

ELAN

LIBRARY REFERENCE MANUAL

January 21, 2000
ELAN V3.3

Peter Borovanský
Horatiu Cirstea
Hubert Dubois
Claude Kirchner
Hélène Kirchner
Pierre-Etienne Moreau
Christophe Ringeissen
Marian Vittek

Authors: Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, H  l  ne Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, Marian Vittek

LORIA: CNRS, INPL, INRIA, UHP, Universit   de Nancy 2

Campus scientifique

615, rue du Jardin Botanique

BP101

54602 Villers-l  s-Nancy CEDEX

FRANCE

E-mail: elan@loria.fr

Copyright   1996, 1997, 1998 Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, H  l  ne Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen and Marian Vittek.

Permission is granted to make and distribute verbatim copies of this book provided the copyright notice and this permission are preserved on all copies.

Contents

1	Library common	4
1.1	Query.eln	5
1.2	anyIdentifier.eln	8
1.3	anyInteger.eln	9
1.4	array.eln	11
1.5	arrayLIN.eln	13
1.6	arrayLOG.eln	15
1.7	basio.eln	17
1.8	bool.eln	19
1.9	builtinInt.eln	21
1.10	builtinStdio.eln	24
1.11	builtinString.eln	27
1.12	builtinSyntacticMatching.eln	29
1.13	cmp.eln	30
1.14	eq.eln	31
1.15	identity.eln	32
1.16	int.eln	33
1.17	integerConstants.eln	36
1.18	io.eln	37
1.19	list.eln	38
1.20	occur.eln	41
1.21	pair.eln	42
1.22	prompt.eln	44
1.23	replace.eln	45
1.24	stdio.eln	46
1.25	string.eln	47
1.26	strlist.eln	49
1.27	tuple.eln	51
2	Library noquote	52
2.1	applySubstOn.eln	53
2.2	atom.eln	55
2.3	atomP.eln	56
2.4	eqSystem.eln	58
2.5	hornClauseSyntax.eln	61
2.6	ident.eln	62
2.7	identifier.eln	63
2.8	sigSyntax.eln	64
2.9	substitution.eln	65
2.10	syntacticUnification.eln	66
2.11	termCommons.eln	71
3	Library strategy	77
3.1	Any.eln	78
3.2	Meta_apply.eln	80
3.3	Meta_capply.eln	82
3.4	Meta_strat.eln	83
3.5	any.eln	86
3.6	booleg.eln	87
3.7	commit.eln	89
3.8	loops.eln	90
3.9	meta.eln	92
3.10	strapply.eln	93
3.11	strat.eln	94
3.12	strcapply.eln	97

3.13	strconc.eln	98
3.14	strsig.eln	99
3.15	symbol.eln	102
3.16	tcstrat.eln	103
4	Library ref	105
4.1	Meta_Apply.eln	106
4.2	REF.eln	110
4.3	ref2string.eln	115
4.4	ref2term.eln	117
4.5	ref_idents.eln	119
4.6	ref_modules.eln	120
4.7	ref_read.eln	121
4.8	ref_rules.eln	125
4.9	ref_sorts.eln	129
4.10	ref_strategies.eln	130
4.11	ref_tables.eln	133
4.12	ref_terms.eln	136
4.13	ref_unify.eln	141
4.14	ref_unify_builtin.eln	146
4.15	ref_write.eln	148
4.16	refstring2string.eln	152
4.17	string2refterm.eln	154
5	Index	156

1 Library common

1.1 Query.eln

This module introduces the necessary built-in operators for dealing with queries. All queries of the top-level command language are simple ELAN terms, which are interpreted by a command-language interpreter. For this reason, all command operators are built-in constructors. Whenever the user calls the command-language interpreter, i.e. (s)he invokes ELAN with the option "-C", the module `Query[I,O,P]` is automatically imported at the top-level of the .lgi file. Parameters I,O,P stand for sorts of input query (i.e. terms typed by the user), results (i.e. the sort of returned terms) and print-outs (i.e. sort of a printed form of results). Relations between these sorts are represented by a query pattern and a printing pattern in the following sense:

- *query pattern* says how to construct a term, which is reduced by ELAN, from the user's input query, i.e. its profile is the following: query-pattern: (I) O;
- *printing pattern* says how to construct printed term form any result produced by ELAN, i.e. its profile is: printing-pattern: (O) P;

Moreover, there is a "check with pattern", which verifies whether an input query satisfies a boolean condition, i.e. "check-with-pattern": (I) Bool. For more details, see the Chapter "Top-level Command Language" in the ELAN user manual.

Consult also the documentation of built-ins in the ELAN manual.

```
5a <the Query speci cation 5a>≡ (7e)
  module Query[I,O,P]
    <imports for Query 5b>
    <sorts for Query 5c>
    <operators for Query 5d>
  end
```

```
5b <imports for Query 5b>≡ (5a)
  import global bool builtinInt anyIdentifier;
  end
```

Uses `builtinInt` 23b 35 46f 101.

The sorts of interest here are `Query[I,O,P]`, the sort of queries where I,O,P are the sorts of input query, results and print-outs. `ide` the sort of identifiers, used only locally. `ids` denoting list of identifiers, also used only locally (i.e. in this module).

```
5c <sorts for Query 5c>≡ (5a)
  sort Query[I,O,P]
    ide
    ids;
  end
```

Let us now define the operators, mostly built-in. We first define the constructors for the local identifiers and list of identifiers (`ide` and `ids`):

```
5d <operators for Query 5d>≡ (5a) 5e>
  operators global
    @ : (ident) ide;
    @ : (ide) ids;
    @,@ : (ide ids) ids;
```

A module name, with its arguments that should be a list of identifiers, is defined by:

```
5e <operators for Query 5d>+≡ (5a) <5d 5f>
  @[@] : (ident ids) ide;
```

Uses `@[@]` 101 and `[@]` 35.

`quit` defines the operator that allows to quit the command interpreter:

```
5f <operators for Query 5d>+≡ (5a) <5e 6a>
  quit : Query[I,O,P] code 90;
```

`help` prints the help menu i.e. the contents of the file `elanlib/help.txt`:

6a `<operators for Query 5d>+≡` (5a) <5f 6b>

`help` : Query[I,0,P] code 112;

`load f` loads the file `f.eln` and extends the current specification:

6b `<operators for Query 5d>+≡` (5a) <6a 6c>

`load @` : (ide) Query[I,0,P] code 91;

`batch f` runs commands from the batch file `f.eln`:

6c `<operators for Query 5d>+≡` (5a) <6b 6d>

`batch @` : (ide) Query[I,0,P] code 107;

`run q` runs the current specification with the query `q`:

6d `<operators for Query 5d>+≡` (5a) <6c 6e>

`run @` : (I) Query[I,0,P] code 94;

`sorts q r s` redefines sorts of query, results and print-outs

6e `<operators for Query 5d>+≡` (5a) <6d 6f>

`sorts @ @ @` : (ide ide ide) Query[I,0,P] code 92;

`startwith(s)t` redefines the starting term (s)t of the `.lgi` file, where `s` is a strategy and `t` is the admissible pattern for the input query:

6f `<operators for Query 5d>+≡` (5a) <6e 6g>

`startwith (@)@` : (ident 0) Query[I,0,P] code 93;

`checkwith b` redefines the checking term of the `.lgi` file, where `b` is a pattern of a boolean condition tested for any input query:

6g `<operators for Query 5d>+≡` (5a) <6f 6h>

`checkwith @` : (bool) Query[I,0,P] code 102;

`printwith t` redefines the printing term used to pretty-print any result:

6h `<operators for Query 5d>+≡` (5a) <6g 6i>

`printwith @` : (P) Query[I,0,P] code 98;

`qs` dumps a stack of queries:

6i `<operators for Query 5d>+≡` (5a) <6h 6j>

`qs` : Query[I,0,P] code 100;

`rs` dumps a stack of results:

6j `<operators for Query 5d>+≡` (5a) <6i 6k>

`rs` : Query[I,0,P] code 101;

`stat` prints statistics about the last run:

6k `<operators for Query 5d>+≡` (5a) <6j 6l>

`stat` : Query[I,0,P] code 96;

`dump` dumps all rules:

6l `<operators for Query 5d>+≡` (5a) <6k 6m>

`dump` : Query[I,0,P] code 95;

`dump l` dumps rules or strategies with the label or name `t`:

6m `<operators for Query 5d>+≡` (5a) <6l 6n>

`dump @` : (ident) Query[I,0,P] code 108;

`dump s` all information about the symbol `s` given by its code:

6n `<operators for Query 5d>+≡` (5a) <6m 6o>

`dump @` : (builtinInt) Query[I,0,P] code 109;

Uses `builtinInt 23b 35 46f 101`.

`display m` changes the printing mode of symbols; `display 1` shows terms in their internal form, while `display 0` uses the traditional representation:

6o `<operators for Query 5d>+≡` (5a) <6n 7a>

`display @` : (builtinInt) Query[I,0,P] code 103;

Uses `builtinInt 23b 35 46f 101`.

trace *n* changes the level of debugging to level *n*:

7a *<operators for Query 5d>+≡* (5a) <6o 7b>

```
    trace @      : (builtinInt)  Query [I,0,P]    code 97;
```

Uses `builtinInt` 23b 35 46f 101.

break *f* sets a break-point on rules or strategies identified by *f* or on the symbol *s* given by its code:

7b *<operators for Query 5d>+≡* (5a) <7a 7c>

```
    break @      : (ident)       Query [I,0,P]    code 104;
```

```
    break @      : (builtinInt)  Query [I,0,P]    code 110;
```

Uses `builtinInt` 23b 35 46f 101.

unbreak *f* deletes the break-point given either by an identifier *f* or by an operator code *s*:

7c *<operators for Query 5d>+≡* (5a) <7b 7d>

```
    unbreak @    : (ident)       Query [I,0,P]    code 105;
```

```
    unbreak @    : (builtinInt)  Query [I,0,P]    code 111;
```

Uses `builtinInt` 23b 35 46f 101.

breaks lists all active break points:

7d *<operators for Query 5d>+≡* (5a) <7c

```
    breaks      :                Query [I,0,P]    code 106;
```

```
end
```

7e *<* 7e>≡*

```
// This file is generated automatically: do not edit it directly.
```

```
<the Query specification 5a>
```


1.2 anyIdentifier.eln

This module is built-in. It introduces the sort `ident` that contains all possible finite identifiers.

8a \langle the anyIdentifier specification 8a $\rangle \equiv$ (8b)

```

module anyIdentifier
/*
    This module is      ***built-in***
    What follows are only ***comments***
    for explaining its semantics

sort ident;
op
global
a      : ident ;
b      : ident ;
aa     : ident ;
ab     : ident ;
aab    : ident ;
... ..
end
*/
end

```

Uses - 23b 35.

8b \langle * 8b $\rangle \equiv$
 // This file is generated automatically: do not edit it directly.
 \langle the anyIdentifier specification 8a \rangle

1.3 anyInteger.eln

This module is built-in. It introduces the sort `int` that contains all integers.

9a \langle the anyInteger specification 9a $\rangle \equiv$ (10b)

```

module anyInteger
/*
    This module is      ***built-in***
    What follows are only ***comments***
    for explaining its semantics

sort int;
op
global

0      : int ;
1      : int ;
-1     : int ;
2      : int ;
-2     : int ;
... ..
end
*/
end

```

Uses - 23b 35 and int 35 46f 101.

9b \langle XX 9b $\rangle \equiv$

```

module anyInteger
   $\langle$ imports for anyInteger 9c $\rangle$ 
   $\langle$ sorts for anyInteger 9d $\rangle$ 
   $\langle$ operators for anyInteger 9e $\rangle$ 
   $\langle$ strategy operator for anyInteger (never defined) $\rangle$ 
   $\langle$ rules for anyInteger 9h $\rangle$ 
   $\langle$ strategies for anyInteger 10a $\rangle$ 
end

```

9c \langle imports for anyInteger 9c $\rangle \equiv$ (9b)

9d \langle sorts for anyInteger 9d $\rangle \equiv$ (9b)

9e \langle operators for anyInteger 9e $\rangle \equiv$ (9b)
 \langle global operators for anyInteger 9f \rangle
 \langle local operators for anyInteger 9g \rangle

9f \langle global operators for anyInteger 9f $\rangle \equiv$ (9e)

9g \langle local operators for anyInteger 9g $\rangle \equiv$ (9e)

9h \langle rules for anyInteger 9h $\rangle \equiv$ (9b)

10a $\langle \textit{strategies for anyInteger 10a} \rangle \equiv$ (9b)

10b $\langle * 10b \rangle \equiv$
// This file is generated automatically: do not edit it directly.
 $\langle \textit{the anyInteger speci cation 9a} \rangle$

1.4 array.eln

This introduces unidimensional arrays (vectors) of length smaller than 32.

```
11a <the array speci cation 11a>≡ (12)
  module array[N,X]
    <imports for array 11b>
    <sorts for array 11c>
    <operators for array 11d>
    <rules for array 11f>
  end
```

```
11b <imports for array 11b>≡ (11a)
  import global int;
  end
```

Uses int 35 46f 101.

the module introduces the sort `array[N,X]` which is parametrized by the size `N` of the array (a natural of course) and by the sort of the array elements `X`:

```
11c <sorts for array 11c>≡ (11a)
  sort array[N,X];
  end
```

```
11d <operators for array 11d>≡ (11a)
  <global operators for array 11e>
```

In the definition of the operators and later of the rules, notice the use of the pre-processor features.

```
11e <global operators for array 11e>≡ (11d)
  operators global
    [ {@}_I=1...N ] : ({X}_I=1...N) array[N,X];
    @[@] : (array[N,X] int) X;
    @[@<-@] : (array[N,X] int X) array[N,X];
    new(@) : (X) array[N,X];
  end
```

Uses @[@<-@] 101, @[@] 101, [@] 35, and int 35 46f 101.

```
11f <rules for array 11f>≡ (11a)
  rules for array[N,X]
    { x_J : X; }_J=0...(N-1)
      u : X;
  global
    [] new(u) => [{u}_J=0...(N-1) ]
  end
  {
    [] [ {x_J}_J=0...(N-1) ] [I<-u] =>
      [{x_J}_J=0...(I-1) u {x_J}_J=(I+1)...(N-1)]
  end
  }_I=0...(N-1)
  end

  rules for X
```

```
{ x_J : X; }_J=0...(N-1)
global
{
  [] [ {x_J}_J=0...(N-1) ] [I] => x_I
  end
  }_I=0...(N-1)
end
```

Uses + 23b 35 and - 23b 35.

12 $\langle * 12 \rangle \equiv$
// This file is generated automatically: do not edit it directly.
 \langle the array specification 11a \rangle

1.5 arrayLIN.eln

15a *(the arrayLIN specification 15a)*≡ (16)

```

module arrayLIN[X]
  <imports for arrayLIN 15b>
  <sorts for arrayLIN 15c>
  <operators for arrayLIN 15d>
  <rules for arrayLIN 15f>
end

```

15b *(imports for arrayLIN 15b)*≡ (15a)

```

import global int X;
end

```

Uses int 35 46f 101.

15c *(sorts for arrayLIN 15c)*≡ (15a)

```

sort arrayLIN[X];
end

```

15d *(operators for arrayLIN 15d)*≡ (15a)

(global operators for arrayLIN 15e)

15e *(global operators for arrayLIN 15e)*≡ (15d)

```

operators global
  new(@)           : (X) arrayLIN[X];
  set(@, @, @)    : (arrayLIN[X] int X) arrayLIN[X];
  @[@<-@]         : (arrayLIN[X] int X) arrayLIN[X] alias set(@, @, @)::;
  get(@, @)       : (arrayLIN[X] int) X;
  @[@]            : (arrayLIN[X] int) X alias get(@, @)::;
  remove(@, @)    : (arrayLIN[X] int) arrayLIN[X];

//local
  (@, @). @       : (int X arrayLIN[X]) arrayLIN[X];
  get_init(@)     : (arrayLIN[X]) X;
end

```

Uses @[@<-@] 101, @[@] 101, [@] 35, and int 35 46f 101.

15f *(rules for arrayLIN 15f)*≡ (15a)

```

rules for X
  i, j: int;
  t: arrayLIN[X];
  elt: X;

global
  [] get_init((i, elt).t) => get_init(t)           end
  [] get_init(new(elt)) => elt                     end
  [] get((i, elt).t, j) => elt                     if i==j   end
  [] get((i, elt).t, j) => get(t, j)              if i<j     end
  [] get((i, elt).t, j) => get_init(t)           if i>j     end

```

```

[] get(new(elt), i) => elt                                end
end

rules for arrayLIN[X]
  i,j: int;
    t: arrayLIN[X];
    elt1, elt2: X;
global
[] set(new(elt2), i, elt1) => (i, elt1).new(elt2)          end
[] set((i,elt1).t, j, elt2) => (i, elt2).t                if i==j    end
[] set((i, elt1).t, j, elt2) => (i, elt1).set(t, j, elt2) if i<j    end
[] set((i, elt1).t, j, elt2) => (j,elt2).(i, elt1).t     if i>j    end

[] remove(new(elt1), i) => new(elt1)                      end
[] remove((i,elt1).t, j) => t                             if i==j    end
[] remove((i, elt1).t, j) => (i, elt1).remove(t, j)      if i<j    end
[] remove((i, elt1).t, j) => (i, elt1).t                 if i>j    end
end

```

Uses < 20 23b 35, == 20 23b 31d 35 62e 88, > 20 23b 35, and int 35 46f 101.

16 < * 16 > ≡
 // This file is generated automatically: do not edit it directly.
 < the arrayLIN specification 15a >

1.6 arrayLOG.eln

15a *(the arrayLOG specification 15a)*≡ (16)

```

module arrayLOG[N,X]
  <imports for arrayLOG 15b>
  <sorts for arrayLOG 15c>
  <operators for arrayLOG 15d>
  <rules for arrayLOG 15f>
end

```

15b *(imports for arrayLOG 15b)*≡ (15a)

```

import global bool int X array[10,X] array[10,arrayLOG[N,X]];
end

```

Uses int 35 46f 101.

15c *(sorts for arrayLOG 15c)*≡ (15a)

```

sort arrayLOG[N,X];
end

```

15d *(operators for arrayLOG 15d)*≡ (15a)

(global operators for arrayLOG 15e)

15e *(global operators for arrayLOG 15e)*≡ (15d)

```

operators global
  new(@)          : (X) arrayLOG[N,X];
  set(@, @, @)    : (arrayLOG[N,X] int X) arrayLOG[N,X];
  @[@<-@]         : (arrayLOG[N,X] int X) arrayLOG[N,X] alias set(@, @, @)::;
  get(@, @)       : (arrayLOG[N,X] int) X;
  @[@]           : (arrayLOG[N,X] int) X alias get(@,@)::;
//local
  (@)            : (array[10,X]) arrayLOG[N,X];
  [@]            : (array[10,arrayLOG[N,X]]) arrayLOG[N,X];
  new(@,@)       : (int X) arrayLOG[N,X];
end

```

Uses @[<-@] 101, @[@] 101, [@] 35, and int 35 46f 101.

15f *(rules for arrayLOG 15f)*≡ (15a)

```

rules for arrayLOG[N,X]
  e      : X;
  a      : arrayLOG[N,X];
  i: int;
global
  [] new(e) => new(N,e) end
  [] new(i,e) => ([e e e e e e e e e e]) if i < 10 end
  [] new(i,e) => [[a a a a a a a a a a]] where a:=()new(i/10,e) end
end

```



```

rules for X
  as      : array[10,arrayLOG[N,X]];
  es      : array[10,X];
  i       : int;
global
[] get([as], i) => get(as[i%10],i/10)          end
[] get((es), i) => es[i%10]                    end
end

rules for arrayLOG[N,X]
  as      : array[10,arrayLOG[N,X]];
  es      : array[10,X];
  i       : int;
  e       : X;
global
[] set([as], i, e) => [as[i%10<-set(as[i%10], i/10, e)]]  end
[] set((es), i, e) => (es[i%10<-e])                    end
end

```

Uses % 23b 35, / 23b 35 35, < 20 23b 35, and int 35 46f 101.

16 < * 16 > ≡
 // This file is generated automatically: do not edit it directly.
 < the arrayLOG speci cation 15a >

1.7 basio.eln

17a *<the basio specification 17a>*≡ (18b)

```

module basio[X]
  <imports for basio 17b>
  <sorts for basio 17c>
  <operators for basio 17d>
  <rules for basio 18a>
end

```

Uses `basio` 37f.

17b *<imports for basio 17b>*≡ (17a)

```

import global bool builtinInt stdio;
end

```

Uses `builtinInt` 23b 35 46f 101.

17c *<sorts for basio 17c>*≡ (17a)

```

sort X;
end

```

17d *<operators for basio 17d>*≡ (17a)

```

<global operators for basio 17e>
<local operators for basio 17f>

```

The semantics of the operators are the following:

- `'write(pid,x)'` writes a term $x:X$ to a file or an output pipe `'pid'` and in the successful case, it returns the same term x . If an error occurs the result of `write` is a term `Error(e)`, where `'e'` is an error index.
- `'read(p)'` reads a term of the type X , if an error occurs it returns a term `Error(e)`.

17e *<global operators for basio 17e>*≡ (17d)

```

operators global
  write(@,@)      : (Pid X) X   pri 200 code -119;
                  // write to a file/output/to a pipe
  read(@)         : (Pid) X     pri 200 code -120;
                  // read from a file/get from a pipe
  iserror(@)     : (X) bool;
  errno(@)      : (X) builtinInt;

```

Uses `-` 23b 35, `/` 23b 35 35, `builtinInt` 23b 35 46f 101, `errno` 25d, `Pid` 25d, and `write` 37f.

17f *<local operators for basio 17f>*≡ (17d)

```

local
  Error(@)       : (builtinInt) X   pri 200 code 123;
end

```

Uses `builtinInt` 23b 35 46f 101 and `Error` 25d.

```

18a  <rules for basio 18a>≡ (17a)
      rules for bool
      x:X; e:builtinInt;
      global
        [] iserror(Error(e)) => true      end
        [] iserror(x) => false           end
      end

      rules for builtinInt
      x:X; e:builtinInt;
      global
        [] errno(Error(e)) => e          end
      end

```

Uses builtinInt 23b 35 46f 101, errno 25d, Error 25d, false 20, and true 20.

```

18b  <* 18b>≡
      // This file is generated automatically: do not edit it directly.
      <the basio speci cation 17a>

```

1.8 bool.eln

This module defines the two built-ins constants for boolean : `true` and `false`. It also defines the logical operators on booleans and the operations for testing ordering on booleans.

```
19a <the bool speci cation 19a>≡ (20)
  module bool
    <sorts for bool 19b>
    <global operators for bool 19c>
  end
```

```
19b <sorts for bool 19b>≡ (19a)
  sort   bool;
  end
```

This module first defines two built-in constants (`true` and `false`) referred by the first and second entries of the table of symbols.

```
19c <global operators for bool 19c>≡ (19a) 19d>
  operators
  global
    true      : bool      builtin 1;
    false     : bool      builtin 0;
```

Uses `false` 20 and `true` 20.

After, we can define all the basic logical operators on booleans (conjunction `and`, disjunction `or` and negation `not(@)`). They are define by built-ins. Then, we define aliases for these operators considering all types of brackets.

```
19d <global operators for bool 19c>+≡ (19a) <19c 19e>
  @ and @      : (bool bool) bool   assocLeft pri 100 code 21;
  @ or @       : (bool bool) bool   assocLeft pri 100 code 22;
  (@ and @)    : (bool bool) bool   assocLeft pri 100 alias @ and @::;
  (@ or @)     : (bool bool) bool   assocLeft pri 100 alias @ or @::;
  not(@)       : (bool) bool        pri 200 code 24;
  not @        : (bool) bool        pri 200 alias not(@)::;
  (not @)      : (bool) bool        pri 200 alias not(@)::;
  (not(@))     : (bool) bool        pri 200 alias not(@)::;
```

Uses `and` 20 88, `not(@)` 20 88, and `or` 20 88.

For defining the operators for testing two booleans, we assume that `false`<`true`. Considering this, we can test the equality (`==`), the inequality (`!=`) and the other tests (`<`, `<=`, `>` and `>=`) using the built-ins definition. After, with aliases, we define the equivalent prefix operators.

```
19e <global operators for bool 19c>+≡ (19a) <19d
  @ == @       : (bool bool) bool   assocLeft pri 200 code 8;
  @ != @       : (bool bool) bool   assocLeft pri 200 code 9;
  @ < @        : (bool bool) bool   assocLeft pri 200 code 10;
  @ <= @       : (bool bool) bool   assocLeft pri 200 code 11;
  @ > @        : (bool bool) bool   assocLeft pri 200 code 12;
  @ >= @       : (bool bool) bool   assocLeft pri 200 code 13;

  eq_bool(@,@) : (bool bool) bool   assocLeft pri 200 alias @ == @::;
  neq_bool(@,@) : (bool bool) bool   assocLeft pri 200 alias @ != @::;
  less_bool(@,@) : (bool bool) bool   assocLeft pri 200 alias @ < @::;
```

```

lesseq_bool(@,@)      : (bool bool) bool    assocLeft pri 200 alias @ <= @:;
greater_bool(@,@)     : (bool bool) bool    assocLeft pri 200 alias @ > @:;
greatereq_bool(@,@)  : (bool bool) bool    assocLeft pri 200 alias @ >= @:;
end

```

Uses != 20 23b 31d 35 62e 88, < 20 23b 35, <= 20 23b 35, == 20 23b 31d 35 62e 88, > 20 23b 35, and >= 20 23b 35.

20 < * 20>≡
 // This file is generated automatically: do not edit it directly.
 <the bool specification 19a>

Defines:

```

!=, used in chunks 19e, 22b, 30, 31c, 33e, 34b, 62d, 74b, and 87.
<, used in chunks 15, 19e, 22b, 30, 33e, 34b, and 39d.
<=, used in chunks 19e, 22b, 30, 33e, and 34b.
==, used in chunks 15f, 19e, 22b, 30, 31c, 33e, 34b, 56c, 62d, 72b, and 87.
>, used in chunks 15f, 19e, 22b, 30, 33e, 34b, and 39.
>=, used in chunks 19e, 22b, 30, 33e, and 34b.
and, used in chunks 19d, 22a, 33e, 41a, 45a, and 87.
false, used in chunks 18a, 19c, 25c, and 43b.
not(@), used in chunks 19d and 87e.
or, used in chunks 19d, 22a, 33e, and 87.
true, used in chunks 18a, 19c, 25c, and 43b.

```

1.9 builtinInt.eln

This module defines the sort `builtinInt` and the standard operations on integers (arithmetical and test operations). This module is used by the `int.eln` module which defines the sort `int` usually used in ELAN programs.

21a *<the builtinInt specification 21a>*≡ (23b)

```

module builtinInt
  <imports for builtinInt 21b>
  <sorts for builtinInt 21c>
  <global operators for builtinInt 21d>

end

```

Uses `builtinInt` 23b 35 46f 101.

21b *<imports for builtinInt 21b>*≡ (21a)

```

import
  global anyInteger;
  local bool;

end

```

21c *<sorts for builtinInt 21c>*≡ (21a)

```

sort builtinInt;

end

```

Uses `builtinInt` 23b 35 46f 101.

The first definitions are for the basic infix operations on integers: addition (+), soustraction (-), multiplication (*), division (/), the modulo operation (%), the logical conjunction of two binary representations of integers (& - it returns the integer corresponding to the binary number obtained), the logical disjunction of two binary representations of integers (| - it returns the integer corresponding to the binary number obtained) and the unary minus (-). All these operators are built-ins.

21d *<global operators for builtinInt 21d>*≡ (21a) 22a▷

```

operators
global
  @ + @ : (builtinInt builtinInt) builtinInt pri 500 assocLeft code 3;
  @ - @ : (builtinInt builtinInt) builtinInt pri 600 assocLeft code 4;
  @ * @ : (builtinInt builtinInt) builtinInt pri 700 assocLeft code 5;
  @ / @ : (builtinInt builtinInt) builtinInt pri 800 assocLeft code 6;
  @ % @ : (builtinInt builtinInt) builtinInt pri 800 assocLeft code 27;
  @ & @ : (builtinInt builtinInt) builtinInt pri 800 assocLeft code 28;
  @ | @ : (builtinInt builtinInt) builtinInt pri 800 assocLeft code 29;
  - @   : (builtinInt) builtinInt          pri 900          code 20;

```

Uses % 23b 35, & 23b 35 59d, * 23b 35, + 23b 35, - 23b 35, / 23b 35 35, | 23b 35, and `builtinInt` 23b 35 46f 101.

Then, for the same operators, we define by aliases firstly the versions considering brackets and then the prefix representations.

```
22a <global operators for builtinInt 21d>+≡ (21a) <21d 22b>
  (@ + @) : (builtinInt builtinInt) builtinInt pri 500 assocLeft alias @ + @;;
  (@ - @) : (builtinInt builtinInt) builtinInt pri 600 assocLeft alias @ - @;;
  (@ * @) : (builtinInt builtinInt) builtinInt pri 700 assocLeft alias @ * @;;
  (@ / @) : (builtinInt builtinInt) builtinInt pri 800 assocLeft alias @ / @;;
  (@ % @) : (builtinInt builtinInt) builtinInt pri 800 assocLeft alias @ % @;;
  (@ & @) : (builtinInt builtinInt) builtinInt pri 800 assocLeft alias @ & @;;
  (@ | @) : (builtinInt builtinInt) builtinInt pri 800 assocLeft alias @ | @;;
  (- @)   : (builtinInt) builtinInt          pri 900          alias - @;;

  plus(@,@) : (builtinInt builtinInt) builtinInt  alias @ + @;;
  minus(@,@) : (builtinInt builtinInt) builtinInt  alias @ - @;;
  time(@,@) : (builtinInt builtinInt) builtinInt  alias @ * @;;
  div(@,@)  : (builtinInt builtinInt) builtinInt  alias @ / @;;
  mod(@,@)  : (builtinInt builtinInt) builtinInt  alias @ % @;;
  and(@,@)  : (builtinInt builtinInt) builtinInt  alias @ & @;;
  or(@,@)   : (builtinInt builtinInt) builtinInt  alias @ | @;;
  umin(@)   : (builtinInt) builtinInt            alias - @;;
  umin_builtinInt(@): (builtinInt) builtinInt    alias umin(@);;
```

Uses % 23b 35, & 23b 35 59d, * 23b 35, + 23b 35, - 23b 35, / 23b 35 35, | 23b 35, and 20 88, builtinInt 23b 35 46f 101, and or 20 88.

Then, we define the tests between two integers: equality (==), inequality (!=), inferiority (<), inferiority or equality (<=), superiority (>) and superiority or equality (>=). They are built-ins. After, we can define the prefix versions corresponding to each ones by aliases.

```
22b <global operators for builtinInt 21d>+≡ (21a) <22a 23a>
  @ == @      : (builtinInt builtinInt) bool      pri 300 code 8;
  @ != @      : (builtinInt builtinInt) bool      pri 300 code 9;
  @ < @       : (builtinInt builtinInt) bool      pri 300 code 10;
  @ <= @      : (builtinInt builtinInt) bool      pri 300 code 11;
  @ > @       : (builtinInt builtinInt) bool      pri 300 code 12;
  @ >= @      : (builtinInt builtinInt) bool      pri 300 code 13;

  eq_builtinInt(@,@)      : (builtinInt builtinInt) bool  pri 300 alias @ == @;;
  neq_builtinInt(@,@)     : (builtinInt builtinInt) bool  pri 300 alias @ != @;;
  less_builtinInt(@,@)    : (builtinInt builtinInt) bool  pri 300 alias @ < @;;
  lesseq_builtinInt(@,@)  : (builtinInt builtinInt) bool  pri 300 alias @ <=@;;
  greater_builtinInt(@,@) : (builtinInt builtinInt) bool  pri 300 alias @ > @;;
  greatereq_builtinInt(@,@) : (builtinInt builtinInt) bool  pri 300 alias @ >=@;;
```

Uses != 20 23b 31d 35 62e 88, < 20 23b 35, <= 20 23b 35, == 20 23b 31d 35 62e 88, > 20 23b 35, >= 20 23b 35, and builtinInt 23b 35 46f 101.

Finally, the conversions from booleans to integers are defined as built-ins: the operator `btoi` for conversion from boolean to `builtinInt` and `itob` for the opposite conversion. To briefly explain how it works, we consider that false corresponds to 0, that true is converted to 1 and that any integer different from 0 is converted to true.

```
23a <global operators for builtinInt 21d>+≡ (21a) <22b
    btoi(@)          : (bool) builtinInt          pri 900 code 25;
    btoi_builtinInt(@) : (bool) builtinInt      pri 900 alias btoi(@)::

    itob(@)         : (builtinInt) bool          pri 900 code 26;
    itob_builtinInt(@) : (builtinInt) bool      pri 900 alias itob(@)::
end
```

Uses `btoi` 23b 35, `builtinInt` 23b 35 46f 101, and `itob` 23b 35.

```
23b <* 23b>≡
// This file is generated automatically: do not edit it directly.
<the builtinInt specification 21a>
```

Defines:

```
!=, used in chunks 19e, 22b, 30, 31c, 33e, 34b, 62d, 74b, and 87.
%, used in chunks 15f, 21d, 22a, 33e, and 34b.
&, used in chunks 21d, 22a, 33e, 34b, 58, 59, and 68.
*, used in chunks 21d, 22a, 33e, 34b, 39c, and 40a.
+, used in chunks 11f, 21d, 22a, 27e, 33e, 34b, 39e, and 98d.
-, used in chunks 8a, 9a, 11f, 17e, 21d, 22a, 33e, 34b, 36c, 39, 41a, 42e, 45a, 98d, and 51.
/, used in chunks 15f, 17e, 21d, 22a, 33e, and 34b.
<, used in chunks 15, 19e, 22b, 30, 33e, 34b, and 39d.
<=, used in chunks 19e, 22b, 30, 33e, and 34b.
==, used in chunks 15f, 19e, 22b, 30, 31c, 33e, 34b, 56c, 62d, 72b, and 87.
>, used in chunks 15f, 19e, 22b, 30, 33e, 34b, and 39.
>=, used in chunks 19e, 22b, 30, 33e, and 34b.
|, used in chunks 21d, 22a, 33e, and 34b.
btoi, used in chunks 23a and 33e.
builtinInt, used in chunks 5-7, 17, 18a, 21-25, 27, 28, 33, 34, 37b, 46e, and 47d.
itob, used in chunks 23a and 33e.
```


1.10 builtinStdio.eln

This module is the elementary module for defining the input/output. We find in this module all the built-ins needed for creating or deleting a pipe.

```
24a <the builtinStdio speci cation 24a>≡ (25d)
  module builtinStdio
    <imports for builtinStdio 24b>
    <sorts for builtinStdio 24c>
    <operators for builtinStdio 24d>
    <rules for builtinStdio 25c>
  end
  Uses builtinStdio 46f.
```

```
24b <imports for builtinStdio 24b>≡ (24a)
  import global builtinInt bool builtinString;
  end
  Uses builtinInt 23b 35 46f 101 and builtinString 101.
```

```
24c <sorts for builtinStdio 24c>≡ (24a)
  sort Pid;
  end
  Uses Pid 25d.
```

```
24d <operators for builtinStdio 24d>≡ (24a)
  <global operators for builtinStdio 24e>
  <local operators for builtinStdio 25b>
```

The following operators are built-ins. `getc` gets the next character from an input file or from an input pipe. `putc` puts a character in an input file or an input pipe. `create` creates a process with a given name (this name can be a name of an executable file, or a script file). The created process is linked with the parent process via two blocking pipes. `create_noblock` creates a process with a given name too but in this case, the two communication pipes created are in a non blocking mode, e.g. it is possible to read a data from an input pipe, even if they are not there (they are not ready yet). It allows to define a non blocking parent process.

```
24e <global operators for builtinStdio 24e>≡ (24d) 24f▷
  operators global
    getc(@)           : (Pid) builtinInt           code 113;
    putc(@,@)        : (Pid builtinInt) builtinInt code 114;
    intern putc(@,@) : (Pid builtinInt) builtinInt  alias putc(@,@) ;;
    create(@)         : (string) Pid               code 115;
    create_noblock(@) : (string) Pid               code 117;
  Uses builtinInt 23b 35 46f 101, create 25d, create_noblock 25d, getc 25d, Pid 25d, putc 25d 46f, and string 28d 101 101.
```

These two operators are built-ins too, but they manage the opening and closing of files. `open` opens a file for reading or writing : `open(file_name,"w")` opens the file `file_name` for writing and the returns the pipe identifier associated of sort `Pid` ; `open(file_name,"r")` open the file `file_name` for reading and returns a pipe identifier too. `close` closes the file relative to the given pipe identifier of sort `Pid`.

```
24f <global operators for builtinStdio 24e>+≡ (24d) <24e 25a▷
  open(@,@)          : (string string) Pid         code 116;
  close(@)           : (Pid) Pid                   code 118;
  Uses close 25d, open 25d, Pid 25d, and string 28d 101 101.
```

These operators that are not built-ins, are described by rewriting rules. `stdin`, `stdout` and `stderr` corresponds to the standard input, the standard output and the standard error. `iserr` detects if a `Pid` is an error and `errno` retruns the number associated to the error if the `Pid` given in argument is an error.

```
25a <global operators for builtinStdio 24e>+≡ (24d) <24f
    stdin      : Pid;
    stdout     : Pid;
    stderr     : Pid;
    iserr(@)   : (Pid) bool;
    errno(@)   : (Pid) builtinInt;
```

Uses `builtinInt` 23b 35 46f 101, `errno` 25d, `iserr` 25d, `Pid` 25d, `stderr` 25d, `stdin` 25d, and `stdout` 25d.

An error is constructed by the operator `Error` with an integer associated which gives its number and reports various errors of the input.output sub-system written in C++. And an element of sort `int` can be of sort `Pid`.

```
25b <local operators for builtinStdio 25b>≡ (24d)
    local
    @      : (builtinInt) Pid          code 121;
    Error(@) : (builtinInt) Pid          code 122;
    end
```

Uses `builtinInt` 23b 35 46f 101, `Error` 25d, and `Pid` 25d.

The numbers associated to `Pid` `stdin`, `stdout` and `stderr` corresponds to the values of these three constants in Unix.

```
25c <rules for builtinStdio 25c>≡ (24a)
    rules for Pid
    x:Pid;
    global
    []  stdin  => 0  end
    []  stdout => 1  end
    []  stderr => 2  end
    end

    rules for bool
    x:Pid; e:builtinInt;
    global
    []  iserr(Error(e)) => true  end
    []  iserr(x) => false  end
    end

    rules for builtinInt
    x:Pid; e:builtinInt;
    global
    []  errno(Error(e)) => e  end
    end
```

Uses `builtinInt` 23b 35 46f 101, `errno` 25d, `Error` 25d, `false` 20, `iserr` 25d, `Pid` 25d, `stderr` 25d, `stdin` 25d, `stdout` 25d, and `true` 20.

```
25d <* 25d>≡
    // This file is generated automatically: do not edit it directly.
    <the builtinStdio specification 24a>
```

Defines:

`close`, used in chunk 24f.
`create`, used in chunk 24e.
`create_noblock`, used in chunk 24e.
`errno`, used in chunks 17e, 18a, and 25.

Error, used in chunks 17f, 18a, and 25.

getc, used in chunk 24e.

iserr, used in chunk 25.

open, used in chunk 24f.

Pid, used in chunks 17e, 24, 25, 37, and 46.

putc, used in chunks 24e and 46.

stderr, used in chunk 25.

stdin, used in chunks 25 and 44e.

stdout, used in chunks 25 and 44e.

1.11 builtinString.eln

This module defines some current built-ins for the sort `string` and aliases for most of them.

```
27a <the builtinString speci cation 27a>≡ (28d)
  module builtinString
    <imports for builtinString 27b>
    <sorts for builtinString 27c>
    <global operators for builtinString 27d>
  end
```

Uses `builtinString` 101.

```
27b <imports for builtinString 27b>≡ (27a)
  import global anyString builtinInt ident;
  end
```

Uses `builtinInt` 23b 35 46f 101.

```
27c <sorts for builtinString 27c>≡ (27a)
  sort string;
  end
```

Uses `string` 28d 101 101.

The operator `strlen` computes the length for a given string.

```
27d <global operators for builtinString 27d>≡ (27a) 27e>
  operators global
    strlen(@) : (string) builtinInt pri 2000 code 150;
```

Uses `builtinInt` 23b 35 46f 101, `string` 28d 101 101, and `strlen` 28d.

The operator `strcat` takes two strings `s1` and `s2` and `strcat(s1,s2)` returns the concatenated corresponding string.

```
27e <global operators for builtinString 27d>+≡ (27a) <27d 27f>
  strcat(@,@) : (string string) string pri 2000 assocRight code 151;
  @+@ : (string string) string pri 2000 assocRight alias strcat(@,@);;
```

Uses + 23b 35, `strcat` 28d, and `string` 28d 101 101.

The following operators selects a character in a string: for example, `s[i]` selects the *i*-th character in the string `s`.

```
27f <global operators for builtinString 27d>+≡ (27a) <27e 27g>
  @[@] : (string builtinInt) builtinInt pri 2000 code 152;
  intern @[@] : (string builtinInt) builtinInt pri 2000 alias @[@];;
```

Uses `@[@]` 101, `[@]` 35, `builtinInt` 23b 35 46f 101, and `string` 28d 101 101.

`s[i<-c]` replaces in the string `s` the *i*-th character by the character corresponding to the integer `c`.

```
27g <global operators for builtinString 27d>+≡ (27a) <27f 27h>
  @[@<-@] : (string builtinInt builtinInt) string pri 2000 code 153;
  intern @[@<-@] : (string builtinInt builtinInt) string pri 2000 alias @[@<-@];;
```

Uses `@[@<-@]` 101, `builtinInt` 23b 35 46f 101, and `string` 28d 101 101.

`substr(s,i,n)` extracts from the string `s` a substring of length `n` from the *i*-th position.

```
27h <global operators for builtinString 27d>+≡ (27a) <27g 28a>
  substr(@,@,@) : (string builtinInt builtinInt) string pri 2000 code 154;
  intern substr(@,@,@) : (string builtinInt builtinInt) string pri 2000
    alias substr(@,@,@);;
```

Uses `builtinInt` 23b 35 46f 101, `string` 28d 101 101, and `substr` 28d 101.

`strspn(s1,s2)` returns the length of the first part of `s1` entirely constituted by characters of `s2`. `strcmp(s1,s2)` compares two strings: it begins by the first character of each string and continues step by step until the two characters are different (then it compares the two different integer corresponding to the characters) or the end of a string is reached. It returns a negative integer if `s1` is inferior to `s2`, 0 if they are identicals and a positive integer if `s1` is superior to `s2`.

```
28a <global operators for builtinString 27d>+≡ (27a) <27h 28b>
      strspn(@,@)      : (string string) builtinInt    pri 2000 code 156;
      strcmp(@,@)     : (string string) builtinInt    pri 2000 code 157;
```

Uses `builtinInt` 23b 35 46f 101, `strcmp` 28d, `string` 28d 101 101, and `strspn` 28d.

`string` converts a given integer to the corresponding string.

```
28b <global operators for builtinString 27d>+≡ (27a) <28a 28c>
      string(@)       : (builtinInt) string           pri 2000 code 158;
      intern string(@) : (builtinInt) string           pri 2000 alias string(@);
```

Uses `builtinInt` 23b 35 46f 101 and `string` 28d 101 101.

`ident2string` converts an element of sort `ident` to the sort `string`.

```
28c <global operators for builtinString 27d>+≡ (27a) <28b>
      ident2string(@) : (ident) string                code 177;
      end
```

Uses `ident2string` 28d and `string` 28d 101 101.

```
28d <* 28d>≡
      // This file is generated automatically: do not edit it directly.
      <the builtinString speci cation 27a>
```

Defines:

```
ident2string, used in chunk 28c.
strcat, used in chunk 27e.
strcmp, used in chunk 28a.
string, used in chunks 24, 27, 28, 37, 44, 46b, 99a, 47, 98b, 49, and 98d.
strlen, used in chunks 27d and 98d.
strspn, used in chunk 28a.
substr, used in chunks 27h and 47.
```

1.12 builtinSyntacticMatching.eln

This is a very short module that defines the built-in operators for matching and unifying.

- 29a *<the builtinSyntacticMatching specification 29a>*≡ (29e)
- ```

module builtinSyntacticMatching[X,Y]
 <sorts for builtinSyntacticMatching 29b>
 <operators for builtinSyntacticMatching 29c>
end

```
- 29b *<sorts for builtinSyntacticMatching 29b>*≡ (29a)
- ```

sort X Y ;
end

```
- 29c *<operators for builtinSyntacticMatching 29c>*≡ (29a)
- ```

<global operators for builtinSyntacticMatching 29d>

```
- 29d *<global operators for builtinSyntacticMatching 29d>*≡ (29c)
- ```

operators
  global
    builtinSyntacticMatching(@,@,@,@) : (X X Y Y) Y code 191;
    builtinSyntacticUnification(@,@,@,@) : (X X Y Y) Y code 175;
end

```
- 29e *<* 29e>*≡
- ```

// This file is generated automatically: do not edit it directly.
<the builtinSyntacticMatching specification 29a>

```

### 1.13 cmp.eln

Note that in the current implementation, X can NOT be a built-in.

```
30a <the cmp specification 30a>≡ (30f)
 module cmp[X]
 <imports for cmp 30b>
 <sorts for cmp 30c>
 <operators for cmp 30d>
 <global operators for cmp 30e>
 end
```

Uses cmp 31d.

```
30b <imports for cmp 30b>≡ (30a)
 import bool;
 end
```

```
30c <sorts for cmp 30c>≡ (30a)
 sort X;
 end
```

```
30d <operators for cmp 30d>≡ (30a)
 operators
 global
 // In Elan, one can only test equality of objects, never their
 // identity (i.e. to be the same internal object)
 @ == @ : (X X) bool code 18;
 @ != @ : (X X) bool code 19;
 @ < @ : (X X) bool code 30;
 @ <= @ : (X X) bool code 31;
 @ > @ : (X X) bool code 32;
 @ >= @ : (X X) bool code 33;
```

Uses != 20 23b 31d 35 62e 88, < 20 23b 35, <= 20 23b 35, == 20 23b 31d 35 62e 88, > 20 23b 35, >= 20 23b 35, and identity 32e 54 59d 65d.

And now textual aliases for the operators above:

```
30e <global operators for cmp 30e>≡ (30a)
 eq_X(@,@) : (X X) bool alias @ == @:;
 neq_X(@,@) : (X X) bool alias @ != @:;
 less_X(@,@) : (X X) bool alias @ < @:;
 lesseq_X(@,@) : (X X) bool alias @ <= @:;
 greater_X(@,@) : (X X) bool alias @ > @:;
 greatereq_X(@,@) : (X X) bool alias @ >= @:;

 end
```

Uses != 20 23b 31d 35 62e 88, < 20 23b 35, <= 20 23b 35, == 20 23b 31d 35 62e 88, > 20 23b 35, and >= 20 23b 35.

```
30f <* 30f>≡
 // This file is generated automatically: do not edit it directly.
 <the cmp specification 30a>
```

## 1.14 eq.eln

This module in fact hides some operations defined in the module `cmp`: it doesn't add any fonctionnality but hide some useless ones given in `cmp[X]`.

We just define here the tests for equality (`==` and alias `eq_X`) and disequality (`!=` and alias `neq_X`) for a given sort `X` without defining an order on elements of this sort (we don't define the `<`, `<=`, `>` and `>=` operators).

```
31a <the eq speci cation 31a>≡ (31d)
 module eq[X]
 <imports for eq 31b>
 <global operators for eq 31c>
 end
```

```
31b <imports for eq 31b>≡ (31a)
 import bool cmp[X];
 end
Uses cmp 31d.
```

For defining these operators, we just make an alias with the same operators defined in the corresponding module `cmp[X]`.

```
31c <global operators for eq 31c>≡ (31a)
 operators
 global

 @ == @ : (X X) bool alias @ == @:;
 @ != @ : (X X) bool alias @ != @:;
 eq_X(@,@) : (X X) bool alias @ == @:;
 neq_X(@,@) : (X X) bool alias @ != @:;

 end
```

Uses != 20 23b 31d 35 62e 88 and == 20 23b 31d 35 62e 88.

```
31d <* 31d>≡
 // This file is generated automatically: do not edit it directly.
 <the eq speci cation 31a>
Defines:
 !=, used in chunks 19e, 22b, 30, 31c, 33e, 34b, 62d, 74b, and 87.
 ==, used in chunks 15f, 19e, 22b, 30, 31c, 33e, 34b, 56c, 62d, 72b, and 87.
 cmp, used in chunks 30a and 31b.
```



## 1.15 identity.eln

This module gives two ways of calling the identity on each element of a given sort X.

```
32a <the identity specification 32a>≡ (32e)
 module identity[X]
 <sorts for identity 32b>
 <global operators for identity 32c>
 <rules for identity 32d>
 end
```

Uses `identity` 32e 54 59d 65d.

```
32b <sorts for identity 32b>≡ (32a)
 sort X;
 end
```

We just have one operator called `identity` which returns its argument.

```
32c <global operators for identity 32c>≡ (32a)
 operators global
 identity(0) : (X) X;
 end
```

Uses `identity` 32e 54 59d 65d.

These two rules define the two ways of obtaining the identity: on one side, we simply use the operator `identity`, on the other side, we can include the second rule in a more general strategy by calling the defined rule `identity`.

```
32d <rules for identity 32d>≡ (32a)
 rules for X
 t : X;
 global
 [] identity(t) => t end
 [identity] t => t end
 end
```

Uses `identity` 32e 54 59d 65d.

```
32e <* 32e>≡
 // This file is generated automatically: do not edit it directly.
 <the identity specification 32a>
```

Defines:

`identity`, used in chunks 30d, 32, 53e, 59a, 65c, and 75a.

## 1.16 int.eln

This module defines the sort `int` usually used in the ELAN programs. No operator here is built-in: each one is defined starting from the definition given in `builtinInt`.

```
33a <the int specification 33a>≡ (35)
 module int
 <imports for int 33b>
 <sorts for int 33c>
 <operators for int 33d>
 <rules for int 34b>
 end
```

Uses `int` 35 46f 101.

We have here the importation of the module `builtinInt`.

```
33b <imports for int 33b>≡ (33a)
 import global builtinInt bool;
 end
```

Uses `builtinInt` 23b 35 46f 101.

```
33c <sorts for int 33c>≡ (33a)
 sort int;
 end
```

Uses `int` 35 46f 101.

```
33d <operators for int 33d>≡ (33a)
 <global operators for int 33e>
 <local operators for int 34a>
```

With respect to the module `builtinInt`, the only new operator is the first global one which says that any `builtinInt` is of sort `int` too. A second one is the operator `valueOf` which takes an `int` and returns the corresponding `builtinInt`. And the third one is a local operator `[@]` allowing the opposite conversion from `builtinInt` to `int`.

All the other operators are already defined in `builtinInt.eln` with a signature considering the sort `builtinInt` instead of the sort `int` used here.

```
33e <global operators for int 33e>≡ (33d)
 operators
 global
 @ : (builtinInt) int pri 1000;

 @ + @ : (int int) int assocLeft pri 500;
 @ - @ : (int int) int assocLeft pri 600;
 @ * @ : (int int) int assocLeft pri 700;
 @ % @ : (int int) int assocLeft pri 800;
 @ & @ : (int int) int assocLeft pri 800;
 @ / @ : (int int) int assocLeft pri 800;
 @ | @ : (int int) int assocLeft pri 800;
 - @ : (int) int pri 900;

 (@ + @) : (int int) int pri 500 alias @ + @:;
 (@ - @) : (int int) int pri 600 alias @ - @:;
 (@ * @) : (int int) int pri 700 alias @ * @:;
```

```

(@ / @) : (int int) int pri 800 alias @ / @:;
(@ % @) : (int int) int pri 800 alias @ % @:;
(@ & @) : (int int) int pri 800 alias @ & @:;
(@ | @) : (int int) int pri 800 alias @ | @:;
(- @) : (int) int pri 900 alias - @:;

plus(@,@) : (int int) int pri 500 alias @ + @:;
minus(@,@) : (int int) int pri 600 alias @ - @:;
time(@,@) : (int int) int pri 700 alias @ * @:;
div(@,@) : (int int) int pri 800 alias @ / @:;
mod(@,@) : (int int) int pri 800 alias @ % @:;
and(@,@) : (int int) int pri 800 alias @ & @:;
or(@,@) : (int int) int pri 800 alias @ | @:;
umin(@) : (int) int pri 900 alias - @:;

@ == @ : (int int) bool pri 300;
@ != @ : (int int) bool pri 300;
@ > @ : (int int) bool pri 300;
@ >= @ : (int int) bool pri 300;
@ <= @ : (int int) bool pri 300;
@ < @ : (int int) bool pri 300;

eq_int(@,@) : (int int) bool pri 300 alias @ == @:;
neq_int(@,@) : (int int) bool pri 300 alias @ != @:;
less_int(@,@) : (int int) bool pri 300 alias @ < @:;
lesseq_int(@,@) : (int int) bool pri 300 alias @ <=@:;
greater_int(@,@) : (int int) bool pri 300 alias @ > @:;
greatereq_int(@,@) : (int int) bool pri 300 alias @ >=@:;

btoi(@) : (bool) int pri 900;
btoi_int(@) : (bool) int pri 900 alias btoi(@):;
itob(@) : (int) bool pri 900;
itob_int(@) : (int) bool pri 900 alias itob(@):;
valueOf(@) : (int) builtinInt pri 900;

```

Uses != 20 23b 31d 35 62e 88, % 23b 35, & 23b 35 59d, \* 23b 35, + 23b 35, - 23b 35, / 23b 35 35, < 20 23b 35, <= 20 23b 35, == 20 23b 31d 35 62e 88, > 20 23b 35, >= 20 23b 35, | 23b 35, and 20 88, btoi 23b 35, builtinInt 23b 35 46f 101, int 35 46f 101, itob 23b 35, or 20 88, and valueOf 35.

```

34a <local operators for int 34a>≡ (33d)
 local
 [@] : (builtinInt) int pri 1000 alias @:;
 end

```

Uses [@] 35, builtinInt 23b 35 46f 101, and int 35 46f 101.

From the list of operators above, we must define all the operators that are not defined by aliases. That is the arithmetical operators with infix representation and without brackets (+, -, \*, %, &, &, |, / and -) and the infix comparison operators (==, !=, >, >=, <= and <). For these definitions, we use the [] operator which allows us to pass directly from the sort `int` to the built-ins sort `builtinInt`. For each definition for sort `int`, we use the corresponding one given in the `builtinInt.eln` module.

We have also to define `btoi` (or `btoi_int` here), `itob` (or `itob_int` here) and `valueOf`. The technic is the same than before: we define the operator by using the definition given in the `builtinInt.eln` module.

```

34b <rules for int 34b>≡ (33a)
 rules for int
 a,b,c : builtinInt;
 d : bool;
 global

```

```

[] [a]+[b] => [c] where c:=()a+b end
[] [a]-[b] => [c] where c:=()a-b end
[] [a]*[b] => [c] where c:=()a*b end
[] [a]%[b] => [c] where c:=()a%b end
[] [a]&[b] => [c] where c:=()a&b end
[] [a]|[b] => [c] where c:=()a|b end
[] [a]/[b] => [c] where c:=()a/b end
[] -[a] => [c] where c:=()-a end
[] btoi_int(d) => [c] where c:=()btoi_builtinInt(d) end
end

```

rules for bool

```

a,b: builtinInt;
c : bool;

```

global

```

[] [a]==[b] => eq_builtinInt(a,b) end
[] [a]!= [b] => neq_builtinInt(a,b) end
[] [a]<=[b] => lesseq_builtinInt(a,b) end
[] [a]<[b] => less_builtinInt(a,b) end
[] [a]>=[b] => greatereq_builtinInt(a,b) end
[] [a]>[b] => greater_builtinInt(a,b) end
[] itob_int([a]) => itob_builtinInt(a) end

```

end

rules for builtinInt

```

a : builtinInt;

```

global

```

[] valueOf([a]) => a end

```

end

Uses != 20 23b 31d 35 62e 88, % 23b 35, & 23b 35 59d, \* 23b 35, + 23b 35, - 23b 35, / 23b 35 35, < 20 23b 35, <= 20 23b 35, == 20 23b 31d 35 62e 88, > 20 23b 35, >= 20 23b 35, | 23b 35, builtinInt 23b 35 46f 101, int 35 46f 101, and valueOf 35.

35 (\* 35)≡  
// This file is generated automatically: do not edit it directly.  
<the int specification 33a>

Defines:

```

!=, used in chunks 19e, 22b, 30, 31c, 33e, 34b, 62d, 74b, and 87.
%, used in chunks 15f, 21d, 22a, 33e, and 34b.
&, used in chunks 21d, 22a, 33e, 34b, 58, 59, and 68.
*, used in chunks 21d, 22a, 33e, 34b, 39c, and 40a.
+, used in chunks 11f, 21d, 22a, 27e, 33e, 34b, 39e, and 98d.
-, used in chunks 8a, 9a, 11f, 17e, 21d, 22a, 33e, 34b, 36c, 39, 41a, 42e, 45a, 98d, and 51.
/, used in chunks 15f, 17e, 21d, 22a, 33e, and 34b.
<, used in chunks 15, 19e, 22b, 30, 33e, 34b, and 39d.
<=, used in chunks 19e, 22b, 30, 33e, and 34b.
==, used in chunks 15f, 19e, 22b, 30, 31c, 33e, 34b, 56c, 62d, 72b, and 87.
>, used in chunks 15f, 19e, 22b, 30, 33e, 34b, and 39.
>=, used in chunks 19e, 22b, 30, 33e, and 34b.
[@], used in chunks 5e, 11e, 15, 27f, 34a, and 47c.
|, used in chunks 21d, 22a, 33e, and 34b.
btoi, used in chunks 23a and 33e.
builtinInt, used in chunks 5-7, 17, 18a, 21-25, 27, 28, 33, 34, 37b, 46e, and 47d.
int, used in chunks 9a, 11, 15, 33, 34, 36, 38, 39, 46, 99b, 47, 98b, 49, 98d, and 51b.
itob, used in chunks 23a and 33e.
valueOf, used in chunks 33e, 34b, 46e, and 47d.

```

## 1.17 integerConstants.eln

This module is NOT loadable. It only describes the semantics of the built-ins elements of the sort `int`, i.e all the integer numbers.

36a *<the integerConstants specification 36a>*≡ (36d)

```

module integerConstants
 <sorts for integerConstants 36b>
 <global operators for integerConstants 36c>
end

```

36b *<sorts for integerConstants 36b>*≡ (36a)

```

sort int;
end

```

Uses `int` 35 46f 101.

36c *<global operators for integerConstants 36c>*≡ (36a)

```

operators
global

 0 : int;
 1 : int;
 -1 : int;
 2 : int;
 -2 : int;
 3 : int;
.
.
.
end

```

Uses - 23b 35 and `int` 35 46f 101.

36d *<\* 36d>*≡

```

// This file is generated automatically: do not edit it directly.
<the integerConstants specification 36a>

```

## 1.18 io.eln

This module only defines the ability to write in a pipe an element of sort X and then, to go to the next line.

```

37a <the io specification 37a>≡ (37f)
 module io[X]
 <imports for io 37b>
 <sorts for io 37c>
 <global operators for io 37d>
 <rules for io 37e>
 end

```

This module imports the modules which define the basis of the input/output by calling `basio[X]` and `stdio`.

```

37b <imports for io 37b>≡ (37a)
 import global bool builtinInt stdio basio[X] basio[string];
 end

```

Uses `basio` 37f, `builtinInt` 23b 35 46f 101, and `string` 28d 101 101.

```

37c <sorts for io 37c>≡ (37a)
 sort X;
 end

```

We define here the common operator `writeln` which takes a pipe identifier of sort `Pid` where it will write and the element to write.

```

37d <global operators for io 37d>≡ (37a)
 operators global
 writeln(@,@) : (Pid X) X pri 200;
 end

```

Uses `Pid` 25d and `writeln` 37f.

`writeln` is defined by using the operator `write` described in the module `basio`.

```

37e <rules for io 37e>≡ (37a)
 rules for X
 x, x1:X; pid:Pid; str:string;
 global
 [] writeln(pid,x) => x
 where x1:=()write(pid,x)
 where str:=()write(pid,"\n")
 end
 end

```

Uses `Pid` 25d, `string` 28d 101 101, `write` 37f, and `writeln` 37f.

```

37f <* 37f>≡
 // This file is generated automatically: do not edit it directly.
 <the io specification 37a>

```

Defines:

`basio`, used in chunks 17a and 37b.  
`write`, used in chunks 17e, 37e, and 44e.  
`writeln`, used in chunk 37.

## 1.19 list.eln

This module presents the standard definition for list with strategies allowing to enumerate the list elements.

```

38a <the list speci cation 38a>≡ (40b)
 module list[X]
 <imports for list 38b>
 <sorts for list 38c>
 <operators for list 38d>
 <strategy operator for list 39d>
 <rules for list 39e>
 <strategies for list 40a>
 end

```

For representing list indexes and for computing list length, we need to use integers which are defined in the module `int` of the standard library:

```

38b <imports for list 38b>≡ (38a)
 import int;
 end

```

Uses int 35 46f 101.

The sorts of interest are `X` (given through the parameter) and `list[X]` :

```

38c <sorts for list 38c>≡ (38a)
 sort X list[X];
 end

```

```

38d <operators for list 38d>≡ (38a)
 operators global
 <nil 38e><cons 38f><append 38g><elem 38h><-th elem 39a><size 39b><ccat 39c>
 end

```

The empty list is denoted as usual `nil`. The operator `nil_X` is introduced in order to allow for disambiguation when necessary.

```

38e <nil 38e>≡ (38d)
 nil_X : list[X];
 nil : list[X] alias nil_X;;

```

There are four ways to use the `cons` operator, all are equivalent:

```

38f <cons 38f>≡ (38d)
 cons(@,@) : (X list[X]) list[X];
 @ @ : (X list[X]) list[X] alias cons(@,@)::;
 @ , @ : (X list[X]) list[X] alias cons(@,@)::;
 @ . @ : (X list[X]) list[X] alias cons(@,@)::;

```

The `append` operator could be used in prefix form like `append(L1,L2)` or in infix form like in `L1 @ L2`.

```

38g <append 38g>≡ (38d)
 append(@,@) : (list[X] list[X]) list[X];
 @ ' @ ' @ : (list[X] list[X]) list[X] alias append(@,@)::;

```

The `elem` operator returns one of the elements of the list given in argument like in `elem(L)`. It is used by the strategy `listExtract` via the labelled rewrite rules `extractrule1` and `extractrule2`.

```

38h <elem 38h>≡ (38d)
 elem(@) : (list[X]) X;

```

The `-th elem` operator computes the  $n$  th element of a list. For example `3-th elem(L)` returns the third element of `L`. One can check that no rule is applicable when the argument is out of range like in `2-th nil`.

```
39a <-th elem 39a>≡ (38d)
 @-th elem(@) : (int list[X]) X; /* n-th element of list */
```

Uses - 23b 35 and int 35 46f 101.

`size` returns the size of the list.

```
39b <size 39b>≡ (38d)
 size(@) : (list[X]) int;
 size_of_X_list(@) : (list[X]) int alias size(@);
```

Uses int 35 46f 101.

The concatenation operator `ccat` takes two syntactic forms `ccat(@*@)` and `ccat(@,@)` which are aliases and that allows to concatenate  $n$  times the same list like in `ccat(4,L)`. The rank of these operators are:

```
39c <ccat 39c>≡ (38d)
 ccat(@*@) : (int list[X]) list[X]; // concat n-times
 ccat(@,@) : (int list[X]) list[X] alias ccat(@*@);
```

Uses \* 23b 35, - 23b 35, and int 35 46f 101.

```
39d <strategy operator for list 39d>≡ (38a)
 stratop global
 listExtract : <X->X> bs;
 end
```

Uses < 20 23b 35 and > 20 23b 35.

```
39e <rules for list 39e>≡ (38a)
 rules for list[X]
 e : X;
 l,l1,l2 : list[X];
 global
 [] append(nil,l) => l end
 [] append(e.l1 , l2) => e . append(l1,l2) end
 end

 rules for X
 e,ee : X;
 l : list[X];
 n, n1: int;
 global
 [] 1-th elem(e.l) => e end
 [] n-th elem(e.l) => n1-th elem(l) where n1:=()n-1 end
 local
 [extractrule1] elem(e.l) => e end
 [extractrule2] elem(e.l) => elem(l) end
 end

 rules for list[X]
 n : int;
 l : list[X];
 global
 [] ccat(0,l) => nil end
 [] ccat(n,l) => l @ ccat(n-1,l) if n>0 end
 end

 rules for int
```



```

e : X;
l : list[X];
global
 [] size(nil) => 0 end
 [] size(e.l) => 1+size(l) end
end

```

Uses + 23b 35, - 23b 35, > 20 23b 35, and int 35 46f 101.

```

40a <strategies for list 40a>≡ (38a)
 strategies for X
 implicit
 // [.] listExtract => iterate*(dc(extractrule2)) ; dc(extractrule1) end
 [.] listExtract => iterate*(dc one(extractrule2)) ; dc one(extractrule1) end
end

```

Uses \* 23b 35.

```

40b <* 40b>≡
 // This file is generated automatically: do not edit it directly.
 <the list specification 38a>

```

**1.20 occur.eln**

41a *<the occur specification 41a>*≡ (41e)

```

module occur[X,Y]
 // The parameters X and Y must be non built-in
 <imports for occur 41b>
 <sorts for occur 41c>
 <operators for occur 41d>
end

```

Uses - 23b 35 and and 20 88.

41b *<imports for occur 41b>*≡ (41a)

```

import bool;
end

```

41c *<sorts for occur 41c>*≡ (41a)

```

sort X Y;
end

```

41d *<operators for occur 41d>*≡ (41a)

```

operators
 global // check if the first term occurs in the second one
 occurs @ in @ : (X Y) bool code 17;
 // Parenthesised version
 occurs(@,@) : (X Y) bool alias occurs @ in @ ;;
end

```

Uses first 43c and second 43c.

41e *<\* 41e>*≡

```

// This file is generated automatically: do not edit it directly.
<the occur specification 41a>

```

## 1.21 pair.eln

This module defines a particular tuple currently used: the pair. It defines the associated sort and two primitives giving access to the components of the pair.

This module has two parameters: X and Y that are the sorts of respectively the first and the second element of the pair.

```
42a <the pair specification 42a>≡ (43c)
 module pair[X,Y]
 <imports for pair 42b>
 <sorts for pair 42c>
 <global operators for pair 42d>
 <rules for pair 43a>
 end
```

Uses pair 43c.

```
42b <imports for pair 42b>≡ (42a)
 import bool;
 end
```

The new sort pair[X,Y] is declared.

```
42c <sorts for pair 42c>≡ (42a)
 sort X Y pair[X,Y];
 end
```

Uses pair 43c.

The first operator [,] is the constructor of a pair.

```
42d <global operators for pair 42d>≡ (42a) <42e>
 operators
 global
 [,@] : (X Y) pair[X,Y];
 Uses pair 43c.
```

first and second are the operators that allows us to have access to the first and the second component of the given pair.

```
42e <global operators for pair 42d>+≡ (42a) <42d 42f>
 first(@) : (pair[X,Y]) X;
 1-th (@) : (pair[X,Y]) X alias first(@);

 second(@) : (pair[X,Y]) Y;
 2-th (@) : (pair[X,Y]) Y alias second(@);
```

Uses - 23b 35, first 43c, pair 43c, and second 43c.

ispair tests if the argument is of sort pair[X,Y].

```
42f <global operators for pair 42d>+≡ (42a) <42e>
 ispair(@) : (pair[X,Y]) bool;

 end
```

Uses ispair 43c and pair 43c.

```

43a <rules for pair 43a>≡ (42a) 43b>
 rules for X
 x : X;
 y : Y;
 global
 [] first([x,y]) => x end
 end

 rules for Y
 x : X;
 y : Y;
 global
 [] second([x,y]) => y end
 end

```

Uses `first` 43c and `second` 43c.

The two last rules define the operator `ispair`. Note that they are applied in the order of the declaration during the evaluation.

```

43b <rules for pair 43a>+≡ (42a) <43a
 rules for bool
 x : X;
 y : Y;
 p : pair[X,Y];
 global
 [] ispair([x,y]) => true end
 [] ispair(p) => false end
 end

```

Uses `false` 20, `ispair` 43c, `pair` 43c, and `true` 20.

```

43c <* 43c>≡
 // This file is generated automatically: do not edit it directly.
 <the pair specification 42a>

```

Defines:

- `first`, used in chunks 41–43 and 45c.
- `ispair`, used in chunks 42f and 43b.
- `pair`, used in chunks 42 and 43b.
- `second`, used in chunks 41–43 and 45c.

## 1.22 prompt.eln

- 44a *<the prompt specification 44a>*≡ (44f)
- ```

module prompt[X]
  <imports for prompt 44b>
  <operators for prompt 44c>
  <rules for prompt 44e>
end

```
- 44b *<imports for prompt 44b>*≡ (44a)
- ```

import local io[string] io[X];
end

```
- Uses string 28d 101 101.
- 44c *<operators for prompt 44c>*≡ (44a)
- ```

<global operators for prompt 44d>

```
- 44d *<global operators for prompt 44d>*≡ (44c)
- ```

operators global
 read : X;
 print(@) : (X) X;
 println(@) : (X) X;
 prompt(@) : (string) X;
end

```
- Uses string 28d 101 101.
- 44e *<rules for prompt 44e>*≡ (44a)
- ```

rules for X
pr, pr1 : string;
t, t1   : X;
global
  [] read => read(stdin)           end
  [] print(t) => write(stdout,t)   end
  [] println(t) => t1   where t1:=()write(stdout,t)
                               where pr:=()write(stdout,"\n")   end
  [] prompt(pr) => read(stdin) where pr1:=()write(stdout,pr)   end
end

```
- Uses stdin 25d, stdout 25d, string 28d 101 101, and write 37f.
- 44f *<* 44f>*≡
- ```

// This file is generated automatically: do not edit it directly.
<the prompt specification 44a>

```

### 1.23 replace.eln

45a *<the replace specification 45a>*≡ (45d)

```

module replace[X,Y]
 // The parameters X and Y must be non built-in
 <sorts for replace 45b>
 <operators for replace 45c>
end

```

Uses - 23b 35 and and 20 88.

45b *<sorts for replace 45b>*≡ (45a)

```

sort X Y;
end

```

45c *<operators for replace 45c>*≡ (45a)

```

operators
global

 // replace all occurrences of the first argument by the
 // second argument in the third argument
 replace @ by @ in @ : (X X Y) Y code 16;

 // syntactic variants
 replace (@) by (@) in (@) : (X X Y) Y alias replace @ by @ in @ ;;
 replace(@,@,@) : (X X Y) Y alias replace @ by @ in @ ;;

end

```

Uses first 43c and second 43c.

45d *<\* 45d>*≡

```

// This file is generated automatically: do not edit it directly.
<the replace specification 45a>

```

## 1.24 stdio.eln

This module redefines the operator `putc` already defined in the imported module `builtinStdio` but here, the sort of the argument is `int` and not `builtinInt` as in the module `builtinStdio`.

```
46a <the stdio specification 46a>≡ (46f)
 module stdio
 <imports for stdio 46b>
 <sorts for stdio 46c>
 <global operators for stdio 46d>
 <rules for stdio 46e>
 end
```

```
46b <imports for stdio 46b>≡ (46a)
 import global builtinStdio int bool string;
 end
```

Uses `builtinStdio` 46f, `int` 35 46f 101, and `string` 28d 101 101.

```
46c <sorts for stdio 46c>≡ (46a)
 sort Pid;
 end
```

Uses `Pid` 25d.

The signature of the operator `putc` is different: we use now the sort `int` and not the built-in sort `builtinInt`. This operator puts a character (the second argument of sort `int` converted to the corresponding character) to an output file or pipe.

```
46d <global operators for stdio 46d>≡ (46a)
 operators global
 putc(@,@) : (Pid int) int ;
 end
```

Uses `int` 35 46f 101, `Pid` 25d, and `putc` 25d 46f.

This rule makes the conversion from an integer to the sort `builtinInt` and then calls the operator `intern` `putc` defined in `builtinStdio`.

```
46e <rules for stdio 46e>≡ (46a)
 rules for int
 p : Pid;
 x : int;
 bix : builtinInt;
 global
 [] putc(p,x) => intern putc(p,bix) where bix:=()valueOf(x) end
 end
```

Uses `builtinInt` 23b 35 46f 101, `int` 35 46f 101, `Pid` 25d, `putc` 25d 46f, and `valueOf` 35.

```
46f <* 46f>≡
 // This file is generated automatically: do not edit it directly.
 <the stdio specification 46a>
```

Defines:

- `builtinInt`, used in chunks 5–7, 17, 18a, 21–25, 27, 28, 33, 34, 37b, 46e, and 47d.
- `builtinStdio`, used in chunks 24a and 46b.
- `int`, used in chunks 9a, 11, 15, 33, 34, 36, 38, 39, 46, 99b, 47, 98b, 49, 98d, and 51b.
- `putc`, used in chunks 24e and 46.

## 1.25 string.eln

This module redefines the operators `s[i]`, `s[i<-c]`, `substr` and `string` already defined in the module `builtinString` but for the sort `builtinInt` instead of the sort `int` used here.

```
99a <the string speci cation 99a>≡ (101)
 module string
 <imports for string 99b>
 <global operators for string 47c>
 <rules for string 47d>
 end
```

Uses `string` 28d 101 101.

```
99b <imports for string 99b>≡ (99a)
 import global builtinString int;
 end
```

Uses `builtinString` 101 and `int` 35 46f 101.

We have the same operator definitions than in the module `builtinString` but the sort used here is `int`. To find the definitions of these four operators, we suggest to read the explanations given in the module `builtinString`.

```
47c <global operators for string 47c>≡ (99a)
 operators global
 @[@] : (string int) int pri 500;
 @[@<-@] : (string int int) string pri 500;
 substr(@,@,@) : (string int int) string pri 500;
 string(@) : (int) string pri 500;
 end
```

Uses `@[@<-@]` 101, `@[@]` 101, `[@]` 35, `int` 35 46f 101, `string` 28d 101 101, and `substr` 28d 101.

The rules for defining these operators consist in transforming each element of sort `int` in an element of sort `builtinInt` with the operator `valueOf` defined in the module `int`. Then, with the correct signature, we can use the corresponding operators given in `builtinString` and prefixed by `intern`.

```
47d <rules for string 47d>≡ (99a)
 rules for int
 s : string;
 x : int;
 bix : builtinInt;
 global
 [] s[x] => intern s[bix]
 where bix:=() valueOf(x)
 end
end
```

```
rules for string
 s : string;
 x,y : int;
 bix,biy : builtinInt;
global
 [] s[x<-y] => intern s[bix<-biy]
 where bix:=() valueOf(x)
 where biy:=() valueOf(y)
```



```

end
[] substr(s,x,y) => intern substr(s,bix,biy)
 where bix:=() valueOf(x)
 where biy:=() valueOf(y)
end
[] string(x) => intern string(bix) where bix:=() valueOf(x) end
end

```

Uses builtinInt 23b 35 46f 101, int 35 46f 101, string 28d 101 101, substr 28d 101, and valueOf 35.

```

101 < * 101 >≡
 // This file is generated automatically: do not edit it directly.
 <the string speci cation 99a>
Defines:
@[<-@], used in chunks 11e, 15, 27g, and 47c.
@[@], used in chunks 5e, 11e, 15, 27f, and 47c.
builtinInt, used in chunks 5-7, 17, 18a, 21-25, 27, 28, 33, 34, 37b, 46e, and 47d.
builtinString, used in chunks 24b, 27a, and 99b.
int, used in chunks 9a, 11, 15, 33, 34, 36, 38, 39, 46, 99b, 47, 98b, 49, 98d, and 51b.
string, used in chunks 24, 27, 28, 37, 44, 46b, 99a, 47, 98b, 49, and 98d.
substr, used in chunks 27h and 47.

```

## 1.26 strlist.eln

This module gives two operators to translate a list of integers into a string and vice versa.

```
98a <the strlist specification 98a>≡ (98e)
 module strlist
 <imports for strlist 98b>
 <operators for strlist 49c>
 <rules for strlist 98d>
 end
```

```
98b <imports for strlist 98b>≡ (98a)
 import global string int list[int];
 end
```

Uses `int` 35 46f 101 and `string` 28d 101 101.

The two global operators defined here are `explode` and `implode`. `explode` takes a string and converts it into a list of integers. `implode` makes the opposite translation. For instance, the string "ABBA" is translated by `explode` into 65.66.66.65.nil and the reversed operation is done by `implode`.

```
49c <operators for strlist 49c>≡ (98a)
 <global operators for strlist 49d>
 <local operators for strlist 49e>
```

```
49d <global operators for strlist 49d>≡ (49c)
 operators global
 explode(@) : (string) list[int];
 implode(@) : (list[int]) string;
```

Uses `explode` 98e, `implode` 98e, `int` 35 46f 101, and `string` 28d 101 101.

This local operator `explode` with three arguments is only used by the `explode` operator with one argument which is exported.

```
49e <local operators for strlist 49e>≡ (49c)
 local
 explode(@,@,@) : (string int list[int]) list[int];
 end
```

Uses `explode` 98e, `int` 35 46f 101, and `string` 28d 101 101.

The definitions consists in exploring the list of integers element by element or the string character by character and in doing the good conversion (string into integer or vice versa).

```
98d <rules for strlist 98d>≡ (98a)
 rules for list[int]
 s : string;
 ls : list[int];
 i : int;
 global
 [] explode(s) => explode(s,strlen(s),nil) end
 [] explode(s,0,ls) => ls end
 [] explode(s,i,ls) => explode(s,i-1,s[i-1].ls) end
 end

 rules for string
```

```
ls : list[int];
i : int;
global
 [] implode(nil) => "" end
 [] implode(i.ls) => string(i)+implode(ls) end
end
```

Uses + 23b 35, - 23b 35, explode 98e, implode 98e, int 35 46f 101, string 28d 101 101, and strlen 28d.

```
98e <* 98e>≡
 // This file is generated automatically: do not edit it directly.
 <the strlist speci cation 98a>
Defines:
 explode, used in chunks 49 and 98d.
 implode, used in chunks 49d and 98d.
```

## 1.27 tuple.eln

This module defines the sort `tuple` and the access operators to each element of the `tuple`.

51a *<the tuple specification 51a>*≡ (51f)

```

module tuple[N,T_1,...,T_N]
 <imports for tuple 51b>
 <sorts for tuple 51c>
 <global operators for tuple 51d>
 <rules for tuple 51e>

end

```

Uses `tuple` 51f.

51b *<imports for tuple 51b>*≡ (51a)

```

import global int;
end

```

Uses `int` 35 46f 101.

51c *<sorts for tuple 51c>*≡ (51a)

```

sort {T_I }_I=1...N tuple[T_1,...,T_N];
end

```

Uses `tuple` 51f.

The operators are defined using the pre-processing syntax for duplication.

51d *<global operators for tuple 51d>*≡ (51a)

```

operators global
 [@ {,@}_I=2...N] : ({T_I}_I=1...N) tuple[T_1,...,T_N];
 {
 I-th(@) : (tuple[T_1,...,T_N]) T_I;
 }_I=1...N
end

```

Uses - 23b 35 and `tuple` 51f.

51e *<rules for tuple 51e>*≡ (51a)

```

{rules for T_I
 { x_J : T_J; }_J=1...N
 global
 [] I-th([x_1 {, x_J}_J=2...N]) => x_I end
 end
}_I=1...N

```

Uses - 23b 35.

51f *<\* 51f>*≡

```

// This file is generated automatically: do not edit it directly.
<the tuple specification 51a>

```

Defines:

`tuple`, used in chunk 51.

## 2 Library noquote

## 2.1 applySubstOn.eln

This module specifies substitutions applicable on terms of the sort `X` given as parameter.

```
53a <the applySubstOn specification 53a>≡ (54)
 module applySubstOn[Fss,Vars,X]
 <imports for applySubstOn 53b>
 <sorts for applySubstOn 53c>
 <global operators for applySubstOn 53d>
 <rules for applySubstOn 53e>
 end
```

We have to import the module `termCommons[Fss,Vars]` for obtaining the definitions for the sorts variable and term. The substitutions to be applied are of the form defined in the module `substitution` and the replacement of terms in terms of sort `X` is defined in the module `replace[term,X]`.

```
53b <imports for applySubstOn 53b>≡ (53a)
 import local
 termCommons[Fss,Vars]
 replace[term,X]
 substitution
 ;
 end
```

Uses `substitution 54` and `term 75c`.

The sort `X` is defined in the module:

```
53c <sorts for applySubstOn 53c>≡ (53a)
 sort X;
 end
```

As unique operator we have the application of a substitution and its alias.

```
53d <global operators for applySubstOn 53d>≡ (53a)
 operators
 global
 @(@) : (substitution X) X;
 apply(@,@) : (substitution X) X alias @(@):;
 end
```

Uses `apply 54 59d` and `substitution 54`.

The rules describe the application of the `identity` substitution, of the composition of substitutions or of a basic substitution. The replacement of a term (variable) by another term is a built-in operation defined in the module `replace`.

```
53e <rules for applySubstOn 53e>≡ (53a)
 rules for X
 $tt : X;
 $s1,$s2 : substitution;
 $v : variable;
 $t : term;

 global
 [] apply(identity,$tt) => $tt end
 [] apply($s1 o $s2 , $tt) => apply($s2,apply($s1,$tt)) end
 [] apply($v->$t,$tt) => replace($v,$t,$tt) end
 end
```

Uses `-> 54 59d 65d`, `apply 54 59d`, `identity 32e 54 59d 65d`, `o 54 59d 65d`, `substitution 54`, `term 75c`, and `variable 75c`.

```
54 ⟨ * 54 ⟩ ≡
 // This file is generated automatically: do not edit it directly.
 ⟨ the applySubstOn specification 53a ⟩
```

Defines:

- >, used in chunks 53e, 59, 65c, 68, and 75a.
- apply, used in chunks 53, 59b, and 68.
- identity, used in chunks 30d, 32, 53e, 59a, 65c, and 75a.
- o, used in chunks 53e, 59a, 65c, and 75a.
- substitution, used in chunks 53, 58d, 59a, 65, 71, and 75a.

## 2.2 atom.eln

This module defines some notions common to all first order atoms. This module is used for example for the case when one uses two kinds of atoms built on different signatures, but defined both by the module `atomP`.

```
55a <the atom speci cation 55a>≡ (55e)
 module atom[Fss,Vars]
 <imports for atom 55b>
 <sorts for atom 55c>
 <global operators for atom 55d>
 end
Uses atom 55e 57.
```

The terms defined in `termCommons[Fss,Vars]` are used locally and the equality between atoms can be used in any module that imports this one.

```
55b <imports for atom 55b>≡ (55a)
 import global eq[atom];
 local int termCommons[Fss,Vars] list[atom];
 end
Uses atom 55e 57.
```

The sort `atom` is defined here and any module that uses atoms should include this one.

```
55c <sorts for atom 55c>≡ (55a)
 sort atom;
 end
Uses atom 55e 57.
```

The operators are defined globally for the other particular importations of the module `atomP`.

```
55d <global operators for atom 55d>≡ (55a)
 operators
 global

 head(@) : (atom) identifier;
 @-th subterm(@) : (int atom) term;

 end
Uses atom 55e 57, head 55e 57 75c, identifier 63e, term 75c, and -th 55e 57 75c.
```

```
55e <* 55e>≡
 // This file is generated automatically: do not edit it directly.
 <the atom speci cation 55a>
```

Defines:

`atom`, used in chunks 55, 56, and 61.  
`head`, used in chunks 55d, 56d, 68, 71e, 74b, and 75a.  
`-th`, used in chunks 55d, 56d, 71e, 74a, and 75a.



## 2.3 atomP.eln

This module is very similar to the module `termCommons [Fss,Vars]`. The argument `Preds` defines the predicate symbols used for building atoms.

```
56a <the atomP specification 56a>≡ (57)
 module atomP [Fss,Vars,Preds]
 <imports for atomP 56b>
 <global operators for atomP 56c>
 <rules for atomP 56d>
 end
```

We should import the definitions for terms and the global definitions for atoms.

```
56b <imports for atomP 56b>≡ (56a)
 import
 global termCommons [Fss,Vars]
 atom [Fss,Vars]
 ;
 local Preds int identifier
 pair [identifier,int]
 list [pair [identifier,int]]
 ;
 end
```

Uses `atom 55e 57` and `identifier 63e`.

The operators are similar to the ones for the terms but this time the predicate symbols are used and we define atoms instead of terms.

```
56c <global operators for atomP 56c>≡ (56a)
 operators
 global

 FOR EACH SS:pair [identifier,int]; P:identifier; N:int
 SUCH THAT SS:=(listExtract) elem (Preds) AND P:=()first (SS)
 AND N:=()second (SS) ANDIF N==0 :{
 P
 : atom;
 P()
 : atom alias P;;
 }

 FOR EACH SS:pair [identifier,int]; P:identifier; N:int
 SUCH THAT SS:=(listExtract) elem (Preds) AND P:=()first (SS)
 AND N:=()second (SS) ANDIF N > 0 :{

 P(@ {,@}~(N-1)) :({term}~N) atom;
 P({,@}~N) : ({term}~N)atom alias P(@{,@}~(N-1));;
 P({@,}~N) : ({term}~N)atom alias P(@{,@}~(N-1));;
 P(,{@,}~N): ({term}~N)atom alias P(@{,@}~(N-1));;
 }
 end
```

Uses `== 20 23b 31d 35 62e 88`, `atom 55e 57`, `identifier 63e`, and `term 75c`.

The rules complete the ones for the terms when dealing with atoms.

```
56d <rules for atomP 56d>≡ (56a)
 FOR EACH SS:pair [identifier,int]; P:identifier; N:int
 SUCH THAT SS:=(listExtract) elem (Preds) AND P:=()first (SS)
 AND N:=()second (SS) ANDIF N > 0 :{
```

```

rules for term
 t_1,...,t_N : term;
 global

 {
 [] I-th subterm(P(t_1,...,t_N)) => t_I
 end
 }_I=1...N

end
}

FOR EACH SS:pair[identifier,int]; P:identifier; N:int
SUCH THAT SS:=(listExtract) elem(Preds) AND P:=()first(SS)
AND N:=()second(SS) :{

 rules for identifier
 t_1,...,t_N : term;
 global
 [] head(P(t_1,...,t_N)) => P end
 end
}

```

Uses = 59d, head 55e 57 75c, identifier 63e, term 75c, and -th 55e 57 75c.

57 < \* 57 > ≡  
 // This file is generated automatically: do not edit it directly.  
 < the atomP specification 56a >

Defines:

atom, used in chunks 55, 56, and 61.  
 head, used in chunks 55d, 56d, 68, 71e, 74b, and 75a.  
 -th, used in chunks 55d, 56d, 71e, 74a, and 75a.

## 2.4 eqSystem.eln

The module defines equations and systems of equations.

58a *(the eqSystem speci cation 58a)*≡ (59d)

```

module eqSystem[Fss,Vars]
 <imports for eqSystem 58b>
 <sorts for eqSystem 58c>
 <global operators for eqSystem 58d>
 <strategy operators for eqSystem 58e>
 <rules for eqSystem 59a>
 <strategies for eqSystem 59c>

end

```

Uses eqSystem 59d.

58b *(imports for eqSystem 58b)*≡ (58a)

```

import global applySubstOn[Fss,Vars,eqSystem];
local bool int termCommons[Fss,Vars];

end

```

Uses eqSystem 59d.

The new sorts introduced by the module are the equations (`equation`) and the systems of equations (`eqSystem`):

58c *(sorts for eqSystem 58c)*≡ (58a)

```

sort eqSystem equation;

end

```

Uses eqSystem 59d and equation 59d.

An equation is built using the = operator that takes as arguments two terms. A system of equations is either a solved system, that is a boolean value, a single equation or a conjunction (&) of equations.

A system of equations can be transformed into a substitution by using the operator `system_to_subst`.

58d *(global operators for eqSystem 58d)*≡ (58a)

```

operators
global
 @ = @ : (term term) equation;

 @ : (bool) eqSystem;
 @ : (equation) eqSystem;
 @ & @ : (eqSystem eqSystem) eqSystem assocLeft;
 (@ & @) : (eqSystem eqSystem) eqSystem alias @ & @:;

 system_to_subst(@) : (eqSystem) substitution;

end

```

Uses & 23b 35 59d, = 59d, eqSystem 59d, equation 59d, substitution 54, system\_to\_subst 59d, and term 75c.

We have a strategy that simplifies systems of equations by eliminating variables from the respective systems.

58e *(strategy operators for eqSystem 58e)*≡ (58a)

```

stratop
 global
 delete_new_variables : <eqSystem> bs;

 end

```

Uses delete\_new\_variables 59d and eqSystem 59d.

A trivial system of equalities (`true`) is transformed into a trivial substitution (`identity`). A system containing only one equation having as left-hand side a variable is immediately transformed into the corresponding substitution. A system with several equations is recursively transformed.

```
59a <rules for eqSystem 59a>≡ (58a) 59b>
 rules for substitution
 P,Q,R : eqSystem;
 $t : term;
 $x : variable;
 global
 [] system_to_subst(true) => identity end
 [] system_to_subst($x=$t) => $x->$t end
 [] system_to_subst(P & Q) => system_to_subst(P) o system_to_subst(Q)
 end
end
```

Uses & 23b 35 59d, -> 54 59d 65d, = 59d, eqSystem 59d, identity 32e 54 59d 65d, o 54 59d 65d, substitution 54, system\_to\_subst 59d, term 75c, and variable 75c.

The rules used in the solving strategy

```
59b <rules for eqSystem 59a>+≡ (58a) <59a
 rules for eqSystem
 P,Q,R : eqSystem;
 n : int;
 $t : term;
 e : equation;
 global
 [new_var_elim] P & var(n) = $t => apply(var(n)->$t,P) end
 [unfold] P & (Q & R) => (P & Q) & R end
 [eqPass] P & e => e & P end
 end
```

Uses & 23b 35 59d, -> 54 59d 65d, = 59d, apply 54 59d, eqSystem 59d, equation 59d, and term 75c.

The strategy that simplifies the system of equations tries to eliminate equations that have a variable as left-hand side of the equality. For this we `unfold` the system of equations and if we find an equation with a variable in the left-hand side we use the rule `new_var_elim` that applies the corresponding substitution on the rest of the system and eliminates the equation from the system. When the equation does not have a variable in the left-hand side we use the rule `eqPass`.

```
59c <strategies for eqSystem 59c>≡ (58a)
 strategies for eqSystem
 implicit
 [.] delete_new_variables => repeat*(dc one(new_var_elim, unfold, eqPass))
 end
end
```

Uses `delete_new_variables` 59d and `eqSystem` 59d.

```
59d <* 59d>≡
 // This file is generated automatically: do not edit it directly.
 <the eqSystem specification 58a>
```

Defines:

- &, used in chunks 21d, 22a, 33e, 34b, 58, 59, and 68.
- >, used in chunks 53e, 59, 65c, 68, and 75a.
- =, used in chunks 56d, 58, 59, 67a, 68, and 74.

apply, used in chunks 53, 59b, and 68.  
delete\_new\_variables, used in chunks 58e and 59c.  
eqSystem, used in chunks 58, 59, and 66–69.  
equation, used in chunks 58, 59b, and 67a.  
identity, used in chunks 30d, 32, 53e, 59a, 65c, and 75a.  
o, used in chunks 53e, 59a, 65c, and 75a.  
system\_to\_subst, used in chunks 58d and 59a.

## 2.5 hornClauseSyntax.eln

The module defines Horn clauses of the form  $A \vdash B$  with  $A$  an atom and  $B$  a list of atoms.

```
61a <the hornClauseSyntax speci cation 61a>≡ (61d)
 module hornClauseSyntax[Fss,Vars]
 <imports for hornClauseSyntax 61b>
 <global operators for hornClauseSyntax 61c>
 end
```

```
61b <imports for hornClauseSyntax 61b>≡ (61a)
 import atom[Fss,Vars] list[atom] pair[atom,list[atom]]
 list[pair[atom,list[atom]]];
 end
Uses atom 55e 57.
```

A new possible representation for the lists of atoms is introduced: the `,` is an alias for the constructor `cons` and the symbol `.` represents the `nil` list. Using the lists of atoms we define the sequents constructed with the symbol `:-` and the lists of sequents that are consecutive sequents.

```
61c <global operators for hornClauseSyntax 61c>≡ (61a)
 operators
 global

 . : list[atom] alias nil;;
 @,@ : (atom list[atom]) list[atom] alias cons(@,@)::;

 @ @ : (atom list[atom]) pair[atom,list[atom]] alias [@,@]::;
 @ ':- @ : (atom list[atom]) pair[atom,list[atom]] alias [@,@]::;

 Epsilon : list[pair[atom,list[atom]]] alias nil;;
 @ @ : (pair[atom,list[atom]] list[pair[atom,list[atom]]])
 list[pair[atom,list[atom]]] alias cons(@,@)::;

 end
Uses ':- 61d and atom 55e 57.
```

```
61d <* 61d>≡
 // This file is generated automatically: do not edit it directly.
 <the hornClauseSyntax speci cation 61a>
```

Defines:

':-, used in chunk 61c.

## 2.6 ident.eln

In this module are defined the sort `ident` and the equalities and inequalities between terms of this sort.

```
62a <the ident speci cation 62a>≡ (62e)
 module ident
 <imports for ident 62b>
 <sorts for ident 62c>
 <global operators for ident 62d>
 end
 Uses ident 62e.
```

The module `anyIdentifier` is a built-in module that defines the built-in identifiers.

```
62b <imports for ident 62b>≡ (62a)
 import global anyIdentifier;
 local bool;
 end
```

```
62c <sorts for ident 62c>≡ (62a)
 sort ident;
 end
 Uses ident 62e.
```

The equality (`==`) and inequality (`!=`) between two terms of sort `ident` are built-in operators and both of them return a boolean as result. If we consider two identical terms `s,t` we obtain `true` as result for (`s == t`) and `false` for (`s != t`).

The alias `eq_ident` is defined for the operator `==` and the alias `neq_ident` is defined for the operator `!=`.

```
62d <global operators for ident 62d>≡ (62a)
 operators
 global

 @ == @ : (ident ident) bool code 14;
 @ != @ : (ident ident) bool code 15;

 eq_ident(@,@) : (ident ident) bool alias @ == @:;
 neq_ident(@,@) : (ident ident) bool alias @ != @:;

 end
 Uses != 20 23b 31d 35 62e 88, == 20 23b 31d 35 62e 88, eq_ident 62e, ident 62e, and neq_ident 62e.
```

```
62e <* 62e>≡
 // This file is generated automatically: do not edit it directly.
 <the ident speci cation 62a>
```

Defines:

`!=`, used in chunks 19e, 22b, 30, 31c, 33e, 34b, 62d, 74b, and 87.  
`==`, used in chunks 15f, 19e, 22b, 30, 31c, 33e, 34b, 56c, 62d, 72b, and 87.  
`eq_ident`, used in chunk 62d.  
`ident`, used in chunks 62 and 63.  
`neq_ident`, used in chunk 62d.

## 2.7 identifier.eln

This module is just a wrapping for the module `ident`.

63a *<the identifier specification 63a>*≡ (63e)

```

module identifier
 <imports for identifier 63b>
 <sorts for identifier 63c>
 <global operators for identifier 63d>

end

```

Uses `identifier` 63e.

63b *<imports for identifier 63b>*≡ (63a)

```

import
global ident
 eq[identifier]
;

end

```

Uses `ident` 62e and `identifier` 63e.

63c *<sorts for identifier 63c>*≡ (63a)

```

sort identifier;
end

```

Uses `identifier` 63e.

63d *<global operators for identifier 63d>*≡ (63a)

```

operators
global

 @ : (ident) identifier pri 0;

end

```

Uses `ident` 62e and `identifier` 63e.

63e *<\* 63e>*≡  
*// This file is generated automatically: do not edit it directly.*  
*<the identifier specification 63a>*

Defines:

`identifier`, used in chunks 55, 56, 63, 64, 66b, 68, and 71–75.



## 2.8 sigSyntax.eln

This module describes a possible syntax for the user specification. If a different syntax is needed by the user another similar module should be created.

```
64a <the sigSyntax speci cation 64a>≡ (64d)
 module sigSyntax
 <imports for sigSyntax 64b>
 <global operators for sigSyntax 64c>
 end
```

Usually a specification is described using identifiers (`identifier`) and several structures (`list,pair`). The corresponding modules (like `int` in our case) should be imported if different sorts are used in describing the syntax of the specification file.

```
64b <imports for sigSyntax 64b>≡ (64a)
 import
 global identifier int;
 local identifier list[identifier] pair[identifier,int]
 list[pair[identifier,int]];
 end
 Uses identifier 63e.
```

In the specification file whose syntax is described in this module, we can have pairs of an identifier and an integer of the form `a : n` where `a` is an identifier and `n` an integer. We can have as well lists of identifiers or lists of pairs of the form described above. The elements of the lists are separated by the standard separators: space or tab.

```
64c <global operators for sigSyntax 64c>≡ (64a)
 operators
 global

 @ ':' @ : (identifier int) pair[identifier,int] alias [,@, @]::;

 Epsilon : list[identifier] alias nil::;
 @ @ : (identifier list[identifier]) list[identifier]
 alias cons(,@, @)::;

 Epsilon : list[pair[identifier,int]] alias nil::;
 @ @ : (pair[identifier,int] list[pair[identifier,int]])
 list[pair[identifier,int]] alias cons(,@, @)::;

 end
 Uses identifier 63e.
```

```
64d <* 64d>≡
 // This file is generated automatically: do not edit it directly.
 <the sigSyntax speci cation 64a>
```

## 2.9 substitution.eln

This module defines the substitutions and it should be imported by any module defining operations with substitutions. The application of a substitution on terms of the sort  $X$  is defined in the module `applySubstOn`.

```
65a <the substitution specification 65a>≡ (65d)
 module substitution
 <sorts for substitution 65b>
 <global operators for substitution 65c>
 end
 Uses substitution 54.
```

The sort `substitution` is introduced and the sorts `variable` and `term` are used for defining different forms of substitutions.

```
65b <sorts for substitution 65b>≡ (65a)
 sort substitution variable term;
 end
 Uses substitution 54, term 75c, and variable 75c.
```

The trivial substitution is denoted by `identity`. A basic substitution is defined using the operator `->` and has as arguments the variable to be substituted and the term that substitutes to it. Several substitutions can be composed using the operator `o` and the result is a substitution.

```
65c <global operators for substitution 65c>≡ (65a)
 operators
 global

 identity : substitution;
 @ -> @ : (variable term) substitution;
 @ o @ : (substitution substitution) substitution assocLeft;

 end
 Uses -> 54 59d 65d, identity 32e 54 59d 65d, o 54 59d 65d, substitution 54, term 75c, and variable 75c.
```

```
65d <* 65d>≡
 // This file is generated automatically: do not edit it directly.
 <the substitution specification 65a>
```

Defines:

`->`, used in chunks 53e, 59, 65c, 68, and 75a.  
`identity`, used in chunks 30d, 32, 53e, 59a, 65c, and 75a.  
`o`, used in chunks 53e, 59a, 65c, and 75a.

## 2.10 syntacticUnification.eln

The module defines the rules and strategies needed for solving matching and unification problems between terms built using the signature **Fss** and the variables from **Vars**.

```
66a <the syntacticUnification specification 66a>≡ (70)
 module syntacticUnification[Fss,Vars]
 <imports for syntacticUnification 66b>
 <operators for syntacticUnification 66c>
 <strategy operators for syntacticUnification 67c>
 <rules for syntacticUnification 68>
 <strategies for syntacticUnification 69>
 end
```

All the definitions concerning the construction of terms using the respective signature and set of variables should be imported.

```
66b <imports for syntacticUnification 66b>≡ (66a)
 import global termCommons[Fss,Vars];
 local Fss int identifier bool pair[identifier,int]
 list[pair[identifier,int]] eq[variable] eq[term]
 eqSystem[Fss,Vars]
 occur[variable,term]
 occur[variable,eqSystem]
 occur[bool,eqSystem]
 ;
 end
```

Uses `eqSystem` 59d, `identifier` 63e, `term` 75c, and `variable` 75c.

Two types of operators are defined: global operators that can be imported by other modules and local operators that are used in the definitions of the global ones.

```
66c <operators for syntacticUnification 66c>≡ (66a)
 operators
 <global operators for syntacticUnification 67a>
 <local operators for syntacticUnification 67b>
 end
```

67a *(global operators for syntacticUnification 67a)*≡ (66c)

```

global
 @ = @ : (term term) equation alias @=@:;
 @ : (equation) eqSystem alias @:;

```

Uses = 59d, eqSystem 59d, equation 59d, and term 75c.

67b *(local operators for syntacticUnification 67b)*≡ (66c)

```

local
 mergingClash(@,@,@) : (eqSystem variable term) bool;

```

Uses eqSystem 59d, mergingClash 70, term 75c, and variable 75c.

The strategy `match` is proposed for solving a system of matching problems and the strategy `unify` is used for solving a system of unification problems.

67c *(strategy operators for syntacticUnification 67c)*≡ (66a)

```

stratop
 global
 unify : <eqSystem> bs;
 unifys : <eqSystem> bs;
 match : <eqSystem> bs;
 matchs : <eqSystem> bs;
 end

```

Uses eqSystem 59d, match 70, and unify 70 151.

The rules are used in order to **delete** some redundant equations, to **coalesce** equations, to solve any **conflict**, to verify inappropriate occurrences of variables (**occCheck**), to **eliminate** variables from equations and to do some auxiliary operations on the systems of equations.

```

68 <rules for syntacticUnification 68>≡ (66a)
 rules for eqSystem
 P,Q,R:eqSystem;
 $s,$t:term;
 $x,$y:variable;
 b : bool;
 global

 [delete] P & $x=$y => P
 if eq_variable($x,$y)
 end

 [delete] P & true => P
 if occurs true in P
 end

 [coalesce] P & $x=$y => $x=$y & apply($x->$y,P)
 if neq_variable($x,$y)
 end

 [conflict] P & $s=$t => false
 if neq_identifier(head($s),head($t))
 and not(isvar($s)) and not(isvar($t))
 end

 [occCheck1] P & $x=$s => false
 if occurs $x in $s and not isvar($s)
 end

 [occCheck2] P & $s=$x => false
 if occurs $x in $s and not isvar($s)
 end

 [eliminate1] P & $x=$s => $x=$s & apply($x->$s,P)
 if not(occurs $x in $s)
 end

 [eliminate2] P & $s=$x => $x=$s & apply($x->$s,P)
 if not(occurs $x in $s)
 end

 [matchClash] P & $s=$x => false
 if not isvar($s)
 end

 [mergeClash] P & $x=$s => false
 if mergingClash(P,$x,$s)
 end

 [mergePass1] P & $x=$s => $x=$s & P
 end

 [mergePass2] P & $x=$s => R
 where b:=() occurs $x in P
 choose
 try if not(b)
 where R:=() $x=$s & P
 try if b
 where R:=()P
 end
 end

 end

```

```

[mergePass3] P => P if false end

// initial and final transitions rules

[trueelim] P & true => P end
[truepass] P & true => true & P end
[trueadd] P => true & P end
[falsepropag] false => false end
[truepropag] true => true end

end

FOR EACH SS:pair[identifier,int]; F:identifier; N:int
SUCH THAT SS:=(listExtract) elem(Fss) AND F:=()first(SS)
 AND N:=()second(SS) :{
 rules for eqSystem
 P:eqSystem;
 s_1,...,s_N:term; t_1,...,t_N:term;
 local
 [decompose] P & F(s_1,...,s_N)=F(t_1,...,t_N) => P { & s_I=t_I }_I=1...N end
 end
}

rules for bool
 P,Q : eqSystem;
 $s,$t : term;
 $x,$y : variable;
global
 [] mergingClash(true,$x,$s) => false end
 [] mergingClash(P & Q,$x,$s) => mergingClash(P,$x,$s)
 or mergingClash(Q,$x,$s) end
 [] mergingClash($x=$s,$y,$t) => eq_variable($x,$y) and neq_term($s,$t) end
end
Uses & 23b 35 59d, -> 54 59d 65d, = 59d, apply 54 59d, eqSystem 59d, head 55e 57 75c, identifier 63e, isvar 75c,
mergingClash 70, term 75c, and variable 75c.

```

69 *(strategies for syntacticUnification 69)*≡

(66a)

```

strategies for eqSystem
implicit

[.] unify => dc one(trueadd);
repeat*(dc one(unfold, delete, decompose, conflict, coalesce,
 occCheck1, occCheck2, eliminate1, eliminate2));
dc one(trueelim, truepropag, falsepropag)
end

[.] unifys => repeat*(dc one(unfold, delete, decompose, conflict,
 coalesce, occCheck1, occCheck2, eliminate1, eliminate2));
dc one(truepass, truepropag)
end

[.] match => dc one(trueadd);
repeat*(dc one(unfold, delete, decompose, conflict,
 matchClash, mergePass1));
dc one(truepass, truepropag, falsepropag);

```

```

 repeat*(dc one(unfold, mergeClash, mergePass2, mergePass3));
 dc one(trueelim, truepropag, falsepropag)
end

[.] matchs =>
 repeat*(dc one(unfold, delete, decompose, conflict,
 matchClash, mergePass1));
 dc one(truepass, truepropag) ;
 repeat*(dc one(unfold, mergeClash, mergePass2, mergePass3));
 dc one(truepass, truepropag)
end
end

```

Uses eqSystem 59d, match 70, and unify 70 151.

```

70 < * 70 > ≡
 // This file is generated automatically: do not edit it directly.
 < the syntacticUnification specification 66a >

```

Defines:

```

match, used in chunks 67c and 69.
mergingClash, used in chunks 67b and 68.
unify, used in chunks 67c, 69, 142, and 144.

```

## 2.11 termCommons.eln

This module defines notions common to all first-order terms modulo a signature (**Fss**) and a set of variables (**Vars**). Terms built w.r.t. some concrete signatures are then defined by instantiating the definitions accordingly.

71a *(the termCommons specification 71a)*≡ (75c)

```

module termCommons[Fss,Vars]
 <imports for termCommons 71b>
 <sorts for termCommons 71c>
 <global operators for termCommons 71d>
 <strategy operator for termCommons 73b>
 <rules for termCommons 73c>
 <strategies for termCommons 75b>
end

```

Several importations has to be done globally or locally. We have to use the equalities between terms (`eq[term]`) and between variables (`eq[variable]`) as well as the notion of substitution (`substitution`). The local importations are needed for defining the various operations on terms.

71b *(imports for termCommons 71b)*≡ (71a)

```

import global eq[term] eq[variable] substitution;
local int bool identifier list[int] list[variable] list[identifier]
 Fss Vars pair[identifier,int] list[pair[identifier,int]];
end

```

Uses `identifier` 63e, `substitution` 54, `term` 75c, and `variable` 75c.

The sorts defined in this module are the terms and the variables.

71c *(sorts for termCommons 71c)*≡ (71a)

```

sort variable term;
end

```

Uses `term` 75c and `variable` 75c.

There are several types of operators: explicit operators that are general to all signatures, implicit operators that depend on the signature given as argument, variable operators and strategy operators.

71d *(global operators for termCommons 71d)*≡ (71a)

```

operators global
 <explicit operators 71e>
 <implicit operators 72a>
 <variable operators 73a>
end

```



We can construct variables indexed by an integer (`var(@)`) and we declare that all variables are terms. We can get the head symbol of a term (`head`) and any of its subterms (`-th subterm`). Using `isvar` one can check if a term is a variable and with `nvocc` one obtains a non-variable positions of a term.

```
71e <explicit operators 71e>≡ (71d)
 var(@) : (int) variable;
 @ : (variable) term pri 1000;

 head(@) : (term) identifier;
 @-th subterm(@) : (int term) term;
 isvar(@) : (term) bool;
 nvocc(@) : (term) list[int];

 @ at @ : (term list[int]) term;
 @[@] at @ : (term term list[int]) term;

 rename_subst(@,@) : (list[variable] int) substitution;
```

Uses at 75c, head 55e 57 75c, identifier 63e, isvar 75c, nvocc 75c, rename\_subst 75c, substitution 54, term 75c, -th 55e 57 75c, and variable 75c.

For each symbol of the signature we define the terms that can be constructed using it as head symbol. Two case are considered: operators of arity zero and non constant operators.

```
72a <implicit operators 72a>≡ (71d)
 <implicit operators const 72b>
 <implicit operators nonconst 72c>
```

This part creates from the user specification the actual rank for operators of arity zero (the constants).

```
72b <implicit operators const 72b>≡ (72a)
 FOR EACH SS:pair[identifier,int]; F:identifier; N:int
 SUCH THAT SS:=(listExtract) elem(Fss) AND F:=()first(SS)
 AND N:=()second(SS) ANDIF N==0 :{
 F() : term;
 F : term alias F():;
```

Uses == 20 23b 31d 35 62e 88, identifier 63e, and term 75c.

From the user specification, this creates the actual rank for non-constant operators.

```
72c <implicit operators nonconst 72c>≡ (72a)
 FOR EACH SS:pair[identifier,int]; F:identifier; N:int
 SUCH THAT SS:=(listExtract) elem(Fss) AND F:=()first(SS)
 AND N:=()second(SS) ANDIF N > 0 :{

 F({,@}~N) : ({term}~N)term;

 F({@,}~N) : ({term}~N)term alias F({,@}~N):;
 F({@,}~N) : ({term}~N)term alias F({,@}~N):;
 F(@ {,@}~(N-1)) : ({term}~N) term alias F({,@}~N):;

 }
```

Uses identifier 63e and term 75c.

The list of variables is given by `varlist_Vars` and the number of variables by `varnum_Vars`.

```
73a <variable operators 73a>≡ (71d)
 FOR EACH SS:identifier SUCH THAT SS:=(listExtract) elem(Vars) :{
 SS : variable;
 }
 varlist_Vars : list[variable];
 varnum_Vars : int;
```

Uses identifier 63e, variable 75c, varlist\_Vars 75c, and varnum\_Vars 75c.

A strategy that gives all the non-variable occurrences is provided.

```
73b <strategy operator for termCommons 73b>≡ (71a)
 stratop
 global
 chooseOccurrence : <list[int]> bs;
 end
 Uses chooseOccurrence 75c.
```

The rules defining the operators are defined below. Most of them depend on the signature and are defined using the preprocessor.

```
73c <rules for termCommons 73c>≡ (71a) 74a>
 rules for list[variable]
 global
 [] varlist_Vars =>
 FOR EACH SS:identifier SUCH THAT SS:=(listExtract) elem(Vars) :{ SS.} nil
 end
 end
 end

 FOR EACH N:int SUCH THAT
 N:=() size_of_variable_list(FOR EACH SS:identifier
 SUCH THAT SS:=(listExtract) elem(Vars)
 :{ SS.} nil)

 :{
 rules for int
 global
 [] varnum_Vars => N end
 end
 }
 }
```

Uses identifier 63e, variable 75c, varlist\_Vars 75c, and varnum\_Vars 75c.

```
74a <rules for termCommons 73c>+≡ (71a) <73c 74b>
 FOR EACH SS:pair[identifier,int]; F:identifier; N:int
 SUCH THAT SS:=(listExtract) elem(Fss) AND F:=()first(SS)
 AND N:=()second(SS) ANDIF N > 0 :{
 rules for term
 t_1,...,t_N : term;
 global

 { [] I-th subterm(F(t_1,...,t_N)) => t_I end
 }_I=1...N

 rules for list[int]
 oocc:list[int];
 local
 { [inOccRule] nvocc(F(t_1,...,t_N)) => J.oocc
 where oocc:=(chooseOccurrence) nvocc(t_J)
 end
 }_J=1...N
 end

 end
 }
```

Uses = 59d, chooseOccurrence 75c, identifier 63e, nvocc 75c, term 75c, and -th 55e 57 75c.

```

74b <rules for termCommons 73c>+≡ (71a) <74a 75a>
FOR EACH SS:pair[identifier,int]; F:identifier; N:int
SUCh THAT SS:=(listExtract) elem(Fss) AND F:=(λ)first(SS)
 AND N:=(λ)second(SS) :{
 rules for identifier
 t_1,...,t_N : term;
 global
 [] head(F(t_1,...,t_N)) => F end
 end
}

FOR EACH SS:pair[identifier,int]; F:identifier; N:int
SUCh THAT SS:=(listExtract) elem(Fss) AND F:=(λ)first(SS)
 AND N:=(λ)second(SS) ANDIF N != 0 :{
 rules for term
 t_1,...,t_N : term;
 $s : term;
 l : list[int];
 global
 { [] F(t_1,...,t_N)[$s] at I.l => F(t_1,...,t_(I-1),t_I[$s] at l,t_(I+1),...,t_N)
 end
 }_I=1...N
 end
}

```

Uses != 20 23b 31d 35 62e 88, = 59d, at 75c, head 55e 57 75c, identifier 63e, and term 75c.



**term**, used in chunks 53, 55, 56, 58, 59, 65–68, 71, 72, 74, and 75a.  
**-th**, used in chunks 55d, 56d, 71e, 74a, and 75a.  
**variable**, used in chunks 53e, 59a, 65–68, 71, 73, and 75a.  
**varlist\_Vars**, used in chunk 73.  
**varnum\_Vars**, used in chunk 73.

### **3 Library strategy**

### 3.1 Any.eln

|     |                                                                                                                                                                                                                                                                                                                                                                                                                              |       |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| 78a | <pre> &lt;the Any speci cation 78a&gt;≡   module Any     &lt;imports for Any 78b&gt;     &lt;sorts for Any 78c&gt;     &lt;operators for Any 78d&gt;     &lt;strategy operator for Any (never defined)&gt;     &lt;rules for Any 78g&gt;     &lt;strategies for Any 78h&gt;   end </pre>                                                                                                                                     |       |
| 78b | <pre> &lt;imports for Any 78b&gt;≡ </pre>                                                                                                                                                                                                                                                                                                                                                                                    | (78a) |
| 78c | <pre> &lt;sorts for Any 78c&gt;≡ </pre>                                                                                                                                                                                                                                                                                                                                                                                      | (78a) |
| 78d | <pre> &lt;operators for Any 78d&gt;≡   &lt;global operators for Any 78e&gt;   &lt;local operators for Any 78f&gt; </pre>                                                                                                                                                                                                                                                                                                     | (78a) |
| 78e | <pre> &lt;global operators for Any 78e&gt;≡ </pre>                                                                                                                                                                                                                                                                                                                                                                           | (78d) |
| 78f | <pre> &lt;local operators for Any 78f&gt;≡ </pre>                                                                                                                                                                                                                                                                                                                                                                            | (78d) |
| 78g | <pre> &lt;rules for Any 78g&gt;≡ </pre>                                                                                                                                                                                                                                                                                                                                                                                      | (78a) |
| 78h | <pre> &lt;strategies for Any 78h&gt;≡ </pre>                                                                                                                                                                                                                                                                                                                                                                                 | (78a) |
| 78i | <pre> (* 78i)≡   // This file is generated automatically: do not edit it directly.   module Any    import global int list[Any] any[int];   end    sort Any Functor;   end    operators global     functor(@)      : (Any) Functor;     explode(@)      : (Any) list[Any];     implode(@,@)    : (Functor list[Any]) Any;   local     @                : (int) Functor;   end    rules for list[Any]     n:int; a:Any; </pre> |       |

```
global
 [] explode((int)n) => nil end
end

rules for Functor
 n:int;
global
 [] functor((int)n) => n end
end

rules for Any
 n:int;
global
 [] implode(n,nil) => (int)n end
end

end
```

Uses do 89e.



### 3.2 Meta\_apply.eln

The module `Meta_apply[X]` parameterized by a sort `X` can only be used in interpreted mode.

```
106a <the Meta_apply specification 106a>≡ (109b)
 module Meta_apply[X]
 // ONLY FOR INTERPRETER
 <imports for Meta_apply 106b>
 <operators for Meta_apply 106c>
 <rules for Meta_apply 108a>
 end
```

```
106b <imports for Meta_apply 106b>≡ (106a)
 import global builtinInt list[X] Meta_strat[X]; end
```

```
106c <operators for Meta_apply 106c>≡ (106a)
 <global operators for Meta_apply 107a>
 <local operators for Meta_apply 107b>
```

The construction `where y:=(META)meta_apply(s,t)` is semantically equivalent to `where y:=(s)t` where `s` is a basic strategy.

The strategy `META` used here just indicates that `(META)meta_apply(s,t)` returns in general several results, which would not be the case with `()meta_apply(s,t)`. `META` is a local constant basic strategy which applies only to terms of the form `meta_apply(s,t)` or `meta_apply(s,t,n)`.

```
107a <global operators for Meta_apply 107a>≡ (106c) 81a>
 operators global
 meta_apply(@,@) :
 (Strategy[X] X) X code -129;
```

Uses `meta_apply` 109b 82e 109b.

`meta_apply(s,t,n)` gives the  $n$ -th result of application of the strategy `s` to the term `t`, if it exists. Otherwise, it gives no result.

`setof(s,t)` produces the list if results of application of the strategy `s` to the term `t`.

`setof(s,t,n)` produces the list if results of application of the strategy `s` to the term `t`, starting from the  $n$ -th.

`setof(s,t,n,m)` produces the list if  $m$  results of application of the strategy `s` to the term `t`, starting from the  $n$ -th.

```
81a <global operators for Meta_apply 107a>+≡ (106c) <107a
 meta_apply(@,@,@) :
 (Strategy[X] X builtinInt) X code -130;
 set_of(@,@) : (Strategy[X] X) list[X];
 set_of(@,@,@) : (Strategy[X] X builtinInt) list[X];
 set_of(@,@,@,@) : (Strategy[X] X builtinInt builtinInt) list[X];
 meta_apply(@,@,@,@) : (Strategy[X] X builtinInt builtinInt) list[X];
```

Uses `meta_apply` 109b 82e 109b.

`meta_apply(s,t,n,m)` gives  $m$  results from the  $n$ -th one. If  $n = 0$ , it gives all solutions.

```
107b <local operators for Meta_apply 107b>≡ (106c)
 local
 meta_apply(@,@,@,@) :
 (Strategy[X] list[X] builtinInt builtinInt) list[X] code -128;
 end
```

Uses `meta_apply` 109b 82e 109b.

```
108a <rules for Meta_apply 108a>≡ (106a)

 rules for list[X]
 s : Strategy[X];
 t : X;
 m,n : builtinInt;
 res : list[X];

 global
 [] meta_apply(s,t,m,n) => res where res:=()meta_apply(s,t.nil,m,n) end
 [] set_of(s,t) => res where res:=(META)meta_apply(s,t.nil,0,0) end
 [] set_of(s,t,n) => res where res:=(META)meta_apply(s,t.nil,0,n) end
 [] set_of(s,t,m,n) => res where res:=(META)meta_apply(s,t.nil,m,n) end
 end
```

Uses `meta_apply` 109b 82e 109b.

```
109b <* 109b>≡
 // This file is generated automatically: do not edit it directly.
 <the Meta_apply speci cation 106a>
```

Defines:

`meta_apply`, used in chunks 107a, 81a, 107b, 108a, 82d, 107–109, and 111c.

Uses `do` 89e.

### 3.3 Meta\_capply.eln

The module `Meta_capply[X]` parameterized by a sort `X` can only be used in compiled mode. Its functionality is the same as `Meta_apply[X]`.

```
82a <the Meta_capply specification 82a>≡ (82e)
 module Meta_capply[X]
 // ONLY FOR COMPILER
 <imports for Meta_capply 82b>
 <operators for Meta_capply 82c>
 end
```

```
82b <imports for Meta_capply 82b>≡ (82a)
 import global builtinInt list[X] string; end
```

```
82c <operators for Meta_capply 82c>≡ (82a)
 <global operators for Meta_capply 82d>
```

```
82d <global operators for Meta_capply 82d>≡ (82c)
 operators global
 meta_apply(@,@) : (string X) X code -129;
 meta_apply(@,@,@) : (string X builtinInt) X code -130;
 end
```

Uses `meta_apply` 109b 82e 109b.

```
82e <* 82e>≡
 // This file is generated automatically: do not edit it directly.
 <the Meta_capply specification 82a>
```

Defines:

`meta_apply`, used in chunks 107a, 81a, 107b, 108a, 82d, 107–109, and 111c.

Uses do 89e.

### 3.4 Meta\_strat.eln

The module `Meta_strat[X]` parameterized by a sort `X` can only be used in interpreted mode. This module exports the sort `Strategy[X]`, which specifies an external form of basic strategies of sort `X`. The signature of basic strategies is defined by several built-in constructors used to convert strategy terms of sort `Strategy[X]` into basic strategies over the sort `X`. This conversion is used for the implementation of `meta_apply` symbols of the previous module.

```

106a <the Meta_strat specification 106a>≡ (109b)
 module Meta_strat[X]
 // ONLY FOR INTERPRETER
 <imports for Meta_strat 106b>
 <sorts for Meta_strat 83c>
 <operators for Meta_strat 106c>
 end

106b <imports for Meta_strat 106b>≡ (106a)
 import global string; end

83c <sorts for Meta_strat 83c>≡ (106a)
 sort Labels[X]
 Strategies[X] Strateg[X]
 Strategy[X];
 end

106c <operators for Meta_strat 106c>≡ (106a)
 <global operators for Meta_strat 107a>

```

| Builtins             | Code | Signature                                 | Comments                                                                                                                                                               |
|----------------------|------|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| @                    | 131  | (Strategy[X]) Strategies[X]               | embeds a basic strategy in a list of basic strategies                                                                                                                  |
| @ , @                | 132  | (Strategy[X] Strategies[X]) Strategies[X] | concatenates a basic strategy to a list of basic strategies                                                                                                            |
| @                    | 133  | (string) Labels[X]                        | embeds a string in a list of rule labels                                                                                                                               |
| @ , @                | 134  | (string Labels[X]) Labels[X]              | concatenates a string to a list of rule labels                                                                                                                         |
| dc(@)                | 135  | (Labels[X]) Strateg[X]                    | dc(1) is a basic strategy that chooses one label in the list of labels 1 and returns all results of the corresponding rule                                             |
| dk(@)                | 136  | (Labels[X]) Strateg[X]                    | dk(1) is a basic strategy that chooses successively all labels in the list of labels 1 and returns all results of the corresponding rules                              |
| dc(@)                | 137  | (Strategies[X]) Strateg[X]                | dc(1s) is a basic strategy that chooses one basic strategy in the list 1s and returns all its results                                                                  |
| dk(@)                | 138  | (Strategies[X]) Strateg[X]                | dk(1s) is a basic strategy that chooses successively all basic strategies in the list 1s and returns all their results                                                 |
| repeat @ endrepeat   | 139  | (Strategy[X]) Strateg[X]                  | repeat s endrepeat is a basic strategy that repeats the application of the basic strategy s until it fails. Similar to repeat*(@).                                     |
| iterate @ enditerate | 140  | (Strategy[X]) Strateg[X]                  | iterate s enditerate is a basic strategy that repeats the application of the basic strategy s until it fails and returns intermediate results. Similar to iterate*(@). |
| @                    | 141  | (Strateg[X]) Strategy[X]                  | embeds the previous constructions into the sort Strategy[X]                                                                                                            |
| @ ; @                | 142  | (Strateg[X] Strategy[X]) Strateg[X]       | concatenates any previous construction to a strategy                                                                                                                   |
| id                   | 143  | () Strateg[X]                             | id is a basic strategy                                                                                                                                                 |
| call @ : X           | 144  | (string) Strateg[X]                       | call s:X represents an application of a basic strategy named s of sort X                                                                                               |

107a (global operators for Meta\_strat 107a)≡

(106c)

```

operators global
@ : (Strategy[X]) Strategies[X] code 131;
@ , @ : (Strategy[X] Strategies[X]) Strategies[X] code 132;
@ : (string) Labels[X] code 133;
@ @ : (string Labels[X]) Labels[X] code 134;
dc(@) : (Labels[X]) Strateg[X] code 135;
dk(@) : (Labels[X]) Strateg[X] code 136;
dc(@) : (Strategies[X]) Strateg[X] code 137;
dk(@) : (Strategies[X]) Strateg[X] code 138;
repeat @ endrepeat : (Strategy[X]) Strateg[X] code 139;
while @ endwhile : (Strategy[X]) Strateg[X] alias repeat @ endrepeat;;
iterate @ enditerate : (Strategy[X]) Strateg[X] code 140;

```

```
@ : (Strateg[X]) Strategy[X] code 141;
@@ : (Strateg[X] Strategy[X]) Strategy[X] code 142;
id : Strateg[X] code 143;
call @': 'X : (string) Strateg[X] code 144;
end
```

Uses `repeat` 92e and `while` 92e.

```
109b <* 109b>≡
// This file is generated automatically: do not edit it directly.
<the Meta_strat speci cation 106a>
```

Uses `do` 89e.

### 3.5 any.eln

The module `any[X]` parameterized by the sort `X` defines the (dynamic) sort `Any` by a coercion that transforms a term of sort `X` into term of sort `Any`. Polymorphic functions and strategies can be defined on the sort `Any`.

```
86a <the any speci cation 86a>≡ (86e)
 module any[X]
 <sorts for any 86b>
 <operators for any 86c>
 end
```

```
86b <sorts for any 86b>≡ (86a)
 sort Any;
 end
```

```
86c <operators for any 86c>≡ (86a)
 <global operators for any 86d>
```

```
86d <global operators for any 86d>≡ (86c)
 operators global
 (X)@ : (X) Any;
 end
```

```
86e <* 86e>≡
 // This file is generated automatically: do not edit it directly.
 <the any speci cation 86a>
 Uses do 89e.
```

### 3.6 booleg.eln

The module `booleg` extends the module `bool` with boolean operators `and`, `or`, `not`, `==`, `!=`.

```
87a <the booleg specification 87a>≡ (88)
 module booleg
 <imports for booleg 87b>
 <sorts for booleg 87c>
 <operators for booleg 87d>
 <rules for booleg 87f>
 end
```

```
87b <imports for booleg 87b>≡ (87a)
 import local bool;
 end
```

```
87c <sorts for booleg 87c>≡ (87a)
 sort booleg;
 end
```

```
87d <operators for booleg 87d>≡ (87a)
 <global operators for booleg 87e>
```

```
87e <global operators for booleg 87e>≡ (87d)
 operators global
 @ : (bool) booleg pri 1000;

 @ and @ : (booleg booleg) booleg assocLeft pri 100;
 @ or @ : (booleg booleg) booleg assocLeft pri 100;
 not(@) : (booleg) booleg pri 200;
 @ == @ : (booleg booleg) booleg assocLeft pri 200;
 @ != @ : (booleg booleg) booleg assocLeft pri 200;
 end
```

Uses `!=` 20 23b 31d 35 62e 88, `==` 20 23b 31d 35 62e 88, `and` 20 88, `not(@)` 20 88, `and` or 20 88.

```
87f <rules for booleg 87f>≡ (87a)
 rules for booleg
 a,b,c:bool;
 global
 [] a and b => c where c:=()a and b end
 [] a or b => c where c:=()a or b end
 [] not(a) => c where c:=()not(a) end
 [] a==b => c where c:=()a==b end
 [] a!=b => c where c:=()a!=b end
 end
```

Uses `!=` 20 23b 31d 35 62e 88, `==` 20 23b 31d 35 62e 88, `and` 20 88, `and` or 20 88.



```
88 (* 88)≡
 // This file is generated automatically: do not edit it directly.
 <the booleg specification 87a>
```

Defines:

`!=`, used in chunks 19e, 22b, 30, 31c, 33e, 34b, 62d, 74b, and 87.

`==`, used in chunks 15f, 19e, 22b, 30, 31c, 33e, 34b, 56c, 62d, 72b, and 87.

`and`, used in chunks 19d, 22a, 33e, 41a, 45a, and 87.

`not(Ⓞ)`, used in chunks 19d and 87e.

`or`, used in chunks 19d, 22a, 33e, and 87.

Uses `do` 89e.

### 3.7 commit.eln

The module `commit[X]` parameterized by a sort `X` contains only one strategy. `commit n do s od` enumerates the `n` first results of the strategy `s`.

```
89a <the commit speci cation 89a>≡ (89e)
 module commit[X]
 <imports for commit 89b>
 <strategy operator for commit 89c>
 <strategies for commit 89d>
 end
```

Uses `commit 89e`.

```
89b <imports for commit 89b>≡ (89a)
 import global int strat[X] list[X] strapply[X]; end
```

```
89c <strategy operator for commit 89c>≡ (89a)
 stratop global
 commit @ do @ od : (builtinInt <X>) <X>;
 end
```

Uses `commit 89e`, `do 89e`, and `od 89e`.

```
89d <strategies for commit 89d>≡ (89a)
 strategies for X
 n : builtinInt;
 s : <X>;
 a,t : X;
 ts : list[X];
 explicit
 [.] [commit n do s od]t => a where ts:=()setof(s,t,0,n)
 where a:=(listExtract)elem(ts)
 end
 end
```

Uses `commit 89e`, `do 89e`, and `od 89e`.

```
89e <* 89e>≡
 // This file is generated automatically: do not edit it directly.
 <the commit speci cation 89a>
```

Defines:

`commit`, used in chunk 89.

`do`, used in chunks 78i, 109b, 82e, 109b, 86e, 88–93, 96–98, 101, 102e, and 104b.

`od`, used in chunks 89–91.

### 3.8 loops.eln

The module `loops[v,X]` is parameterized by the name of the iteration variable `v` of sort `int` and a sort `X`. It defines several strategy operators that iterate a strategy whose parameter belongs to an integer interval.

90a *(the loops specification 90a)*≡ (91b)

```

module loops[v,X]
 <imports for loops 90b>
 <operators for loops 90c>
 <strategy operator for loops 90e>
 <strategies for loops 91a>
end

```

90b *(imports for loops 90b)*≡ (90a)

```

import global int strat[X] replace[int,<X->X>]; end

```

90c *(operators for loops 90c)*≡ (90a)

```

<global operators for loops 90d>

```

90d *(global operators for loops 90d)*≡ (90c)

```

operators global
 v : int;
end

```

`for v=n to m do S od` is equivalent to  
`S(v:=n) ; S(v:=n+1) ; ... ; S(v:=m)`.

It successively applies the strategy `S` with the parameter `v` taking the values `n, n+1, ..., m`.

`for v=n downto m do S od` is equivalent to  
`S(v:=n) ; S(v:=n-1) ; ... ; S(v:=m)`.

It successively applies the strategy `S` with the parameter `v` taking the values `n, n-1, ..., m`.

`some v=n to m do S od` is equivalent to  
`dk(S(v:=n),S(v:=n+1),...,S(v:=m))`.

It applies the strategy `S` with all values of the parameter `v` in `n, n+1, ..., m`.

90e *(strategy operator for loops 90e)*≡ (90a)

```

stratop global
 for v=@ to @ do @ od : (int int <X>) <X>;
 for v=@ downto @ do @ od : (int int <X>) <X>;
 some v=@ to @ do @ od : (int int <X>) <X>;
end

```

Uses `do 89e` and `od 89e`.

```

91a <strategies for loops 91a>≡ (90a)
strategies for X
 n, m, mm : int;
 s, ss : <X>;
 t : X;
implicit
[.] some v=n to m do s od =>
 dk(replace(v,n,s), some v= n+1 to m do s od)
 if n <= m
end
[.] for v=n to m do s od =>
 replace(v,n,s); for v= n+1 to m do s od
 if n <= m
end

[.] for v=n to m do s od => id if n > m
end
[.] for v=n downto m do s od =>
 replace(v,n,s); for v= n-1 downto m do s od
 if n >= m
end
[.] for v=n downto m do s od => id if n < m
end
end

```

Uses do 89e and od 89e.

```

91b <* 91b>≡
// This file is generated automatically: do not edit it directly.
<the loops speci cation 90a>

```

Uses do 89e.

### 3.9 meta.eln

The module `meta[Y]` parameterized by a strategy sort `Y` defines several iteration strategies.

```
92a <the meta speci cation 92a>≡ (92e)
 module meta[Y]
 <imports for meta 92b>
 <strategy operator for meta 92c>
 <strategies for meta 92d>
 end
```

```
92b <imports for meta 92b>≡ (92a)
 import global int strat[Y];
 end
```

```
92c <strategy operator for meta 92c>≡ (92a)
 stratop global
 TRY @ : (<Y>) <Y>;
 repeat @ : (<Y>) <Y>;
 repeat-@ @ : (int <Y>) <Y>;
 while @ : (<Y>) <Y>;
 dcwhile @ : (<Y>) <Y>;
 end
```

Uses `dcwhile` 92e, `repeat` 92e, `repeat-` 92e, `TRY` 92e, and `while` 92e.

```
92d <strategies for meta 92d>≡ (92a)
 strategies for Y
 n : int; s : <Y>;
 implicit
 [.] repeat s => first(s;repeat s,id) end
 [.] repeat-0 s => id end
 [.] repeat-n s => s; repeat-n-1 s end
 [.] while s => dk(id, s;while s) end
 [.] dcwhile s => dc(s;dcwhile s, id) end
 [.] TRY s => first(s,id) end
 end
```

Uses `dcwhile` 92e, `repeat` 92e, `repeat-` 92e, `TRY` 92e, and `while` 92e.

```
92e <* 92e>≡
 // This file is generated automatically: do not edit it directly.
 <the meta speci cation 92a>
```

Defines:

- `dcwhile`, used in chunk 92.
- `repeat`, used in chunks 107a and 92.
- `repeat-`, used in chunk 92.
- `TRY`, used in chunk 92.
- `while`, used in chunks 107a and 92.

Uses `do` 89e.

### 3.10 strapply.eln

93a *(the strapply specification 93a)*≡ (93f)

```

module strapply[X]
 (imports for strapply 93b)
 (operators for strapply 93c)
 (rules for strapply 93e)
end

```

93b *(imports for strapply 93b)*≡ (93a)

```

import global strat[X] builtinInt Meta_strat[X] Meta_apply[X];
end

```

93c *(operators for strapply 93c)*≡ (93a)

```

(global operators for strapply 93d)

```

setof(s,t) produces the list if results of application of the strategy s to the term t.

setof(s,t,n) produces the list if results of application of the strategy s to the term t, starting from the n-th.

setof(s,t,n,m) produces the list if m results of application of the strategy s to the term t, starting from the n-th.

93d *(global operators for strapply 93d)*≡ (93c)

```

operators global
 setof(@,@) : (<X> X) list[X];
 setof(@,@,@) : (<X> X builtinInt) list[X];
 setof(@,@,@,@): (<X> X builtinInt builtinInt) list[X];
end

```

93e *(rules for strapply 93e)*≡ (93a)

```

rules for list[X]
 s : <X>;
 t : X;
 lst : list[X];
 m, n : builtinInt;

global
 [] setof(s,t) => set_of(call "eval":X,[s]t) end
 [] setof(s,t,m) => set_of(call "eval":X,[s]t,m) end
 [] setof(s,t,m,n) => set_of(call "eval":X,[s]t,m,n) end
end

```

Uses eval 96 104b.

93f *(\* 93f)*≡

```

// This file is generated automatically: do not edit it directly.
(the strapply specification 93a)

```

Uses do 89e.

### 3.11 strat.eln

The module `strat [X]` parameterized by a sort `X` has to be imported for working with type-preserving strategies of sort `<X->X`. It defines the `eval` strategy of the interpreter of the language of defined strategies, and several specific strategy constructors, which are not present in the case of type-changing strategies. The rest of strategy constructors is defined in the module `strategy/strsig.eln`.

```

94a <the strat speci cation 94a>≡ (96)
 module strat[X]
 //--- Use this module, if you want to work with type-preserving
 //--- strategies over the type X, e.g. X->X
 <imports for strat 94b>
 <sorts for strat 94c>
 <strategy operator for strat 94d>
 <strategies for strat 95>
 end

94b <imports for strat 94b>≡ (94a)
 import global
 eq[X] strsig[X,X] bool; // user_strategy[X];
 end

94c <sorts for strat 94c>≡ (94a)
 sort
 StrList[X,X] <X->X;
 end

94d <strategy operator for strat 94d>≡ (94a)
 stratop
 global
 id : <X> code -186;
 if @ then @ orelse @ fi
 : (<X> <X> <X>) <X> code -187;
 @ : (X) <X> pri 2000; // strategy coercion
 return @ : (X) <X>; // returns argument as result
 eval_ONE : <X> bs;
 eval_DC : <X> bs;
 eval_DK : <X> bs;
 eval : <X> bs;
 end

```

Uses `eval` 96 104b, `eval_DC` 96 104b, `eval_DK` 96 104b, and `eval_ONE` 96 104b.

```

95 <strategies for strat 95>≡ (94a)
 strategies for X //<X->X>
 a, b, l, r : X;
 S, S1, S2 : <X->X>;
 SS : StrList[X,X];
explicit
[DK] [dk(SS)]a => b where b:=(eval_DK)((SS))a end
[DK1] ((S1,SS))a => b where b:=[S1]a end
[DK2] ((S1,SS))a => b where b:=(eval_DK)((SS))a end

[DC] [dc(SS)]a => b where b:=(eval_DC)((SS))a end
[DC1] ((S1,SS))a => b where b:=[S1]a end
[DC2] ((S1,SS))a => b where b:=(eval_DC)((SS))a end

[ONE] [one(SS)]a => b where b:=(eval_ONE)((SS))a end
[ONE1] ((S1,SS))a => b where b:=[S1]a end
[ONE2] ((S1,SS))a => b where b:=(eval_ONE)((SS))a end

[IF1] [if true then S1 else S2 fi]a => b where b :=[S1]a end
[IF2] [if false then S1 else S2 fi]a => b where b :=[S2]a end

[IF01] [if S then S1 orelse S2 fi]a =>[S1]b where b:=[S]a end
[IF02] [if S then S1 orelse S2 fi]a =>[S2]a end
[IF03] [S1]a => b where b:=[S1]a end
[DSTR] [return a]b => a end
[ID] [id]a => a end
[CONST] [a]b => b if eq_X(a,b) end

// the following rules are added because of 'missing rule warning'
[FSYM] [S]b => b if false end
[LAB] [S]b => b if false end
[DSTR] [S]b => b if false end
[CONCAT] [S]b => b if false end

implicit // evaluation ELAN-strategy of the interpreter
[.] eval_ONE => dc one(dk(ONE1) , dk(ONE2)) end
[.] eval_DC => dc(dk(DC1) , dk(DC2)) end
[.] eval_DK => dk(DK1 , DK2) end
[.] eval => first(
 dk(FSYM) ,
 dk(DSTR) ,
 dk(DK) ,
 dk(DC) ,
 dk(ONE) ,
 dk(LAB) ,
 dc(ID) ,
 dc(CONST) ,
 dk(CONCAT) ,
 dk(IF1) , dk(IF2) ,
 dc(dk(IF01) , dk(IF02)); dk(IF03)
)
end
end

```



```
96 ⟨ * 96 ⟩ ≡
 // This file is generated automatically: do not edit it directly.
 ⟨ the strat speci cation 94a ⟩
```

Defines:

`eval`, used in chunks 93–95, 97e, 103d, and 104a.

`eval_DC`, used in chunks 94d, 95, 103, and 104a.

`eval_DK`, used in chunks 94d, 95, 103, and 104a.

`eval_ONE`, used in chunks 94d, 95, 103, and 104a.

Uses `do 89e`.

### 3.12 strcapply.eln

97a *<the strcapply specification 97a>*≡ (97f)

```

module strcapply[X]
 <imports for strcapply 97b>
 <operators for strcapply 97c>
 <rules for strcapply 97e>
end

```

97b *<imports for strcapply 97b>*≡ (97a)

```

import global strat[X] Meta_capply[X]; end

```

97c *<operators for strcapply 97c>*≡ (97a)

```

<global operators for strcapply 97d>

```

97d *<global operators for strcapply 97d>*≡ (97c)

```

operators global
 setof(@,@) : (<X> X) list[X];
end

```

setof(s,t) produces the list if results of application of the strategy s to the term t.

97e *<rules for strcapply 97e>*≡ (97a)

```

rules for list[X]
 s : <X>;
 t : X;
 lst : list[X];

global
 [] setof(s,t) => set_of(call("eval"),[s]t) end
end

```

Uses eval 96 104b.

97f *<\* 97f>*≡

```

// This file is generated automatically: do not edit it directly.
<the strcapply specification 97a>

```

Uses do 89e.

### 3.13 strconc.eln

The module `strconc[X,Y,Z]` parameterized by three sorts `X,Y,Z` introduces a concatenation symbol `;` of two strategies of sorts `<X->Y>` and `<Y->Z>`.

```

98a <the strconc speci cation 98a>≡ (98e)
 module strconc[X,Y,Z]
 <imports for strconc 98b>
 <strategy operator for strconc 98c>
 <rules for strconc 98d>
 end

```

```

98b <imports for strconc 98b>≡ (98a)
 import
 global
 strsig[X,Y] strsig[Y,Z] strsig[X,Z];
 end

```

| Builtins                      | Code | Signature                                                                                           | Comments                                                                                                                                                  |
|-------------------------------|------|-----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>;</code> <code>@</code> | -188 | $(\langle X \rightarrow Y \rangle \langle Y \rightarrow Z \rangle) \langle X \rightarrow Z \rangle$ | <code>s1 ; s2</code> concatenates the strategies <code>s1:⟨X-&gt;Y⟩</code> and <code>s2:⟨Y-&gt;Z⟩</code> to get a strategy of sort <code>⟨X-&gt;Z⟩</code> |

```

98c <strategy operator for strconc 98c>≡ (98a)
 stratop global
 @ ';' @ : (<X->Y> <Y->Z>) <X->Z> assocLeft code -188;
 end

```

```

98d <rules for strconc 98d>≡ (98a)
 rules for Z
 s1 : <X->Y; s2 : <Y->Z;
 x : X; y : Y; z : Z;
 global
 [CONCAT] [s1 ; s2]x => z where y := [s1]x where z := [s2]y end
 end

```

```

98e (* 98e)≡
 // This file is generated automatically: do not edit it directly.
 <the strconc speci cation 98a>

```

Uses do 89e.

### 3.14 strsig.eln

The module `strsig[X,Y]` parameterized by the two sorts `X,Y` defines the signature of the defined strategies. It introduces several built-in constructors used to construct strategy terms of the strategy language. The module `strsig.eln` contains strategy constructors used both for type-preserving and type-changing strategies.

```

99a <the strsig speci cation 99a>≡ (101)
 module strsig[X,Y]
 <imports for strsig 99b>
 <sorts for strsig 99c>
 <strategy operator for strsig 100>
 end

99b <imports for strsig 99b>≡ (99a)
 import global bool eq[Y]; end

99c <sorts for strsig 99c>≡ (99a)
 sort <X->Y> StrList[X,Y]
 IfApplication IfApplicationList
 Assignment AssignmentList;
 end

```

| Builtins              | Code | Signature                                     | Comments                                                                                                           |
|-----------------------|------|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| @]@                   | -180 | (<X->Y> X) Y                                  | see its alias [@@ below.                                                                                           |
| [@@                   | -180 | (<X->Y> X) Y                                  | [S] t applies the strategy S to the term t                                                                         |
| dk(@)                 | -182 | (Strlist[X,Y]) <X->Y>                         | dk(1) is the defined strategy that chooses successively all strategies in the list 1 and returns all their results |
| dc(@)                 | -181 | (Strlist[X,Y]) <X->Y>                         | dc(1) is the defined strategy that chooses one strategy in the list 1 and returns all its results                  |
| dc one(@)             | -190 | (Strlist[X,Y]) <X->Y>                         | dc one (1) is the defined strategy that chooses one strategy in the list 1 and returns one result                  |
| fail                  | -183 | () <X->Y>                                     | the defined strategy that always gives an empty set of results                                                     |
| @,@                   | -185 | (<X->Y> Strlist[X,Y])                         | concatenates a defined strategy to a list of strategies                                                            |
| Epsilon               | -189 | () Strlist[X,Y]                               | the empty list of strategies                                                                                       |
| if @ then @ else @ fi | -184 | (bool <X->Y> <X->Y>) <X->Y>                   | if b then s1 else s2 fi is the defined strategy that chooses either s1 if b is true otherwise s2                   |
|                       | 146  | (Assignment) AssignmentList                   | embeds an assignment in a list of assignments                                                                      |
| , @                   | 147  | (AssignmentList Assignment) AssignmentList    | add an assignment to the end of a list of assignments                                                              |
| if @                  | 145  | (bool) IfApplication                          | a condition if an IfApplication                                                                                    |
|                       | 146  | (IfApplication) AssignmentList                | embeds an IfApplication in a list of assignments                                                                   |
| , @                   | 147  | (AssignmentList IfApplication) AssignmentList | add an IfApplication to the end of a list of assignments                                                           |

100 <strategy operator for strsig 100>≡

(99a)

```

stratop global
 @@ : (<X->Y> X) Y pri 1 code -180;
 [@@ : (<X->Y> X) Y pri 1 alias @@:;
 ((@))@ : (StrList[X,Y] X) Y;
 //---
 dk(@) : (StrList[X,Y]) <X->Y> code -182;
 dc(@) : (StrList[X,Y]) <X->Y> code -181;
 first(@) : (StrList[X,Y]) <X->Y> alias dc(@):;
 one(@) : (StrList[X,Y]) <X->Y> code -190;

 fail : <X->Y> code -183;
 //---
 @ @ : (<X->Y> StrList[X,Y]) StrList[X,Y] code -185;
 @,@ : (<X->Y> StrList[X,Y]) StrList[X,Y] alias @ @:;
 @ , @ : (<X->Y> StrList[X,Y]) StrList[X,Y] code -179 alias @ @:;
 // ca possera de pbs

 Epsilon : StrList[X,Y] code -189;
 //---
 if @ then @ else @ fi : (bool <X->Y> <X->Y>) <X->Y> code -184;
 //---
 let @ in @ : (AssignmentList <X->Y>) <X->Y>;
 (let @ in @) : (AssignmentList <X->Y>) <X->Y> alias let @ in @:;

```

```

@ : (Assignment) AssignmentList code 146;
@ , @ : (AssignmentList Assignment) AssignmentList code 147;

//---
[@ => @] : (X Y) <X->Y>;
[@ => @ @] : (X Y IfApplicationList) <X->Y>;

if @ : (bool) IfApplication code 145;
@ : (IfApplication) IfApplicationList code 146;
@ , @ : (IfApplicationList IfApplication) IfApplicationList code 147;
end

```

```

101 < * 101 >≡
 // This file is generated automatically: do not edit it directly.
 < the strsig speci cation 99a >

```

Uses do 89e.

### 3.15 symbol.eln

The module `symbol[N, T_1, ..., T_N, TT]` defines the sort `symbol[T_1, ..., T_N, TT]` to be given to a function `symbol` with arity `N`, and rank `(T_1, ..., T_N) TT`.

```
102a <the symbol speci cation 102a>≡ (102e)
 module symbol[N, T_1, ..., T_N, TT]
 <sorts for symbol 102b>
 <operators for symbol 102c>
 end
```

```
102b <sorts for symbol 102b>≡ (102a)
 sort Symbol[T_1, ..., T_N, TT];
 end
```

```
102c <operators for symbol 102c>≡ (102a)
 <global operators for symbol 102d>
```

```
102d <global operators for symbol 102d>≡ (102c)
 operators global
 @ (@ {, @}_I=2...N) : (Symbol[T_1, ..., T_N, TT] {T_I}_I=1...N) TT;
 end
```

```
102e <* 102e>≡
 // This file is generated automatically: do not edit it directly.
 <the symbol speci cation 102a>
```

Uses do 89e.

### 3.16 tcstrat.eln

The module `tcstrat[X,Y]` parameterized by the sorts `X,Y` defines the evaluation mechanism for defined type-changing strategies of sort `X->Y`.

```
103a <the tcstrat specification 103a>≡ (104b)
 module tcstrat[X,Y]
 //--- Use this module, if you want to work with type-changing
 //--- strategies of the following kind <X->Y>
 <imports for tcstrat 103b>
 <sorts for tcstrat 103c>
 <strategy operator for tcstrat 103d>
 <rules for tcstrat 103e>
 <strategies for tcstrat 104a>
 end
```

```
103b <imports for tcstrat 103b>≡ (103a)
 //---
 import global bool strsig[X,Y] eq[Y]; end
```

```
103c <sorts for tcstrat 103c>≡ (103a)
 sort StrList[X,Y] <X->Y>; end
```

```
103d <strategy operator for tcstrat 103d>≡ (103a)
 stratop global
 eval_ONE : <Y> bs;
 eval_DC : <Y> bs;
 eval_DK : <Y> bs;
 eval : <Y> bs;
 end
```

Uses `eval` 96 104b, `eval_DC` 96 104b, `eval_DK` 96 104b, and `eval_ONE` 96 104b.

```
103e <rules for tcstrat 103e>≡ (103a)
 rules for Y
 //strategies for Y
 a, l : X; b, r : Y;
 s1, s2 : <X->Y>; ss1 : StrList[X,Y];
 global
 //explicit
 [IF1] [if true then s1 else s2 fi]a => b where b :=[s1]a end
 [IF2] [if false then s1 else s2 fi]a => b where b :=[s2]a end

 [DK] [dk(ss1)]a => b where b:=(eval_DK)((ss1))a end
 [DK1] ((s1,ss1))a => b where b:=[s1]a end
 [DK2] ((s1,ss1))a => b where b:=(eval_DK)((ss1))a end

 [DC] [dc(ss1)]a => b where b:=(eval_DC)((ss1))a end
 [DC1] ((s1,ss1))a => b where b:=[s1]a end
 [DC2] ((s1,ss1))a => b where b:=(eval_DC)((ss1))a end

 [ONE] [one(ss1)]a => b where b:=(eval_ONE)((ss1))a end
 [ONE1] ((s1,ss1))a => b where b:=[s1]a end
```



```
[ONE2] ((s1,ss1))a => b where b:=(eval_ONE)((ss1))a end
```

```
// the following rules are added because of 'missing rule warning'
```

```
[ID] b => b if false end
[CONST] b => b if false end
[IFO1] b => b if false end
[IFO2] b => b if false end
[IFO3] b => b if false end
[FSYM] b => b if false end
[LAB] b => b if false end
[DSTR] b => b if false end
[CONCAT] b => b if false end
```

```
end
```

Uses eval\_DC 96 104b, eval\_DK 96 104b, and eval\_ONE 96 104b.

104a  $\langle$ strategies for tcstrat 104a $\rangle \equiv$  (103a)

```
strategies for Y // evaluation ELAN-strategy of the interpreter
```

```
implicit
```

```
[.] eval_ONE => dc one(dk(ONE1) , dk(ONE2)) end
```

```
[.] eval_DC => dc(dk(DC1) , dk(DC2)) end
```

```
[.] eval_DK => dk(DK1 , DK2) end
```

```
[.] eval =>
```

```
first(
```

```
 dk(FSYM) ,
```

```
 dk(DSTR) ,
```

```
 dk(DK) ,
```

```
 dk(DC) ,
```

```
 dk(ONE) ,
```

```
 dk(LAB) ,
```

```
 dc(ID) ,
```

```
 dc(CONST) ,
```

```
 dk(CONCAT) ,
```

```
 dk(IF1) , dk(IF2) ,
```

```
 dc(dk(IF01) ,
```

```
 dk(IF02)); dk(IF03))
```

```
end
```

```
end
```

Uses eval 96 104b, eval\_DC 96 104b, eval\_DK 96 104b, and eval\_ONE 96 104b.

104b  $\langle$ \* 104b $\rangle \equiv$

```
// This file is generated automatically: do not edit it directly.
```

```
 \langle the tcstrat specification 103a \rangle
```

Defines:

eval, used in chunks 93–95, 97e, 103d, and 104a.

eval\_DC, used in chunks 94d, 95, 103, and 104a.

eval\_DK, used in chunks 94d, 95, 103, and 104a.

eval\_ONE, used in chunks 94d, 95, 103, and 104a.

Uses do 89e.

## 4 Library ref

## 4.1 Meta\_Apply.eln

The module `Meta_Apply` provides several variants of the `meta-apply` function, that apply a strategy `S` on a term `t` in an environment defined by a program `P`. These three components are either encoded in the REF format, or passed as strings. Results are produced equally in the REF format, or as strings represented terms. Builtin versions of these functions work over strings of REF-terms.

```
106a <the Meta_Apply specification 106a>≡ (109b)
 module Meta_Apply
 <imports for Meta_Apply 106b>
 <operators for Meta_Apply 106c>
 <strategy operator for Meta_Apply 107c>
 <rules for Meta_Apply 108a>
 <strategies for Meta_Apply 108b>
 end
```

```
106b <imports for Meta_Apply 106b>≡ (106a)
 import
 global
 builtinInt list[string] list[Term] REF;
 local
 refstring2string ref2string[Strategy]
 ref2string[Term] ref2string[Program] string2refterm;
 end
```

Uses `refstring2string` 153 and `string2refterm` 155.

```
106c <operators for Meta_Apply 106c>≡ (106a)
 <global operators for Meta_Apply 107a>
 <local operators for Meta_Apply 107b>
```

The following operators are all called `meta_apply` and differ by their arguments:

`meta_apply(S,t:sort,P,n-th)` gives the `n`-th result of a strategy application `S` on the term `t` of sort `sort` w.r.t. the program `P`.

`meta_apply("strat","term",P,n-th)` is like the previous one, but the strategy, the term and the obtained result are in REF format.

`meta_apply(S,t:sort,P,n-th,m)` gives a list of `m` results from the `n`-th one, of a strategy application `S` on the term `t` of sort `sort` w.r.t. the program `P`. The first result has index 0.

`meta_apply("strat","term",P,n-th,m)` is like the previous one, but the strategy, the term and the obtained results are in REF format.

`strings2refterms` and `refterms2strings` are two inverse conversion functions from strings to terms in REF format and conversely.

```
107a <global operators for Meta_Apply 107a>≡ (106c)
 operators global
 meta_apply(@,@:'@,@,@) : (Strategy Term Sort Program builtinInt) Term;

 meta_apply(@,@,@,@) : (string string Program builtinInt) string;
 meta_apply(@,@:'@,@,@,@): (Strategy Term Sort Program builtinInt builtinInt)
 list[Term];
 meta_apply(@,@,@,@,@): (string list[string] Program builtinInt builtinInt)
 list[string];

 strings2refterms(@:'@,@): (list[string] string Program) list[Term];
 refterms2strings(@,@): (list[Term] Program) list[string];
```

Uses `meta_apply` 109b 82e 109b, `refterms2strings` 109b, and `strings2refterms` 109b.

The following local operators also called `meta_apply` have the following differences:

in `meta_apply(strat,term,lgi,spc,n-th)`, `lgi` and `spc` are filenames of a "constant" specification used as the program `P`.

in `meta_apply(strat,term,ref,n-th)`, `ref` is a filename of a REF file, which contains an image of a specification of the program `P`.

`meta_apply(strat,term,lgi,spc,n-th,number)` is the same as "code 170" for the list version of this builtin function.

`meta_apply(strat,term,ref,n-th,number)` is the same as "code 172" for the list version of this builtin function.

```
107b <local operators for Meta_Apply 107b>≡ (106c)
 local
 meta_apply(@,@,@,@,@) : (string string string string builtinInt) string code 170;
 meta_apply(@,@,@,@) : (string string string builtinInt)string code 172;
 meta_apply(@,@,@,@,@,@): (string list[string] string string builtinInt builtinInt)
 list[string] code 171;

 meta_apply(@,@,@,@,@) : (string list[string] string builtinInt builtinInt)
 list[string] code 173;
```

end

Uses `meta_apply` 109b 82e 109b.

```
107c <strategy operator for Meta_Apply 107c>≡ (106a)
 stratop global
 meta_apply_strategy : <Term -> Term> bs;
 end
```

Uses `meta_apply_strategy` 109b.

```

108a <rules for Meta_Apply 108a>≡ (106a) 109a▷
rules for string
str, term, lgi, spc, aux_file_name
 : string;
P, P1 : Program;
n : builtinInt;
global
[] meta_apply(str,term,SPEC(lgi,spc),n) =>
 meta_apply(str,term,lgi,spc,n) end
[] meta_apply(str,term,P,n) =>
 meta_apply(str,term,aux_file_name,n)
 where aux_file_name:=()get_aux_file_name+".ref"
 where P1:=()write_file(aux_file_name,P)
end
end

rules for list[string]
str, lgi, spc, aux_file_name
 : string;
terms : list[string];
P, P1 : Program;
frm, n : builtinInt;
global
[] meta_apply(str,terms,SPEC(lgi,spc),frm,n) =>
 meta_apply(str,terms,lgi,spc,frm,n) end
[] meta_apply(str,terms,P,frm,n) =>
 meta_apply(str,terms,aux_file_name,frm,n)
 where aux_file_name:=()get_aux_file_name+".ref"
 where P1:=()write_file(aux_file_name,P)
end
end

rules for Term
S : Strategy;
T : Term;
P : Program;
sorte : Sort;
n : builtinInt;
str : string;
term8, term9: Term;
global
[meta_apply_rule]
 meta_apply(S,T:sorte,P,n) =>
 string2refterm(str:Sort2string(sorte),P)
 where str:=(META)meta_apply(
 term2string(S),term2string(T),P,n)
end
end

```

Uses meta\_apply 109b 82e 109b, string2refterm 155, term2string 134, and write\_file 151.

```

108b <strategies for Meta_Apply 108b>≡ (106a)
strategies for Term
implicit
[.] meta_apply_strategy => dk(meta_apply_rule) end
end

```

Uses meta\_apply\_strategy 109b.

109a  $\langle$ rules for Meta\_Apply 108a $\rangle$ + $\equiv$  (106a)  $\triangleleft$ 108a

```

rules for list[Term]
S : Strategy;
T : Term;
P : Program;
sorte : Sort;
frm : builtinInt;
n : builtinInt;
x : string;
xs : list[string];
sortstr : string;
global
[] meta_apply(S,T:sorte,P,frm,n) =>
 strings2refterms(xs:Sort2string(sorte),P)
 where xs:=(META)meta_apply(
 term2string(S),term2string(T).nil, P, frm,n)
end
[] strings2refterms(nil:sortstr,P) => nil end
[] strings2refterms(x.xs:sortstr,P) => string2refterm(x:sortstr,P).
 strings2refterms(xs:sortstr,P) end
end

rules for list[string]
P : Program;
x : Term;
xs : list[Term];
global
[] refterms2strings(nil,P) => nil end
[] refterms2strings(x.xs,P) => refterm2string(x,P).refterms2strings(xs,P)
end
end

```

Uses meta\_apply 109b 82e 109b, refterms2strings 109b, refterm2string 155, string2refterm 155, strings2refterms 109b, and term2string 134.

109b  $\langle$ \* 109b $\rangle$  $\equiv$   
 // This file is generated automatically: do not edit it directly.  
 $\langle$ the Meta\_Apply specification 106a $\rangle$

Defines:

meta\_apply, used in chunks 107a, 81a, 107b, 108a, 82d, 107–109, and 111c.  
 meta\_apply\_strategy, used in chunks 107c and 108b.  
 refterms2strings, used in chunks 107a and 109a.  
 strings2refterms, used in chunks 107a and 109a.

## 4.2 REF.eln

The module REF.eln is the ELAN specification of the exchange format REF. The introduced sort Program represents a computational system as an ELAN program, which consists of signature, rules, strategies and also an input query format. This module also defines the global structure of a REF module, which represents a printed form of a program of the sort Program.

The Reduce ELAN Format FILE STRUCTURE is described below:

```

Identifiers : Tab all identifiers of a program
end separator
Sorts : Tab table of all sorts of a program
end separator
Modules : Tab table of all imported modules
end separator
RuleNames : Tab all rule names of a program
end separator
StrategyNames : Tab all strategy names
end separator
{
gram.rules : GrammarRules grammar rules of one sort
end separator
}*
EndDef end end of the symbol grammar
{
rule : RuleStatement named and unnamed rules
end separator
}*
EndDef end end of rewrite rules
{
strat : StrategyStatement definition of a built-in strategy
end separator
}*
EndDef end
Query : QueryStatement query from .lgi
end separator

```

*Warning!!! : Any change of the REF structure should be realized in all parts of the ELAN system depending on this structure, e.g. the REF parser of ELAN (module aterm.t), the REF parser of the compiler (module REFParser.jj), the pretty-printer and parser boxes (modules ref2result, query2term), and in the REF library (modules elanlib/ref/\*)*

```

110a <the REF specification 110a>≡ (114b)
 module REF
 <imports for REF 110b>
 <sorts for REF 111a>
 <operators for REF 111b>
 <rules for REF 114a>
 end

```

```

110b <imports for REF 110b>≡ (110a)
 import global
 int bool ident string

```

```

 list[Term] list[WhereStatement] list[Lexem] list[list[Lexem]]
 list[int]
 list[GrammarRule] list[GrammarRules]
 list[RuleStatement] list[StrategyStatement]
 list[SwitchBranch] list[list[WhereStatement]]
 list[TabItem] list[StrategyStatement];
end

```

111a *(sorts for REF 111a)*≡ (110a)

```

sort // Exported sorts
 Term // ELAN first-order terms
 WhereStatement // a construction of local assignment
 Rule // a rewrite rule, with lhs, rhs and local affectations
 RuleStatement // a rewrite rule with auxiliary information
 SwitchStatement // a structured condition of a rewrite rule
 SwitchBranch // one branch of a structured condition of a rewrite rule
 QueryStatement // the format of the input query
 StrategyChoice // a list of choices of a dk, dc, etc.
 Strategy // a single strategy
 StrategyStatement // a strategy definition with flags
 Lexem // an element of a grammar defining the signature
 Sort // sorts of the signature
 GrammarRule GrammarRules // rules of the signature
 TabItem Tab // tables of symbols, like rulenames, sortnames, strategynames, etc.
 Program; // a representation of an ELAN program.
end

```

111b *(operators for REF 111b)*≡ (110a)  
*(global operators for REF 111c)*

111c *(global operators for REF 111c)*≡ (111b)

```

operators global
// ----- LEXEMS
Type(@) : (int) Lexem;
Char(@) : (int) Lexem;
Num(@) : (int) Lexem;
String(@) : (int) Lexem;
Ident(@) : (int) Lexem;
Blank : Lexem;
// ----- GRAMMAR
@': '@': '@': '@': '@': '@': '@': '@': '@': '@
: (int fsym:int int int int int int int
 profile:list[Lexem]
 loc_strat:list[int]) GrammarRule;
// 1, 2, 3 as follows:
// 1 = rule belongs to topgram
// 2 = rule belongs to globgram
// 3 = belongs to both
// index of symbol (used in Terms, see FSYM)
// priority (0-007777)

```



```

// syntactic infos
// 1=RLEFTASSOC, 2=RRIGHTASSOC, 4=RBINSTR 8=PRINTABLE
// code of semantic function of a built-in symbol
// matching infos
// 0=free, 1=c, 2=ac symbol
// a flag for strategy compilation
GrammarForSort @ ':' @ ':' @ : (int int profile:list[GrammarRule]) GrammarRules;
// index of sort, gram. rules
// builtin-flag of the sort
EndDef : GrammarRules;
// ----- TABLES
@':'@ : (int string) TabItem; // for all symbol tables
@':'@ : (int list[int]) TabItem; // for visibilities
Identifiers @ : (Identifiers:list[TabItem]) Tab;
@ : (string) Sort;
Sorts @ : (Sorts:list[TabItem]) Tab;
Modules @ : (Modules:list[TabItem]) Tab;
RuleNames @ : (RuleNames:list[TabItem]) Tab;
StrategyNames @ : (StrategyNames:list[TabItem]) Tab;
// ----- TERMS
VAR(@,@) : (int int) Term; // variable, its index, type
INT(@) : (int) Term; // integer constant
IDENT(@) : (int) Term; // index of identifier
STRING(@) : (list[int]) Term; //
FSYM(@,@) : (list[Term] int) Term; // subterms, symbol code
// ----- REWRITE RULES
IFF(@) : (Term) WhereStatement; // condition
PWHERE(@,@,@) : (Term int Term) WhereStatement;
WHERE(@,@,@) : (Term int Term) WhereStatement; // variable, of pattern,
// strategy index
// rhs of where statement
TRY(@) : (list[list[WhereStatement]]) WhereStatement;

@,@,@ : (lhs:Term // lhs
rhs:Term // rhs
wheres:list[WhereStatement]) Rule;

RULE(@,@,@,@,@,@,@): (rule_name_index: int // rule name index
rule_sort: int // rule type index
module_index: int // module index of rule
rule_flags: int // infos
ac_flags: int // ahead
nof_vars: int // number of variables
rwrule: Rule) RuleStatement;

NOSWITCH(@,@) : (list[WhereStatement] Term) SwitchStatement;
// where and ifs, rhs
SWITCH(@,@) : (list[WhereStatement] list[SwitchBranch]) SwitchStatement;
@,@ : (Term SwitchStatement) SwitchBranch; // condition

@,@ : (SW_lhs:Term // lhs
SW_rhs: SwitchStatement) Rule;

SWRULE(@,@,@,@,@,@,@): (sw_rule_name_index: int // rule name index
sw_rule_sort: int // rule type index
sw_module_index: int // module index of rule
sw_rule_flags: int // infos

```

```

 sw_ac_flags:int // ahead
 sw_nof_vars:int // number of variables
 sw_rule:Rule) RuleStatement;

EndDef : RuleStatement;// terminator I
// ----- BUILT-IN STRATEGIES

DK(@) : (DK_choices:StrategyChoice) StrategyStatement;
ONE(@) : (ONE_choices:StrategyChoice) StrategyStatement;
DC(@) : (DC_choices:StrategyChoice) StrategyStatement;
@ : (Strategy) StrategyChoice;
@ , @ : (StrategyChoice Strategy) StrategyChoice;
@ , @ : (StrategyChoice Strategy) StrategyChoice;

dk(@) : (dk_choices:list[int]) StrategyStatement;
one(@) : (one_choices:list[int]) StrategyStatement;
dc(@) : (dc_choices:list[int]) StrategyStatement;

repeat*(@) : (Strategy) StrategyStatement;
repeat+(@) : (Strategy) StrategyStatement;
iterate*(@) : (Strategy) StrategyStatement;
iterate+(@) : (Strategy) StrategyStatement;
id : StrategyStatement;
fail : StrategyStatement;
call(@) : (int) StrategyStatement;

@ : (StrategyStatement) Strategy;
@ ; @ : (Strategy StrategyStatement) Strategy;

STRATEGY(@,@,@,@) : (strategy_name_index:int //index of the strategy name
 strategy_sort:int // index of strategy type
 module_index:int
 // index of module of strategy definition
 body:Strategy) // concatenation of substrategies
 StrategyStatement;
EndDef : StrategyStatement; // terminator II
// ----- LGI MODULE
QUERY(@,@,@,@,@) : (int // source type index
 int // result type index,
 int // main strategy index,
 Term // startwith term
 Term) // checkwith term
 QueryStatement;

// ----- PROGRAM
@,@,@,@,@,@,@,@,@ :
 (Identifiers:Tab // table of all identifiers
 Sorts:Tab // table of all sorts
 Modules:Tab // table of all imported modules
 RuleNames:Tab // table of rule labels
 StrategyNames:Tab // table of strategy names
 TopGlobalGrammar:list[GrammarRules]
 // intersection of two following grammars
 TopGrammar:list[GrammarRules]
 // grammar visible in .lgi module
 GlobalGrammar:list[GrammarRules] // all global symbols
 AllRules:list[RuleStatement] // all rules of a program
 AllStrategies:list[StrategyStatement]// all strategies

```

```

 Query:QueryStatement) // description of .lgi module
 Program;
//-----
SPEC(@,@) : (string string) Program code 174;
 // .lgi .spc - abbreviation of a specification:
 // a constructor introduced as an abbreviation
 // of a "constant" specification used in meta_apply

// auxiliary functions
Sort2string(@) : (Sort) string;

end

```

Uses meta\_apply 109b 82e 109b.

```

114a <rules for REF 114a>≡ (110a)
 rules for string
 type : string;
 global
 [] Sort2string(type) => type end
 end

```

```

114b <* 114b>≡
 // This file is generated automatically: do not edit it directly.
 <the REF specification 110a>

```

### 4.3 ref2string.eln

The module `ref2string` parameterised by the sort `X` provides two conversion functions between strings of REF-terms and terms of the user's defined signature. It also exports a pretty-printing and a parsing function parameterized by a signature in the REF-format.

115a *(the ref2string specification 115a)*≡ (134)

```

module ref2string[X]
 <imports for ref2string 115b>
 <sorts for ref2string 115c>
 <operators for ref2string 115d>
 <rules for ref2string 115f>

end

```

115b *(imports for ref2string 115b)*≡ (115a)

```

import global string REF
 divers2string//[X]
 refstring2string;

local io[X];
end

```

Uses `refstring2string` 153.

115c *(sorts for ref2string 115c)*≡ (115a)

```

sort X; end

```

115d *(operators for ref2string 115d)*≡ (115a)

```

<global operators for ref2string 115e>

```

115e *(global operators for ref2string 115e)*≡ (115d)

```

operators global
 refstring2term(@) : (string) X code -168; // in meta.c
 term2refstring(@) : (X) string code -167; // in meta.c

 //----- this work w.r.t. ThisProgram
 term2string(@) : (X) string code -169; // prettyprinter
 //----- general prettyprinter
 term2string(@,@) : (X Program) string; // prettyprinter
 term2string(@,@) : (X string) string; // prettyprinter
 //----- general parser
 string2term(':'@,@) : (string Sort Program) X; // parser
 string2term(':'@,@) : (string string string) X; // parser

end

```

Uses `refstring2term` 134, `string2term` 134, `term2refstring` 134, `term2string` 134, and `ThisProgram` 153.

115f *(rules for ref2string 115f)*≡ (115a)

```

rules for string
 t : X;
 P : Program;
 fname : string;
global

```

```

[] term2string(t,P) => refstring2string(
 term2refstring(t),P) end
[] term2string(t,fname) => refstring2string(
 term2refstring(t),fname) end
end

rules for X
P : Program;
sorte : Sort;
fname : string;
s, sorte : string;
global
[] string2term(s:sorte,P) =>
 refstring2term(string2refstring(s:sorte,P))
end
[] string2term(s:sorte, fname) =>
 refstring2term(string2refstring(s:sorte,
 fname))
end
end

```

Uses `refstring2string` 153, `refstring2term` 134, `string2refstring` 153, `string2term` 134, `term2refstring` 134, and `term2string` 134.

134 < \* 134 > ≡  
 // This file is generated automatically: do not edit it directly.  
 < the ref2string specification 115a >

Defines:

```

refstring2term, used in chunks 115 and 117f.
string2term, used in chunks 115, 117f, and 154f.
term2refstring, used in chunks 115 and 117f.
term2string, used in chunks 108a, 109a, 115, 117f, and 154f.

```

#### 4.4 ref2term.eln

The module `ref2term[X]` parameterized by a sort `X` implements two conversion functions from a term to its REF format and back.

```
117a <the ref2term speci cation 117a>≡ (123)
 module ref2term[X]
 <imports for ref2term 117b>
 <sorts for ref2term 117c>
 <operators for ref2term 117d>
 <rules for ref2term 117f>
 end
```

Uses `ref2term` 123.

```
117b <imports for ref2term 117b>≡ (117a)
 import global string REF ref2string[Term]; end
```

```
117c <sorts for ref2term 117c>≡ (117a)
 sort X; end
```

```
117d <operators for ref2term 117d>≡ (117a)
 <global operators for ref2term 117e>
```

```
117e <global operators for ref2term 117e>≡ (117d)
 operators global
 ref2term(@) : (Term) X;
 term2ref(@) : (X) Term;
 end
```

Uses `ref2term` 123 and `term2ref` 123.

```
117f <rules for ref2term 117f>≡ (117a)
 rules for Term
 x : X;
 global
 [] term2ref(x) => string2term(
 term2refstring(x):"Term","REF.ref")
 end
 end

 rules for X
 t : Term;
 global
 [] ref2term(t) => refstring2term(term2string(t,"REF.ref"))
 end
 end
```

Uses `refstring2term` 134, `ref2term` 123, `string2term` 134, `term2ref` 123, `term2refstring` 134, and `term2string` 134.

```
123 <* 123>≡
 // This file is generated automatically: do not edit it directly.
 <the ref2term speci cation 117a>
```

Defines:

```
ref2term, used in chunk 117.
term2ref, used in chunk 117.
```

## 4.5 ref\_idents.eln

133a *<the ref\_idents specification 133a>*≡ (134)

```

module ref_idents
 <imports for ref_idents 133b>
 <operators for ref_idents 133c>
 <rules for ref_idents 133f>
end

```

133b *<imports for ref\_idents 133b>*≡ (133a)

```

import global
 int
 REF
 ref_tables
 list[TabItem]
;
end

```

133c *<operators for ref\_idents 133c>*≡ (133a)

```

<global operators for ref_idents 133d>

```

133d *<global operators for ref\_idents 133d>*≡ (133c)

```

operators global
 get_ident(@,@) : (int Program) string;
 get_ident(@,@) : (int Tab) string;
end

```

Uses `get_ident` 134.

133f *<rules for ref\_idents 133f>*≡ (133a)

```

rules for string
 tis : list[TabItem];
 P : Program;
 n : int;
global
 [] get_ident(n,P) => get_ident(n,P.Identifiers) end
 [] get_ident(n,Identifiers tis) => get_string(n,tis) end
end

```

Uses `get_ident` 134 and `get_string` 134.

134 *<\* 134>*≡

```

// This file is generated automatically: do not edit it directly.
<the ref_idents specification 133a>

```

Defines:

`get_ident`, used in chunks 133 and 148f.



## 4.6 ref\_modules.eln

- 120a *<the ref\_modules specification 120a>*≡ (120g)
- ```

module ref_modules
  <imports for ref_modules 120b>
  <operators for ref_modules 120c>
  <rules for ref_modules 120f>
end

```
- 120b *<imports for ref_modules 120b>*≡ (120a)
- ```

import global
 int
 REF
 ref_tables
 list[TabItem]
;
end

```
- 120c *<operators for ref\_modules 120c>*≡ (120a)
- ```

  <global operators for ref_modules 120d>
  <local operators for ref_modules 120e>

```
- 120d *<global operators for ref_modules 120d>*≡ (120c)
- ```

operators global
 get_module(@,@) : (int Program) string;

```
- Uses `get_module` 120g.
- 120e *<local operators for ref\_modules 120e>*≡ (120c)
- ```

local
  get_module(@,@) : (int Tab) string;
end

```
- Uses `get_module` 120g.
- 120f *<rules for ref_modules 120f>*≡ (120a)
- ```

rules for string
tis : list[TabItem];
P : Program;
n : int;
global
 [] get_module(n,P) => get_module(n,P.Modules) end
 [] get_module(n,Modules tis) => get_string(n,tis) end
end

```
- Uses `get_module` 120g and `get_string` 134.
- 120g *<\* 120g>*≡
- ```

// This file is generated automatically: do not edit it directly.
<the ref_modules specification 120a>

```

Defines:

`get_module`, used in chunk 120.

4.7 ref_read.eln

Read functions for a program in REF format.

- 121a *(the ref_read specification 121a)*≡ (123)
- ```

module ref_read[verbose]
 <imports for ref_read 121b>
 <operators for ref_read 121c>
 <rules for ref_read 121f>
end

```
- 121b *(imports for ref\_read 121b)*≡ (121a)
- ```

import global
  REF list[GrammarRules]
  string io[string]
  eq[GrammarRules] eq[RuleStatement] eq[StrategyStatement]
  io[RuleStatement] io[list[RuleStatement]]
  io[Tab] prompt[Tab]
  io[GrammarRules] prompt[list[GrammarRules]]
  io[list[GrammarRules]]
  io[StrategyStatement]
  io[list[StrategyStatement]] prompt[list[StrategyStatement]]
  io[QueryStatement]
  ;
end

```
- 121c *(operators for ref_read 121c)*≡ (121a)
- ```

<global operators for ref_read 121d>
<local operators for ref_read 121e>

```
- 121d *(global operators for ref\_read 121d)*≡ (121c)
- ```

operators global
  read_ref_file(@)      : (string) Program;
  read_file(@)         : (string) Program;

```
- Uses read_file 123 and read_ref_file 123.
- 121e *(local operators for ref_read 121e)*≡ (121c)
- ```

local
 read_rules(@) : (Pid) list[RuleStatement];
 read_syntab(@): (Pid) list[GrammarRules];
 read_strat(@) : (Pid) list[StrategyStatement];
end

```
- 121f *(rules for ref\_read 121f)*≡ (121a)
- ```

rules for Program
  pid, pid1          : Pid;
  ref_file_name      : string;
  void0, void1, void2, void3,
  void4, void5, void6, void7,
  void8, void9, void10, void11,

```

```

void12          : string;
r               : RuleStatement;
rs, rs1        : list[RuleStatement];
strats, strats1 : list[StrategyStatement];
idtab, idtab1,
sorttab, sorttab1,
Modules, Modules1,
rnamestab, rnamestab1,
snamestab, snamestab1
                : Tab;
TopGlobalGrammar, topGlobalGrammar1      : list[GrammarRules];
TopGrammar, TopGrammar1                  : list[GrammarRules];
GlobalGrammar, GlobalGrammar1            : list[GrammarRules];
Query, Query1                            : QueryStatement;
global
[]    read_ref_file(ref_file_name) =>
      read_file(ref_file_name+".ref")
end
[]    read_file(ref_file_name) =>
      idtab,sorttab,Modules,
      rnamestab,snamestab,
      TopGlobalGrammar,TopGrammar,GlobalGrammar,
      rs,strats,Query
      //--- how to get it ---
      where pid := ()open(ref_file_name,"r")

      where idtab := ()read(pid)
IF verbose == 1 : {where void0 := ()write(stdout,"--- Identifiers is loaded\n")}
      where sorttab := ()read(pid)
      //where sorttab1 := ()write(stdout,sorttab)
IF verbose == 1 : {where void1 := ()write(stdout,"--- Sorts is loaded\n")}
      where Modules := ()read(pid)
      //where Modules1 := ()write(stdout,Modules)
IF verbose == 1 : {where void2 := ()write(stdout,"--- Modules is loaded\n")}
      where rnamestab := ()read(pid)
      //where rnamestab1 := ()write(stdout,rnamestab)
IF verbose == 1 : {where void5 := ()write(stdout,"--- RuleNames tab is loaded\n")}
      where snamestab := ()read(pid)
      //where snamestab1 := ()write(stdout,snamestab)
IF verbose == 1 : {where void6 := ()write(stdout,"--- StrategyNames tab is loaded\n")}
      where TopGlobalGrammar := ()read_syntab(pid)
      //where TopGlobalGrammar1 := ()write(stdout,topGlobalGrammar)
IF verbose == 1 : {where void4 := ()write(stdout,"--- TopGlobalGrammar is loaded\n")}
      where TopGrammar := ()read_syntab(pid)
      //where TopGrammar1 := ()write(stdout,TopGrammar)
IF verbose == 1 : {where void10 := ()write(stdout,"--- TopGrammar is loaded\n")}
      where GlobalGrammar := ()read_syntab(pid)
      //where GlobalGrammar1 := ()write(stdout,GlobalGrammar)
IF verbose == 1 : {where void11 := ()write(stdout,"--- GlobalGrammar is loaded\n")}
      where rs := ()read_rules(pid)
      //where rs1 := ()write(stdout,rs)

```

```

IF verbose == 1 : {where void7 := ()write(stdout,"--- AllRules is loaded\n")}

                where strats := ()read_strat(pid)
                  //where strats1 := ()write(stdout,strats)
IF verbose == 1 : {where void8 := ()write(stdout,"--- AllStrategies is loaded\n")}

                where Query := ()read(pid)
                  //where Query1 := ()write(stdout,Query)
IF verbose == 1 : {where void9 := ()write(stdout,"--- query is loaded\n")}

                where pid1 := ()close(pid) end
end

rules for list[RuleStatement]
pid      : Pid;
r        : RuleStatement;
rs       : list[RuleStatement];
global
  []     read_rules(pid) => r.rs
        where r:=()read(pid)
          if neq_RuleStatement(r, EndDef)
            where rs:=()read_rules(pid)

        end
  []     read_rules(pid) => nil
        end
end

rules for list[GrammarRules]
pid      : Pid;
fs, fs1  : GrammarRules;
fss      : list[GrammarRules];
global
  []     read_syntab(pid) => fs.fss
        where fs:=()read(pid)
          if neq_GrammarRules(fs,EndDef)
            where fss:=()read_syntab(pid)

        end
  []     read_syntab(pid) => nil
        end
end

rules for list[StrategyStatement]
pid      : Pid;
fs, fs1  : StrategyStatement;
fss      : list[StrategyStatement];
global
  []     read_strat(pid) => fs.fss
        where fs:=()read(pid)
          if neq_StrategyStatement(fs, EndDef)
            where fss:=()read_strat(pid)

        end
  []     read_strat(pid) => nil
        end
end

```

```
123  ⟨ * 123 ⟩ ≡  
      // This file is generated automatically: do not edit it directly.  
      ⟨ the ref_read specification 121a ⟩
```

Defines:

```
read_file, used in chunks 121 and 152e.  
read_ref_file, used in chunk 121.
```

4.8 ref_rules.eln

Functions for the rule set of a program in REF format.

148a *(the ref_rules speci cation 148a)*≡ (151)

```

module ref_rules
  (imports for ref_rules 148b)
  (operators for ref_rules 148c)
  (rules for ref_rules 148f)
end

```

148b *(imports for ref_rules 148b)*≡ (148a)

```

import global
  int
  REF
  ref_tables ref_terms ref_strategies
  list[RuleStatement]    list[WhereStatement]
  io[Term] io[string]
  ;
end

```

148c *(operators for ref_rules 148c)*≡ (148a)

```

(global operators for ref_rules 148d)
(local operators for ref_rules 148e)

```

148d *(global operators for ref_rules 148d)*≡ (148c)

```

operators global
  nof_rules(@)      : (Program) int;
  nof_named_rules(@) : (Program) int;
  get_rule(@,@)     : (int Program) RuleStatement;

  get_rule_name(@,@) : (int Program) string;
  get_rule_name(@,@) : (int Tab) string;

  /*
  gen_new_rule_name(@) : (Program) int;
  */
  add_rule_name(@,@,@) : (int string Tab) Tab;

  add_rule_name(@,@)   : (string Program) Program;
  add_rule_name(@,@)   : (string Tab) Tab;
  member_rule_name(@,@) : (string Program) int;
  member_rule_name(@,@) : (string Tab) int;

  print_rule(@,@,@)   : (Pid RuleStatement Program) int;
  print_wheres(@,@,@) : (Pid list[WhereStatement] Program) int;

```

Uses add_rule_name 151, gen_new_rule_name 151, get_rule 151, get_rule_name 151, member_rule_name 151, nof_named_rules 151, nof_rules 151, print_rule 151, and print_wheres 151.

```

148e  <local operators for ref_rules 148e>≡ (148c)
      local
        nof_named_rules(@)    : (list[RuleStatement]) int;
        get_rule(@,@)        : (int list[RuleStatement]) RuleStatement;
      end

```

Uses `get_rule` 151 and `nof_named_rules` 151.

```

148f  <rules for ref_rules 148f>≡ (148a)
      rules for string
      ind      : int;
      P        : Program;
      tab      : list[TabItem];
      global
        [] get_rule_name(-1,P) => ""                end
        [] get_rule_name(ind,P) =>
            get_rule_name(ind,P.RuleNames)         end
        [] get_rule_name(ind,RuleNames tab) =>
            get_string(ind,tab)                     end
      end

      rules for int
      pid      : Pid;
      Rule     : RuleStatement;
      P        : Program;
      lhs_void, rhs_void, where_void : int;
      void1, void2, void3, void4, void : string;
      wh       : WhereStatement;
      whs      : list[WhereStatement];
      var, var_void : Term;
      loop,sindex,str_void : int;
      t        : Term;
      global
        [] print_rule(pid,Rule,P) => 999
            where lhs_void:=()print(pid,Rule.rwrule.lhs:Rule.rule_sort,P)
            where void:=()write(pid," => ")
            where rhs_void:=()print(pid,Rule.rwrule.rhs:Rule.rule_sort,P)
            where void1:=()write(pid,"\n")
            where where_void:=()print_wheres(pid,Rule.rwrule.wheres,P)
        end
        [] print_wheres(pid,nil,P) => 999
        end
        [] print_wheres(pid,IFF(t).whs,P) => 999
            where void:=()write(pid,"\tif ")
            where lhs_void:=()print(pid,t,P)
            where void3:=()write(pid,"\n")
            where loop:=()print_wheres(pid,whs,P)
        end
        [] print_wheres(pid,WHERE(var,-1,t).whs,P) => 999
            where void:=()write(pid,"\twhere ")
            where var_void:=()write(pid,var)
            where void1:=()write(pid,":=()")
            where lhs_void:=()print(pid,t,P)
            where void3:=()write(pid,"\n")
            where loop:=()print_wheres(pid,whs,P)
        end
        [] print_wheres(pid,WHERE(var,sindex,t).whs,P) => 999

```

```

        where void:=()write(pid,"\twhere ")
        where var_void:=()write(pid,var)
        where void1:=()write(pid,":=")
        where void4:=()write(pid,get_strategy_name(sindex,P))
        where void2:=()write(pid,")")
        where lhs_void:=()print(pid,t,P)
        where void3:=()write(pid,"\n")
        where loop:=()print_wheres(pid,whs,P)
    end
end

rules for Program
s      : string;
P, P1  : Program;
ind    : int;
global
    [] add_rule_name(s,P) =>
        P[.RuleNames<-add_rule_name(s,P.RuleNames)]
    end
end

rules for Tab
ind      : int;
name     : string;
tab      : list[TabItem];
global
    [] add_rule_name(ind,name, RuleNames tab) =>
        RuleNames add_string(ind,name,tab)      end
    end

rules for Tab
tis1,tis : list[TabItem];
s        : string;
global
    [] add_rule_name(s,RuleNames tis) => RuleNames tis1
        // cf.constants in rtdatas.h
        where tis1:=()add_hash_string(s,tis,2000)
    end
end

rules for int
P      : Program;
Rule   : RuleStatement;
Rules  : list[RuleStatement];
global
    [] nof_rules(P)      => size(P.AllRules)      end
    [] nof_named_rules(P) => nof_named_rules(P.AllRules) end
    [] nof_named_rules(nil) => 0                  end
    [] nof_named_rules(Rule.Rules) => nof_named_rules(Rules)
        if Rule.rule_name_index == -1          end
    [] nof_named_rules(Rule.Rules) => 1+nof_named_rules(Rules) end
/*
    [] gen_new_rule_name(P) =>
        new_string_index(P.RuleNames.RuleNames)    end
*/
end

```



```

rules for int
ind      : int;
ti       : TabItem;
tis      : list[TabItem];
s        : string;
tsize, n, m : int;
P        : Program;
global
  [] member_rule_name(s,P) =>
      member_rule_name(s,P.RuleNames)
  end
  [] member_rule_name(s,RuleNames tis) =>
      member_string(s,tis)
end

```

```

rules for RuleStatement
P      : Program;
rindex : int;
Rule   : RuleStatement;
Rules  : list[RuleStatement];
global
  [] get_rule(rindex,P)      => get_rule(rindex,P.AllRules)
  end
  [] get_rule(rindex,Rule.Rules) => Rule
      if Rule.rule_name_index == rindex
  end
  [] get_rule(rindex,Rule.Rules) => get_rule(rindex,Rules)
  end
end

```

Uses add_hash_string 134, add_rule_name 151, add_string 134, gen_new_rule_name 151, get_rule 151, get_rule_name 151, get_strategy_name 132, get_string 134, member_rule_name 151, member_string 134, new_string_index 134, nof_named_rules 151, nof_rules 151, print 151, print_rule 151, and print.wheres 151.

```

151 < * 151 >≡
    // This file is generated automatically: do not edit it directly.
    < the ref_rules specification 148a >

```

Defines:

```

add_rule_name, used in chunk 148.
gen_new_rule_name, used in chunk 148.
get_rule, used in chunk 148.
get_rule_name, used in chunk 148.
member_rule_name, used in chunk 148.
nof_named_rules, used in chunk 148.
nof_rules, used in chunk 148.
print_rule, used in chunk 148.
print.wheres, used in chunk 148.

```

4.9 ref_sorts.eln

- 148a *<the ref_sorts specification 148a>*≡ (151)
- ```

module ref_sorts
 <imports for ref_sorts 148b>
 <operators for ref_sorts 148c>
 <rules for ref_sorts 148f>
end

```
- 148b *<imports for ref\_sorts 148b>*≡ (148a)
- ```

import global
  int
  REF
  ref_tables
  list[TabItem]
;
end

```
- 148c *<operators for ref_sorts 148c>*≡ (148a)
- ```

<global operators for ref_sorts 148d>
<local operators for ref_sorts 148e>

```
- 148d *<global operators for ref\_sorts 148d>*≡ (148c)
- ```

operators global
  get_sort(@,@)      : (int Program) string;

```
- Uses `get_sort` 151.
- 148e *<local operators for ref_sorts 148e>*≡ (148c)
- ```

local
 get_sort(@,@) : (int Tab) string;
end

```
- Uses `get_sort` 151.
- 148f *<rules for ref\_sorts 148f>*≡ (148a)
- ```

rules for string
  tis      : list[TabItem];
  P        : Program;
  n        : int;
global
  [] get_sort(n,P) => get_sort(n,P.Sortes)      end
  [] get_sort(n,Sortes tis) => get_string(n,tis) end
end

```
- Uses `get_sort` 151 and `get_string` 134.
- 151 *<* 151>*≡
- ```

// This file is generated automatically: do not edit it directly.
<the ref_sorts specification 148a>

```

Defines:

`get_sort`, used in chunk 148.

## 4.10 ref\_strategies.eln

Functions for the strategies set of a program in REF format.

130a *(the ref\_strategies specification 130a)*≡ (132)

```

module ref_strategies
 (imports for ref_strategies 130b)
 (operators for ref_strategies 130c)
 (rules for ref_strategies 131)
end

```

130b *(imports for ref\_strategies 130b)*≡ (130a)

```

import global
 int
 REF
 ref_tables
 list[StrategyStatement]
;
end

```

130c *(operators for ref\_strategies 130c)*≡ (130a)

```

(global operators for ref_strategies 130d)
(local operators for ref_strategies 130e)

```

130d *(global operators for ref\_strategies 130d)*≡ (130c)

```

operators global
 get_strategy(@,@) : (int Program) StrategyStatement;

 get_strategy_name(@,@) : (int Program) string;
 get_strategy_name(@,@) : (int Tab) string;

 add_strategy_name(@,@,@) : (int string Tab) Tab;

 add_new_strategy(@,@,@) : (int StrategyStatement Program)
 Program;

 add_strategy_name(@,@) : (string Program) Program;
 add_strategy_name(@,@) : (string Tab) Tab;
 member_strategy_name(@,@) : (string Program) int;
 member_strategy_name(@,@) : (string Tab) int;

```

Uses add\_strategy\_name 132, get\_strategy 132, get\_strategy\_name 132, and member\_strategy\_name 132.

130e *(local operators for ref\_strategies 130e)*≡ (130c)

```

local
 get_strategy(@,@) : (int list[StrategyStatement])
 StrategyStatement;
end

```

Uses get\_strategy 132.

```

131 <rules for ref_strategies 131>≡ (130a)
 rules for string
 ind : int;
 P : Program;
 tab : list[TabItem];
 global
 [] get_strategy_name(-1,P) => "" end
 [] get_strategy_name(ind,P) =>
 get_strategy_name(ind,P.StrategyNames) end
 [] get_strategy_name(ind,StrategyNames tab) =>
 get_string(ind,tab) end
 end

 rules for Program
 s : string;
 P, P1 : Program;
 new_str,
 str : StrategyStatement;
 ind : int;
 global
 [] add_strategy_name(s,P) =>
 P[.StrategyNames<-add_strategy_name(s,P.StrategyNames)]
 end
 [] add_new_strategy(ind,str,P) => P1
 where new_str:=()str[.strategy_name_index<-ind]
 where P1:=()P[.AllStrategies<-new_str.P.AllStrategies]
 end
 end

 rules for int
 P : Program;
 ind : int;
 ti : TabItem;
 tis : list[TabItem];
 s : string;
 tsize, n, m : int;
 global
 [] member_strategy_name(s,P) =>
 member_strategy_name(s,P.StrategyNames)
 end
 [] member_strategy_name(s,StrategyNames tis) =>
 member_string(s,tis) end
 end

 rules for Tab
 ind : int;
 name : string;
 tab : list[TabItem];
 global
 [] add_strategy_name(ind, name, StrategyNames tab) =>
 StrategyNames add_string(ind, name, tab) end
 end

 rules for Tab
 tis1,tis : list[TabItem];

```

```

s : string;
global
 [] add_strategy_name(s,StrategyNames tis) => StrategyNames tis1
 // cf.constants in rtdatas.h
 where tis1:=()add_hash_string(s,tis,500)
 end
end

rules for StrategyStatement
P : Program;
sindex : int;
Strategy : StrategyStatement;
Strategies : list[StrategyStatement];
global
 [] get_strategy(sindex,P) => get_strategy(sindex,P.AllStrategies)
 end
 [] get_strategy(sindex,Strategy.Strategies) => Strategy
 if Strategy.strategy_name_index == sindex
 end
 [] get_strategy(sindex,Strategy.Strategies) =>
 get_strategy(sindex,Strategies)
 end
end

```

Uses add\_hash\_string 134, add\_strategy\_name 132, add\_string 134, get\_strategy 132, get\_strategy\_name 132, get\_string 134, member\_strategy\_name 132, and member\_string 134.

132 < \* 132 >≡  
 // This file is generated automatically: do not edit it directly.  
 < the ref\_strategies speci cation 130a >

Defines:

```

add_strategy_name, used in chunks 130d and 131.
gen_new_strategy_name, never used.
get_strategy, used in chunks 130 and 131.
get_strategy_name, used in chunks 148f, 130d, and 131.
member_strategy_name, used in chunks 130d and 131.

```

## 4.11 ref\_tables.eln

- 133a *(the ref\_tables speci cation 133a)*≡ (134)
- ```

module ref_tables
  <imports for ref_tables 133b>
  <operators for ref_tables 133c>
  <rules for ref_tables 133f>
end

```
- 133b *(imports for ref_tables 133b)*≡ (133a)
- ```

import global
 int bool
 REF
 ;
end

```
- 133c *(operators for ref\_tables 133c)*≡ (133a)
- ```

<global operators for ref_tables 133d>
<local operators for ref_tables 133e>

```
- 133d *(global operators for ref_tables 133d)*≡ (133c)
- ```

operators global
 get_string(@,@) : (int list[TabItem]) string;
 new_string_index(@) : (list[TabItem]) int;

 add_string(@,@,@) : (int string list[TabItem]) list[TabItem];
 member_string(@,@) : (string list[TabItem]) int;
 member(@,@) : (int list[TabItem]) bool;
 add_hash_string(@,@,@) : (string list[TabItem] int) list[TabItem];
 hash_function(@,@,@,@) : (int string int int) int;

```
- Uses add\_hash\_string 134, add\_string 134, get\_string 134, hash\_function 134, member 134 134, member\_string 134, and new\_string\_index 134.
- 133e *(local operators for ref\_tables 133e)*≡ (133c)
- ```

local
  add_hash_loop(@,@,@,@) : (string list[TabItem] int int) list[TabItem];
end

```
- 133f *(rules for ref_tables 133f)*≡ (133a)
- ```

rules for bool
 i, ind : int;
 ti : string;
 tis : list[TabItem];
global
 [] member(i,nil) => false end
 [] member(i,i:ti.tis) => true end
 [] member(i,ind:ti.tis) => member(i,tis) end
end

rules for list[TabItem]
 ind, ind1 : int;

```

```

hash, tsize : int;
ti : TabItem;
tis : list[TabItem];
s, s1 : string;
global
 [] add_string(ind,s,nil) => ind:s.nil end
 [] add_string(ind,s,ind1:s1.tis) => ind:s.ind1:s1.tis
 if ind < ind1 end
 [] add_string(ind,s,ind1:s1.tis) => ind1:s1.
 add_string(ind,s,tis) end

 [] add_hash_string(s,tis,tsize) =>
 add_hash_loop(s,tis,tsize,hash)
 where hash:=()hash_function(strlen(s),s,0,tsize)
end
 [] add_hash_loop(s,tis,tsize,hash) => add_string(hash,s,tis)
 if not member(hash,tis)
end
 [] add_hash_loop(s,tis,tsize,hash) =>
 add_hash_loop(s,tis,tsize,(hash+211)%tsize)
end
end

rules for int
ind : int;
ti : TabItem;
tis : list[TabItem];
s : string;
tsize, n, m : int;
global
 [] member_string(s,nil) => -1 end
 [] member_string(s,ind:s.tis) => ind end
 [] member_string(s,ti.tis) => member_string(s,tis) end

 [] new_string_index(nil) => 1 end
 [] new_string_index(ind:s.nil) => ind+1 end
 [] new_string_index(ti.tis) => new_string_index(tis) end

 [] hash_function(0,s,m,tsize) => m % tsize end
 [] hash_function(n,s,m,tsize) =>
 hash_function(n-1,s,s[n-1]+m,tsize) end
end

rules for string
ind : int;
ti : TabItem;
tis : list[TabItem];
s : string;
global
 [] get_string(ind,nil) => "" end
 [] get_string(ind,ind:s.tis) => s end
 [] get_string(ind,ti.tis) => get_string(ind,tis) end
end

```

Uses add\_hash\_string 134, add\_string 134, get\_string 134, hash\_function 134, member 134 134, member\_string 134, and new\_string\_index 134.

```
134 (* 134)≡
 // This file is generated automatically: do not edit it directly.
 ⟨the ref_tables speci cation 133a⟩
```

Defines:

```
add_hash_string, used in chunks 148f, 131, and 133.
add_string, used in chunks 148f, 131, and 133.
get_string, used in chunks 133f, 120f, 148, 131, and 133.
hash_function, used in chunk 133.
member, used in chunk 133.
member_string, used in chunks 148f, 131, and 133.
new_string_index, used in chunks 148f and 133.
```



## 4.12 ref\_terms.eln

148a *<the ref\_terms specification 148a>*≡ (151)

```

module ref_terms
 <imports for ref_terms 148b>
 <operators for ref_terms 148c>
 <rules for ref_terms 148f>
end

```

148b *<imports for ref\_terms 148b>*≡ (148a)

```

import global
 int
 REF
 ref_idents
 io[string] io[int]
 io[list[GrammarRules]]
;
end

```

148c *<operators for ref\_terms 148c>*≡ (148a)

```

<global operators for ref_terms 148d>
<local operators for ref_terms 148e>

```

148d *<global operators for ref\_terms 148d>*≡ (148c)

```

operators global
 print(@,@:'@,@) : (Pid Term int Program) int;
 print_Fsym(@:'@,@,@,@): (list[Term] int Pid int Program) int;

 print(@,@,@) : (Pid Term Program) int;
 print_Fsym(@,@,@,@) : (list[Term] Pid int Program) int;

 max_var(@) : (Term) int;
 shift_vars(@,@) : (Term int) Term;

```

Uses max\_var 151, print 151, print\_Fsym 151, and shift\_vars 151.

148e *(local operators for ref\_terms 148e)*≡ (148c)

```

local
 print(@,@,@) : (Pid list[Term] Program) int;
 max_var(@,@) : (Term int) int;
 max_var(@,@) : (list[Term] int) int;
 shift_vars(@,@) : (list[Term] int) list[Term];

 print_Fsym(@:'@,@,@,@,@) : (list[Term] int Program Pid
 int list[GrammarRules]) int;
 print_fs(@:'@,@,@,@,@) : (list[Term] int Program Pid
 int GrammarRules) int;

 print_Fsym(@,@,@,@,@) : (list[Term] Program Pid int list[GrammarRules]) int;
 print_fs(@,@,@,@,@) : (list[Term] Program Pid int GrammarRules) int;

 print_fsym(@,@,@,@,@) : (list[Term] Program Pid
 int list[GrammarRule]) int;
 print_lexem(@,@,@,@) : (list[Term] Program Pid
 list[Lexem]) int;

end

```

Uses max\_var 151, print 151, print\_Fsym 151, and shift\_vars 151.

148f *(rules for ref\_terms 148f)*≡ (148a)

```

rules for int
 t : Term;
 P : Program;
 ts : list[Term];
 pid : Pid;
 n, m, n_void, m_void : int;
 sorte : int;
 void,
 void1,
 void2 : string;
global
 [] print(pid,INT(n),P) => 999
 where n_void:=()write(pid,n) end
 [] print(pid,VAR(n,m),P) => 999
 where void1:=()write(pid,"VAR("
 where n_void:=()write(pid,n)
 where void2:=()write(pid,")") end
 [] print(pid,FSYM(ts,n),P) => 999
 where n_void:=()print_Fsym(ts,pid,n,P) end

 [] print(pid,INT(n):sorte,P) => 999
 where n_void:=()write(pid,n) end
 [] print(pid,VAR(n,m):sorte,P) => 999
 where void1:=()write(pid,"VAR("
 where n_void:=()write(pid,n)
 where void2:=()write(pid,")") end
 [] print(pid,FSYM(ts,n):sorte,P) => 999
 where n_void:=()print_Fsym(ts:sorte,pid,n,P) end

end

rules for int
 t : Term;
 mv : int;

```

```

i,j : int;
is : list[Term];
global
 [] max_var(t) => max_var(t,-1) end
 [] max_var(VAR(i,j),mv) => i if i >= mv end
 [] max_var(FSYM(is,i),mv) => max_var(is,mv) end
//else
 [] max_var(t,mv) => mv end
 [] max_var(nil,mv) => mv end
 [] max_var(t.is,mv) => max_var(is,max_var(t,mv)) end
end

rules for Term
t : Term;
mv : int;
i,j : int;
is : list[Term];
global
 [] shift_vars(VAR(i,j),mv) => VAR(i+mv,j) end
 [] shift_vars(FSYM(is,i),mv) => FSYM(shift_vars(is,mv),i) end
 [] shift_vars(t,mv) => t end
end

rules for list[Term]
t : Term;
mv : int;
i,j : int;
is : list[Term];
global
 [] shift_vars(nil,mv) => nil end
 [] shift_vars(t.is,mv) => shift_vars(t,mv).shift_vars(is,mv) end
end

rules for int
pid : Pid;
void : string;
P : Program;
gr : GrammarRules;
grs : list[GrammarRules];
fsym,
n,m,
n_void,
m_void,
bin,
sorte : int;
grrule : GrammarRule;
grrules : list[GrammarRule];
ls : list[Lexem];
t : Term;
ts : list[Term];
global
// *** with sort
 [] print_Fsym(ts:sorte,pid,fsym,P) =>
 print_Fsym(ts:sorte,P,pid,fsym,P.TopGlobalGrammar)
end

 [] print_Fsym(ts:sorte,P,pid,fsym,nil) => 0 end

```

```

[] print_Fsym(ts:sorte,P,pid,fsym,gr.grs) => n_void
 where n_void:=(λ)print_fs(ts:sorte,P,pid,fsym,gr)
 if n_void > 0
 end
[] print_Fsym(ts:sorte,P,pid,fsym,gr.grs) => n_void
 where n_void:=(λ)print_Fsym(ts:sorte,P,pid,fsym,grs)
 end

[] print_fs(ts:sorte,P,pid,fsym,GrammarForSort sorte:bin:grrules)
 =>
 print_fsym(ts,P,pid,fsym,grrules)
 end
[] print_fs(ts:sorte,P,pid,fsym,gr) => 0
 end

// *** without sort
[] print_Fsym(ts,pid,fsym,P) =>
 print_Fsym(ts,P,pid,fsym,P.TopGlobalGrammar)
end

[] print_Fsym(ts,P,pid,fsym,nil) => 0
 end
[] print_Fsym(ts,P,pid,fsym,gr.grs) => n_void
 where n_void:=(λ)print_fs(ts,P,pid,fsym,gr)
 if n_void > 0
 end
[] print_Fsym(ts,P,pid,fsym,gr.grs) => n_void
 where n_void:=(λ)print_Fsym(ts,P,pid,fsym,grs)
 end

[] print_fs(ts,P,pid,fsym,GrammarForSort sorte:bin:grrules)
 =>
 print_fsym(ts,P,pid,fsym,grrules)
 end
[] print_fs(ts,P,pid,fsym,gr) => 0
 end
// ***

[] print_fsym(ts,P,pid,fsym,nil) => 0
 end
[] print_fsym(ts,P,pid,fsym,grrule.grrules) => n_void
 if grrule.fsym == fsym
 where n_void:=(λ)print_lexem(ts,P,pid,grrule.profile)
 end
[] print_fsym(ts,P,pid,fsym,grrule.grrules) => n_void
 where n_void:=(λ)print_fsym(ts,P,pid,fsym,grrules)
 end

[] print_lexem(ts,P,pid,nil) => 999
 end
[] print_lexem(t.ts,P,pid,Type(sorte).ls) => n_void
 where m_void:=(λ)print(pid,t:sorte,P)
 where n_void:=(λ)print_lexem(ts,P,pid,ls)
 end
[] print_lexem(ts,P,pid,Ident(n).ls) => n_void
 where void:=(λ)write(pid,get_ident(n,P.Identifiers))
 where n_void:=(λ)print_lexem(ts,P,pid,ls)
 end
[] print_lexem(ts,P,pid,Char(n).ls) => n_void
 where void:=(λ)write(pid,string(n))
 where n_void:=(λ)print_lexem(ts,P,pid,ls)
 end
[] print_lexem(ts,P,pid,Num(n).ls) => n_void
 where m_void:=(λ)write(pid,n)
 where n_void:=(λ)print_lexem(ts,P,pid,ls)
 end
[] print_lexem(ts,P,pid,Blank.ls) => n_void
 where void:=(λ)write(pid," ")
 where n_void:=(λ)print_lexem(ts,P,pid,ls)
 end
[] print_lexem(ts,P,pid,String(n).ls) => n_void
 where void:=(λ)write(pid,"$")
 where n_void:=(λ)print_lexem(ts,P,pid,ls)
 end
end

```

Uses `get_ident` 134, `max_var` 151, `print` 151, `print_Fsym` 151, and `shift_vars` 151.

```
151 <* 151>≡
 // This file is generated automatically: do not edit it directly.
 <the ref_terms speci cation 148a>
```

Defines:

```
max_var, used in chunk 148.
print, used in chunk 148.
print_Fsym, used in chunk 148.
shift_vars, used in chunk 148.
```

### 4.13 ref\_unify.eln

```

148a <the ref_unify specification 148a>≡ (151)
 module ref_unify
 <imports for ref_unify 148b>
 <sorts for ref_unify 141c>
 <operators for ref_unify 148c>
 <strategy operator for ref_unify 142>
 <rules for ref_unify 148f>
 <strategies for ref_unify 144>
 end

148b <imports for ref_unify 148b>≡ (148a)
 import global
 int bool
 REF
 occur[Term,Term]
 replace[Term,eqSystem] replace[Term,Term]
 eq[Term]
 ;
 end

141c <sorts for ref_unify 141c>≡ (148a)
 sort substitution equation eqSystem; end

148c <operators for ref_unify 148c>≡ (148a)
 <global operators for ref_unify 148d>

148d <global operators for ref_unify 148d>≡ (148c)
 operators global
 isvar(@) : (Term) bool;

 nosubstitution: substitution;
 identity : substitution;
 @ -> @ : (Term Term) substitution;
 @ o @ : (substitution substitution) substitution assocLeft;

 apply(@,@) : (substitution eqSystem) eqSystem;
 apply(@,@) : (substitution Term) Term;

 @ : (bool) eqSystem;
 @ : (equation) eqSystem;
 @ & @ : (eqSystem eqSystem) eqSystem assocLeft;
 (@ & @) : (eqSystem eqSystem) eqSystem alias @ & @:;

 system_to_subst(@) : (eqSystem) substitution;

 @ = @ : (Term Term) equation;
 @ : (equation) eqSystem alias @:;
 end

```

```
142 <strategy operator for ref_unify 142>≡ (148a)
 stratop global
 unify : <eqSystem->eqSystem> bs;
 unifys : <eqSystem->eqSystem> bs;
 end
Uses unify 70 151 and unifys 151.
```

```

148f (rules for ref_unify 148f)≡ (148a)
rules for eqSystem
 tt : eqSystem;
 s1,s2 : substitution;
 v : Term;
 t : Term;
global
 [] apply(identity,tt) => tt end
 [] apply(s1 o s2 ,tt) => apply(s2,apply(s1,tt)) end
 [] apply(v->t,tt) => replace(v,t,tt) end
end

rules for Term
 tt : Term;
 s1,s2 : substitution;
 v : Term;
 t : Term;
global
 [] apply(identity,tt) => tt end
 [] apply(s1 o s2 ,tt) => apply(s2,apply(s1,tt)) end
 [] apply(v->t,tt) => replace(v,t,tt) end
end

rules for substitution
P,Q,R : eqSystem;
x,t : Term;
global
 [] system_to_subst(true) => identity end
 [] system_to_subst(x=t) => x->t if isvar(x) end
 [] system_to_subst(x=t) => t->x end
 [] system_to_subst(P & Q) => system_to_subst(P) o
 system_to_subst(Q)
end

rules for bool
xi,xt,yi,yt : int;
global
 [] isvar(VAR(xi,xt)) => true end
 [] isvar(VAR(xi,xt)) => false end
end

rules for eqSystem
P,Q,R : eqSystem;
s,t,x,y : Term;
xi,xt,yi,yt : int;
//n : int;
e : equation;
global
// [new_var_elim] P & var(n) = $t => apply(var(n)->$t,P) end
[unfold] P & (Q & R) => (P & Q) & R end
[eqPass] P & e => e & P end

[delete] P & s=s => P
end
[coalesce] P & VAR(xi,xt)=VAR(yi,yt) =>
VAR(xi,xt)=VAR(yi,yt) &

```



```

 apply(VAR(xi,xt)->VAR(yi,yt),P)
 if xi != yi or xt != yt
end
[conflict] P & s=t => false
 if s != t
 and not(isvar(s)) and not(isvar(t))
end
[occCheck1] P & x=s => false
 if occurs x in s and not isvar(s)
end
[occCheck2] P & s=x => false
 if occurs x in s and not isvar(s)
end
[eliminate1] P & x=s => x=s & apply(x->s,P)
 if not(occurs x in s)
end
[eliminate2] P & s=x => x=s & apply(x->s,P)
 if not(occurs x in s)
end
[matchClash] P & s=x => false
 if not isvar(s)
end
/-- initial and final transitions rules
[trueelim] P & true => P end
[truepass] P & true => true & P end
[trueadd] P => true & P end
[falsepropag] false => false end
[truepropag] true => true end

end

rules for eqSystem
P : eqSystem;
code_f, code_g : int;
s, t : Term;
ss, tt : list[Term];
global
[decompose] P & FSYM(ss,code_f) = FSYM(tt,code_g)
=>
false
if code_f != code_g
end
[decompose] P & FSYM(s.ss,code_f)=FSYM(t.tt,code_f)
=>
P & s = t & FSYM(ss,-1) = FSYM(tt,-1)
end
end
end

```

```

144 <strategies for ref_unify 144>≡
 strategies for eqSystem
 implicit

 [.] unify => dc one(trueadd);
 repeat*(dc one(unfold, delete, decompose,
 conflict, coalesce, occCheck1,
 occCheck2, eliminate1, eliminate2));

```

(148a)

```
 dc one(trueelim, truepropag, falsepropag)
end
[.] unifys => repeat*(dc one(unfold, delete, decompose, conflict,
 coalesce, occCheck1, occCheck2,
 eliminate1, eliminate2)) ;
 dc one(truepass, truepropag)
end
end
```

Uses unify 70 151 and unifys 151.

```
151 < * 151 >≡
 // This file is generated automatically: do not edit it directly.
 <the ref_unify specification 148a>
```

Defines:

unify, used in chunks 67c, 69, 142, and 144.  
unifys, used in chunks 142 and 144.

#### 4.14 ref\_unify\_builtin.eln

- 146a *<the ref\_unify\_builtin specification 146a>*≡ (147)
- ```

module ref_unify_builtin
  <imports for ref_unify_builtin 146b>
  <operators for ref_unify_builtin 146c>
  <rules for ref_unify_builtin 146f>
end

```
- 146b *<imports for ref_unify_builtin 146b>*≡ (146a)
- ```

import global
 REF
 ref_unify
 list[Term] eq[list[Term]]
 builtinSyntacticMatching[Term,list[Term]]
;
end

```
- 146c *<operators for ref\_unify\_builtin 146c>*≡ (146a)
- ```

<global operators for ref_unify_builtin 146d>
<local operators for ref_unify_builtin 146e>

```
- 146d *<global operators for ref_unify_builtin 146d>*≡ (146c)
- ```

operators global
 mgu(@,@) : (Term Term) substitution;
 unifyFail : list[Term];

```
- Uses mgu 147 and unifyFail 147.
- 146e *<local operators for ref\_unify\_builtin 146e>*≡ (146c)
- ```

local
  listTerm_variable(@,@) : ( Term list[Term] ) list[Term];
  listTerm_variable(@,@) : ( list[Term] list[Term] ) list[Term];
  system_to_subst(@,@)  : (list[Term] list[Term]) substitution;
end

```
- 146f *<rules for ref_unify_builtin 146f>*≡ (146a)
- ```

rules for substitution
mgu, listVar, ts: list[Term];
t1, t2 : Term;
m, n : int;
global
 [] mgu(t1,t2) => system_to_subst(listVar,mgu)
 where listVar:=()
 listTerm_variable(t1,listTerm_variable(t2,nil))
 where mgu:=()
 builtinSyntacticUnification(t1,t2,listVar,unifyFail)
 if mgu != unifyFail
end
 [] mgu(t1, t2) => nosubstitution
end

```

```

end

rules for list[Term]
mgu, listVar, ts: list[Term];
t1, t2 : Term;
m, n : int;
global
 [] listTerm_variable(VAR(n,m),listVar) => VAR(n,m).listVar
end
 [] listTerm_variable(FSYM(ts,m),listVar) =>
 listTerm_variable(ts,listVar)
end
 [] listTerm_variable(t1,listVar) => listVar
end
//-----
 [] listTerm_variable(nil,listVar) => listVar
end
 [] listTerm_variable(t1.ts,listVar) =>
 listTerm_variable(ts,listTerm_variable(t1,listVar))
end
end

rules for substitution
lv : list[Term];
lt : list[Term];
t : Term;
v : Term;
global
 [] system_to_subst(lv,nil) => identity end
 [] system_to_subst(v.lv,t.lt) => v->t o system_to_subst(lv,lt) end
end

```

Uses mgu 147 and unifyFail 147.

147 < \* 147 > ≡  
 // This file is generated automatically: do not edit it directly.  
 < the ref\_unify\_builtin specification 146a >

Defines:

mgu, used in chunk 146.  
 unifyFail, used in chunk 146.

## 4.15 ref\_write.eln

- 148a *<the ref\_write specification 148a>*≡ (151)
- ```

module ref_write[verbose]
  <imports for ref_write 148b>
  <operators for ref_write 148c>
  <rules for ref_write 148f>
end

```
- 148b *<imports for ref_write 148b>*≡ (148a)
- ```

import global
 REF list[GrammarRules]
 string io[string] io[int] io[list[GrammarRule]]
 eq[GrammarRules] eq[RuleStatement] eq[StrategyStatement]
 io[RuleStatement] io[list[RuleStatement]]
 io[Tab] prompt[Tab]
 io[GrammarRules] prompt[list[GrammarRules]] io[list[GrammarRules]]
 io[StrategyStatement]
 io[list[StrategyStatement]] prompt[list[StrategyStatement]]
 io[QueryStatement]
 ;
end

```
- 148c *<operators for ref\_write 148c>*≡ (148a)
- ```

<global operators for ref_write 148d>
<local operators for ref_write 148e>

```
- 148d *<global operators for ref_write 148d>*≡ (148c)
- ```

operators global
 write_ref_file(@,@) : (string Program) Program;
 write_file(@,@) : (string Program) Program;

```
- Uses `write_file` 151 and `write_ref_file` 151.
- 148e *<local operators for ref\_write 148e>*≡ (148c)
- ```

local
  write_rules(@,@) : (Pid list[RuleStatement]) list[RuleStatement];
  write_syntab(@,@) : (Pid list[GrammarRules]) list[GrammarRules];
  write_strat(@,@) : (Pid list[StrategyStatement]) list[StrategyStatement];
end

```
- 148f *<rules for ref_write 148f>*≡ (148a)
- ```

rules for Program
 pid, pid1 : Pid;
 ref_file_name : string;
 void0, void1, void2, void3,
 void4, void5, void6, void7,
 void8, void9, void10, void11,
 void12 : string;
 end0, end1, end2, end3,
 end4, end5, end6, end7,

```

```

end8, end9, end10, end11,
end12 : string;
program : Program;

r : RuleStatement;
rs : list[RuleStatement];
strats : list[StrategyStatement];
idtab,
sorttab,
Modules,
rnamestab,
snamestab : Tab;
TopGlobalGrammar, TopGrammar,
GlobalGrammar : list[GrammarRules];
Query : QueryStatement;
global
 [] write_ref_file(ref_file_name,program) =>
 write_file(ref_file_name+".ref",program)
end
 [] write_file(ref_file_name,program) => program
 //--- how to do it ---
 where pid := ()open(ref_file_name,"w")

 where idtab := ()write(pid,program.Identifiers)
 where end0 := ()write(pid,"\nend\n\n")
IF verbose == 1 : {where void0 := ()write(stdout,"--- Identifiers is written\n")}

 where sorttab := ()write(pid,program.Sorts)
 where end1 := ()write(pid,"\nend\n\n")
IF verbose == 1 : {where void1 := ()write(stdout,"--- Sorts is written\n")}

 where Modules := ()write(pid,program.Modules)
 where end2 := ()write(pid,"\nend\n\n")
IF verbose == 1 : {where void2 := ()write(stdout,"--- Modules is written\n")}

 where rnamestab := ()write(pid,program.RuleNames)
 where end5 := ()write(pid,"\nend\n\n")
IF verbose == 1 : {where void5 := ()write(stdout,"--- RuleNames tab is written\n")}

 where snamestab := ()write(pid,program.StrategyNames)
 where end6 := ()write(pid,"\nend\n\n")
IF verbose == 1 : {where void6 := ()write(stdout,"--- StrategyNames tab is written\n")}

 where TopGlobalGrammar := ()write_syntab(pid,program.TopGlobalGrammar)
IF verbose == 1 : {where void4 := ()write(stdout,"--- TopGlobalGrammar is written\n")}

 where TopGrammar := ()write_syntab(pid,program.TopGrammar)
IF verbose == 1 : {where void10 := ()write(stdout,"--- TopGrammar is written\n")}

 where GlobalGrammar := ()write_syntab(pid,program.GlobalGrammar)
IF verbose == 1 : {where void11 := ()write(stdout,"--- GlobalGrammar is written\n")}

 where rs := ()write_rules(pid,program.AllRules)
IF verbose == 1 : {where void7 := ()write(stdout,"--- AllRules is written\n")}

 where strats := ()write_strat(pid,program.AllStrategies)
IF verbose == 1 : {where void8 := ()write(stdout,"--- AllStrategies is written\n")}

```

```

 where Query := ()write(pid,program.Query)
 where end9 := ()write(pid,"\nend\n\n")
IF verbose == 1 : {where void9 := ()write(stdout,"--- query is written\n")}

 where pid1 := ()close(pid)
IF verbose == 1 : {where void12 := ()write(stdout,"--- file is closed\n")}
end
end

rules for list[RuleStatement]
pid : Pid;
void : string;
r, r1 : RuleStatement;
rs, rs1 : list[RuleStatement];
global
 [] write_rules(pid,nil) => nil
 where void := ()write(pid,"\nEndDef end\n\n")
 end
 [] write_rules(pid,r.rs) => nil
 where r1 := ()write(pid,r)
 where void := ()write(pid,"\nend\n\n")
 where rs1 := ()write_rules(pid,rs)
 end
end

rules for list[GrammarRules]
pid : Pid;
void, void1, void3, void5 : string;
void2,void4 : int;
fs, fs1 : GrammarRules;
fss, fss1 : list[GrammarRules];
profile1,profile:list[GrammarRule];
a1, a2 : int;
global
 [] write_syntab(pid,nil) => nil
 where void := ()write(pid,"\nEndDef end\n\n")
 end
 [] write_syntab(pid,GrammarForSort a1:a2:profile.fss) => nil
 where void1:=()write(pid,"GrammarForSort ")
 where void2:=()write(pid,a1)
 where void3:=()write(pid,":")
 where void4:=()write(pid,a2)
 where void5:=()write(pid,":\n")
 where profile1:=()write(pid,profile)
 where void := ()write(pid,"\nend\n\n")
 where fss1 := ()write_syntab(pid,fss)
 end
end

rules for list[StrategyStatement]
pid : Pid;
void : string;
fs, fs1 : StrategyStatement;
fss, fss1 : list[StrategyStatement];
global

```

```

[] write_strat(pid,nil) => nil
 where void := ()write(pid,"\nEndDef end\n\n")
end
[] write_strat(pid,fs.fss) => nil
 where fs1 := ()write(pid,fs)
 where void := ()write(pid,"\nend\n\n")
 where fss1 := ()write_strat(pid,fss)
end
end
```

Uses `write_file` 151 and `write_ref_file` 151.

```
151 ⟨ * 151 ⟩ ≡
 // This file is generated automatically: do not edit it directly.
 ⟨ the ref_write speci cation 148a ⟩
```

Defines:

`write_file`, used in chunks 108a, 148, and 152e.  
`write_ref_file`, used in chunk 148.



## 4.16 refstring2string.eln

152a *(the refstring2string specification 152a)*≡ (153)

```

module refstring2string
 <imports for refstring2string 152b>
 <operators for refstring2string 152c>
 <rules for refstring2string 152e>
end

```

Uses refstring2string 153.

152b *(imports for refstring2string 152b)*≡ (152a)

```

import global builtinInt list[string] list[Term]
 REF
 ref_write[0] ref_read[0]; // 0 for verbose
local divers2string//[Term]
 io[Program] io[string]
 ;
end

```

152c *(operators for refstring2string 152c)*≡ (152a)  
*(global operators for refstring2string 152d)*

152d *(global operators for refstring2string 152d)*≡ (152c)

```

operators global
 spec2ref(@,@) : // exports a spec, returns a Program
 (string string) Program;
 spec2ref(@,@) : // exports a spec, returns its ref-file name
 (string string) string code 162; // ".spc"+".lgi"->".ref"
 //-----
 ThisProgram Program : // returns SPEC(curr_lgi,curr_spc)
 code 165; // ref of the running program
 //-----
 refstring2string(@,@) : // ("FSYM(...)",P)
 (string Program) string;
 refstring2string(@,@) : // ("FSYM(...)", "file.ref")
 (string string) string code 160;
 //-----
 string2refstring(@:'@,@) : // ("FSYM(...)": "list",P)
 (string Sort Program) string;
 string2refstring(@:'@,@) : // ("FSYM(...)": "list", "file.ref")
 (string string string) string code 161;
end

```

Uses refstring2string 153, spec2ref 153, string2refstring 153, and ThisProgram 153.

152e *(rules for refstring2string 152e)*≡ (152a)

```

rules for Program
 lgi, spc, tmp_file_name : string;
global
 [] spec2ref(lgi,spc) => read_file(tmp_file_name)
 where tmp_file_name:=()spec2ref(lgi,spc) end
end

```

```

rules for string
P, P1 : Program;
aux_file_name : string;
termstring : string;
sorte : Sort;
lgi, spc : string;
global
[] refstring2string(termstring,SPEC(lgi,spc)) =>
 refstring2string(termstring,spec2ref(lgi,spc))
end
[] refstring2string(termstring,P) =>
 refstring2string(termstring,aux_file_name)
 where aux_file_name:=()get_aux_file_name+".ref"
 where P1:=()write_file(aux_file_name,P)
end
//-----
[] string2refstring(termstring:sorte,SPEC(lgi,spc)) =>
 string2refstring(termstring:Sort2string(sorte),
 spec2ref(lgi,spc))
end
[] string2refstring(termstring:sorte,P) =>
 string2refstring(termstring:Sort2string(sorte),aux_file_name)
 where aux_file_name:=()get_aux_file_name+".ref"
 where P1:=()write_file(aux_file_name,P)
end
end

```

Uses read\_file 123, refstring2string 153, spec2ref 153, string2refstring 153, and write\_file 151.

153 < \* 153 >≡  
// This file is generated automatically: do not edit it directly.  
< the refstring2string specification 152a >

Defines:

refstring2string, used in chunks 106b, 115, 152, and 154f.  
spec2ref, used in chunk 152.  
string2refstring, used in chunks 115f, 152, and 154f.  
ThisProgram, used in chunks 115e and 152d.

## 4.17 string2refterm.eln

154a *<the string2refterm specification 154a>*≡ (155)

```

module string2refterm
 <imports for string2refterm 154b>
 <sorts for string2refterm 154c>
 <operators for string2refterm 154d>
 <rules for string2refterm 154f>
end

```

Uses `string2refterm` 155.

154b *<imports for string2refterm 154b>*≡ (154a)

```

import global string REF
 divers2string//[X]
 ref2string[Term];
local
 io[string];
end

```

154c *<sorts for string2refterm 154c>*≡ (154a)

```

sort X; end

```

154d *<operators for string2refterm 154d>*≡ (154a)

*<global operators for string2refterm 154e>*

154e *<global operators for string2refterm 154e>*≡ (154d)

```

operators global
 string2refterm('@':'@,@) : (string string Program) Term;
 refterm2string(@,@) : (Term Program) string;
end

```

Uses `refterm2string` 155 and `string2refterm` 155.

154f *<rules for string2refterm 154f>*≡ (154a)

```

rules for Term
s, sorte : string;
P : Program;
global
 [] string2refterm(s:sorte,P) =>
 string2term(string2refstring(s:sorte,P):"Term","REF.ref")
end
end

rules for string
x : Term;
P : Program;
void : string;
void1 : string;
void2 : string;
global

```

```

[] refterm2string(x,P) =>
 refstring2string(term2string(x,"REF.ref"),P)
// where void1:=()writeln(stdout,"term2string=")
// where void:=()writeln(stdout,term2string(x,"REF.ref"))
// where void2:=()writeln(stdout,"-----")
 end
end
```

Uses refstring2string 153, refterm2string 155, string2refstring 153, string2refterm 155, string2term 134,  
and term2string 134.

```
155 <* 155>≡
 // This file is generated automatically: do not edit it directly.
 <the string2refterm speci cation 154a>
```

Defines:

**refterm2string**, used in chunks 109a and 154.  
**string2refterm**, used in chunks 106b, 108a, 109a, and 154.

## 5 Index