ELAN

USER MANUAL

January 27, 2000
ELAN V3.4

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

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

LORIA: CNRS, INPL, INRIA, UHP, Université de Nancy 2
Campus scientifique
615, rue du Jardin Botanique
BP101
54602 Villers-lès-Nancy CEDEX
FRANCE
E-mail: elan@loria.fr

Copyright ©1996 – 2000 Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène 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.

elan n. 1. Enthusiastic vigor and liveness. 2. Style; flair. [Fr < OFr. eslan, rush < eslancer, to throw out: es-, out (< Lat. ex-) + lancer, to throw (< LLat. lanceare, to throw a lance < Lat. lancea, lance).]

The American Heritage Dictionary

1. ÉLAN. n.m. (vers 1420 selon Bloch ; de élancer.
   || 1 Mouvement pour s’élancer. Calculer son élan (Cf. Calculer, cit. 7). …
   || 2 Par anal. Mouvement par lequel la voix reprend, s’élance. …
   || 3 Fig. Mouvement ardent, subit qu’un vif sentiment inspire. V. Mouvement; ardent, impulsion, poussée. Brusque élan. Élan impétueux. Élans de jeunesse, de passion. …

2. ÉLAN. n.m. (XVIIe s.; hellent au XVIIe s., ellend au XVIe; haut allem. elend, du baltique elnis). Mammière artiodactyle (Cervidés), des pays du Nord. Élan du Canada. V.

Orignal.

LE ROBERT, extraits.
Contents

1 A short introduction to ELAN ................................. 7
  1.1 What could you do in ELAN? .................................. 7
  1.2 A very simple example ....................................... 8
  1.3 A more generic example ..................................... 9
  1.4 An extended example ....................................... 11

2 How to use ELAN .................................................. 15
  2.1 How to get, install and use ELAN .......................... 15
  2.2 Using ELAN .................................................. 16
    2.2.1 How to run the ELAN interpreter ....................... 16
    2.2.2 Top level interpreter ................................ 21
    2.2.3 How to run the ELAN compiler ......................... 26
    2.2.4 A simple interface to the ELAN system ............... 28
  2.3 Keep in touch ............................................. 29

3 ELAN: the language ............................................. 31
  3.1 Lexicographical conventions .............................. 31
    3.1.1 Separators ............................................. 31
    3.1.2 Lexical units ......................................... 31
    3.1.3 Comments ............................................. 32
  3.2 Element modules ........................................... 32
  3.3 Definition of signatures .................................. 33
    3.3.1 Sort declarations ................................... 33
    3.3.2 Operator declarations ................................ 34
    3.3.3 Function declaration options ........................ 36
    3.3.4 Strategy declaration ................................ 37
  3.4 Definition of rules ....................................... 38
    3.4.1 Rule syntax ........................................... 38
    3.4.2 The where construction ................................ 40
    3.4.3 Choose-Try ............................................ 41
    3.4.4 Conditions ............................................ 42
    3.4.5 Labels visibility ..................................... 42
  3.5 Definition of strategies .................................. 42
    3.5.1 Elementary strategies syntax ........................ 43
    3.5.2 Defined strategies ................................... 49
  3.6 Evaluation mechanism ..................................... 50
    3.6.1 Rewrite rules on terms ............................... 50
    3.6.2 Rewrite rules on strategies .......................... 54
  3.7 Modules ................................................... 57

January 27, 2000

ELAN user manual
3.7.1 Visibility rules ................................................. 57
3.7.2 Built-in modules ............................................. 57
3.7.3 Parameterised modules ...................................... 58
3.7.4 LGI modules .................................................. 58
3.8 Pre-processing ................................................. 60
  3.8.1 Simple duplication ........................................ 61
  3.8.2 Duplication with argument .............................. 61
  3.8.3 Enumeration using FOR EACH .......................... 61
3.9 Differences with previous version of the language ........ 62

4 ELAN: the system ............................... 65
  4.1 Global architecture ...................................... 65
  4.2 The preprocessor ......................................... 65
  4.3 The parser ................................................. 65
  4.4 The interpreter ........................................... 66
  4.5 The compiler .............................................. 66
    4.5.1 Associative and Commutative symbols ............. 67
    4.5.2 Non-linear rules ................................... 68
    4.5.3 Built-ins .......................................... 68
    4.5.4 Input/Output facilities ............................. 69
  4.6 The REF format ........................................... 69

5 The standard library and the built-ins ......... 73
  5.1 The ELAN standard library .............................. 73
    5.1.1 common ............................................. 73
    5.1.2 ref ................................................ 74
    5.1.3 noquote .......................................... 74
    5.1.4 strategy ........................................... 74
  5.2 The built-ins ........................................... 74
    5.2.1 Boolean ........................................... 75
    5.2.2 Numbers ........................................... 75
    5.2.3 Identifiers ......................................... 77
    5.2.4 Elementary term computations ...................... 78
    5.2.5 Input/Output ....................................... 79
    5.2.6 Strategies .......................................... 81
    5.2.7 REF Format ......................................... 84
  5.3 Calling external programs ......................... 85

6 Contributed works ............................... 89
  6.1 Description of the ELAN contributions ............. 89
  6.2 ELAN's annotated bibliography ....................... 90
Foreword

This manual presents the version V3.4 of the ELAN language and of its environment.

This is mainly an update of version V3.3 with bug corrections, improvements in the code generated by the compiler and with a new user interface facilitating the use of the system.

This is the user manual of ELAN version V3.4
We will be pleased to know any inaccuracy, error or typos.
Your comments on any part of the language, the system or this manual are welcome.
elan@loria.fr
Chapter 1

A short introduction to ELAN

This chapter presents in a top-down approach the ELAN main features. If you are beginning to use ELAN, you should certainly have a look at this chapter first. Complementarily Chapter 3 presents a bottom-up complete description of the language.

For a gently introduction to ELAN we recommend to read [BKK+98b] or to access to http://www.loria.fr/ELAN/.

1.1 What could you do in ELAN?

Relying on the premiss that inferences rules can be quite conveniently described by rewrite rules, we started in the early nineties to design and implement a language in which inference systems can be represented in a natural way, and executed reasonably efficiently.

This leads us quickly to formalise such a language using the rewriting logic introduced by J. Meseguer [Mes92] and to see ELAN as a logical framework where the frame logic is rewriting logic extended with the fundamental notion of strategies [Vit94, KKV95]. A rewrite theory and an associated strategy is called a computational system.

In ELAN, a logic can be expressed by specifying its syntax, its inference rules and a description of how these rules should be applied. The syntax of the logic can be described using mixfix operators as in the OBJ language [GKK+87] or SDF [HHK89]. The inference rules of the logic are described by conditional rewrite rules with where assignments allowing to introduce variables local to a rule.

From this description of the logic and a theory written in this logic, we infer automatically a computational system consisting of a rewriting theory plus strategies.

Since we wanted ELAN to be modular, with a syntactic level well-adapted to the user’s needs, the language is designed in such a way that three programming levels are possible.

- First the design of a logic is done by the so-called super-user, with the help of a powerful preprocessor that expands specific macros into ELAN constructions.
- Such a logic can be used by the (standard) user in order to write a specification.
- Finally, the end-user can evaluate queries in the specification, following the semantics described by the logic.

This corresponds to the general diagram given in Figure 1.1. The query is interactively given by the end-user at the ELAN prompt level.
1.2 A very simple example

Let us fully detail a complete example. The next module illustrates what the programmer has to write in order to describe the derivative operation on simple polynomials. The first part contains:

- the module name: poly1
- modules you want to import: int (the integer library module)
- sort declaration: variable and poly
- operators definitions: constructors and functions declarations

The second part contains the computation definition. Given a query, ELAN repeatedly normalises the term using unlabelled rules. This is done in order to perform functional evaluation and thus it is recommended for the user to provide a confluent and terminating unlabelled rewrite system to ensure termination and unicity of the result. This normalisation process is built in the evaluation mechanism and consists in a leftmost innermost normalisation. This yields always a single result.

You can control the ELAN construction of terms by giving parsing annotations like associativity or priority to operators.

```plaintext
module poly1
import global int;
end
sort variable poly;
end
operators global
X : variable;
@ : (variable) poly;
@ : (int) poly;
@ + @ : (poly poly) poly assocRight pri 1;
```
1.3 A more generic example

The top level of the logic description given in the following module describes the way to run
the system. The logic declaration just introduces the sorts of interest (query and result) and
defines .eln modules to be imported: this is the “top level”.

LPL poly1 description
query of sort poly
result of sort poly
import poly1
start with ()query
end

Once defined, we can use the two previous modules by calling the interpreter as detailed in
section 2.2.1.

% elan poly1.lgi
enter query term finished by the key word ’end’: deriv(X) end

□ start with term: deriv(X)
□ result term: 1
deriv(3*X*X + 2*X + 7) end

□ start with term: deriv(3*X*X+2*X+7)
□ result term: 0*X*X+3*1*X+X+1+0*X+2*1+0

1.3 A more generic example

The next example, introduces more ELAN features. It consists of the specification of elementary
polynomials built over a finite set of variables, and the derivative functions to compute the
derivated form with respect to a given variable. Tasks are divided as follows:

1. the super-user describes in a generic way the derivative and factorisation inferences, i.e. a
   logic for polynomial differentiation,

2. the user gives the specification of an algebra in which (s)he wants to derivate polynomials;
   in this case, this is quite simple since it amounts to specify the variables of the considered
   polynomial algebra,

3. the end-user gives a differentiation problem.
The diagram in Figure 1.1 thus instantiates as described in Figure 1.2.

The description of the logic and of the specification is done in the ELAN syntax described in the Chapter 3 and it can be further extended by the super-user. These descriptions are assumed to be done in files with specific extensions:

1. **.lgi** for the top level logic description, this file is written by the ELAN super-user,

2. **.eln** for a module used in a logic description, this file is also written by the ELAN super-user,

3. **.spc** for a specification, i.e. a program written in the defined logic by an ELAN user.

The **.eln** module of our example is the following:

```elang
module poly2[Vars] 1
import global in Vars eq[variable] identifier list[identifier]; 2
end 3
sort variable poly; 4
end 5
operators global 6
FOR EACH Id:identifier SUCH THAT Id := (listExtract) elem(Vars) : 7
{ Id : variable; } 8
@ : (variable) poly; 9
@ : (int) poly; 10
@ + @ : (poly poly) poly assocRight p r i 1 (AC); 11
@ * @ : (poly poly) poly assocRight p r i 2 (AC); 12
deriv(@, @) : (poly variable) poly; 13
end 14
rules for poly 15
p, p1, p2 : poly; 16
x, y : variable; 17
n : int; 18
global 19
[] 0 + p => p 20
t] 0 * p => 0 21
end 22
end 23
```

January 27, 2000

ELAN user manual
1.4 An extended example

Then, the logic declaration describes the way to parse the specification file that contains a finite list of variables.

**LPL**

poly2 description
specification description
part Vars of sort list[identifier]
  import identifier list[identifier]
end

query of sort poly
result of sort poly
import poly2[Vars]
start with ([] query
end

To instantiate the Logic and build the Computational System, you have to give a specification. An example of such specification is:

```
specification someVariables
  Vars X.Y.Z.nil
end
```

It contains a list of variables that is read and transformed into a rewrite rule: Vars => X.Y.Z.nil

The **FOR EACH** preprocessor construction used in the poly2.elin module performs a textual replacement that extracts identifiers \( X, Y, Z \) from the list \( X.Y.Z.nil \) and declare them of sort variable.

You can notice that operators \( \ast \) and \( + \) are now declared as associative and commutative, which is helpful to implement some simplification rules. One rule contains a conditional part.

Now let us call ELAN with the specification someVariables.spc:

```
% elan poly2 someVariables.spc
enter query term finished by the key word 'end':
  deriv(3\ast X\ast X + 2\ast X + 7 , X) end

[] start with term: deriv(3\ast X\ast X+2\ast X+1,X)
[] result term: X\ast X+3+2
```

The result has been simplified in a better way, but it is still difficult to read due to the lack of parenthesis.

1.4 An extended example

This last example shows how to play with labelled rules and strategies.

Unlabelled rules are applied repeatedly, at any position, to normalize the term. A labelled rule is applied only when a strategy calls it. In this case, the rule is applied at top position on the term. In the next module, you can recognize labelled rules: names are given between brackets.
You can define strategies in the same way you define rules and use them in local affectation parts introduced by the key word \texttt{where}. When a rule is applied, before constructing the right-hand-side, conditional parts are evaluated and variables involved in local affectation parts are instantiated by the result of the application of a strategy on a term.

\begin{verbatim}
module poly3(Vars) 1
import global int Vars eq[variable] identifier list[identifier]; 2
end 3
sort variable poly; 4
end 5
operators global 6
FOR EACH Id:identifier SUCH THAT Id:=extract(Vars) : 7
{ Id : variable; }
\@ : ( variable ) poly; 8
\@ : ( int ) poly; 9
\@ + \@ : ( poly poly ) poly assocRight pri 1 (AC); 10
(\@ + \@) : ( poly poly ) poly alias \@ + \@; 11
\@ * \@ : ( poly poly ) poly assocRight pri 2 (AC); 12
(\@ * \@) : ( poly poly ) poly alias \@ * \@; 13
deriv(\@,\@) : ( poly variable ) poly; 14
end 15
stratop\_global 16
last\_simplify : <poly> bs; 17
simplify : <poly> bs; 18
end 19
rules for poly 20
P, p1, p2, p3, p4, p5 : poly; 21
e1, e2, e3, e4 : poly; 22
x, y : variable; 23
n, n1, n2, n3 : int; 24
global 25
[1] P+0 => P end 26
[1] P\textasciitilde 0 => 0 end 27
[1] P\textasciitilde 1 => P end 28
[1] n\textasciitilde n2 => n where n:=() n\textasciitilde n2 end 29
[1] deriv(x,x) => 1 end 30
[1] deriv(y,x) => 0 if x ! = y end 31
[1] deriv(n,x) => 0 end 32
[1] deriv(p1+p2,x) => p3 where p3:=simplify deriv(p1,x)+deriv(p2,x) end 33
end 34
[1] deriv(p1*p2,x) => p5 where p1:=simplify p1*deriv(p2,x) where p3:=simplify deriv(p1,x)*p2 where p5:=simplify p3+p1 end 35
end 36
local 37
[factorize] P + p1+p1 => P + 2*p1 end 38
[factorize] P + n1*p1 + n2*p1 => P + n3*p1 where n3:=()n1+n2 end 39
[factorize] P + (p1*p2) + (p1*p3) => P + p1 where p1:=simplify p1*p2+p3 end 40
[expand] P + (p1+p2)*p3 => P + e1+e2 where e2:=simplify p2*p3 where e1:=simplify p1*p3 end 42
end 43
strategies for poly 44
implicit 45
[1] last\_simplify => \texttt{repeat}(dc one(expand)) : \texttt{repeat}(dc one(factorize)) 46
end 47
\end{verbatim}
end
[] simplify => repeat*(dc one(expand)) ; repeat*(dc one(factorize))
end
end

And now we can use it:

% elan poly3 someVariables.spc
enter query term finished by the key word 'end':
deriv(3*X*X + 2*X + 7 , X) end

☐ start with term: deriv(3*X*X+2*X+1,X)
☐ result term: ((X*6)+2)
☐ end
Chapter 2

How to use ELAN

2.1 How to get, install and use ELAN

The latest version of the ELAN system includes the language interpreter, the compiler, a simple Java interface, the standard library and a lot of applications and examples. It runs on the following architectures and systems: Compact-Alpha (ex DEC-ALPHA) under Unix OSF1, SUN4 under Solaris and Intel-PC under Linux, and it can be obtained from the ftp server ftp.loria.fr from the directory /pub/oria/protheo/softwares/Elan/elan.3.4.tar.gz. This package contains the binary executable files of ELAN, the class files of the Java compiler and of the graphical user interface, the library made of more than 90 basic ELAN modules, several basic ELAN examples, the manual of ELAN in different forms, and an html presentation of the ELAN system. This downloaded distribution file, after un-zipping and un-tarring, has the following structure (cf. FILES file):

This downloaded ELAN distribution file, after un-zipping and un-tarring, has the following structure:

- bin/ - Executable ELAN files and ELAN tools (scripts)
- alpha/ - executable image for DEC-ALPHA (alpha)
- sun4u/ - executable image for ULTRA-SUN4 (sun4u)
- i686/ - executable image for PC-Linux (i686)

- elanlib/ - a library of ELAN modules
  - Compiler/
    - alpha/ i686/ sun4u/ - the architecture dependent part of the runtime library
    - classes/ - the class files of the Java compiler
  - common/ - common files
    - (standard data types: int, bool, pairs, lists, etc.)
  - noquote/ - the non-quoted part of the library
    - (to use ELAN specification files)
  - ref/ - the REF term library
  - strategy/ - the library of defined strategies

- interface/ - the class files of the Java graphical user interface
- html/ - a very short introduction to ELAN
- Elan.gif - the ELAN logo

- manual/ - manual.[dvi|pdf|ps] and library.[dvi|pdf|ps]

- examples/ - Selection of standard ELAN programs
  - (can be used to test the current ELAN installation)
Another tar file corresponding to elan-contrib/ contains some ELAN contributions
(mainly for Constraint Programming and Automated Deduction):

Programming/ - Programming languages
Proving/ - Proving tools
Solving/ - Constraint solving tools

If the distribution file has been un compressed into a directory PWD, the installation of ELAN can be completed by:

- including the path PWD/bin/`uname -m` into your environment variable PATH, and
- setting-up your environment variable ELANLIB to PWD/elanlib.

If any problems occur during your installation, we recommend to read the README file at the top of the directory containing the ELAN distribution. The completely installed version of the system ELAN takes for one architecture about 12 MB of disk-space, and any additional architecture takes about 2MB.

2.2 Using ELAN

The language can be used either through an interpreter or a compiler as described below in section 2.2.1 and 2.2.3 respectively. In order to ease the call of the various options of the interpreter or of the compiler, we provide a simple user interface described in the section 2.2.4.

2.2.1 How to run the ELAN interpreter

You can now execute ELAN by just typing the elan command with the following usage:

```
******** ELAN version 3.4 (DD/MM/YY.HH:MM) *******

(c) LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1999

elan usage : elan [options] lgi_file [spc_file]

options : 
--dump, -d : dump of signature, strategies and rules
--trace, -t [num] : tracing of execution (default max)
--statistic, -s/-S : statistics: short/long
--warningsoff, -w : suppress warning messages of the parser
--quiet, -q : quiet regime of execution
--batch, -b : batch regime (no messages at all)
--elanlib : elanlib
--secondlib, -l lib : second elan library (by default ..)
--command, -C : command language
--export, --export : export to a .ref file
--import : import from a .ref file
```

Both the logic description file (`lgi_file`) and the specification file (`spc_file`) may be written without their suffixes. Default suffixes for ELAN files are:

.lgi – for the top level logic description file,
.eln – for modules used in the top level logic description file,
.spc – for the specification file,
The system ELAN searches these files (i.e. .eln, .lgi, .spc, .ref) in the following directories with the decreasing priorities:

- in the current directory,
- in the directory described by the value of a system variable SECONDELANLIB,
- in the directories described as ELANLIB/commons, ELANLIB/noquote, ELANLIB/strategy and ELANLIB/ref, in this order.

The value of the variable ELANLIB can be locally replaced by the switch --elanlib. The default value of the variable SECONDELANLIB is set to the parent directory, however, it can be also locally replaced by the switch --secondlib (or, -l) (for details, see below).

In Chapter 1, we have shown, how to simply run the ELAN interpreter with a logic description. We now illustrate on several examples the usage of the different switches.

-d, (or, --dump) dumps out the signature of the described logic (i.e. all sorts and function symbols with their profiles), definitions of all strategies and rewrite rules. For the example poly3 from Chapter 1:

> elan -d poly3 someVariables

we obtain a list of rewrite rules of the form:

```
local rule expand:poly/poly3[Vars] [1/3/fsym=233/vars=9]
  ( VAR(0)+(VAR(1)+VAR(2))*VAR(3)+VAR(4))
  =>
    ( VAR(0)+(VAR(5)+VAR(6)+VAR(7)+VAR(8)))
where VAR(8) := (simplify:poly/poly3[Vars]) (VAR(2)*VAR(4))
where VAR(7) := (simplify:poly/poly3[Vars]) (VAR(2)*VAR(3))
where VAR(6) := (simplify:poly/poly3[Vars]) (VAR(1)*VAR(4))
where VAR(5) := (simplify:poly/poly3[Vars]) (VAR(1)*VAR(3))

local rule expand:poly/poly3[Vars] [2/4/fsym=233/vars=6]
  ( VAR(0)+(VAR(1)+VAR(2))*VAR(3))
  =>
    ( VAR(0)+(VAR(4)+VAR(5))
where VAR(5) := (simplify:poly/poly3[Vars]) (VAR(2)*VAR(3))
where VAR(4) := (simplify:poly/poly3[Vars]) (VAR(1)*VAR(3))
```

Names of rewrite rules (e.g. expand) are displayed together with their sorts (e.g. poly), over which these rules work, and with a module name, where they have been defined (e.g. poly3[Vars]). An internal code of the top-most function symbol of its left-hand side and the number of variables of each rule are also displayed as an additional information. Variables appearing in rewrite rules are displayed in a uniform way: VAR(i), where i is an internal index.

The print-out continues with a list of strategies in the form:

```
strategy global simplify:poly/poly3[Vars]
  repeat* (  
    dc one(expand:poly )
  );
  repeat* ( 
    dc one(factorize:poly )
  )
```

January 27, 2000
The label of the strategy is composed of the name of the strategy, its sort and the name of the module where it has been defined (as for named rules).

The specification of the signature of the described logic is composed of a set of defined (and internal) sorts:

Sorts

\[ \text{intern int, ident, identifier, string, list[identifier],} \]
\[ \text{intern ident, int, variable, intern string, bool,} \]
\[ \text{poly,} \]

and of a list of function symbols with their profiles in the following form:

Function symbols

\[ \text{ Function symbol } \]
\[ \text{ profile } \]
\[ \text{ description } \]
\[ (\text{variable})\text{poly} \]
\[ (\text{int})\text{poly} \]
\[ (\text{poly poly})\text{poly} \]

assocRight
\[ \text{ pri } 0 \text{ code 231; } \]
\[ \text{ pri } 0 \text{ code 232; } \]
\[ \text{ pri } 1 \text{ code 233; } \]
\[ \text{ pri } 2 \text{ code 234; } \]
\[ \text{ pri } 0 \text{ code 235; } \]

\[ \text{ etc } \]

\[ \text{ -t (or, --trace) allows tracing ELAN programs. The maximal depth of tracing can be specified, like -t 9, however, by default (i.e. \text{-t }), it traces all levels. The number specified as the trace level counts matchings of a left-hand side, verifications of the conditional part of a rule, and evaluations of strategies applied over terms in local affectations. Thus, if the user wants to trace his/her rewrite derivation up to the 5-th level, the trace argument should be specified as -t 15. } \]

\[ > \text{ elan -t poly3 someVariables} \]
\[ \text{ etc enter } \]
\[ \text{ query term finished by the key word 'end':} \]
\[ \text{ deriv(X*X,X) end} \]

\[ \square \text{ start with term:} \]
\[ \text{ deriv(X*X,X) } \]

\[ \text{ applying strategy 'simplify:poly/poly3[Vars]' on X } \]
\[ \text{ trying 'expand:poly' on X } \]
\[ \text{ failing 'expand:poly' trying 'factorize:poly' on X } \]
\[ \text{ failing 'factorize:poly' trying 'factorize:poly' on X } \]
\[ \text{ failing 'factorize:poly' setting VAR(4) on X } \]

\[ \text{ [reduce] start:} \]
\[ \text{ [0] deriv(X*X,X) } \]

\[ \text{ [reduce] start:} \]
\[ \text{ [0] \text{ deriv(X*X,X) } \]
\[ \text{ [0] (X*deriv(X,X)) } \]
\[ \text{ [1] (X*1) } \]
\[ \text{ [2] X } \]

\[ \text{ [reduce] stop:} \]

\[ \text{ January 27, 2000} \]

ELAN user manual
[2] X
[reduce] stop :
  applying strategy "simplify:poly/poly3[Vars]" on X
  trying "expand:poly" on X
  fail of "expand:poly"
  trying "expand:poly" on X
  fail of "expand:poly"
  trying "factorize:poly" on X
  fail of "factorize:poly"
  trying "factorize:poly" on X
  fail of "factorize:poly"
  trying "factorize:poly" on X
  fail of "factorize:poly"
  setting VAR(5) on X
[reduce] start:
  [0] (X\times X)
[reduce] stop :
  applying strategy "simplify:poly/poly3[Vars]" on (X\times X)
  trying "expand:poly" on (X\times X)
  fail of "expand:poly"
  trying "expand:poly" on (X\times X)
  fail of "expand:poly"
  trying "factorize:poly" on (X\times X)
  fail of "factorize:poly"
  trying "factorize:poly" on (X\times X)
  fail of "factorize:poly"
  trying "factorize:poly" on (X\times X)
  fail of "factorize:poly"
  setting VAR(3) on (X\times X)

\[1\] (X\times X)
[reduce] stop :
  trying "expand:poly" on (X\times X)
  fail of "expand:poly"
  trying "expand:poly" on (X\times X)
  fail of "expand:poly"
  trying "factorize:poly" on (X\times X)
  fail of "factorize:poly"
  trying "factorize:poly" on (X\times X)
  fail of "factorize:poly"
  trying "factorize:poly" on (X\times X)
  fail of "factorize:poly"

\[\square\] result term:
  (X\times X)
\[\square\] end

\texttt{-s (-S, \texttt{or} --statistics)} displays a brief (-s) or a complete (-S) version of statistics. These statistics contain the information about the running time of the last query, average speed in rewrites per second, the statistics of named and unnamed rules, which were tried and applied.

\texttt{\$ elan -S poly3 someVariables
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . etc
 enter query term finished by the key word 'end':
 deriv(X\times X, X) end

\[\square\] start with term :
  deriv((X\times X), X)
result term: (X+X)

end

Statistics:
total time (0.007+0.000)=0.007 sec (main+subprocesses)
average speed 714 inf/sec
5 nonamed rules applied, 23 tried
0 named rules applied, 20 tried

named rules
applied tried rule for symbol
0 8 expand:poly
0 12 factorize:poly

nonamed rules
applied tried rule for symbol
0 4 (poly + poly )
2 12 (poly * poly )
3 7 deriv(poly, variable )

end of statistics

The complete statistics show a detailed list of applications and tries for named and nonamed rules. The brief statistics do not precise the last paragraph about unnamed rules.

-w (or, --warningoff) eliminates all ambiguity warnings produced during parsing of the ELAN description of a logic.

-q (or, --quiet) suppresses all error messages and warnings during the execution of the ELAN program.

-b (or, --batch) suppresses all messages, thus this mode is useful when the ELAN system is executed from a script file in the batch mode.

--elanlib dir locally redefines the value of the system variable ELANLIB by the string dir.

--secondlib dir (or, --secondlib dir) does the same with the variable SECONDelanlib.

--export fname, --export fname exports the input specification into the REF format stored in the file fname. The switch cexport is used only when the produced REF file is compiled by the ELAN compiler.

--import fname imports a specification in the REF format from the REF file fname produced, in general, by the switch --export.

> elan --export poly3.ref poly3 someVariables.spc

******** ELAN version 3.4 ********
(c) LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1999
Import form file poly3

Importing Identifiers
Importing Sorts
Importing Modules
Importing RuleNames
Importing StrategyNames
Import rule extractrule1:identifier/list[identifier]!L0 form module list[identifier]
Import rule extractrule2:identifier/list[identifier]!L0 form module list[identifier]
Import rule expand:poly/poly3[Vars]!L0 form module poly3[Vars]
Import rule expand:poly/poly3[Vars]!L0 form module poly3[Vars]
Import rule factorize:poly/poly3[Vars]!L0 form module poly3[Vars]
Import rule factorize:poly/poly3[Vars]!L0 form module poly3[Vars]
Import rule factorize:poly/poly3[Vars]!L0 form module poly3[Vars]
Import strategy listExtract:identifier/list[identifier]
Import strategy simplify:poly/poly3[Vars]
Import strategy last_simplify:poly/poly3[Vars]

enter query term finished by the key word 'end':

-C runs the ELAN interpreter with a simple command language interpreted on the top-most level. This mode is described in Section 2.2.2.

After loading all modules specified in the logic description file, ELAN requires to enter a query term, which has to be finished by the word 'end' or by character ^D (control-D). Its evaluation can be interrupted by typing ^C, the interpreter then proposes a simple run-time menu:

- ExecutionAbort - A - aborts the current execution and allows to enter another query term,
- Continue - C - continues the interrupted execution,
- Dump - D - dumps the signature, rules and strategies of the currently loaded logic,
- Exit - E - quits the ELAN system,
- Statistics - S, s - writes the actual statistics corresponding to the last evaluated query,
- ChangeTrace - T - changes trace level of the current derivation such that the interpreter asks for a new trace level,
- ChangeQuiet - Q - switches between two displaying modes: quiet – unquiet,

2.2.2 Top level interpreter

The command language of the ELAN interpreter improves the user’s interface by offering to the user several commands, which allow to:

- change several parameters of the system,
- debug programs using break-points,
- keep the history of queries and their results,
- runs the current logic with different strategies, different entries, etc.
When the ELAN interpreter is started with the option -C, instead of asking:

enter query term finished by the key word 'end'

ELAN prompts the user:

enter command finished by ':;'

We show a brief description of all commands of the top level command language (also found in the file ELANLIB/help.txt), which will be later illustrated on several examples.

`quit` terminates the session and leaves the ELAN interpreter.

`help` shows the help file ELANLIB/help.txt.

`load mod` loads (i.e. imports) an additional module, where the identifier `mod` describes its name (possibly with its arguments).

`batch mod` calls a script file with the name `mod` (an identifier), which contains a stream of commands of the command language. It tries to evaluate all commands of the script file, and it returns the control to the interactive mode. The script files can be also concatenated.

`qs` shows the queue of input queries (i.e. the history up to the last ten queries). The terms in this queue of queries `qs` can be referenced by the identifiers `Q`, `Q1`, ..., `Q9` respecting their sorts. These sorted identifiers could be used in constructions of further queries.

`rs` shows the queue of the latest results (i.e. the history up to the last ten results). Terms in this queue of results `rs` are referenced by identifiers `R`, `R1`, ..., `R9`. In the case of non-deterministic computations, there is no correspondence between query terms `Qi` and result terms `Ri`.

`startwith (str)term` redefines the 'start with' pattern originally defined in the logic description file .lg. The pattern `term` may contain also symbols `query` standing as a place-holder of an input term entered by the user using the command `run`, or symbols `Q`, ..., `Qi` or `R`, ..., `Ri` standing for values of previously put queries or results.

`checkwith term` redefines the 'check with' pattern originally defined in the logic description file. The boolean pattern `term` may contain any symbol amongst `query`, `Q`, ..., `Qi` or `R`, ..., `Ri`.

`printwith term` sets-up the 'print-with' pattern. The 'print-with' pattern may contain the symbol `result`, which is a place-holder for result terms. Intuitively, the meaning of the 'print-with' term `P` is that any result `r` of the computation is substituted for the place-holder `result` in the pattern `P`. The substituted term `P[result/r]` is normalized and printed out. This command can be used for the automatic transformation of results into different formats, e.g. a transformation of very large result terms into more readable or incomplete notation, or a conversion into a LaTeX notation. The default value of the 'print-with' pattern is `result`, i.e. the identity.

`run term` evaluates the term `term` as a query w.r.t. rewrite rules of the current logic and eventually returns results. The entry term `term` may contain identifiers `Q`, `Q1`, ..., `Q9`, (resp. `R`, `R1`, ..., `R9`) referring to values of preceding queries or results, e.g. items of the queue of queries `qs` (resp. results `rs`).

`sorts type type type` sets-up sorts of the symbols `query`, `result` and of the 'print-with' pattern.

`stat` prints the statistics,
dump prints rules, strategies and the signature,

dump name dumps only named rules with the name name, strategies named by name or unnamed rules with the head symbol name.

dump n dumps only unnamed rules of a function symbol with the internal code n.

display n if n is not zero, all printed terms are in the internal form. The internal form of a term attaches to a symbol name its internal code, which allows to identify overloaded function symbols. display 0 switches all print-outs into traditional (thus, overloaded) format.

trace n sets-up the trace level to the value n.

break name (resp. unbreak name) sets (resp. removes) break-points to all named rules with the name name, to all strategies named by name and to all unnamed rewrite rules with the head symbol name.

break n (resp. unbreak n) sets (resp. removes) break-points to unnamed rules with the head symbol with the internal code n.

breaks shows a list of all breakpoints.

We illustrate the command language on a small ELAN session with the example poly3 from the previous Chapter. When the switch -C is specified, the ELAN interpreter loads the module Query describing the syntax of commands.

> elan -q -C poly3 someVariables

******** ELAN version 3.4 ********

(c) LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1999

handling module identifier
  handling module ident
    handling module bool
      end of bool
    end of ident
  handling module eq[identifier]
    handling module cmp[identifier]
      end of cmp[identifier]
    end of eq[identifier]
  end of identifier
handling module list[identifier]
  handling module int
    handling module builtinInt
      end of builtinInt
    end of int
  end of list[identifier]
handling module poly3[Vars]
  handling module eq[variable]
    handling module cmp[variable]
      end of cmp[variable]
    end of eq[variable]
  end of poly3[Vars]
handling module Query[poly,poly,poly]
  end of Query[poly,poly,poly]

We can run simple queries using the command run:
enter command finished by ';':
run deriv(X*X,X);
  □ start with term:
      deriv((X*X),X)
  □ result term:
      (X*X)
  □ end

enter command finished by ';':
run deriv(X*X+X*X+X*+1,X);
  □ start with term:
      deriv(((X*(X*X))+(X*X)+(X+1)),X)
  □ result term:
      (X*(X+2)+X))
  □ end

The queries and their results are stored in two queues, which can be listed by commands qs, rs:

enter command finished by ';':
qs;
  Queries ... 2 : elements
  Q1  poly  deriv((X*X),X)
  Q  poly  deriv(((X*(X*X))+(X*X)+(X+1)),X)

enter command finished by ';':
rs;
  Results ... 2 : elements
  R1  poly  (X*X)
  R  poly  (X*(X+2))

The previous queries and their results can be referenced in the construction of the new ones:

enter command finished by ';':
run deriv(R,X);  
  □ start with term:
      deriv((X6+2),X)
  □ result term:
      (X6+2)
  □ end

enter command finished by ';':
run deriv(Q,Q,X);
  □ start with term:
      deriv(deriv((1+(X*(((X*2)+X))))),X)*deriv((1+(X*(((X*2)+X))))),X)
  □ result term:
      (24+(X*72))
  □ end

Function symbols defined by unnamed rules can be debugged, for example, if we know their internal code. In the following example, the command dump displays the symbol with the internal code 216, and then, we can put a break-point on this function.

dump 235;
function dump
  'deriv' '(', @ ',', @ ')': (poly variable)poly pri 0 code 235;

enter command finished by ';':
break 235;
function dump
  'deriv' '(', @ ',', @ ')': (poly variable)poly pri 0 code 235;

Having break-points set, the execution shows all entry and exit points of traced functions, or strategies:
2.2 Using ELAN

```plaintext
enter command finished by ';':
run deriv(X*X,X);

☐ start with term:
  deriv((X*X),X)
[0] deriv((X*X),X)
    [0] deriv(X,X)
    [1] 1
    [0] deriv(X,X)
    [1] 1
[1] (X*X)

☐ result term:
  (X*X)

☐ end

We can switch all outputs into the internal format by the command display 1:

enter command finished by ';':
display 1;

enter command finished by ';':
run deriv(X*X,X);

☐ start with term:
  deriv_235((X_228*X_228),X_228)
[0] deriv_235((X_228*X_228),X_228)
    [0] deriv_235(X_228,X_228)
    [1] 1
    [0] deriv_235(X_228,X_228)
    [1] 1
[1] (X_228*X_228)

☐ result term:
  (X_228*X_228)

☐ end

We can change the 'start-with' pattern to (last_simplify)deriv(query,X), and the command stat informs us about the current setting and the performance of the last executed query.

enter command finished by ';':
startwith (last_simplify)deriv(query,X);

enter command finished by ';':
stat;
Input type  : poly
Output type : poly
Print type  : poly
Strategy    : last_simplify
Start_with  : deriv_235( VAR(0),X_228)
Check_with  : true_1
Print_with  : VAR(0)

---------
Statistics:
total time    (0.013+0.010)=0.023 sec (main+subprocesses)

average speed 384 inf/sec
  5 nonamed rules applied,  23 tried
  0 named rules applied,  20 tried

named rules
  applied   tried   rule for symbol
    0       8         expand:poly
    0      12        factorize:poly

nonamed rules
```

January 27, 2000
2.2.3 How to run the ELAN compiler

The ELAN compiler transforms a logic description into an executable binary file. The compiler (for detailed description, see [Vit96, MK98]) produces an efficient C code, which is later compiled by the cc or the gnu gcc compiler. The executable binary code is dependent on the particular architecture, because a small part of the ELAN library performing the basic non-deterministic operations is written in assembly language [Mor98]. Up to now, this is the only reason that makes the compiler available only on the architectures DEC-ALPHA, SUN4 and Intel-PC. This release of the ELAN compiler does not compile the whole ELAN language: the main restriction is that the ELAN compiler does not treat input/output and reflective capabilities of the ELAN language. There is also a restriction on the class of rules containing AC-symbols that can be compiled. Thus, any logic description containing AC-symbols should be carefully studied before being compiled (see Section 4.5 for more details).

To run the ELAN compiler, a Java Virtual Machine (JAVA 2 SDK, aka jdk 1.2, or newer), a C compiler (gcc recommended) and a C-shell (tcsf recommended) have to be installed on your system. To complete the installation, the script PWD/bin/elanc has to be modified: CLASSPATH and JAVA variables have to be set according to your system. Then, you can now execute the ELAN compiler by just typing the elanc command with the following usage:

```
  elanc [[elam options][compiler options]] file[].lgi] [file.[spc]]
```

Options:
- `output <name>`
- `verbose`
- `nocode`
- `nosplit`
- `quiet`
- `debug`
- `noOptimiseChoicePoint [default]`
- `optimiseChoicePoint`

By default, the compiler produces a lot of C files and a lgiName.makefile file, which are later compiled.

**The switch** `-output nqueens` produces the executable file with the name nqueens.

**The switch** `-nosplit` generates only two files: lgiName.c and lgiName.core.c. This switch is useful to compile small specifications.

**The switch** `-nocode` directly builds an executable file and deletes all generated C files.

**The switch** `-quiet` suppresses all printed messages during compilation phase.

**The switch** `-debug` activates the debugging mode: more messages are printed during compilation and the generated code becomes less efficient.

**The switch** `-noOptimiseChoicePoint [default]` avoids the deterministic analysis phase.

**The switch** `-optimiseChoicePoint` reduces the number of set choice-points: this switch reduces the runtime needed memory and improve the efficiency. It is recommended to not activate this switch if your specification contains AC symbols.
We illustrate the compiler on a simple example of a distributed file. The first example shows the compilation of the program nqueens with the query queens(8) (given in the .1gi file):

```
> elanc -output queens -nosplit -nocode -optimiseChoicePoint nqueens
elan --export nqueens.ref ./nqueens.1gi

******* ELAN version 3.4 *******

(c) LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1999
handling module nqueens
handling module int
handling module builtinInt
end of bool
end of builtinInt
end of int
end of nqueens

Export file nqueens.ref has been created

java -classpath classes.zip:elanlib/Compiler/classes REM nqueens.ref
-output queens -nosplit -nocode -optimiseChoicePoint
Reduce ELAN Code Machine. Reading from file nqueens.ref . .
Parsing............
Compiling nonamed rules..............
Compiling strategies..
Building queens...

> queens
result = .(([](4),[](2),[](7),[](3),[](6),[](8),[](5),[](1),nil)),(nil)))
result = .(([](5),[](2),[](4),[](7),[](3),[](6),[](6),[](1),nil)))
result = .(([](3),[](2),[](8),[](6),[](4),[](7),[](1),nil)))
...
after 91 solutions:
result = .(([](5),[](7),[](2),[](6),[](3),[](1),nil)))

rewrite_step = 227356

It is also possible to get the whole trace of the execution. In order to produce this information the executable code has to be compiled in the debugging mode and the switch -trace has to be used:

> elanc -debug -nosplit nqueens
...
Reduce ELAN Code Machine. Reading from file nqueens2.ref . .
Parsing
Compiling nonamed rules..............
Compiling strategies..
execute the following line to build your program
  gmake -f nqueens.make
> gmake -f nqueens.make
gcc -pipe -g3 -DDEBUG -DPDEBUG -DBITSET32 ...
...

> a.out -trace
| start with: fun_211_queens(_,[](8))
| rewrite[1] queens(_,[](8),nil)
| start with: str_176(queens(_,[](8),nil))
| start with: fun_211_greater_int(_,[](8),nil)

January 27, 2000
A more detailed description of the compiler can be found in Section 4.5.

Another way to call the ELAN compiler (also known as Reduced ELAN Machine) consists in running Java as follows:

```
java -classpath $[ELANLIB]/Compiler/classes REM <REF file name> <options>
```

The `<REF file name>` must be created by the option `--export <REF file name>` of the ELAN interpreter simply used here to parse the `<LGI file name>`.

### 2.2.4 A simple interface to the ELAN system

The current distribution includes now a first version of a Graphical User Interface that allows us to call, in the same framework, both the interpreter and the compiler. This interface is written in Java with the Swing classes, and you need the JAVA 2 SDK (aka jdk 1.2, or newer) to use it. This means that the java binary should be in your PATH, and the command java `--version` should return a number greater than 1.2. Note that the same remark holds also for the compiler, also written in Java. To use the interface, you only have to call the script `elan` located in the directory `bin/` of the distribution. This (simple) script assumes the existence of the variable `ELANLIB` already needed by both the interpreter and the compiler. You must also configure some variables defined in `elan`, like `GMAKE` (to compile), `XTERM` (to open a new window), `XEMACS` (to edit files), which contain the locations of the related well-known tools. The content of these variables must be changed according to your system. After this installation step, you should be able to execute the interface successfully, and the new ELAN logo should appear in a window including several buttons:

**Files** To choose the `.lgi` file (resp. `.spc`) you want to call ELAN with. It is also possible to edit `.lgi`, `.spc` and `.eln` files with an emacs-like editor.

**Compiler → Execute** To open a new window, where you can compile with the button `Compile` and then `Execute` the generated program. In the latter, a new window is created, where you can modify the strategy you want to use and the sort of the term you want to normalize (see also Section 4.5.4). By default, the rewrite process is started with the `start with` instruction located in the `.lgi` (and `.ref`) file used as input of the compilation phase.

**Interpreter → Execute** To open a new window, where you can either `Interpret` with `Execute` with no option, or `Interpret` with the option `--trace` thanks to the button `Trace`.
Help  To see an ELAN presentation.
Quit  To really quit.

2.3  Keep in touch

In case you have any problems, questions or comments about ELAN, or just only to be keeping in touch with the ELAN team, please use the following e-mail address:

elan@loria.fr or post on the elan-users mailing-list:

elan-users@loria.fr

More informations on ELAN can be found on the web page:

http://www.loria.fr/ELAN/.

ELAN is issued from the PROTHEO research team, information about this research project can be found at the following address:

Chapter 3

ELAN: the language

This chapter presents a full bottom-up description of the language features that, contrary to Chapter 1, uses a construction of the language only when it has been introduced before.

The ELAN syntax is given in the Extended Backus-Naur Form (EBNF). We assume that the reader is familiar with this well-known notation, where for instance \{ X \} + (resp. \{ X \} *) denotes a non-empty (resp. a possibly empty) repetition of X, and \[ X \] means that X is optional, and may be omitted.

All examples given below are running under the current implementation of the ELAN version V3.4.

3.1 Lexicographical conventions

3.1.1 Separators

All ASCII characters whose code are less or equal than 32 are considered by the parser as separators. Thus spaces, tabulations and newlines are separators.

3.1.2 Lexical unities

A lexical unity in ELAN is either an identifier, a natural number or a special character.

Identifiers are composed by concatenation of letters, numerals and the character “_”, and they should begin with a letter.

\[ <letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z \]
\[ \ | a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z \]

\[ <numeral> ::= 0|1|2|3|4|5|6|7|8|9 \]

\[ <identifier> ::= <letter> \{ <letter> | <numeral> | _ \} * \]

\[ <quoted identifier> ::= ’\{ <letter> | <qchar> | <numeral> | _ \} + ’ \]

\[ <elan identifier> ::= <identifier> except ELAN's keywords \]

All characters different from the previously introduced ones (letters, numerals, separators) and the characters ‘{’, ‘}’, ‘,’ are considered as special characters and are referenced as \<char>.

\[ <nchar> ::= <char> except ‘Ø’ and ‘.’ \]
A number is the concatenation of numerals.

<number> ::= { <numeral> } +

3.1.3 Comments

All strings enclosed between the strings "/*" and "*/" or between the string "//" and the end of line, are considered by the parser as comments.

3.2 Element modules

Element modules are the basic unit blocks of the language. They allow to define computational systems i.e. to define a rewrite theory together with a strategy. For modularity reasons, importations are made possible in ELAN modules, under the condition that no cycle exists in the importation relation. This is described in the modularity section 3.7.

The syntax is the following:

<module> ::= module <formal module name>
   [ <imports> ]
   [ <sort definition> ]
   [ <operator definition> ]
   [ <stratop definition> ]
   { <family of rules> } *
   { <family of strategies> } *
end

The module of name ModuleName should be declared in the file ModuleName.eln. Only one, and exactly one module can be described in a given file.

Importations can be made local or global:

<imports> ::= import { <instantiated module name> } + ; end
   | import [ <global imports> ] [ <local imports> ] end

<global imports> ::= global { <instantiated module name> } + ;
<local imports> ::= local { <instantiated module name> } + ;

An instantiated module name like List[List[Int]] is described by:

<instantiated module name> ::= <identifier>
   | <identifier> [ <instantiated module name>
   { , <instantiated module name> } * ]

Sorts are always global:

<sort definition> ::= sort { <sort name> } + ; end
but operators can be exported, and thus declared global, or just local in which case they can be used only in the module where they are defined.

\[
<\text{operator definition}> ::= \text{operators} \\
\hspace{1em} [ \text{global operator definition}> ] [ \text{local operator definition}> ] \\
\text{end}
\]

\[
<\text{global operator definition}> ::= \text{global} \{ <\text{operator declaration}> \}^+ ;
\]

\[
<\text{local operator definition}> ::= \text{local} \{ <\text{operator declaration}> \}^+ ;
\]

And similarly for strategy operator definition:

\[
<\text{stratop definition}> ::= \text{stratop} \\
\hspace{1em} [ \text{global stratop definition}> ] [ \text{local stratop definition}> ] \\
\text{end}
\]

\[
<\text{global stratop definition}> ::= \text{global} \{ <\text{strategy declaration}> \}^+ ;
\]

\[
<\text{local stratop definition}> ::= \text{local} \{ <\text{strategy declaration}> \}^+ ;
\]

\[
<\text{family of rules}> \text{ is defined in Section 3.4.} <\text{family of strategies}> \text{ is defined in Section 3.5.}
\]

### 3.3 Definition of signatures

ELAN is a multi-sorted language, and signatures are composed by the definitions of the sorts and operators symbols. The operator syntax is given in a mix-fix form which allows the super-user to define very conveniently its own syntax.

#### 3.3.1 Sort declarations

A sort declaration is either a sort name (an identifier) or a sort name followed by the list of sort names it depends on:

\[
<\text{sort name}> ::= <\text{identifier}> \\
| <\text{identifier}> [ <\text{sort name}> \{ \text{, } <\text{sort name}> \}^* ]
\]

**Example 3.1** bool, int, list[bool], pair[int,list[bool]], list[X] are sort names in ELAN.

**Remark:** A sort name is always global in ELAN. Two sort declarations using the same sort name, will result in the creation of only one sort.

**Built-in sorts** For efficiency, sorts could be defined as built-ins. See section 5.2 for details.
3.3.2 Operator declarations

A function symbol, also called operator, is given either by defining a new symbol together with its rank or by aliasing an existing operator. Optional properties can be specified:

\[
<\text{operator declaration}> ::= \langle\text{new operator declaration} \rangle \\
| \langle\text{operator alias} \rangle
\]

\[
<\text{new operator declaration}> ::= <\text{operator name}> : <\text{rank}> [ \{ <\text{operator option}> \} ^+ ]
\]

\[
<\text{operator alias}> ::= \langle\text{new operator declaration} \rangle \text{ alias } <\text{old operator name}>
\]

In case of aliasing, \( <\text{new operator declaration} \rangle \) is only another name for the function symbol \( <\text{old name}> \). The two names are then accessible and refer to the same object.

▷ The last alias defined is considered by the parser as having the highest priority with respect to the previously defined ones.

\[
<\text{old operator name} > ::= <\text{operator name}>
\]

\[
<\text{operator name} > ::= <\text{name}>
\]

The name of a function symbol contains information that will be used by the mixfix parser to read a term:

\[
<\text{name} > ::= \{ <\text{symbol} > \} ^+
\]

\[
<\text{symbol} > ::= <\text{single lexem}>
| @
| <\text{quoted single lexem}>
| ' \langle\text{number} \rangle ' 
\]

\[
<\text{single lexem} > ::= <\text{elan identifier}>
| <\text{number}>
| <\text{uchar}>
\]

\[
<\text{quoted single lexem} > ::= <\text{quoted identifier}>
\]

where, as defined on page 31, \(<\text{elan identifier} >\) is any identifier except keywords. Notice that indeed keywords can be used without their special meaning when placed between quotes.

The characters ‘\{’, ‘\}’, ‘\’’ are restricted to the pre-processor use. Thus they cannot be used in any module, but they can be used in the specification file or in the description of the query, since both are not pre-processed. The ‘\@’ character indicates the end of the name and the character ‘\’\’ is the place-holder for an operator ‘\@’gment. These two last characters can be used with another semantics than their built-in one, when placed between quotes.

A quoted number after the character ‘\’\’ represents the definition of a character by its ascii code. This facility can be used in order to add to the syntactic form of a term non-visible characters or the characters ‘\{’, ‘\}’, ‘\’’.

Finally an operator declaration is achieved by defining its rank, which is just a list of sort names with optional description of selectors.
3.3 Definition of signatures

\[ \text{<rank> ::= <sort name>} \]
\[ \quad | \quad \{ \text{<sort name>} \} + \text{<sort name>} \]
\[ \quad | \quad \{ \text{<identifier> : <sort name>} \} + \text{<sort name>} \]

Example 3.2 With the two declarations:

\[ @ + @ : (\text{int int}) \text{int} \]
\[ +(@, @) : (\text{int int}) \text{int} \quad \text{alias @+@:} \]

the strings “x + y” and “+(x, y)” represent the same term.

When defining a constructor operator, it is possible to define the related selectors and modifiers, as outlined in the next example.

Example 3.3 In the following module example, the selector first and second are specified and the system will automatically generate the corresponding selector and modifier operators and the corresponding rewrite rules.

```plaintext
module simplePairSelectors
import bool;
end

sort X Y Pair;
end
operators
global
[@.@] : (first: X second: Y) Pair;
isPair(@) : (Pair) bool;
end
end
```

This is similar to the following module:

```plaintext
module simplePair
import bool;
end

sort X Y Pair;
end
operators
global
[@.@] : (X Y) Pair;
isPair(@) : (Pair) bool;
\begin{itemize}
  \item automatically generated selectors
  \item @.first : (Pair) X;
  \item @.second : (Pair) Y;
\end{itemize}
\begin{itemize}
  \item automatically generated modifiers
  \item @.[first<->@] : (Pair X) Pair;
  \item @.[second<->@] : (Pair Y) Pair;
\end{itemize}
end
end
```

where the following rewrite rules are in fact automatically generated:

\[ (x, y).\text{first} \Rightarrow x \]
\[ (x, y).\text{second} \Rightarrow y \]
\[ (x, y)[.\text{first<->z}] \Rightarrow (z, y) \]
\[ (x, y)[.\text{second<->w}] \Rightarrow (x, w) \]
3.3.3 Function declaration options

As in languages like OBJ or ASF+SDF, an operator declaration carries many informations about the syntax and the semantics of the operator. We also need to specify complementary information like the precedence of the operator for the parser, or some of its semantic properties like associativity. This is done in ELAN using the following syntax:

\[
\text{<operator option> ::= assocLeft}
\mid \text{assocRight}
\mid \text{pri <number>}
\mid (\text{AC})
\mid \text{code <number>}
\mid \text{builtin <number>}
\]

Parsing Associative operators The first two options are useful for defining syntactic associativity of symbols:

Example 3.4 The declarations:
\[
\begin{align*}
0 \ast 0 & \quad : \text{(int int) int} \quad \text{assocLeft} \\
*(0, 0) & \quad : \text{(int int) int} \quad \text{alias @*@:}
\end{align*}
\]
allow to parse the term \(x \ast y \ast z\) as \(*((x, y), z)\).

Priorities The priority parsing problems can be solved using the priority option:

Example 3.5 The declarations:
\[
\begin{align*}
0 + 0 & \quad : \text{(int int) int} \quad \text{assocLeft pri 10} \\
0 \ast 0 & \quad : \text{(int int) int} \quad \text{assocLeft pri 20}
\end{align*}
\]
allow to parse an expression like \(x + y \ast z\) as usual in \(x + (y \ast z)\).

Coercions can be defined and even hidden as in the following example:

Example 3.6 Assume that equations are constructed with the equality symbol:
\[
0 = 0 \quad : \text{(term term) equation;}
\]
and equational systems as conjunctions of equational systems:
\[
0 \& 0 \quad : \text{(eqSystem eqSystem) eqSystem;}
\]
An equation should be considered as an equational system, which corresponds to consider the sort equation as a subsort of eqSystem. This is done by the declaration of an invisible operator:
\[
0 \quad : \text{(equation) eqSystem;}
\]
This (invisible) coercion allows parsing the expression \((P \& e)\) where \(P :eqSystem; e:equation\).

Semantic properties Semantic properties can be used. Currently only associativity and commutativity (abbreviated AC) is allowed and only for binary operators. The user should be aware that in interpreted mode, AC matching can be quite inefficient. It is strongly recommended to use the compiled mode in presence of AC operators.

Example 3.7 One can define a union operator which is associative and commutative in the following way:
\[
0 \cup 0 \quad : \text{(set set) set} \quad (\text{AC})
\]
User built-in operators  In order to improve efficiency of the evaluation, built-in operators can be specified using the code and builtin options. The natural number specified after the code and builtin keywords specifies the index of this symbol in the symbol table of ELAN. A builtin option is for built-in operators without attached code defining its semantics, while the code option is for built-in operators with an attached procedure.

Example 3.8 The symbol true is defined in the bool module as the first symbol of the symbol table:

    true : bool builtin 1

Section 5.2 summarizes the codes already defined for ELAN's standard built-ins.
▷ Adding built-in operators is of course a delicate matter since the ELAN source code has to modified.

3.3.4 Strategy declaration

A strategy operator is similar to a function symbol as defined previously except that at least one of the sorts involved in the strategy rank is a strategy sort.

(strategy declaration) ::= (new strategy declaration)
    | (strategy alias)

(new strategy declaration) ::= (strategy name) : (strategy rank) [ (strategy option) ]

(strategy alias) ::= (new strategy declaration) alias (old strategy name)

In case of aliasing, (new strategy declaration) is only another name for the strategy operator (old strategy name). The two names are then accessible and refer to the same object.
▷ The last alias defined is considered by the parser as having the highest priority with respect to the previously defined ones.

(old strategy name) ::= (strategy name)

(strategy name) ::= (name)

(strategy rank) ::= (strategy sort name)
    | ( (strategy args) ) (sort name)
    | ( { (strategy arg) } + ) (strategy sort name)

(strategy arg) ::= (sort name)
    | (strategy sort name)

(strategy args) ::= (sort name) (strategy args)
    | (strategy sort name) { (strategy arg) } *

January 27, 2000
<strategy sort name> ::= < <strategy sort name> -> <strategy sort name> >
   | < <sort name> -> <sort name> >
   | < <sort name> -> <strategy sort name> >
   | < <strategy sort name> -> <strategy sort name> >

<strategy option> ::= <elementary strategy option>
   | <defined strategy option>

<elementary strategy option> ::= bs

<defined strategy option> ::= assocLeft
   | assocRight
   | pri <number>
   | ( AC )

The bs attribute (for basic strategy) indicates that the definition of the strategy is given by a rewrite rule whose right-hand side only uses the constructions of the elementary strategies language defined in Section 3.5.1.

3.4 Definition of rules

Rewriting is the basic concept underlying the design of ELAN. The language allows defining labelled conditional rules with local assignments. It is important to realize the difference between:

- labelled rules whose evaluation is fully controlled by the user defined strategies and,

- non-labelled rules which are intended to perform functional evaluation. They are applied using a built-in left-most inner-most strategy.

A condition controlling the firing of a rule is a term of sort bool introduced by the if keyword. A local assignment is introduced by the keyword where. Its main purpose is to assign to local variables the result of the application of a strategy on a term.

3.4.1 Rule syntax

Rules are introduced in families, possibly restricted to a single element.

<family of rules> ::= rules for <sort name>
   [ { <variable declare> } ]
   [ global { <labelled or non-labelled rule> } ]
   [ local { <labelled rule> } ]
end

<labelled or non-labelled rule> ::= <labelled rule>
   | <non-labelled rule>

<labelled rule> ::= [ <rule label> ] <rule body> end

<rule label> ::= <identifier> [ ( variable { , <variable> } * ) ]

<non-labelled rule> ::= [ ] <rule body> end
3.4 Definition of rules

<rule body> ::= <term> == <term> [ { <if-where-choose> } * ]

<if-where-choose> ::= if <boolean term>
| where <variable name> ::= ( [ <strategy term> ] ) <term>
| where <variable name> ::= [ <strategy term> ] <term>
| where { <sort name> } <term> ::= ( [ <strategy term> ] ) <term>
| where { <sort name> } <term> ::= [ <strategy term> ] <term>
| choose
( { try <if-where-choose> } ) +
end

<variable declare> ::= [ { <variable name> }, ] * <variable name> ;
<variable name> ::= <elan identifier>

<strategy term> defined in Section 3.5.1 is a constant strategy name declared with the attribute bs. Note that it is optional.

<strategy term> defined in Section 3.5.2 is a term built on the signature of strategy operators.

Remark: The syntax for terms is not defined here since it is user defined and induced from the rank of the operators. Because of the very liberal way of defining operator ranks, the term syntax in mixfix in the tradition of algebraic specification languages like ASF+SDF or OBJ. The term parser is based on Earley's context free parser [Ear70] where the grammar is extracted from the operator rank definitions.

— Release Note: Since V.3.0, the choose-try construction, the patterns in where facility and the user defined strategies call by “[ ]” have been introduced.

The rule labels accept now arguments that should be variables used in the rule.

In the where construction with a term, the sort is just used to declare the sort of the term used as pattern. This is useful for the parser which then just checks that this is also the sort of the result on the right hand side of “==”. Note that the sort declarations is useless for the where construction using only variables, since the sort of the variable is always attached to it.

There are two ways for calling strategies, according to the strategy language used by the programmer:
▷ A strategy call of the form “( )” can only be done using a strategy that is declared as bs.

▷ A strategy call of the form “[ ]” can only be done using a strategy that is not declared as bs.

Example 3.9 Here is one way for defining the idempotency rules for booleans. Notice that the two rules are local to the module so that their label can only be used for defining strategies in this module. Note also that the two rules have, on purpose, the same label:

```
rules for boolean
  z : boolean;
local
[identity] z and z == z end
end

rules for boolean
  z : boolean;
local
[identity] z or z == z end
end
```

January 27, 2000
Since several rules may have the same label, the resulting ambiguities are handled by the strategy evaluation as explained later. The rule label is optional and rules may have no name. This last case is designed in such a way that the intended semantics is functional, and it is currently the responsibility of the rewrite rule designer to check that the rewrite system composed of all unlabelled rules is confluent and terminating. Unlabelled rules are handled by a built-in strategy (left-most inner-most) as fully described in Section 3.6.

3.4.2 The where construction

The where construction has several purposes. The first one is to modularize the computations by computing auxiliary results affecting them to local variables, thus providing computation sharing. The second is to direct the evaluation mechanism using strategies and the built-in left-most inner-most evaluation reduction. Let us review the possibilities offered by ELAN.

The where statement is first useful when one wants to call a strategy on subterms.

Example 3.10 As a first introducing example, let us consider the module:

```elang
module ruleWithWhere

sort foo; end

operators global
  a : foo;
  b : foo;
end

stratop global
  strat : <foo -> foo> bs;
end

rules for foo
  z : foo ;

  global
  [r0] a => b end
  [r1] a => z where z := \b end
  [r2] a => z where z := (strat)\b end
end

// ....
end
```

Using the rule r0, the term a reduces in one step to b. Using the rule r1, the term b is first normalized into b' using non-labelled rules and then a is replaced by b'. Using the rule r2, the term b is first normalized into b' using non-labelled rules and then the strategy strat, defined elsewhere by the super-user, is applied on b' and the result is substituted to a.

One can also perform more complex affectations of the local variables by using patterns.

The current syntax of the where statement with patterns can be rephrased as follows:

```elang
where (sort)p ::= ()term | (s)term | [s]term
```

where the pattern p should be of sort sort. AC-operators as well as non-linearity are allowed for patterns.

This pattern in where capability is exemplified in the next subsection.

January 27, 2000

ELAN user manual
3.4.3 Choose-Try

The construction Choose try allows factorizing common parts of rules with the same left-hand side. For instance the two rules

\[
[l] \ l \Rightarrow r \ \text{where} \ y_1 := (\text{u}_1 \\
\text{if} \ c_2'(l, y_1) \\
\text{where} \ y_3 := (\text{u}_3)
\]

\[
[l] \ l \Rightarrow r \ \text{where} \ y_1 := (\text{u}_1 \\
\text{if} \ c_2''(l, y_1) \\
\text{where} \ y_3 := (\text{u}_3)
\]

are factorized into one rule:

\[
[l] \ l \Rightarrow r \ \text{where} \ y_1 := (\text{u}_1 \\
\text{choose} \\
\text{try if} \ c_2'(l, y_1) \\
\text{try if} \ c_2''(l, y_1) \\
\text{end} \\
\text{where} \ y_3 := (\text{u}_3)
\]

In this factorized form, the term u_1 is normalized only once and the assignment to y_1 is performed also only once.

**Example 3.11** This example shows the use of patterns as well as the Choose try construction.

```elan
module quick
import global int list[int]; end
sort pair; end
operators global
sort(∅) : (list[int]) list[int];
π: (int list[int] list[int]) pair;
[∅, ∅] : (list list) pair;
end
rules for list[int]
z : int;
zs, sl : list[int];
global
[[]] sort(∅) => ∅ end
[[]] sort(z, zs) => append(sort(z), z, sort(l)) where (pair l) = π p: z, zs, nil, nil end
end
rules for pair
p, z : int;
z, sl, l1, l2 : list[int];
global
[[]] π p, nil, nil, l => [p, l] end
[[]] π p, z, zs, l1, l2 => π p, z, zs, l1, l2

\[
\text{choose try} \ \text{where} \ l1 := (\text{z}) \\
\text{where} \ l2 := (\text{z}, l) \\
\text{if} \ p < z \\
\text{try where} \ l1 := (\text{z}, s) \\
\text{where} l2 := (\text{z}) \\
\text{if not} (p < z) \\
\end
end
end
end
```

January 27, 2000
3.4.4 Conditions

The boolean term \( c \) given in a condition is reduced using the strategy that results from normalization using the non labelled rules and by the user defined strategies. If the computation stops and the resulting term is the boolean \texttt{true} then the condition is considered to be satisfied and the rule can be applied. If the computation stops and the resulting term is not \texttt{true} then the condition is considered as false.

3.4.5 Labels visibility

The \texttt{global} and \texttt{local} attributes make sense only for labelled rules. When a labelled rule is:

- \texttt{local}, then its label can only be used in the module in which the rule is declared,
- \texttt{global}, then its label can be used in all the modules importing the module defining the rule, with the visibility described in the section on modularity (see section 3.7).

▷ According to their semantics, non-labelled rules are always global.

— Release Note: Since version V.3.0, the notion of single rule has been removed and the notion of rule locality has been introduced, so that rule labels can now be declared to be local to a module or, on the contrary, globally accessible. See section 3.9 for details concerning these changes.

Remark: We will see next on page 61 how rule families can be constructed automatically in ELAN using the pre-processor construction “FOR EACH”.

3.5 Definition of strategies

The notion of strategy is one of the main originalities of ELAN. In practice, a strategy is a way to describe which rewrite computations the user is explicitly interested in. It specifies when a given rule should be applied in the term to be reduced. From a theoretical point of view, a strategy can be viewed either as a function or as a subset of all proof terms defined by the current rewrite theory. The application of a strategy to a term results in the (possibly empty) collection of all terms that can be derived from the starting term using this strategy [KKV95, Vit94]. When a strategy returns an empty set of terms, we say that it \textit{fails}.

▷ A strategy enumerates all the terms it describes, should this collection be finite or not. Consequently the user should note that in case (s)he writes a strategy that enumerates an infinity of terms, then the evaluation process will of course not terminate.

Two strategy languages are offered in ELAN. The language of \textit{elementary strategies} is mainly based on regular expressions built from rule labels, while the language of \textit{defined strategies} allows the user to define his own elaborated strategies, possibly recursively. Although the language of defined strategies embeds all the constructions of the language of elementary strategies, the latter is maintained since it is based on a more efficient implementation.

Families of strategy rules are intended to give a semantics to the strategy operators. The syntax of strategy rule families is the following:

\[
<\text{family of strategies}> ::= \text{strategies for} \quad <\text{sort name}>
\quad [\quad \{ \quad <\text{strategy variable declare}> \} \quad + \quad ]
\quad \{ \quad \text{implicit} \quad \{ \quad <\text{elementary strategy rule}> \} \quad + \quad \}
\quad \mid \quad \text{implicit} \quad \{ \quad <\text{defined strategy rule}> \} \quad + \quad \}
\quad \mid \quad \text{explicit} \quad \{ \quad <\text{defined strategy rule}> \} \quad + \quad \}
\quad \text{end}
\]

January 27, 2000
3.5 Definition of strategies

<strategy variable declare> ::= [ { <variable name>, }* <variable name> ]
   <sort or strategy sort name> ;

<sort or strategy sort name> ::= <sort name>
   | <strategy sort name>

The keyword explicit is only used with defined strategies and introduces a strategy rule that
contains explicitly in its left-hand side the application operator @@@ that will be introduced
in Section 3.5.2. Otherwise, the strategy rule is introduced by the keyword implicit.

3.5.1 Elementary strategies syntax

The general syntax of elementary strategy rules is the following

<elementary strategy rule> ::= [] <Estrategy name> => <Estrategy term> end

An elementary strategy rule has no label. Thus it is always global.

<Estrategy term> ::= <Estrategy name>
   | choosing
   | concatenation
   | iterator
   | normalize
   | fail
   | id

<choosing> ::= dc ( <Estrategy term> { , <Estrategy term> }* )
   | dk ( <Estrategy term> { , <Estrategy term> }* )
   | first ( <Estrategy term> { , <Estrategy term> }* )
   | dc one ( <Estrategy term> { , <Estrategy term> }* )
   | first one ( <Estrategy term> { , <Estrategy term> }* )

<concatenation> ::= <Estrategy term> ; <Estrategy term>

<iterator> ::= iterate* ( <Estrategy term> )
   | iterate+ ( <Estrategy term> )
   | repeat* ( <Estrategy term> )
   | repeat+ ( <Estrategy term> )

<normalize> ::= normalize ( <Estrategy term> )
   | normalise ( <Estrategy term> )

<Estrategy name> is a constant strategy name declared with the attribute bs.
Labelled rules

A labelled rule is the most elementary strategy and is called a \textit{primal strategy}.

The application of a rewrite rule in ELAN yields a set of results. There are in general several ways to apply a given conditional rule with local assignments. This is first due to equational matching (e.g. AC-matching) and second to the \textit{where} assignment, since it may itself recursively return several possible assignments for variables when using for example non-deterministic strategies.

Note that there may be several rules with the same label. If no rule labelled $\ell$ applies on the term $t$, the set of results is empty and we say that the rule $\ell$ fails.

A labelled rule $\ell$ can be considered as the simplest form of a strategy which returns all results of the rule application.

Thus the language provides basic constructions to handle this non-determinism through choice strategy operators.

Choice strategy operators

Let us first concentrate on expressing choices among the set of results on a primal strategy $\ell$. By default the strategy $\ell$ returns all results of this primal strategy. If at most one result is needed, $\ell$ has to be encapsulated by the strategy operator \texttt{dc one} that returns a non-deterministically chosen result, or \texttt{first one} that returns the first result, when it exists.

\textbf{Example 3.12} Let us consider the following module (incompletely described here):

\begin{verbatim}
empty : set;
@ : ( elem ) set;
@ U @ : ( set set ) set (AC);
element(@) : ( set ) elem;

rules for elem
S : set;
e : elem;

local
[extractrule] element(S U e) => e end
end

The application of the rule extractrule on the term empty U a U b can be done in two ways, since the AC-matching algorithm returns a complete set of matches consisting in the two substitutions: \{ $S \mapsto \text{empty U a}$, $e \mapsto b$ \} and \{ $S \mapsto \text{empty U b}$, $e \mapsto a$ \}.

Using the strategy extractrule on this term results in the set consisting of two terms a and b. \texttt{dc one(extractrule)} returns either a or b depending of the implementation of the non deterministic choice. \texttt{first one(extractrule)} returns either a or b depending of the implementation of the AC-matcher.

More choice operators are defined and apply in general on a list of arguments.

- The \texttt{dk} operator, with a variable arity, is an abbreviation of \textit{dont know choose}.\texttt{dk(S$_1$, \ldots, S$_n$)} takes all strategies given as arguments, and returns, for each of them the set of all its results. \texttt{dk(S$_1$, \ldots, S$_n$)} fails if all strategies $S_1, \ldots, S_n$ fail.

- The \texttt{dc} operator, with a variable arity, is an abbreviation of \textit{dont care choose}.\texttt{dc(S$_1$, \ldots, S$_n$)} selects only one strategy that does not fail among its arguments, say $S_i$, and returns all its results. \texttt{dc(S$_1$, \ldots, S$_n$)} fails if all strategies $S_1, \ldots, S_n$ fail. How to choose $S_i$ is not specified.

January 27, 2000

ELAN user manual
• A specific way to choose an $S_i$ is provided by the first operator that selects the first strategy that does not fail among its arguments, and returns all its results. So if $S_i$ is selected, this means that all strategies $S_1, \ldots, S_{i-1}$ have failed. Again first($S_1, \ldots, S_n$) fails if all strategies $S_1, \ldots, S_n$ fail.

• If only one result is wanted, one can use the operators first one or dc one that select a non-failing strategy among their arguments (either the first or anyone respectively), and return a non-deterministically chosen result of the selected strategy.

Example 3.13 Consider the following module:

```plaintext
module testStrat 1
sort foo; 2
  end 3
operators 4
  global 5
  a : foo; 6
  b : foo; 7
  c : foo; 8
  d : foo; 9
  e : foo; 10
  f : foo; 11
end 12
stratop 13
  global 14
  strat : <foo -> foo> bs ; 15
end 16
rules for foo 17
  global 18
  [r1] a => c end 19
  [r2] a => d end 20
  [r3] b => e end 21
  [r4] b => f end 22
  [r5] c => b end 23
end 24

strategies for foo 25
  implicit 26
  [] strat => dc(dk(r1), dk(r2), dk(r3, r4)) end 27
end 28
```

Then the strategy strat applied to a results in c, and the same strategy applied to b results in \{e,f\}.

Note that the same rule label can be used for naming different rules. In fact this is equivalent to design rules with different labels and a choice operator to link them. For instance in Example 3.13, we could have the label r3 again instead of r4 for the fourth rule. The strategy could then be defined as dc(dk(r1), dk(r2), dk(r3)).

Strategies concatenation

Two strategies are concatenated by applying the second strategy on all the results of the first one.

The concatenation operator denoted "::" builds the sequential composition of two strategies $S_1$ and $S_2$. The strategy $S_1; S_2$ fails if $S_1$ fails, otherwise it returns all results (maybe none) of $S_2$ applied to the results of $S_1$.

Example 3.14 In the same context as in the previous example (Example 3.13), the strategy:

```plaintext
strategies for foo 1
  implicit 2
  [] conc => 3
    dc(r1) ; 4
    dc(r5) ; 5
```

January 27, 2000

ELAN user manual
\texttt{dk(r3, r1)} \\
end \\
end

dk(r3, r1)

when applied on the term a results in the application of first \texttt{r1}, then \texttt{r5}, and finally \texttt{r3} and \texttt{r4} in all possible ways, yielding the results \{e,f\}.

**Strategy iterators**

In order to allow for the automatic concatenation of the same strategy, ELAN offers four powerful iterators:

The strategy \texttt{iterate^*} corresponds to applying zero, then one, then two, ..., \( n \) times the strategy to the starting term, until the strategy fails. Thus \((\texttt{iterate^*(s)})^t\) returns \(\bigcup_{n=0}^{\infty} (s^n)^t\).

Notice that \texttt{iterate^*} returns the results one by one, even when the iteration of the strategy never fails. This strategy cannot fail since it returns at least the initial term.

Its variant \texttt{iterate^+} does not return the initial term but returns \(\bigcup_{n=1}^{\infty} (s^n)^t\). It can fail if \(s\) fails on the initial term.

The strategy \texttt{repeat} iterates the strategy until it fails and returns just the terms resulting from the last unfailing call of the strategy. The two variants are thus defined as:

\((\texttt{repeat^*(s)})^t = (s^n)^t\) where \((s^{n+1})^t\) fails, \(n \geq 0\) and \(s^0 = \text{id}\),

\((\texttt{repeat^+(s)})^t = (s^n)^t\) where \((s^{n+1})^t\) fails and \(n > 0\).

**Example 3.15** To illustrate the basic behaviour of the elementary strategies, let us consider the following module:

```elang
module stratFail
import global int bool;
end

sort foo : end

operators
global
  a : foo ; b : foo ;
end

stratop
global
  trylt : <foo -> foo> ba ;
  repeatS : <foo -> foo> ba ;
  repeatP : <foo -> foo> ba ;
end

rules for foo
global
[a2b] a => b
dfd[:1] a2b [f2a] a => b
end

strategies for foo
implicit
  /\ trylt => dc(a2b) end
  /\ repeatS => repeat*(dc(a2b)) end
  /\ repeatP => repeat^+(dc(a2b)) end
end
```

When applying the strategies on the terms \(a\) or \(b\), we are obtaining the following results:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>trylt</td>
<td>{b}</td>
<td>{}</td>
</tr>
<tr>
<td>repeatS</td>
<td>{b}</td>
<td>{b}</td>
</tr>
<tr>
<td>repeatP</td>
<td>{b}</td>
<td>{}</td>
</tr>
</tbody>
</table>

January 27, 2000
In particular, the difference between the repeat* and repeat+ is due to the fact that when dc(a2b) is applied zero time, this is by definition the identity strategy, returning the initial term.

Example 3.16 The extraction of the elements of a list can be realized in the following way:

```plaintext
module iterRepeat
import global int bool;
end

sort elem nlist; end

operators
global
  Θ : ( int ) nlist ;
  Θ . Θ : ( int nlist ) nlist ;
element(Θ) : ( nlist ) int ;
end

stratop
global
  all0 : <int → int> bs ;
  allRepS : <int → int> bs ;
  allRepP : <int → int> bs ;
  allIter : <int → int> bs ;
end

rules for int
e : int;
l : nlist;

global
  [extract] element(e) => e end
  [extract] element(e.l) => e end
  [extract] element(e.l) => element(l) end
end

strategies for int
implied
  [all0 => dk(extract) ] end
  [allRepS => repeat* ( dk(extract) ) ] end
  [allRepP => repeat+ ( dk(extract) ) ] end
  [allIter => itemte* ( dk(extract) ) ] end
end

We then obtain the following results for the different strategies applied on the terms 1 or element(1.2.3) respectively:

<table>
<thead>
<tr>
<th>strategy</th>
<th>1</th>
<th>element(1.2.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>all0</td>
<td>∅</td>
<td>{1, element(2.3)}</td>
</tr>
<tr>
<td>allRepS</td>
<td>1</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>allRepP</td>
<td>∅</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>allIter</td>
<td>1</td>
<td>{element(1.2.3), element(2.3), element(3), 1, 2, 3}</td>
</tr>
</tbody>
</table>

The normalize strategy

Since labelled rules are applied only at the top of the terms, it is useful to have at hand a way to apply a family of labelled rules everywhere in a term to get it normalized. This is provided in ELAN by the normalize strategy. This strategy takes a strategy and use it in order to normalize the term on which it is applied.
There is no assumption of the way the rules are applied, e.g. neither innermost nor outermost.

**Example 3.17** A typical example of the use of normalize is the normalization process of the typed lambda calculus, whose main module is described as follows:

```haskell
module normLambda
import global int;
  local bool eqTerm;
end
sort Term;
end
operators global
  @ : (int) Term ;
  (@ @) : (Term Term) Term ;
  la @ : (Term) Term ;
local
  repl(@, @, @) : (int Term) Term ;
  free(@, @) : (int Term) bool ;
end
stratop global
  betaEta : <Term -> Term> bs ;
end
rules for bool
v, w : int;
M, N, Q : Term;

global
[1] free(v, v) => true end
[2] free(v, M N) => free(v, M) or free(v, N) end
[3] free(v, la M) => free(v, M + 1) end
[4] free(v, M) => false end
end

rules for Term
v, w : int;
M, N, Q : Term;

global
[1] repl(v, v, M) => M end
[2] repl(v, w, M) => w end
[3] repl(v, M N, Q) => repl(v, M, Q) repl(v, N, Q) end
[4] repl(v, la N, M) => la repl(v, N, M + 1) end
[5] beta1(la M N) => repl(1, M, N) end
[6] eta1(la M 1) => M if not free(1, M) end
end
strategies for Term
  implicit
  [1] betaEta => normalize first one beta eta end
end
```

**Identity and Failure**

Two more constructions are available:

- **id** is the identity strategy that does nothing, and never fails.

- **fail** always fails and returns an empty set of results.
3.5.2 Defined strategies

The defined strategy language extends the elementary one by the possibility to define recursive strategies. The application of a defined strategy is itself performed by an ELAN program that defines the interpretation of the new constructions introduced by the user. All the modules defining the appropriate syntax and semantics are provided in the ELAN library.

The syntax of the defined strategy language is described in an ELAN module called `strs{g[X,Y]` when strategies ranks contain sorts of the form `<X->Y>` (strategies are non sort-preserving), and the module `str[X]` when the strategies concern sorts of the form `<X->X>` (strategies are sort-preserving). To use non sort-preserving strategies, one has to import the module `tstrat[X,Y]` where X and Y are different sorts. These modules implement the interpreter of the defined strategy language using a set of labelled rewrite rules and a basic strategy `eval`. The syntax of defined strategies is given below:

\[
\text{<defined strategy rule> ::=}
\]
\[
\begin{align*}
& | \quad [ \ ] \text{<strategy body> end} \\
& | \quad [,] \text{<strategy body> end} \\
& | \quad [ \text{<rule label> } ] \text{<strategy body> end}
\end{align*}
\]

\[
\text{<strategy body> ::= <Dstrategy term> == > <Dstrategy term> [ \{ <if-where-choose> \} + ]}
\]

`<if-where-choose>` is defined in Section 3.4.1.

`<Dstrategy term>` is a term built on the signature of strategy operators. It has a strategy sort. The different labels `[`[], `[` and `[` for strategy rules correspond to different evaluation modes explained in Section 3.6.2.

**Example 3.18** An example of the use of these features is the following:

```elang
module map[X]
import
global
  strat[X] strat[list[X]] X list[X];
end stratop
global
  map(θ) : (<X -> X>) list[X] -> list[X];
  start θ : (int) list[X] -> list[X];
end rules for X
  z : X;
  n : int;
end global
  [add(n)] z => n + n end
  [sub(n)] z => z - n end
end strategies for list[X]
  s : <X -> X>; k, h, t : X;
  t, t1 : list[X]; n : int;
end explicit
  [1] [map(s)] nil => nil end
  [1] [map(s)] h.t => h.t.t1 where h1 := [p] h.
```

January 27, 2000

ELAN user manual
\[ \text{where } \textit{if:} = \text{map}(s) \]  
\[ \text{end} \]

**Example 3.19** The following module shows how to program loops with the defined strategy language.

\[
\begin{align*}
\text{module } & \text{loop} \\
\text{import } & \text{global strat[int] int} \\
& \text{strong [int,int,int]:} \\
\text{end} \\
\text{stratop } & \text{global} \\
& \text{loop } : \langle \text{int} \rightarrow \text{int} \rangle; \\
\text{end} \\
\text{rules for } & \text{int} \quad z : \text{int}; \\
\text{local} & \quad [\text{sub1}] \quad z \rightarrow z - 1 \text{ if } z > 1 \text{ end} \\
\text{end} \\
\text{strategies for } & \text{int} \\
\text{implicit} & \quad [\text{loop}] \rightarrow \text{dk(id, sub1;loop)} \text{ end} \\
\text{end}
\end{align*}
\]

3.6 Evaluation mechanism

This section does not introduce any new syntactic material, but gives informal explanations on the operational semantics of the language.

3.6.1 Rewrite rules on terms

As we have seen, there are two kinds of rules: labelled and unlabelled. They define two classes of rules that are applied on the term to be evaluated in the following way:

- **Step 1** The current term is normalized using the unlabelled rules. This is done in order to perform functional evaluation and thus it is recommended to the user to provide a confluent and terminating unlabelled rewrite system to ensure termination and unicity of the result. This normalization process is built-in in the evaluation mechanism and consists in a leftmost innermost normalization. This yields always a single result.

- **Step 2** Then one tries to apply on the normalized term a labelled rule following the strategy described in the logic description. This leads to a (possibly empty) collection of terms. If this set is empty, then the evaluation backtracks to the last choice point; if it is not empty, then the evaluation goes on by setting a new choice point and evaluating one of the returned terms by going to **Step 1**.

In a slightly more formal way, a rule

\[ [\ell] \quad l \rightarrow r \quad s_1 \ldots s_n \]

where the \( s_i \) are either \textbf{where} or \textbf{if} or \textbf{choose} \textbf{try} expressions, is applied on a term \( t \) by:
1. Matching \( l \) against \( t \). This computes a multiset of substitutions (because of equational matching). If this set contains more than two elements, one is chosen and the other ones are stored for possible future backtracking. Let \( \sigma \) be the chosen substitution.

2. The evaluation goes on by evaluating the expressions \( s_1, \ldots, s_n \), one by one and in this order (i.e. from 1 to \( n \)).

3. If \( s_i \) is of the form \textbf{where} \( x_i := (\text{strat}_i)t_i \), then one of the results (call it \( t'_i \)) of the application of the strategy \( \text{strat}_i \) on the term \( t_i \) is chosen, and the substitution \( \sigma \) is extended by \( x_i \mapsto t'_i \). The other results are stored for possible backtracking, and the evaluation goes on with \( s_{i+1} \). If the strategy \( \text{strat}_i \) fails on \( t_i \), then we backtrack to the previous choice point.

4. If \( s_i \) is of the form \textbf{where} \( p_i := (\text{strat}_i)t_i \), then one of the results (call it \( t'_i \)) of the application of the strategy \( \text{strat}_i \) on the term \( t_i \) is chosen, and the substitution \( \sigma \) is extended by the matching substitution \( \sigma_i \) from \( p_i \) to \( t'_i \). The other results are stored for possible backtracking, and the evaluation goes on with \( s_{i+1} \). If the strategy \( \text{strat}_i \) fails on \( t_i \), then we backtrack to the previous choice point. Note that when \( p_i \) contains AC symbols AC-matching is called and there is an additional backtracking mechanism to find an adequate AC-match \( \sigma_i \).

5. If \( s_i \) is of the form \textbf{if} \( c_i \), then the term \( c_i \) is evaluated following the normalization strategy. If the result is the boolean constant \textbf{true}, then one evaluates the next expression \( s_{i+1} \), otherwise one backtracks to \( s_{i-1} \).

6. If \( s_i \) is of the form
   
   \begin{verbatim}
   choose
   try \( s_1 \) \ldots \( s_m \)
   \[
   \ldots \\text{try} \ s_j \ldots \ s_k \\text{try} \ s_{j+1} \ldots \ s_l \end{verbatim}
   then we can mimic the evaluation mechanism by replacing all occurrences of \( l \) in strategy expressions by \( p \) rules \( \ell_1, \ldots, \ell_p \) (in this order) defined as follows for \( j = 1, \ldots, p \):
   
   \[
   [\ell_j] \quad l \rightarrow r \quad s_1 \ldots s_{i-1} s_j \ldots s_k s_{i+1} \ldots s_n
   \]

   Since \textbf{choose try} is a recursive construction, this unfolding process is performed by a leftmost-innermost mechanism, until we get rule expressions with no occurrence of \textbf{choose try}.

\[\triangleright\text{One should note that the term to be evaluated is first normalized by ELAN using the non-labelled rules and the resulting term is then reduced using the given strategy. The following example illustrates this behaviour.}\]

\noindent**Example 3.20**

\begin{verbatim}
module normalizeFirst
    \end{verbatim}

\noindent// The module identity is part of the standard library

January 27, 2000
import
    local identity[term];
end

sort term; end
operators
global
    a : term;
    f(a) : (term) term;
end
stratop
global
    s1 : <term -> term> bs;
    s2 : <term -> term> bs;
end
rules for term
    z,y,z : term;
    [r1] f(z) => z end
    [] f(z) => z end
end
strategies for term
    implicit
        [] s1 => dc(r1) end
        [] s2 => dc(r1,identity) end
end
end
end

Applying the strategy s1 on the term f(a) gives no result since:

1. f(a) is normalized by the non-labelled rule into the term a,

2. then the normalization process tries to apply the labelled rule r1 on a and fails. So no strategy applies! The set of results is empty.

When applying the strategy s2 on f(a), for the same reason the reduction process fails to apply the rule r1 but the strategy identity can then be applied and the result is then a since the term is firstly normalized.

Example 3.21 Let us consider a brute force specification of the eight queens problem: How can one put eight queens on one chessboard in such a way that they do not interfere? In this example p1 refers to the position of the queen in the first column, p2 to the position of the second queen which should be in the second column and so on up to p8.

module queens
    import
        bool int list[int];
    end

operators
global
    queens : list[int];
    local
        ok(O,O,O) : (int int list[int]) bool;
end
stratop
global
    queens : <list[int] -> list[int]> bs;
    local
        try_step : <int -> int> bs;
end

January 27, 2000

ELAN user manual
3.6 Evaluation mechanism

rules for list [int]
pl, p2, p3, p4, p5, p6, p7, p8 : int;
pp1, pp2, pp3, pp4, pp5, pp6, pp7, pp8 : list [int];
local
[queensrule] queens => p8.pp7
  // - first: the position on the first row is chosen
  where p1 := (try_step) 0
  // - second: the position on the second row is also chosen
  where p2 := (try_step) 0
  where pp1 := () p1.nil
  // - third: the positions on the first and second rows are checked to be compatible
  if ok (1, p2, pp1)
    // - quarto: choose a position on the third row
    // and continue ...
    where p3 := (try_step) 0
    where pp2 := () p2.pp1
    if ok (1, p3, pp2)
      where p4 := (try_step) 0
      where pp3 := () p3.pp2
      if ok (1, p4, pp3)
        where p5 := (try_step) 0
        where pp4 := () p4.pp3
        if ok (1, p5, pp4)
          where p6 := (try_step) 0
          where pp5 := () p5.pp4
          if ok (1, p6, pp5)
            where p7 := (try_step) 0
            where pp6 := () p6.pp5
            if ok (1, p7, pp6)
              where p8 := (try_step) 0
              where pp7 := () p7.pp6
              if ok (1, p8, pp7)
            end
          end
        end
      end
    end
  end
end
rules for bool
p, d, diff : int;
l : list [int];
global
[ok] ok (diff, d, nil) => true
[ok] ok (diff, d, p, nil) => false if d == p
[ok] ok (diff, d, p, 1) => false if d - p == diff
[ok] ok (diff, d, p, 1) => false if p - d == diff
[ok] ok (diff, d, 1) => ok (diff + 1, d, 1)
end
rules for int
z : int;
local
[tryrule] z => 1 end
[tryrule] z => 2 end
[tryrule] z => 3 end
[tryrule] z => 4 end
[tryrule] z => 5 end
[tryrule] z => 6 end
[tryrule] z => 7 end
[tryrule] z => 8 end
end

strategies for int
implicit
[try_step] => dk (tryrule) end
end

strategies for list [int]
implicit

Note that the strategy above returns only one solution. If it is changed to a \textbf{dk}, we get all possibilities.

Note also the way all the numbers between 1 and 8 are enumerated in \textit{ELAN} using a \textbf{dk} strategy (see \texttt{tryrule}).

\subsection*{3.6.2 Rewrite rules on strategies}

Elementary strategies are defined by rules of the form \[ \text{[] c} \rightarrow \text{strat} \] where \texttt{c} is a constant of sort \texttt{<s \rightarrow s'>} and \texttt{strat} a term built on elementary strategy constructors. Such rules are always unlabelled. Application of such a strategy \texttt{c} on a term \texttt{t}, denoted \((\texttt{c})\texttt{t}\) is performed by a \texttt{C} function generated by \textit{ELAN}.

Defined strategies are like ordinary terms except that they have a functional sort of the form \texttt{<s \rightarrow s'>}. To understand how rules are applied on these strategy terms, it is helpful to distinguish two kinds of rule purposes: either the rule defines the result of the application of a strategy to a term (and involves implicitly or explicitly the application operator \([\texttt{[]}] \texttt{t}\)), or the rule defines a computation on strategy terms.

- Consider first the case of a rule which defines the result of the application of a strategy to a term.
  
  - When the rule is labelled by \([.]\), it is used by the strategy interpreter (i.e. the \textit{ELAN} program given in the module \texttt{strat[X]} for sort preserving strategies or \texttt{strat[X,Y]} for others). For that, the rule is automatically transformed into a rule with the label \texttt{DSTR} and is applied with the default strategy \texttt{eval} of the strategy interpreter.
  
  - When the rule is labelled by \([\ell]\), it is used by the \textit{ELAN} interpreter as any other labelled rule governed by a strategy. So the user has to provide a strategy involving \(\ell\).

- Rules that define a computation on strategy terms are evaluated exactly like rules on (ordinary) terms:
  
  - If the rule is unlabelled, the leftmost innermost predefined strategy of the interpreter is applied.
  
  - If it is labelled by \([\ell]\), the evaluation process needs a user-defined strategy involving \(\ell\).

\textbf{Example 3.22} We define here a small strategy library composed of several defined strategies that provide primitives for depth-first search in a computation tree. This strategy library is used to build a strategy that finds out a path (or, all paths) towards an exit of a labyrinth.

The strategy library supports four primitives for depth-first search. The current situation of the partially discovered search tree is represented by a path from the initial state (the root of the tree) to the current state (the left-most leaf). It also remembers all possible alternatives (or, choice points) met along this path. Each step of the path is represented by a list of states, where its head is the currently chosen state, and the rest represents all non-explored alternative states. Each step of the path is a list of states, thus, the whole path could be naturally represented as \texttt{list[\texttt{list[state]}]} . This sort also describes the current situation during the depth-first search in this tree.

Four primitive strategies are defined over the sort \texttt{list[\texttt{list[state]}]}.
• **call(S)** applies the strategy S on the current state, where S is of sort \((\text{state} \rightarrow \text{state})\). All possible results of this application, if there are any, are grouped together into a new step (level) attached to the current path.

• **next** throws away the current state and continues searching with the next possibility of the previous step (the last choice-point).

• **exit** leaves the current state ignoring all alternatives and it returns the control to the previous step of the current path.

• **cut** eliminates all alternative states of the last step.

```plaintext
module space

import global list[state]
   list[ list[state] ]
   eq[ state ] eq[ list[state] ]
   strat[ state ]
   strat[ list[state] ];
end

sort state; end

stratop global
  next : < list[ list[ state ] ] -> list[ list[ state ] ]>
  exit : < list[ list[ state ] ] -> list[ list[ state ] ]>
  cut : < list[ list[ state ] ] -> list[ list[ state ] ]>
  call() : < state -> state > list[ list[ state ] ] -> list[ list[ state ] ]>
  loop : < list[ list[ state ] ] -> list[ list[ state ] ]>
  narrow : < list[ list[ state ] ] -> list[ list[ state ] ]>

  local
  occurs @ in @ : ( state list[ list[ state ] ] ) bool;
end

rules for bool
  state1, state2 : state;
  level : list[ state ];
  space : list[ list[ state ] ];

global
  [] occurs state1 in nil => false end
  [] occurs state1 in state2.level.space => true if state1 == state2 end
  [] occurs state1 in state2.level.space => occurs state1 in space end
end

strategies for list[ list[ state ] ]
  state : state;
  s : < state -> state >;
  level : list[ state ];
  space : list[ list[ state ] ];
  new_level : list[ state ];
  new_space : list[ list[ state ] ];

explicit
  [] [ next ] state.level.space => level.space end
  [] [ exit ] state.level.space => space end
  [] [ cut ] state.level.space => state.nil.space end
  [] [ call(s) ] state.level.space => new_level.state.level.space
      where new_level := ( setof( s state )
                                    if new_level != nil end
                               )
  [] [ loop ] state.level.space => state.level.space if occurs state in space end
  [] [ narrow ] state.level.space => state.nil.new_space
      where new_space := narrow space end
  [] [ narrow ] nil => nil end
end
```
The variable **state** represents the current state, the variable **level** represents all alternatives of the current state, i.e. \( \text{state.} \text{level :} \text{list[state]} \) is one step of the path, where the others steps are recorded in the variable space.

The only non-trivial strategy among the four primitives is the strategy \( \text{call(S)} \), which uses the function symbol \( \text{set} \) of to collect all results of an application of a strategy \( S \).

In order to design a strategy helping to find out an exit from a labyrinth, we split the problem into a part dependent on the labyrinth and an independent one. The dependent part contains a specification of a state, which is a pair of coordinates, and a definition of four basic moves in the labyrinth. They are realized as state transforming strategies \( \text{left, right, up, down} \) of sort \( \langle \text{state} \rightarrow \text{state} \rangle \) such that these strategies fail, if some movement in a given state is not possible (because of a wall). All exits of the labyrinth are specified by a strategy \( \text{exitable} \), which fails, if the current state has ‘no exit doors’. Otherwise, it is equivalent to the identity strategy.

```elang
module robot
import global state space
strat[state] strat[list[state]]
int bool labget
stronc[list[state]], list[list[state]] list[list[state]];
end
stratop global
up : <state -> state>; down : <state -> state>; left : <state -> state>; right : <state -> state>; exitable : <state -> state>;
search : <list[list[state]] -> list[list[state]]>; altsearch : <list[list[state]] -> list[list[state]]>;
local
moves : <state -> state>;
end

rules for state
z,y,dz,dy,dir : int;
local
[move(dz,dy,dir)]
state(z,y) => state(z+dz,y+dy)
if room(z,y) & dir != 0
end

strategies for state
implicit
[] up => move(1,0,U) end
[] down => move(-1,0,D) end
[] left => move(0,-1,L) end
[] right => move(0,1,R) end
[] exitable => move(0,0,E) end
[] moves => dk(right, dk(down, dk(left, up))) end

strategies for list[list[state]]
implicit
[] search => if call(exitable) then narrow
otherwise if loop then next;search
otherwise dk(call(moves),search, next;search)
fi
[] altsearch => dk(call(exitable);narrow,
if loop then next
otherwise dk(call(moves), next)
```

January 27, 2000
The independent part of the strategy consists of a definition of the strategy search of sort \langle list \circ list \circ state \rangle \rightarrow list \circ list \circ state] \rangle and an auxiliary strategy moves of sort \langle state \rightarrow state \rangle. The strategy search succeeds on an exitable state (a room with exit doors). Otherwise, if the current path contains a loop, it backtracks and takes the next possible state to go on. If it does not contain a loop, it either moves to the neighbour state and goes on searching, or, if no exit has been found, it ignores the current choice, and backtracks to the next alternative.

3.7 Modules

The modularity constructions in the current version of ELAN are as simple as possible and the semantics of parametrisation as well as importation is textual expansion.

The top-level information that should be provided by the super-user is the description of the logic(s) he wants to make available. This is achieved in ELAN by using two kinds of modules. The first one describes the top level of the logic and are presented in files having the ‘.lgi’ extension, the second ones contain all subsequently needed informations. They are called element modules and they are presented in files having the extension ‘.elm’. Element modules have been described in Section 3.2.

3.7.1 Visibility rules

The visibility rules in ELAN are the following:

- Only sorts are global. This means that they can be used in any module without any importation command. As a consequence, their name is unique in the ELAN universe built for a run.

- Operators, rewrite rules and strategies can on the contrary be local or global. A local operator will be known only in its own module (i.e. the module where it is defined). If an operator is declared to be global, then it is known in all modules importing the module where it is declared. This can be slightly modified, when qualifying the importation command by local or global. By default an importation without local or global specification is assumed to be local.

- A special case is for aliases. The visibility rules for them are the same as for operators. The only interesting case is for a module that locally imports a signature but exports the alias definition. In this case, only the alias name of the operator will be known outside and not its original name: this is quite useful for renaming purposes.

— Release Note: The main change concerning the visibility rules is that now only sorts are always global. See section 3.9 for a summary of all syntax modifications.

3.7.2 Built-in modules

For convenience and efficiency several modules are provided as built-ins. See section 5.2 for a full description of the current available ones.
3.7.3 Parameterised modules

Parameterised modules can be defined in ELAN as follows:

\[
<\text{formal module name}> ::= <\text{identifier}>
\quad | \quad <\text{identifier}> \{ <\text{identifier}>, <\text{identifier}> \}^* 
\]

The module instantiation by its actual parameters is currently done by the syntactical replacement of the formal arguments by the actual ones.

Example 3.23 The classical example is the one of parameterised lists:

```
module simpleList[X]
  sort X simpleList[X]; end
operators
  global
  nil : simpleList[X];
  @ . @ : \{ X simpleList[X] \} simpleList[X];
  append(\@, @) : \{ simpleList[X] simpleList[X] \} simpleList[X];
end
rules for simpleList[X]
  e : X;
  list : simpleList[X];
  global
  [] append(nil, 1) => 1 end
  [] append(e, list, 12) => e . append(list, 12) end
end
```

and we get list of integers by instantiating X by int and importing int as in:

```
LPL simpleList description
  query of sort simpleList[int]
  result of sort simpleList[int]
  import int simpleList[int]
  start with (query
end
```

3.7.4 LGI modules

The top level description of the logic consists of:

1. the description of the syntax that the user is allowed to use in order to give his specifications.

2. the description of a term, called the query, that will be reduced in the rewrite logic defined by the (super) user.

The syntax is the following:

```
<pl description> ::= LPL <identifier> description
  \[ <\text{specification description}> \]
query
  of sort <\text{sort name}>
result of sort <\text{sort name}>
import \{ <\text{instantiated module name}> \} +
  \[ check with <\text{boolean term}> \]
start with <\text{start term}>
end
```
3.7 Modules

<specification description> ::= specification description
   { <specification part description> } +
end

<specification part description> ::= part <identifier>
   of sort <sort name>
   import { <instantiated module name> } +
   [ check with <boolean term> ]

<start term> ::= ( [ <strategy name> ] ) <query term>

The syntax of of <query term> is built over the user-defined signature enriched by the constant query. The sort of query is given by the query declaration above.

Example 3.24 If no specification is needed, the specification description part is skipped, as in the following logic description used for running Example 3.13:

```lpl
LPL testStrat description
query foof of sort foof
result foof
import testStrat
start with (strat) query
end
```

Example 3.25 Here is a simple example of the top level description of how unification can be implemented:

```lpl
LPL unification description
   // This part describes the syntax in which the user should
   // complete the definition of the logic: i.e. how (s)he gives
   // the specification.
   specification description
   part Vars of sort list[identifier]
      import sigSyntax
   part Ops of sort list[var[identifier,int]]
      import sigSyntax
   end
   // The following part describes the way the user should gives the query
   // and how it will be evaluated
   query of sort constraint
      result of sort constraint
      import unification[Vars,Ops]
      constraint[Ops,Vars]
      termCommons[Ops,Vars]
      start with (unify) query
   end
```

In this unification logic, the user should provide the specification information as follows:

```lpl
specification simplesig
   Vars 2 3 2 2 w w
   Ops a:0 b:0 c:0 f:2 g:1
end of specification
```

The user should use the (super-user defined) keywords Vars and Ops to define respectively the variables and the operators, namely a of arity 0, f of arity 2, etc...

January 27, 2000

ELAN user manual
Semantics

When parsing a specification, ELAN generates new modules that complete the definition of the rewrite theory to be used during the evaluation of the query. For instance, in the case of the previous example, these new modules are:

```plaintext
module Vars
import anyIdentifier sigSyntax; end
operators
  global
    Vars : list[identifier];
end
rules for list[identifier]
  global
    [] Vars => x y z ;
end
end

module Ops
import anyIdentifier identifier sigSyntax list[pair[identifier,int]]; end
operators global
Ops : list[pair[identifier,int]]; end
rules for list[pair[identifier,int]]
  global
    [] Ops => a 0 b 0 c 0 f 2 g 1 end
end
end
```

When a user provides a specification to an ELAN logic, this simply provides more information about the context in which the user wants to work, like the name of function symbols or basic axioms to use.

3.8 Pre-processing

Another original feature proposed by ELAN is the use of a pre-processing phase that allows to describe the logic to be encoded in an easier way.

As described in Figure 1.1, the pre-processor is performing textual replacements starting from informations given by the super-user in the modules, and by the user in the specification file. The pre-processing phase in ELAN can just be thought of as a macro expansion mechanism which extends the parameterization process described before.

In order to express the syntax of the pre-processor construction, we need the notion of constant expression and of text defined as follows:

```plaintext
<constant expression> ::= <number>
  | ( <subexpression> )

<subexpression> ::= <number>
  | <subexpression> + <subexpression>
  | <subexpression> - <subexpression>
```

January 27, 2000
3.8 Pre-processing

| <subexpression> * <subexpression> |
| <subexpression> / <subexpression> |
| <subexpression> % <subexpression> |
| ( <subexpression> ) |

<lexem> ::= <identifier>
| <number>
| <char>

<text> ::= { <lexem> } +

3.8.1 Simple duplication

A first feature is to allow simple duplications of a part of text. The syntax is:

<simple repetition> ::= <lexem> ~ <constant expression>
| { <text> } ~ <constant expression>

Example 3.26 The text “a{b}”3” is processed into “a, b, b, b”.

3.8.2 Duplication with argument

It is often necessary to allow description of objects like \( f(t_1, \ldots, t_5) \). This is possible using the syntax:

<arg.repetition> ::= { <text> }_<identifier>

<constant expression> ...
<constant expression>

Example 3.27 The text “P { & s_1=t_1 \_ i=1 \ldots 3” is processed into “P & s_1=t_1 & s_2=t_2 & s_3=t_3”.

A special form of this duplication with arguments is the explicit construction of a list of indexed identifiers allowed using the syntax:

<arg.repetition> ::= <identifier> _ <number> <char> ... <char> <identifier> _ <number>

Example 3.28 The text “t_1, t_2, t_3, t_4, t_5” is processed into “t_1, t_2, t_3, t_4, t_5”.

3.8.3 Enumeration using FOR EACH

This construction allows to make the link between the specification given by the user and the logic described by the super-user. A natural motivation for this construction is given by the “standard” way inference rules are used. For example when describing how unification works, the following transformation rule is given:

Decompose \( P \land f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \rightarrow P \land s_1 = t_1 \land \ldots \land s_n = t_n \)

It is generic in the sense that the operator \( f \) is unspecified. It can be a + of arity 2, or a if@then@else@ of arity 3, or just a constant. We do not want, when specifying how the logic works, to give only the specific cases, we need to be as generic as possible.

ELAN provides via the FOR EACH construction of the pre-processor, a way to be generic. The syntax is the following:

January 27, 2000
ELAN user manual
<for each const.> ::= FOR EACH <varexpt> SUCH THAT <varaffect> : { <text> }

<varexpt> ::= <varname> { , <varname> } * : <sort name>
   | <varexpt> ; <varexpt>

<varaffect> ::= <varname> ::= ( [ <strategy name> ] ) <term>
   | <varaffect> AND <varaffect>
   | <varaffect> ANDIF <boolean term>

▷ Since the symbols {, }, are reserved for pre-processor use, it is not possible to use them for any other purpose, even quoted.

▷ The pre-processor uses the character ‘_’ (underline) for assembling identifiers. For example, the text ‘a f _1 x b’ is pre-processed into the sequence of three identifiers a, f_1 x and b. The character ‘_’ is thus part of an identifier even if there is a space between it and the rest of the string. It is thus better for the user not to use the _ symbol, except for building identifiers of course.

Example 3.29 The rule Decompose that we mentioned at the beginning of this section can be expressed in the following way:

\[
\begin{align*}
&\text{FOR EACH } SS:\text{pair}\{\text{identifier, int}\} ; F:\text{identifier} ; N:\text{int} \\
&\text{SUCH THAT } SS:=\{\text{listExtract}\} \text{ elem}(SS) \text{ AND } F:=\{ \text{first}\{SS\} \text{ AND } N:=\{ \text{second}\{SS\} \} : \\
&\text{rules for constraint} \\
&\quad s_1, \ldots, s_N \text{: term; } l_1, \ldots, l_N \text{: term;}
&\begin{align*}
&\quad \text{local} \\
&\quad \text{[decompose]} \ P \& F(s_1, \ldots, s_N) = F(l_1, \ldots, l_N) \Rightarrow \ P \{ \& s_j = l_j \} \text{ if } j = 1 \ldots N \\
&\quad \text{end}
&\end{align*}
&\text{end}
\end{align*}
\]

If the specification given by the user consists in the operator symbols f and g of respective arities 2 and 1, then the pre-processor expands the previous FOR EACH construction into:

\[
\begin{align*}
&\text{rules for constraint} \\
&\quad s_1, s_2, l_1, l_2 : \text{term;}
&\begin{align*}
&\quad \text{local} \\
&\quad \text{[decompose]} \ P \& f(s_1, s_2) = f(l_1, l_2) \Rightarrow \ P \& s_1 = l_1 \text{ and } s_2 = l_2 \text{ end} \\
&\quad \text{[decompose]} \ P \& g(s_1) = g(l_1) \Rightarrow \ P \& s_1 = l_1 \text{ end} \\
&\end{align*}
&\text{end}
\end{align*}
\]

3.9 Differences with previous version of the language

The language has evolved a lot between ELAN version 1.17 and ELAN version 3.0. The changes concern the syntax of:

- modules,
- rules and
- strategies.

January 27, 2000
The main changes are emphasized in the corresponding sections of the language description, and the main concern has been to uniformize the language constructions.

The transformations needed to change from the syntax of version 1.17 to the current one (i.e. later than 2.00) are summarized below.

Concerning the declaration parts:

- import ... becomes import ... end
- sort ... becomes sort ... end
- op ... endop becomes operators ... end

Now, there is only one constructions to define rules, you have to replace:

- rule for ... by rules for ...
- rule <name> for ... by rules for ... [name] ...
- suppress declare keyword
- body or bodies by global or local
- end of rule, end of rules by end

The rule labels accept now arguments that should be variables involved in the rule. Definition of strategies looks like rules definitions, you have to replace:

- strategy ... by strategies for ... global|local [name] ...
- iterate ... enditerate by iterate*(...)
- while ... endwhile by repeat*(...)
- repeat ... endrepeat by repeat*(...)
- dont know choose(...) by dk(...)
- dont care choose(...) by dc(...)
- end of strategy by end
- end of module by end

Local affectations and conditions are now evaluated from top to bottom. The order of presentation has been reversed, so that the old syntax:

```
[] l => r
  if c2
    where v2:=(...) ...
  where v1:=(...) ...
  if c1
```

becomes in the new syntax:

```
[] l => r
  if c1
    where v1:=(...) ...
    where v2:=(...) ...
  if c2
```
Chapter 4

ELAN: the system

This section is devoted to the user environment of the ELAN language, and is partly borrowed from [BKK+98b].

4.1 Global architecture

The ELAN environment consists of several components, as depicted in Figure 4.1. The preprocessor expands a few concise constructions allowed in the language. The parser checks the syntax of programs and verifies that terms are syntactically well-formed. The interpreter is an interactive tool allowing the user to check that the results he expects are indeed obtained. Initially, the preprocessor and the parser were integrated in the interpreter and so it was not possible to call them as stand-alone processes. The compiler transforms specifications into independent executable C code (Section 4.5). It is a stand-alone tool without any preprocessing and parsing facilities. The preprocessing and parsing phases are provided by the interpreter, which communicates the result of these processes to the compiler via a data exchange format called REF. Section 4.6 describes some tools to translate ELAN to REF and vice versa. Especially, the translation from ELAN to REF performed by the query2ref tool relies mainly on reusing the parser initially integrated in the interpreter.

4.2 The preprocessor

The ELAN syntax provides a few fancy constructions to perform textual replacements. So the preprocessor may be used to automatically generate parts of specifications used to analyse the rest of a program. It should be emphasised that there is strong interaction between the parser, the preprocessor and the interpreter. See Section 3.8 for a description of the preprocessor functionality.

4.3 The parser

The ELAN parser is based on Earley’s one [Ear70] and is using sort informations in order to determine as soon as possible in the analysis the right choice. This explains why in the current version, some construction of the language require to give the sort information together with the term.
4.4 The interpreter

The interpreter takes a well-formed program and a well-formed query (both checked by the parser) and applies the rules and strategies defined in the program to the query. In order to find which rules can apply, the selection is guided by the top symbol of the rules: only those rules whose left-hand side has the same top symbol as the term to be reduced are selected. They are then tried in the order given in the program. Another kind of choices arbitrarily made by the interpreter is for the strategy $\text{dc}(S_1, ..., S_n)$ that should select randomly a non-failing strategy among $S_1, ..., S_n$. In practice, the interpreter selects the first one, so implements $\text{dc}$ and $\text{first}$ in the same way. Note however that there exists a version of ELAN which concurrently executes the $n$ strategies and selects the first one which terminates without failure [BC98].

Once a set of rules is selected, a many-to-one matching algorithm is applied. When associative and commutative ($AC$ for short) operators are involved, an external one-to-one $AC$-matching algorithm described in [Eke95] is called. This algorithm is not fully integrated in the interpreter, so data structure conversions are required and lower the efficiency of the $AC$-matching, already quite complex. Once a match is found, local evaluations are performed and if all succeed, the result term is built, taking advantage of the right-hand side of the rule and of term sharing.

4.5 The compiler

The ELAN compiler transforms a logic description into an executable binary file. The compiler (for detailed description, see [Vit96, MK98]) produces an efficient C code, which is later compiled by the cc or the gnu gcc compiler. The executable binary code is dependent on the particular architecture, because a small part of the ELAN library performing the basic non-deterministic operations is written in assembly language [Mor98]. Up to now, this is the only reason that makes
the compiler available only on the architectures DEC-ALPHA, SUN4 and Intel-PC. This release of the ELAN compiler does not compile the whole ELAN language: the main restriction is that the ELAN compiler does not treat Input/Output and reflective capabilities of the interpreter. There is also a restriction on the class of rules containing AC-symbols that can be compiled.

4.5.1 Associative and Commutative symbols

The current compiler cannot compile all possible patterns that contain AC symbols. In practice it is not a problem since a simple "manual" program transformation can be done in order to make any specification compilable.

The class of patterns that can be compiled is the following:

- 1 AC symbol and 1 variable with multiplicity: $f_{AC}(x^n)$ with $n \geq 2$.

  Considering that $\ast$ is an AC operator, the following rules are correct:

  \[
  \begin{align*}
  & \Box \; x + x \quad \Rightarrow \text{id} \quad \text{end} \\
  & \Box \; x + x + x \quad \Rightarrow x \quad \text{end}
  \end{align*}
  \]

- 1 AC symbol and 2 variables: $f_{AC}(x, y^n)$ with $n \geq 1$

  Suppose that $\cup$ is an AC union operator, we can define the two following rules:

  \[
  \begin{align*}
  & \Box \; \text{lookup}(x, x \cup y) \Rightarrow x \quad \text{end} \\
  & \Box \; x \cup x \cup y \quad \Rightarrow x \cup y \cup y \quad \text{end}
  \end{align*}
  \]

- 1 AC symbol, 0 or 1 variable and $n$ syntactic subterms: $f_{AC}(t_1, \ldots, t_n)$ or $f_{AC}(x, t_1, \ldots, t_n)$

  \[
  \Box \; x \cup \text{empty} \quad \Rightarrow x \quad \text{end}
  \]

- 2 AC symbols and 3 variables: $f_{AC}(x, g_{AC}(y, z))$

  This kind of patterns can be used to describe the distributivity law for example (where $\ast$ and $\ast$ are AC):

  \[
  \begin{align*}
  & \Box \; x \ast (y+z) \quad \Rightarrow x \ast y + x \ast z \quad \text{end}
  \end{align*}
  \]

The AC symbol may occur at a position different from the root, but remind that only one subterm containing AC symbols can occurs in the pattern. This possibility was used in the example:

\[
\Box \; \text{lookup}(x, x \cup y) \Rightarrow x \quad \text{end}
\]

where lookup is not an AC operator.

Suppose now that we have to compile the following rule (extracted from the example given on page 13):

\[
[\text{factorize}] \quad P + n1*p1 + n2*p1 \Rightarrow P + n3*p1
\]

where \( n3 := (n1+n2) \quad \text{end} \)

According to the class of patterns that can be compiled, this rule cannot be compiled without any transformation because the left-hand side contains 3 AC operators. In order to make the compilation possible, an AC level has to be removed and replaced by a variable and a local assignment has to be introduced:

January 27, 2000
[factorize] \[ P + ac1 \rightarrow P + n3*p1 \]
where (poly) \( n1*p1 + n2*p1 \) := () \( ac1 \)
where \( n3:=n1+n2 \) end

With this simple transformation the pattern \( P + ac1 \) can be compiled and the local assignment where (poly) \( n1*p1 + n2*p1 \) := () \( ac1 \) is used to instantiate the variables \( n1, n2 \) and \( p1 \). There is of course no limitation on the class of patterns that may appear in a local assignment (see Section 3.4.2 for more details concerning local assignments).

The following example has a complex left-hand side that contains 3 levels of AC operators.

[expand] \[ P + (p1+p2)*(p3+p4) \rightarrow P+e1+e2+e3+e4 \]
where \( e4:=(\text{simplify}) \ p2*p4 \)
...

In order to make this rule compilable, we can simply abstract the term \( (p1+p2)*(p3+p4) \) by the variable \( ac1 \) and add the corresponding local assignment to get the following rule:

[expand] \[ P + ac1 \rightarrow P+e1+e2+e3+e4 \]
where (poly) \( (p1+p2)*(p3+p4) \) := () \( ac1 \)
where \( e4:=(\text{simplify}) \ p2*p4 \)
...

But, to benefit from the many-to-one compilation process and get a more efficient program, it is better to keep the most specific term in the left-hand side. Considering the same example, it is possible to abstract the two “small” terms \( (p1+p2) \) and \( (p3+p4) \) (instead of \( (p1+p2)*(p3+p4) \)) by variables \( ac1 \) and \( ac2 \):

[expand] \[ P + ac1*ac2 \rightarrow P+e1+e2+e3+e4 \]
where (poly) \( (p1+p2) \) := () \( ac1 \)
where (poly) \( (p3+p4) \) := () \( ac2 \)
where \( e4:=(\text{simplify}) \ p2*p4 \)
...

4.5.2 Non-linear rules

The ELAN compiler does not support non-left-linear rules (a rule is called non-left-linear if a variable occurs more than once in the left-hand side). The many-to-one matching algorithm only compiles linear rules. The ELAN parser automatically transforms non-left-linear rules into left-linear conditional rules by renaming variables that occurs more than once and adding some conditions to ensure their equality.

For example, the rule \( f(x,x) \Rightarrow g(x) \) is transformed into \( f(x,y) \Rightarrow g(x) \) if \( x==y \).

4.5.3 Built-ins

To be more efficient, builtin data type such as builtInInt and identifier are compiled in a particular way.

The main drawback of this approach is that only completely defined functions can be defined on builtin data type.

Let \( f(\emptyset) \) : (builtInInt) builtInInt be an operator. For each integer \( n \), \( f(n) \) must be reducible to an integer. Suppose defined the rule \( \emptyset \ f(n) \Rightarrow n-1 \) if \( n > 1 \), the term \( f(0) \) can not be reduced to a builtin integer. In this case, behaviour is undefined. This explains why builtInInt are never used directly: it is recommended to use the sort int which is no longer a builtin data type.
4.5.4 Input/Output facilities

Input/Output facilities are not yet fully implemented in the current version of the compiler. However, it is now possible, in the generated program, to enter a query term in the ELAN syntax, just like in the interpreter. In the same way, result terms computed by the generated program are now pretty-printed in the ELAN syntax. Therefore, it is no more necessary to give a fixed ground term in the .1gi file, and .1gi files involving the query keyword can now be compiled. In the current version, the query is parsed dynamically in the generated program by reusing a subpart of the parser integrated to the interpreter.

The generated program can be executed with several switches used to specify the different possible formats of input (query) and output (result) terms:

Input switches

- `-REFInput`: the query term must be in the REF format (see Section 4.6).
- `-noInput`: the query ground term is supposed to be located in the .1gi file, where it replaces the query keyword.

Output switches

- `-REFOutput`: result terms will be given in the REF format (see Section 4.6).
- `-noOutput`: results are computed but not written out. The interest of this switch is rather limited, but it can be useful for people who only want the statistics of the computation, like the number of rewrite steps per seconds.
- `-internalOutput`: result terms are given by using a prefix notation. This is the original syntax used in the first version of the compiler.

The generated program, called by default `a.out`, can be executed with some other switches not necessarily related to Input/Output, like `a.out -quiet` or `a.out -debug`.

At this point, it is still possible to change the content of the start with instruction declared in the .1gi file. The switch `-strategy <strategy name> : <sort name>` will apply the strategy `<strategy name>` on a query term of sort `<sort name>`.

Similarly, the switch `-sort <sort name>` will apply the empty strategy on a query term `<sort name>`. Note that `-strategy` and `-sort` are mutually exclusive.

The list of available sorts (resp. strategy names) can be found in the .ref file. You can also use the graphical user interface (Section 2.2.4) to control the setting of `-strategy` and `-sort`.

4.6 The REF format

The Reduced ELAN Format, REF for short, is a term-like representation of ELAN programs introduced both for the interconnection of software components involved in the ELAN system, and for combining ELAN programs. This format is also of greatest interest for the implementation of reflection techniques in ELAN. A detailed description of its applications can be found in [BKK+98b].

The REF format has been initially developed to reuse the powerful Earley’s parser integrated in the interpreter. By loading with the interpreter an ELAN program possibly in mixfix notation, we can get a REF program which can be viewed as the external representation of the interpreter.
working memory. REF is easier to parse than ELAN since the REF grammar is simply LALR.
The ELAN interpreter generates a REF file prog.ref thanks to the option --export:

```
elan --export prog.ref prog.lgi [prog.spc]
```

The REF program contained in prog.ref can be directly executed by the interpreter as follows:

```
elan --import prog.ref
```

Then, the behavior of the interpreter is exactly the one expected with the command:

```
elan prog.lgi [prog.spc]
```

but without the time-consuming preprocessing and parsing phases.

The core of the ELAN compiler, called REM for “Reduced ELAN Machine” and implemented in Java, only knows the REF syntax. This explains why we need a script elanc to first call the interpreter in order to obtain a REF file, and second to execute REM with this REF file as input.

Then, a compiled C program, say a.out, is generated by REM. This program can be executed with different options like a.out -REFInput, a.out -REFOutput or even better a.out -REFInput -REFOutput:

- The option -REFInput reads a new query in REF syntax to replace the ground query occurring initially in prog.lgi.
- The option -REFOutput writes all result terms computed by a.out in REF syntax, instead of the ELAN syntax used by default.

So, we still have to convert an ELAN query to a REF query, and conversely to retrieve the ELAN syntax of result terms written in REF syntax.

For this purpose, one can use another tool to read a query and to pretty-print result terms of the a.out program, both in ELAN syntax, similarly to the interpreter. This tool is called prettyIO and may be used as follows:

```
prettyIO a.out prog.ref
```

Note that prog.ref is needed to find the ELAN syntax of functions implemented by a.out, and so to encode a query term provided by the user in ELAN syntax, and to decode terms written in REF syntax by a.out -REFInput -REFOutput. The executable prettyIO is implemented as a script calling some basic tools:

- query2ref prog.ref reads repeatedly (from stdin) a sort, and parses a term of this sort written in ELAN syntax. The result (written to stdout) is the corresponding term in REF syntax. By using the option --query as follows:

  query2ref prog.ref --query

  we only parse terms of the sort of the query, as indicated in prog.lgi (this information also occurs in prog.ref).

- ref2result prog.ref reads repeatedly (from stdin) terms encoded in REF syntax and writes (to stdout) the related terms in ELAN syntax, by using the signature occurring in prog.ref.
The script prettyIO is located in the directory bin/ like elan.c, and the compiled C/C++ programs query2ref and ref2result are located in the subdirectory of bin/ related to the current system architecture, like the interpreter elan.

The user should be aware that prettyIO is much less interesting in the current version, since it is now possible to parse a query term and to pretty-print all results by simply using the generated program a.out itself.
Chapter 5

The standard library and the built-ins

This chapter presents the main features of the ELAN standard library. Remember that the path to this library is specified in your environment variable ELANLIB. ELAN can be used without any reference to this library, except for what concerns the use of the built-in objects which are documented in the second part of this chapter.

This library has been designed to be small and as efficient as possible. In particular no AC operators is used. The resulting code is more efficient, at the price of sometimes heavier descriptions.

5.1 The ELAN standard library

The standard library is fully documented in [BCD+98] to which we refer for details. We give below the list of files defined in the standard library. Some of them define built-in sorts and operators and in this case, they also appear in the next section (section 5.2).

5.1.1 common

In this category, the most common abstract data types are defined.

anyIdentifier.eln  identity.eln
anyInteger.eln     int.eln
array.eln          integerConstants.eln
arrayLIN.eln       io.eln
arrayLOG.eln       list.eln
basio.eln          occur.eln
bool.eln           pair.eln
builtinInt.eln     prompt.eln
builtinStdio.eln   replace.eln
builtinString.eln  stdio.eln
builtinSyntacticMatching.eln  string.eln
cmp.eln            strlist.eln
Query.eln          tuple.eln
eq.eln

January 27, 2000   ELAN user manual
5.1.2 ref

All the modules needed to deal with the Reduced ELAN Format.
Meta_Apply.eln ref_modules.eln ref_terms.eln
REF.eln ref_read.eln ref_unify.eln
ref_rules.eln ref_unify_builtin.eln
ref2string.eln ref_sorts.eln ref_write.eln
ref2term.eln ref_strategies.eln refstring2string.eln
ref_ids.eln ref_tables.eln string2refterm.eln

5.1.3 noquote

These modules describe a possible syntax for the user specifications. More complicated syntax
(e.g. mixfix) can also be defined.
  applySubstOn.eln hornClauseSyntax.eln syntacticUnification.eln
atom.eln ident.eln termCommons.eln
atomP.eln identifier.eln sigSyntax.eln
eqSystem.eln substitution.eln

5.1.4 strategy

These modules are used for the definition and evaluation of defined strategies.
  Any.eln boolneg.eln meta.eln strict.eln
  any.eln Meta_apply.eln strapply.eln symbol.eln
  Meta_apply.eln commit.eln strat.eln teststrat.eln
  Meta_strat.eln strcapply.eln loops.eln strconc.eln

5.2 The built-Ins

There are two types of built-in symbols:

- The built-in constructors, which are non-reducible symbols, built into the ELAN
  system. Terms built over them are either constructed, or interpreted by the ELAN system,
  e.g. the symbol Error defined in the module /elanlib/common/builtinStdio.eln as follows:


\[
\text{Error}(\emptyset) : \text{(builtinInt)} \text{ Pid code 122;}
\]

which is used to report various errors of the I/O sub-system written in C++.

- The built-in functions, which represent symbols with pre-defined semantics, in
general, not easily specifiable in ELAN itself. The symbol open in the module /elan-
lib/common/builtinStdio.eln defined as follows:

\[
\text{open}(\emptyset,\emptyset) : \text{(string string)} \text{ Pid code 116;}
\]

can be taken as an example. In this case, the declaration (profile) of the symbol open
represents an interface between the ELAN specification and its built-in semantics. In this
case, the code “116” identifies its semantics.
If there is an overloaded built-in symbol whose exact semantics depends on its profile, a procedure implementing the built-in symbol should know, which instance of the built-in symbol is executed. The overloaded symbol \texttt{read} is an example of this problem:

\begin{verbatim}
read(0) : (Pid) X pri 200 code -120;
\end{verbatim}

When the built-in symbol \texttt{read} is used for several sorts \texttt{X}, there exist several built-in symbols \texttt{read} in the user's specification. All of them are linked with the same semantic function (with the code 120) parameterized by the concrete profile of \texttt{read}. Thus, the sort of a \texttt{read} term is known in the semantic procedure due to the profile of actually applied symbol \texttt{read}. The difference between two kinds of \textit{built-in function} symbols is expressed by the sign of code numbers.

5.2.1 Booleans

At the beginning there is nothing, so \textsc{ELAN} provides the true and false values and introduces the \texttt{bool} module. These two values are built-in and are deeply connected to the implementation of conditions in rewrite rules.

/elanelib/common/bool.eln

<table>
<thead>
<tr>
<th>Builtins</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{false}</td>
<td>0</td>
<td>() bool</td>
<td>The logical \texttt{false}.</td>
</tr>
<tr>
<td>\texttt{true}</td>
<td>1</td>
<td>() bool</td>
<td>The logical \texttt{true}.</td>
</tr>
<tr>
<td>\texttt{0 and @}</td>
<td>21</td>
<td>(bool bool) bool</td>
<td>The logical conjunction.</td>
</tr>
<tr>
<td>\texttt{0 or @}</td>
<td>22</td>
<td>(bool bool) bool</td>
<td>The logical disjunction.</td>
</tr>
<tr>
<td>\texttt{not(0)}</td>
<td>24</td>
<td>(bool) bool</td>
<td>The logical negation.</td>
</tr>
<tr>
<td>\texttt{0 == 0}</td>
<td>8</td>
<td>(bool bool) bool</td>
<td>The logical equality test.</td>
</tr>
<tr>
<td>\texttt{0 !@}</td>
<td>9</td>
<td>(bool bool) bool</td>
<td>The logical inequality test.</td>
</tr>
<tr>
<td>\texttt{0 &lt; 0}</td>
<td>10</td>
<td>(bool bool) bool</td>
<td>A logical test assuming that \texttt{false}&lt;\texttt{true}.</td>
</tr>
<tr>
<td>\texttt{0 &lt;= 0}</td>
<td>11</td>
<td>(bool bool) bool</td>
<td>A logical test assuming that \texttt{false}&lt;\texttt{true}.</td>
</tr>
<tr>
<td>\texttt{0 &gt; 0}</td>
<td>12</td>
<td>(bool bool) bool</td>
<td>A logical test assuming that \texttt{false}&gt;\texttt{true}.</td>
</tr>
<tr>
<td>\texttt{0 &gt;= 0}</td>
<td>13</td>
<td>(bool bool) bool</td>
<td>A logical test assuming that \texttt{false}&gt;\texttt{true}.</td>
</tr>
</tbody>
</table>

/elanelib/common/cmp.eln

To enrich the booleans, \textit{polymorphic} equality, disequality and inequalities are defined and are also built-in:

<table>
<thead>
<tr>
<th>Builtins</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ == @</td>
<td>18</td>
<td>(X X) bool</td>
<td>Tests the equality between two elements of the same sort \texttt{X}.</td>
</tr>
<tr>
<td>@ !@</td>
<td>19</td>
<td>(X X) bool</td>
<td>Tests the inequality between two elements of the same sort \texttt{X}.</td>
</tr>
</tbody>
</table>

5.2.2 Numbers

Numbers can of course be created "by hand", but we choose in \textsc{ELAN} to provide built-in integers.
<table>
<thead>
<tr>
<th>Buitlins</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + 0</td>
<td>3</td>
<td>(builtinInt builtinInt) builtinInt</td>
<td>The addition on integers.</td>
</tr>
<tr>
<td>0 - 0</td>
<td>4</td>
<td>(builtinInt builtinInt) builtinInt</td>
<td>The soustraction on integers.</td>
</tr>
<tr>
<td>0 * 0</td>
<td>5</td>
<td>(builtinInt builtinInt) builtinInt</td>
<td>The multiplication on integers.</td>
</tr>
<tr>
<td>0 / 0</td>
<td>6</td>
<td>(builtinInt builtinInt) builtinInt</td>
<td>The division on integers.</td>
</tr>
<tr>
<td>0 % 0</td>
<td>27</td>
<td>(builtinInt builtinInt) builtinInt</td>
<td>The modulo operation.</td>
</tr>
<tr>
<td>0 &amp; 0</td>
<td>28</td>
<td>(builtinInt builtinInt) builtinInt</td>
<td>a &amp; b makes the logical conjunction between the binary representations of a and b and returns the corresponding integer.</td>
</tr>
<tr>
<td>0</td>
<td>29</td>
<td>(builtinInt builtinInt) builtinInt</td>
<td>a</td>
</tr>
<tr>
<td>- 0</td>
<td>20</td>
<td>(builtinInt) builtinInt</td>
<td>The opposite of any integer.</td>
</tr>
<tr>
<td>0 == 0</td>
<td>8</td>
<td>(builtinInt builtinInt) bool</td>
<td>The equality test.</td>
</tr>
<tr>
<td>0 != 0</td>
<td>9</td>
<td>(builtinInt builtinInt) bool</td>
<td>The inequality test.</td>
</tr>
<tr>
<td>0 &lt; 0</td>
<td>10</td>
<td>(builtinInt builtinInt) bool</td>
<td>The strict ordering test between integers.</td>
</tr>
<tr>
<td>0 &lt;= 0</td>
<td>11</td>
<td>(builtinInt builtinInt) bool</td>
<td>The ordering test between integers.</td>
</tr>
<tr>
<td>0 &gt; 0</td>
<td>12</td>
<td>(builtinInt builtinInt) bool</td>
<td>The strict ordering test between integers.</td>
</tr>
<tr>
<td>0 &gt;= 0</td>
<td>13</td>
<td>(builtinInt builtinInt) bool</td>
<td>The ordering test between integers.</td>
</tr>
<tr>
<td>eq_builtin</td>
<td>8</td>
<td>(builtinInt builtinInt) bool</td>
<td>The equality test.</td>
</tr>
<tr>
<td>neq_builtin</td>
<td>9</td>
<td>(builtinInt builtinInt) bool</td>
<td>The inequality test.</td>
</tr>
<tr>
<td>less_builtin</td>
<td>10</td>
<td>(builtinInt builtinInt) bool</td>
<td>The strict ordering test between integers.</td>
</tr>
<tr>
<td>lesseq_builtin</td>
<td>11</td>
<td>(builtinInt builtinInt) bool</td>
<td>The ordering test between integers.</td>
</tr>
</tbody>
</table>
| Builtin | Code | Signature                        | Comments |}
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>greater builtin Int(0,0)</td>
<td>12</td>
<td>(builtinInt builtinInt) bool</td>
<td>The ordering test between integers.</td>
</tr>
<tr>
<td>greatereq builtinInt(0,0)</td>
<td>13</td>
<td>(builtinInt builtinInt) bool</td>
<td>The ordering test between integers.</td>
</tr>
<tr>
<td>btoi(0)</td>
<td>25</td>
<td>(bool) builtinInt</td>
<td>The conversion from a boolean to an integer (false becomes 0 and true becomes 1).</td>
</tr>
<tr>
<td>itob(0)</td>
<td>26</td>
<td>(builtinInt) bool</td>
<td>The conversion from any integer to a boolean (0 becomes false and any other true).</td>
</tr>
</tbody>
</table>

5.2.3 Identifiers

/elantl/lib/noquote/ident.eln

| Builtin | Code | Signature          | Comments |}
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@ == @</td>
<td>14</td>
<td>(ident ident) bool</td>
<td>Tests the equality between two identifiers.</td>
</tr>
<tr>
<td>@ ! = @</td>
<td>15</td>
<td>(ident ident) bool</td>
<td>Tests the inequality between two identifiers.</td>
</tr>
</tbody>
</table>

/elantl/lib/common/builtinString.eln

| Builtin | Code | Signature                  | Comments |}
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen(0)</td>
<td>150</td>
<td>(string) builtinInt</td>
<td>Computes the length of a string.</td>
</tr>
<tr>
<td>strcat(0,0)</td>
<td>151</td>
<td>(string string) string</td>
<td>Concatenates two strings.</td>
</tr>
<tr>
<td>@[]</td>
<td>152</td>
<td>(string builtinInt)</td>
<td>$s[i]$ selects the $i$th character of the string $s$.</td>
</tr>
<tr>
<td>@[@&lt;=@]</td>
<td>153</td>
<td>(string builtinInt builtinInt) string</td>
<td>$s[i&lt;=c]$ replaces the $i$th character of the string $s$ by the character $c$.</td>
</tr>
<tr>
<td>substr(0,0,0)</td>
<td>154</td>
<td>(string builtinInt builtinInt) string</td>
<td>substr($s,i,n$) extracts from the string $s$ a substring of length $n$ from the $i$th position.</td>
</tr>
<tr>
<td>strspn(0,0)</td>
<td>156</td>
<td>(string string) builtinInt</td>
<td>strspn($s1,s2$) returns the length of the first part of $s1$ entirely constituted by characters of $s2$.</td>
</tr>
<tr>
<td>Builtin</td>
<td>Code</td>
<td>Signature</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>-------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>strcmp(0,0)</td>
<td>157</td>
<td>(string string) builtin</td>
<td>strcmp(s1,s2) compares two strings character by character and returns an integer (a negative one if s1 is less than s2, 0 if they are equal and otherwise a positive integer).</td>
</tr>
<tr>
<td>string(0)</td>
<td>158</td>
<td>(builtinInt) string</td>
<td>Conversion from an integer (that is, the associated character) to a string.</td>
</tr>
<tr>
<td>ident2string(0)</td>
<td>177</td>
<td>(ident) string</td>
<td>Conversion from an identifier to a string.</td>
</tr>
</tbody>
</table>

### 5.2.4 Elementary term computations

Since it is of primarily use in symbolic computation on terms (and remember that everything in ELAN is a term except the built-ins), the occurrence relation and the replacement operation are provided as built-ins.

/elanlib/common/occur.eln

The module occur[X,Y] is parameterized by two sorts X,Y that must be non built-in.

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>occurs @ in @</td>
<td>17</td>
<td>(X Y) bool</td>
<td>occurs s in t checks if s occurs in t.</td>
</tr>
</tbody>
</table>

/elanlib/common/replace.eln

The module replace[X,Y] is parameterized by two sorts X,Y that must be non built-in.

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace @ by @ in @</td>
<td>16</td>
<td>(X X Y) Y</td>
<td>replace s by u in t replaces all occurrences of s by the u in t.</td>
</tr>
</tbody>
</table>

/elanlib/common/builtinSyntacticMatching.eln

Matching and unification operations are provided as built-ins.

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>builtinSyntacticMatching(0,0,0,0,0)</td>
<td>191</td>
<td>(X X Y Y) Y</td>
<td>builtinSyntacticMatching(t1,t2,vars,failure) tries to match terms t1 to t2 (i.e. t1 &lt;&lt; t2), where vars:Y represents list of variables occurred in t1:X and t2:X and failure:Y is a value returned in a non-successful case. Otherwise, a matching substitution is returned in the form of a list of terms associated to the list of variables vars. The sort Y is supposed to be list[X]. It works only for empty theories.</td>
</tr>
</tbody>
</table>
5.2 The built-ins

<table>
<thead>
<tr>
<th>Builtins</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>builtinSyntactic Unification(0,0,0,0)</td>
<td>175</td>
<td>(X Y Y Y) Y</td>
<td>the same as the previous built-in function, except that a most general unifier is searched.</td>
</tr>
</tbody>
</table>

5.2.5 Input/Output

/elanlib/common/Query.eln

The module Query[I,O,P] defines the parameterized sort Query[I,O,P] and the sort ide (embedding the sort ident of identifiers). All symbols defined in this module are built-in constructors w.r.t. the classification mentioned before.

<table>
<thead>
<tr>
<th>Builtins</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>quit</td>
<td>90</td>
<td>() Query[I,O,P]</td>
<td>quit just quit the command interpreter</td>
</tr>
<tr>
<td>help</td>
<td>112</td>
<td>() Query[I,O,P]</td>
<td>help prints the help menu (in the file elanlib/help.txt)</td>
</tr>
<tr>
<td>load @</td>
<td>91</td>
<td>(ide) Query[I,O,P]</td>
<td>load f loads the file f.eln and extends the current specification</td>
</tr>
<tr>
<td>batch @</td>
<td>107</td>
<td>(ide) Query[I,O,P]</td>
<td>batch f runs commands of the batch file f.eln</td>
</tr>
<tr>
<td>run @</td>
<td>94</td>
<td>(I) Query[I,O,P]</td>
<td>run q runs the current specification with the query q</td>
</tr>
<tr>
<td>sorts @ @ @</td>
<td>92</td>
<td>(ide ide ide) [I,O,P]</td>
<td>sorts q r s redefines sorts of query, results and print-outs</td>
</tr>
<tr>
<td>startwith (O) @</td>
<td>93</td>
<td>(ident O) Query[I,O,P]</td>
<td>startwith (s)t redefines the starting term (s)t of the .lgi file, where s is a strategy and t is a pattern of the input query</td>
</tr>
<tr>
<td>checkwith @</td>
<td>102</td>
<td>(bool) Query[I,O,P]</td>
<td>checkwith b redefines the checking term of the .lgi file, where b is a pattern of a boolean condition tested before any input query</td>
</tr>
<tr>
<td>printwith @</td>
<td>98</td>
<td>(P) Query[I,O,P]</td>
<td>printwith t redefines the printing term used to pretty-print any result</td>
</tr>
<tr>
<td>q5</td>
<td>100</td>
<td>Query[I,O,P]</td>
<td>dump the stack of queries</td>
</tr>
<tr>
<td>rs</td>
<td>101</td>
<td>Query[I,O,P]</td>
<td>dump the stack of results</td>
</tr>
<tr>
<td>stat</td>
<td>96</td>
<td>Query[I,O,P]</td>
<td>print statistics</td>
</tr>
<tr>
<td>dump</td>
<td>95</td>
<td>Query[I,O,P]</td>
<td>dump all rules</td>
</tr>
<tr>
<td>dump @</td>
<td>108</td>
<td>(ident) Query[I,O,P]</td>
<td>dump(1) dumps rules or strategies with the label/name 1</td>
</tr>
<tr>
<td>dump @</td>
<td>109</td>
<td>(builtinInt) Query[I,O,P]</td>
<td>dump(s) gives all information about the symbol s given by its code</td>
</tr>
<tr>
<td>display @</td>
<td>103</td>
<td>(builtinInt) Query[I,O,P]</td>
<td>display m changes the printing mode of symbols: display 1 shows terms in internal form, while display 0 uses the traditional representation</td>
</tr>
<tr>
<td>Builtins</td>
<td>Code</td>
<td>Signature</td>
<td>Comments</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>-----------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>trace @</td>
<td>97</td>
<td>(builtinInt) Query[I,0,P]</td>
<td>trace n changes the level of debugging to level n</td>
</tr>
<tr>
<td>break @</td>
<td>104</td>
<td>(ident) Query[I,0,P]</td>
<td>break f sets a break-point on rules or strategies with the label/name f</td>
</tr>
<tr>
<td>break @</td>
<td>110</td>
<td>(builtinInt) Query[I,0,P]</td>
<td>break s sets a break-point on the symbol s given by its code</td>
</tr>
<tr>
<td>unbreak @</td>
<td>105</td>
<td>(ident) Query[I,0,P]</td>
<td>unbreak f deletes the break-point on rules or strategies with the label/name f</td>
</tr>
<tr>
<td>unbreak @</td>
<td>111</td>
<td>(builtinInt) Query[I,0,P]</td>
<td>unbreak s deletes the break-point on the symbol s</td>
</tr>
<tr>
<td>breaks</td>
<td>106</td>
<td>() Query[I,0,P]</td>
<td>lists all set break points</td>
</tr>
</tbody>
</table>

/elannlib/common/io.eln

<table>
<thead>
<tr>
<th>Builtins</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(@,@)</td>
<td>-119</td>
<td>(Pid X) X</td>
<td>Writes on output Pid the argument of sort X.</td>
</tr>
<tr>
<td>read(@)</td>
<td>-120</td>
<td>(Pid) X</td>
<td>Reads from input Pid and return an element of sort X.</td>
</tr>
<tr>
<td>Error(())</td>
<td>123</td>
<td>(builtinInt) X</td>
<td>Reports various errors of the I/O sub-system written in C++.</td>
</tr>
</tbody>
</table>

/elannlib/common/builtinStdio.eln

<table>
<thead>
<tr>
<th>Builtins</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>getc(0)</td>
<td>113</td>
<td>(Pid) builtinInt</td>
<td>Gets the next character from an input file or pipe.</td>
</tr>
<tr>
<td>putc(0)</td>
<td>114</td>
<td>(Pid builtinInt) builtinInt</td>
<td>Puts a character to an output file or pipe.</td>
</tr>
<tr>
<td>create(0)</td>
<td>115</td>
<td>(string) Pid</td>
<td>Creates a process with a given name. The created process is linked with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the parent process via two blocking pipes.</td>
</tr>
<tr>
<td>create,noblock(0)</td>
<td>117</td>
<td>(string) Pid</td>
<td>Creates a process with a given name such that two communication pipes are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>in a non blocking mode.</td>
</tr>
<tr>
<td>open(0,0)</td>
<td>116</td>
<td>(string string) Pid</td>
<td>Opens a file for reading or writing. The first argument is the name of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the file, and the second argument is either &quot;w&quot;, or &quot;r&quot;.</td>
</tr>
<tr>
<td>close(0)</td>
<td>118</td>
<td>(Pid) Pid</td>
<td>Closes a file or kill a process denoted by the pipe number of sort Pid.</td>
</tr>
</tbody>
</table>

January 27, 2000

ELAN user manual
### 5.2.6 Strategies

/elanlib/strategy/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.

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@</code></td>
<td>131</td>
<td><code>(Strategy[X])</code></td>
<td>Embeds a basic strategy in a list of basic strategies.</td>
</tr>
<tr>
<td><code>@ , @</code></td>
<td>132</td>
<td><code>(Strategy[X])</code></td>
<td>Concatenates a basic strategy to a list of basic strategies.</td>
</tr>
<tr>
<td><code>@</code></td>
<td>133</td>
<td><code>(string) Labels[X]</code></td>
<td>Embeds a string in a list of rule labels.</td>
</tr>
<tr>
<td><code>@ , @</code></td>
<td>134</td>
<td><code>(string Labels[X])</code></td>
<td>Concatenates a string to a list of rule labels.</td>
</tr>
<tr>
<td><code>dc(@)</code></td>
<td>135</td>
<td><code>(Labels[X]) Strateg[X]</code></td>
<td><code>dc(1)</code> is a basic strategy that chooses one label in the list of labels 1 and returns all results of the corresponding rule.</td>
</tr>
<tr>
<td><code>dk(@)</code></td>
<td>136</td>
<td><code>(Labels[X]) Strateg[X]</code></td>
<td><code>dk(1)</code> is a basic strategy that chooses successively all labels in the list of labels 1 and returns all results of the corresponding rules.</td>
</tr>
<tr>
<td><code>dc(@)</code></td>
<td>137</td>
<td><code>(Strategies[X]) Strateg[X]</code></td>
<td><code>dc(1s)</code> is a basic strategy that chooses one basic strategy in the list 1s and returns all its results.</td>
</tr>
<tr>
<td><code>dk(@)</code></td>
<td>138</td>
<td><code>(Strategies[X]) Strateg[X]</code></td>
<td><code>dk(1s)</code> is a basic strategy that chooses successively all basic strategies in the list 1s and returns all their results.</td>
</tr>
<tr>
<td><code>repeat*(@)</code></td>
<td>139</td>
<td><code>(Strategy[X]) Strateg[X]</code></td>
<td><code>repeat*(s)</code> is a basic strategy that repeats the application of the basic strategy <code>s</code> until it fails.</td>
</tr>
<tr>
<td><code>iterate*(@)</code></td>
<td>140</td>
<td><code>(Strategy[X]) Strateg[X]</code></td>
<td><code>iterate*(s)</code> is a basic strategy that repeats the application of the basic strategy <code>s</code> until it fails and returns intermediate results.</td>
</tr>
<tr>
<td><code>@</code></td>
<td>141</td>
<td><code>(Strategy[X]) Strategy[X]</code></td>
<td>Embeds the previous constructions into the sort <code>Strategy[X]</code>.</td>
</tr>
<tr>
<td><code>@ ; @</code></td>
<td>142</td>
<td><code>(Strategy[X] Strategy[X])</code></td>
<td>Concatenates any previous construction to a strategy.</td>
</tr>
<tr>
<td><code>id</code></td>
<td>143</td>
<td><code>()</code> Strateg[X]</td>
<td><code>id</code> is a basic strategy.</td>
</tr>
<tr>
<td><code>call @ : X</code></td>
<td>144</td>
<td><code>(string) Strateg[X]</code></td>
<td><code>call s:X</code> represents an application of a basic strategy named <code>s</code> of sort <code>X</code>.</td>
</tr>
</tbody>
</table>
The module `Meta_apply[X]` parameterized by a sort X can only by used in interpreted mode (cf. the module `Meta_capply[X]` for the compiled mode).

The construction where \( y := \text{\(\text{META}\) meta\_apply}(s, t) \) exported from this module is semantically equivalent to where \( y := (s)t \) where s is a basic strategy, i.e. it gives all results of application s on t. There exists also a slightly modified construction where \( y := \text{\(\text{META}\) meta\_apply}(s, t, n) \) giving only the n-th solution of the application of \((s)t\), if it exists.

The ELAN built-in 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). The built-in strategy `META` is applicable only to terms of the form meta\_apply(s, t) or meta\_apply(s, t, n).

The module `Meta_apply.eln` exports several variants of the symbol `set\_of(s, t, m, n)` returning a list of m results of the application \((s)t\) from the n-th result. The symbols `set\_of(s, t)`, `set\_of(s, t, m)` and `set\_of(s, t, m, n)` are implemented using a result-collecting built-in symbol `meta\_apply(s, t, m, n)`: (Strategy[X] list[X] builtinInt builtinInt) list[X] defined in this module.

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>meta_apply(0,0)</code></td>
<td>-129</td>
<td>(Strategy[X] X) X</td>
<td>meta_apply(s, t) returns all results of application of the basic strategy s on the term t.</td>
</tr>
<tr>
<td><code>meta_apply(0,0)</code></td>
<td>-130</td>
<td>(Strategy[X] X</td>
<td>meta_apply(s, t, n) returns the n-th result of application of the basic strategy s on the term t, if it exists.</td>
</tr>
<tr>
<td><code>meta_apply(0,0)</code></td>
<td>-128</td>
<td>(Strategy[X] list[X]</td>
<td>The local built-in symbol meta_apply(s, t, nil,m,n) returns n results starting from the m-th of application of the basic strategy s on the term t, if they exist.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list[X] builtinInt</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>builtinInt) X</td>
<td></td>
</tr>
</tbody>
</table>

The module `Meta_capply[X]` parameterized by a sort X can only by used in compiled mode. It represents a simplified form of the module `Meta_apply[X]`, where a construction of a strategy of sort `Strategy[X]` is restricted to a reference of an existing strategy (i.e. a strategy already occurring in the compiled program). Thus, in a `meta_apply` call, only a name (i.e. a string) of a referenced strategy is used instead of a strategy term of sort `Strategy[X].`

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>meta_apply(0,0)</code></td>
<td>-129</td>
<td>(string X) X</td>
<td>meta_apply(s, t) returns all results of application of the strategy named s on the term t.</td>
</tr>
<tr>
<td><code>meta_apply(0,0)</code></td>
<td>-130</td>
<td>(string X builtinInt) X</td>
<td>meta_apply(s, t, n) returns the n-th result of application of the strategy named s on the term t, if it exists.</td>
</tr>
</tbody>
</table>
The module \texttt{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 \texttt{strsig.eln} contains strategy constructors used both for type-preserving and type-changing strategies.

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ @</td>
<td>-180</td>
<td>((X\rightarrow Y)\ X\ Y)</td>
<td>[S] takes the strategy (S) and applies it to the term (t).</td>
</tr>
<tr>
<td>dk(@)</td>
<td>-182</td>
<td>((\text{Strlist}[X,Y])\ X\rightarrow Y)</td>
<td>dk(1) is the defined strategy that chooses successively all strategies in the list (1) and returns all their results.</td>
</tr>
<tr>
<td>dc(@)</td>
<td>-181</td>
<td>((\text{Strlist}[X,Y])\ X\rightarrow Y)</td>
<td>dc(1) is the defined strategy that chooses one strategy in the list (1) and returns all its results.</td>
</tr>
<tr>
<td>dc one(@)</td>
<td>-190</td>
<td>((\text{Strlist}[X,Y])\ X\rightarrow Y)</td>
<td>dc one (1) is the defined strategy that chooses one strategy in the list (1) and returns one result.</td>
</tr>
<tr>
<td>fail</td>
<td>-183</td>
<td>() (\ X\rightarrow Y)</td>
<td>The defined strategy that always gives an empty set of results.</td>
</tr>
<tr>
<td>@, @</td>
<td>-185</td>
<td>((X\rightarrow Y)\ \text{Strlist}[X,Y])</td>
<td>Concatenates a defined strategy to a list of strategies.</td>
</tr>
<tr>
<td>Epsilon</td>
<td>-189</td>
<td>() \text{Strlist}[X,Y]</td>
<td>The empty list of strategies.</td>
</tr>
<tr>
<td>if @ then</td>
<td>-184</td>
<td>((\text{bool} \ X\rightarrow Y)\ \ X\rightarrow Y)</td>
<td>if (b) then (s1) else (s2) fi is the defined strategy that chooses either (s1) if (b) is true otherwise (s2).</td>
</tr>
<tr>
<td>@ else @ fi</td>
<td></td>
<td>(\ X\rightarrow Y)</td>
<td>Embeds an assignment in a list of assignments.</td>
</tr>
<tr>
<td>0</td>
<td>146</td>
<td>(\text{Assignment}) \text{AssignmentList}</td>
<td>A condition if an IfApplication.</td>
</tr>
<tr>
<td>0 , 0</td>
<td>147</td>
<td>(\text{Assignment}) \text{Assignment} \text{AssignmentList}</td>
<td>Embeds an IfApplication in a list of assignments.</td>
</tr>
<tr>
<td>if @</td>
<td>145</td>
<td>((\text{bool})\ \text{IfApplication})</td>
<td>(\ X\rightarrow Y)</td>
</tr>
<tr>
<td>0</td>
<td>146</td>
<td>(\text{IfApplication}) \text{AssignmentList}</td>
<td>Add an IfApplication to the end of a list of assignments.</td>
</tr>
<tr>
<td>0 , 0</td>
<td>147</td>
<td>(\text{AssignmentList}) \text{IfApplication} \text{AssignmentList}</td>
<td>Add an IfApplication to the end of a list of assignments.</td>
</tr>
</tbody>
</table>

The module \texttt{strconc[X,Y,Z]} parameterized by three sorts \(X,Y,Z\) introduces a concatenation symbol \(;\) of two strategies of sorts \(X\rightarrow Y\) and \(Y\rightarrow Z\).

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ ; @</td>
<td>-188</td>
<td>((X\rightarrow Y)\ (Y\rightarrow Z)) (X\rightarrow Z)</td>
<td>(s1) ; (s2) concatenates the strategies (s1:\ X\rightarrow Y) and (s2:\ Y\rightarrow Z) to get a strategy of sort (X\rightarrow Z).</td>
</tr>
</tbody>
</table>
/elanlib/strategy/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.

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>-186</td>
<td>() &lt;X-&gt;X&gt;</td>
<td>Identity strategy.</td>
</tr>
<tr>
<td>if @ then @ fi</td>
<td>-187</td>
<td>(&lt;X-&gt;X&gt; &lt;X-&gt;X&gt; &lt;X-&gt;X&gt;) &lt;X-&gt;X&gt;</td>
<td>Orelse strategy.</td>
</tr>
</tbody>
</table>

5.2.7 REF Format

/elanlib/ref/Meta_Apply.eln

The module ref/Meta_Apply.eln provides several variants of the function meta_apply, which apply a strategy on a term in an environment defined by a program. 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 representing terms.

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Code</th>
<th>Signature</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta_apply(0,0,0,0,0)</td>
<td>170</td>
<td>(string string string builtinInt) string</td>
<td>meta_apply(s,t,file,spec,n) gives the n-th result of the application of the strategy s on the term t w.r.t. the program stored in files named file.lgi and spec.spc.</td>
</tr>
<tr>
<td>meta_apply(0,0,0,0,0)</td>
<td>171</td>
<td>(string list[string] string string builtinInt) string</td>
<td>meta_apply(s,t,nil,file,spec,n) gives the n-th result of the application of the strategy s on the term t w.r.t. the program in files named file.lgi and spec.spc.</td>
</tr>
<tr>
<td>meta_apply(0,0,0,0)</td>
<td>172</td>
<td>(string string string builtinInt) string</td>
<td>meta_apply(s,t,file,n) gives the n-th result of the application of the strategy s on the term t w.r.t. the program in the REF-format stored in file named file.ref.</td>
</tr>
<tr>
<td>meta_apply(0,0,0,0,0)</td>
<td>173</td>
<td>(string list[string] string string builtinInt) string</td>
<td>meta_apply(s,t,nil,file,n) gives the n-th result of the application of the strategy s on the term t w.r.t. the program in the REF-format stored in file named file.ref.</td>
</tr>
</tbody>
</table>

/elanlib/ref/ref2string.eln

The module ref/ref2string.eln 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.
5.3 Calling external programs

ELAN allows using external programs like for instance the UNIF system [Rin97]. Let us describe how it works.

The external program is assumed to take as input a term and to return to ELAN a set of terms. One could imagine the input term as an input constraint and the returned terms as the solved forms computed by the program. The program is called as a sub-process of the ELAN session and it is linked to it via two UNIX pipes which are respectively attached to the standard input and output of the process (i.e. stdin, stdout).

Since it is useful to reuse a given process as much as possible, the description of the called process gives the number of terms that can be sent to the process before killing it and replacing it by another one.

Data (i.e. terms) are sent in and out under a textual format. This format is determined by the signature of the module in which the process call is performed.

The process call is realized through the use of the strategies dcall and dkccall (that stand for dont care call and dont know call), with three arguments:
- the first is the name of the process,
- the second is the number of terms these strategies will be ready to receive,
- the third is the number of terms these strategies will return.

A context can build a Call module to send terms to an external process, using the Call module's signature.

The Call module can have contexts associated with it, to allow for the sending of terms by a Call module to other contexts.

The signature of the module in which the process call is performed is determined by the context of the module.
the second is the number of terms that should be transmitted to the process before killing it,
and the third is the (unique) sort of initial and result terms.
The syntax is therefore the following:

\[
\langle \text{strategy} \rangle ::= \text{dkcall}( \langle \text{identifier} \rangle, \langle \text{number} \rangle, \langle \text{sort name} \rangle ) \ \\
\ | \quad \text{dcall}( \langle \text{identifier} \rangle, \langle \text{number} \rangle, \langle \text{sort name} \rangle )
\]

The difference between the two calls is that the first one returns all the computed results
while the second one returns only the first result, even if several are computed.
In order to be able to communicate with ELAN, the process which is called should obey to
the following syntactic restrictions.
The output should have the following syntax:

\[
\langle \text{processus output} \rangle ::= \{ \# \}^+ \langle \text{next solution} \rangle
\]

\[
\langle \text{next solution} \rangle ::= \langle \text{number} \rangle \{ \ast \}^+ \langle \text{term} \rangle \{ \# \}^+ \langle \text{next solution} \rangle
\ \\
\ | \quad \text{END} \{ \# \}^+
\]

where \langle \text{term} \rangle is one result.
In the same way, the process should accept the syntax of the initial term, described as follows:

\[
\langle \text{processus input} \rangle ::= \langle \text{term} \rangle \langle \text{next solution demand} \rangle
\]

\[
\langle \text{next solution demand} \rangle ::= \ . \langle \text{next solution demand} \rangle
\ \\
\ | \quad \ ;
\]

where \langle \text{term} \rangle is the term passed to the built-in process.
The way the communication is organised between ELAN and the built-in process is as follows.
ELAN first sends the initial term to the process and every time it needs another result, it sends
the symbol “,” and waits for the answer of the process on its standard output. The answer
should be in the syntax corresponding to the non-terminal \langle \text{next solution} \rangle. If ELAN does not
need anymore solution, it sends the symbol “;” which makes the process stop, or it sends an
“end of file” if the limit of the process use is reached.

\[\Rightarrow\] Since the synchronization between ELAN and the called process in made through pipes, the two
process should reinitialize their buffers after each large output. As a consequence the called process
should call the flush(stdout) procedure after returning each solution.

Example 5.1 Assume that one would like to use the shell command “sort” for sorting a list of
“elements”. The first step is to create a file containing the shell script calling the sort command
and that follows the syntactic conventions required by ELAN as just described above. Let us call
this file elansort:

```
#!/bin/sh
cat /dev/null > /tmp/inputFile
read r
while (test "\$r" != "") do
  echo \$r >> /tmp/inputFile
  read r
done
echo "##0**"
```

January 27, 2000
sort < /tmp/inputFile
echo "##END##
read r
while (test "$r" != ";") do
    read r
done
/bin/rm /tmp/inputFile

Then, we construct the syntax of “elements” by using a preprocessor instruction involving a list (Vars) of identifiers taken from a specification file:

import global List;
    local Vars anyIdentifier list[identifier] ;
end

operators global
    FOR EACH SS:identifier SUCH THAT SS:=(listExtract) elem(Vars) :{
        SS     : elem ;
    }
end

Lists are defined as sequences of elements, without any separator:

sort list; end

operators global
    @     : ( elem ) list;
    @ @   : ( list list ) list  assocLeft;
end

Since elansort could only be used for sorting elements given successively on each line of a file, we have to convert lists in the right syntax before calling the external program:

module listSort[Vars] 1
import global Elen[Vars] ;
    local List eSort[Vars] ;
end

operators local 2
    @ "\^\'"     : ( list ) list ;
    @ newline   : ( list ) list alias @ "\^\'" ;
    convert(@)   : ( list ) list ;
end 3

stratop global
stratadd1 : < list -> list > bs ;
exitSort : < list -> list > bs ;
end 4

rules for list
    P, Q : list ;
    a: elem ;
global 5
    [addnewline] P => P newline end
    []    convert(P a) => convert(P) Q
    where Q:={} a newline end
    []    convert(a) => a newline end

January 27, 2000
The main strategy `extSort` applies first the conversion rule and then applies the rule calling the external program through the `callSort` strategy defined in the following ELAN module:

```elang
module callSort [Vars]
import global Elem[Vars];
local Vars anyIdentifier list [identifier]
List ;
end
operators local
  FOR EACH SS:identifier SUCH THAT SS:= (listExtract ) elem (Vars) :{
    SS : elem alias SS :
    @ : (elem ) list alias @ ;
    @ @ : (list list) list assocLeft alias @ @ ;
  }
strategy global
callSort : < list -> list > bs ;
end

strategies for list
implicit
  [] callSort => dcall (elansort1 , list ) end
end

This module can be considered as an encapsulation of the `elansort` program. According to the alias declarations, results are written in the initial syntax, as sequences of elements separated by spaces.
Chapter 6

Contributed works

6.1 Description of the ELAN contributions

Many computational processes in Automated Deduction and Constraint Programming can be expressed as instances of a general approach that consists of applying transformation rules on formulas with some strategy, until reaching specific normal forms. Such processes are naturally modelled in ELAN, and can be classified according to the area of interest, namely programming, proving and solving.

Programming. One of the first application was to prototype the fundamental mechanisms of logic and functional programming languages like first-order resolution and $\lambda$-calculus. The general framework of Constraint Logic Programming can be easily designed in the ELAN framework [KR98], since its operational semantics is clearly formalised as rewrite rules, although the application strategy is often defined in an informal way. Some implementations [Bor95] related to a calculus of explicit substitutions (the first-order rewrite system $\lambda\eta$ that mimics $\lambda$-calculus) open the way of implementing higher-order logic programming languages via a first-order setting. Another calculus of explicit substitutions based on the $\pi$-calculus is used to provide a formal specification of Input/Output for ELAN [Vir96].

Proving: ELAN was used in order to implement a predicate prover based on the rules proposed by J.-R. Abrial, and implemented in the B-tools [CK97]. We developed also a propositional sequent calculus, completion procedures for rewrite systems [KM95], sufficient conditions for the termination problem [GG97]. A library for automata construction and manipulation has been designed. Approximation automata are used to check conditions for reachability, sufficient completeness, absence of conflicts in systems described by non-conditional rewrite rules [Gen98b, Gen98a].

Checking. A very special case of proof is just exhaustive check of all the possibilities. Using in particular “dont know” strategies, it was efficiently rediscovered by exhaustive search that the well known Needam and Schroeder authentification protocol is unsafe [Cir99].

Solving. The notion of rewriting controlled by strategies is used in [Cas98a, KR98] to describe in a unified way the constraint solving mechanism as well as the meta-language needed to manipulate the constraints [Cas97a, CK98]. This provides programs that are very close to the proof theoretical setting used now to describe constraint manipulations like unification or numerical constraint solving. ELAN offers a constraint programming environment where the formal description of a constraint solver is directly executable. ELAN has been tested on several examples of constraint solvers for various computation domains and combinations like abstract domains [KR98, Rin97] (term algebras) and more concrete ones (finite domains, integers). In [Cas96a, Cas96b, Cas97b, Cas98c], it is shown how to use computational systems as a general framework for handling Constraint Satisfaction Problems (CSP for short). The approach
leads to the design in ELAN of COLETTE, a solver for constraints over integers and finite domains [Cas98b]. We also combine rules, constraints and strategies in order to deal with problems like planning and scheduling [DK99].

6.2 ELAN’s annotated bibliography

[BKK+96b, BKK97, BKK+98b] General presentations of the different ELAN system releases.

[BKK96a, BK97, Bor98b] A description of the operational as well as denotational semantics of the new ELAN strategies, available since version 2.0. This allows the user to define its own strategies as a computational system.

[BKK98a] Towards a functional operational semantics of ELAN.

[BJMR98] Some useful applications related to the REF format.

[BC98] The facilities provided in ELAN for the cooperation of constraint solvers.


[Bor95] An ELAN implementation of higher-order unification by using a calculus of explicit substitutions.

[Cas96b, Cas96a, Cas97b] The use of ELAN to specify various constraint manipulation algorithms for CSP’s.

[Cas98a, KR98, Cas97b, Cas98b, Cas98c] The rule-based approach to implement Constraint Programming and Constraint Solving Techniques.

[CK99a, CK99b] The presentation of the rewriting calculus that provides a full semantics to ELAN.


[DK98] The strategy language is used to build plans.


[KKV95] The first presentation of the general ideas developed in ELAN, with the definition of computational systems (including the definition of strategies) and the application of ELAN to design and to prototype constraint programming languages.

[KM96] An illustration of the reflective power of ELAN and rewriting logic.

[KM95] The ELAN implementation of some rule-based completion procedures.


[MK97, MK98, KM98, Mor99] A description of the new compilation techniques developed for ELAN, in particular for associativity and commutativity.

[Rin97] The use of ELAN to interface various unification algorithms and to build combination algorithms for unification.
[Vir96] An implementation of input/output in ELAN via an explicit substitution calculus for the $\pi$-calculus.

[Vit96] A description of the first ELAN compiler.

[Vit94] The first main version of the ELAN system in full details, from design to implementation. In French.
Acknowledgements

We are very grateful to Mark van den Brand whose comments, enlighting questions and always pertinent suggestions allowed us to come up with a much better state of this manual. Many thanks to Carlos Castro, Thomas Genet, Quang-Huy Nguyen and Christelle Scharff for many fruitful discussions and constructive criticisms about ELAN, for their careful reading of the manual and checking of examples. Many thanks also to Joumadi Benhassen for his feedback. Any remaining errors are of course entirely ours.

Specials thanks are going to José Meseguer for the many discussions we had on rewriting logic and rewrite based programming languages and to Steven Eker for his participation to a early stage of this project and for always useful interactions.

Many thanks also to all ELAN's users for their feedback on the language and the system.
Index

:=, 39
AC, 36
AC, 36, 38
alias
  strategy, 37
alias, 34–37
aliasing, 34, 37
AND, 62
ANDIF, 62
<arg.repetition>, 61
assocLeft, 36
assocLeft, 36, 38
assocRight, 36
assocRight, 36, 38

<boolean term>, 39, 58, 59, 62
bs, 38, 39, 43
builtin, 36, 37

<char>, 31, 32, 61
check with, 58, 59
choose, 39
choose-try, 41
<choosing>, 43
code, 36
code, 36, 37
Coercions, 36
comments, 32
<concatenation>, 43
condition, 38
<constant expression>, 60, 61
dc, 43
dc one, 43, 44
dccall, 85, 86
declaration
  function, 34
  operator, 34
  sort, 33
  strategy, 37
defined
  strategy, 49
defined strategies, 42
<defined strategy option>, 38
<defined strategy rule>, 42, 49
description, 58
dk, 43, 53
dkcall, 85, 86
<Dstrategy term>, 39, 49
<elan identifier>, 31, 34, 39
elementary strategies, 42
elementary strategy, 43
<elementary strategy option>, 38
<elementary strategy rule>, 42, 43
END, 86
dc, 32, 38, 39, 42, 43, 49, 58
duser, 7
<Estrategy name>, 43, 59
<Estrategy term>, 39, 43
evaluation
  mechanism in ELAN, 50
explicit, 42, 43

fail
  strategy, 42
fail, 43
<family of rules>, 32, 38
<family of strategies>, 32, 42
first, 43
first one, 43, 44
FOR EACH, 62
<for each const.>, 62
<formal module name>, 32, 57
function symbol, 34

global, 38
<global operator definition>, 33
<global strlop definition>, 33

id, 43
<identifier>, 31–33, 35, 38, 57–59, 61, 86
if, 38

January 27, 2000
if, 39

<if-where-choose>, 39, 49
implicit, 42, 43
import, 58, 59
<imports>, 32
<instantiated module name>, 32, 58, 59
iterate*, 43, 46
iterate+, 43, 46
<iterator>, 43

<labelled or non-labelled rule>, 38
<labelled rule>, 38
labelled rules, 38
letter, 31
<letter>, 31
<lexem>, 61
lexical unity, 31
<LGI file name>, 28
local, 38
local assignment, 38
<local operator definition>, 33
<local stratop definition>, 33
LPL, 58
<lpl description>, 58

<module>, 32
module, 32

name
  sort, 32, 33
  <name>, 34, 37
  <nchar>, 31, 34
  <new operator declaration>, 34
  <new strategy declaration>, 37
  <next solution>, 86
  <next solution demand>, 86
  <non-labelled rule>, 38
non-labelled rules, 38
normalise, 43
<normalize>, 43
normalize, 43
number, 32
<number>, 32, 34, 36, 38, 60, 61, 86
<numeral>, 31, 32

of sort, 58, 59
<old name>, 34
<old operator name>, 34
<old strategy name>, 37
operator, 34

<operator alias>, 34
<operator declaration>, 34
operator definition, 33
<operator definition>, 32
<operator name>, 34
<operator option>, 34, 36
<options>, 28

part, 59
pattern
  in where, 40
patterns, 40
pri, 36
pri, 36, 38
primal strategy, 44
priority, 36
<processus input>, 86
<processus output>, 86

<qchar>, 31, 32
query, 58
<query term>, 59
<quoted identifier>, 31, 34
<quoted single lexem>, 34

rank, 34
  strategy, 37
<rank>, 34, 35
<REF file name>, 28
repeat, 46
repeat*, 43, 46
repeat+, 43, 46
result of sort, 58
<rule body>, 38, 39
<rule label>, 38, 49
rules
  labelled, 38
  non-labelled, 38
rules for, 38

selectors, 34
separators, 31
<simple repetition>, 61
<single lexem>, 34

sort
  built-in, 33
  declaration, 33
definition, 32
  in pattern, 40
name, 32, 33
strategy, 37
\textbf{sort}, 32
sort declaration, 33
\textlt{sort definition}\textgreater, 32
\textlt{sort name}\textgreater, 32, 33, 35, 37–39, 42, 43, 58, 59, 62, 69, 86
\textlt{sort or strategy sort name}\textgreater, 43
\textlt{specification description}\textgreater, 58
\textbf{specification description}, 58
\textlt{specification part description}\textgreater, 58, 59
\textlt{start term}\textgreater, 58, 59
\textbf{start with}, 58
\textbf{strategies for}, 42
strategy, 42
defined, 42, 49
elementary, 42, 43
failure, 42
family, 42
primal, 44
\textlt{strategy}\textgreater, 86
\textlt{strategy alias}\textgreater, 37
\textlt{strategy arg}\textgreater, 37
\textlt{strategy args}\textgreater, 37
\textlt{strategy body}\textgreater, 49
\textlt{strategy declaration}\textgreater, 37
\textlt{strategy name}\textgreater, 37, 62, 69
strategy operator, 37
\textlt{strategy option}\textgreater, 37, 38
\textlt{strategy rank}\textgreater, 37
\textlt{strategy sort name}\textgreater, 37, 38, 43
\textlt{strategy variable declare}\textgreater, 42, 43
stratop definition, 33
\textlt{stratop definition}\textgreater, 32
\textlt{subexpression}\textgreater, 60, 61
\textbf{SUCH THAT}, 62
super-user, 7
\textlt{symbol}\textgreater, 34
\textlt{term}\textgreater, 39, 62, 86
\textlt{text}\textgreater, 61, 62
\textbf{try}, 39
user, 7
\textlt{var affect}\textgreater, 62
\textlt{var decl}\textgreater, 62
\textlt{variable}\textgreater, 38
\textlt{variable declare}\textgreater, 38, 39
\textlt{variable name}\textgreater, 39, 43
\textlt{var name}\textgreater, 62

visibility
of alias, 57
of operator, 57
of sort, 57
rule labels, 42

where, 40
with pattern, 40
where, 39
Bibliography


January 27, 2000

ELAN user manual


January 27, 2000