Elan 1.5 Manual

Manual for the Elan Programming Environment
for personal computers

April 9, 1996

C.H.A. Koster
Th.P. van der Weide

University of Nijmegen
Department of Informatics
The Netherlands

Copyright ©1989 by University of Nijmegen. All rights reserved.

Elan is an educational programming language for learning and teaching systematic programming. The Elan project group at the University of Nijmegen in the Netherlands has developed an educational programming environment around version 1.5 of Elan. This document is the manual for that programming environment which is herewith made freely available to the international computing community for educational purposes. No commercial use is allowed without written prior consent from the authors.

INTRODUCTION

About this manual

This is the user manual for version 1.5 of the Elan Programming Environment developed at the university of Nijmegen in the Netherlands, which is currently available on various UNIX machines, MS-DOS machines, Apple Macintosh, Atari and Amiga. It is distributed without warranty, for use in teaching. It offers a nearly complete version of the educational language Elan in the form of the Elan Programming Environment.

This manual gives a short description for the user of the Elan Programming Environment. It assumes the reader to have a good knowledge of programming in algorithmic languages.

This manual can not teach you how to program. For an English textbook on systematic programming with Elan we suggest:

C.H.A. Koster, TOP-DOWN PROGRAMMING with Elan,
Ellis Horwood Series in Computers and their Applications,
1987, ISBN 0-7458-0187-0 (£ 19.50 / $ 37.45).

which is marketed in the USA by Wiley, and can be ordered in the UK from

John Wiley and Sons ltd, FREEPOST, Baffinslane,
CHICHESTER, West Sussex, PO19 1YP, England.

The present manual starts with an example session, introducing the reader in a hands-on fashion to the use of the Elan Programming Environment. Then follows a brief introduction to Elan-0, the subset of Elan which is intended for introductory programming, followed by some examples. The second part of this manual starts with an overview of the full Elan language, in the form of annotated syntax diagrams. Then follows a description of the the user interface of the programming environment and some more ambitious examples. In an appendix an overview of the available library is given.

About the language

Elan was developed in 1974 by a group at the Technical University of Berlin (see [1]) as an alternative to BASIC in teaching, and approved for use in secondary schools in Germany by the "Arbeitskreis Schulsprache". It is presently in use in a number of schools in Western Germany, Belgium, The Netherlands and Hungary for informatics teaching in secondary education, and used at the university of Nijmegen in the Netherlands for teaching systematic programming to students from various disciplines and in teacher courses.

Elan was especially designed for one specific application area: the teaching of systematic programming. The language is not oriented towards general usage or towards other application areas. It can be seen as a didactic framework embodying a number of ideas about systematic programming and supporting, through specific language mechanisms, the learning of the two complementary programming styles

- Top-Down programming, using suitable control structures and data structures, refinements, and shorthand declarations; and
- Bottom-Up programming, using procedure-, operator- and type-declarations in conjunction with encapsulation and interfaces;

as well as a number of related programming styles (recursive programming, modular programming, syntax-directed programming).

Elan is a typical algorithmic language in the ALGOL family, more related to ALGOL68 than to PASCAL. The language is not an experiment in language design; both syntactically and semantically it is quite conventional. Its control structures are the conventional Dijkstra structures, in conjunction with a leave-statement. Its data structures are limited to the (fixed size) row and structure - not as rich as ALGOL 68 but much simpler due to the absence of the reference concept.

In order to support the learning of systematic programming, it stresses instead the use of abstraction mechanisms. Its striking features are:

- The *refinement* as a syntactic construct for the support of Top-Down programming;
- Declarations for *procedures*, *types* and *operators* for the support of Bottom-Up programming; and
- *Packets with explicit interfaces* for the support of modular programming (not realized in version 1.5).

About this implementation

From the start, Elan was conceived by its designers as a family of (upward compatible) languages:

- Elan-0 A subset sufficient for learning well-structured Top-Down programming, intended as a viable alternative to BASIC for school use;
- Elan-1 A large subset, suitable for learning both Top-Down and Bottom-Up programming;
- Elan-2 The full standard language. Suitable for use in a classroom community because of its packet- and precompilation mechanism;
- Elan-3 Extensions to the standard language for specialized teaching in data structures in a university environment;
- Elan-4 A language for teaching the concepts of systems implementation.

Of these 5 language levels, the first two have been implemented at the university of Nijmegen, with support from the Ministry of Education (NIVO project). Levels 3 and 4 are still the subject of experimentation. A compiler for Elan-3 is in development.

This manual concerns the Elan-1 Programming Environment. Elan-1 differs from the the full language chiefly in the fact that it does not have the packet mechanism (and consequently no precompilation). It is, however, completely adequate for teaching systematic programming in secondary schools, and in courses for teachers and university students. The simplest industry standard PC, with at least one floppy drive and 256 Kb of memory, is sufficient for its use. For people having even smaller computers (like Apple II, Commodore 64 and Philips 2000), implementations of the smaller Elan-0 Programming Environment are available from

Katholieke Universiteit Nijmegen,
Informatica/Elan project,
Toernooiveld 1,
6525 ED Nijmegen, The Netherlands.
Email:elan@cs.kun.nl

The Elan Programming Environment is a portable environment. On any machine, it will behave as described here, apart from some frills and idiosyncrasies typical for the machine used (such as self-explanatory pull-down menus, etc.). Therefore it allows a high degree of machine independence in producing courseware.

It should be realized that the present version 1.5 is an intermediate product. It is relatively slow, with severe limitations on the available memory. A more complete implementation of Elan is in preparation, including a packet library facility with pre-compilation and a compiler back-end. In the mean time the present version should be sufficient for most teaching purposes - and it will only get better.

Any problems or requests about the Elan Programming Environment can be send to the same address. Note however, that the University of Nijmegen gives no warranty, and does not promise to correct errors reported in this version.

About other implementations

Two implementations of the full language (Elan-2) are available from other sources, the TUB and the EUMEL implementation.

The TUB implementation follows the standard faithfully. It is available as a compiler on various machines (IBM 370, Nixdorff, large PC's) from

W. Koch,
Forschungsgruppe Softwaretechnik,
Technische Universität Berlin, FR 5-6,
Franklinstrasse 28-29,
D - 1000 Berlin 10.

The EUMEL implementation offers an extented version of the full language, embedded in a complete multiuser environment and enriched with a large number of application packets. It is available as a programming environment on various machines (PC/AT with hard disc, Z8000 and MC68000 based systems with hard disc) from

ERGOS Gmbh,
Bergstr. 7,
D - 5200 Siegburg.

Bibliography

The Elan standard is described in:

[1] G. Hommel, J. Jäckel, S. Jähnichen, K. Kleine, W. Koch, C.H.A. Koster,
ELAN-sprachbeschreibung,
Akademische Verlagsgesellschaft, Wiesbaden 1979, ISBN 3-400-00384-0.

Some Elan textbooks in various languages are:

[2] C.H.A. Koster, Top-Down Programming with Elan.
Ellis Horwood, 1987, ISBN 0-7458-0187-0.

[3] C.H.A. Koster, systematisch leren programmeren,
deel 1: Top-Down programmering,
Academic Service, 1988, ISBN 906233-371-0.

[4] C.H.A. Koster, programozás felülnézetben,
Müszaki Könyvkiadó, Budapest 1988, ISBN 963-10-7579-6.

[5] L.H. Klingen, J. Liedtke, Programmieren mit ELAN.
Teubner Verlag, Stuttgart 1983, ISBN 3-519-02507-8.

Some textbooks using Elan:

[6] Baeten et.al., Initiatie in de informatica,
handleiding voor de bijscholing leraren secundair onderwijs,
2 delen, Acco Leuven/Amersfoort, 1985.

[7] R. Danckwerts, D. Vogel, K. Bovermann,
Elementare Methoden der Kombinatorik, mit Programmbeispielen in Elan,
Teubner Verlag, Stuttgart 1985, ISBN 3-519-02529-9.

[8] L.H. Klingen, J. Liedtke, Elan in 100 Beispielen,
Teubner Verlag, Stuttgart 1985, ISBN 3-519-02521-3.

[9] A. Otto, Analysis mit dem Computer,
Teubner Verlag, Stuttgart 1985, ISBN 3-519-02528-0.

A toolkit for an advanced data structure course can be obtained (in the form of a floppy disc) from the project group in Nijmegen.

Chapter 1

AN EXAMPLE-SESSION

Upon receiving your wonderful new Elan Programming Environment your first inclination, no doubt, is to sit down in front of your computer and try it out. This desire is understandable, but will lead to severe disappointments unless you already know how to program and know the language Elan. This manual cannot teach you programming, for that we refer you to the textbooks mentioned in the introduction. In chapter 4 you find an overview of the Elan language. In chapter 5 you find a description of the Elan Programming Environment. *The best strategy is to read those two chapters first before embarking upon the present chapter.*

The example session in the current chapter introduces you in a hands-on fashion to the use of the Elan Programming Environment. By following the session to the letter you obtain a quick tour of most of its facilities.

In this chapter we will mark output from the Elan Programming Environment with a line in the left margin; the input of the user is given without such a mark. The cursor will be depicted as ■. In input from the user we will explicitly indicate the "return" or "enter" key by `<RET>` and the "delete" key by `<DEL>`.

1.1 Starting...

After you have booted and started the Elan Programming Environment, the following identifying message appears on the screen:

```
|       Elan Programming Environment
|       version 1.5
|       Copyright KUN, Aug 1988
```

Immediately underneath you will find the *focus*, i.e. the name of the refinement that is the current focus of interest. The default focus is

```
|       program ?■
```

The question mark means that `program` is not the name of a known refinement, i.e. it has not yet obtained a meaning. At the bottom of the screen, on the status line, you find the message

```
|                                            Command, please...
```

indicating what is expected from you. By giving the help-command

```
        h
```

you are shown for one second a list of the applicable commands. The commands are described in chapter 5 of this manual. The help-command serves only as a reminder.

1.2 ... and stopping

Once having learned how to start you must learn how to stop, just like the apprentice sorcerer. You can stop the programming environment by giving the quit-command

```
q
```

where upon you are asked for a confirmation

```
|        QUIT ?■
```

The possible answers (as elicited by the help-command) are `y` and `n`. Upon answering

```
n
```

the quit-command does nothing and the confirmation message disappears from your screen. Upon answering

```
y
```

the programming environment halts and returns control to the system on which it is running. Upon quitting, the program that is then in memory is forgotten (but of course all programs that have been written to a file are retained).

1.3 Reading a program

As an example we will read, execute and modify the program called `race` (see 3.1), stored in a file with the same name, which can be found on the floppy disc containing the Elan Programming Environment. We make sure the floppy disc is inserted and give the read-command

```
r
```

The programming environment reacts by asking

```
|        Name:■
```

We now have to indicate the name of the file. We type in

```
race<RET>
```

Notice that this time we have to give a `<RET>` at the end in order to indicate that the name is completed. Since commands consist of one single letter they do not need such an indication. The status line now says

```
|                                Reading...
```

while the text of the refinements of the program passes over the screen. Upon completion of the reading, the interpreter gives a view, a list of all known refinements and an indication of the remaining memory space.

```
|       Free space:    17935
|
|       program
|       | start horses
|       | | display title, put horse at start pos,
|       | | mark start of track, mark end of track
|       | choose horse, move horse, finished, display winner
|
|       program:■
```

The status line again says

```
|                              Command, please...
```

This view you now have can also be obtained by giving the list-command

```
l
```

1.4 Inspecting and executing a refinement

The refinement `program` has, by the reading, obtained a value, which can be displayed on the screen by means of the show-command

```
s
```

```
|       program:
|         start horses;
|         REP
|           choose horse;
|           move horse
|         UNTIL finished
|         ENDREP;
|         display winner.
|
|       program:■
```

The focus is still `program`, but now its meaning is defined, as is shown by the fact that it is followed by a `:` instead of a `?`. We can execute the program with this refinement as root by giving the execute-command

```
x
```

Just try it, and see what happens.

1.5 Shifting the focus

In order to inspect or execute another refinement, e.g. `start horses`, we have to shift the focus by means of the focus-command

```
f
```

```
|        Name:■

         start horses<RET>

|        start horses:■
```

We can now inspect this refinement by means of the show-command. The focus-command (`f`) can be used to focus on any name, be it already defined or unknown. For focussing on known names, a shorthand mechanism exists: we may replace the last part of the name by a `*`, e.g.

```
         start h*<RET>
```

Notice that if we had given a shorter part of the name `start*<RET>` we would have obtained the object `start pos` instead.

1.6 Modifying a refinement

We will now modify one of the refinements of the program. We are focussed on `start horses` and give a show-command

```
         s

|        start horses:
|          LET no of horses = 6;
|          LET start pos = 12;
|          LET end pos = 50 ;
|          ROW no of horses INT VAR horse;
|          display title;
|          INT VAR i;
|          FOR i FROM 1 UPTO no of horses
|          REP
|            put horse at startpos;
|            mark start of track;
|            mark end of track;
|          ENDREP.
|
|        start horses:■
```

We wish to change the `6` in the first line of the body of the refinement into `4`. To that end we give an edit-command

```
         e
```

The same text appears, with on the status line

```
|                                Input, please...
```

By means of the cursor keys (marked with arrows) we move the cursor to the right position, strike out the `6` by means of the `<DEL>`-key and insert a `4` instead. We then strike the `<RET>`-key to indicate that the modification is complete.

We now want to see what effect this modification has on the execution of the program. We first have to focus again on the root

```
f
```

```
Name:■
```

We make use of the shorthand mechanism

```
prog*<RET>
```

```
program:■
```

By giving the execute-command we can verify that now only four horses take part in the race.

1.7 Saving a program

We can now save the the program to a file by means of the write-command

```
w
```

```
Name:■
```

We have to choose a name for the file, e.g.

```
race4<RET>
```

On the status line the message

```
                                    Writing...
```

appears and after some time we again obtain the prompt

```
program:■
```

1.8 Clearing the memory

We can clear the whole memory by means of the clear-command

```
c
```

whereupon the interpreter asks for confirmation

```
CLEAR ?■
```

The two possible answers are

```
n
```

whereupon the command has no effect, and

```
y
```

which causes the interpreter to come back to its initial state with an empty memory.

1.9 Directory of files

The programs saved in this fashion can be read back from file into memory by means of the read-command. Since we may in time forget the names of our files, it is possible to get a directory, i.e. list of file names. We obtain this by a somewhat roundabout way, viz. by trying to read a file that is definitively not there, e.g.

```
r

Name:■

rrrrr<RET>
```

We can now inspect the directory to decide what is the name of the file we want to read.

1.10 Entering a new program

It is of course possible to input a program from scratch. It is not at all advisable to program whilst sitting in front of the computer screen, so we assume the reader to have written the following little program, that he now wishes to enter into the computer:

```
my first program:
  read two numbers;
  print their sum.

read two numbers:
  read first number;
  read second number.

read first number:
  INT VAR first;
  put ("First number = ");
  get (first);
  line.

read second number:
  INT VAR second;
  put ("Second number = ");
  get (second);
  line.

print their sum:
  line;
  put ("Sum = ");
  put (first + second).
```

After starting the Elan Programming Environment, first focus on the name of the refinement which is to serve as the root of the program. Since it is rather hard to change the name of the root once it is chosen, we have to choose it with care.

```
f
```

```
|       Name:■

        my first program<RET>
```

The prompt tells us we have focussed successfully.

```
|       my first program ?■
```

We now supply a definition for the root by means of the editor.

```
        e

|       my first program:
|       ■
```

We type:

```
        read two numbers;print their sum.<RET>

|       my first program:■
```

The prompt now tells us that the root is a refinement. Notice that we need not take particular pains to make a nice layout, because the *pretty printing* function of the show-command will standardize the layout anyway.

```
        s

|       my first program:
|         read two numbers;
|         print their sum.
|
|       my first program:■
```

1.11 Automatic guidance of input

The prospect of entering the remaining refinements one by one, by first focussing on their name and then using the editor, is rather forbidding. Luckily there is a possibility to avoid all the effort of focussing and retyping names: we can be guided by the Elan Programming Environment, so that we have to input only the bodies of the refinements.

We are focussed on the root and try to execute it.

```
|       my first program:■

        x
```

Immediately we are warned of the fact that the first refinement is unknown.

```
|       read two numbers
|                                read two numbers■
```

```
|       Can't identify          Backtrace command, please...
```

We are now in backtrace-mood, and give the edit-command

```
        e
```

The Elan Programming Environment responds by

```
|       read two numbers:
|       ■
```

After we enter the body, we return to the root of the program, which we try to execute again.

In this way, we are made to enter one refinement at a time, until the program is complete. This guidance during input allows us to type most refinement names only once, thus alleviating one of the drawbacks of Top-Down programming and encouraging the use of meaningful identifiers.

1.12 Entering procedures and types

Once you reach the stage of Bottom-Up programming, you must learn how to input your own types, procedures and operators. You do this by first focussing on any name (for instance the name of the type or procedure you want to enter), give the edit-command and then enter its complete declaration followed by a semicolon.

Some examples. Let us first enter a procedure `new number`.

```
        f

|       Name:■

        new number<RET>

|       new number ?■
```

The Elan Programming Environment takes this for a refinement name.

```
        e

|       new number:
|       ■
```

We edit the text on the screen until we have obtained a complete procedure declaration, followed by a semicolon.

```
        INT PROC new number:<RET>
          INT VAR n;<RET>
          get(n);<RET>
          n<RET>
        ENDPROC new number;<RET>
```

We can now see from the prompt that our focus is a procedure

```
        INT PROC new number■
```

As a second example, let us enter a type-declaration.

```
        f

|       Name:■

        COMPL<RET>

|       COMPL ?■
```

We massage this text on the screen into the type-declaration we want to have.

```
        TYPE COMPL = STRUCT (REAL re, im);<RET>

|       TYPE COMPL■
```

Do not forget the semicolon at the end.

The order of refinements, type-declarations, procedure-declarations and operator-declarations is free, but the first refinement which is entered becomes the root of the program.

1.13 Inspecting the standard library

By means of the focus- and show-command we can inspect not only the definitions which we have entered ourselves but also those of the standard library (see Appendix A). In chapter 5.2.4 the mechanics of this facility are explained. We will use it to look at the available additions.

```
        f

|       Name:■

        +<RET>

|       INT OP + (INT CONST a)■
|       REAL OP + (REAL CONST a)
|       INT OP + (INT CONST a, b)
|       REAL OP + (REAL CONST a, b)
|       TEXT OP + (TEXT CONST a, b)
```

We observe there are several definitions, of which the first one is presently the focus. By means of the (repeated) use of the next command

```
        n
```

we can make any of these definitions the focus. We can inspect it by giving a show-command

```
        s
```

```
|       INT OP + (INT CONST a):
|         CODE 30
|       ENDOP +
|
|       INT OP + (INT CONST a)■
|       REAL OP + (REAL CONST a)
|       INT OP + (INT CONST a, b)
|       REAL OP + (REAL CONST a, b)
|       TEXT OP + (TEXT CONST a, b)
```

which does not really makes us much wiser.

A name with more than one definition (such as + or `put`) is called generic. Modifying definitions for generic names from the standard library is rather tricky. The easiest way is to focus on some harmless refinement name (like `a`) and then massaging the resulting screen text into the desired definition.

1.14 Standard packets and standard programs

In a classroom situation it will not always be the case that a complete program is written from scratch. Rather the teacher will supply a packet of refinements, or even a complete program that the pupils have to use. For this reason, the Elan Programming Environment has a simple facility for reading standard packets and standard programs.

A *standard program* is a complete Elan program, to be used without modifications or additions. Its first refinement must have the name `program`.

A *standard packet* is a collection of declarations that the pupils may use in a program of their own without the necessity to type them in. None of its refinements may have the name `program`.

For reading these, a variant of the read-command has been introduced, the packet-command (`p`), which has the property that it keeps the contents of the packet practically invisible: the pupil can inspect but cannot modify or delete them.

1.15 Example of a standard program

As an example we can try to read any program as a standard program. We give the packet-command

```
        p

|       Name:■
```

In typing the name (followed by `<RET>`) it is not shown on the screen. Similarly, in reading the refinements of the program only their names (not their bodies) are shown. Immediately after the reading a list of all refinements of the program is displayed on the screen, but this is not reproducible by means of the list-command because the refinements become “invisible”. We can execute etc. the program in the usual way but we can not modify it. At most we can add new refinements.

1.16 Example of a standard packet

The program `karel` (see 3.7) is a standard packet but not a standard program. The difference is that the root of a standard program must be named `program` whereas for a standard packet it must be different from `program`. This gives us the opportunity to add our own refinement `program` as root of the program, in which all the refinements of the packets can be used. The packet as it were extends the language with a number of concrete algorithms.

In order to use this standard packet we first read it with a packet-command. Then we read the standard packet `environ` in the same way. Finally we read the program `newspaper` by means of a read-command and give the execute-command. Only the refinements of this last program can be modified or deleted. The refinements of the packets can not be modified or deleted and will not be saved by a write-command.

1.17 Conclusion

In this quick tour we have missed many important aspects of the programming environment, such as

- deleting a refinement by means of a delete-command (`d`);
- the incremental syntax check during input of new refinements;
- the backtrace facility for semantic errors;
- the edit-on-the fly upon detection of syntactic or semantic errors;
- the trace facility.

We have not discussed useful tricks, such as

- how to change the name of a refinement (take it into the editor, edit its name, delete the original);
- how to inspect the value of a variable or constant after execution of the program (focus on its name and show it);
- the use of the `<BREAK>`-key to halt execution or input.

Since these are all consequences of things described in chapter 5, it may be useful to read that chapter again.

Chapter 2

A BRIEF INTRODUCTION TO ELAN-0

Elan-0 is a small subset of Elan, which is intended for learning systematic Top-Down programming. The best way to get acquainted with Elan and to appreciate the programming style it aims to support is to start with the Elan-0 subset.

This chapter of the manual is intended for readers who already have some experience in programming in an algorithmic language. Notice that a careful distinction is made between remarks which pertain to Elan as a whole and remarks confined to the Elan-0 subset. In chapter 4 the syntax of the full language (with the exception of packets) can be found, which is available in the Elan Programming Environment.

2.1 Some concepts of programming languages

The following terminology and concepts are applicable to any programming language, but in particular to Elan, since they lie at the heart of its design. Readers not interested in philosophy may skip this section.

2.1.1 Entities

A program is a text, viz. the formulation of some algorithm, expressed in a programming language. This text obeys the syntax of the programming language, and according to that syntax consists of a nested collection of constructs. Among these constructs, three general classes can be distinguished: *algorithms*, *objects* and *types*, which have the distinction that they can be denoted by names. The program text consists of such *entities*, glued together by control structures, data structures and further composition mechanisms such as expressions and functional application.

2.1.2 Elementary and composed entities

An entity which is denoted by a name we will term *elementary*; the others are *composed.* It may very well be that a composed entity at one place of the program obtains a name, by means of a *declaration*, and at other places of the program is denoted by that name. Such a declaration serves as an abstraction mechanism. Besides a shortening of the program, this naming of entities allows the introduction of various *levels of abstraction.*

2.1.3 Abstract and concrete entities

Some elementary entities belong to the language and are available without further effort by the programmer, with their specific semantics. These we will term *concrete.* Others we will term *abstract*: they are constructed by the programmer and abstracted by means of a declaration. We might define programming along these lines as the construction of abstract algorithms.

2.1.4 Composition mechanisms

In the design of an algorithm a programmer should not try to express it in all detail at once. Instead, the programmer should split it into major parts which can be refined in a stepwise fashion down to the elementary entities belonging to the programming language. Algorithms are thus composed of other algorithms which in their turn are either composed or elementary.

Particular composition mechanisms for the algorithms on the one hand, and the objects and types on the other hand, are termed *control structures* and *data structures*, respectively.

2.1.5 Where does Elan fit in this taxonomy?

Traditionally, composition mechanisms have been seen as the central mechanisms in systematic ("structured") programming. Elan has rather conventional control structures and relatively simple data structures. In contrast, in Elan the abstraction mechanisms are seen as the central issue in any programming methodology and consequently highly developed.

The Elan Programming Environment further stresses this view by making the definition of abstract algorithms (refinements) the basis for the development and manipulation of programs. It considers a program as a collection of refinements, rather than one text.

2.2 The notation of algorithms

2.2.1 Names of algorithms

Names are either the names of concrete algorithms, which means that they are predefined in the Elan Interpreter, or they are introduced by the programmer by the definition of abstract algorithms using refinements. Here we explain their formation rules.

Names for abstract algorithms can be freely invented. Such a name has the form of an *identifier*, consisting of a leading (lower case) letter, followed by letters, digits, and possibly embedded spaces. The latter serve to enhance the readability of programs. In contrast to full Elan, such spaces in Elan-0 are considered significant and are part of the name. Some examples:

```
find upper limit
word occurs on this page
```

Names shall be chosen such that they express concisely *what* is done by an algorithm, not spelling out in detail *how* it is performed. Inventing suitable names is a non-trivial task and needs experience which can be gained only by the study of good examples, exercises, and by the contemplation of programming problems.

2.2.2 Control structures

The first step towards a precise description of algorithms is the use of a collection of stylistic patterns for connecting together the algorithmic steps performed in the execution of programs. These patterns are called *control structures*. Elan-0 knows the following:

- sequence;
- repetition;

- choice.

The forms available in Elan are also available in most other programming languages. Elan uses a notation with keywords written in upper case. They will be introduced by way of examples in the next subchapters.

2.2.2.1 Sequence

The sequential execution of the steps of an algorithm is denoted by placing a semicolon between them.

```
read the current page ;
turn page over ;
read the current page
```

These are the names of three abstract algorithms to be executed one after the other. The semicolon is a *separator* between them (not a terminator), and can be read as "and then".

We call such a sequence a *paragraph*, and its constituents *units*. The example paragraph above contains 3 such units and its execution consists of the execution, in that order, of the three units. In this way the *effect* of the paragraph is the composition of the effects of its units. The *value* of a paragraph (if any) is the value of its last unit.

2.2.2.2 Repetition

Repetition can be expressed in a number of ways. An important paradigm of repetition is the while-loop, e.g.

```
WHILE
  the word does not occur on this page
REP
  look at the following page
ENDREP
```

The *condition* between WHILE and REP is a unit which, upon evaluation, yields a truth value (true or false).

The paragraph between the keywords REP and ENDREP is executed only if the condition yields true. After the execution of the paragraph (the "loop-body") the condition is evaluated again, and accordingly the loop-body may be executed repeatedly. The execution ends as soon as the condition upon evaluation yields false. For those who prefer more readable keywords, REP may be also written as REPEAT, and ENDREP as ENDREPEAT.

```
WHILE
  the sun is shining
REP
  have a drink ;
  sing another song
ENDREP
```

In this kind of repetition a test of applicability is performed before each cycle ("pre-checked-loop"). If in this example it is rainy, the loop-body is never executed, and you have to stay sober.

For cases where there is always something to be done first, and checked for repetition afterwards ("post-checked-loop") Elan has the following complementary repetitive construct:

```
REP
  drink a glass of tea with rum ;
  sing a christmas carol
UNTIL
  it stops snowing
ENDREP
```

In this case the condition which stands between `UNTIL` and `ENDREP` is executed after each execution of the first paragraph, the repetition continues if the condition yields false and stops if the condition yields true.

For cases where the number of repetitions is known in advance, Elan offers a "counting-loop", also called for-loop.

```
FOR i FROM min UPTO max
REP
  tally i th entry
ENDREP
```

The units `min` and `max` yielding integers are evaluated once at the beginning of the for-loop. The controlled integer variable `i` gets the value of `min`. As long as `i` is less or equal to `max`, the loop-body between `REP` and `ENDREP` is executed, followed by an increment of the "loop-variable" `i` by one. Within the loop-body `i` may be used, but not assigned to. Its value is undefined after the execution of the for-loop.

In case you want to count not upwards, but downwards from a larger value to a smaller one, there is a variant of the counting-loop:

```
FOR nr FROM stock DOWNTO minimum
REP
  sell one
ENDREP
```

In this variant, `nr` is counted down from `stock` to `minimum`. If `stock` was already below `minimum`, then the loop-body is not executed.

In case the value of the from-part is one and the controlled variable is of no interest, the for- and from-parts may be left out, leading to the shorter form

```
UPTO 20
REP
  hit him over the head ;
  pick him up
ENDREP
```

A later section will demonstrate the use of the counting-loop in processing rows of data elements.

2.2.2.3 Choice

Choosing between two alternatives depending on a condition is written in Elan as:

```
IF condition
THEN
  part for condition true
ELSE
  part for condition false
FI
```

If the condition evaluates to true, then the paragraph between `THEN` and `ELSE` (the "then-part") is executed, and the rest up to the `FI` skipped. In the contrary case, the then-part is skipped, and the paragraph between `ELSE` and `FI` is executed (the "else-part"). The keyword `FI` may also be spelled as `ENDIF`.

This primary form of choice has two variations for common needs. If there is nothing to do in the else-part, then the choice can be simplified by leaving out the else-part:

```
IF it looks like rain
THEN
  take the umbrella with you
FI
```

In case the weather is fine you do nothing (`ELSE do nothing` was omitted). For decision cascades like

```
IF condition 1
THEN
  action 1
ELSE
  IF condition 2
  THEN
    action 2
  ELSE
    IF
      ...
    FI
  FI
FI
```

Elan offers another variant of the choice construction:

```
IF condition 1
THEN
  action 1
ELIF condition 2
THEN
  action 2
ELIF
  ...
ELSE
  action for all conditions above failing
FI
```

Also in this case, an empty else-part may be omitted.

2.2.2.4 Control structures yielding a value

The execution of a paragraph or a choice can also *yield* a value, namely that of the last unit executed. The paragraph

```
compute sum of integers 1 to 10 ;
that sum + 1
```

suggests that first the integers 1, 2, ... 10 are summed up, resulting in a value of 55, and then the value 56 is yielded by that paragraph. If the last unit of a paragraph yields a value, then the paragraph as a whole yields that value.

In case of a choice, the value yielded is that of the paragraph in the then-part or in the else-part, depending on the condition. These values must have the same type. As an example, the unit

```
IF a < b THEN a ELSE b FI
```

yields the smaller of the two integer values `a` and `b`. Repetitions cannot yield a value.

2.2.3 Refinement

The most striking construct of Elan is the *refinement*, a simple mechanism for defining abstract algorithms which forms the basis for the *Top-Down* programming style, which can be summarized:

> "A program is developed by first giving a rough but potentially correct formulation composed of abstract entities. Thereupon each of these abstract entities is similarly defined, in terms of other abstract and concrete entities, until at last all necessary abstract entities have a suitable definition."

A refinement gives a name to a paragraph, and looks like

```
name:  paragraph.
```

Executing the name of a refinement means executing its constituent paragraph (its *body*). The value of the refinement is the value of its body.

In distinction to procedures (which could also in principle be used as refinements) refinements may appear in any order, and may in particular appear after any invocation of the refinement. Refinements can not have parameters, in order to keep the "visual overhead" in their definition and application to a minimum. A refinement does not form a separate scope of naming. Therefore it is possible to put a declaration in one refinement and use it in another. For these reasons, refinements are better suited than procedures for capturing the "fleeting abstractions" in programming.

In Elan-0, a program consists of one or more refinements, where the first refinement is the *root* of the program. Such programs correspond to procedure-bodies in full Elan.

2.2.4 Leave-statement

Elan has a special mechanism for terminating the execution of a particular refinement, which looks like

```
LEAVE refinement name
```

This mechanism can in particular be used to terminate repetitions from the inside.

Although it looks much like the infamous goto-statement, the leave-statement is very different: It does not allow arbitrary continuation of program execution at some other part of the program with all its known dangers, but completes the execution of an algorithm. For this reason, you may name in a leave-statement only a refinement of whose execution the execution of the leave-statement is part. It is even possible to leave a refinement with a value, by

```
LEAVE refinement name WITH expression
```

This causes the refinement with that name to be left, yielding the value of the expression.

2.3 The notation of objects

Algorithms to be executed on a computer do not work on thin air, but need objects to operate on. In this chapter we describe what kinds of objects are available in Elan-0, and what are the basic algorithms to handle them. Objects occur in programs in two forms:

- Elementary objects are variables and constants which come into life by the execution of *declarations*;
- Composed objects are the *expressions*. Expressions are a convenient notation for the computation of values. A particular kind of expression are *denotations*, a way of writing down values in the program text.

2.3.1 Declaring objects

In Elan all elementary objects must be declared. This declaration may occur at any place in the program, provided there is only one such place, and in executing the program the declaration of an object precedes all its applications. Thus there is no necessity to collect all declarations at the head of the program but declarations can appear at the place where they are first needed. A declaration may even occur within the body of a loop.

A declaration is a unit which introduces an elementary object with four attributes:

- a type;
- a name;
- a value, which may be undefined; and
- an access right.

A declaration makes the name of the object known throughout the whole Elan program.

2.3.1.1 Names

The names of objects are identifiers, just like those for algorithms, and may be chosen freely, as long as they differ from the names of any other entities in the program.

2.3.1.2 Types

The type of an object serves two purposes:

- It expresses what operations can be applied to the objects in question. In a way, it controls that apples cannot be added to oranges;
- It determines the internal representation of values in the computer's memory.

There are four basic types in Elan, denoted in program texts by the keywords:

- `INT`
  integral numbers in the range -2147483647 to 2147483647;
- `REAL`
  real numbers in a precision of 14 decimals;
- `BOOL`
  truth values, true and false; and
- `TEXT`
  sequences of characters.

Furthermore, objects of any type can be composed into one dimensional rows. These composed types are discussed in the subchapter on rows.

2.3.1.3 Values

Since values of the various types are kept within the memory of the computer, little can be said about them. Knowledge of their internal representation is not at all necessary to understand their relevant properties. It should be noted that the Elan programming environment detects and reports the manipulation of undefined values, e.g. an attempt to use the value of an uninitialized variable.

2.3.1.4 Access attributes

The access attribute of an object is either `VAR` or `CONST` and the objects are correspondingly classed as variables and constants. The value of a variable can be changed by an assignment, whereas the value of a constant cannot.

2.3.2 Declarations

A variable declaration may give an initial value to a variable,

```
INT VAR middle ::  ( 1 + max ) DIV 2
```

otherwise its initial value is undefined, as in

```
VAR x
```

Two or more variable declarations can be combined into one unit

```
VAR left pointer ::  1 ,
    right pointer ::  max ,
    middle pointer
```

A constant declaration must give an initial value to a constant

```
INT CONST small ::  4711 ;
TEXT CONST capital ::  "monaco"
```

Each execution of a declaration causes a new elaboration of the initialization. In this way a constant may have a different value after each declaration executed. The term "constant" is therefore somewhat misleading: its value is not changeable by an assignment, therefore it is constant over part or all of the program.

2.3.3 Denotations

Values of basic type can be written down (denoted) in a program text by denotations:

- INT
  Natural numbers can be written as a sequence of digits, e.g. 3, 198, 10000 are all three valid INT denotations.

  Negative numbers can be written as expressions consisting of a monadic minus operator and an INT denotation, e.g. -57, as you might have suspected.

- REAL
  Elan admits the conventional fixed point and floating point representation for (approximations) of real numbers, like 3.1415269, 1.3e-8.

- BOOL
  There are only two values for truth values, which are denoted by the constants TRUE and FALSE respectively. People allergic to capital letters may use the concrete standard constants true and false instead.

- TEXT
  Text denotations consist of sequences of characters enclosed in quotation marks, e.g. "Hello world!"
  All printable characters of your computer are allowed within text denotations. If a quote character is to appear in a text denotation, then it has te be doubled, e.g. "He said ""Don't!""" The empty text consisting of no characters is denoted by "".

2.3.4 Expressions

An expression in Elan is composed in the conventional fashion out of operands, monadic operators and dyadic operators with brackets to indicate grouping. An operand may in Elan-0 be the name of an object, a denotation, a subscription, an invocation of a refinement or the call of a standard procedure. The priority of operators, from high to low, is as follows:

- the monadic operators +, -, NOT, HEAD, TAIL, LENGTH
- *, DIV, MOD
- dyadic +, -
- =, <>, >, >=, <, <=
- AND

- `OR`
- `SUB`
- `INCR`, `DECR`, `CAT`
- the assignment operator `:=`

2.3.5 Assignments

The assignment is a unit of the form

```
var := expr
```

where `var` is a variable of some type and `expr` an expression of the same type.

The assignment serves to change the value of the variable. It is executed by first executing the expression and then making its value the new value of the variable. The value of all other variables remains unchanged. Thus the assignment

```
x := x + 1
```

which can be read as "x becomes x plus one" does not mean that in some curious way x becomes equal to itself plus one, but rather that first the sum of the present value of x and one is computed and then that value is made to be the new value of x.

2.3.6 Composed objects: rows

In Elan-0 the only composed objects are (one-dimensional) row-variables and row-constants. A declaration for a row-variable looks like

```
ROW 5 INT VAR weight
    |   |   |     |____name of the variable
    |   |   |____access attribute
    |   |____type of each element
    |____number of elements
```

After this declaration, the row-variable `weight` has 5 elements, numbered from 1 to 5. Each element has an (as yet undefined) value of the type `INT`.

The subscription

```
row [ i ]
```

denotes its `i`-th element provided the value of the expression `i` (the *index*) is between one and the number of elements. A subscription of a row-variable has the access attribute `VAR`, so it can be assigned to, e.g.

```
weight [ i ] := 144
```

After this assignment, the element whose index is equal to the value of `i`, which must be in the range `1` to `5`, is equal to `144`. Similarly, a row-constant can be declared like

```
ROW 5 INT CONST first 5 primes ::[ 2, 3, 5, 7, 11 ]
```

The construct with the square brackets in this example is a *row-display*, which acts as a denotation for a row. As with other constant declarations, the initialization (with a row-display or another row) in a row-constant declaration is obligatory. A subscription from a row-constant has the access attribute `CONST`, so it can not be assigned to.

Apart from the possibility of manipulating individual elements of a row, it is also possible to deal with the row as a whole, e.g. in the assignment

```
weight := first 5 primes
```

In assignments and initializations, the number of elements in the left and right hand side must match. The elements of a row can in their turn be rows again, with a declaration like

```
ROW 10 ROW 20 INT table
```

and a subscription like

```
table [ i ][ j + 1 ]
```

provided, of course, that `1 <= i <= 10` and `1 <= j + 1 <= 20`.

2.3.7 Synonym declarations

In Elan the number of elements ("upper-bound", "cardinality") in a row declaration must be given by a denotation, that is, it can not be dependent on the execution of the program. A row therefore has a statically fixed number of elements. (This is something of a nuisance but reflects the fact that rows in Elan do not serve like arrays in other languages. They are intended to be somewhat inflexible and primitive, and more convenient data types should be built upon them. In a beginners environment, their simplicity is an advantage).

Since this upper bound may have to appear in many places in a program, a mechanism is available which allows the naming of denotations. Such a *synonym declaration* looks like

```
LET max = 5
```

which causes the identifier `max` to stand for the denotation `5` in suitable places. The identifier `max` can now be used as a denotation, e.g.:

```
ROW max INT VAR weight ;
FOR t FROM 1 UPTO max
REP
  get ( weight [ t ] )
ENDREP
```

In this example, the elements of the `weight` are read successively with the operation `get`. As shown in this example, rows and for-loops work together nicely; the use of synonyms assures that the same range of index values is used.

Chapter 3

EXAMPLES ELAN-0 SUBSET

The following examples are intended to give a flavour of the Elan-0 subset and of the style of programming that can be achieved with it. Further examples (and much more advanced examples) can be found on the distribution disc.

3.1 Horse race

The following program performs a race with 6 horses. It was developed by pupils from a school in Hungary, under supervision of Peter Hanak. It could be extended with a mechanism to accept bets. The program makes use of the `cursor` function to organize the layout of the screen. It uses `choose128` to produce (quasi-)random numbers.

```
program :
  start horses ;
  REP
    choose horse ;
    move horse
  UNTIL
    finished
  ENDREP ;
  display winner .

start horses :
  LET no of horses = 6 ;
  LET start pos = 12 ;
  LET end pos = 50 ;
  ROW no of horses INT VAR horse ;
  display title ;
  INT VAR i ;
  FOR i FROM 1 UPTO no of horses
  REP
    put horse at startpos ;
    mark start of track ;
    mark end of track
  ENDREP .

display title :
  cursor ( endpos DIV 2 , 1 ) ;
  put ( "H O R S E   R A C E" ) .
```

```
put horse at startpos :
  horse [ i ] := startpos .

choose horse :
  INT CONST ix ::  ( choose128 MOD no of horses ) + 1 .

mark start of track :
  cursor ( 1 , 2 * i + 1 );
  put ( i ) ;
  put ( "*" ) .

mark end of track :
  cursor ( endpos , 2 * i + 1 ) ;
  put ( "|" ) .

move horse :
  cursor ( horse [ ix ] , 2 * ix + 1 ) ;
  put ( ".*" ) ;
  horse [ ix ] INCR 1 .

finished :
  horse [ ix ] >= endpos .

display winner :
  cursor ( start pos - 1 , 2 * no of horses + 4 ) ;
  put ( "The winner is horse number:" ) ;
  put ( ix ) ;
  line ( 2 ) .
```

3.2 Drawing a box

This example is a suitable demonstration for Top-Down programming. How does one draw a box on a conventional screen?

```
draw box :
  ask size ;
  draw lid ;
  draw sides ;
  draw bottom .

ask size :
  INT VAR size ;
  line ;
  put ( "size = " ) ;
  get ( size ) ;
  line ( 2 ) .

draw lid :
  put ( size * stripe ) ;
  line .
```

```
draw sides :
  INT VAR i ;
  FOR i FROM 1 UPTO size
  REP draw slice
  ENDREP .

draw bottom :  draw lid .

stripe :  "--" .

draw slice :
  put ( pip ) ;
  put ( ( size - 2 ) * blank ) ;
  put ( pip ) ;
  line .

pip :  "!!" .

blank :  " " .
```

3.3 Circular shifting

The operations on texts HEAD and TAIL that are available in Elan-0 differ markedly from the standard operations in the language, being much more list-oriented. In standard Elan they can be programmed as

```
TEXT OP HEAD ( TEXT CONST a ) :
  a SUB 1
ENDOP HEAD ;

TEXT OP TAIL ( TEXT CONST a ) :
  subtext ( 2 , LENGTH a , a )
ENDOP TAIL ;
```

The following program repeatedly rotates its input text by one position until the original is obtained again.

```
rotate :
  declare texts ;
  REP
    shift s circularly ;
    display s
  UNTIL
    s = original
  ENDREP .

declare texts :
  TEXT VAR original ;
  put ( "Text, please?" ) ;
  get (original ) ;
  TEXT VAR s ::  original .
```

```
display s :
  line ;
  put ( s ) .

shift s circularly :
  s := ( TAIL s ) + ( HEAD s ) .
```

3.4 Converting numbers into texts

Designing a program for conversion between numbers and texts is a good way to get used to the respective properties of numbers and texts. The program illustrates the use of CAT and SUB.

```
outint :
  put ( "This program converts a positive integer" ) ;
  line ;
  put ( "to a given radix in range 1 to 16." ) ;
  line ;
  ask for radix ;
  WHILE
    another number ;
  REP
    represent number according to radix ;
    print representation
  ENDREP.

ask for radix :
  INT VAR radix ;
  put ( "Radix, please:" ) ;
  get ( radix ) ;
  line .

another number :
  ask for number ;
  number > 0 .

ask for number :
  INT VAR number ;
  put ( "number, please:" ) ;
  get ( number ) ;
  line .

represent number according to radix :
  TEXT VAR representation ::  "" ;
  REP
    isolate a digit ;
    add this digit to the representation
  UNTIL
    no more digits
  ENDREP .
```

```
isolate a digit :
  INT CONST digit ::  number MOD radix ;
  number := number DIV radix .

add this digit to the representation :
  representation := representation of digit + representation .

no more digits :
  number = 0 .

representation of digit :
   IF digit > 15
   THEN "?"
   ELSE "0123456789ABCDEF" SUB ( digit + 1 )
   FI .

print representation :
    put ( "Representation = " ) ;
    put ( representation ) ;
    line .
```

3.5 Guessing numbers

The following example is due to Leo Geurts (Centrum voor Wiskunde en Informatica, Amsterdam). By means of the halving algorithm it guesses a number chosen by the human opponent.

```
how to play :
  tell the rules ;
  all numbers 0 to 99 are possible ;
  REP
    make a guess ;
    wait for reply ;
    halve the range
  UNTIL
    good OR impossible
  ENDREP ;
  tell result .

tell the rules :
  put ( "Think of a number from 0 to 99." ) .

wait for reply :
  TEXT VAR reply ;
  get ( reply ) ;
  WHILE reply <> "y" AND reply <> "h" AND reply <> "l"
  REP
    line ;
    put ( "Possible answers are y(es), h(igh), l(ow)" ) ;
    get ( reply )
  ENDREP .
```

```
halve the range :
  IF reply = "l"
  THEN
    min := guess + 1
  ELIF reply = "h"
  THEN
    max := guess - 1
  FI .

good :
  reply = "y" .

impossible :
  min > max .

all numbers 0 to 99 are possible :
  INT VAR min ::  0 ;
  INT VAR max ::  99 .

make a guess :
  INT CONST guess ::  ( min + max ) DIV 2 ;
  line ;
  put ( "Is it" ) ;
  put ( guess ) ;
  put ( "?" ) .

 tell result :
  line ;
  IF good
  THEN
    put ( "Good!" )
  ELSE
    put ( "Cheat!" )
  FI .
```

3.6 Guess my age

This example is also due to Leo Geurts. This time the player has to guess a randomly chosen "age" It is a good opportunity to learn, by experiment, the value of the halving algorithm.

```
how to guess :
  choose a number ;
  look for a customer ;
  let him guess ;
  WHILE not guessed
  REP
    give a hint ;
    another guess
  ENDREP ;
  applause .
```

```
choose a number :
  INT VAR number ::  choose128 .

look for a customer :
  put ( "Guess my age " ) .

let him guess :
  INT VAR guess ;
  get ( guess ) .

not guessed :
  guess <> number .

give a hint :
  line ;
  IF guess > number
  THEN
    put ( "Too high, try again " )
  ELSE
    put ( "Too low, try again  " )
  FI .

another guess :
  get ( guess ) .

applause :
  line ;
  put ( "Correct!" ) ;
  line ( 3 ) .
```

3.7 Karel the Robot

In an introductory programming course, it is good practice to make use of a restricted and protected programming environment within which the pupils can develop their first programs. We will describe here Karel the Robot, a didactic aid based on

> R.E. Pattis,
> Karel the Robot,
> a gentle introduction to the art of programming,
> John Wiley and sons, 1981.

Its purpose is to confront a pupil with the essential ingredients of programming languages: actions, tests, control structures and refinements, without having to introduce such complicated concepts as variables, objects, values and types.

With the Elan Programming Environment, a *standard packet* for Karel the Robot is included.

3.7.1 The world of Karel

Karel spends his life on the corners of streets and avenues that run vertically and horizontally over the screen. The street corners form a rectangular grid, whose size is chosen such that it just fits on the screen.

On some of the corners stand walls so that Karel cannot pass. On others may lie beepers that Karel can pick up. He can also mark a corner by dropping a beeper. Karel may walk over his world, but of course he should not leave the screen.

3.7.2 Karel's actions

After reading the standard packet `karel` by means of the packet-command (`p`) he can perform the following actions:

- `start karel`
  Sets up the world, with Karel in the lower left corner with his nose pointing to the north;
- `turn right`
  Karel makes a right turn;
- `turn left`
  Karel makes a left turn;
- `move`
  Karel moves one corner forward in the direction of his nose;
- `take beeper`
  Karel takes the beeper (if any) from the present corner and puts it into his pocket;
- `drop beeper`
  Karel takes a beeper from the limitless supply in his pocket and leaves it at the present corner;
- `stop`
  Makes an end to Karel and his world, lifting him from the screen.

3.7.3 Karel's conditions

Karel knows the following conditions:

- `facing north`
  True only if Karel's nose presently points to the north;
- `facing south`
  Similarly;
- `facing east`
  Similarly;
- `facing west`
  Similarly;
- `facing wall`
  True only if the next corner contains a wall;
- `beeper`
  True only if there is a beeper lying at the present corner.

3.7.4 The basic Karel-packet

The standard packet `karel`, written in Elan-0, realizes the actions and tests just described. As a first example we show a program that makes Karel trundle around a square of 16 * 16. It should be entered after the packet `karel` has been read (p).

```
draw square :
  start karel ;
  UPTO 4
  REP draw side
  ENDREP ;
  stop .

draw side :
  UPTO 16
  REP drop beeper ;
      move
  ENDREP ;
  turn right .
```

3.7.5 Karel's environment

The packet environment can be used to decorate Karel's world with a house, a stair or a labyrinth. The program will for that purpose have to call one of the actions

- `make labyrinth`;
- `make stair`;
- `make house`.

The program which follows simulates Karel fetching his morning paper. It makes use of both the packet `karel` and the preceding environment packet. It is used as an introductory example in the textbook (see [2]).

```
program :
  start karel ;
  make house ;
  walk to garden ;
  take newspaper ;
  walk back to bed .

walk to garden :
  walk along wall ;
  walk through entrance ;
  walk to newspaper .

take newspaper :
  take beeper .

walk back to bed :
  walk back to entrance ;
  walk back through entrance ;
  walk back along wall .
```

```
walk along wall :
  turn right ;
  move ;
  move ;
  move ;
  move ;
  move .

walk to newspaper :
  turn left ;
  move ;
  move ;
  turn left ;
  move ;
  move ;
  turn right ;
  move .

walk back to entrance :
  turn about ;
  move ;
  turn left ;
  move ;
  move ;
  turn right ;
  move ;
  move .

walk back through entrance :
  turn right ;
  move ;
  move ;
  move ;
  move .

walk back along wall :
  turn right ;
  move ;
  move ;
  move ;
  move ;
  move .

turn about :
  turn right ;
  turn right .

walk through entrance :
  turn left ;
  move ;
  move ;
  move ;
  move .
```

Chapter 4

AN OVERVIEW OF ELAN-1

Elan-1 is a sublanguage of standard Elan. It comprises full Elan apart from the packet-mechanism. It strives to be compatible as much as is possible with the EUMEL implementation of Elan, incorporating a number of minor language modifications that have been mutually agreed upon. The EUMEL concept of DATASPACE is not implemented.

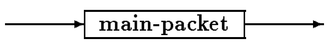
In this overview we follow the syntax of the language, introducing Elan terminology for the most important constituents of a program and making some remarks about the syntax of the various constructs. The more noticeable differences with standard Elan and EUMEL are indicated.

This chapter is not an Elan course and certainly not a course in programming! It serves merely as a convenient overview of the language. In this chapter the names of syntactic constructs will be written in italic.

4.1 Programs

A program in the subset Elan-1 is an Elan *program* consisting of one *main-packet* only.

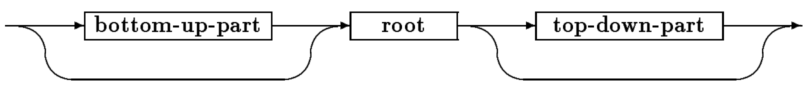
program



We assume the reader to be familiar with the notation of syntax-diagrams. We use those diagrams not only for describing structure but also for introducing useful terminology. For didactic reasons, there will be a large number of relatively simple syntax-diagrams rather than a small number of complicated ones.

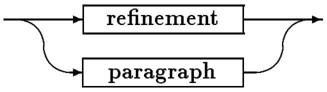
The structure of the *main-packet* is somewhat simpler than it is defined by the standard.

main-packet



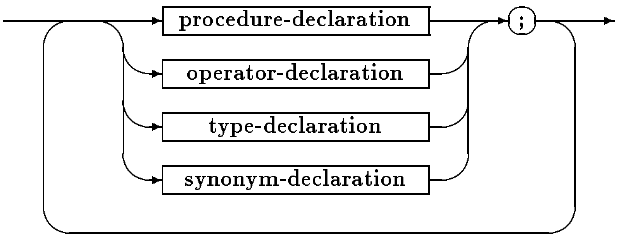
The *root* of the *packet* is either (as in standard Elan) a *paragraph* or (as in Elan-0) a *refinement.* The effect of the execution of an Elan-1 program is the effect of the execution of its *bottom-up-part* followed by the execution of its *root*, which may in its turn involve the execution of some *refinements* in the *top-down-part.*

root



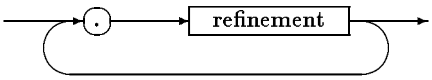
If the *root* of the program is a *paragraph*, it is treated by the Elan Programming Environment as an unnamed *refinement*, which obtains the name `program`.

bottom-up-part



The *bottom-up-part* comprises those *declarations* which precede the *root* and can be considered as a language fundament on which the rest of the program is to be built.

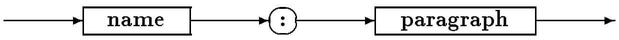
top-down-part



The *top-down-part* comprises the *refinements* that can be invoked from the *root.* The *refinements* in the *top-down-part* can not be invoked from a *declaration* in the *bottom-up-part.*

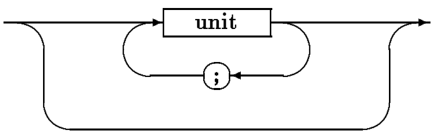
The Elan Programming Environment accepts as input any sequence of *declarations*, other *units* and *refinements* and distributes those elements over the three components of the *main-packet* in their order of arrival. Therefore the layout and order of input to the Elan Programming Environment can be in a very relaxed syntactic style. In outputting the program the Elan Programming Environment imparts its own strict layout conventions and structure to the program.

refinement



A *refinement* gives a *name* to a *paragraph.* It is invoked by executing its *name.* It is not a *declaration* in the proper sense, since it may be invoked (e.g. from the *root*) before it has been executed. The meaning (effect and value) of a *refinement* invocation is the same as that of its *paragraph.*

paragraph



The effect of the execution of a *paragraph* is the effect of the sequential execution of its *units* in textual order. The value (if any) of a *paragraph* is the value of its last *unit.* Notice that a *paragraph* may be empty. In that case it has neither an effect nor a value.

4.2 Declarations

Declarations are constructs that, upon execution, bind a *name* to a value (*object-declaration*), to a declarer (*type-declaration*) or to a routine (*procedure-declaration*, *operator-declaration*).

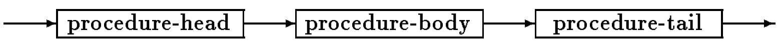
Apart from this dynamic effect they have a static effect. The occurrence of a declaration makes the name, defined by it, visible in a specific part of the program, the *scope* of the declaration, together with the type and access attributes with which it is defined.

4.2.1 Bottom-Up declarations

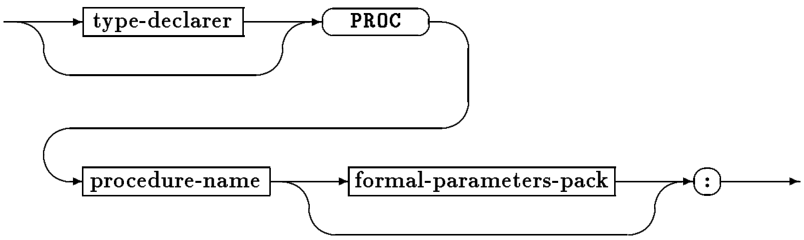
We will now describe those declarations that occur in the *bottom-up-part* of a program.

4.2.1.1 Procedure-declarations

procedure-declaration

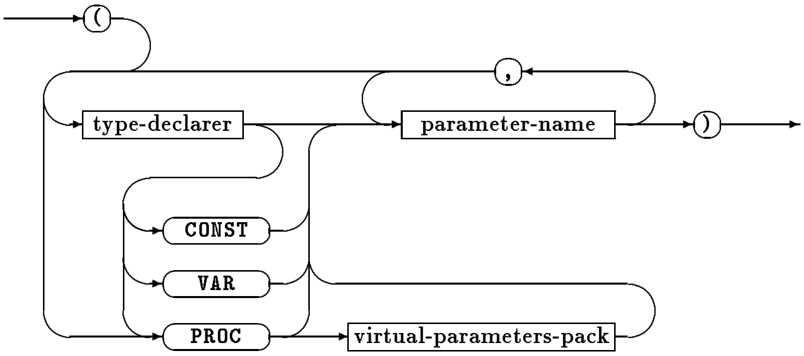


procedure-head



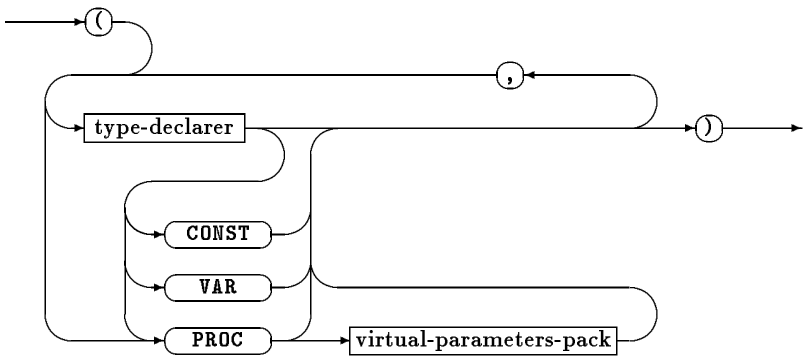
The head gives the *procedure-name*, the types and access attributes of its parameters (if any) and the type of its result (if any).

formal-parameters-pack



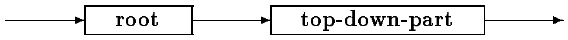
Parameters are either objects or procedures, as can be deduced from this rather unwieldy syntax diagram. In case a procedure is to be passed as a parameter, the types of its parameters are in their turn to be specified precisely by means of a *virtual-parameters-pack*.

virtual-parameters-pack

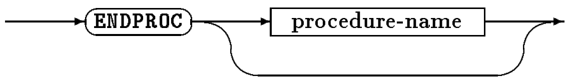


A *procedure-body* has the same form as a *main-packet* without a *bottom-up-part*.

procedure-body



procedure-tail

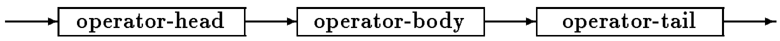


The name of the procedure is repeated after the `ENDPROC` symbol. In the Elan Programming Environment this name may be left out (in which case it is inserted automatically).

4.2.1.2 Operator-declarations

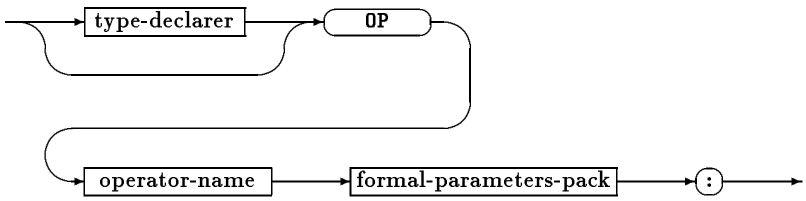
An *operator-declaration* follows the pattern of a *procedure-declaration*.

operator-declaration



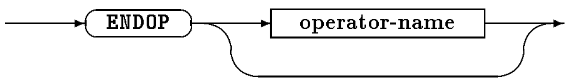
Only its head and tail are somewhat different.

operator-head



An *operator-head* must contain either one (for *monadic-operators*) or two (for *dyadic-operators*) parameters.

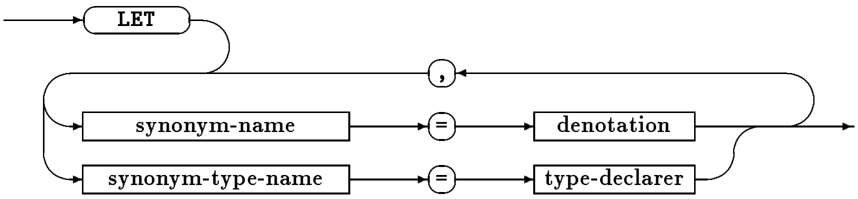
operator-tail



4.2.1.3 Synonym-declarations

Synonym-declarations bind either a *name* to a *denotation* (making that name synonymous with that *denotation*) or a *type-name* to a *type-declarer* (making that *type-name* synonymous to that *type-declarer*).

synonym-declaration

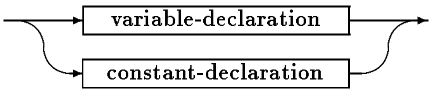


The synonym has all the properties of the *denotation* or *type-declarer* it is bound to - it may be used at all places where that would be valid, with the same meaning.

4.2.2 Top-Down declarations

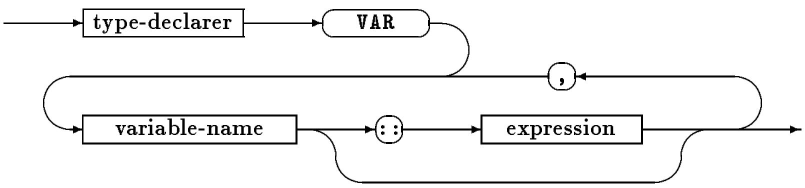
Apart from the Bottom-Up declarations already described, Elan has Top-Down declarations for declaring objects. Every object in an Elan program has a *name*, a *type*, an *access attribute* (`VAR` or `CONST`) and (during execution) a *value*, which may be undefined. There are two forms of *object-declaration*:

object-declaration



A *variable-declaration* declares one or more objects (*variables*) with the access attribute `VAR` of one same type.

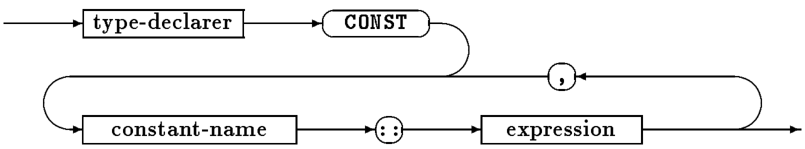
variable-declaration



The *expression* preceded by the curious sign :: is the *initialization* for the variable. Its value (or, if there is no initialization, an undefined value) is assigned to the *variable-name*.

A *constant* is an object with the access attribute `CONST` and always obtains a well defined value at its declaration.

constant-declaration

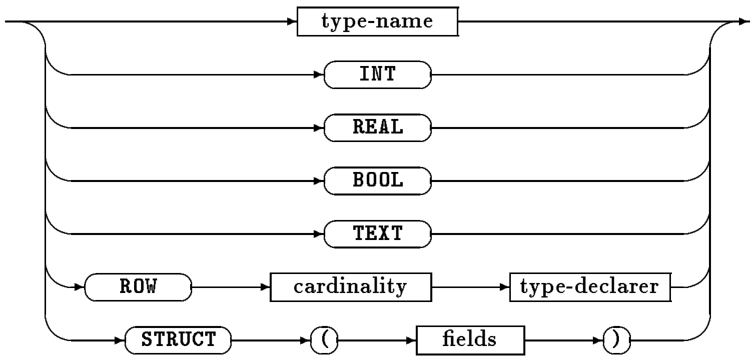


A variable can obtain another value assignation, whereas a constant can not be assigned to. If so desired, the access attribute `CONST` may be left out, but this is not advised since it leads to degradation of syntactic error-detection. An *object-declaration* may very well occur in a *repetition*. Each time the declaration is executed, its initialization is executed anew.

4.2.3 Declarers and the type-system

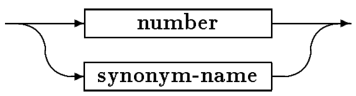
Elan has a constructive type system based on four concrete elementary types and two mechanisms for composing types, and gives to its users the opportunity to add further (abstract) types.

type-declarer



4.2.3.1 Rows

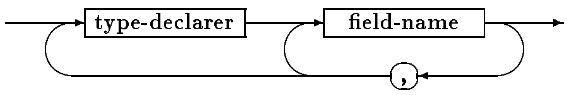
cardinality



The elements of a row all have the same type, and are numbered from 1 up to the *cardinality* of the declarer, which must be either a *number* or a synonym for a *number*. This *cardinality* is therefore fixed, it can not be the result of an *expression* other than a *number*.

4.2.3.2 Structures

fields



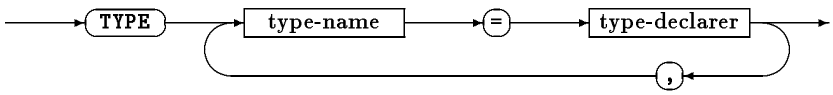
The *fields* of a structure may have different types, and each field is tagged by a (different) *field-name*.

4.2.4 Type-declarations

An abstract type can be introduced either by a *synonym-declaration*, in which case the *type-name* merely stands for the *type-declarer* in its definition, or by a *type-declaration* (see following section), in which case it is to be distinguished from the *type-declarer* in its definition. In both cases it is represented by a (bold) *type-name*.

A *type-declaration* gives a *type-name* to a *type-declarer*, its *realization*. In the packet in which the abstract type is defined, the *concretizer* `CONCR` may be used to convert an object of that type to the type of the realization. In that packet, *selection* and *subscription* also have access to the realization of a type. Outside its defining packet, an abstract type is elementary and there is no direct way to access its realization (but this may be achieved indirectly, by the use of algorithms defined in the same packet as the type). Because of these stringent restrictions on the visibility of the realization, *type-declarations* are much more suitable for Bottom-Up and modular programming than the *synonym-declarations*, which are intended for Top-Down programming.

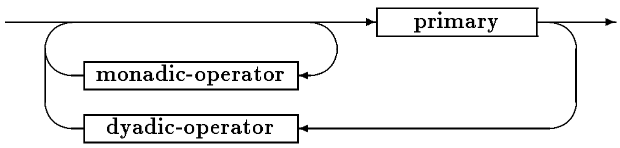
type-declaration



4.3 Expressions and their constituents

Expressions can be composed by various mechanisms (*calls*, *operators* and *control-structures*) out of smaller constructs (the *primaries*). *Expressions* may either yield a value of some type, or they may be actions, which yield no value (and have a hypothetical type `VOID`).

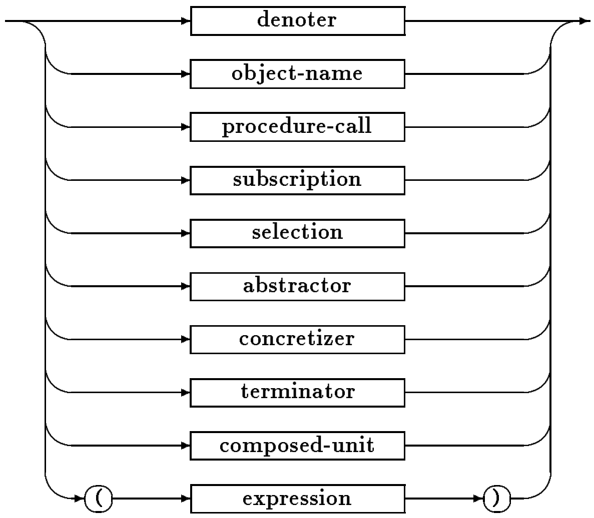
expression



The *dyadic-operators* have the following priorities:

1	`:=, INCR, DECR, CAT`
2	all abstract dyadic operators
3	`OR, XOR`
4	`AND`
5	`=, <>`
6	`<=, <, >=, >`
7	`+, -`
8	`*, /, DIV, MOD`
9	`**`
10	all monadic operators

primary

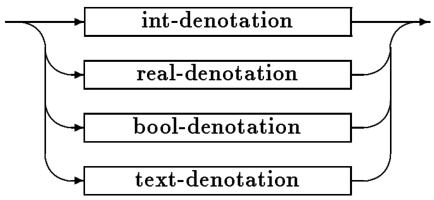


These constructs are explained in the following sections.

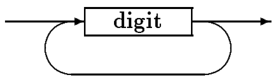
4.3.1 Denoters

Denoters serve to denote a value of a concrete type.

denoter

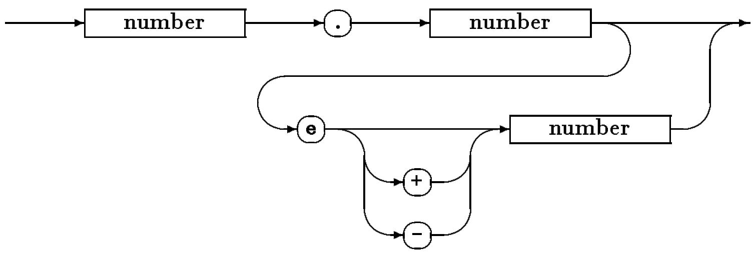


int-denotation, number

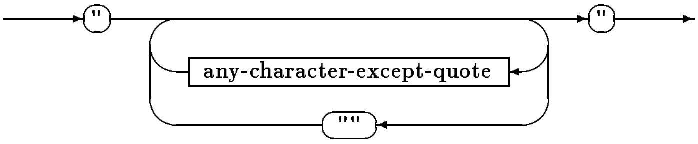


Notice that the *int-denotation* denotes an unsigned number, which must be preceded by a monadic minus to obtain a negative number.

real-denotation

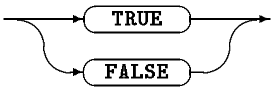


text-denotation



A *text-denotation* denotes the sequence of characters, obtained by stripping off its outermost quotes. Most characters, including the space, stand for themselves, but the quote-character is used as an escape.

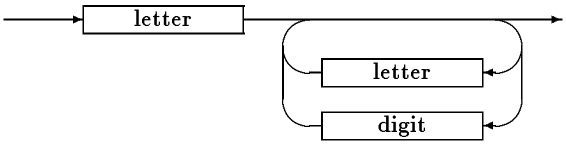
bool-denotation



4.3.2 Names

Names of objects and algorithms are written in small (lower case) *letters* possibly followed by some further small *letters* or *digits.*

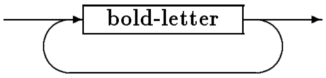
name



Within a *name*, spaces may be used freely to enhance readability. Whereas in Elan such spaces are redundant and do not form part of the *name*, in the Elan Programming Environment single embedded spaces are considered part of the *name*. All occurrences of one same *name* must have the same layout.

Types and operators have *bold-names*, written in capital letters.

bold-name, type-name

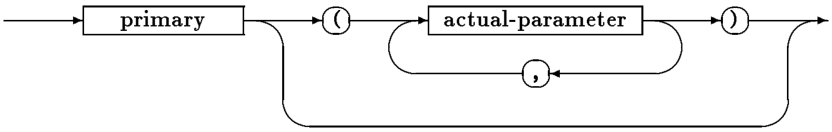


Spaces are not allowed in a *bold-name*. A number of operators have special symbols as names, such as + etc. These are listed in section 4.3.

4.3.3 Calls

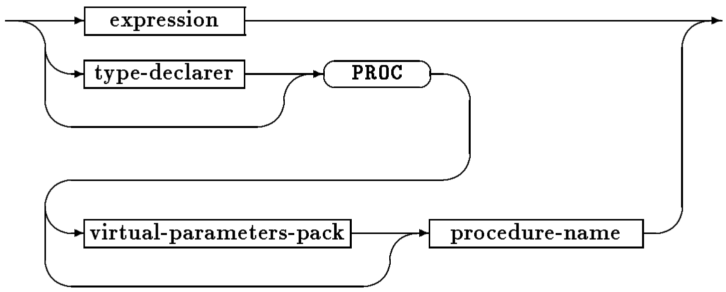
The *call* of a procedure with parameters is written in the usual prefix style.

procedure-call



The *primary* must yield a procedure. The *actual-parameters* must agree in number and type with the heading of one of the definitions of that procedure. Notice that a procedure without parameters is called by mentioning its name, without parameters or brackets.

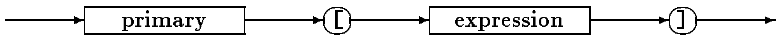
actual-parameter



4.3.4 Subscriptions

Elan has only one-dimensional arrays (rows) of a size (*cardinality*) fixed at compile time. A row can be subscripted to obtain one of its elements (which in its turn may be a row, a structure or a simple value). A multidimensional array can be realized as a row of rows.

subscription

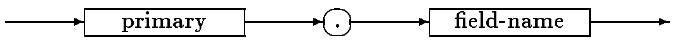


The *primary* must yield a row. The *expression* must yield an integer, whose value lies between 1 and the *cardinality* of that row. The type of a *subscription* is the type of the element yielded. The *subscription* inherits the access attribute of the *primary*. This implies that an assignation to a *subscription* is possible only if the *primary* is a variable.

4.3.5 Selections

A structure can be selected from, to obtain one of its *fields* (which in its turn may be a row, a structure or a simple value).

selection

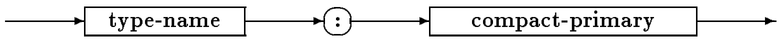


The *primary* must yield a structure, one of whose *fields* has the *field-name* as name. The type of the *selection* is the type of the field yielded. As is the case for the *subscription*, the *selection* inherits the access attribute of its *primary*.

4.3.6 Abstractors

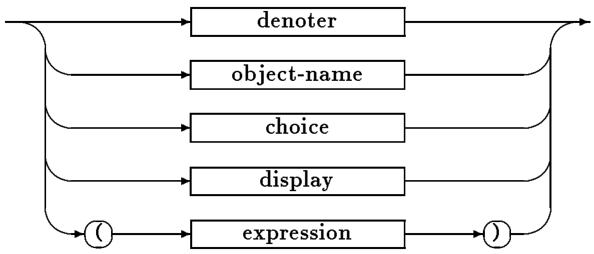
The *abstractor* serves to denote values of an abstract type. It will often be used together with a *display*, to abstract from a composed realization to an abstract object.

abstractor



The part after the colon (a *compact-primary*) must have the type of the realization of the *type-name*.

compact-primary

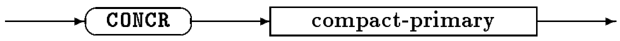


The value of an *abstractor* is the value of its *compact-primary.* The type of an *abstractor* is the type of the *type-name*, rather than its realization. Since the realization of a type is visible only in its defining packet, an *abstractor* with a specific *type-name* can only be used in the packet defining that *type-name.*

4.3.7 Concretizers

A *concretizer* serves to break the abstraction of a value.

concretizer

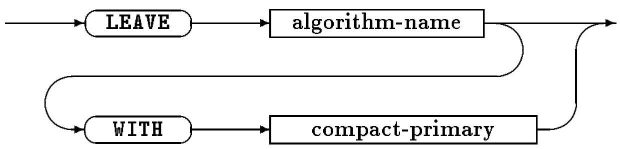


The *compact-primary* must be of some abstract type; the *concretizer* is then of the type of its realization. It yields the value of its *compact-primary* and inherits the access attribute of its *compact-primary.* A *concretizer* can only be used in an environment where the realization of its abstract type is visible (see 4.2.4).

4.3.8 Terminators

A *terminator* serves to terminate a *refinement* or a procedure.

terminator

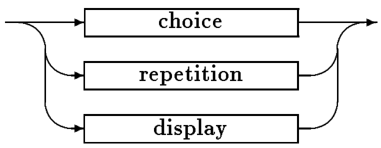


The *algorithm-name* must either be the name of the directly surrounding procedure or of some visible *refinement*, of whose execution the *terminator* forms part. If a with-part is given, the algorithm named is left yielding the value of the *compact-primary*, otherwise it is left yielding no value, and has the (hypothetical) type VOID.

4.4 Composed-units

The *control-structures* of Elan are the *choice*, *repetition* and *display.*

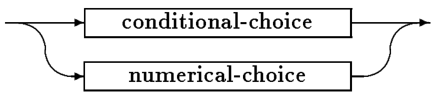
composed-unit, control-structure



4.4.1 Choice

The *choice* is a construct for choosing between a number of alternative algorithms. There are two forms of *choice*.

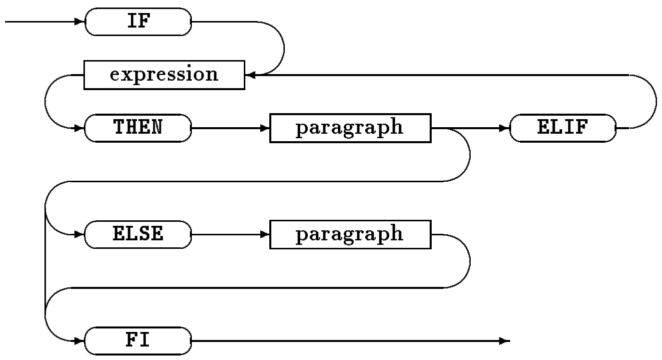
choice



4.4.1.1 Conditional-choice

The *conditional-choice* chooses between two or more algorithms on the basis of one or more boolean *expressions* (conditions).

conditional-choice

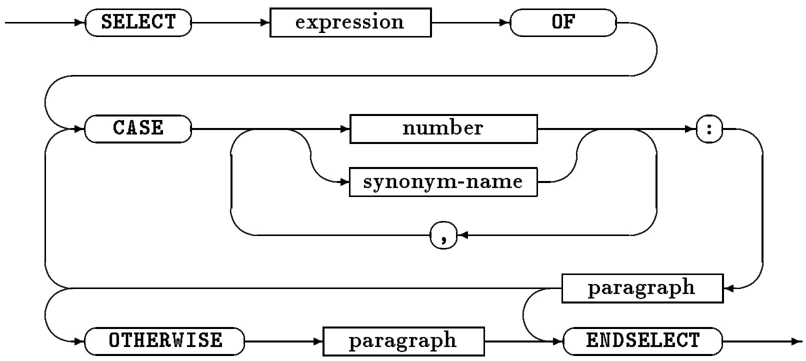


The type of the *conditional-choice* is the type of the *paragraph* executed. Therefore all *paragraphs* in the *conditional-choice* must have the same type.

4.4.1.2 Numerical-choice

The *numerical-choice* is a somewhat baroque construct for choosing between a number of cases on the basis of an integer.

numerical-choice

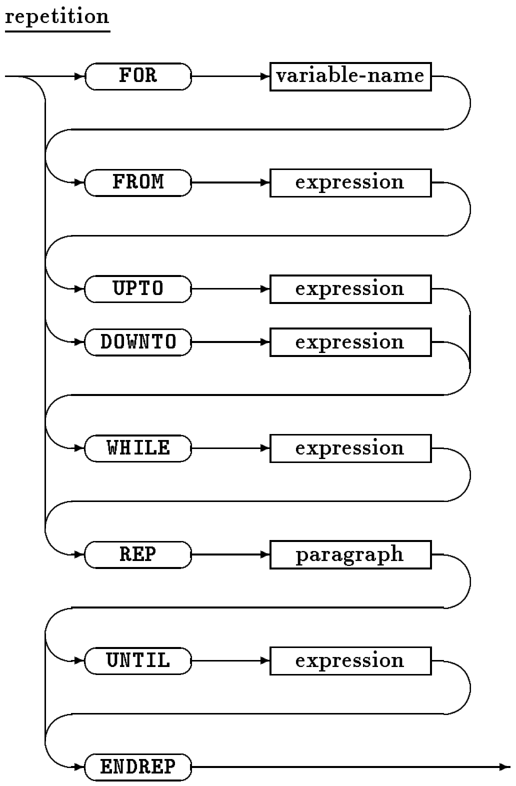


The cases must be labeled either with a *number* or with a synonym for a *number*, and therefore have a value fixed at compile-time. The labels of the cases must all be different and need not be ordered.

The *numerical-choice* is executed by first computing the value of the *expression*, which must yield an integer, and then executing the *paragraph* whose case label has that value, if any. If there is no *paragraph* labeled with this value, the otherwise-part (if any) is executed. The type of the *numerical-choice* is the type of the *paragraph* executed. Therefore all *paragraphs* in the *numerical-choice* must have the same type.

4.4.2 Repetition

There is one, at first glance rather huge construct, which serves to express *repetitions*. Its purpose is to repeat a *paragraph* under various circumstances.



Since many of its constituents are optional, it can be tuned to different applications.

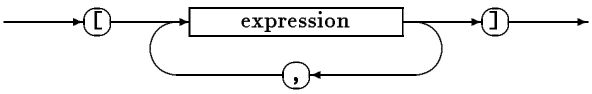
The *expressions* after `WHILE` and `UNTIL` are conditions, and therefore have to be boolean. The other *expressions*, as well as the variable after `FOR` (*controlled variable*) have to be integer. The *repetition* does not yield a value as result (it is an action).

The semantics of the *repetition* is quite conventional. The for-part can be omitted if the controlled variable does not occur in the body of the *repetition*. The constituents `FROM 1`, `UPTO maxint`, `DOWNTO minint`, `WHILE true` and `UNTIL false` can be omitted (i.e. are assumed if the corresponding constituents is omitted).

4.4.3 Display

The *display* is a construct serving to denote values for composed types (structures and rows) and, in conjunction with the *abstractor*, for abstract types.

display



If the *display* is used to denote a row, the number of its constituent *expressions* has to be equal to the *cardinality* of the row, whereas each *expression* has to be of the type of the elements of the row.

If the *display* is used to denote a structure, the number of its constituent *expressions* has to be the same as the number of *fields* of the structure, whereas each *expression* has to be of the type of the corresponding field of the structure.

The *display* is executed by executing its constituent *expressions* collaterally, and composing a composed object from the values in their textual order.

Chapter 5

THE ELAN PROGRAMMING ENVIRONMENT

The user interface of the Elan Programming Environment is by itself rather simple, but its description gets rather complicated because of the many ways of interaction with the user. For concise description, we will denote the "return" or "enter" key by <RET> and the "delete" or "rubout" key by <DEL>, the "backspace" key by <DEL LEFT>. The <BREAK> is a special key to notify the Elan Programming Environment to stop whatever it is doing. Its realization may well be different for the various implementations (CNTL-C, BREAK-key, special buttons, etc.; read the documentation which is distributed on the floppy disc with the software). In screen output, the cursor is depicted as ■.

5.1 Components of the Elan Programming Environment

The Elan Programming Environment conceptually consists of six functional modules, communicating by way of a common data structure (the *memory*) and (in some cases recursively) invoking each other's services.

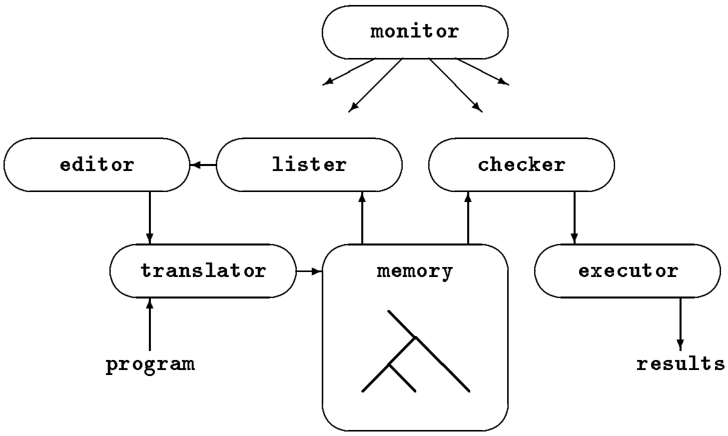
5.1.1 Modules of the Elan Programming Environment

The modules of the Elan Programming Environment are:

- The *monitor*, acting as a command interpreter, provides most of the user interface and invokes all other modules.
- The *translator* accepts input of definitions, translates them to internal form and stores them into memory.
- The *lister* provides a structured overview of the defined names and their definitions.
- The *editor* takes care of all text input, for example when entering or editing a definition.
- The *checker* performs the checking of some static properties of the program prior to its execution.
- The *executor* executes checked programs.

The relationship between these modules is displayed in the following figure:

<u>modules</u>



5.1.2 The memory

The *memory* holds the following information:

- a table of all names with their attributes (the *name list*),
- the *stack*, the *heap* and
- the definitions forming the program, in some internal form.

Unfortunately, present-day computers do not retain the contents of their memory upon being switched off, so that the memory must be explicitly written to a file in order to be able to read it back later.

During execution the executor tries to recycle all space in the heap taken up by objects that are no longer needed. It may however happen that a program needs more space than is available. Thus the memory can get exhausted; the only way out is to save the program in a file (by means of a write-command) and to start again.

5.1.3 File store

The Elan Programming Environment was designed to run under any operating system on a variety of compilers. It uses the file system of the host computer for storing files, adhering to the conventions of that system.

5.2 The user interface

5.2.1 Moods

From the standpoint of the user, the Elan Programming Environment can be in a number of *moods* (states), each with its own behaviour. In each of those moods, the Elan Programming Environment accepts a number of single-letter *commands*. These moods, with their associated commands are:

- command-mood
  `c d e f l p q r s t v w x` (see 5.3)
- edit-mood
  `<RET> <BREAK>` (see 5.4)

- execute-mood
  `<BREAK>` (see 5.5)
- backtrace-mood
  `e n q s` (see 5.5.2)
- trace-mood
  `b c e n p q s x <BREAK>` (see 5.5.3)
- verify-mood
  `b c e n p q s x <BREAK>` (see 5.5.4)

The current mood is indicated by a message in the right-hand corner of the bottom line of the screen, the *status line*. The left-hand corner of the status line is used for error messages.

Each mood has its own set of possible commands, some of which will cause the Elan Programming Environment to switch into another mood.

5.2.2 Commands

Commands consist of single letters. On some computers they may be realized as special function keys or as choices from a menu. Commands are not echoed on the screen, and are obeyed immediately, without waiting for a `<RET>` key to be pressed. This contributes to the interactive feeling of the Elan Programming Environment.

To an unknown or unacceptable command the Elan Programming Environment reacts by showing for about one second a menu of all acceptable commands (we use the *negative menu* technique: a menu is displayed only when an unacceptable command is given). In all moods except edit-mood and execute-mood the letter `h` (help-command) can be used to provoke such a menu.

5.2.3 Names

The Elan Programming Environment knows at any time a collection of names, each of which may be *undefined*, or have one or more definitions. Under the guidance and control of the Elan Programming Environment names can be manipulated and definitions supplied to form a program. All names which are known to the Elan Programming Environment are stored in the *name list*.

A name with which one or more definitions associated have been we will call a *defined* name. Depending on the way in which their definitions are brought into the Elan Programming Environment these are either *visible* or *hidden*. Definitions from packets (for example the standard Elan Library) are hidden, the others are visible. In an overview of known names the Elan Programming Environment will show only those names which have associated a visible definition.

Names which do not have any associated definition are called *undefined*. Once a name is mentioned to the Elan Programming Environment it will remain in the name list even when it is undefined. It is important to know this when using abbreviated names (see later).

Some names, as, for example, the keywords of Elan. Look like valid names, but no definition can be associated with them. These are called *undefinable* .

5.2.4 Focus and prompt

There is always one name, called the focus, which is the current center of interest. After the completion of each command, the Elan Programming Environment prompts with the focus, indicating its readiness to accept further commands. The focus is extended by a type indication, indicating whether there exists a definition associated with the name, and if so, what kind of definition (refinement, `PROC`edure, `OP`erator, `LET` or `TYPE`).

Undefined names are recognized by a question mark:

```
program ?■
```

or

```
RESL ?■
```

Refinement names have a colon mark:

```
take next:■
```

The names of procedures and operators are *generic*, i.e. can have more than one definition. The headings of all definitions of the name are shown. Procedures and operators are indicated like:

```
PROC put (INT a):■
PROC put (REAL a):
PROC put (TEXT a):
```

or

```
TEXT OP CAT (TEXT CONST a):■
```

LET definitions for values and types have a let mark:

```
LET n■
```

or

```
LET VECTOR■
```

TYPE definitions are displayed like:

```
TYPE T■
```

Initially, the focus is the name program, being as yet undefined. Thus, the initial prompt looks like:

```
program ?■
```

Another name can be chosen as focus by the focus-command (see 5.3).

Upon focussing on a generic name the cursor appears at the heading of the first definition. By (repeatedly) giving a next-command (n), it is possible to navigate over the various definitions of a generic algorithms, e.g. (after one n):

```
PROC put (INT a):
PROC put (REAL a):■
PROC put (TEXT a):
```

It is now the second definition which is the focus for e.g. an edit- or show-command.

5.2.5 Program and root

To the Elan Programming Environment, a program does not consist of one sequential text but rather of a number of independent definitions, whose order plays no role. In order to construct a program, its definitions have to be input one by one. Some commands (the check-command and the execute-command) require the focus to be a refinement, taken as a description of the composition of the program. This refinement is called the *root* of the program. By default the Elan Programming Environment assumes the first refinement entered or read from a file to be the name of the root of the program.

5.2.6 Initial state

In the initial state the Elan Programming Environment is in command-mood and its memory contains only the definitions of the standard Elan Library. On the screen can be seen an identification of the Elan Programming Environment. The initial prompt indicates that the focus is `program`, as yet undefined, which serves as a suitable root for many programs.

5.3 The command-mood

The command-mood can be recognized by the message `Command, please...` on the right-hand side of the status line. In command-mood, the commands listed below are accepted. Most of them have the focus as an implicit parameter. After the execution of each command the Elan Programming Environment prompts with the current focus and returns to the command-mood.

- `f` focus-command

  The focus-command shows `Focussing...` on the status line, and asks for the name which should become the focus of interest. So the focus-command can be used to focus on a new name, thus allowing navigation over program parts.

  The focus-command allows a defined name to be abbreviated by replacing its last characters by a star (e.g. `prog*`). The Elan Programming Environment then searches the name list for a match, and complains `Known name ?` if no match is possible.

- `e` edit-command

  The edit-command serves to assign a new definition to the focus, or to modify the definition of the focus.

  The name and (if present) the body of the definition whose name is the focus are shown on the screen and an opportunity is given for local editing. On the status line `Input, please...` is displayed (see 5.4 for a description of the edit-mood).

- `s` show-command

  The definition of the focus is shown on the screen. In case the focus is generic, the definition indicated by the cursor is shown (the cursor can be moved to another definition of focus by the next-command `n`). Immediately after the execution of a program it is also possible to focus on an object created during the execution; in that case its type, access attribute, name and value can be inspected by the show-command.

- `n` next-command

  The next-command serves to move the cursor over the various definitions of a generic name. It has no effect for non-generic names.

- `x` execute-command

  The execute-command serves to execute the program whose root is the focus. It shows `Executing...` on the status line. After execution of a program, its objects (variables and constants) can be inspected by means of the show-command. At the start of an execution, all objects are made undefined. See section 5.5 for a description of the execute-mood.

- `t` trace-command

  Same as the execute-command, but the execution takes place in trace-mood. This mood is recognized by the request `Trace command, please...` on the status line. See section 5.5 for a further description of the trace-mood.

- `v` verify-command

  Special form of tracing execute-command, in which the types of the syntactical constructs are shown rather than their values. This mode is recognized by the request `Verify command, please...` on the status line. See section 5.5 for a further description of the verify-mood.

- `l` list-command

  The list-command serves to give a overview of all names having a visible definition. This overview begins with a bottom-up part (a list of all names with type `LET`, `PROC`, `OP` or `TYPE`).

  After this a structured overview of the refinements of the program is given. If the focus is a refinement, the names of all refinements which appear in its body are displayed with a suitable indentation on the next line(s). The process is invoked recursively if a refinement is expressed in terms of other refinements, with the exception that no name appears more than once in this overview. As a consequence the overview gives an indication about the structure of the focus, considered as root of a program.

  After this overview a list is given of all refinements which remain, and so are no part of the program having the focus as root. Some of them might be candidates for deletion.

  Names from the built-in library (see appendix A) and from a packet are not listed. It is however possible to focus on a library name and show its definition.

- `d` delete-command

  The delete-command deletes the definitions of the current focus.

  The Elan Programming Environment asks for confirmation (to be answered with `y` or `n`) before doing anything so drastic. It gives a warning before deleting definitions from the standard library.

- `r` read-command

  The read-command serves to read a program, which was previously written to a file, back into memory. It reports `Reading...` on the status line. It adds the definitions of that program to the current contents of the memory, overwriting refinements with the same name. It may be necessary to first save the current program by writing it to a file (`w`), or to first clear the memory (`c`). The read-command asks for the name of a file. If a file with that name is present, the program it contains is read. After reading, `program` will be the new focus. During the reading of the program, the names of the definitions are listed on the display. Finally a structured overview is given of all definitions.

  If a file with the desired name was not present the Elan Programming Environment produces an overview of all program files that are present (again a negative menu).

  The conventions for file names are those of the operating system running the Elan Programming Environment.

- `p` packet-command

  The packet-command also serves to read a program into memory. It reports `Reading...` on the status line. It is a variant of the read-command, with the following differences:

  - Definitions read by the packet-command are hidden: they are not mentioned by the list-command. They can be inspected by the show-command. However, they can not be modified by an edit-command, with the exception of (generic) procedures and operators.
  - In writing a program consisting wholly or partly of hidden definitions, only the visible definitions are written.

- `w` write-command

  The write-command serves to write all visible definitions from memory with the root as the first refinement onto a file. It reports `Writing...` on the status line. It asks the user for a file name. Any existing file of that name is overwritten. Hidden definitions (see the packet-command) are not written.

  The conventions for file names are those of the operating system running the Elan Programming Environment.

- `c` clear-command

  The clear-command, upon confirmation, clears the memory, bringing the Elan Programming Environment back into its initial state. It reports `Clearing...` on the status line.

- `q` quit-command

  The quit-command, upon confirmation, halts the Elan Programming Environment and returns control to the operating system of the computer.

5.4 Reading and editing of programs

While reading or editing a program, the Elan Programming Environment accepts text from a file or from the keyboard, one definition at a time, and translates it to internal form, performing a running Context-Free syntax check as the lines come in.

Upon finding a syntactic error, the translator attempts to arrive at a syntactically correct program in interaction with the user, offering the incorrect definition for local editing as often as necessary. By hitting `<BREAK>` the user may end this attempt.

The editing of existing parts of a program is based on replay: some definition is displayed on the screen, and can be modified at will by the user, after which the translator reads the text on the screen as input. The new definition may overwrite the old one.

5.4.1 Local editing

Local editing is performed in edit-mood, which is recognized by `Input, please...` on the status line. Local editing allows the user to enter and modify a text field (which may well be longer than a line on the screen). Local editing is ended by hitting `<RET>` (or aborted by hitting `<BREAK>`). Up to this event, the text field can be modified by modifying its image on the screen.

During local editing, the cursor can be moved freely over the screen within specific boundaries. Assume zero or more lines have been entered and the cursor is somewhere within this field.

- By means of `<RET>` the user indicates that this line of input is completed; the translator now starts processing it. After processing a complete definition the translator returns to command-mood, otherwise it recursively asks the editor for a next line of input.

- With the `<BREAK>` key the text modification or entering can be aborted in such a way that it has no effect.
- The `<DEL>` key deletes the character under the cursor, if any.
- The `<DEL LEFT>` key deletes the character to the left of the cursor, if any.
- The arrow keys allow the movement of the cursor over the text field (but not outside it).
- Any key which is not a control key inserts the corresponding character at the position of the cursor, and moves the cursor right.

5.4.2 Incremental correction

If the translator is of the opinion that a specific symbol was omitted or if it expects a specific syntactic construction (identifier; attribute `VAR` or `CONST`; unit or declaration ...) it gives an error message to that effect in the lower left corner of the screen and offers the directly surrounding definition for local editing. The cursor is positioned at the place of the supposed error. As an example, suppose the user types as a unit

```
ROW INT VOR a
```

The translator complains `Number of elements ?` and offers

```
ROW ■ INT VOR a
```

The user dutifully inserts `10` and hits the return key. The translator now complains `VAR/CONST ?` and offers

```
ROW 10 INT ■ VOR a
```

The user changes `VOR` into `VAR` and hits the return key `<RET>`, whereupon the translator accepts the rest of the line in silence and waits for the next line. This same incremental correction facility is available for correcting input during reading from a file.

As the Elan Programming Environments does not impose restrictions on the order of the definitions and the user might introduce all kinds of weird operators, like `THAN`, situations can occur in which the error signalling is not so helpful. For example, if the user types

```
IF a>=0 THAN a ELSE -a FI
```

the Elan Programming Environment will now complain `THEN ?` and offer

```
IF a>=0 THAN a ■ ELSE -a FI
```

apparently considering `THAN` to be meant as an operator and so interpreting this construction as

```
IF a >= (0 THAN a) ELSE -a FI
```

The error reported on the bottom line of the screen should be seen as just a suggestion! It is the best the programming environment can do, but it is up to the user to decide what it was he wanted to express and how to repair the error.

5.4.3 Break

If the user hits `<BREAK>` during input, the edit-mood is aborted and the Elan Programming Environment reverts to command-mood. This facility is helpful in order to recover from a total tangle of errors during input from a file (read-command).

5.5 The execute- and related moods

The executor first calls the checker and, if no errors were found, starts executing the program. If an error occurs, a back-trace is given (see 5.5.2).

Execution can be terminated by giving a `<BREAK>`, upon which the Elan Programming Environment reverts to trace-mood.

5.5.1 The checker

The checker is invoked automatically before execution. It takes at most a few seconds, showing `Checking...` on the status line. It checks, amongst others, the following:

- is every applied name defined exactly once?
- do types match in
  - source and destination of an assignation?
  - all parts of a conditional?
  - a numerical choice?
- are all operations defined for the type of operands given?
- are all conditions of type `BOOL`?
- are all indices and bounds of type `INT`?
- are all identifiers, used as bounds, synonym identifiers?
- are only rows subscripted?
- are only fields selected from?

Furthermore, it performs the identification of generic algorithms, reporting errors by giving a backtrace.

5.5.2 The backtrace-mood

Once the Elan Programming Environment has found a semantic error, it enters backtrace-mood, displaying `Backtrace command, please...` on the right of the status line, with an error description on the left of the status line. The backtrace-mood offers to the user the opportunity of unfolding the static nesting of the present unit and listing it. After each unit listed, the Elan Programming Environment waits for the next wish of the user, which can be one of the following:

- e edit-command

  modify the definition containing this unit and return to the command-mood.

- n next-command

  continue backtracing; return to command-mood if current unit is root of program, otherwise display unit immediately surrounding current unit.

- s show-command

  show the definition and the value of a given name.

- q quit-command

  stop backtracing, return to command-mood.

This mechanism serves to pinpoint semantic errors in the program.

5.5.3 The trace-mood

The trace-mood can be entered by the user by means of a trace-command (`t`). It can also be entered from a program by a call of `assert` (see appendix A.10). In the trace-mood, which is indicated by `Trace command, please...` on the status line, the executor lists every unit before executing it, and asks for a command as follows:

- If the unit is simple (i.e. not a invocation of a procedure, refinement or operator and not a control structure), it is executed. If this unit yields a result, this is displayed.

- If the unit is such an invocation or a control structure, it is listed and the executor waits for one of the following commands:

  - `b` backtrace-command

    The Elan Programming Environment switches to the backtrace-mood.
  - `c` continue-command

    The remainder of the program is executed without trace.
  - `e` edit-command

    Execution is terminated, and the definition that contains the current unit is offered for local editing.
  - `n` next-command

    The body of the current unit is executed in trace mode.
  - `p` previous-command

    All the remaining units (entered by the last next-command) are executed (without trace), if the body yields a result this is displayed and the trace-mood is re-entered.
  - `q` quit-command

    Return to the command mode.
  - `s` show-command

    A known name (may be abbreviated) is asked and the definition and the definition of that name is displayed.
  - `x` execute-command

    The body is executed (without trace), its result (if any) is displayed and trace-mood is re-entered.

At the end of each body the executor waits again for a valid trace-command.

5.5.4 The verify-mood

In some cases deeper insight in the cause of a semantic error may be acquired from stepwise inspection of types of program constructs. This can be accomplished by the verify-command, which causes the checker to run in trace-mood. Its behaviour is analogous to the trace-mood, with the exception that the types of the program constructs are displayed rather than their value.

Chapter 6

EXAMPLES ELAN-1 SUBSET

The following examples are intended to demonstrate specific Elan-1 features, such as structures, procedures, operators and graphics facilities, as well as the definition of abstract data types by means of a packet.

6.1 Points and line segments

This section contains a packet for defining and drawing points and line segments in the *R2*. It uses straight integer-graphics (see A.7). In order to reduce complexity no clipping is performed. Drawing outside the (hardware-dependent) graphics-screen will result in distorted pictures. The packet should be read by means of the packet-command (p). The Elan Programming Environment can read packets but has no special precompilation facilities.

```
PACKET points and lines
DEFINES POINT, LINE,
        point, line, AS, *, +, =,
        draw point, draw line, put coord:

TYPE POINT = STRUCT (REAL x, y, TEXT name);

TYPE LINE = STRUCT (POINT a, b, TEXT name);

POINT PROC point (REAL x, y):
  # denotation-procedure for a point #
  POINT: [x, y, ""]
ENDPROC point;

POINT OP * (REAL factor, POINT point):
  POINT: [factor * point.x, factor * point.y, ""]
ENDOP *;

BOOL OP = (POINT point1, point2):
  point1.x = point2.x AND point1.y = point2.y
ENDOP =;

LINE PROC line (POINT point1, point2):
  # denotation-procedure for a line #
  LINE: [point1, point2, ""]
ENDPROC line;
```

```
LINE OP + (LINE line1, line2):
  assert (line1.b = line2.a, "lines not connected");
  LINE: [line1.a, line2.b, ""]
ENDOP +;

LINE OP * (REAL factor, LINE line):
  LINE: [factor * line.a, factor * line.b, ""]
ENDOP *;

BOOL OP = (LINE line1, line2):
  line1.a = line2.a AND line1.b = line2.b OR
  line1.a = line2.b AND line1.b = line2.a
ENDOP =;

POINT OP AS (POINT point, TEXT name):
  # giving a name to a point #
  POINT: [point.x, point.y, name]
ENDOP AS;

LINE OP AS (LINE line, TEXT name):
  # giving a name to a line #
  LINE: [line.a, line.b, name]
ENDOP AS;

PROC put coord (POINT point):
  # showing the name and coordinates of a point #
  IF NOT (point.name = "")
  THEN put (point.name + " = ");
  FI;
  put ("(" + text (point.x) + "," + text (point.y) + ")")
ENDPROC put coord;

PROC put coord (LINE line):
  # showing the name and coordinates of a line segment #
  IF NOT (line.name = "")
  THEN put (line.name + " = ");
  FI;
  put ("(" + text (line.a.x) + "," + text (line.a.y) + ")");
  put ("(" + text (line.b.x) + "," + text (line.b.y) + ")")
ENDPROC put coord;

PROC draw point (POINT point):
  # drawing a point and its name on the screen #
  move (int (point.x), int (point.y / aspect));
  plot pixel;
  put (point.name)
ENDPROC draw point;

PROC draw line (LINE line):
  # drawing a line and its name on the screen #
  INT x name ::  int ((line.a.x + line.b.x) / 2.0),
      y name ::  int ((line.a.y + line.b.y) / 2.0 / aspect),
      len ::  length (line.name) * character width;
  draw point (line.a);
```

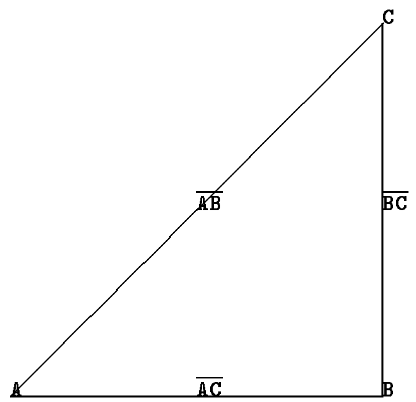
```
    draw point (line.b);
    move (x name, y name);
    put (line.name);
    move (x name, y name);
    draw (x name + len, y name);
    move (int (line.a.x), int (line.a.y / aspect));
    draw (int (line.b.x), int (line.b.y / aspect))
  ENDPROC draw line;

  ENDPACKET points and lines;
```

6.2 Points and line segments example

This example demonstrates the use of the previous packet. It draws a triangle, annotating its points and sides with names. Note that the coordinates of a, b and c should be chosen to fit the screen of the particular computer (see A.7).

```
program:
  POINT a ::  point (100.0, 400.0) AS "A",
        b ::  point (400.0, 100.0) AS "B",
        c ::  point (400.0, 400.0) AS "C";
  LINE ab ::  line (a, b) AS "AB",
       bc ::  line (b, c) AS "BC",
       ca ::  ab + bc AS "AC";
  enter graphics mode;
  draw line (ab);
  draw line (bc);
  draw line (ca);
  TEXT VAR wait ::  inchar;
  enter text mode.
```

The resulting picture looks like:



6.3 Intersection and projection

This section contains an addendum to the first packet. Procedures for finding the intersection of two lines and the projection of a point on a line are defined. Note that intersection of parallel lines will result in an error message.

```
POINT OP X (LINE line1, line2):
  calculate parameters;
  the intersection.

calculate parameters:
  REAL VAR rc1, b1, rc2, b2;
  BOOL vertical1 := (line1.a.x = line1.b.x);
  IF NOT vertical1
  THEN
    rc1 := (line1.a.y - line1.b.y) / (line1.a.x - line1.b.x);
    b1 := line1.a.y - rc1 * line1.a.x
  ELSE
    rc1 := maxreal
  FI;
  BOOL vertical2 := (line2.a.x = line2.b.x);
  IF NOT vertical2
  THEN
    rc2 := (line2.a.y - line2.b.y) / (line2.a.x - line2.b.x);
    b2 := line2.a.y - rc2 * line2.a.x
  ELSE
    rc2 := maxreal
  FI;
  assert (rc1 <> rc2, "intersection of two parallel lines").

the intersection:
  IF   vertical1
  THEN POINT: [line1.a.x, rc2 * line1.a.x + b2, ""]
  ELIF vertical2
  THEN POINT: [line2.a.x, rc1 * line2.a.x + b1, ""]
  ELSE POINT: [(b2 - b1) / (rc1 - rc2),
               (rc1 * b2 - b1 * rc2) / (rc1 - rc2),
               ""]
  FI.

ENDOP X;

POINT OP ON (POINT point, LINE line):
  calculate parameters;
  the projection.

calculate parameters:
  REAL VAR rc1, b1, rc2, b2;
  BOOL vertical ::  (line.a.x = line.b.x),
       horizontal ::  (line.a.y = line.b.y);
  IF NOT (horizontal OR vertical)
  THEN
    rc1 := (line.a.y - line.b.y) / (line.a.x - line.b.x);
    b1 := line.a.y - rc1 * line.a.x;
    rc2 := -1.0 / rc1;
    b2 := point.y - rc2 * point.x;
  FI.
```

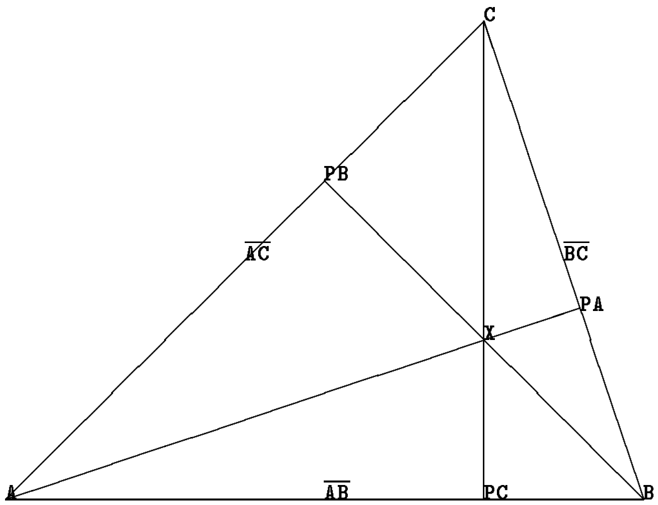
```
the projection:
  IF   horizontal
  THEN POINT: [point.x, line.a.y, ""]
  ELIF vertical
  THEN POINT: [line.a.x, point.y, ""]
  ELSE POINT: [(b2 - b1) / (rc1 - rc2),
               (rc1 * b2 - b1 * rc2) / (rc1 - rc2),
               ""]
  FI.


ENDOP ON;
```

6.4 Intersection and projection example

This example demonstrates the use of the X (intersect) and ON (projection) operators. A triangle with its projection lines is drawn, showing the intersection of the three projection lines trough one point. Again, the coordinates of a, b and c should be judiciously chosen to fit the screen.

```
program:
  POINT a ::  point (100.0, 370.0) AS "A",
        b ::  point (460.0, 370.0) AS "B",
        c ::  point (370.0, 100.0) AS "C";
  LINE ab ::  line (a, b) AS "AB",
       bc ::  line (b, c) AS "BC",
       ca ::  line (c, a) AS "AC";
  POINT a on bc ::  (a ON bc) AS "PA",
        b on ca ::  (b ON ca) AS "PB",
        c on ab ::  (c ON ab) AS "PC";
  POINT x ::  (line (c, c on ab) X line (a, a on bc)) AS "X";
  enter graphics mode;
  draw line (ab);
  draw line (bc);
  draw line (ca);
  draw line (line (a, a on bc));
  draw line (line (b, b on ca));
  draw line (line (c, c on ab));
  draw point (x);
  move (1, graphics y limit - 2 * line height);
  put coord (x);
  TEXT VAR wait ::  inchar;
  enter text mode.
```

The resulting picture looks like:




```
X = ( 3.700000e+02, 2.800000e+02)
```

Further examples are included with the distributed software.

Appendix A

ELAN 1V5 STANDARD LIBRARY

A library of concrete algorithms, objects and types is available in Elan. Conceptually, they are declared in the "standard packets" which are presupposed for every execution. Version 1.5 of the Elan Programming Environment includes practically all standard packets (see [1]) apart from matrices and vectors, as well as a small number of extensions from the Elan-0 and EUMEL libraries. In this chapter the headings of all definitions in the library are listed. No attempt is made to explain their meaning, since this can be found in the Elan textbooks.

A.1 Integer

```
TYPE INT

PROC get (INT VAR i)
PROC put (INT CONST i)

BOOL OP = (INT CONST i1, i2)
BOOL OP <> (INT CONST i1, i2)
BOOL OP < (INT CONST i1, i2)
BOOL OP <= (INT CONST i1, i2)
BOOL OP > (INT CONST i1, i2)
BOOL OP >= (INT CONST i1, i2)

INT OP + (INT CONST i1, i2)
INT OP - (INT CONST i1, i2)
INT OP * (INT CONST i1, i2)
INT OP DIV (INT CONST i1, i2)
INT OP MOD (INT CONST i1, i2)
INT OP ** (INT CONST i1, i2)

INT OP + (INT CONST i)
INT OP - (INT CONST i)

OP INCR (INT VAR i1, INT CONST i2)
OP DECR (INT VAR i1, INT CONST i2)
```

```
INT OP SIGN (INT CONST i)
INT OP ABS (INT CONST i)
INT PROC sign (INT CONST i)
INT PROC abs (INT CONST i)
BOOL PROC even (INT CONST i)
BOOL PROC odd (INT CONST i)
INT PROC max (INT CONST i1, i2)
INT PROC min (INT CONST i1, i2)

INT PROC trunc (REAL CONST r)
INT PROC round (REAL CONST r)
INT PROC int (TEXT CONST t)
INT PROC ascii (TEXT CONST t)
INT PROC code (TEXT CONST t)
INT PROC digit (TEXT CONST t)

INT PROC maxint
INT PROC minint
```

A.2 Real

```
TYPE REAL

PROC get (REAL VAR r)
PROC put (REAL CONST r)

BOOL OP = (REAL CONST r1, r2):
BOOL OP <> (REAL CONST r1, r2):
BOOL OP < (REAL CONST r1, r2):
BOOL OP <= (REAL CONST r1, r2):
BOOL OP > (REAL CONST r1, r2):
BOOL OP >= (REAL CONST r1, r2):

REAL OP + (REAL CONST r1, r2):
REAL OP - (REAL CONST r1, r2):
REAL OP * (REAL CONST r1, r2):
REAL OP / (REAL CONST r1, r2):
REAL OP / (INT CONST i1, i2):
REAL OP MOD (REAL CONST r1, r2):
REAL OP ** (REAL CONST r, INT CONST i):
REAL OP ** (REAL CONST r1, r2):

REAL OP + (REAL CONST r):
REAL OP - (REAL CONST r):

OP INCR (REAL VAR r1, REAL CONST r2):
OP DECR (REAL VAR r1, REAL CONST r2):
```

```
INT OP SIGN (REAL CONST r):
REAL OP ABS (REAL CONST r):
INT PROC sign (REAL CONST r)
REAL PROC abs (REAL CONST r)
REAL PROC max (REAL CONST r1, r2)
REAL PROC min (REAL CONST r1, r2)

REAL PROC real (INT CONST i)
REAL PROC real (TEXT CONST t)

REAL PROC maxreal
REAL PROC smallreal

LET pi = 3.141592653589793
REAL PROC sin (REAL CONST r)
REAL PROC cos (REAL CONST r)
REAL PROC tan (REAL CONST r)
REAL PROC arcsin (REAL CONST r)
REAL PROC arccos (REAL CONST r)
REAL PROC arctan (REAL CONST r)

LET e = 2.71828182845904
REAL PROC sqrt (REAL CONST r)
REAL PROC exp (REAL CONST r)
REAL PROC ln (REAL CONST r)
REAL PROC log10 (REAL CONST r)
REAL PROC log2 (REAL CONST r)
```

A.3 Text

```
TYPE TEXT

PROC get (TEXT VAR t)
PROC get (TEXT VAR t, INT CONST maxlen)
PROC get (TEXT VAR t, TEXT CONST delimiter)
PROC put (TEXT CONST t)

BOOL OP = (TEXT CONST t1, t2)
BOOL OP <> (TEXT CONST t1, t2)
BOOL OP < (TEXT CONST t1, t2)
BOOL OP <= (TEXT CONST t1, t2)
BOOL OP > (TEXT CONST t1, t2)
BOOL OP >= (TEXT CONST t1, t2)

INT OP LENGTH (TEXT CONST t)
INT PROC length (TEXT CONST t)
```

```
TEXT OP + (TEXT CONST t1, t2)
OP CAT (TEXT VAR t1, TEXT CONST t2)
TEXT OP * (INT CONST i, TEXT CONST t)
TEXT PROC compress (TEXT CONST t)
TEXT PROC text (TEXT CONST t, INT CONST length)
TEXT PROC text (TEXT CONST t, INT CONST length, from)
TEXT PROC subtext (TEXT CONST t, INT CONST from)
TEXT PROC subtext (TEXT CONST t, INT CONST from, to)
TEXT OP SUB (TEXT CONST t,INT CONST i)
PROC replace (TEXT VAR t, INT CONST from, TEXT CONST new)
PROC change (TEXT VAR t, TEXT CONST old, new)
PROC change all (TEXT VAR t, TEXT CONST old, new)
INT PROC pos TEXT CONST t, pat)

TEXT OP HEAD (TEXT CONST t)
TEXT OP TAIL (TEXT CONST t)

TEXT PROC ascii (INT CONST i)
TEXT PROC code (INT CONST i)
TEXT PROC digit (INT CONST i)

TEXT PROC text (INT CONST i)
TEXT PROC text (INT CONST i, width)
TEXT PROC text (REAL CONST r)
TEXT PROC text (REAL CONST r, INT CONST width)
TEXT PROC text (REAL CONST r, INT CONST width, after period)

LET niltext = ""
LET blank = " "
LET quote = """"
```

A.4 Boolean

```
TYPE BOOL

LET true = TRUE
LET false = FALSE

BOOL OP NOT (BOOL CONST b)
BOOL OP AND (BOOL CONST b1, b2)
BOOL OP OR (BOOL CONST b1, b2)
BOOL OP = (BOOL CONST b1, b2)
BOOL OP <> (BOOL CONST b1, b2)
```

A.5 File

```
TYPE FILE
TYPE TRANSPUTDIRECTION
```

```
FILE PROC sequential file (TRANSPUTDIRECTION CONST d,
                           TEXT CONST t)

TRANSPUTDIRECTION PROC input
TRANSPUTDIRECTION PROC output

PROC close (FILE CONST f)
PROC erase (FILE CONST f)

PROC line (FILE CONST f)
PROC line (FILE CONST f, INT CONST i)

PROC putline (FILE CONST f, TEXT CONST t)
PROC put (FILE CONST f, INT CONST i)
PROC put (FILE CONST f, REAL CONST r)
PROC put (FILE CONST f, TEXT CONST t)

PROC getline (FILE CONST f, TEXT VAR t)
PROC get (FILE CONST f, INT VAR i)
PROC get (FILE CONST f, REAL VAR r)
PROC get (FILE CONST f, TEXT VAR t)
PROC get (FILE CONST f, TEXT VAR t, INT CONST maxlen)
PROC get (FILE CONST f, TEXT VAR t, TEXT CONST delimiter)

INT PROC max line length (FILE CONST f)
TEXT PROC name (FILE CONST f)
BOOL PROC opened (FILE CONST f)
BOOL PROC new (FILE CONST f)
BOOL PROC eof (FILE CONST f)

PROC new file (TEXT CONST name)
PROC old file (TEXT CONST name)
PROC close file
PROC erase file (TEXT CONST name)

PROC write (INT CONST i)
PROC write (REAL CONST r)
PROC write (TEXT CONST t)
PROC write line

PROC read (INT VAR i)
PROC read (REAL VAR r)
PROC read (TEXT VAR t)

BOOL PROC file ended
```

A.6 Screen handling

```
PROC line
PROC line (INT CONST i)
PROC page
```

```
PROC beep

INT PROC xsize
INT PROC ysize
PROC cursor (INT CONST i1, i2)
PROC get cursor (INT VAR i1, i2)

PROC edit (TEXT VAR t, INT CONST start pos)
PROC edit (TEXT VAR t, INT CONST start pos, left margin,
           TEXT CONST end)
PROC edit (TEXT VAR t,
           INT CONST start pos, left margin, right margin,
           TEXT CONST end, TEXT VAR c)

PROC inchar (TEXT VAR t)
TEXT PROC inchar
TEXT PROC incharety
```

A.7 Graphics

```
INT PROC graphics x limit
INT PROC graphics y limit
REAL PROC aspect

PROC enter graphics mode
PROC enter text mode
BOOL PROC in text mode

PROC color (INT CONST i)
PROC clear graphics screen

INT PROC current x position
INT PROC current y position
PROC move (INT CONST x, y)
PROC draw (INT CONST x, y)
PROC plot pixel

PROC plot text (TEXT CONST t)
INT PROC character width
INT PROC line height
```

The console input-output routines are applicable also in graphics mode (example: `put`, `get`, `line`, `page`, `inchar`, `cursor`, `get cursor`,`xsize`, `ysize`, etc.).

A.8 Turtle-graphics

Turtle-graphics is not a part of the standard library, but a packet, which can be read into the Elan Programming Environment by means of the packet-command (`p`).

```
     enter turtle graphics
     leave turtle graphics
PROC turtle window (INT CONST xmin, xmax, ymin, ymax,
                    REAL CONST x range, y range)

PROC move (REAL CONST x, y)
PROC move (REAL CONST length)
PROC draw (REAL CONST length)

PROC turn (REAL CONST radian)
PROC turn (INT CONST angle)
PROC turn left
PROC turn right
```

A.9 Random numbers

```
PROC initialize random (INT CONST i)
PROC initialize random (REAL CONST r)

INT PROC random (INT CONST i1, i2)
PROC random (INT CONST i)
REAL PROC random
INT PROC choose128
```

A.10 Miscellaneous

```
BOOL OP ISNIL (NILTYPE CONST x)
BOOL PROC isnil (NILTYPE CONST x)

BOOL PROC last conversion ok

BOOL PROC yes (TEXT CONST question)
BOOL PROC no (TEXT CONST question)

INT PROC freespace
INT PROC exectime
PROC sleep (INT CONST seconds)

PROC trace on
PROC stop
PROC assert (BOOL CONST b)
PROC assert (BOOL CONST b, TEXT CONST t)
```

Note: there are some other routines in the standard library not mentioned in the above list, but those are for internal use within the library, therefore we strongly advise against using them.

Appendix B

ASCII-TABLE

The Elan interpreter uses a slightly modified ASCII code.

	0	1	2	3	4	5	6	7
0	nul			0	@	P		p
1	cls		!	1	A	Q	a	q
2	cll		"	2	B	R	b	r
3	brk		#	3	C	S	c	s
4	del		$	4	D	T	d	t
5	right		%	5	E	U	e	u
6	left		&	6	F	V	f	v
7	bel		‘	7	G	W	g	w
8	up		(	8	H	X	h	x
9	down	brk	)	9	I	Y	i	y
A	lf		*	:	J	Z	j	z
B			+	;	K	[	k	{
C			’	<	L		l	|
D	cr		-	=	M	]	m	}
E			.	>	N	^	n	
F			/	?	O	’	o	

The control characters have the following meanings:

`nul`	null character
`cls`	clear the screen
`cll`	clear to end of line
`del`	delete character under cursor
`right`	cursor right
`left`	cursor left
`up`	cursor up
`down`	cursor down
`bel`	produce the bell-sound
`lf`	line feed
`cr`	carriage return
`brk`	break

Contents