b) Else unifies `Head` with `Clause` and `true` with `Body`,

c) Searches sequentially through each dynamic user-defined procedure in the database and creates a list L of all the terms `clause(H, B)` such that

1) the database contains a clause whose head can be converted to a term `H`, and whose body coan be converted to a term `B`, and

2) `H` unifies with `Head`, and

3) `B` unifies with `Body`.

d) If a non-empty list is found, proceeds to 8.9.3.1 f,

e) Else the predicate fails.

f) Chooses the first element of the list L, removes the clause corresponding to it from the database, and the predicate succeeds.

g) If all the elements of the list L have been chosen, then the predicate fails,

h) Else chooses the first element of the list L which has not already been chosen, removes the clause, if it exists, corresponding to it from the database and the predicate succeeds.

`retract/1` is re-executable. On backtracking, continue at 8.9.3.1 g.

8.9.3.2 Template and modes

```
retract(+clause)
```

8.9.3.3 Errors

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

b) `Head` is not a predication
— `type_error(callable, Head)`.

c) The predicate indicator `Pred` of `Head` is not that of a dynamic procedure
— `permission_error(modify, static_procedure, Pred)`.

8.9.3.4 Examples

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

```
:- dynamic(legs/2).
legs(A, 4) :- animal(A).
legs(octopus, 8).
legs(A, 6) :- insect(A).
legs(spider, 8).
legs(B, 2) :- bird(B).

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

:- dynamic(foo/1).
foo(X) :- call(X), call(X).
foo(X) :- call(X) -> call(X).


retract(legs(octopus, 8)).
   Succeeds, retracting the clause
      'legs(octopus, 8)'.

retract(legs(spider, 6)).
   Fails.

retract( (legs(X, 2) :- T) ).
   Succeeds, unifying T with bird(X),
      and retracting the clause
         'legs(B, 2) :- bird(B)'.

retract( (legs(X, Y) :- Z) ).
   Succeeds, unifying Y with 4,
      and Z with animal(X),
      noting the list of clauses to be retracted
         = [ (legs(A, 4) :- animal(A)),
             (legs(A, 6) :- insect(A)),
             (legs(spider, 8) :- true) ],
      and retracting the clause
         'legs(A, 4) :- animal(A)'.
   On re-execution, succeeds,
      unifying Y with 6, and Z with insect(X),
      and retracting the clause
        'legs(A, 6) :- insect(A)'.
   On re-execution, succeeds, unifying Y with 8,
      and X with spider, and Z with true,
      and retracting the clause
        'legs(A, 8) :- animal(A)'.
   On re-execution, fails.

retract(insect(I)), write(I),
      retract(insect(bee)), fail.
   'retract(insect(I))' succeeds,
      unifying I with 'ant',
      noting the list of clauses to be retracted
         = [insect(ant), insect(bee)],
      and retracting the clause 'insect(ant)'.
   'write(ant)' succeeds, outputting 'ant'.
   'retract(insect(bee))' succeeds,
      noting the list of clauses to be retracted
         = [insect(bee)],
      and retracting the clause 'insect(bee)'.
   'fail' fails.
   On re-execution, 'retract(insect(bee))' fails.
   On re-execution, 'write(ant)' fails.
   On re-execution, 'retract(insect(I))' succeeds,
       unifying I with 'bee',
       noting the list of clauses to be retracted
          = [insect(bee)],
       [the clause 'insect(bee)' has already
          been retracted.]
   'write(bee)' succeeds, outputting 'bee'.
   'retract(insect(bee))' fails.
   On re-execution, 'write(bee)' fails.
   On re-execution, 'retract(insect(I))' fails.
   Fails.

retract(( foo(A) :- A, call(A) )).
   Succeeds, retracting the clause
      'foo(X) :- call(X), call(X)'.

retract(( foo(A) :- A -> B )).
   Succeeds, unifying A with B,
   and retracting the clause
      'foo(X) :- call(X) -> call(X)'.

retract( (X :- in_eec(Y)) ).
   instantiation_error.

retract( (4 :- X) ).
   type_error(callable, 4).
```

```
retract( (atom(X) :- X == '[]') ).
   permission_error(modify_clause,
      static_procedure, atom/1).
```

After these examples, the database is empty.

8.9.4 abolish/1

8.9.4.1 Description

`abolish(Pred)` is true.

Procedurally, `abolish(Pred)` removes from the database all clauses for the dynamic procedure specified by the predicate indicator `Pred` leaving the database in the same state as if the procedure had never existed.

8.9.4.2 Template and modes

```
abolish(@predicate_indicator)
```

8.9.4.3 Errors

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

b) `Pred` is a term `Name/Arity` and either `Name` or `Arity` is a variable
— `instantiation_error`.

c) `Pred` is a term `Name/Arity` and `Arity` is neither a variable nor an integer
— `type_error(integer, Arity)`.

d) `Pred` is a term `Name/Arity` and `Name` is neither a variable nor an atom
— `type_error(atom, Name)`.

e) The procedure specified by `Pred` is not that of a dynamic procedure
— `permission_error(modify, static_procedure, Pred)`.

8.9.4.4 Examples

```
abolish(foo/2).
   Succeeds, also undefines foo/2 if there exists
   a dynamic procedure with predicate foo/2.

abolish(foo/_).
   instantiation_error.

abolish(abolish/1).
   permission_error(modify_clause,
      static_procedure, abolish/1).
```

8.10 All solutions

These predicates create a list of all the solutions of a goal.

8.10.1 findall/3

8.10.1.1 Description

`findall(Term, Goal, Bag)` is true iff `Bag` unifies with the list of values to which a variable `X` not occurring in `Term` or `Goal` would be instantiated by successive resatisfaction of
`call(Goal), X=Term`
after systematic replacement of all variables in `X` by new variables.

Procedurally, `findall(Term, Goal, Bag)` is executed as follows:

a) Creates an empty list L,

b) Executes `call(G)`,

c) If it fails, proceeds to 8.10.1.1 g,

d) Else if it succeeds, appends a renamed copy (refrenamed-copyofaterm) of `Term` to L,

e) Re-executes `call(G)`,

f) Proceeds to 8.10.1.1 c,

g) Unifies L with `Bag`,

h) If the unification succeeds, the predicate succeeds,

i) Else the predicate fails.

8.10.1.2 Template and modes

```
findall(@term, @callable_term, ?list)
```

8.10.1.3 Errors

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

b) `Goal` is not a callable term
— `type_error(callable, Goal)`.

8.10.1.4 Examples

```
findall(X, (X=1; X=2), S).
   Succeeds, unifying S with [1, 2].

findall(X+Y, (X=1), S).
   Succeeds, unifying S with [1+_].

findall(X, fail, L).
   Succeeds, unifying S with [].

findall(X, (X=1; X=1), S).
   Succeeds, unifying S with [1, 1].

findall(X, (X=2; X=1), [1, 2]).
   Fails.

findall(X, Goal, S).
   instantiation_error.

findall(X, 4, S).
   type_error(callable, 4).
```

8.10.2 bagof/3

bagof/3 assembles as a list the solutions of a goal for each different instantiation of the free variables in that goal. The elements of each list are in order of solution, but the order in which each list is found is undefined.

8.10.2.1 Description

bagof(Template, Goal, Instances) is true iff:

— G is the iterated-goal term (7.1.6.3) of Goal, and

— FV is a witness (7.1.1.2) of the free variables set (7.1.1.4) of Goal with respect to Template, and

— Instances is a non-empty list of Template such that G is true, and

— Each element of Instances corresponds to a single binding of FV, and

— The elements of Instances are in order of solution.

Procedurally, bagof(Template, Goal, Instances) is executed as follows:

a) Let Witness be a witness (7.1.1.2) of the free variables set (7.1.1.4) of Goal with respect to Template,

b) Let G be the iterated-goal term (7.1.6.3) of Goal,

c) Executes the goal findall(Witness+Template, G, S),

d) If S is the empty list, then fails,

e) Else proceeds to step 8.10.2.1 f.

f) Chooses any element, W+T, of S.

g) Let WT_list be the largest proper sublist (7.1.6.4) of S such that, for each element WW+TT of WT_list, WW is a variant (7.1.6.1) of W,

h) Let T_list be the list such that, for each element WW+TT of WT_list, there is a corresponding element TT of T_list,

i) Let S_next be the largest proper sublist of S such that WW+TT is an element of S_next iff WW+TT is not an element WT_list,

j) Replaces S by S_next,

k) If T_list unifies with Instances, unifies Witness with each WW defined in 8.10.2.1 g, and succeeds,

l) Else proceeds to step 8.10.2.1 d.

bagof/3 is re-executable. On backtracking, continue at 8.10.2.1 d.

NOTES

1 Step 8.10.2.1 f does not define which element of those eligible will be chosen. The order of solutions for bagof/3 is thus undefined.

2 If the free variables set of Goal with respect to Template is empty, and Iterated_Goal succeeds, then the predicate can succeed only once.

3 This definition implies that the variables of Template and the variables in the existential variables set (7.1.1.3) of Goal remain uninstantiated after each success of bagof(Template, Goal, Instances).

8.10.2.2 Template and modes

```
bagof(@term, +callable_term, ?list)
```

8.10.2.3 Errors

a) G is a variable
— instantiation_error.

b) G is not a callable term
— type_error(callable, G).

8.10.2.4 Examples

```
bagof(X, (X=1 ; X=2), S).
   Free variables set: {}.
   Succeeds, unifying S with [1,2].

bagof(X, (X=1 ; X=2), X).
   Free variables set: {}.
   Succeeds, unifying X with [1,2].

bagof(X, fail, S).
   Free variables set: {}.
   Fails.

bagof(1, (Y=1 ; Y=2), L).
   Free variables set: {Y}.
   Succeeds, unifying L with [1],
      and Y with 1.
   On re-execution, succeeds, unifying L with [1],
      and Y with 2.
   [The order of solutions is undefined]

bagof(f(X, Y), (X=a ; Y=b), L).
   Free variables set: {}.
   Succeeds, unifying L with [f(a, _), f(_, b)].

bagof(X, Y^((X=1, Y=1) ; (X=2, Y=2)), S).
   Free variables set: {}.
   Succeeds, unifying S with [1, 2].

bagof(X, Y^((X=1 ; Y=1) ; (X=2, Y=2)), S).
   Free variables set: {}.
   Succeeds, unifying S with [1, _, 2].

bagof(X, (Y^(X=1 ; Y=2) ; X=3), S).
   Free variables set: {Y}.
   Warning: the procedure ^/2 is undefined.
   Succeeds, unifying S with [3], and Y with _.
   [Assuming the value associated with the flag
   'undefined_predicate' is 'warning'.]

bagof(X, (X=Y ; X=Z ; Y=1), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y, Z].
   On re-execution, succeeds, unifying S with [_],
      and Y with 1.

bagof(X, (X=Y ; X=Z), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y, Z].

bagof(X, a(X, Y), L).
   Clauses of a/2:
      a(1, f(_)).
```

```
      a(2, f(_)).
   Free variables set: {Y}.
   Succeeds, unifying L with [1, 2],
      and Y with f(_).

bagof(X, b(X, Y), L).
   Clauses of b/2:
      b(1, 1).
      b(1, 1).
      b(1, 2).
      b(2, 1).
      b(2, 2).
      b(2, 2).
   Free variables set: {Y}.
   Succeeds, unifying L with [1,1,2],
      and Y with 1.
   On re-execution, succeeds,
      unifying L with [1,2,2], and Y with 2.
   [The order of solutions is undefined]

bagof(X, Y^Z, L).
   instantiation_error.

bagof(X, 1, L).
   type_error(callable, 1).
```

The following fully worked examples explain bagof/3 in greater detail.

```
** Example: bagof(f(X,Y), (X=a;Y=b), L).

   Template = f(X,Y)
   Goal = (X=a;Y=b)
   Instances = L

   Iterated-goal term = (X=a;Y=b)
   Free variables set of
      Goal with respect to Template: {}
   step c -- findall(w+f(X,Y), (X=a;Y=b), S).
      S = [w+f(a,_), w+f(_,b)]
   step f -- W+T = w+f(_,b)
   step g -- WT_list = [w+f(a,_), w+f(_,b)]
   step h -- T_list = [f(a,_), f(_,b)]
   step i -- S_next = []
   Succeeds, unifying L with [f(a,_),f(_,b)].

   On re-execution,
   step d ---  Fails.

** Example: bagof(X,Y^((X=1;Y=1);(X=2,Y=2)),B).

   Template = X
   Goal = Y^((X=1;Y=1);(X=2,Y=2))
   Instances = B

   Iterated-goal term = ((X=1;Y=1);(X=2,Y=2))
   Free variables set of
      Goal with respect to Template: {}
   step c -- findall(w+X, ((X=1;Y=1);(X=2,Y=2)),
         S).
      S = [w+1, w+_, w+2]
   step f -- W+T = w+_
   step g -- WT_list = [w+1, w+_, w+2]
   step h -- T_list = [1, _, 2]
   step i -- S_next = []
   Succeeds, unifying B with [1, _, 2]

   On re-execution,
   step d ---  Fails.


** Example: bagof(X,(Y^(X=1;Y=2);X=3),C).

   Template = X
   Goal = (Y^(X=1;Y=2);X=3)
   Instances = C

   Iterated-goal term = (Y^(X=1;Y=2);X=3)
   Free variables set of
      Goal with respect to Template: {Y}
   step c -- findall(w(Y)+X, (Y^(X=1;Y=2);X=3),
         S).
      S = [w(_)+3]
   step f -- W+T = w(_)+3
   step g -- WT_list = [w(_)+3]
   step h -- T_list = [3]
   step i -- S_next = []
   Succeeds, unifying C with [3], and Y with _.

   On re-execution,
   step d ---  Fails.

   Note -- This assumes the first alternative
      fails because the procedure ^/2 has
      no defining clauses in the database,
      and the value associated with flag
      'undefined_predicate' is 'fail'.

** Example: bagof(X, (X=Y ; X=Z ; Y=1), D).

   Template = X
   Goal = (X=Y ; X=Z ; Y=1)
   Instances = D

   Iterated-goal term = (X=Y ; X=Z ; Y=1)
   Free variables set of
      Goal with respect to Template: {Y, Z}
   step c -- findall(w(Y,Z)+X, (X=Y ; X=Z ; Y=1),
         S).
   S = [w(X1,_)+X1, w(_,X2)+X2, w(1,_)+X3]
   step f -- W+T = w(_,X2)+X2
   step g -- WT_list = [w(X1,_)+X1, w(_,X2)+X2]
   step h -- T_list = [X1, X2]
   step i -- S_next = [w(1,_)+X3]
   Succeeds, unifying D with [X1, X2],
      and Y with X1, and Z with X2.

   On re-execution,
   step f -- W+T = w(1,_)+X3
   step g -- WT_list = [w(1,_)+X3]
   step h -- T_list = [X3]
   step i -- S_next = []
   Succeeds, unifying D with [X3], and Y with 1.

   On re-execution,
   step d ---  Fails.
```

8.10.3 setof/3

`setof/3` assembles as a list the solutions of a goal for each different instantiation of the free variables in that goal. The elements of each list are distinct and ordered, but the order in which each list is found is undefined.

8.10.3.1 Description

`setof(Template, Goal, Instances)` is true iff

— `G` is the iterated-goal term (7.1.6.3) of `Goal`, and

— `FV` is a witness (7.1.1.2) of the free variables set (7.1.1.4) of `Goal` with respect to `Template`, and

— Instance_list is a non-empty list of Template such that G is true, and

— Each element of Instance_list corresponds to a single binding of FV, and

— Instances is the sorted list (7.1.6.5) of Instance_list.

Procedurally, setof(Template, Goal, Instances) is executed as follows:

a) Let Witness be a witness of the free variables set (7.1.1.4) of Goal with respect to Template,

b) Let G be the iterated-goal term (7.1.6.3) of Goal,

c) Execute the goal findall(Witness+Template, G, S),

d) If S is the empty list, the predicate fails.

e) Else proceed to step 8.10.3.1 f.

f) Choose any element, W+T, of S.

g) Let WT_list be the largest proper sublist (7.1.6.4) of S such that, for each element WW+TT of WT_list, WW is a variant (7.1.6.1) of W,

h) Let T_list be a list such that, for each element WW+TT of WT_list, there is a corresponding element TT of T_list,

i) Let SS be the largest proper sublist of S such that WW+TT is an element of S_next iff WW+TT is not an element WT_list,

j) Let S_next be the sorted list (7.1.6.5) of S,

k) Replace S by S_next,

l) If T_list unifies with Instances, the predicate succeeds and unifies Witness with each WW defined in 8.10.3.1 g,

m) Else proceed to step 8.10.3.1 d.

setof/3 is re-executable. On backtracking, continue at 8.10.3.1 d.

8.10.3.2 Template and modes

```
setof(@term, +callable_term, ?list)
```

8.10.3.3 Errors

a) G is a variable
— instantiation_error.

b) G is not a callable term
— type_error(callable, G).

8.10.3.4 Examples

```
setof(X, (X=1; X=2), S).
   Free variables set: {}.
   Succeeds, unifying S with [1,2].

setof(X, (X=1; X=2), X).
   Free variables set: {}.
   Succeeds, unifying X with [1,2].

setof(X, (X=2; X=1), S).
   Free variables set: {}.
   Succeeds, unifying S with [1,2].

setof(X, (X=2; X=2), S).
   Free variables set: {}.
   Succeeds, unifying S with [2].

setof(X, (X=Y; X=Z), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y, Z] or [Z, Y].
   [The solution is implementation dependent.]

setof(X, fail, S).
   Free variables set: {}.
   Fails.

setof(1, (Y=2 ; Y=1), L).
   Free variables set: {Y}.
   Succeeds, unifying L with [1], and
     Y with 1.
   On re-execution, succeeds,
     unifying L with [1], and Y with 2.
   [The order of solutions is undefined]

setof(f(X,Y), (X=a ; Y=b), L).
   Free variables set: {}.
   Succeeds, unifying L with [f(_,b),f(a,_)].

setof(X, Y^((X=1, Y=1) ; (X=2, Y=2)), S).
   Free variables set: {}.
   Succeeds, unifying S with [1,2].

setof(X, Y^((X=1 ; Y=1) ; (X=2, Y=2)), S).
   Free variables set: {}.
   Succeeds, unifying S with [_,1,2].

setof(X, (Y^(X=1 ; Y=2) ; X=3), S).
   Free variables set: {Y}.
   Warning: the procedure ^/2 is undefined.
   Succeeds, unifying S with [3], and Y with _.
   [Assuming the value associated with the flag
   'undefined_predicate' is 'warning'.]

setof(X, (X=Y ; X=Z ; Y=1), S).
   Free variables set: {Y, Z}.
   Succeeds, unifying S with [Y,Z] or [Z,Y].
   On re-execution, succeeds, unifying S with [_],
      and Y with 1.

setof(X, a(X, Y), L).
   Clauses of a/2:
      a(1, f(_)).
      a(2, f(_)).
   Free variables set: {Y}.
   Succeeds, unifying L with [1, 2],
      and Y with f(_).

The following examples assume that member/2
   is defined with the following clauses:
      member(X, [X | _]).
      member(X, [_ | L]) :-
         member(X, L).

setof(X, member(X,[f(U,b),f(V,c)]), L).
   Free variables set: {U, V}.
   Succeeds, unifying L with [f(U,b),f(V,c)] or
      with [f(V,c),f(U,b)].

setof(X, member(X,[f(U,b),f(V,c)]),
      [f(a,c),f(a,b)]).
   Free variables set: {U, V}.
   Implementation dependent.
```

```
setof(X, member(X,[f(b,U),f(c,V)]),
      [f(b,a),f(c,a)]).
   Free variables set: {U, V}.
   Succeeds, unifying U with a, and V with a.

setof(X, member(X,[V,U,f(U),f(V)]), L).
   Free variables set: {U, V}.
   Succeeds, unifying L with [U,V,f(U),f(V)] or
      with [V,U,f(V),f(U)].

setof(X, member(X,[V,U,f(U),f(V)]),
      [a,b,f(a),f(b)]).
   Free variables set: {U, V}.
   Implementation dependent.
   Succeeds, unifying U with a, and V with B;
     or, unifying U with b, and V with a.

setof(X, member(X,[V,U,f(U),f(V)]),
      [a,b,f(b),f(a)]).
   Free variables set: {U, V}.
   Fails.

setof(X,
      (exists(U,V)^member(X,[V,U,f(U),f(V)])),
      [a,b,f(b),f(a)]).
   Free variables set: {}.
   Succeeds.

The following examples assume that b/2 is defined
   with the following clauses:
      b(1, 1).
      b(1, 1).
      b(1, 2).
      b(2, 1).
      b(2, 2).
      b(2, 2).

setof(X, b(X, Y), L).
   Free variables set: {Y}.
   Succeeds, unifying L with [1, 2], and Y with 1.
   On re-execution, succeeds,
      unifying L with [1, 2], and Y with 2.
   [The order of solutions is undefined]

setof(X-Xs,Y^setof(Y,b(X,Y),Xs),L).
   Free variables set: {}.
   Succeeds, unifying L with [1-[1,2],2-[1,2]].
   [Each list is independently ordered]

setof(X-Xs,setof(Y,b(X,Y),Xs),L).
   Free variables set: {Y}.
   Succeeds, unifying L with [1-[1,2],2-[1,2]],
      and Y with _.
   [Each list is independently ordered]

setof(X-Xs,bagof(Y,d(X,Y),Xs),L).
   Clauses of d/3:
      d(1,1).
      d(1,2).
      d(1,1).
      d(2,2).
      d(2,1).
      d(2,2).
   Free variables set: {Y}.
   Succeeds,
      unifying L with [1-[1,2,1],2-[2,1,2]],
      and Y with _.
```

8.11 Stream selection and control

These predicates link an external source/sink with a Prolog stream, its stream identifier and stream alias. They enable the source/sink to be opened and closed, and its properties found during execution.

NOTE — The use of these predicates may cause a Resource Error (7.12.2 h) because, for example, the program has opened too many streams, or a file or disk is full. The use of these predicates may also cause a System Error (7.12.2 j) because the operating system is reporting a problem.

The precise reasons for such errors, the side effects which have occurred, and the way they can be circumvented cannot be specified in this draft International Standard.

8.11.1 current_input/1

8.11.1.1 Description

`current_input(Stream)` is true iff the stream identifier `Stream` identifies the current input stream (7.10.2.4).

Procedurally, `current_input(Stream)` unifies `Stream` with the stream identifier of the current input stream.

8.11.1.2 Template and modes

```
current_input(?stream)
```

8.11.1.3 Errors

None.

8.11.2 current_output/1

8.11.2.1 Description

`current_output(Stream)` is true iff the stream identifier `Stream` identifies the current output stream (7.10.2.4).

Procedurally, `current_output(Stream)` unifies `Stream` with the stream identifier of the current output stream.

8.11.2.2 Template and modes

```
current_output(?stream)
```

8.11.2.3 Errors

None.

8.11.3 set_input/1

8.11.3.1 Description

`set_input(S_or_a)` is true.

Procedurally, set_input(S_or_a) is executed as follows:

a) If S_or_a is not a stream identifier or alias for an input stream which is currently open, then there shall be an error,

b) Else set the stream associated with stream identifier or alias S_or_a to be the current input stream, and succeeds.

8.11.3.2 Template and modes

```
set_input(@stream_or_alias)
```

8.11.3.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an output stream
— permission_error(input, stream, S_or_a).

8.11.4 set_output/1

8.11.4.1 Description

set_output(S_or_a) is true.

Procedurally, set_output(S_or_a) is executed as follows:

a) If S_or_a is not a stream identifier or alias for an output stream which is currently open, then there shall be an error,

b) Else set the stream associated with stream identifier or alias S_or_a to be the current output stream, and succeeds.

8.11.4.2 Template and modes

```
set_output(@stream_or_alias)
```

8.11.4.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

8.11.5 open/3

8.11.5.1 Description

open(Source_sink, Mode, Stream) is true iff
open(Source_sink, Mode, Stream, []).

8.11.5.2 Template and modes

```
open(@source_sink, @io_mode, -stream)
```

8.11.5.3 Errors

a) Source_sink is a variable
— instantiation_error.

b) Mode is a variable
— instantiation_error.

c) Source_sink is neither a variable nor a source/sink
— domain_error(source_sink, Source_sink).

d) Mode is neither a variable nor an atom
— type_error(atom, Mode).

e) Stream is not a variable
— type_error(variable, Stream).

f) Mode is an atom but not an I/O mode
— domain_error(io_mode, Mode).

g) The source/sink specified by Source_sink cannot be opened
— permission_error(open, source/sink, Source_sink).

8.11.5.4 Examples

```
open('/user/dave/data', read, DD).
   Succeeds.
   [It opens the text file '/user/dave/data'
   for input, and unifies DD with a
   stream identifier for the stream.]
```

8.11.6 open/4

8.11.6.1 Description

open(Source_sink, Mode, Stream, Options) is true.

Procedurally, open(Source_sink, Mode, Stream, Options) is executed as follows:

a) Opens the source/sink Source_sink for input or output as indicated by I/O mode Mode and the list of stream-options Options.

b) Unifies Stream with the stream identifier which is to be associated with this stream,

c) The predicate succeeds.

8.11.6.2 Template and modes

```
open(@source_sink, @io_mode, -stream,
   @io_options)
```

8.11.6.3 Errors

a) `Source_sink` is a variable
— `instantiation_error`.

b) `Mode` is a variable
— `instantiation_error`.

c) `Options` is a variable
— `instantiation_error`.

d) `Options` is a list with an element `E` which is a variable
— `instantiation_error`.

e) `Source_sink` is neither a variable nor a source/sink
— `domain_error(source_sink, Source_sink)`.

f) `Mode` is neither a variable nor an atom
— `type_error(atom, Mode)`.

g) `Options` is neither a variable nor a list
— `type_error(list, Options)`.

h) An element `E` of the `Options` list is neither a variable nor a stream-option
— `domain_error(stream_option, E)`.

i) `Stream` is not a variable
— `type_error(variable, Stream)`.

j) `Mode` is an atom but not an I/O mode
— `domain_error(io_mode, Mode)`.

k) The source/sink specified by `Source_sink` cannot be opened
— `permission_error(open, source_sink, Source_sink)`.

l) An element `E` of the `Options` list is `alias(A)` and `A` is already associated with an open stream
— `permission_error(open, source_sink, alias(A))`.

m) An element `E` of the `Options` list is `reposition(true)` and it is not possible to reposition this stream
— `permission_error(open, source_sink, reposition(true))`.

NOTE — A permission error when `Mode` is `write` or `append` means that `Source_sink` does not specify a sink that can be created, for example, a specified disk or directory does not exist. If `Mode` is `read` then it is also possible that the file specification is valid but the file does not exist.

8.11.6.4 Examples

```
open('/user/dave/data', read, DD, []).
   Succeeds.
   [It opens the text file '/user/dave/data'
   for input, and unifies DD with a
   stream identifier for the stream.]
```

8.11.7 close/1

8.11.7.1 Description

`close(S_or_a)` is true iff
`close(S_or_a, []).`

8.11.7.2 Template and modes

```
close(@stream_or_alias)
```

8.11.7.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

8.11.8 close/2

This built-in predicate closes the stream associated with stream identifier or alias `S_or_a` if it is open. The behaviour of this predicate may be modified by specifying a list of close-options (7.10.2.12) in the `Options` parameter.

8.11.8.1 Description

`close(S_or_a, Options)` is true.

Procedurally, `close(S_or_a, Options)` is executed as follows:

a) If `S_or_a` is an atom which is not the alias of a currently open stream, then the predicate succeeds,

b) Else if `S_or_a` is a valid stream representation but does not represent a currently open stream, then the predicate succeeds,

c) Else, any output which is currently buffered by the processor for the stream associated with `S_or_a` is sent to that stream,

d) If the stream identifier or alias `S_or_a` is the standard input stream or the standard output stream, then the predicate succeeds,

e) Else if the stream associated with `S_or_a` is not the current input stream then proceeds to 8.11.8.1 g,

f) The current input stream becomes the standard input stream `user_input`,

g) If the stream associated with `S_or_a` is not the current output stream then proceeds to 8.11.8.1 i,

h) The current output stream becomes the standard output stream `user_output`,

i) Closes the stream associated with `S_or_a` and deletes any alias associated with that stream,

j) The predicate succeeds.

The above implies that when a stream `Stream` has already been closed, a subsequent call `close(S_or_a)` simply succeeds.

8.11.8.2 Template and modes

```
close(@stream_or_alias, @close_options)
```

8.11.8.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Options` is a variable
— `instantiation_error`.

c) `Options` is a list with an element `E` which is a variable
— `instantiation_error`.

d) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

e) `Options` is neither a variable nor a list
— `type_error(list, Options)`.

f) An element `E` of the `Options` list is neither a variable nor a close-option
— `domain_error(close_option, E)`.

8.11.9 flush_output/0

NOTE — Flushing an output stream is explained in clause 7.10.2.10.

8.11.9.1 Description

`flush_output` is true.

Procedurally, `flush_output` is executed as follows:

a) Any output which is currently buffered by the processor for the current output stream is sent to that stream,

b) The predicate succeeds.

8.11.9.2 Template and modes

```
flush_output
```

8.11.9.3 Errors

None.

8.11.10 flush_output/1

NOTE — Flushing an output stream is explained in clause 7.10.2.10.

8.11.10.1 Description

`flush_output(S_or_a)` is true.

Procedurally, `flush_output(S_or_a)` is executed as follows:

a) Any output which is currently buffered by the processor for the stream associated with stream identifier or alias `S_or_a` is sent to that stream,

b) The predicate succeeds.

8.11.10.2 Template and modes

```
flush_output(@stream_or_alias)
```

8.11.10.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

d) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

8.11.11 stream_property/2

8.11.11.1 Description

`stream_property(Stream, Property)` is true iff the stream identified by the stream identifier `Stream` has stream property (7.10.2.13) `Property`.

Procedurally, `stream_property(Stream, Property)` is executed as follows:

a) Computes SP, the set of all pairs `(S,P)` such that `S` is a currently open stream which has property `P`,

b) If SP is empty, the predicate fails,

c) Else, chooses one pair `(S,P)` in SP and removes it from the set,

d) Unifies `S` with `Stream` and `P` with `Property`,

e) If the unification succeeds, the predicate succeeds,

f) Else, proceeds to 8.11.11.1 b.

`stream_property(Stream, Property)` is re-executable. On backtracking, continue at 8.11.11.1 b.

NOTE — When used in non-determinate ways, stream_property shall exhibit logical semantics for state changes. For example:

```
:- stream_property(S, P),
   write(S:P),
```

```
    nl,
    close(S),
    fail.
```

shall show all the properties that all open streams had before this goal was run. Note that this example may call `close(S)` several times for each stream S, but this does not cause any problem since close simply succeeds if called on a stream which is already closed.

8.11.11.2 Template and modes

```
stream_property(?stream, ?stream_property)
```

8.11.11.3 Errors

None.

8.11.11.4 Examples

```
stream_property(S, file_name(F))
    If S is instantiated, succeeds,
       unifying F with the name of the file
       to which it is connected,
    Else succeeds, unifying S with a
       stream identifier and F with the name
       of the file to which it is connected;
       on re-execution, succeeds in turn with
       each stream which is connected to a file.

stream_property(S, output)
    If S is instantiated, succeeds iff output
       is permitted on this stream,
    Else succeeds, unifying S with a
       stream identifier which is open for
       output; on re-execution, succeeds in turn
       with each stream which is open for output.
```

8.11.12 at_end_of_stream/0

8.11.12.1 Description

`at_end_of_stream` is true iff the current input stream has a stream position end-of-stream or past-end-of-stream (7.10.2.9, 7.10.2.13).

8.11.12.2 Template and modes

```
at_end_of_stream
```

8.11.12.3 Errors

None.

8.11.13 at_end_of_stream/1

8.11.13.1 Description

`at_end_of_stream(S_or_a)` is true iff the stream associated with stream identifier or alias `S_or_a` has a stream position end-of-stream or past-end-of-stream (7.10.2.9, 7.10.2.13).

8.11.13.2 Template and modes

```
at_end_of_stream(@stream_or_alias)
```

8.11.13.3 Errors

None.

8.11.14 set_stream_position/2

8.11.14.1 Description

`set_stream_position(S_or_a, Position)` is true.

Procedurally, `set_stream_position( S_or_a, Position)` is executed as follows:

a) Sets the stream position of the stream associated with stream identifier or alias `S_or_a` to `Position`,

b) Succeeds.

NOTE — Normally, `Position` will previously have been returned as a `position/1` stream property of the stream.

8.11.14.2 Template and modes

```
set_stream_position(@stream_or_alias,
   @stream_position)
```

8.11.14.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Position` is a variable
— `instantiation_error`.

c) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

d) `Position` is neither a variable nor a stream position
— `domain_error(stream_position, Position)`.

e) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

f) `S_or_a` has stream property `reposition(false)`
— `permission_error(reposition, stream, S_or_a)`.

8.12 Character input/output

These built-in predicates enable a single character or byte to be input and output from a stream.

8.12.1 get_char/1

8.12.1.1 Description

`get_char(Char)` is true iff
`(current_input(S), get_char(S, Char))`.

8.12.1.2 Template and modes

`get_char(?character)`

8.12.1.3 Errors

a) The current input stream has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `existence_error(past_end_of_stream, current_input_stream)`.

8.12.1.4 Examples

```
get_char(Char).
   current input stream
      qwerty ...
   Char is unified with the atom 'q' and
   the current input stream becomes
      werty ...
```

8.12.2 get_char/2

8.12.2.1 Description

`get_char(S_or_a, Char)` is true iff

— The stream associated with stream identifier or alias `S_or_a` is a text stream, and `Char` unifies with the next character to be read from `S_or_a`, or

— The stream associated with stream identifier or alias `S_or_a` is a binary stream, and `Char` unifies with the next byte to be read from `S_or_a`.

Procedurally, `get_char(S_or_a, Char)` is executed as follows:

a) If the stream associated with `S_or_a` has stream properties `end_of_stream(past)` and `eof_action(A)` then performs the action appropriate to the value of `A` specified in clause 7.10.2.11.

b) If the stream position is end-of-stream, proceeds to 8.12.2.1 h,

c) If the stream associated with `S_or_a` is a text stream, reads the next character `C` from the stream associated with `S_or_a`,

d) If the stream associated with `S_or_a` is a binary stream, reads the next byte `C` from the stream associated with `S_or_a`,

e) Advances the stream position of the stream associated with `S_or_a` by one character,

f) If `C` unifies with `Char`, the predicate succeeds,

g) Else the predicate fails.

h) Sets the stream position so that it is past-end-of-stream,

i) If the atom `end_of_file` unifies with `C`, the predicate succeeds,

j) Else the predicate fails.

8.12.2.2 Template and modes

`get_char(@stream_or_alias, ?character)`

8.12.2.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

d) `S_or_a` has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `existence_error(past_end_of_stream, S_or_a)`.

e) `S_or_a` is an output stream
— `permission_error(input, stream, S_or_a)`.

8.12.2.4 Examples

```
get_char(Stream, Char).
   The contents of Stream are
      qwerty ...
   Char is unified with 'q' and
   Stream is left as
      werty ...

get_char(Stream, Char).
   The contents of Stream are
      'qwerty' ...
   Char is unified with '\'' (the
   atom containing just a single
   quote) and Stream is left as
      qwerty' ...

get_char(my_file, '\13\').
   The contents of my_file are
      \13\10\newline ...
   Succeeds and my_file is left as
      \10\newline ...

get_char(Stream, p).
   The contents of Stream are
      qwerty ...
   Fails and
   Stream is left as
      werty ...

get_char(user_output, X).
   The contents of user_output are
      qwerty ...
   permission_error(input, stream, user_output).
   user_output is left as
      qwerty ...
```

```
get_char(S, Char).
   Stream position of S is end-of-stream.
   Char is unified with end_of_file and
   Stream position of S is past-end-of-stream.
```

8.12.3 put_char/1

8.12.3.1 Description

`put_char(Char)` is true iff
`(current_output(S), put_char(S, Char))`.

8.12.3.2 Template and modes

```
put_char(@character)
```

8.12.3.3 Errors

a) `Char` is a variable
— `instantiation_error`.

b) `Char` is neither a variable nor a character
— `type_error(character, Char)`.

c) `Char` is neither a variable nor a character (7.1.4.1)
— `representation_error(character)`.

8.12.3.4 Examples

```
put_char(t).
   current output stream
      ... qwer
   Succeeds and leaves that stream
      ... qwert
```

8.12.4 put_char/2

8.12.4.1 Description

Procedurally, `put_char(S_or_a, Char)` is executed as follows:

a) Outputs the character `Char` to the stream associated with stream identifier or alias `S_or_a`.

b) Changes the stream position on the stream associated with `S_or_a` to take account of the character which has been output,

c) The predicate succeeds.

8.12.4.2 Template and modes

```
put_char(@stream_or_alias, @character)
```

8.12.4.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Char` is a variable
— `instantiation_error`.

c) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

d) `Char` is neither a variable nor a character
— `type_error(character, Char)`.

e) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

f) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

g) `Char` is neither a variable nor a character (7.1.4.1)
— `representation_error(character)`.

8.12.4.4 Examples

```
put_char(Stream,t).
   If the stream associated with Stream contains
      ... qwer
   Succeeds and leaves that stream
      ... qwert

put_char(my_file, C).
   instantiation_error.

put_char(Str, 'ty').
   type_error(character, ty).

put_char(Str, 'A').
   If the stream associated with Stream contains
      ... qwer
   Succeeds and leaves that stream
      ... qwerA
```

8.12.5 nl/0

8.12.5.1 Description

`nl` is true iff
`(current_output(S), nl(S))`.

8.12.5.2 Template and modes

```
nl
```

8.12.5.3 Errors

None.

8.12.5.4 Examples

```
nl, put_char(a).
   Current output stream
      ... qwer
   Succeeds and leaves that stream
      ... qwer
      a
```

8.12.6 nl/1

8.12.6.1 Description

`nl(S_or_a)` is true.

Procedurally, `nl(S_or_a)` is executed as follows:

a) Outputs a new line character to the stream associated with `S_or_a`,

b) Succeeds.

NOTES

1 This built-in predicate terminates the current line or record.

2 `nl(S_or_a)` is equivalent to `put_char(S_or_a, '\n')`.

8.12.6.2 Template and modes

```
nl(@stream_or_alias)
```

8.12.6.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

d) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

8.12.6.4 Examples

```
nl(st), put_char(st, a).
   If the stream associated with st contains
      ... qwer
   Succeeds and leaves that stream
      ... qwer
      a

nl(Str).
   instantiation_error.

nl([my_file]).
   domain_error(stream_or_alias, [my_file]).
```

8.13 Character code input/output

These built-in predicates enable a single byte to be input and output from a stream.

8.13.1 get_code/1

8.13.1.1 Description

`get_code(Code)` is true iff
`(current_input(S), get_code(S, Code))`.

8.13.1.2 Template and modes

```
get_code(?character_code)
```

8.13.1.3 Errors

a) The current input stream has stream properties `end_of_stream(past)` and `eof_action(error)` (7.10.2.9, 7.10.2.11, 7.10.2.13)
— `existence_error(past_end_of_stream, current_input_stream)`

8.13.1.4 Examples

```
get_code(Code).
   current input stream
      qwerty ...
   Code is unified with 0'q and
   the current input stream becomes
      werty ...
```

8.13.2 get_code/2

8.13.2.1 Description

If the stream associated with stream identifier or alias `S_or_a` is a text stream then `get_code(S_or_a, Int)` is true iff `Int` unifies with the character code (7.1.2.2) corresponding to the next character to be read from `S_or_a`, else if `S_or_a` is a binary stream `get_code(S_or_a, Int)` is true iff `Int` unifies with the next byte to be read from `S_or_a`.

Procedurally, `get_code(S_or_a, Int)` is executed as follows:

a) If the stream associated with `S_or_a` has stream properties `end_of_stream(past)` and `eof_action(A)` then performs the action appropriate to the value of `A` specified in clause 7.10.2.11.

b) If there is no more data in the stream, proceeds to 8.13.2.1 g,

c) Else reads the next character with character code `I`, from the stream associated with `S_or_a`,

d) Advances the stream position of the stream associated with `S_or_a` by one character,

e) If `I` unifies with `Int`, the predicate succeeds,

f) Else the predicate fails.

g) Sets the stream position so that it is past-end-of-stream,

h) If the integer -1 unifies with `Int`, the predicate succeeds,

i) Else the predicate fails.

8.13.2.2 Template and modes

```
get_code(@stream_or_alias, ?character_code)
```

8.13.2.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a has stream properties end_of_stream(past) and eof_action(error) (7.10.2.9, 7.10.2.11, 7.10.2.13)
— existence_error(past_end_of_stream, S_or_a)

e) S_or_a is an output stream
— permission_error(input, stream, S_or_a).

8.13.2.4 Examples

```
get_code(Stream, Code).
   The contents of Stream are
      qwerty ...
   Code is unified with 0'q and
   Stream is left as
      werty ...

get_code(Stream, Code).
   The contents of Stream are
      'qwerty' ...
   Code is unified with 0''' and
   Stream is left as
      qwerty' ...

get_code(my_file, 0'\13\).
   The contents of my_file are
      0'\13\0'\10\\newline ...
   Succeeds and my_file is left as
      0'\10\newline ...

get_code(Stream, 0'p).
   The contents of Stream are
      qwerty ...
   Fails and
   Stream is left as
      werty ...

get_code(user_output, X).
   The contents of Stream are
      qwerty ...
   permission_error(input, stream, user_output).
   Stream is left as
      qwerty ...

get_code(S, Code).
   If stream position of the
      stream associated with S is end-of-stream
   then
      Succeeds, unifying Code with 1, and
      Sets stream position of S to
         past-end-of-stream.
```

8.13.3 put_code/1

8.13.3.1 Description

put_code(Code) is true iff
(current_output(S), put_code(S, Code)).

8.13.3.2 Template and modes

```
put_code(@character_code)
```

8.13.3.3 Errors

a) Code is a variable
— instantiation_error.

b) Code is neither a variable nor an integer
— type_error(integer, Code).

c) Code is neither a variable nor a character code (7.1.2.2)
— representation_error(character_code).

8.13.3.4 Examples

```
put_code(0't).
   current output stream
      ... qwer
   Succeeds and leaves that stream
      ... qwert
```

8.13.4 put_code/2

8.13.4.1 Description

Procedurally, put_code(S_or_a, Code) is executed as follows:

a) Outputs the character Code to the stream associated with stream identifier or alias S_or_a.

b) Changes the stream position on the stream associated with S_or_a to take account of the character which has been output,

c) The predicate succeeds.

8.13.4.2 Template and modes

```
put_code(@stream_or_alias, @character_code)
```

8.13.4.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) Code is a variable
— instantiation_error.

c) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

d) Code is neither a variable nor an integer
— type_error(integer, Code).

e) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

f) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

g) Code is neither a variable nor a character code (7.1.2.2)
— representation_error(character_code)

8.13.4.4 Examples

```
put_code(Stream, 0't).
   If the stream associated with Stream contains
      ... qwer
   Succeeds and leaves that stream
      ... qwert

put_code(my_file, C).
   instantiation_error.

put_code(Str, 'ty').
   type_error(integer, 'ab').

put_code(Str, 65).
   If the stream associated with Stream contains
      ... qwer
   Succeeds and leaves that stream
      ... qwerA
```

8.14 Term input/output

These predicates enable a Prolog term to be input from or output to a stream. The syntax of such terms can also be altered by changing the operators, and making some characters equivalent to one another.

8.14.1 read_term/2

8.14.1.1 Description

read_term(Term, Options) is true iff
(current_input(S),
read_term(S, Term, Options)).

8.14.1.2 Template and modes

```
read_term(?term, +read_options_list)
```

8.14.1.3 Errors

a) One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
— syntax_error.

b) Options is a variable
— instantiation_error.

c) Options is a list with an element E which is a variable
— instantiation_error.

d) Options is neither a variable nor a list
— type_error(list, Options).

e) An element E of the Options list is neither a variable nor a valid read-option
— domain_error(read_option, E).

8.14.2 read_term/3

8.14.2.1 Description

read_term(S_or_a, Term, Options) is true iff Term unifies with T, where T. is a read-term which has been constructed by inputting and parsing characters from the stream associated with stream identifier or alias S_or_a (see 7.10.4).

The read-options (7.10.3) specified in Options will be instantiated to provide additional information about the term which is read.

NOTE — The effect of this predicate may be modified by calling the built-in predicate char_conversion/2 (8.14.15), and if the value associated with the flag char_conversion (7.11.2.1) is on.

8.14.2.2 Template and modes

```
read_term(@stream_or_alias, ?term,
  +read_options_list)
```

8.14.2.3 Errors

a) One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
— syntax_error.

b) S_or_a is a variable
— instantiation_error.

c) Options is a variable
— instantiation_error.

d) Options is a list with an element E which is a variable
— instantiation_error.

e) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

f) Options is neither a variable nor a list
— type_error(list, Options).

g) An element E of the Options list is neither a variable nor a valid read-option
— domain_error(read_option, E).

h) An element E of the Options list is alias(A) and A is already associated with an open stream
— domain_error(read_option, E).

i) An element E of the Options list is reposition(true) and it is not possible to reposition this stream
— permission_error(reposition, E).

j) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

k) S_or_a is an output stream
— permission_error(input, stream, S_or_a).

8.14.3 read/1

8.14.3.1 Description

read(Term) is true iff
(current_input(S), read_term(S, Term, [])).

8.14.3.2 Template and modes

read(?term)

8.14.3.3 Errors

a) One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
— syntax_error.

8.14.3.4 Examples

```
read(T).
   current input stream is
term1. term2. ...
   Succeeds, unifying T with term1.
   The current input stream is left as
 term2. ...

read(term1).
   current input stream is
term1. term2. ...
   Succeeds.
   current input stream is left as
 term2. ...

read(T).
   current input stream is
3.1.  term2. ...
   Succeeds, unifying T with 3.1.
   The current input stream is left as
  term2. ...

read(4.1).
   current input stream is
3.1.  term2. ...
   Fails.
   The current input stream is left as
  term2 ...

read(T).
   current input stream is
foo 123. term2. ...
   and foo is not a current prefix operator
   syntax_error.
   The current input stream is left as
 term2. ...

read(T).
   current input stream is
3.1
   syntax_error.
   The current input stream is left with
   position past-end-of-stream.
```

8.14.4 read/2

8.14.4.1 Description

read(S_or_a, Term) is true iff
read_term(S_or_a, Term, []).

8.14.4.2 Template and modes

read(@stream_or_alias, ?term)

8.14.4.3 Errors

a) One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
— syntax_error.

b) S_or_a is a variable
— instantiation_error.

c) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

d) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

e) S_or_a is an output stream
— permission_error(input, stream, S_or_a).

8.14.5 write_term/2

8.14.5.1 Description

write_term(Term, Options) is true iff
(current_output(S),
write_term(S, Term, Options)).

8.14.5.2 Template and modes

write_term(@term, @write_options_list)

8.14.5.3 Errors

a) Options is a variable
— instantiation_error.

b) Options is a list with an element E which is a variable
— instantiation_error.

c) Options is neither a variable nor a list
— type_error(list, Options).

d) An element E of the Options list is neither a variable nor a valid write-option
— domain_error(write_option, E).

8.14.6 write_term/3

8.14.6.1 Description

`write_term(S_or_a, Term, Options)` is true.

Procedurally, `write_term(S_or_a, Term, Options)` is executed as follows:

a) Outputs `Term` to the stream associated with stream identifier or alias `S_or_a` in a form which is defined by the write-options list (7.10.5) `Options`,

b) Succeeds.

8.14.6.2 Template and modes

```
write_term(@stream_or_alias, @term,
  @write_options_list)
```

8.14.6.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `Options` is a variable
— `instantiation_error`.

c) `Options` is a list with an element `E` which is a variable
— `instantiation_error`.

d) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

e) `Options` is neither a variable nor a list
— `type_error(list, Options)`.

f) An element `E` of the `Options` list is neither a variable nor a valid write-option
— `domain_error(write_option, E)`.

g) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

h) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

8.14.6.4 Examples

```
write_term(S, [1,2,3], []).
   Succeeds, outputting the characters
[1,2,3]
   to the stream associated with S.

write_term(S, [1,2,3], [ignore_ops(true)]).
   Succeeds, outputting the characters
.(1,.(2,.(3,[ ])))
   to the stream associated with S.

write_term(S, '1<2', []).
   Succeeds, outputting the characters
1<2
   to the stream associated with S.

write_term(S, '1<2', [quoted(true)]).
   Succeeds, outputting the characters
'1<2'
   to the stream associated with S.

write_term(S, '$VAR'(0), [numbervars(true)]).
    Succeeds, outputting the character
A
    to the stream associated with S.

write_term(S, '$VAR'(1), [numbervars(true)]).
    Succeeds, outputting the character
B
    to the stream associated with S.

write_term(S, '$VAR'(25), [numbervars(true)]).
    Succeeds, outputting the characters
Z
    to the stream associated with S.

write_term(S, '$VAR'(26), [numbervars(true)]).
    Succeeds, outputting the character
A1
    to the stream associated with S.

write_term(S, '$VAR'(51), [numbervars(true)]).
    Succeeds, outputting the characters
Z1
    to the stream associated with S.

write_term(S, '$VAR'(52), [numbervars(true)]).
    Succeeds, outputting the characters
A2
    to the stream associated with S.
```

8.14.7 write/1

8.14.7.1 Description

`write(Term)` is true iff
`(current_output(S), write(S, Term))`.

8.14.7.2 Template and modes

```
write(@term)
```

8.14.7.3 Errors

None.

8.14.8 write/2

8.14.8.1 Description

`write(S_or_a, Term)` is true iff
`write_term(S_or_a, Term, [numbervars(true)])`.

8.14.8.2 Template and modes

```
write(@stream_or_alias, @term)
```

8.14.8.3 Errors

a) `S_or_a` is a variable

— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

8.14.8.4 Examples

```
write(out, [1,2,3]).
   Succeeds, outputting the characters
[1,2,3]
   to the stream associated with the alias 'out'.

write(out, 1<2).
   Succeeds, outputting the characters
1<2
   to the stream associated with the alias 'out'.

write(out, '1<2').
   Succeeds, outputting the characters
1<2
   to the stream associated with the alias 'out'.

write(out, '$VAR'(0)<'$VAR'(1)).
   Succeeds, outputting the characters
A<B
   to the stream associated with the alias 'out'.
```

8.14.9 writeq/1

8.14.9.1 Description

writeq(Term) is true iff
(current_output(S), writeq(S, Term)).

8.14.9.2 Template and modes

```
writeq(@term)
```

8.14.9.3 Errors

None.

8.14.10 writeq/2

8.14.10.1 Description

writeq(S_or_a, Term) is true iff
write_term(S_or_a, Term,
[quoted(true), numbervars(true)]).

8.14.10.2 Template and modes

```
writeq(@stream_or_alias, @term)
```

8.14.10.3 Errors

a) S_or_a is a variable
— instantiation_error.

b) S_or_a is neither a variable nor a stream identifier or alias
— domain_error(stream_or_alias, S_or_a).

c) S_or_a is not associated with an open stream
— existence_error(stream, S_or_a).

d) S_or_a is an input stream
— permission_error(output, stream, S_or_a).

8.14.10.4 Examples

```
writeq(out, [1,2,3]).
   Succeeds, outputting the characters
[1,2,3]
   to the stream associated with the alias 'out'.

writeq(out, 1<2).
   Succeeds, outputting the characters
1<2
   to the stream associated with the alias 'out'.

writeq(out, '1<2').
   Succeeds, outputting the characters
'1<2'
   to the stream associated with the alias 'out'.

writeq(out, '$VAR'(0)<'$VAR'(1)).
   Succeeds, outputting the characters
A<B
   to the stream associated with the alias 'out'.
```

8.14.11 write_canonical/1

8.14.11.1 Description

write_canonical(T) is true iff
(current_output(S), write_canonical(S, T)).

8.14.11.2 Template and modes

```
write_canonical(@term)
```

8.14.11.3 Errors

None.

8.14.12 write_canonical/2

8.14.12.1 Description

write_canonical(S_or_a, Term) is true iff
write_term(S_or_a, Term,
[quoted(true), ignore_ops(true)]).

8.14.12.2 Template and modes

```
write_canonical(@stream_or_alias, @term)
```

8.14.12.3 Errors

a) `S_or_a` is a variable
— `instantiation_error`.

b) `S_or_a` is neither a variable nor a stream identifier or alias
— `domain_error(stream_or_alias, S_or_a)`.

c) `S_or_a` is not associated with an open stream
— `existence_error(stream, S_or_a)`.

d) `S_or_a` is an input stream
— `permission_error(output, stream, S_or_a)`.

8.14.12.4 Examples

```
write_canonical(out, [1,2,3]).
   Succeeds, outputting the characters
'.'(1,'.'(2,'.'(3,[])))
   to the stream associated with the alias 'out'.

write_canonical(out, 1<2).
   Succeeds, outputting the characters
<(1,2)
   to the stream associated with the alias 'out'.

write_canonical(out, '1<2').
   Succeeds, outputting the characters
'1<2'
   to the stream associated with the alias 'out'.

write_canonical(out, '$VAR'(0)<'$VAR'(1)).
   Succeeds, outputting the characters
<('$VAR'(0),'$VAR'(1))
   to the stream associated with the alias 'out'.
```

8.14.13 op/3

This predicate enables the predefined operators (see 6.3.4.4 and table 5) to be altered during execution.

8.14.13.1 Description

`op(Priority, Op_specifier, Operator)` is true.

Procedurally, `op(Priority, Op_specifier, Operator)` is executed as follows:

a) If `Operator` is an atom, creates the set Ops containing just that one atom,

b) Else if `Operator` is a list of atoms, creates the set Ops consisting of all the atoms in the list,

c) Chooses an element `Op` in the set Ops and removes it from the set,

d) If `Op` is not currently an operator with the same operator class (prefix, infix or postfix) as `Op_specifier`, then proceeds to 8.14.13.1 f,

e) The operator property of `Op` with the same class as `Op_specifier` is removed, so that `Op` is no longer an operator of that class,

f) If `Priority=0`, proceeds to 8.14.13.1 h,

g) `Op` is made an operator with specifier `Op_specifier` and priority `Priority`,

h) If Ops is non-empty, proceeds to 8.14.13.1 c,

i) Else, the predicate succeeds.

In the event of an error being detected in an `Operator` list argument, it is undefined which, if any, of the atoms in the list is made an operator before the exception is raised.

NOTES

1 Operator notation is defined in 6.3.4. See also operator directives (7.4.2.4).

2 A `Priority` of zero can be used to remove an operator property from an atom.

3 It does not matter if the same atom appears more than once in an `Operator` list; this is not an error and the duplicates simply have no effect.

4 In general, the predefined operators can be removed, or their priority can be changed. However, it is an error to attempt to change the meaning of the `,` operator from its predefined status, see 6.3.4.3.

8.14.13.2 Template and modes

```
op(@integer, @operator_specifier,
   @atom_or_atom_list)
```

8.14.13.3 Errors

a) `Priority` is a variable
— `instantiation_error`.

b) `Op_specifier` is a variable
— `instantiation_error`.

c) `Operator` is a variable
— `instantiation_error`.

d) `Operator` is a list with an element `E` which is a variable
— `instantiation_error`.

e) `Priority` is neither a variable nor an integer
— `type_error(integer, Priority)`.

f) `Op_specifier` is neither a variable nor an atom
— `type_error(atom, Op_specifier)`.

g) `Operator` is neither a variable nor an atom nor a list
— `type_error(list, Operator)`.

h) An element `E` of the `Operator` list is neither a variable nor an atom
— `type_error(atom, E)`.

i) `Priority` is not between 0 and 1200 inclusive
— `domain_error(operator_priority, Priority)`.

j) `Op_specifier` is not a valid operator specifier
— `domain_error(operator_specifier, Op_specifier).`

k) `Operator` is `','`
— `permission_error(modify, operator, Operator).`

l) An element `E` of the `Operator` list is `','`
— `permission_error(modify, operator, E).`

m) `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).`

8.14.13.4 Examples

```
op(30, xfy, ++).
   Succeeds, making ++ a right associative
      infix operator with priority 30.

op(0, yfx, ++).
   Succeeds, making ++ no longer an
      infix operator.

op(max, xfy, ++).
   type_error(integer, max).

op(-30, xfy, ++).
   domain_error(operator_priority, -30).

op(1201, xfy, ++).
   domain_error(operator_priority, 1201).

op(30, XFY, ++).
   instantiation_error.

op(30, yfy, ++).
   domain_error(operator_specifier, yfy).

op(30, xfy, 0).
   type_error(list, 0).

op(30, xfy, ++), op(40, xfx, ++).
   Succeeds, making ++ a non-associative
      infix operator with priority 40.

op(30, xfy, ++), op(50, yf, ++).
   permission_error(create, operator, ++).
   [There cannot be an infix and a
      postfix operator with the same name.]
```

8.14.14 current_op/3

8.14.14.1 Description

`current_op(Priority, Op_specifier, Operator)` is true iff `Operator` is an operator with properties defined by specifier `Op_specifier` and precedence `Priority`.

Procedurally, `current_op(Priority, Op_specifier, Operator)` is executed as follows:

a) Searches the current operator definitions and creates a set S of all the triples `(P,Spec,Op)` such that there is an operator:

1) whose name, `Op`, unifies with `Operator`,

2) whose specifier, `Spec`, unifies with `Op_specifier`, and

3) whose priority, `P`, unifies with `Priority`,

b) If a non-empty set is found, proceeds to 8.14.14.1 d,

c) Else the predicate fails.

d) Chooses an element of the set S and the predicate succeeds.

e) If all the elements of the set S have been chosen, then the predicate fails,

f) Else chooses an element of the set S which has not already been chosen, and the predicate succeeds.

`current_op(Priority, Op_specifier, Operator)` is re-executable. On backtracking, continue at 8.14.14.1 e.

The order in which operators are found by `current_op/3` is implementation dependent.

When the operator `,` (comma) is a member of the set S it is represented by the atom `','`.

NOTES

1 The definition above implies that if a program calls `current_op/3` and then modifies an operator definition by calling `op/3`, and then backtracks into the call to `current_op/3`, then the changes are guaranteed not to affect that `current_op/3` goal. That is, `current_op/3` behaves as if it were implemented as a dynamic predicate whose clauses are retracted and asserted when `op/3` is called.

2 An operator `Old_op` which has been removed by `op(0, Op_specifier, Old_op)` is not found by `current_op/3`.

8.14.14.2 Template and modes

```
current_op(?integer, ?operator_specifier,
   ?atom)
```

8.14.14.3 Errors

None.

8.14.14.4 Examples

```
current_op(P, xfy, OP).
   If the predefined operators have not been
      altered, then
   Succeeds, unifying P with 1100,
      and OP with ';'.
   On re-execution, succeeds unifying
      P with 1050, and OP with '->'.
   On re-execution, succeeds unifying
      P with 1000, and OP with ','.
   [The order of solutions is
      implementation dependent.]
```

8.14.15 char_conversion/2

8.14.15.1 Description

`char_conversion(Input_char, Output_char)` is true.

Procedurally, `char_conversion(Input_char, Output_char)` is executed as follows:

a) If `Input_char` is equal to `Output_char`, the predicate succeeds.

b) Else update the character-conversion relation with the conversion (`Input_char` → `Output_char`), and the predicate succeeds.

NOTES

1 See also char-conversion directives (7.4.2.5).

2 A character `Input_char` and `Output_char` should be quoted in order to ensure that they have not been converted by a character-conversion directive when the Prolog text is read.

3 The character-conversion relation affects only characters read by term input (8.14). When it is necessary to convert characters read by character input/output built-in predicates (8.12), it will be necessary to program the conversion explicitly using `current_char_conversion/2` (8.14.16).

8.14.15.2 Template and modes

```
char_conversion(@character, @character)
```

8.14.15.3 Errors

a) `Input_char` is a variable
— `instantiation_error`.

b) `Output_char` is a variable
— `instantiation_error`.

c) `Input_char` is neither a variable nor a character
— `type_error(character, Input_char)`.

d) `Output_char` is neither a variable nor a character
— `type_error(character, Output_char)`.

8.14.15.4 Examples

`char_conversion('&', ',')`
Updates the char-conversion relation with (`&` → `','`) .
Succeeds.

`char_conversion('''', '\'')`
Updates the char-conversion relation with (' → `'`) where ' is a character in an extended character set equivalent to the single quote.
Succeeds.

`char_conversion('`**a**`', a)`
Updates the char-conversion relation with (**a** → `a`) where **a** is a character in an extended character set equivalent to the small letter character `a`.
Succeeds.

After these three goals, when the value associated with flag `char_conversion` is on, all occurrences of `&`, ', and **a** as unquoted characters read by term input predicates are converted to `,`, `\`, and `a` respectively, for example the three characters **aaa** are converted to the characters `aaa`. However the characters `'`**aaa**`'` represent an atom **aaa** because they are enclosed by the single quotes.

`char_conversion('`**a**`', 'a')`
Updates the char-conversion relation by removing the conversion (**a** → `a`).
Succeeds.

8.14.16 current_char_conversion/2

8.14.16.1 Description

`current_char_conversion(Input_char, Output_char)` is true iff the character-conversion relation contains the conversion (`Input_char` → `Output_char`).

Procedurally, `current_char_conversion(Input_char, Output_char)` is executed as follows:

a) Creates a set S of all the conversions (`In` → `Out`) in the the current character-conversion relation such that:

1) `In` unifies with `Input_char`, and

2) `Out`, unifies with `Output_char`,

b) If a non-empty set is found, proceeds to 8.14.16.1 d,

c) Else the predicate fails.

d) Chooses an element of the set S which has not already been chosen, unifies `In` with `Input_char`, and `Out` with `Output_char`, and the predicate succeeds.

e) If all the elements of the set S have been chosen, then the predicate fails,

f) Else proceeds to 8.14.16.1 d.

`current_char_conversion(Input_char, Output_char)` is re-executable. On backtracking, continue at 8.14.16.1 e.

The order in which character-conversions are found by `current_char_conversion/2` is implementation dependent.

NOTES

1 The definition above implies that if a program calls `current_char_conversion/2` and then modifies the character-conversion relation by calling `char_conversion/2`, and then backtracks into the call to `current_char_conversion/2`, then the changes are guaranteed not to affect that `current_char_conversion/2` goal.

2 A character-conversion (`C` → `CC`) which has been removed by `char_conversion(C, C)` is not found by `current_char_conversion/2`.

8.14.16.2 Template and modes

```
current_char_conversion(?character,
  ?character)
```

8.14.16.3 Errors

None.

8.14.16.4 Examples

```
current_char_conversion(C, a)
```
Assume the char-conversion relation is
(**a** → a, *a* → a),
Succeeds, unifying C with **a**.
On re-execution, succeeds, unifying C with *a*.

8.15 Logic and control

These predicates are simply derived from the control constructs (7.8) and provide additional facilities for affecting the control flow during resolution.

8.15.1 fail_if/1

8.15.1.1 Description

`fail_if(Term)` is true *iff* `call(Term)` is false.

Procedurally, `fail_if(Term)` is executed as follows:

a) Executes `call(Term)`,

b) If it succeeds, the predicate fails,

c) Else if it fails, the predicate succeeds.

NOTE — A predicate with the same meaning as `fail_if/1` is implemented in many existing processors with a name `not/1` which is misleading because the predicate gives *negation by failure* rather than true negation. Other processors implement this feature with a predicate `\+/1`.

8.15.1.2 Template and modes

```
fail_if(@callable_term)
```

8.15.1.3 Errors

a) `Term` is a variable
— `instantiation_error`.

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

NOTE — Errors produced by the execution of the goal `fail_if(Term)` are regarded as errors in `Term`.

8.15.1.4 Examples

```
fail_if(true).
   Fails.

fail_if(!).
   Fails, the cut has no effect.

fail_if((!, fail)).
   Succeeds, the cut has no effect.

(X=1; X=2), fail_if((!, fail)).
   Succeeds, unifying X with 1.
   On re-execution, succeeds unifying X with 2.

fail_if(4 = 5).
   Succeeds.

fail_if(3).
   type_error(callable, 3).

fail_if(X).
   instantiation_error.

fail_if(X = f(X)).
   Undefined.
```

8.15.2 once/1

8.15.2.1 Description

`once(Term)` is true *iff* `call(Term)` is true.

Procedurally, `once(Term)` is executed as follows:

a) Executes `call(Term)`,

b) If it succeeds, the predicate succeeds,

c) Else if it fails, the predicate fails.

NOTE — `once(Term)` behaves as `call(Goal)`, but is not re-executable.

8.15.2.2 Template and modes

```
once(+callable_term)
```

8.15.2.3 Errors

a) `Term` is a variable
— `instantiation_error`.

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

NOTE — Errors produced by the execution of the goal `once(Term)` are regarded as errors in `Term`.

8.15.2.4 Examples

```
once(!).
   Succeeds (the same as true).

once(!), (X=1; X=2).
   Succeeds, unifying X with 1.
   On re-execution, succeeds unifying X with 2.

once(repeat).
   Succeeds (the same as true).

once(fail).
   Fails.

once(X = f(X)).
   Undefined.
```

8.15.3 repeat/0

8.15.3.1 Description

`repeat` is true.

Procedurally, `repeat` succeeds.

`repeat` is re-executable.

8.15.3.2 Template and modes

```
repeat
```

8.15.3.3 Errors

None.

8.15.3.4 Examples

```
repeat, write("hello "), fail.
   Writes
hello hello hello hello hello ...
   indefinitely.

repeat, !, fail.
   Fails, equivalent to (!, fail).
```

8.16 Constant processing

These predicates enable constants to be processed as a sequence of characters (7.1.4.1) and character codes (7.1.2.2). Facilities exist to split and join atoms, and to convert a single character to and from the corresponding character code, and to convert a number to and from a list of characters.

NOTES

1 The characters forming an atom are defined in 6.1.2 b.

2 These predicates assume the characters of an atom can be numbered: clause 6.1.2 b defines that the characters of a non-empty atom are numbered from one upwards.

8.16.1 atom_length/2

8.16.1.1 Description

`atom_length(Atom, Length)` is true iff integer `Length` equals the number of characters in the atom `Atom`.

Procedurally, `atom_length(Atom, Length)` unifies `Length` with the number of characters in `Atom`.

8.16.1.2 Template and modes

```
atom_length(+atom, ?integer)
```

8.16.1.3 Errors

a) `Atom` is a variable
— `instantiation_error`.

b) `Atom` is neither a variable nor an atom
— `type_error(atom, Atom)`.

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

8.16.1.4 Examples

```
atom_length('enchanted evening', N).
   Succeeds, unifying N with 17.

atom_length('enchanted\
 evening', N).
   Succeeds, unifying N with 17.

atom_length('', N).
   Succeeds, unifying N with 0.

atom_length('scarlet', 5).
   Fails.

atom_length(Atom, 4).
   instantiation_error.

atom_length(1.23, 4).
   type_error(atom, 1.23).

atom_length(atom, '4').
   type_error(integer, '4').
```

8.16.2 atom_concat/3

8.16.2.1 Description

`atom_concat(Atom_1, Atom_2, Atom_12)` is true iff the atom `Atom_12` is the atom formed by concatenating the characters of the atom `Atom_2` to the characters of the atom `Atom_1`.

Procedurally, `atom_concat(Atom_1, Atom_2, Atom_12)` unifies `Atom_12` with the concatenation of `Atom_1` and `Atom_2`.

`atom_concat(Atom_1, Atom_2, Atom_12)` is re-executable when only `Atom_12` is instantiated. On re-execution successive values for `Atom_1` and `Atom_2` are generated.

8.16.2.2 Template and modes

```
atom_concat(?atom, ?atom, +atom)
atom_concat(+atom, +atom, -atom)
```

8.16.2.3 Errors

a) `Atom_1` and `Atom_12` are variables
— `instantiation_error`.

b) `Atom_2` and `Atom_12` are variables
— `instantiation_error`.

c) `Atom_1` is neither a variable nor an atom
— `type_error(atom, Atom_1)`.

d) Atom_2 is neither a variable nor an atom
— type_error(atom, Atom_2).

e) Atom_12 is neither a variable nor an atom
— type_error(atom, Atom_12).

8.16.2.4 Examples

In the examples below,

```
S1 = 'hello',
S2 = ' world',
S4 = 'small world'.

atom_concat(S1, S2, S3).
   Succeeds, unifying S3 with 'hello world'.

atom_concat(T, S2, S4).
   Succeeds, unifying T with 'small'.

atom_concat(S1, S2, S4).
   Fails.

atom_concat(T1, T2, S1).
   Succeeds, unifying T1 with '',
      and T2 with 'hello'.
   On re-execution, succeeds,
       unifying T1 with 'h', and T2 with 'ello'.

atom_concat(small, S2, S4).
   instantiation_error.
```

8.16.3 sub_atom/4

8.16.3.1 Description

sub_atom(Atom, Start, Length, Sub_atom) is true iff atom Sub_atom is the atom with Length characters starting at the Start-th character of atom Atom.

Procedurally, sub_atom(Atom, Start, Length, Sub_atom) unifies Sub_atom with an atom Atom which has Length characters identical with the Length characters of atom Atom that start with the Start-th character of Atom.

sub_atom(Atom, Start, Length, Sub_atom) is re-executable. On re-execution all possible values for Start, Length and Sub_atom are generated.

8.16.3.2 Template and modes

```
sub_atom(+atom, ?integer, ?integer, ?atom)
```

8.16.3.3 Errors

a) Atom is a variable
— instantiation_error.

b) Atom is neither a variable nor an atom
— type_error(atom, Atom).

c) Sub_atom is neither a variable nor an atom
— type_error(atom, Sub_atom).

d) Start is neither a variable nor an integer
— type_error(integer, Start).

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

8.16.3.4 Examples

```
sub_atom('Banana', 4, 2, S2).
   Succeeds, unifying S2 with 'an'.

sub_atom('charity', _, 3, S2).
   Succeeds, unifying S2 with 'cha'.
   On re-execution, succeeds,
      unifying S2 with 'har'.
   On re-execution, succeeds,
      unifying S2 with 'ari'.
   On re-execution, succeeds,
      unifying S2 with 'rit'.
   On re-execution, succeeds,
      unifying S2 with 'ity'.

sub_atom('ab', Start, Length, Sub_atom).
   Succeeds, unifying Start with 1,
      and Length with 0, and Sub_atom with ''.
   On re-execution, succeeds,
      unifying Start with 1, and Length with 1,
      and Sub_atom with 'a'.
   On re-execution, succeeds,
      unifying Start with 1, and Length with 2,
      and Sub_atom with 'ab'.
   On re-execution, succeeds,
      unifying Start with 2, and Length with 0,
      and Sub_atom with ''.
   On re-execution, succeeds,
      unifying Start with 2, and Length with 1,
      and Sub_atom with 'b'.
   On re-execution, succeeds,
      unifying Start with 3, and Length with 0,
      and Sub_atom with ''.
```

8.16.4 atom_chars/2

8.16.4.1 Description

atom_chars(Atom, List) is true iff List is a list whose elements are the characters corresponding to the successive characters of atom Atom.

Procedurally, atom_chars(Atom, List) is executed as follows:

a) If Atom is an atom then List is unified with a list of characters which shall be identical to the sequence of characters which form the abstract syntax of Atom (see 6.1.2 b),

b) Else if List is a list of characters, then Atom is unified with the atom whose abstract syntax has the same sequence of characters,

c) Else the predicate fails.

8.16.4.2 Template and modes

```
atom_chars(+atom, +list)
atom_chars(+atom, -list)
atom_chars(-atom, +list)
```

8.16.4.3 Errors

a) `Atom` and `List` are variables
— `instantiation_error`.

b) `Atom` is neither a variable nor an atom
— `type_error(atom, Atom)`.

c) `List` is neither a variable nor a list nor a partial list
— `type_error(list, List)`.

8.16.4.4 Examples

```
atom_chars('', L).
   Succeeds, unifying L with [].

atom_chars([], L).
   Succeeds, unifying L with ['[', ']'].

atom_chars('''', L).
   Succeeds, unifying L with [''''].

atom_chars('ant', L).
   Succeeds, unifying L with
      ['a', 'n', 't'].

atom_chars(Str, ['s', 'o', 'p']).
   Succeeds, unifying Str with 'sop'.

atom_chars('North', ['N' | X]).
   Succeeds, unifying X with
      ['o', 'r', 't', 'h'].

atom_chars('soap', ['s', 'o', 'p']).
   Fails.

atom_chars(X, Y).
   instantiation_error.
```

8.16.5 atom_codes/2

8.16.5.1 Description

`atom_codes(Atom, List)` is true iff `List` is a list whose elements correspond to the successive characters of atom `Atom`, and the value of each element is the character code for the corresponding character.

Procedurally, `atom_codes(Atom, List)` is executed as follows:

a) If `Atom` is an atom then `List` is unified with a list of character codes (7.1.2.2) corresponding to a sequence of characters which shall be identical to the sequence of characters which form the abstract syntax of `Atom` (see 6.1.2 b),

b) Else if `List` is a list of character codes, then `Atom` is unified with the atom whose abstract syntax has the sequence of characters corresponding to the same list of character codes,

c) Else the predicate fails.

8.16.5.2 Template and modes

```
atom_codes(+atom, +list)
atom_codes(+atom, -list)
atom_codes(-atom, +list)
```

8.16.5.3 Errors

a) `Atom` and `List` are variables
— `instantiation_error`.

b) `Atom` is neither a variable nor an atom
— `type_error(atom, Atom)`.

c) `List` is neither a variable nor a list nor a partial list
— `type_error(list, List)`.

8.16.5.4 Examples

```
atom_codes('', L).
   Succeeds, unifying L with [].

atom_codes([], L).
   Succeeds, unifying L with [0'[, 0']].

atom_codes('''', L).
   Succeeds, unifying L with [0''''].

atom_codes('ant', L).
   Succeeds, unifying L with
   [0'a, 0'n, 0't].

atom_codes(Str, [0's, 0'o, 0'p]).
   Succeeds, unifying Str with 'sop'.

atom_codes('North', [0'N | X]).
   Succeeds, unifying X with
   [0'o, 0'r, 0't, 0'h].

atom_codes('soap', [0's, 0'o, 0'p]).
   Fails.

atom_codes(X, Y).
   instantiation_error.
```

8.16.6 char_code/2

8.16.6.1 Description

`char_code(Char, Code)` is true iff the character code (7.1.2.2) for the character `Char` is `Code`.

Procedurally, `char_code(Char, Code)` unifies `Code` with character code for the character `Char`.

8.16.6.2 Template and modes

```
char_code(+character, +character_code)
char_code(+character, -character_code)
char_code(-character, +character_code)
```

8.16.6.3 Errors

a) `Char` and `Code` are variables
— `instantiation_error`.

b) `Char` is neither a variable nor a character (7.1.4.1)
— `representation_error(character)`.

c) `Code` is neither a variable nor a character code (7.1.2.2)
— `representation_error(character_code)`.

8.16.6.4 Examples

```
char_code('a', Code).
   Succeeds, unifying Code with the
      character code for the character 'a'.

char_code(Str, 99).
   Succeeds, unifying Str with the character
      whose character code is 99.

char_code(Str, 0'c).
   Succeeds, unifying Str with the character 'c'.

char_code(Str, 163).
   If there is an extended character whose
         character code is 163 then
      Succeeds, unifying Str with that
         extended character,
   else
      representation_error(character_code).

char_code('b', 84).
   Succeeds iff the character 'b' has the
      character code 84.

char_code('ab', Int).
   type_error(character, ab).

char_code(C, I).
   instantiation_error.
```

8.16.7 number_chars/2

8.16.7.1 Description

`number_chars(Number, List)` is true iff `List` is a list whose elements are the characters corresponding to a character sequence of `Number` which could be output (7.10.6 b, 7.10.6 c).

Procedurally, `number_chars(Number, List)` is executed as follows:

a) If `List` is a list of characters, then that sequence of characters is parsed according to the syntax rules for numbers and negative numbers (6.3.1.1, 6.3.1.2). If the parse is successful, `Number` is unified with the resulting value, else the predicate fails.

b) Else if `Number` is an integer or float, then `List` is unified with a list of characters which shall be identical to the sequence of characters which would be output by `write_canonical(Number)` (see 7.10.6 b, 7.10.6 c, 8.14.11),

c) Else the predicate fails.

NOTES

1 The sequence of characters ensures that, for every number `X`, the following goal is true:
`number_chars(X,C), number_chars(Y,C), X == Y.`

2 This definition ensures that, the following goal is true:

```
C=['.', '1'],
   number_chars(X,C), number_chars(X,C).
```

8.16.7.2 Template and modes

```
number_chars(+number, +list)
number_chars(+number, -list)
number_chars(-number, +list)
```

8.16.7.3 Errors

a) `Number` and `List` are variables
— `instantiation_error`.

b) `Number` is neither a variable nor a number
— `type_error(number, Number)`.

c) `List` is neither a variable nor a list of characters
— `domain_error(character_list, List)`.

d) `List` is not parsable as a number
— `syntax_error`.

8.16.7.4 Examples

```
number_chars(33, L).
   Succeeds, unifying L with ['3', '3'].

number_chars(33, ['3', '3']).
   Succeeds.

number_chars(33.0, L).
   Succeeds, unifying L with an
   implementation dependent list of characters,
   e.g. ['3', '.', '3', 'E', +, '0', '1'].

number_chars(X,
      ['3', '.', '3', 'E', +, '0']).
   Succeeds, unifying X with a value
      approximately equal to 3.3.

number_chars(3.3,
      ['3', '.', '3', 'E', +, '0']).
   Implementation dependent: may succeed or fail.

number_chars(A, [-, '2', '5']).
   Succeeds, unifying A with -25.

number_chars(A, ['\n', ' ', '3']).
   [The new line and space characters are
      not significant.]
   Succeeds, unifying A with 3.

number_chars(A, ['3', ' ']).
   Fails.

number_chars(A, ['0', x, f])
   Succeeds, unifying A with 15.

number_chars(A, ['0', ''', a])
   Succeeds, unifying A with the
   collating sequence integer for the
   character 'a'.

number_chars(A, ['4', '.', '2']).
   Succeeds, unifying A with 4.2.

number_chars(A,
      ['4', '2', '.', '0', 'e', '-', '1']).
   Succeeds, unifying A with 4.2.
```

8.16.8 number_codes/2

8.16.8.1 Description

number_codes(Number, List) is true iff List is a list whose elements are the character codes corresponding to a character sequence of Number which could be output (7.10.6 b, 7.10.6 c).

Procedurally, number_codes(Number, List) is executed as follows:

a) If List is a list of character codes, then the sequence of characters corresponding to those character codes is parsed according to the syntax rules for numbers and negative numbers (6.3.1.1, 6.3.1.2). If the parse is successful, Number is unified with the resulting value, else the predicate fails.

b) Else if Number is an integer or float, then List is unified with a list of character codes corresponding to a sequence of characters which shall be identical to the sequence of characters which would be output by write_canonical(Number) (see 7.10.6 b, 7.10.6 c, 8.14.11),

c) Else the predicate fails.

NOTE — The sequence of character codes representing the characters of a number shall be such that for every value X, the following goal is true:

```
number_codes(X, C), number_codes(Y, C), X==Y.
```

8.16.8.2 Template and modes

```
number_codes(+number, +list)
number_codes(+number, -list)
number_codes(-number, +list)
```

8.16.8.3 Errors

a) Number and List are variables
— instantiation_error.

b) Number is neither a variable nor a number
— type_error(number, Number).

c) List is neither a variable nor a list of character codes
— domain_error(character_code_list, List).

d) List is not parsable as a number
— syntax_error.

8.16.8.4 Examples

```
number_codes(33, L).
   Succeeds, unifying L with [0'3, 0'3].

number_codes(33, [0'3, 0'3]).
   Succeeds.

number_codes(33.0, L).
   Succeeds, unifying L with an
   implementation dependent list of characters,
   e.g. [0'3, 0'., 0'3, 0'E, 0'+, 0'0, 0'1].

number_codes(33.0,
      [0'3, 0'., 0'3, 0'E, 0'+, 0'0, 0'1]).
   Implementation dependent: may succeed or fail.

number_codes(A, [0'-, 0'2, 0'5]).
   Succeeds, unifying A with -25.

number_codes(A, [0' , 0'3]).
   [The space character is not significant.]
   Succeeds, unifying A with 3.

number_codes(A, [0'0, 0'x, 0'f])
   Succeeds, unifying A with 15.

number_codes(A, [0'0, 0''', 0'a])
   Succeeds, unifying A with the
      collating sequence integer for the
      character 'a'.

number_codes(A, [0'4, 0'., 0'2]).
   Succeeds, unifying A with 4.2.

number_codes(A,
      [0'4, 0'2, 0'., 0'0, 0'e, 0'-, 0'1]).
   Succeeds, unifying A with 4.2.
```

8.17 Implementation defined hooks

These built-in predicates enable a program to find the current value of any flag (7.11), and to change the current value of some flags.

8.17.1 set_prolog_flag/2

8.17.1.1 Description

set_prolog_flag(Flag, Value) is true *iff*:

a) Flag is a flag, and

b) Value is a value that is within the implementation defined range of values for Flag.

Procedurally, set_prolog_flag(Flag, Value) is executed as follows:

a) If Flag is a flag (7.11), and Value is a value that is within the implementation defined range of values for Flag, proceeds to 8.17.1.1 c,

b) Else the predicate fails.

c) Associates the value Value with the flag Flag, and the predicate succeeds.

8.17.1.2 Template and modes

```
set_prolog_flag(@flag, @term)
```

8.17.1.3 Errors

a) Flag is a variable
— instantiation_error.

b) Value is a variable
— instantiation_error.

c) Flag is neither a variable nor an atom
— type_error(atom, Flag).

d) `Flag` is an atom but is invalid in the processor
— `domain_error(prolog_flag, Flag).`

e) `Value` is inappropriate for `Flag`
— `domain_error(flag_value, Flag + Value).`

8.17.1.4 Examples

```
set_prolog_flag(undefined_predicate, fail).
   Succeeds, associating the value fail
   with flag undefined_predicate.

set_prolog_flag(X, off).
   instantiation_error.

set_prolog_flag(5, decimals).
   type_error(atom, 5).

set_prolog_flag(date, 'July 1988').
   domain_error(flag, date).

set_prolog_flag(debug, trace).
   domain_error(flag_value, debug+trace).
```

8.17.2 current_prolog_flag/2

8.17.2.1 Description

`current_prolog_flag(Flag, Value)` is true iff `Flag` is a flag supported by the processor, and `Value` is the value currently associated with it.

Procedurally, `current_prolog_flag(Flag, Value)` is executed as follows:

a) Searches the current flags supported by the processor and creates a set S of all the terms `flag(F, V)` such that (1) there is a flag `F` which unifies with `Flag`, and (2) the value `V` currently associated with `F` unifies with `Value`,

b) If a non-empty set is found, proceeds to 8.17.2.1 d,

c) Else the predicate fails.

d) Chooses an element of the set S and the predicate succeeds.

e) If all the elements of the set S have been chosen, then the predicate fails,

f) Else chooses an element of the set S which has not already been chosen, and the predicate succeeds.

`current_prolog_flag(Flag, Value)` is re-executable. On re-execution, continue at 8.17.2.1 e above.

The order in which flags are found by `current_prolog_flag(Flag, Value)` is implementation dependent.

NOTE — All flags are found, whether defined by this draft International Standard or implementation defined.

8.17.2.2 Template and modes

```
current_prolog_flag(?flag, ?term)
```

8.17.2.3 Errors

a) `Flag` is not a variable or an atom
— `type_error(atom, Flag).`

8.17.2.4 Examples

```
current_prolog_flag(debug, off).
   Succeeds iff the value currently associated
   with the flag 'debug' is 'off'.

current_prolog_flag(F, V).
   Succeeds, unifying 'F' with one of the
   flags supported by the processor, and 'V'
   with the value currently associated with
   the flag 'F'.
   On re-execution, successively unifies 'F'
   and 'V' with each other flag supported by
   the processor and its associated value.

current_prolog_flag(5, _).
   type_error(atom, 5).
```

8.17.3 halt/0

8.17.3.1 Description

Procedurally, `halt` is executed as follows:

a) Exits from the processor,

b) Returns to whatever system invoked Prolog.

Any other effect of `halt/0` is implementation defined.

NOTE — This predicate neither succeeds nor fails.

8.17.3.2 Template and modes

```
halt
```

8.17.3.3 Errors

None.

8.17.3.4 Examples

```
halt.
   Implementation defined.
```

8.17.4 halt/1

8.17.4.1 Description

Procedurally, `halt(X)` is executed as follows:

a) Exits from the processor,

b) Returns to whatever system invoked Prolog passing the value of `X` as a message.

Any other effect of `halt/1` is implementation defined.

NOTE — This predicate neither succeeds nor fails.

8.17.4.2 Template and modes

```
halt(@integer)
```

8.17.4.3 Errors

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

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

8.17.4.4 Examples

```
halt(1).
   Implementation defined.

halt(a).
   type_error(integer, a).
```

---

9 Evaluable functors

This clause defines the evaluable functors which shall be implemented by a standard-conforming Prolog processor.

9.1 The simple arithmetic functors

The basic arithmetic functions are defined mathematically in the style, and conforming with the requirements, of IS10967-1.

9.1.1 Evaluable functors and operations

Each evaluable functor corresponds to one or more operations according to the types of the values which are obtained by evaluating the argument(s) of the functor.

The following table identifies the integer or floating point operations corresponding to each functor:

Evaluable functor	Operation
`'+'/2`	add_I, add_F, add_{FI}, add_{IF}
`'-'/2`	sub_I, sub_F, sub_{FI}, sub_{IF}
`'*'/2`	mul_I, mul_F, mul_{FI}, mul_{IF}
`'//'/2`	$intdiv_I$
`'/'/2`	div_F, div_{II}, div_{FI}, div_{IF}
`rem/2`	rem_I
`mod/2`	mod_I
`'-'/1`	neg_I, neg_F
`abs/1`	abs_I, abs_F
`sqrt/1`	$sqrt_I$, $sqrt_F$
`sign/1`	$sign_I$, $sign_F$
`float_truncate/2`	$trunc_F$
`float_round/2`	$round_F$
`float_integer_part/1`	$intpart_F$
`float_fractional_part/1`	$fractpart_F$
`float/1`	$float_{I\rightarrow F}$, $float_{F\rightarrow F}$
`floor/1`	$floor_{F\rightarrow I}$
`truncate/1`	$truncate_{F\rightarrow I}$
`round/1`	$round_{F\rightarrow I}$
`ceiling/1`	$ceiling_{F\rightarrow I}$

NOTE — `'+'`, `'-'`, `'*'`, `'//'`, `'/'`, `'rem'`, `'mod'` are infix predefined operators (see 6.3.4.4).

9.1.2 Integer operations and axioms

The following operations are specified:

$$
\begin{aligned}
&add_I : I \times I \rightarrow I \cup \{\textbf{overflow}\}\\
&sub_I : I \times I \rightarrow I \cup \{\textbf{overflow}\}\\
&mul_I : I \times I \rightarrow I \cup \{\textbf{overflow}\}\\
&intdiv_I : I \times I \rightarrow I \cup \{\textbf{overflow}, \textbf{zero\_divisor}\}\\
&rem_I : I \times I \rightarrow I \cup \{\textbf{zero\_divisor}\}\\
&mod_I : I \times I \rightarrow I \cup \{\textbf{zero\_divisor}, \textbf{undefined}\}\\
&neg_I : I \rightarrow I \cup \{\textbf{overflow}\}\\
&abs_I : I \rightarrow I \cup \{\textbf{overflow}\}\\
&sign_I : I \rightarrow I
\end{aligned}
$$

The behaviour of the integer operations are defined in terms of a rounding function $rnd_I(x)$ (see 9.1.2.1).

For all values x and y in I, the following axioms shall apply:

$$
\begin{aligned}
add_I(x,y) &= x + y \quad \text{if } x + y \in I\\
&= \textbf{overflow} \quad \text{if } x + y \notin I\\
\\
sub_I(x,y) &= x - y \quad \text{if } x - y \in I\\
&= \textbf{overflow} \quad \text{if } x - y \notin I\\
\\
mul_I(x,y) &= x * y \quad \text{if } x * y \in I\\
&= \textbf{overflow} \quad \text{if } x * y \notin I\\
\\
intdiv_I(x,y) &= rnd_I(x/y)\\
&\quad \text{if } y \neq 0 \text{ and } rnd_I(x/y) \in I\\
&= \textbf{overflow}\\
&\quad \text{if } y \neq 0 \text{ and } rnd_I(x/y) \notin I\\
&= \textbf{zero\_divisor}\\
&\quad \text{if } y = 0\\
\\
rem_I(x,y) &= x - (rnd_I(x/y) * y)\\
&\quad \text{if } y \neq 0\\
&= \textbf{zero\_divisor}\\
&\quad \text{if } y = 0\\
\\
mod_I(x,y) &= x - (\lfloor x/y \rfloor * y)\\
&\quad \text{if } y \neq 0\\
&= \textbf{zero\_divisor}\\
&\quad \text{if } y = 0\\
\\
neg_I(x) &= -x \quad \text{if } -x \in I\\
&= \textbf{overflow} \quad \text{if } -x \notin I\\
\\
abs_I(x) &= |x| \quad \text{if } |x| \in I\\
&= \textbf{overflow} \quad \text{if } |x| \notin I\\
sign_I(x) &= 1\\
\quad \text{if } x \geq 0\\
&= -1\\
\quad \text{if } x < 0
\end{aligned}
$$

NOTE — IS 10967 (LIA) permits mod_I to have one or both definitions of mod_I^1 and mod_I^2: