ISO/IEC 13211. When evaluating this proposal, please comment each predicate individually by presenting your arguments for either accepting or rejecting its inclusion in the next revision of the Part 1 standard.

This proposal is written as an extension to the ISO/IEC 13211–1 Prolog standard, adopting a similar structure. Specifically, this proposal either adds new sections and clauses to, or modifies the reading of existing clauses on ISO/IEC 13211–1.

This draft proposal may contain in several places informative text, type-set in *italics*. Such informative text is used for editorial comments deemed useful during the development of this draft and may not be included in the final version.

Contributors

This list includes so far the people present at the ISO meeting collocated with the ICLP'06 and people participating on the mailing list discussions.

- Bart Demoen (Belgium)
- Jan Wielemaker, (Netherlands)
- Joachim Schimpf (UK)
- Jonathan Hodgson (USA)
- Katsuhiko Nakamura (Japan)
- Klaus Daessler (Germany)
- Mary Kroening (USA)
- Michael Covington (USA)

- Neng-Fu Zhou (USA)
- Paulo Moura (Portugal)
- Pierre Deransart (France)
- Péter Szabó (Hungary)
- Péter Szeredi (Hungary)
- Rémy Haemmerlé (France)
- Richard O'Keefe (NZ)
- Roberto Bagnara (Italy)
- Roger Scowen (UK)
- Ulrich Neumerkel (Austria)

1 Scope

This proposal is designed to promote the applicability and portability of Prolog by adding to ISO/IEC 13211–1:1995 a set of built-in predicates and flags that are either common practice and implemented in most Prolog systems or are needed to clarify implementation-dependent behavior. As such, this proposal includes specifications for:

a) A set of flags allowing a programmer to query a system about the floating-point arithmetic implementation and to declare the default encoding for Prolog text

b) Commonly used term testing predicates already available in most Prolog systems that are missing from ISO/IEC 13211–1:1995

c) Commonly used meta-predicates which should be available as built-in predicates in order to provide adequate performance

d) A set of built-in predicates for list processing, providing functionality similar to the atomic processing built-in predicates present on ISO/IEC 13211–1:1995

e) Commonly used evaluable functors that are missing from ISO/IEC 13211–1:1995

NOTE — This part of ISO/IEC 13211 will eventually merge with ISO/IEC 13211–1:1995 resulting in a new version of the Part 1 standard.

6 Syntax

6.3.4.4 The operator table

The bitwise exclusive or operator is added to the operator table with the same specification as the bitwise or and and bitwise and operators:

Priority	Specifier	Operators(s)
500	yfx	`+ - /\ \/ ><`

The unary plus operator is added to the operator table with the same specification as the unary minus or the bitwise complement operators:

Priority	Specifier	Operators(s)
200	fy	`+ - \`

7 Language concepts and semantics

7.1 Types

7.1.6 Related terms

7.1.6.9 Pair

`P` is a pair if it is a compound term `'-'(Key, Value)` where `Key` and `Value` are terms.

NOTE — In Prolog text and this part of ISO/IEC 13211 a pair `'-'(Key, Value)` is normally written as `Key-Value` or `(Key)-(Value)` depending on whether or not `Key` and `Value` are operators.

7.4 Prolog text

7.4.2 Directives

7.4.2.10 `encoding/1`

A directive `encoding(Encoding)` specifies that the Prolog text being prepared for execution uses the declared encoding. When used, this directive shall be the first term, on the first line, in a Prolog text with no extra layout characters (6.5.4) before the `:-/1` directive operator. Moreover, a single layout character shall be used between the directive operator and the directive functor. No layout characters or comments shall appear between the directive opening and closing parenthesis.

7.10 Input/ouput

7.10.2 Streams

7.10.2.11 Options on stream creation

An implementation may optionally support the following stream-options:

`bom(Bool)` — If `Bool` (7.1.4.2) is `true` then a Unicode encoding *Byte Order Mark* shall be written when opening a text stream for writing in mode `write` or is probed for when opening the text stream for reading. This option shall be ignored when opening a stream in mode `append`. If `Bool` is `false` then a Unicode encoding *Byte Order Mark* shall not be written when opening the text stream for writing.

When no `bom(Bool)` stream-option is specified, the default value shall be `true` when the text stream is opened for reading and `false` when the text stream is opened for writing.

`encoding(Encoding)` — `Encoding` is an atom representing the text encoding that shall be used when opening the stream for writing or the text encoding of the stream opened for reading.

When opening the text stream for reading with the default `bom(Bool)` stream-option value or by explicitly specifying the `bom(true)` stream-option, if a *Byte Order Mark* is detected, it will be used to set the corresponding Unicode text stream encoding, overriding any `encoding(Encoding)` that might be also specified.

NOTES

1 These stream-options imply the stream-option `type(text)`.

2 The set of supported text encodings is implementation-defined.

7.10.2.13 Stream properties

An implementation may optionally support the following stream properties:

`bom(Bool)` — If present and if `Bool` (7.1.4.2) is `true`, a Unicode encoding *Byte Order Mark* was detected while opening the text stream for reading or a *Byte Order Mark* was written while opening the text stream for writing.

`encoding(Encoding)` — Encoding used for the text stream.

NOTE — These stream properties imply the stream property `type(text)`.

7.11 Flags

7.11.2 Flags defining float type *F*

7.11.2.1 Flag: float_mantissa_digits

Possible values: the default value only

Default: implementation defined

Changeable: No

Description: The value of this flag is the number of significant digits on the mantissa of a normalized floating point number (in base 10), an implementation defined integer value.

7.11.2.2 Flag: float_epsilon

Possible values: the default value only

Default: implementation defined

Changeable: No

Description: The value of this flag is the distance from `1.0` to the next largest floating point number, an implementation defined value. Thus, it allows the programmer to query an implementation about the relative accuracy when performing arithmetic with floating point numbers.

7.11.2.3 Flag: float_min_exponent

Possible values: the default value only

Default: implementation defined

Changeable: No

Description: The value of this flag is smallest possible value of the exponent of a normalized floating point number, an implementation defined integer value.

7.11.2.4 Flag: float_max_exponent

Possible values: the default value only

Default: implementation defined

Changeable: No

Description: The value of this flag is greatest value of the exponent of a normalized floating point number, an implementation defined integer value.

7.11.3 Other flags

7.11.3.6 Flag: unification_subject_to_occurs_check

Possible values: `fail`, `cyclic`, `unsafe`

Default: implementation defined

Changeable: No

Description: This read-only flag describes the behaviour of the Prolog system when a variable is unified with a compound term that contains it (STO unification, 3.165). The flag value `fail` implies that STO unification simply fails. The flag value `unsafe` means that if an STO unification is encountered the further behavior of the system is undefined. The flag value `cyclic` implies that STO unifications will be successful and result in the creation of cyclic terms. Moreover, this flag indicates that the Prolog system is capable of handling certain operations on cyclic terms *safely*, namely unifying, comparing, and copying of cyclic terms is assured to terminate.

NOTES

1 The flag value `fail` means that the built-in predicate `=/2` (8.2.1) behaves exactly as the built-in predicate `unify_with_occurs_check/2` (8.2.2). The flag value `unsafe` may imply that the STO unification itself, or further unifications or built-in predicate calls may not terminate, or cause the system to fail or raise an exception.

2 A conforming Prolog processor which supports the creation of cyclic terms shall be accompanied by documentation that specifies which built-in predicates may be safely called with arguments which are cyclic terms.

Examples:

```
| ?- X = f(X).
```

- `fail`: fails
- `cyclic`: succeeds and unifies `X` with a cyclic term `f(f(f(...)))`.
- `unsafe`: undefined. Often succeeds, but subsequent use of `X`, as e.g. in `X=X`, causes an error.

```
| ?- X = f(X), Y = f(Y), X = Y.
| ?- g(X,Y,X) = g(f(X),f(Y),Y).
| ?- X = f(X), Y = f(Y), X == Y.
| ?- X = f(X), asserta(p(X)).
```

For all the above four examples:

- `fail`: fails
- `cyclic`: succeeds and unifies both `X` and `Y` with a cyclic term `f(f(f(...)))`.
- `unsafe`: undefined. Often causes an error.

7.11.3.7 Flag: encoding

Possible values: implementation defined (but always an atom)

Default: implementation defined

Changeable: implementation defined

Description: This flag represents the default encoding for text streams. An implementation shall document if the flag value can be changed by programmer as well all the supported encodings.

7.12 Errors

7.12.2 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 `pair` (see 7.12.2 b of ISO/IEC 13211-1).

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

7.13 Predicate properties

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

- `static` — The predicate is static
- `dynamic` — The predicate is dynamic
- `built_in` — The predicate is a built-in predicate
- `multifile` — The predicate is the subject of a multifile directive

A processor may support one or more additional predicate properties as an implementation specific feature. Implementation-defined properties are not required to be atomic terms.

8 Built-in predicates

The following sections extends, with the specified number, the corresponding ISO/IEC 13211–1 sections:

8.2 Term unification

8.2.4 subsumes/2

8.2.4.1 Description

`subsumes(General, Specific)` is true iff there is a substitution θ, including the empty substitution, such that the term General is instantiated to $General\theta = Specific$. This predicate provides a one-way unification.

8.2.4.2 Template and modes

`subsumes(?term, @term)`

8.2.4.3 Errors

None.

8.2.4.4 Examples

```
   subsumes(f(X,Y), f(Z,Z)).
      Succeeds, unifying both X and Y to Z.

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

8.3 Type testing

8.3.9 callable/1

8.3.9.1 Description

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

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.
```

8.3.10 ground/1

8.3.10.1 Description

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

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/1

8.3.11.1 Description

`acyclic(Term)` is true iff `Term` is an acyclic term. For implementations not supporting STO unification 7.11.3.6, calls to this predicate simply succeed.

8.3.11.2 Template and modes

`acyclic(@term)`

8.3.11.3 Errors

None.

8.3.11.4 Examples

```
acyclic(a(1, _)).
   Succeeds.

X = f(X), acyclic(X).
   Fails.
```

8.3.12 cyclic/1

8.3.12.1 Description

`cyclic(Term)` is true iff `Term` is a cyclic term. For implementations not supporting STO unification 7.11.3.6, calls to this predicate simply fail.

8.3.12.2 Template and modes

`cyclic(@term)`

8.3.12.3 Errors

None.

8.3.12.4 Examples

```
cyclic(a(1, _)).
   Fails.

X = f(X), cyclic(X).
   Succeeds.
```

8.4 Term comparison

8.4.2 compare/3

8.4.2.1 Description

`compare(Order, Term1, Term2)` is true iff `Order` corresponds to the standard order between `Term1` and `Term2`. The argument `Order` is unified with the atom `<` when `Term1` is less than `Term2`, with the atom `=` when `Term1` and `Term2` are equal, and with the atom `>` when `Term1` is greater than `Term2`.

8.4.2.2 Template and modes

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

8.4.2.3 Errors

a) `Order` is neither a variable nor an atom
— `type_error(atom, Order)`

b) `Order` an atom other than `<`, `=`, 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, 3, 3.0).
   Succeeds, unifying Order with >.
```

8.5 Term creation and decomposition

8.5.5 numbervars/3

8.5.5.1 Description

`numbervars(Term, Start, End)` is true. This predicate unifies each free variable on `Term` with a compound term with the format `'$VAR'(N)` where `N` is an integer starting from `Start` and ending at `End`−`1`.

8.5.5.2 Template and modes

`numbervars(?nonvar, +integer, -integer)`

8.5.5.3 Errors

a) `Start` is a variable
— `instantiation_error`

b) `Start` is neither a variable nor an integer
— `type_error(integer, Start)`

8.5.5.4 Examples

```
numbervars(foo(A, B, A), 0, End).
   Succeeds, unifying A with '$VAR'(0), B with '$VAR'(1),
   and End with 2.
```

8.8 Clause retrieval and information

8.8.3 predicate_property/2

8.8.3.1 Description

`predicate_property(Head, Property)` is true iff the procedure associated with the argument `Head` (3.84) has predicate property `Property`.

Procedurally, `predicate_property(Head, Property)` is executed as follows

a) Determines the principal functor `P` and arity `N` associated with `Head`. `P/N` is the associated predicate indicator

b) Searches the complete database and creates a set *SetPP* of all terms `PP` such that `P/N` identifies a procedure which has predicate property `PP` and `PP` is unifiable with `Property`

c) If *SetPP* is non empty set proceeds to 8.8.3.1 e,

d) Else the goal fails

e) Chooses the first element `PPP` of *SetPP*, unifies `PPP` with `Property` and the predicate succeeds

f) If all elements of *SetPP* have been chosen the predicate fails

g) Else chooses the first element `PPP` of *SetPP* that has not already been chosen, unifies `PPP` with `Property` and the goal succeeds

`predicate_property(Head, Property)` is re-executable. On backtracking, continue at 8.8.3.1 f.

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

NOTES

1 A processor may support, as an implementation specific feature, additional predicate properties.

2 For a dynamic predicate, all proprieties related to its definition shall be removed when the predicate is abolished.

8.8.3.2 Template and modes

`predicate_property(@callable_term, ?predicate_property)`

8.8.3.3 Errors

a) `Head` is a variable
— `instantiation_error`

b) `Head` is neither a variable nor a callable term
— `type_error(callable, Head)`

c) `Property` is neither a variable nor a predicate property
— `domain_error(predicate_property, Property)`

8.8.3.4 Examples

```
predicate_property(once(_), built_in).
   Succeeds.

predicate_property(atom_codes(_, _), Property).
   Succeeds unifying Property with static.
   On re-execution, succeeds unifying Property with built_in.
```

8.9 Clause creation and destruction

8.9.3 retract/1

8.9.3.1 Errors

There is a typo on the current standard in the specification of the `permission_error` *exception that should use the atom* `modify` *instead of* `access` *in order to match the specification of other database predicates.*

a) ...

b) ...

c) The predicate indicator `Pred` of `Head` is that of a static procedure
— `permission_error(modify, static_procedure, Pred)`

8.9.5 retractall/1

8.9.5.1 Description

`retractall(Head)` is true.

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

a) Determines the principal functor `P` and arity `N` associated with `Head`. `P/N` is the associated predicate indicator

b) If the database contains a dynamic procedure whose predicate indicator is `P/N`, then proceeds to 8.9.5.1 d,

c) Else the goal succeeds.

d) Retracts from the database all clauses whose head unifies with `Head` and the goal succeeds

NOTES

1 The dynamic predicate shall continue to be known by the system even when all of its clauses are removed.

2 This predicate does not change any of the standard predicate properties of the referenced predicate (as reported by `predicate_property(Head, Property)`), even when all of its clauses are removed.

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, Generate)`

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).
insect(spider).

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

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

```
retractall(insect(elephant)).
   Succeeds.

retractall(mammal(_)).
   Succeeds.

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

8.10 All solutions

8.10.4 forall/2

8.10.4.1 Description

`forall(Generate, Test)` is true iff for all possible bindings of `Generate`, the goal `Test` is true. Procedurally, abstracting error checking, the predicate shall behave as being defined by `\+ (call(Generator), \+ call(Test))`.

8.10.4.2 Template and modes

`forall(@callable_term, @callable_term)`

8.10.4.3 Errors

a) `Generate` is a variable
— `instantiation_error`

b) `Generate` is neither a variable nor a callable term
— `type_error(callable, Generate)`

c) `Test` is a variable
— `instantiation_error`

d) `Test` is neither a variable nor a callable term
— `type_error(callable, Test)`

8.10.4.4 Examples

The following examples assume that the predicate `a/1` and `b/1` are defined with the following clauses:

```
a(1). a(2). a(3).
b(1, a). b(2, b). b(3, c).

forall(fail, true).
   Succeeds.
```

```
forall(a(X), b(X, _)).
   Succeeds.

forall(a(X), b(_, X)).
   Fails.

forall(b(_, Y), write(Y))
   Succeeds, outputting the characters
abc
   to the current output stream.
```

8.15 Logic and control

8.15.4 call/2-N

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`.

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 for compound terms
— `representation_error(max_arity)`

8.15.4.4 Examples

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

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

8.15.5 call_cleanup/2

8.15.5.1 Description

`call_cleanup(Goal, Cleanup)` is true iff `call(Goal)` is true. When the execution of `Goal` terminates, either by deterministic success, by failure, by its choice-points being cut, or by raising an exception, the goal `Cleanup` is executed. The success or failure of `Cleanup` is ignored, as are any choice-points created while proving it. An exception thrown by `call(Goal)` may be caught by `Cleanup`. An exception thrown by `Cleanup` is handled as normal.

8.15.5.2 Template and modes

`call_cleanup(+callable_term, @callable_term)`

8.15.5.3 Errors

a) `Goal` is a variable
— `instantiation_error`

b) `Goal` is neither a variable nor a callable term
— `type_error(callable, Goal)`

c) `Cleanup` is a variable
— `instantiation_error`

d) `Cleanup` is neither a variable nor a callable term
— `type_error(callable, Cleanup)`

8.15.5.4 Examples

```
call_cleanup(true, write(terminated)).
   Succeeds, outputting the atom
terminated
   to the current output stream.

catch(call_cleanup(throw(e), catch(true, E, throw(E))), F, true).
   Succeeds, unifying F with e.
```

8.18 List processing

8.18.1 append/3

8.18.1.1 Description

`append(List1, List2, List3)` is true iff `List3` is a list resulting from the concatenation of `List1` and `List2`.

8.18.1.2 Template and modes

`append(?list, ?list, ?list)`

8.18.1.3 Errors

None.

8.18.1.4 Examples

Some example calls of `append/3`:

```
append([], List, List).
   Succeeds.
```

8.18.2 length/2

8.18.2.1 Description

`length(List, Length)` is true iff `Length` is the length of the list `List`.

8.18.2.2 Template and modes

`length(?list, ?integer)`

8.18.2.3 Errors

a) `Length` is neither a variable nor an integer
— `type_error(integer, Length)`

8.18.2.4 Examples

```
length([1, 2, 3], Length).
   Succeeds, unifying Length with 3.

length(List, 3).
   Succeeds, unifying List with [_, _, _]

length(List, -2).
   Fails

length(List, Length).

List = []
Length = 0 ;
```

```
List = [_]
Length = 1 ;

List = [_, _]
Length = 2
yes
```

8.18.3 member/2

8.18.3.1 Description

`member(Element, List)` is true iff `Element` is a member of list `List`.

8.18.3.2 Template and modes

`member(?term, ?list)`

8.18.3.3 Errors

None.

8.18.3.4 Examples

```
member(2, [1, 2, 3]).
   Succeeds.
```

8.18.4 sort/2

8.18.4.1 Description

`sort(List, Sorted)` is true iff `Sorted` is a list containing the non-duplicated elements of `List` sorted in ascending order following standard order (7.2).

8.18.4.2 Template and modes

`sort(@list, ?list)`

8.18.4.3 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)`

8.18.4.4 Examples

```
sort([1, 2, 1, 8, 4], Sorted).
   Succeeds, unifying Sorted with [1, 2, 4, 8].
```

8.18.5 keysort/2

8.18.5.1 Description

`keysort(List, Sorted)` is true iff `List` is a list of elements with the format `Key-Value` and `Sorted` is a list containing the elements of `List` sorted according to the value of `Key` in ascending order following standard order (7.2). The relative order of elements of `List` with the same key shall not change in the `Sorted` list.

8.18.5.2 Template and modes

`keysort(@list, ?list)`

8.18.5.3 Errors

a) `List` is a partial list
— `instantiation_error`

b) `List` is neither a partial list nor a list
— `type_error(list, List)`

c) An element `Element` of `List` is a variable
— `instantiation_error`

d) An element `Element` of `List` is neither a variable nor a `'-'/2` compound term
— `type_error(pair, Element)`

e) `Sorted` is neither a partial list nor a list
— `type_error(list, Sorted)`

8.18.5.4 Examples

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

9 Evaluable functors

9.1 The simple arithmetic functors

The unary plus evaluable arithmetic functor is added.

9.1.1 Evaluable functors and operations

Evaluable functor	Operation
`(+)/1`	pos_I, pos_F

9.1.3 Integer operations and axioms

The following operations are specified:

$pos_I : I \to I$

For all $x \in I$, the following axioms shall apply:

$pos_I(x) = x$

9.1.4 Floating point operations and axioms

The following operations are specified:

$pos_F : F \to F$

For all $x \in F$, the following axioms shall apply:

$pos_F(x) = x$

9.3 Other arithmetic functors

9.3.8 log/2

9.3.8.1 Description

`log(B, X)` evaluates the expression `B` with value `VB`, the expression `X` with value `VX`, and has the value of the logarithm to base `VB` of `VX`.

9.3.8.2 Template and modes

```
log(int-exp, float-exp) = float
log(int-exp, int-exp) = float
```

9.3.8.3 Errors

a) `B` is a variable
— `instantiation_error`

b) `B` is not a variable and `VB` is not an integer
— `type_error(integer, VB)`

c) `VB` is zero or negative
— `evaluation_error(undefined)`

d) `X` is a variable
— `instantiation_error`

e) `X` is not a variable and `VX` is not a number
— `type_error(number, VX)`

f) `VX` is zero or negative
— `evaluation_error(undefined)`

9.3.8.4 Examples

```
log(10, 10.0).
   Evaluates to 1.0.
```

9.3.9 gcd/2

9.3.9.1 Description

`gcd(I, J)` evaluates the expression `I` with value `VI`, the expression `J` with value `VJ`, and has the value of the greatest common divisor of `VI` of `VJ`.

9.3.9.2 Template and modes

`gcd(int-exp, int-exp) = integer`

9.3.9.3 Errors

a) `I` is a variable
— `instantiation_error`

b) `I` is not a variable and `VI` is not an integer
— `type_error(integer, VI)`

c) `J` is a variable
— `instantiation_error`

d) `J` is not a variable and `VJ` is not an integer
— `type_error(integer, VJ)`

9.3.9.4 Examples

```
gcd(2, 3).
   Evaluates to 1.
```

9.3.10 min/2

9.3.10.1 Description

`min(X, Y)` evaluates the expression `X` with value `VX`, the expression `Y` with value `VY`, and has the value of the minimum of `VX` and `VY`. When used with expressions of mixed-types, the result is implementation-dependent; an implementation may chose either to return a value or to throw an exception.

9.3.10.2 Template and modes

```
min(float-exp, float-exp) = float
min(float-exp, int-exp) = implementation-dependent result
min(int-exp, float-exp) = implementation-dependent result
min(int-exp, int-exp) = integer
```

9.3.10.3 Errors

a) `X` is a variable or `Y` is a variable
   — `instantiation_error`

b) `X` is not a variable and `VX` is not a number
   — `type_error(number, VX)`

c) `Y` is not a variable and `VY` is not a number
   — `type_error(number, VY)`

9.3.10.4 Examples

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

min(2.0, 3.0)
   Evaluates to 2.0.

min(0, 0.0).
   Implementation-dependent result.
```

9.3.11 max/2

9.3.11.1 Description

`max(X, Y)` evaluates the expression `X` with value `VX`, the expression `Y` with value `VY`, and has the value of the maximum of `VX` and `VY`. When used with expressions of mixed-types, the result is implementation-dependent; an implementation may chose either to return a value or to throw an exception.

9.3.11.2 Template and modes

```
max(float-exp, float-exp) = float
max(float-exp, int-exp) = implementation-dependent result
max(int-exp, float-exp) = implementation-dependent result
max(int-exp, int-exp) = integer
```

9.3.11.3 Errors

a) `X` is a variable or `Y` is a variable
— `instantiation_error`

b) `X` is not a variable and `VX` is not a number
— `type_error(number, VX)`

c) `Y` is not a variable and `VY` is not a number
— `type_error(number, VY)`

9.3.11.4 Examples

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

max(2.0, 3.0)
   Evaluates to 3.0.

max(0, 0.0).
   Implementation-dependent result.
```

9.4 Bitwise functors

9.4.6 (><)/2 – bitwise exclusive or

9.4.6.1 Description

`'><'(B1, B2)` evaluates the expressions `B1` and `B2` with values `VB1` and `VB2` and has the value such that each bit is set iff only one of the corresponding bits in `VB1` and `VB2` is set.

The value shall be implementation defined if `VB1` and `VB2` is negative.

9.4.6.2 Template and modes

```
'><'(int-exp, int-exp) = integer
```

NOTE — '><' is an infix predefined operator (see 6.3.4.4).

9.4.6.3 Errors

a) `B1` is a variable
— `instantiation_error`

b) `B2` is a variable
— `instantiation_error`

c) `B1` is not a variable and `VB1` is not an integer
— `type_error(integer, VB1)`

d) `B2` is not a variable and `VB2` is not an integer
— `type_error(integer, VB2)`

9.4.6.4 Examples

```
'><'(10, 12).
   Evaluates to the value 6.

'><'(125, 255).
   Evaluates to to the value 130.

'><'(-10, 12).
   Evaluates to an implementation defined value.

'><'(77, N)
   instantiation_error.

'><'(foo, 2)
   type_error(integer, foo).
```

9.5 Trigonometric functors

Assuming that we will be adding the trigonometric functions described below to the revised core standard, it's probably best if we gather all the trigonometric functions under their own section. the following order is assumed below: sin/1, cos/1, tan/1, asin/1, acos/1, atan/1, and atan/2.

9.5.3 tan/1

9.5.3.1 Description

`tan(X)` evaluates the expression `X` with value `VX` and has the value of the tangent of `VX` (measured in radians).

9.5.3.2 Template and modes

```
tan(float-exp) = float
tan(int-exp) = float
```

9.5.3.3 Errors

a) `X` is a variable
   — `instantiation_error`

b) `X` is not a variable and `VX` is not a number
   — `type_error(number, VX)`

9.5.3.4 Examples

```
tan(pi).
   Evaluates to 0.0.
```

9.5.4 asin/1

9.5.4.1 Description

`asin(X)` evaluates the expression `X` with value `VX` and has the value of the arc sine of `VX` (in radians).

9.5.4.2 Template and modes

```
asin(float-exp) = float
asin(int-exp) = float
```

9.5.4.3 Errors

a) `X` is a variable
   — `instantiation_error`

b) `X` is not a variable and `VX` is not a number
   — `type_error(number, VX)`

9.5.4.4 Examples

```
asin(1.0).
   Evaluates to a value approximately equal to 1.570796326795.
```

9.5.5 acos/1

9.5.5.1 Description

`acos(X)` evaluates the expression `X` with value `VX` and has the value of the arc cosine of `VX` (in radians).

9.5.5.2 Template and modes

```
acos(float-exp) = float
acos(int-exp) = float
```

9.5.5.3 Errors

a) `X` is a variable
— `instantiation_error`

b) `X` is not a variable and `VX` is not a number
— `type_error(number, VX)`

9.5.5.4 Examples

```
acos(0.0).
   Evaluates to a value approximately equal to 1.570796326795.
```

9.5.6 atan/2

9.5.6.1 Description

`atan(Y, X)` evaluates the expression `Y` with value `VY`, the expression `X` with value `VX`, and computes the principal value of the arc tangent of `VY/VX` (in radians), using the signs of both arguments to determine the quadrant of the return value. When both arguments are `0.0`, an implementation-dependent value is returned.

9.5.6.2 Template and modes

```
atan(float-exp, float-exp) = float
atan(float-exp, int-exp) = float
atan(int-exp, float-exp) = float
atan(int-exp, int-exp) = float
```

9.5.6.3 Errors

a) `X` is a variable
— `instantiation_error`

b) `Y` is not a variable and `VY` is not a number
— `type_error(number, VY)`

c) `X` is not a variable and `VX` is not a number
— `type_error(number, VX)`

9.5.6.4 Examples

```
atan(0.0, -0.0).
   Evaluates to a value approximately equal to 3.14159265358979.
```

9.6 Hyperbolic trigonometric functors

Should we also add the sinh/1, cosh/1, tanh/1, asinh/1, acosh/1, and atanh/1 hyperbolic trigonometric functors?

9.7 Mathematical constants

9.7.1 pi/0

9.7.1.1 Description

`pi` evaluates to the floating-point number which best approximates the mathematical constant π, the ratio of a circle's circumference to its diameter.

9.7.1.2 Examples

```
pi.
   Evaluates to the corresponding mathematical constant.
   The accuracy of the result is implementation-dependent.
```

9.7.2 e/0

9.7.2.1 Description

`e` evaluates to the floating-point number which best approximates the mathematical constant e, the base of natural logarithms.

9.7.2.2 Examples

```
e.
   Evaluates to the corresponding mathematical constant.
   The accuracy of the result is implementation-dependent.
```

9.7.3 epsilon/0

9.7.3.1 Description

`epsilon` evaluates to the distance from `1.0` to the next largest floating point number, an implementation defined value. Thus, it allows the programmer to retrieve the relative accuracy when performing arithmetic with floating point numbers.

9.7.3.2 Examples

```
epsilon.
   Evaluates to an implementation defined value.
```