PROGRAMMING WITH ELAN

Part1: Top-Down Programming

The systematic design
of sequential algorithms
over simple data structures
and their realization
in Elan.

C. H. A. Koster
Informatics Department
University of Nijmegen
The Netherlands

November 17, 1998

Introduction

This textbook is intended for teachers, for students in the first year of university and in high school as well as for the more advanced students in secondary schools who wish to acquire a solid understanding of systematic programming.

In view of the wide range and diverse character of the intended audience, it makes very few demands on the knowledge of mathematics. The examples introduced have been chosen such that they presume only an elementary knowledge of mathematics, but of course many illuminating and useful examples can be found in that field. Some of the larger examples are concerned wit certain application areas of Informatics, but there is no orientation towards any particular application area. The teacher or reader can add such an orientation by the choice of suitable examples for practical work.

The central issue of all programming methodologies is the controlled use of abstraction. In this particular textbook we rely much more on linguistic abstraction than on mathematical abstraction. The language used, the Educational Language Elan, can be seen as a daughter of ALGOL68, designed specifically for the teaching and practice of systematic programming. The didactic approach followed here can be applied in using other languages, provided they possess the necessary concepts. But only Elan is specially designed to support a number of systematic programming methods:

- Top-Down programming,
- recursive programming,
- Bottom-Up programming, and
- modular programming.

This first part deals predominantly with Top-Down programming. It exploits the fact that, in Elan, the refinement is not a paper-and-pencil technique but a central language mechanism, supported by a sophisticated programming environment. It furthermore deals with the possibilities offered by the exploitation of recursion. The second part will be concerned with Bottom-Up and Modular programming, and presupposes a good understanding of Top-Down programming.

This somewhat arbitrary and dogmatic division is motivated by a difference in the intended audience: the second part is aimed at a more professional user, who wants to learn how to design and implement non-trivial programs in a systematic fashion. The second part will therefore also be much more formal in its approach.

The book attempts to be very careful and consistent in the introduction of concepts and terms, both regarding the programming language and regarding the programming process itself. Standard terminology is used as far as possible, but jargon is avoided. The concepts and terms used here are not confined to Elan, but can be used to understand programming in any language. Once systematic programming in Elan has been mastered, it should be easy to learn to use other languages as well.

It should be pointed out that, apart from studying the book, the hands-on experience of doing a number of exercises is indispensable, in order to acquire the necessary skills and to experience the possibilities and limitations of the methods taught. For such practical exercises, some computer with an Elan implementation will have to be available.

From the publisher of this book, Ellis Horwood, interpreters for two subsets of the full language can be obtained. The smallest subset, Elan-0, is available on small microcomputer systems and is sufficient for exercises about the first ten chapters of the book. For the rest of the book the larger subset, Elan-1, is needed, which is available on MS-DOS computers and other medium-size microcomputer systems. Another implementation (of full ELAN) is the EUMEL-system, available from the GMD, Postfach 1240, D-5205 St. Augustin (Germany).

In order to avoid confusion between the different subsets, it is clearly indicated in the text whenever some remark applies only to a particular subset.

It is hoped that this book and the various Elan implementations will provide a sensible alternative to BASIC and an improvement over PASCAL in the teaching of systematic programming.

Nijmegen, October 1988

C.H.A. Koster

Chapter 1

Algorithms

In programming we try to tell a computer what to do, i.e. we describe what the computer should do in such a form that the computer can obey that description. Such a description we call an *algorithm.* In formulating algorithms we can learn quite a lot from studying other descriptions, intended to tell people what to do: instruction manuals, cooking recipes, knitting patterns, and so on.

In this chapter we will investigate some algorithms from daily life, and identify some important aspects of algorithms. In particular we shall draw attention to the importance of abstraction in the formulation of algorithms. The chapter ends with a tentative definition of the notion of algorithm.

1.1 Algorithms in daily life

The following examples, whether actual examples from daily life or contrived, show a number of important aspects of algorithms.

1.1.1 Paybox telephone

As a first example of an algorithm we give instructions for the use of a public paybox telephone, such as might be found next to the telephone.

INSTRUCTIONS FOR USE
1. Take the telephone from the hook,
2. wait for the dialtone,
3. put in coins,
4. choose the desired number.

Inspecting this algorithm somewhat more closely, we notice a number of important points.

- The algorithm is a description that is intended to guide the actions of a human being. The description is *executable*, in contrast to the notice that the telephone is property of the telephone company.

- The execution of the algorithm proceeds in steps. The sequence of steps taken in a specific execution of the algorithm we call a *process* described by that algorithm. The executor of the algorithm, in this case a human being, is called the *processor.*

- One step in the execution of the algorithm consists in its turn of the execution of one or more (other) algorithms that are indicated in the algorithm by a *name* (such as **take the telephone from the hook)**. In these instructions such named algorithms are considered as *elementary.* The given algorithm is composed from elementary algorithms. Whoever wants to execute the algorithm must know the elementary algorithms and be capable of executing them. He must know the language in which the algorithm is written; he must know what a dialtone is and how to put in coins.

- We demand from an algorithm that it is sufficiently precise, i.e. that each step as well as the order of their execution is not open to conflicting interpretations. To that end, the algorithm must be formulated at a suitable level of detail *(level of abstraction).*

  In order to make clear how to take the phone from the hook, we may of course indicate which hand has to be extended to the phone (distinguishing between left- and right-handed people), with what force (in newtons) the various fingers have to grasp the phone and with what speed that end of the phone to which the cord is not connected has to be moved with its flat side until it is no more than one sixteenth of an inch from the right ear of the processor (if the right ear is absent one has to take the left ear). But this is by no means the end of the story: we must define how and when the individual muscles of hand and arm are to be applied. In this way the description threatens to sink in a bottomless swamp of more and more extensive details. Only a coherent choice of not too detailed elementary algorithms (**take the telephone from the hook, wait for the dialtone**) and a formulation of the algorithm on the thus defined level of abstraction allows us to be precise.

  Precision entails the choice of a specific level of abstraction.

- The algorithm, i.e. the text of the description, must be *finite,* and even short when the processor is to be a human being. The execution of the algorithm (the process) is not necessarily finite (think of an algorithm for walking a treadmill).

1.1.2 Cooking recipe

In cookbooks one finds recipes, i.e. instructions for the preparation of some food, written with a typical use of language and typical notational conventions.

> CHICKEN à LA MARSEILLAISE.
> 2 broilers, 6 tomatoes, 2 paprikas, 1/2 cup of white wine, 1/2 cup of bouillon, 1/2 lemon, salt, pepper, thyme and a clove of garlic.
> Divide the broilers into pieces. Salt and pepper and put into the clay pot, which has been soaked in water. Add the finely cut paprikas, the quartered and peeled tomatoes and the garlic. Add the white wine and bouillon which has been seasoned with pepper, thyme and lemon juice, mix thoroughly. Put the closed pot into the oven and cook the dish at a temperature of 225 Centigrade in circa 90 minutes.

For a cook who is not too inexperienced this is a precise, executable and, of course, finite description of a process that he learns to know under the name of "preparation of chicken à la Marseillaise" ([EXN70]).

The execution of this algorithm consists of the execution of other algorithms **(cut, season)** manipulating objects **(broilers, pepper).** Again we notice a number of important aspects of this algorithm.

- At the start of the description a list is given of the necessary ingredients that will be used in performing the algorithm. The algorithm describes operations on *named objects,* such as the peeling of the tomatoes, and the cutting of the paprikas. Some objects must be available for the execution of the algorithm *(input),* others become available as the result of the execution of the algorithm *(output)* and yet others exist only during the process *(local* objects).
- Objects can be *elementary,* i.e. indicated in the algorithm by their name and showing no further structure, or they can be considered as *composed,* when their structure is essential to the execution of the algorithm. In the previous example the telephone consisted of a phone, a hook, a dial, etc. This is also a matter of abstraction level — to a chemist, salt can look quite different than to a cook. And a physicist may even be interested in the state of the sodium ions in the salt.

  There is an interdependency between algorithms and objects: the choice of the elementary objects decides the level of detail of the elementary algorithms operating on them and conversely, the choice of the elementary algorithms fixes the kinds of objects to which they are applied.

Together the algorithms and objects form the building stones of the algorithmic universe, that we can subdivide as follows:

algorithms

  elementary algorithms
  composed algorithms

objects

  elementary objects
  composed objects

1.1.3 Knitting pattern

In fashion magazines one can find knitting patterns like the following.

> STRIPED PULLOVER (3 - 4 YEARS)
> Materials: 250 gms middle weight dralon/mohair mix in each of pink, white, dark green and light green.
> Ribbing: 2 knit, 2 purl ending with purl.
> Body: knit in stocking stitch (knit across and purl on the way back).
> Stitch tension: 26 stitches and 37 rows = 10 x 10 cm.
> Stripe pattern: * 4 rows pink, 4 rows white, 4 rows dark green, 4 rows light green, 2 rows pink, 2 rows white, 2 rows dark green, 2 rows light green, repeat from * three times ending with 4 rows pink, 4 rows white, 4 rows dark green. Pattern equals 108 rows.
> Back: Cast on 130 stitches in pink.
> Begin with 4 rows of ribbing for the border then continue in stocking stitch following the stripe pattern. After 78 rows (21 cm) . . .

Again we make a number of observations.

- The algorithm (of which we showed only a fragment) consists of the description of a number of *sub-algorithms,* such as **Ribbing, Body** or **Stripe pattern.** These sub-algorithms in their turn are described in terms of elementary algorithms, such as **purl** and **knit**.

  These sub-algorithms are defined in the algorithm itself, and they can be *invoked* by mentioning their name.
- Some of these sub-algorithms have *parameters* to influence the result of their execution. As an example, **4 rows white** will give a white stripe in the pullover, whereas **2 rows dark green** will give a somewhat smaller dark green stripe.
- In the sub-algorithm **Stripe pattern** the marker * serves to indicate a place from which a part of it has to be repeated. The repetition is a descriptional trick to keep the algorithm short.
- The algorithm contains a number of statements that are not by themselves intended to be executed (such as **Stitch tension** and **Pattern equals**

108 rows) but serve to allow a plausibility check on the execution of the algorithm. They are redundant but useful.

For somebody who does not know how to knit, this algorithm is nearly incomprehensible. It is written in a language that can be understood only by people who know the elementary algorithms. But without the formalism, one would have to describe the movements that the hands, the needles and the threads would have to make during the knitting, which in itself would again raise the problem of describing such movements.

However precisely one tries to describe some actions algorithmically, there always remain some elementary algorithms whose meaning is not indicated in the algorithm. An algorithm is always written in such a way that a great many details of the process described are abstracted from. Such an *abstraction* is unavoidable.

One form of abstraction lies in the fact that we give a name to an algorithm, so that this name can be used in other places instead of a detailed algorithm (this also leads to brevity in this description). Once one knows what is the description of such an elementary algorithm, at places where this algorithm has to be applied it suffices to mention only its name. The abstract algorithms form an extension of the elementary algorithms.

1.1.4 Driving a car

The following example could have been taken from an instruction manual for driving a car (on the continent or in the U.S.A.).

STARTING FROM THE KERB

1. Wait until the direct environment of the car is free (beware of playing children), apply the hand-brake, put the gear shift into neutral and if the engine is cold pull out the choke.
2. Push the start button briefly.
3. Repeat from step 2 if car engine is not running.
4. Look over left shoulder and wait until road is free.
5. Left foot: depress clutch.
   Right hand: disengage hand-brake.
6. Left foot: keep clutch depressed.
   Right hand: put into first gear.
7. Left foot: slowly lift clutch.
   Right foot: push accelerator.
   Both hands: turn steering wheel to the left.
8. Left foot: release clutch.
   Right foot: push accelerator.
   Both hands: turn the steering wheel so that the car moves in the right direction.

Without concerning ourselves with the complexity or correctness of this algorithm we make the following remarks about its structure.

- The algorithm consists of a number of steps that have to be executed in the order indicated *(serial execution).*
- The first step consists of four parts that can be performed in any order, or even simultaneously *(collateral execution).*
- The last three steps contain parts that have necessarily to be performed simultaneously *(parallel execution).*
- Step 3 consists of the *conditional repetition* of step 2 and one of the parts of step 1 is also conditional.

The collateral parts of an algorithm can be performed in parallel, provided sufficient processors, in this case hands and feet, are available, but they can also be performed in any order one after another or simultaneously without influencing the results (apart from some influence on the execution time). The parallel parts of the algorithm however have to be performed simultaneously in order to reach the desired result. We call an algorithm *sequential* if it does not make any essential use of parallelity. This algorithm therefore is non-sequential.

For a number of reasons, non-sequential algorithms are much harder to understand than sequential ones. In the remainder of this book we therefore restrict ourselves to sequential algorithms, in this way also laying a basis for the understanding of non-sequential ones.

1.2 Summary

An algorithm is a *precise, finite* description of one or more *processes.* This description is composed of other algorithms and, ultimately, *elementary algorithms.*

An algorithm describes a process in the form of manipulations of *objects* that are composed of other objects and finally of *elementary* objects.

The structure of algorithms and the structure of objects manipulated by those algorithms show a strong similarity that can be depicted as follows:

- program text, consisting of
  - algorithms
    - elementary algorithms
      - concrete elementary algorithms
      - abstract elementary algorithms
    - composed algorithms
  - objects
    - elementary objects
      - concrete elementary objects
      - abstract elementary objects

composed objects

An algorithm must be *executable* for a *processor,* be it a human being or an automaton, that knows the *formalism* in which the algorithm is written and the meaning of the elementary algorithms.

An algorithm can be *sequential* or *non-sequential.* A sequential algorithm has the property that all its parts may be executed *serially.* A non-sequential algorithm has some parts that have to be executed in *parallel.*

Chapter 2

Notation of algorithms

In the previous chapter we found, more or less intuitively, a number of aspects of algorithms and objects. In this chapter we will introduce a notation for most of these aspects, as a preparation for the introduction of the programming language.

We shall not immediately start with a systematic treatment of a language but try to focus more on the stepwise development of algorithms. The purpose of this chapter is to get used to a number of essential algorithmic ingredients of programming languages such as actions, conditions, control structures and refinements, while avoiding (at first) difficult concepts like objects, types and data structures.

2.1 Karel the robot

In this chapter we shall make use of a didactic model, introduced in the book *Karel the Robot* [PAT81]. This model, that is obviously based on ideas of S. Papert [PAP80], allows us to avoid in first instance the notion of the object from informatics and instead rely on the world of daily experience.

Karel the robot lives in a simple algorithmic world on a flat surface. He spends all his time on street corners. In the following figure you see him standing on the corner of the second street and the 8th avenue, with his nose pointing east:

```
. . . . . . . . . .
. . . . . . . . . .
. > . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

At the start of his world, Karel stands in the origin (first Street, first Avenue) in the lower left-hand corner of the screen, with his nose pointing north.

Apart from Karel himself, his world contains also beepers and walls. Beepers can lie on street corners. Karel can pick up a beeper and put it in his pocket, or take one from his pocket and drop it on the corner where he is standing. To that end, he has a pocket containing an unlimited supply of beepers.

On the screen a wall will be depicted as `X`, and a beeper as `o`. In order to keep Karel within the visible part of his world, the screen is surrounded by an imaginary wall (just outside of the screen).

```
. . . . . . . . . .
. . X X X X . . . .
. . . . . X o . . .
. . . . . X o . . .
. . . . . X V . . .
. . . . . X . . . .
. . . . . X X X X X
. . . . . . . . . X
. . . . . . . . . X
. . . . . . . . . .
```

Walls may block street corners. Karel cannot stand on such a corner. Therefore he sometimes has to take a roundabout route in order to arrive where we want him to go.

2.2 Karel fetches his morning paper (1)

In the course of an example we shall design an algorithm for Karel, while at the same time introducing a specific notation for algorithms.

Karel is lying in bed in his home, facing west. In front of the outside wall of his house lies a newspaper, under the guise of a beeper.

We shall write an algorithm for Karel to fetch his newspaper and take it to bed with him.

2.2.1 First attempt

We shall split Karel's task into three parts, that he has to perform in the listed order.

```
go to garden;
take newspaper;
go back to bed.
```

Each of the three sentences appearing here we shall consider as the name of an algorithm. We write their

```
. . . . . . . . . . . .
. . . . X X X X X X . .
. . . . X . . . . X . .
. . X X X . . . . X . .
. . . . . . . . . X . .
. . X X X . . . . X . .
. . . o X . . . . X . .
. . . . X . . . . X . .
. . . . X . . . . X . .
. . . . X < . . . X . .
. . . . X X X X X X . .
. . . . . . . . . . . .
```

names, separated by semicolons, in order to indicate that they have to be performed sequentially and in the given order. At the end of the algorithm we put a period. Such a sequence of actions (units) separated by semicolons, we call a *paragraph.*

We choose the names of algorithms to be such that they indicate *what* the execution of the algorithm achieves. The names of the algorithms are therefore verbal formulations of their effect, that have to be chosen with care.

Now it would of course be unreasonable to suppose that an algorithm like `go to garden` is one of Karel's concrete algorithms. We will rather attempt to describe *how* the algorithm `go to garden` has to be performed in terms of simpler algorithms. The notation that we use for this purpose is called a *refinement,* which gives a name to a paragraph.

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

This refinement defines the algorithm `go to garden` as the sequential execution of the other algorithms mentioned. This does not take us much farther, because those algorithms are still much too particular to let them be part of Karel's repertoire of concrete algorithms. At any rate we have already gained some detail. We define analogously

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

As a newspaper we shall of course take a beeper.

```
take newspaper:
  take beeper.
```

2.2.2 Concrete algorithms of Karel

What are Karel's concrete algorithms? We shall choose at least the following:

`start karel` makes the world of Karel appear on your computer screen; Karel appears in the origin, with his nose pointing north.

`turn right` makes Karel turn 90 degrees to the right on his flat world.

`turn left` same to the left.

`move` makes him move to the next street corner in the direction of his nose. We shall have to take care that Karel does not try to move through a wall.

`take beeper` Karel takes the beeper, which must be present at this street corner, and puts it in his pocket.

`drop beeper` Karel takes a beeper from his pocket and drops it at the current street corner. We shall have to take care that there is no beeper lying there already.

`stop` puts an end to Karel's world.

These instructions are reasonably suitable for a simple robot like Karel. Later on we shall add some more to his repertoire.

2.2.3 Continuation of the example

In terms of these concrete algorithms we can now define the remaining refinements.

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

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

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

2.2.4 The limited repetition

The algorithms given above are of course rather boring, because they contain so many repetitions of the algorithm `move`. If we want Karel to move over larger distances this will get even worse. We obviously need a notation to indicate that a specific paragraph has to be repeated a number of times. For that, we introduce a notation, the *limited repetition*:

```
UPTO number
REP
  action
ENDREP
```

which means that the `action` is repeated this `number` of times.

We can now write (it is not much shorter, but certainly less boring):

```
walk along wall:
  turn right;
  UPTO 5
  REP
    move
  ENDREP.
```

The text of this refinement can be shortened even more (without any effect on its meaning) by using a somewhat denser layout.

```
walk along wall:
  turn right;
  UPTO 5 REP move ENDREP.
```

The refinement `walk to newspaper` is also simplified somewhat.

```
walk to newspaper:
  turn left;
  UPTO 2 REP move ENDREP;
  turn left;
  UPTO 2 REP move ENDREP;
  turn right;
  move.
```

We can exploit the regularity in this algorithm to make another realization.

```
walk to newspaper:
  UPTO 2
  REP
    turn left;
    move;
    move
  ENDREP;
  turn right;
  move.
```

We may even write:

```
walk to newspaper:
  UPTO 2
  REP turn left;
      UPTO 2 REP move ENDREP
  ENDREP;
  turn right;
  move.
```

According to the jargon of Elan, the body of this refinement is a paragraph consisting of three *units* separated by semicolons, the first of which is a repetition; the action repeated is in its turn a paragraph consisting of two units, of which the second is a repetition. We see that a repetition may occur as a unit within another repetition. This is called a *nested* use of the repetition (think of a number of bowls, one nested within another).

We realize the remaining refinements.

```
go back to entrance:
  turn about;
  move;
  turn left;
  UPTO 2 REP move ENDREP;
  turn right;
  UPTO 2 REP move ENDREP.

turn about:
  turn right;
  turn right.
```

Of course we might just as well have taken `turn left`.

```
walk back through entrance:
  turn right;
  UPTO 4 REP move ENDREP.

walk back along wall:
  turn right;
  UPTO 5 REP move ENDREP.
```

Our first concrete program is now complete. We can execute it with the aid of the Karel-environment which is delivered along with our Elan implementation, so that we can see Karel move over the screen. The structure of our program can be depicted in a diagram which indicates for each refinement which other refinements it uses, as follows:

```
program
   go to garden
      walk along wall
      walk through entrance
      walk to newspaper
   take newspaper
   go back to bed
      go back to entrance
         turn about
      walk back through entrance
      walk back along wall
```

2.2.5 Summing up

Up to now we have met two mechanisms for the composition of algorithms:

- the *paragraph* consisting of units, separated by semicolons, and

- the *repetition*, written with the keywords `UPTO ... REP ... ENDREP`.

Such composition mechanisms for algorithms are termed *control structures*. Furthermore we have seen how abstract algorithms can be defined with the aid of *refinements*.

2.3 Karel fetches his morning paper (2)

We now modify the problem statement somewhat, in order to illustrate more language constructions and concepts. We assume the same problem as before, except that

- the newspaper is not lying at a specific place but is lying somewhere in Karel's front garden at one of the places indicated in the following figure by a question mark;
- Karel does not necessarily start with his nose facing north, so that we first have to turn him with his nose to the north. In the figure we indicate Karel, whose nose direction we do not know, by means of a letter K.

```
. . . . . . . . . . . .
. . . . X X X X X X . .
. . . . X . . . . X . .
. . X X X . . . . X . .
. . . . . . . . . X . .
. . X X X . . . . X . .
. . . . X . . . . X . .
. . . ? X . . . . X . .
. . . ? X . . . . X . .
. . . ? X K . . . X . .
. . . ? X X X X X X . .
. . . . . . . . . . . .
```

The program now begins slightly differently.

```
go to front garden;
find the newspaper;
go back to bed.
```

```
go to front garden:
  turn to the north;
  walk along wall;
  walk through entrance;
  walk to wall.
```

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

2.3.1 The choice

In refining `turn to the north` we have to decide, in one way or another, whether Karel is already pointing his nose in the right direction, and otherwise turn him in the desired direction. We shall have to choose between different actions, depending on the *condition* whether Karel is already pointing north. The notation that we introduce for that purpose is the *choice.*

```
IF condition
THEN the one
ELSE the other
FI
```

which means: if the `condition` is true then `the one` is done and otherwise `the other`. Using this notation we write

```
turn to the north:
  IF not pointing north
  THEN turn around
  ELSE do nothing
  FI.
```

```
turn around:
  IF pointing south
  THEN turn about
  ELSE
    IF pointing east
    THEN turn left
    ELSE turn right
    FI
  FI.
```

Notice the nesting of choices in this refinement.

```
turn about:
  turn left;
  turn left.
```

How do we instruct Karel to do nothing? The simplest thing is to leave out the second alternative of the choice.

```
turn to the north:
  IF not pointing north
  THEN turn around
  FI.
```

The meaning of this is: If Karel is not pointing north the further execution of `turn to the north` consists of `turn around` and otherwise the execution of the algorithm is immediately complete.

2.3.2 The conditional repetition

By the use of a limited repetition we can give a somewhat shorter definition for `turn to the north`:

```
turn to the north:
  UPTO 3
  REP IF not pointing north
      THEN
         turn right
      FI
  ENDREP.
```

This is a typical robot solution. In the stupidest way imaginable we three times decide whether Karel is already pointed the right way and otherwise make a right turn. If Karel happens to be pointed the right way initially, all the work is for nothing. Furthermore it takes some thought to establish that three times is enough. The repetition with a fixed number of turns (the *limited repetition)* is not what we need here. We want the repetition to take place as long as a specific condition is met (a *conditional repetition).* To that end we introduce a new notation

```
WHILE condition
REP action
ENDREP
```

which means: as long as the `condition` is true, the `action` is performed and then the `condition` tested again, until it turns out that the `condition` does not hold any more.

We can now write

```
turn to the north:
  WHILE not pointing north
  REP
    turn right
  ENDREP.
```

2.3.3 Conditions

A *condition* is an algorithm that as result of its execution yields a truth value (yes or no). We give Karel the following repertoire of concrete conditions

`pointing north` yes only if Karel's nose is pointing north (and no otherwise).

`pointing east` yes only if Karel's nose is pointing east.

`pointing south` yes only if Karel's nose is pointing south.

`pointing west` yes only if Karel's nose is pointing west.

`wall ahead` yes only if the following corner is covered by a wall so that Karel cannot go there.

`wall right` yes only if the street corner to Karel's right contains a wall.

`at beeper` yes only if there is a beeper at the current street corner.

We furthermore introduce a notation for the *denial* of a condition:

```
NOT at beeper
```

will mean: yes only if the current street corner does not have a beeper, and no otherwise. Using this notation we may write

```
not pointing north:
   NOT pointing north.
```

The meaning of this definition is rather subtle: we define an abstract algorithm named `not pointing north`, that yields a truth value, the denial of the result of the condition `pointing north`.

The introduction of this operator `NOT`, that inverts truth values, saves us the introduction of particular algorithms for the denial of each of those conditions.

2.3.4 Continuation of the example

Once Karel's nose is pointing the right way, he can be brought to the front garden by the refinement `go to front garden` which we had already written. In Karel's front garden, the newspaper lies in one of the following four possible places:

```
. . . . . . . . . . . .
. . . . X X X X X X . .
. . . . X . . . . X . .
. . X X X . . . . X . .
. . . . . . . . . X . .
. . X X X . . . . X . .
. . . > X . . . . X . .
. . . ? X . . . . X . .
. . . ? X . . . . X . .
. . . ? X . . . . X . .
. . . ? X X X X X X . .
. . . . . . . . . . . .
```

Once he has arrived in his garden at the first question mark, with his nose pointing south, we let Karel perform a systematic search.

```
find the newspaper:
  move;
  IF at newspaper
  THEN
    take newspaper
  FI;
  move;
  IF at newspaper
  THEN
    take newspaper
  FI;
  move;
  IF at newspaper
  THEN
    take newspaper
  FI;
  move;
  IF at newspaper
  THEN
    take newspaper
  FI;
  four steps back.

at newspaper:
  at beeper.

take newspaper:
  take beeper.

four steps back:
  turn about;
  UPTO 4
  REP move
  ENDREP.
```

Exploiting the regularity in the previous algorithms, we may make use of the limited repetition.

```
find the newspaper:
  UPTO 4
  REP
    move;
    IF at newspaper
    THEN take newspaper
    FI
  ENDREP;
  four steps back.
```

But that is yet another of those typical robot solutions. A better technique is:

```
find the newspaper::
  walk until newspaper;
  take newspaper;
  turn about;
  walk back until wall.

walk until newspaper:
  WHILE not at newspaper
  REP
    move
  ENDREP.

not at newspaper:
  NOT at beeper.
```

What happens if there is no newspaper at all in the front garden? With this last solution, Karel disappears south like a meteor, because he does not meet a beeper, until he drops off the screen.

The previous solution with the limited repetition is somewhat more robust: in that case at least Karel returns to his bed without his newspaper.

In designing an algorithm, questions like these have to be put explicitly. An algorithm is intended to function under specific conditions. If these are not satisfied (for instance by a human error) strange things may happen.

2.3.5 Another form of conditional repetition

There is something artificial in the realization of `walk until newspaper`: Actually we don't want Karel to walk as long as he is not at the newspaper, but to walk until he is at the newspaper. With the current formulation, Karel repeatedly looks whether he is at the newspaper, and moves if he isn't. It is a "prechecked loop". The first time around, the condition will certainly be false and Karel will have to make a move. There exists in Elan another form of conditional repetition (a "postchecked loop"), which would be slightly more natural in this example:

```
walk until newspaper:
  REP move
  UNTIL at newspaper
  ENDREP.

at newspaper:
  at beeper.
```

Notice that we got rid of a NOT sign. We define similarly:

```
walk back until wall:
REP move
UNTIL wall ahead
ENDREP.
```

A repetition of the form

```
REP action
UNTIL condition
ENDREP
```

means the following: The `action` is performed once, and after that, as long as the `condition` is false, the `action` is performed and then the `condition` tested again, until it becomes true. This repetition is therefore equivalent to the paragraph

```
action;
WHILE NOT condition
REP action
ENDREP
```

It is useful to have both forms of conditional repetition available, although strictly speaking one, e.g. the WHILE-form, would satisfy all needs.

2.3.6 Conclusion

Writing a few missing refinements, or taking them from the previous example, we again obtain a complete program, whose structure can be depicted as:

```
program
   go to front garden
      turn to the north
         not pointing north
         turn around
      walk along wall
      walk through entrance
      walk to wall
   find the newspaper
      walk until newspaper
         at newspaper
      take newspaper
      turn about
      walk back until wall
   go back to bed
      walk back to entrance
      walk back through entrance
      walk back along wall
```

2.4 Delimiters

The notation that we have introduced makes use of magic words *(delimiters)* like `IF` and `REP` that give the impression of being (parts of) English words. Note that these words are not used with their natural meaning and also are not names of algorithms or objects. Delimiters are parts of specific conventional figures of style and for that reason are written in large capital letters, in distinction to *names* that consist of small letters and possibly digits.

Some people may find that abbreviated delimiters like `REP` are too cryptic and would rather see complete words. For this reason Elan allows alternative representations for a number of symbols like:

```
REP       REPEAT
ENDREP    ENDREPEAT
FI        ENDIF
```

We shall however systematically use the shorter versions in preference.

2.5 Summary

We have now made our acquaintance with the Top-Down programming style, in which we introduce abstract algorithms as needed and afterwards define them, until the concrete level has been reached.

Abstract algorithms are introduced by means of a *refinement*:

```
name : paragraph .
```

The *name* is written with small letters, digits and (possibly) spaces, and starts with a letter.

A *paragraph* consists of one or more units, separated from one another by semicolons:

```
unit ; unit ; ... ; unit
```

Such a *unit* may either be elementary (e.g. the invocation of an algorithm) or it may be composed (e.g. with the aid of a control structure) from other units.

Apart from the paragraph, the *choice* and the *repetition* belong to the control structures. The *choice* has two forms:

```
IF condition
THEN this paragraph
ELSE other paragraph
FI
```

and

```
IF condition
THEN paragraph
FI
```

The *condition* is a unit yielding a truth value, on the basis of which a choice is made. For the repetition we give three forms. The *limited repetition* has the form

```
UPTO expression
REP paragraph
ENDREP
```

in which the *expression* is an algorithm yielding a whole number, the number of times that the paragraph has to be executed. The *conditional repetition* has two forms

```
WHILE condition
REP paragraph
ENDREP
```

and

```
REP paragraph
UNTIL condition
ENDREP
```

In both forms, the number of repetitions of the paragraph is determined by the condition.

All these constructions belong to the language Elan. Furthermore we have introduced a number of concrete actions and conditions that belong to the environment of Karel the robot and appear only in this chapter. They serve to provide some exercises with Karel. Immediately after this chapter you may forget them again.

If we want to go further in the direction of learning a real programming language we have to get to know its concrete algorithms and especially its notations and concepts for dealing with objects. We shall introduce those in the succeeding chapters.

2.6 Exercises

In these exercises we shall make use of the Karel-packet that is distributed together with the Elan interpreter. At the start of each exercise we bring Karel to the origin of an empty world, with his nose pointing north, by invoking `start karel`.

1. One of Karel's concrete algorithms `turn left` and `turn right` is in fact superfluous. Define this one in terms of the other.

2. (Square) Let Karel move around a square whose side has length 10.

3. (Potato field) Consider the screen as a field with furrows in the horizontal direction. Let Karel seed with potatoes a field of 16 furrows of 20 moves each. Karel must return to the origin at the end.

4. (Fisherman) Teach Karel how to draw a Dutch folkloristic figure looking like in Fig. 2.1.

   Try to express the drawing of this fisherman (and his flag) as neatly as possible in terms of the drawing of his component parts.

5. (House) Do the same for drawing a house (of your own design).

6. (Staircase) The lowest line of the screen is considered to be a floor on which stands a staircase. The north is therefore interpreted as "above". Develop an algorithm that can walk over a floor with stairs (independent of the place, form and height thereof) until a beeper is found. A typical staircase can be constructed by calling `make stairs`.

7. (Maze) Karel finds himself in a maze. He must try to get to the outside by walking with his right hand touching the wall. Outside the maze, to the right of its entrance, lies a beeper. Develop an algorithm to get out of the maze. An example of a maze can be constructed by means of `make labyrinth`.

```
. . . . . . . . . . . . o o o
. . . . . o o o o o . . o . o
. . . . . o . . . o . . o o o
. . . . . o . . . o . . o . .
. . . . . o o . o o . o o o .
. . . . . . o . o . . o . o .
. o o o o o o . o o o o . o .
. o . . . . . . . . . . . o .
. o . o o o . . . o o o o o .
. o . o . o . . . o . . . . .
. o o o . o . . . o . . . . .
. . . . . o . . . o . . . . .
. . . o o o . . . o o o . . .
. . . o . . . . . . . o . . .
. . . o . . o o o . . o . . .
. . . o . . o . o . . o . . .
. . . o . . o . o . . o . . .
. . . o o o o . o o o o . . .
```

Figure 2.1: The fisherman with his flag

Chapter 3

The whole numbers

In this chapter we shall begin the systematic introduction of Elan. First we shall deal with the four concrete elementary types: the whole numbers, real numbers, truth values and texts. These concrete types of Elan are more or less the same as in other programming languages and have, on most computers, a direct representation. (The direct representation means that the computer can operate on such elementary objects with single machine instructions, e.g. it can add or multiply directly two integers or two reals. In this respect the texts, and in some cases the reals, are exceptions. Texts — as we shall see later — are sequences of characters, so called (character) strings, and most computers can operate on a single character only.)

The concrete types of Elan are defined in the *standard packets,* (hypothetical) pieces of program containing declarations for all concrete objects, types and algorithms. In the second volume of this book we shall also learn how to construct our own packets, a facility for Bottom-Up programming that is typical for modern languages like Elan and ADA. (This description is likely unsufficient for the reader to understand the notion of packet. For a while, this understanding is not necessary, and it will be explained later in full detail.)

The first elementary type we describe implements the whole numbers. The name of this type is `INT` and the conventional name for those whole numbers that can be represented within the computer is *integers.* For each of the types we describe in this and the following chapters we present

- (denotation) how do you write down a value of this type,
- (operations) what operations are specific for this type, and
- a number of examples.

3.1 Integer denotations

In our algorithms we want to indicate in some way that the computer has to manipulate specific whole numbers. In order to indicate such a whole number we have to *denote* it in the algorithm. (Any notion, i.e. numbers too, may have arbitrarily many different representations. To avoid misunderstanding, a group of people agrees upon how to denote a given notion; this is called *denotation.* E.g. in the decimal system *10* means a mathematical notion, namely a given whole number. This same mathematical notion is denoted by *ten* in English, *tien* in Dutch, and *tíz* in Hungarian. This same number can be written as *1010* in the binary and as *A* in the hexadecimal system. The Romans denoted it as *X.* When we count something we often put bars onto paper like 𝍸𝍸 and this may also mean the number ten. While using a programming language its denotations must be applied. Hence, let us see how integers are denoted in Elan.)

The conventional denotation for integers is the decimal notation for (positive) whole numbers, i.e. a sequence of one or more decimal digits. We shall indicate this fact in the form of a *syntax diagram.*

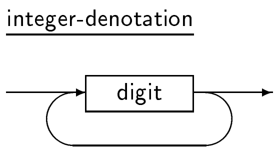


Figure 3.1: Integer denotation

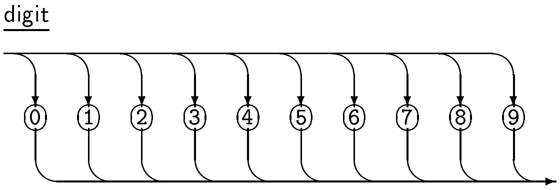


Figure 3.2: Digit

Those two diagrams define the notions integer-denotation (Fig. 3.1) and digit (Fig. 3.2). By starting in the upper left hand corner and going through them keeping to the right and downwards or, following the direction of flow, to the left and upwards we see that an integer-denotation consists of one or more digits, and that a digit is one of the digits 0 to 9.

In later chapters we shall introduce much more complicated constructs with the aid of syntax diagrams. In appendix A of this volume you will find a complete description of the syntax of Elan in another formalism, *context-free grammar,* which is introduced in chapter 11.

The value that is represented by an integer-denotation follows from its usual interpretation as a decimal number. Observe that strictly speaking only positive whole numbers can be expressed with the aid of an integer-denotation; we can prefix it with a + or − but this sign does not form part of the denotation — it is an operator. Observe furthermore that different denotations may represent the same value, e.g. 0023 and 23.

The diagrams given above do not indicate any upper limit to the size of an integer-denotation. Of course in reality there is always a smallest and a largest number that can be represented in the computer. Those whole numbers that can be represented in a (specific) computer are called the *integers* (of that computer). The range between the smallest and largest integer of a computer is called the *integer range* of that computer. In Table **??** we have indicated for a number of computers the smallest and largest whole number that can be represented on that computer.

	smallest	largest integer
calculator.	-10^8+1	$10^8-1=99999999$
32-bits mach.	-2^{31}	$2^{31}-1=2147483647$
16-bits mach.	-2^{15}	$2^{15}-1=32767$

From this table it appears that both positive and negative whole numbers can be represented, and that the limits of the range are in general at or near a power of 2. Observe the asymmetry in many integer ranges. The strange number 32767 holds for most mini- and microcomputers. For didactic reasons, the Elan-0 interpreter has a power of ten as its limit.

The fact that integers are usually represented internally in the binary system (and not in the decimal system) is in itself of no consequence except for the fact that the integer range has somewhat curious limits. However, it is not necessary to understand the binary system in order to work with integers.

Although the integers in a computer have been expressly provided in order to represent the whole numbers there are a number of pitfalls that make it necessary to be cautious in programming with them.

3.1.1 Overflow

During a computation on integers, like, for example, addition, it may occur that the result is not representable on the particular computer. We call this situation *overflow.* Some computers are nice enough to report overflow, others deliver as a result of the addition, without any comment, an integer that has no obvious relationship with the intended answer. For this reason it is necessary to avoid overflow.

Even when the final result of a computation is representable, it may occur that an intermediate result gives rise to overflow and destroys this result. As a consequence, a simple algebraic law like $(a+b)+c=a+(b+c)$ may not always be valid on a computer; it holds only if no overflow occurs.

In this context it may be interesting for the reader to determine what percentage of all possible additions of integers gives rise to overflow, and similarly for multiplications.

3.1.2 Machine dependence

A second source of problems comes from the fact that different computers have different integer ranges. These differences can be quite large. In comparison with the PDP 11 it turns out that the CDC Cyber deals with integers that are about 10^{10} times greater.

In order to make it possible to write programs that can be executed on diverse computers, an Elan program can make use of a constant with the name `maxint` whose value is the greatest concrete whole number representable on this computer. The smallest representable whole number is usually equal to `- maxint`, on some machines it may be one smaller. Thus it is guaranteed that the integer range contains at least the interval `- maxint : maxint`. By an adroit use of this constant it is possible to formulate algorithms in such a way that they are independent of the integer range of the machine used.

3.1.3 Terminology

In order to avoid confusion, the following terms have to be carefully distinguished:

whole number	a certain mathematical concept
integer	a whole number that can be represented in the computer
`INT`	the name of the type of the integers
integer-denotation	the way to denote a (positive) integer.

3.2 Some important concepts

We will now introduce, in the course of an example, a number of important concepts and notations that are not only applicable to integers but that allow the manipulation of all types of objects.

Let start with the example. It is told that Blaise Pascal was already as a boy in school a clever mathematician. One day, when his teacher wanted to keep the class occupied and quiet for some time, he gave to his pupils the task of computing the sum of all numbers from 1 up to 100. Pascal discovered the summation formula for the arithmetic progression and finished suspiciously soon. Of course a modern pupil would, rather than think, go for his pocket computer. Let's do something similar. We let the computer compute this sum, according to the schema

```
begin with the number 1;
begin with the sum zero;
WHILE number <= 100
REP
  add the number to the sum;
  increase the number by one
ENDREP;
show the result.
```

In this schema, `number` and `sum` appear as a *variable*: a name with a fixed meaning and a changeable value. Such a name is introduced by means of a *declaration*; to a variable you may give another value by means of an *assignment.* The result can be displayed by means of an *output algorithm.* For each of those fundamental actions, Elan has a specific notation, specific concrete algorithms belonging to the language.

3.2.1 Declaration of a variable

We use the term *object* for a name, occurring in the program, with which during execution of the program a value is associated. A *variable* is an object with a *name,* a *type* and a changeable *value.* A variable (or another object) may be used in a program only after the execution of an *object declaration.* In order to introduce an integer variable `i` with actual value 1, we have to write

```
INT VAR i:: 1
```

The type of `i` is here given as `INT` which means that `i` can have only integers as values. The sign `::` followed by the expression `1` causes the *initialization* of `i` with the value one. It can be read as "initialized to". Such an initialization is optional; if it is left out the variable has an undefined initial value, as in

```
INT VAR j
```

The declaration of a number of variables of one same type may be combined, as in

```
INT VAR i:: 1, j
```

which is a contraction of the two preceding declarations, or as in the contraction of

```
INT VAR lower; INT VAR upper
```

to

```
INT VAR lower, upper
```

The relevant syntax diagram is given in Fig. 3.2.1.

A *name* ("identifier") is written with (lower case) letters and digits, and has to start with a letter (Fig. 3.2.1). It may contain spaces to enhance readability, but they do form part of the name only in Elan-0 and Elan-1.

The rather boring syntax diagram for letter will be omitted here.

3.2.2 Assignment

The value of a variable can be changed by means of an *assignment* ("assignation"). But beware: the value itself is not modified, the variable just obtains a new value!

As an example, after the previous declaration `i` possesses the value one (Fig. 3.3). Immediately after the

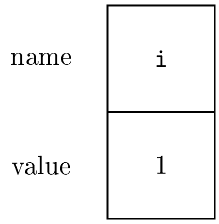


Figure 3.3: Value one

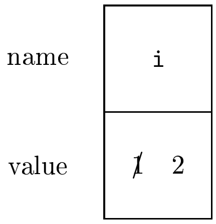


Figure 3.4: Value two

assignment

```
i:= 2
```

`i` possesses the value two (Fig. 3.4).

Such an assignment is written as the "becomes symbol" `:=` with to its left the name of a variable and to its right an expression (Fig. 3.5).

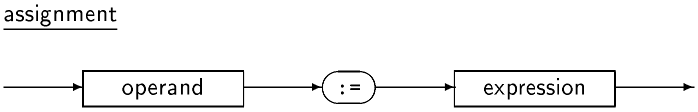


Figure 3.5: Assignment

The notions of *variable* and *expression* will be discussed in detail in section 7.3 and 7.2, respectively.

The effect of the execution of an assignment like

```
v:= expr
```

is as follows:

- the value of the expression `expr` is computed,
- this value is made to be the value of `v`. An eventual previous value of `v` is lost.

3.2.2.1 Examples of assignments

The effect of an assignment can best be studied in a number of examples like the following. For all of them we assume that `a` and `b` are integer variables, declared as

```
INT VAR a, b
```

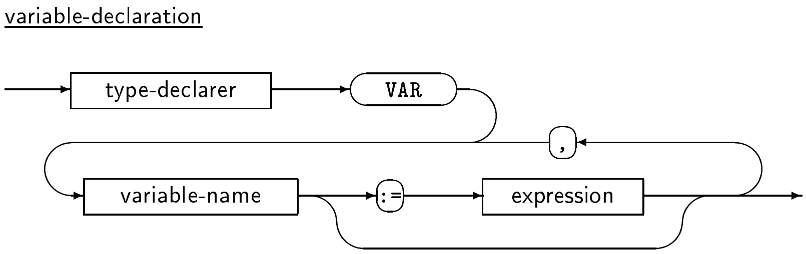
Consider the following assignments:

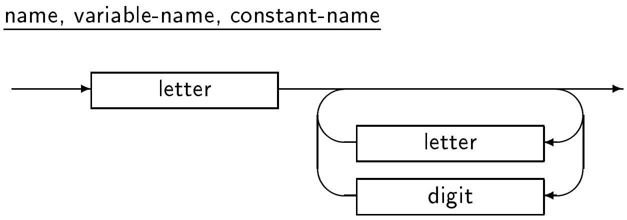
- `a:= 13`

  (Now `a` has the value thirteen.)

- `a:= 13; b := 7`

  (The assignment to `b` does not change the value of `a`.)





- `a:= 13; b := a`

  (Now `a` and `b` have the same value.)

- `a:= 13; b := a - 6`

  (Now `b` is six less than `a`.)

- `a:= 13; a := 14`

  (The effect of the first assignment is lost because of the second.)

- `a:= 13; a := a`

  (The current value of `a` is again assigned to `a`.)

- `a:= 13; a := a + 1`

  (The result of adding the current value of `a` and 1 is assigned to `a`.)

- ```
  a:= 6; b := 7;
  a:= a + b; b := a - b - b;
  a:= a + b; b := a - b - b
  ```

  (Observe that this curious piece of program doubles the initial values of `a` and `b`, provided no overflow occurs.)

3.2.3 Output

In order to make visible the value of a variable or, more in general, an expression you can write

```
put (expression)
```

Here `put` is the name of an *output algorithm* and the expression between the brackets is the *parameter* with which it is called. The value of this expression is computed (yielding, for example, an integer). This value is then written on the output medium in the form of a denotation, possibly preceded by a sign. The output medium is usually the screen of the microcomputer or terminal at which we are working. On this screen a cursor indicates the position where the next output will appear.

Actually it is more difficult to describe exactly the effect of the output algorithm than to try it out, so just sit down in front of the computer and try a very small program like

```
program:
   put (1); put (2 + 3); put (4 - 10).
```

Observe how the cursor moves over the screen during writing. If you attempt to write too much on one line, the cursor leaves this line and continues at the beginning of the next line. Such a move to a new line can be obtained also by a call of the algorithm `line`. In this way, a nicer layout of the screen can be achieved than by writing haphazardly on the screen. It is possible to give more than one new line by calling the algorithm with an integer parameter, like

```
line (5)
```

A call of `line` with the parameter `1` has the same effect as a call without a parameter.

3.2.4 Input

It is possible to input numbers during the execution of a program, by means of the input algorithm `get`, with a variable as its argument. Let `number` be an integer variable whose value we want to input. The call

```
get (number)
```

has the following effect:

- The execution of the program is stopped while the computer waits until a line has been typed in at the keyboard followed by the RETURN- or ENTER-key. The characters typed in appear on the screen at the current cursor position.
- This line should contain an integer denotation, possibly preceded by a plus or minus sign. The value of that (signed) denotation is computed.
- This value is assigned to the variable `number`.

Actually things are more complicated, because on a line more than one denotation may appear. The first call of `get` waits until the line has been fully typed
```

in, but the next call can then proceed with the next denotation without further waiting, until the line has been completely processed, after which a further call of `get` has to wait for input again.

3.2.5 Continuation of the example

In order to apply all this we shall now write a complete program, which computes the sum of the numbers `1` to `max`, in which `max` is a variable we read in.

```
problem of pascal:
   read the maximum;
   start with the number 1;
   begin with the sum 0;
   WHILE number <= max
   REP add number to sum;
      take the next number
   ENDREP;
   show the result.
```

We shall now refine each of the subalgorithms.

```
read the maximum:
   INT VAR max;
   get (max).

start with the number 1:
   INT VAR number:: 1.

start with the sum 0:
   INT VAR sum:: 0.

add number to sum:
   sum:= sum + number.

take the next number:
   number:= number + 1.

show the result:
   line;
   put (sum).
```

We have followed very closely the formulation of the algorithm given at the beginning of the chapter, which results in a rather large number of small refinements. In learning to program it does not hurt to be overly explicit. The consequence is that the algorithm is easy to follow but rather lengthy. This doesn't mean that all programs should be refined to this level of detail. Refinements are means to direct the creative thinking in programming and to fix the intermediate stages of that thinking. As our programming experience grows, somewhat greater leaps of imagination become possible, but initially we try to refine as clearly as possible, in such a way that the solution reflects very explicitly the thoughts of the programmer.

3.2.6 Declaration of a constant

Besides variables there also exist *constants,* objects with (after their declaration) a fixed unchangeable value. A constant is always initialized in its declaration (Fig. 3.6).

There may be all kinds of reasons to give a specific name to a value and use that name rather than the denotation: it may be that the value by itself is uninteresting but its importance lies in the role it plays. For example, on a specific machine the constant `maxint`, which is already known to us, may have a declaration like

```
INT CONST maxint:: 2147483647
```

This is the value which `maxint` has on an IBM 370 computer. On other machines this constant might have another value but we would still call it `maxint`. A declaration of an integer constant that might be of use in a chess program is

```
INT CONST number of fields:: 8 * 8
```

Again the declaration for a number of integer constants can be combined into one composed declaration. Example:

```
INT CONST ace:: 14, king:: 13, queen:: 12,
jack:: 11
```

Once a constant has been declared it has an unchangeable value. In distinction to variables it is not possible to assign to a constant. Not even by accident. This too may be a reason to use a constant rather than a variable wherever applicable. As a rule of thumb: if we do not wish to assign to an object it had better be a constant.

The difference between variables and constants is only a difference in *access*: from both a variable and a constant a value can be obtained ("read access"), but only to a variable can a new value be assigned ("write access").

3.3 Integer operations

In the programming language Elan a number of concrete algorithms for the manipulation of integers are given that do not need a further declaration in a program. These have been declared in a *standard packet,* a box full of useful algorithms, objects and types given to you for free (see appendix B).

3.3.1 Arithmetic operations

To begin with, we have the well-known arithmetical operations (Table 3.1).

operator	meaning	example	result
+	addition	1 + 1	(= 2)
−	subtraction	5 − 9	(= −4)
*	multiplication	3 * 4	(= 12)
DIV	division	13 DIV 6	(= 2) !!
MOD	rest	13 MOD 6	(= 1)
**	to the power	5 ** 2	(= 25)

Table 3.1: Arithmetical operations on integers

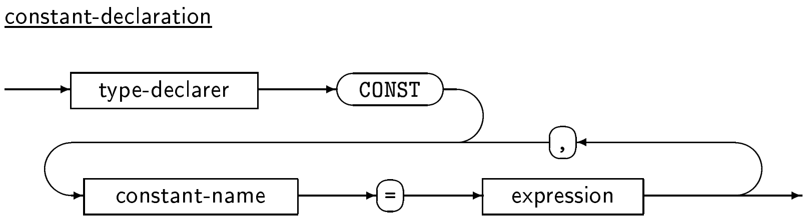


Figure 3.6: Constant declaration

Besides these *dyadic* integer operators, which have two integers as operands and an integer result, the `+` and the `-` also exist as *monadic* integer operators, that is having one integer operand and with an integer result. Examples:

```
+ 8
- 4
```

For positive arguments the result of the modulo operator `MOD` is always positive and smaller than the nominator. The operators `DIV` and `MOD` should not be used with denominator zero, otherwise overflow occurs. They have the following properties:

$$\mathtt{xDIV} - \mathtt{y} = -(\mathtt{xDIVy}) \tag{3.1}$$

$$\mathtt{xMOD} - \mathtt{y} = \mathtt{xMODy} \tag{3.2}$$

$$(-\mathtt{x})\mathtt{DIVy} = -(\mathtt{xDIVy}) \tag{3.3}$$

$$(-\mathtt{x})\mathtt{MODy} = -(\mathtt{xMODy}) \tag{3.4}$$

3.3.2 Expressions

With the aid of operators, operands and brackets, expressions ("formulae") can be formed in the usual manner, keeping in mind that each of the operators has its own priority ("precedence"). Raising to the power has a higher priority than multiplication and division, which in turn have a higher priority than addition and subtraction. Priorities being equal, the operations are executed from left to right. Examples:

```
a * b + c DIV d = (a * b) + (c DIV d)
a DIV b * c     = (a DIV b) * c
```

The monadic operators always have the highest priority. Examples:

```
  3 * - 1  (= -3)
- 1 **  2  (= +1)
```
watch out !!

Because of the high priority of the monadic operator `-` this last expression yields the value `+1` and not `-1` as one might expect.

If desired, another order can be indicated by means of brackets, as in:

```
- (1 ** 2) (= -1)
```

3.3.3 Comparison operators

There exist a number of dyadic operators having two integer arguments and a truth value as result. They are shown in Table 3.2.

operator	meaning
<	less than
<=	at most
>	greater than
>=	at least
=	equal to
<>	unequal to

Table 3.2: Comparison operations on integers

These can be used in a condition like

```
x <= 0
```

The priorities of these comparison operators are lower than that of the arithmetic operators, so that `a + 1 <= x` means the same as `(a + 1) <= x`.

3.3.4 Operations combined with assignment

The combination of an addition with an assignment to an integer variable as in `x:= x + 1` appears so frequently that a shorter way of writing has been introduced. Using this, the assignment

```
subtotal:= subtotal + term
```

can be written as

```
subtotal INCR term
```

This is not only shorter but it also makes more explicit the intention to increment the variable (it is not just any old assignment). The assigning operator `INCR` needs as its left operand an integer variable. Its right operand can be any integer expression.

Analogously, there also exists an operator `DECR` to indicate subtraction combined with assignment, as in

```
income DECR expenses
```

which means the same as

```
income:= income - expenses
```

These operators are very useful, particularly if the variable has a long name.

3.4 Example: Maximum of a series of whole numbers

As an example of the use of these concepts, we shall address the problem of finding the largest of a nonempty sequence of positive whole numbers.

We assume that the numbers are input via the standard input medium (the keyboard). The amount of numbers that will arrive is not known beforehand, but it is at least one (the sequence is not empty). The end of the sequence is indicated by a negative number. We wish to determine the maximum, i.e. find the value of the largest number.

The algorithm which solves our problem is based on the idea of reading, after some preparation, the numbers one by one, and remembering at each moment the largest one found up to that point. In this way we have obtained at the end the largest one of all the numbers read.

```
determining the maximum:
   take zero as initial maximum;
   read the first number;
   REP
      look if it is larger;
      read the next number
   UNTIL all positive numbers considered
   ENDREP;
   write the definitive maximum.
```

We use a variable to build up the current maximum. Before the repetition, this variable has to be initialized. Two strategies offer themselves:

- initialization to zero, which is the maximum of zero positive numbers;
- initialization to the first number read, because the maximum is at least equal to that number.

We shall follow the first strategy.

```
take zero as initial maximum:
   INT VAR current maximum:: 0.
```

Observe that frequently a refinement can be realized as an initialized declaration. It serves to establish a well-defined situation, which can easily be given a name.

```
read the first number:
   INT VAR number;
   get (number).

look if it is larger:
   IF number > current maximum
   THEN
      current maximum:= number
   FI.

read the next number:
   line;
   get (number).
```

Because of the call of `line`, each number which is read in will appear on a separate line.

```
all positive numbers considered:
   number < 0.

write the final maximum:
   line (3);
   put (current maximum);
   line.
```

By means of the call of `line` we have put the solution on a line of its own.

The idea behind this algorithm is the fact that the assertion

> *the value of the variable* `current maximum` *is the maximum of all numbers read until now*

is true after every step of the repetition. Before the repetition this assertion is true because the maximum of an empty sequence of positive numbers is 0. When the repetition is completed all numbers have been read and the assertion is still true. Therefore the `current maximum` at that point is equal to the final maximum. This property is also reflected in the choice of the name `current maximum` for the variable, which might also have been called `maximum up to now` or something like that.

The termination of the algorithm is assured by the fact that in every step of the repetition a number is read, while only a finite amount of numbers will be input before a negative number is given.

Exercise

What would the solution look like according to the second strategy **(take the first number as initial maximum)?**

3.5 Example: Fibonacci numbers

It is told that in the year 1202 the Italian mathematician Leonardo Pisano, also known as Leonardo Fibonacci (Filius Bonaccio), solved the important economic problem:

> *How many pairs of rabbits can be produced in the course of one year, starting from one pair of rabbits?*

Of course he made some simplifying assumptions. Each pair of mature rabbits produces one pair of young each month, one of which is male and the other is female. A pair of newborn rabbits after one month is able to procreate similarly. Rabbits never die unless they are eaten. (Obviously in this world of rabbits things go differently than in the world of humans. For more information about the life of rabbits we refer to [ADA72].

The solution to this problem is found as follows. Assume the number of rabbit pairs in month n to be $F(n)$. (The letter F is used here in commemoration of Fibonacci.) Assume that among these $F(n)$ pairs of rabbits the number of mature ones is $V(n)$. What is the number of pairs in month $n+1$, $F(n+1)$? In the month $n+1$ there will be $F(n)$ mature pairs of rabbits because the number of mature pairs in a specific month is equal to the total number of pairs in the preceding month. Moreover, each of the $V(n)$ mature rabbit pairs have produced a pair of young, so that $F(n+1) = F(n) + V(n) = F(n) + F(n-1)$. When we assume that our pair of rabbits in month 1 is born in that month (the number of rabbits in the preceding month was zero, presumably because they were all eaten), then the problem can be formulated as:

Find the value of $F(12)$ when it is given that

$$F(0) = 0$$

$$F(1) = 1$$

and

$$F(n+1) = F(n) + F(n-1) \text{ for } n \geq 1.$$

After some computation we find that master Fibonacci, starting in January with one pair of young rabbits, could feast by Christmas on one hundred and forty four pairs of rabbits.

We shall now investigate the problem of finding, with the aid of the relationship for the Fibonacci numbers given above, the limit of the ratio between two consecutive numbers. In order to get some insight into the behaviour of Fibonacci numbers we will first deduce the value of this limit.

Let us call the limit of $F(n)/F(n-1)$ for n to infinity F. We already know that $F(n) = F(n-1) + F(n-2)$, so for sufficiently large n

$$F(n-1) = F(n)/F,$$

$$F(n-2) = F(n-1)/F = F(n)/(F * F),$$

and therefore

$$F(n) = F(n)/F + F(n)/(F * F)$$

which can be simplified to

$$F^2 - F - 1 = 0$$

from which we deduce that

$$F = \frac{F(n)}{F(n-1)} = \frac{1 \pm \sqrt{5}}{2}$$

for n to infinity.

We choose the positive solution (= 1.61803...), since because of our choice of $F(0)$ and $F(1)$ all $F(n)$ are positive. Therefore each Fibonacci number is about 1.6 times larger than the preceding Fibonacci number.

Considering for simplicity the Fibonacci numbers as a geometric sequence with this limit as its ratio, it follows that the largest Fibonacci number which is still representable on the IBM 370 computer (with `maxint = 2147483647`) would be $F(43)$. Through experiment we can find out that this estimate is too pessimistic and that even $F(46)$ can still be represented for this value of `maxint`.

An algorithm that computes the first 46 Fibonacci numbers and prints their ratios can be formulated as follows (in order to prevent problems with overflow, we shall stop at the 46th Fibonacci number):

```
initialize;
WHILE not yet last number
REP
   compute next number;
   print number and ratio
ENDREP.
```

From the relationship for Fibonacci numbers it follows that in order to compute $F(n+1)$ we need only the two preceding numbers $F(n)$ and $F(n-1)$; other Fibonacci numbers need not be remembered. In this algorithm we shall call those two terms the variables `last number` and `last number but one`. Also we need a variable to indicate the sequence number of the number computed. The initial values of `last number`, `last number but one` and `sequence number` follow from $F(0) = 0$ and $F(1) = 1$. We obtain:

```
initialize:
   INT VAR last but one:: 0, last:: 1,
           sequence number:: 1.
```

We can continue as long as the `sequence number` is smaller than 46.

```
not yet last number:
   sequence number < 46.
```

The action `compute next number` has to establish the relationship $F(n+1) = F(n) + F(n-1)$. In each step of the repetition, `last` has to be the newly computed number of the previous step and the `last but one` has to be the `last` of the previous step. For this shifting of values we need a temporary name to indicate the value of the newly computed number. This value is constant during the shift.

```
compute next number:
   INT CONST new::
           last + last but one;
   last but one:= last ;
   last:= new ;
   sequence number INCR 1.
```

Observe that the order of the first three lines is crucial.

We will print the `sequence number`, `last` and the ratio on a new line.

```
print number and ratio:
   line;
   put(sequence number);
   put(last);
   put( real(last) / real(last but one) ).
```

In the expression given the ratio between the `last` and `the last but one` we have (rather prematurely) used the division operator `/` and the conversion algorithm `real`, which yield as their result a real number. We shall discuss real numbers and their operators in the next chapter.

The program given above has the drawback that it is suitable only for computers whose `maxint` is at least 2147483647. Therefore it is better, rather than to count until a specific sequence number, to try to find precisely the largest Fibonacci number which is representable on our computer. We can achieve this by continuing the computation of the new number as long as no overflow can arise and stopping just before overflow would arise. We need only modify the algorithm `not yet last number`. There will be no overflow as long as

```
last + last but one <= maxint
```

But we cannot use this test since it might cause exactly the overflow we want to avoid! We therefore write:

```
not yet last number:
   last <= maxint - last but one.
```

which gives the same result without causing overflow.

Table 3.3 has been produced by the thus modified program on an IBM 370 computer. We see that the computed ratio does not change after sequence number 40. From this, however, we cannot deduce that that number is really the limit: a computer computes real numbers with some imprecision, as we will see in the next chapter.

From [KNU72] we quote the limit truncated to 41 decimal digits:

1.61803398874989484820458683436563811772030

We see that the ratio found by us is somewhat too low in the last decimal.

In this example we proved that for large values of n the Fibonacci numbers behave as a geometric sequence. In higher mathematics such seqences are called asymptotically geometric seqences.

The Fibonacci numbers as defined by the equations above play an important role in all kinds of natural processes. As an example, the limit of the ratio between two consecutive Fibonacci numbers is a number which is called the *Golden ratio.* In botany the Fibonacci numbers appear in the Phyllotaxis (the ordering of leaves along a stem), and we have just dealt with a problem from zoology.

3.6 Network of types and operations

In this chapter, we have introduced a number of operations (procedures and operators) on integers which are joint to other operations (often with the same name) on other types which will be introduced later. How can we keep an overview of the operations belonging to a type, otherwise than by some enumeration?

A very telling representation is the *Network of Types and Operations* (NTO) [CRA87]: a type is represented as an oval, inscribed with the name of that type, and an operator or procedure as a rectangle, again inscribed with its name. These ovals and rectangles are connected by arrows:

- An arrow from an oval *(type)* to a rectangle *(operation)* indicates that that operation has an argument of that type. In case an operation has more

2	1	1.0000000000000000
3	2	2.0000000000000000
4	3	1.5000000000000000
5	5	1.6666666666666665
6	8	1.5999999999999999
7	13	1.6250000000000000
8	21	1.6153846153846152
9	34	1.6190476190476188
10	55	1.6176470588235292
11	89	1.6181818181818182
12	144	1.6179775280898876
13	233	1.6180555555555554
14	377	1.6180257510729612
15	610	1.6180371352785146
16	987	1.6180327868852458
17	1597	1.6180344478216817
18	2584	1.6180338134001251
19	4181	1.6180340557275541
20	6765	1.6180339631667064
21	10946	1.6180339985218033
22	17711	1.6180339850173577
23	28657	1.6180339901755969
24	46368	1.6180339882053250
25	75025	1.6180339889579018
26	121393	1.6180339886704431
27	196418	1.6180339887802426
28	317811	1.6180339887383028
29	514229	1.6180339887543225
30	832040	1.6180339887482036
31	1346269	1.6180339887505408
32	2178309	1.6180339887496480
33	3524578	1.6180339887499890
34	5702887	1.6180339887498587
35	9227465	1.6180339887499084
36	14930352	1.6180339887498896
37	24157817	1.6180339887498967
38	39088169	1.6180339887498940
39	63245986	1.6180339887498951
40	102334155	1.6180339887498947
41	165580141	1.6180339887498947
42	267914296	1.6180339887498947
43	433494437	1.6180339887498947
44	701408733	1.6180339887498947
45	1134903170	1.6180339887498947
46	1836311903	1.6180339887498947

Table 3.3: Fibonacci numbers and their ratios

than one argument, we will also draw more than one arrow to that particular rectangle, in which the arguments are indicated strictly from left to right.

- An arrow from a rectangle *(operation)* to an oval *(type)* means that either that operation yields a result of that type or it changes the value of its argument. In the latter case the access right of that argument must be VAR which is indicated by a double arrow.

- A double-headed arrow from an oval *(type)* to a rectangle *(operation)* and back indicates that that operation awaits an argument which must have a value and it modifies this value. This means that the access right of this argument must be VAR which again is indicated by a double arrow.

Notice that an NTO is a very convenient notation for operators, and also for procedures with a couple of parameters. However, if the number of parameters exceeds three such a graphical representation is not really useful.

Below and in the following chapters, the signatures of the four basic types in the form of NTO's will be presented. However, none of the NTO's will contain the *denotation* of that type or the *assignation* (:=) and *initialisation* (::) as these are always present with the type.

By combining the NTO's of the different types you may, if you wish, obtain the complete picture.

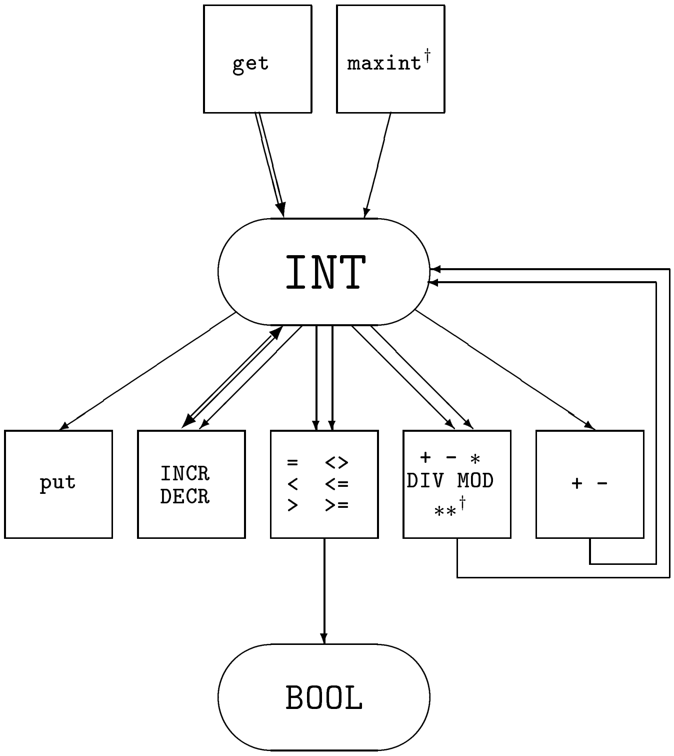
3.6.1 NTO of the integers

The NTO for the type INT has to be composed of a number of parts:

- the monadic +, -;
- the dyadic +, -, *, DIV, MOD, **, INCR and DECR;
- the comparison operators =, <>, <, <=, > and >=;
- the output procedure put;
- the input procedure get;
- the integer constant maxint.

All these parts can be found back in the scheme of Fig. 3.7. This NTO in its turn forms part of a larger framework together with other NTO's which we have not included in this picture.

For a complete set of operations on integers see appendix B.



† Not Elan-0

Figure 3.7: NTO of integers

3.7 Exercises

1. Find a sequence of assignments that interchanges the value of two variables a and b. Can this also be done without the use of an auxiliary variable (mind the overflow)?

2. (Interval sum) Write a program that finds, for a given number s, all pairs of integers i and j with $1 \leq i \leq j$ such that the sum of all numbers in the interval i up to and including j is equal to s.

3. (Counting grid points) Read an integer r and count how many points of the unit grid fall within the circle with radius r, i.e. how many pairs (x, y) exist with $x^2 + y^2 \leq r^2$.

4. (Maximum sum of divisors) Determine that number in the interval $[a : b]$ with $1 \leq a \leq b$ for which the sum of divisors is maximal. Note that 1 and the number itself are also divisors.

5. (Pythagoras numbers) Determine for a given $k > 0$ all numbers i and j with $0 < i \leq j$ such that $i^2 + j^2 = k^2$.

6. (Pythagoras numbers) Find all pairs (i, j) with $1 \leq i \leq i_{max}$ and $1 \leq j \leq j_{max}$ that form Pythagoras numbers (i.e. $i^2 + j^2$ is a square number).

7. (Ulam's rules) The number theoretician Ulam established that a sequence of natural numbers always ends up with 1 if, starting with any nonzero number, the following rules are applied repeatedly:

   - an even number is divided by 2,
   - an odd number is multiplied by 3 and then incremented by 1.

Write a program executing Ulam's rules and observe its behaviour.

8. (Perfect numbers) Numbers equal to the sum of their true divisors are called perfect. Write an efficient program producing all perfect numbers less than one million (ten thousand with Elan0).

9. (Friendly numbers) Let $s(n)$ denote the sum of the true divisors of a number n. If for the numbers n_1 and n_2 $s(n_1) = n_2$ and $s(n_2) = n_1$, i.e. $s(s(n_1) = n_1$, then n_1 and n_2 are called friendly numbers.

   Write an efficient program producing all friendly numbers less than one million (ten thousand with Elan0). Mind the symmetric pairs!

10. (Related numbers) Again, let $s(n)$ denote the sum of the true divisors of a number n. If for the numbers $n_1, n_2, n_3, \ldots, n_k$ $s(n_1) = n_2, s(n_2) = n_3, \ldots, s(n_k) = n_1$ then $n_1, n_2, \ldots, n_k$ are called related numbers.

    Write an efficient program producing all related numbers less than one million (ten thousand with Elan0) for $k = 1, 2, \ldots$. Mind the permutations!

11. (Five sailors, many coconuts and a monkey) Five sailors are shipwrecked at a small island, together with their monkey. To their good luck, the island is rich in coconuts, of which they collect a big pile before they retire to rest. One of the sailors wakes up in the night, feels hungry, gives one nut to the monkey and then honestly takes his fifth; this division is possible without rest. When the second sailor wakes up and feels hungry he proceeds similarly, and so do the others; the division is in every case possible without rest. When they come together later in the morning, although wondering at the tiny pile, they divide equally, and also without rest, what they have found — but this time leaving out the monkey [KLI85].

    Write a program to solve this difficult Diophantine equation and answer the question: how many coconuts did the sailors collect on the previous day?

Chapter 4

The real numbers

The next elementary type implements the real numbers. The name of this type is `REAL`, and its values are customarily called *reals.* Real numbers play a role in all kinds of physical and technical computations; for this reason, reals are available in practically every programming language.

4.1 The denotation of reals

The customary denotation for reals is a decimal notation. It consists of a sequence of decimal digits in which a decimal period should occur and which may be followed by a part indicating a power of ten.

The syntax diagrams for real denotations are shown in Fig. 4.1, 4.2 and 4.3.

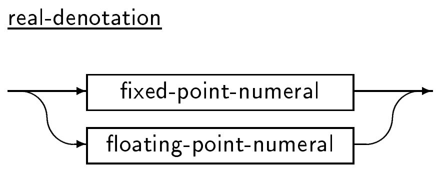


Figure 4.1: Real-denotation

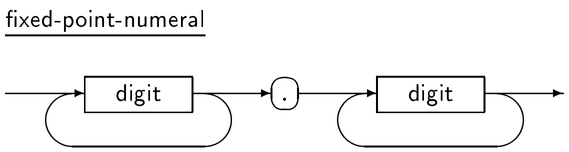


Figure 4.2: Fixed-point-numeral

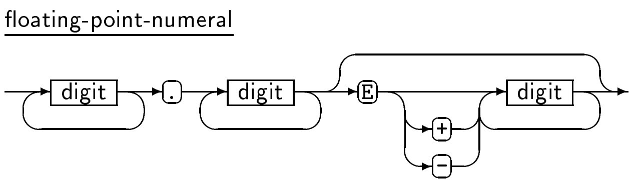


Figure 4.3: Floating-point-numeral

Some examples of real denotations are

```
1.432E-19    1.0    1234.0    0.001234
```

whereas the following notations have a strong resemblance to them, but are not correct according to the syntax diagrams (check this!).

```
1.E+2      0001E2      E.2      3.
```

The value represented by a real denotation is its usual interpretation as a decimal number; the exponent indicates a power of ten as a scale factor.

Just as in the case of integer denotations, negative reals can be denoted by a denotation for a positive real, preceded by the operator `-`.

One same real value can be denoted in a number of different ways. For example a certain approximation of the number π might be denoted as `3.141592654` or `0.3141592654E+001` or `314159265.4E-8`.

4.1.1 Precision errors

The following list of properties of reals makes clear that they are but a meagre substitute for true real numbers.

- Reals are represented internally not in base 10 but generally in base 2. Of course this makes no difference for most properties of the real numbers, but it makes the distance between our conceptions and the realities of the computer noticeable.
- The number of binary digits ("bits") that a computer uses for the internal representation of a real is limited and differs from computer to computer.
- The number of binary digits used for the internal representation of the exponent is also limited and different.

These facts have a number of consequences.

- Within a particular computer, only a finite number of real numbers can be represented exactly or, more precisely, only a finite subset of the rational numbers. Maybe the reals could better have been called "rats": rational numbers.
- All other real numbers within a specific range are represented approximately by one of the exactly representable rational numbers.
- Therefore the value of this representation differs by some amount from the intended real number. This difference is called the *absolute representation error.*

- In representing the result of a real operation, the result is often truncated to some number of bits, giving rise to *rounding errors.*
- Because of the representation of real numbers as floating point numbers, the absolute representation error is large for those real numbers that have a large absolute value. (In consequence of the limited precision the least significant digits get lost.)
- When a computer internally uses the binary system, many decimal fractions like 0.1 are not exactly representable. This has as a consequence that on such computers $10*0.1$ is not equal to 1.0.

4.1.2 Machine dependence

The properties of reals differ from computer to computer. In order to allow you to formulate programs involving reals that will work on any computer, in the standard packets two constants have been defined.

The constant `smallreal` is the (in absolute value) smallest positive real that, when added to 1.0, gives a result differing from 1.0. Therefore the number `smallreal` is a measure for the *relative precision.*

The constant `maxreal` is the largest real that can still be represented in the computer. The number `maxreal` therefore gives an idea of the *range* of reals in the computer.

Table 4.1 gives, for a number of computers, the values of `smallreal` and `maxreal`. The notation used in the table also indicates the internal number system.

	smallreal	maxreal	
CDC Cyber	2^{-47}	$(2^{-48}-1)$	$*(2^{1022})$
SIEMENS 2002	10^{-12}	$(1-10^{-10})$	$*(10^{49})$
IBM 370	16^{-6}	$(1-16^{-6})$	$*(16^{63})$
DEC 20	2^{-63}	$(1-2^{-63})$	$*(2^{127})$
PDP 11	2^{23}	$(1-2^{-23})$	$*(2^{127})$

Table 4.1: `smallreal` and `maxreal` on some computers

4.1.3 Overflow and underflow

Apart from overflow, which occurs when a computed result is larger than `maxreal`, in real arithmetics also *underflow* can occur, when a result is too small to be represented, e.g. when the exponent is more negative than the number representation allows. In that case the result may be rounded to zero. That is somewhat less tragic than overflow.

4.1.4 Terminology

Notice the distinctions between the following terms:

real number a mathematical concept;

real a rational number, representable in the language, used as an approximation to a real number;

`REAL` the name of the type of the reals;

`real denotation` the denotation for (positive) reals in a program.

4.2 Real constants and variables

Declarations for real constants and variables have the same form as those for integer constants and variables, apart from the fact that they start with the type name `REAL`.

The following constants are standard in Elan and therefore need not be declared by the programmer.

```
REAL CONST maxreal  :: very large,
           smallreal:: very small,
           pi:: 3.141592653589793238 ...;
```

The constants `maxreal` and `smallreal` have been discussed in the previous paragraph. The constant `pi` is the well-known ratio between the circumference and diameter of a circle.

Notice that Elan, like so many programming languages, does not have one general notion of number but comprises two different concepts, reals and integers, that are much more related to the construction of the central processing unit in computers than to our expectations. As an example, `12` is an integer denotation and therefore is not a real denotation. Since in an assignment the types of left-hand and right-hand side have to be equal, `12` cannot be assigned to a real variable. Therefore we cannot write:

```
REAL VAR x; x:= 12
```

but have to write

```
REAL VAR x; x:= 12.0
```

4.3 Operations on reals

4.3.1 Arithmetical operations

Reals are especially intended for performing computations. All the usual arithmetic operations for real numbers are therefore included among the standard operators in Elan (Table 4.2).

operator	meaning	example	result
+	addition	1.0 + 3.0	(= 4.0)
−	subtraction	7.5 − 1.1	(= 6.4)
*	multiplication	2.0 * 4.7	(= 9.4)
/	division	3.8 / 2.0	(= 1.9)
MOD	rest	3.4 MOD 2.0	(= 1.0)

Table 4.2: Arithmetical operations on reals

These dyadic operators are defined for real operands and yield a real result.

The priority of these arithmetical operators for reals is the same as those for integers; the priority is a property of the operator which is inherited with its name.

Exponentiation has as its left operand a real and as its right operand an integer which has to be larger than zero (Table 4.3).

operator	meaning	example	result
**	exponentiation	−1.3 ** 2	(= 1.69)

Table 4.3: Exponentiation

4.3.2 Real functions

The *functions*, listed in Table 4.4, with one real argument and a real result are also available in Elan:

Name	Meaning
`sqrt`	taking the square root
`exp`	e-to-the-power
`ln`	the natural logarithm
`sin`	the sine function
`cos`	the cosine function
`tan`	the tangent function
`arcsin`	the arcsine function
`arccos`	the arccosine function
`arctan`	the arctangent function
`abs`	taking the absolute value

Table 4.4: Real functions

The six trigonometric functions expect their argument to be in radians, respectively deliver their result in radians. These are not operators but *procedures* whose argument has to be placed between the brackets `(` and `)`. (Procedures will be discussed in detail in chapter 10). A function call can appear as an operand in an expression. Examples:

```
sin (x) ** 2 + cos (x) ** 2
exp (1.0)
```

Some of these functions have limitations on their arguments that are customary in mathematics. As an example, the function `sqrt` ("square root") is defined for nonnegative arguments only. Therefore never try to take the square root of a negative number.

4.3.3 Comparison operators

Two reals can be compared with the aid of the well-known operators:

```
=   <=   <   >   >=   <>
```

Be careful in comparing reals: owing to the possibility of rounding errors, the representation of a result depends strongly on its history, and it is possible that an expected equality does not occur. Instead of `a = b` one should rather write

```
abs(a - b) < threshold
```

with a sufficiently small value for `threshold`.

4.3.4 Conversion operations

The functions, listed in Table 4.5, have been introduced to allow the conversion from an integer to an equivalent real number or from a real number to an integer value.

The function `round` yields an integer result, the nearest integer.

Name	Meaning
`real`	to convert an integer into an equivalent real
`round`	for rounding a real number to the nearest integer
`trunc`	for truncating a real number down to the nearest integer

Table 4.5: Converting functions

```
round  (4.3) =  4.0
round  (4.8) =  5.0
round (-4.3) = -4.0
round (-4.8) = -5.0
```

A real number ending on .5 is supposed to be rounded up, but of course if it ends on .4999999999 it may be rounded downwards.

The function `trunc` delivers the greatest integer which does not exceed its operand.

```
trunc  (4.3) =  4.0
trunc  (4.8) =  4.0
trunc (-4.3) = -5.0
trunc (-4.8) = -5.0
```

Because of the inherent imprecision of the reals you can not find out easily that an expression like `trunc (10.0 * 0.1)` yields zero or one. Its explanation is the following. The decimal fraction 0.1 is an infinite fraction of the form 0.0001010101... in the binary system and is normalized as $0.1010101\ldots * 2^{-3}$. The computer, of course, can store a finite number of digits only. Suppose the computer represents the mantissa with an even number of bits which means that the first omitted bit is a 1. When the computer rounds the number according to the omitted bits the mantissa will end up in ...011; therefore the number will be larger than 0.1 and its tenfold larger than 1! In this case the above expression yields 1. On the other hand, if the computer does not round the mantissa then the number will be less than 0.1 and its tenfold less than 1. Thus, the expression results 0.

An important application of conversion operators is to allow arithmetic on operands of differing type, for instance `REAL` and `INT`:

```
INT VAR k:: 0;
REAL CONST dx:: 0.001;
WHILE k <= 100
REP
   k INCR 1;
   REAL const x:: real(k) * dx
   ...
ENDREP
```

4.3.5 Input and output of reals

Just like integers, reals can be read and written by means of the concrete algorithms `get` and `put`. When called with a real variable as its argument, the procedure `get` reads a number in the same form as a real denotation, possibly preceded by a sign. Similarly, `put` with a real argument outputs a denotation for that argument, possibly preceded by a `-` sign and some spaces. (So far, we used the names `get` and `put` to identify concrete algorithms of Elan and now, suddenly, we say `get` is the name of a procedure. There is no contradiction since procedures, like programs and operations, are algorithms and in a sense the term *procedure* is used as a synonym for algorithm. The notion of procedure will be defined more precisely in chapter 10).

The number of digits written by `put` depends on the precision with which the computer represents reals. This number is in any case sufficient to show all significant digits of a real, but you have to be aware of the possibility that rounding errors have occurred. Therefore do not be surprised when you have computed with great labour a result which should be equal to two and it is printed as `0.199999994E+1` or some such.

4.4 Example: The roots of a quadratic equation

Although the reals are specially intended to perform computations with real numbers, in practical computation a great deal of prudence has to be exercised. Reals form only a meagre realization of the real numbers. We cannot just take a formula and turn it into a program. In many cases its result will be rather different from what we expect, especially when intermediate results approach the precision limits of the reals. We will illustrate this with a shocking example.

A well-known formula for the roots of a (non-degenerate) quadratic equation is the abc-formula. The equation

$$ax^2 + bx + c = 0$$

has as roots, according to your highschool mathematics

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The formulation of an algorithm to compute these roots for a number of values for a, b and c is an exercise in the linear notation of formulae.

```
roots of quadratic:
   read the coefficients;
   see if there are roots.

read the coefficients:
   REAL VAR a, b, c;
   line;
   put ("                      2");
   line;
   put ("The roots of        ax + bx +
c");
   line (2);
   put ("Give a: ");
   get (a);
   line;
   put ("Give b: ");
   get (b);
   line;
   put ("Give c: ");
   get (c);
   line.
```

We have used here the procedure `put` with a text as its parameter which of course we do not introduce until chapter 6. We hope its use is self-explanatory: a call like `put("Give a: ")` writes the text `Give a:` (without the quotes) on the screen, starting at the current position.

```
see if there are roots:
   IF the equation is degenerate
   THEN
     compute roots of degenerate equation
   ELSE
     IF discriminant is negative
     THEN
       report that there are no real roots
     ELSE
       compute the roots;
       print them
     FI
   FI.

the equation is degenerate:
   a = 0.0.

compute roots of degenerate equation:
   IF b = 0.0
   THEN
     IF c = 0.0
     THEN
       report that there are too many
roots
     ELSE
       report that there are no roots
     FI
   ELSE
     give the only root
   FI.

report that there are too many roots:
   put ("There are too many roots.");
   line.

report that there are no roots:
   put ("There are no roots.");
   line.
```

```
report that there are no real roots:
   put ("There are no real roots.");
   line.

give the only root:
   put ("The only root is: x = ");
   put ( - c / b);
   line.

discriminant is negative:
   REAL CONST discriminant:: b * b - 4.0 *
a * c;
   discriminant < 0.0.

compute the roots:
   REAL CONST x1:: (-b -
sqrt(discriminant)) / (2.0 * a);
   REAL CONST x2:: (-b +
sqrt(discriminant)) / (2.0 * a).

print them:
   line;
   put ("x1 = ");
   put (x1);
   line;
   put ("x2 = ");
   put (x2);
   line.
```

In Table 4.6 we have given for the quadratic equation

$$(x - 10.0^i) * (x - 1.0) = 0 \text{ for } i = 1, 2, 3, \ldots$$

the results according to this algorithm in the left column and the correct values in the right column. The computations have been done on a computer with

`smallreal = 1.1920929E-7` and `maxreal = 1.7014117E+38`.

We observe that beyond $i = 7$ according to the abc-formula the value of the smallest root is 0 instead of 1! These strange results are not the fault of the algorithm, but are due to the fact that we use reals rather than real numbers. We see that the smallest root can have a large relative error, especially if the roots of the equation differ greatly in absolute value. This is easy to explain when we note that in that case a and c are small with respect to b. Therefore b and the square root of the discriminant $\sqrt{(b^2 - 4ac)}$ will be about equal to one another. Owing to the limited precision, the difference between those two values will therefore have a large relative error.

We get better results by using the following mathematically equivalent formulae for obtaining the roots of this quadratic equation:

$$x_1 = -\frac{\frac{2c}{b}}{1 + \sqrt{1 - \frac{4ac}{b^2}}}$$

and

$$x_2 = -\frac{c}{ax_1}$$

With these formulae we obtain as the smallest root x_1 and as the (in absolute value) largest root x_2.

In serious computations the pitfalls may be much larger than illustrated by this example. In such cases you should go for advice to the branch of mathematics that knows how to cope with limited accuracy, viz. *numerical mathematics*.

4.5 Example: Mean and variance

An important problem in production processes is the control of the quality of the product. Assume that we have a factory producing bottles of noodle soup. The number of noodles in one bottle of noodle soup should be on the average equal to fifty. It should not be too low, otherwise we will no longer be entitled to call our product noodle soup. It should not be too high, because noodles are relatively expensive in comparison to the other ingredients of the soup.

One way to check the quality of our noodle soup is to count the number of noodles in every bottle produced and adjust the production process as soon as we see important deviations.

Important deviations occur whenever the average number of noodles per bottle strays too far from our goal value (i.e. 50). But that is not enough. We would also get into trouble if one bottle of the soup were to contain one hundred noodles and the next bottle no noodles at all. If we alternate bottles like that, the average will be fifty but we still may have trouble selling our stuff. We should insist that the variance of the number of noodles per bottle is not too large.

Of course it is not feasible to count the number of noodles in each and every bottle. In practice we will therefore count only the noodles from a relatively small fraction of the bottles. Any handbook of elementary statistics will tell us how often we have to measure a batch of bottles in order to obtain (according to the formulae given below) a reasonable indication of the mean and the variance of the number of noodles per bottle.

Let W_i be the number of noodles in the ith bottle of the batch and let n be the total number of bottles in the batch. The average is given by the formula:

$$\frac{\sum_{i=1}^{n} W_i}{n}$$

and the variance (exactly, *corrected empirical variance* is its name in mathematical statistics) by:

$$\sqrt{\frac{\sum_{i=1}^{n} W_i^2 - n * average^2}{n - 1}}$$

Notice that for $n = 1$ the corrected empirical variance is undefined.

The mean and corrected empirical variance can now be computed according to these formulae by a program, which has to read the size of the batch and the values for W_i and compute the sum and the sum of squares of W_i.

$(x - 10.0^i) * (x - 1.0) = 0.0$ for $i = 1, 2, 3, \ldots$			
with abc-formula		**with improved formula**	
x_1	x_2	x_1	x_2
+1.0000000E +0	+1.0000000E +1	+1.0000000E +0	+1.0000000E +1
+1.0000000E +0	+1.0000000E +2	+1.0000000E +0	+1.0000000E +2
+1.0000000E +0	+1.0000000E +3	+1.0000000E +0	+1.0000000E +3
+1.0000000E +0	+1.0000000E +4	+1.0000000E +0	+1.0000000E +4
+1.0000000E +0	+1.0000000E +5	+1.0000000E +0	+1.0000000E +5
+1.0000000E +0	+1.0000000E +6	+1.0000000E +0	+1.0000000E +6
+1.0000000E +0	+1.0000000E +7	+1.0000000E +0	+1.0000000E +7
+0.0000000E +0	+1.0000000E +8	+1.0000000E +0	+1.0000000E +8
+0.0000000E +0	+1.0000000E +9	+1.0000000E +0	+1.0000000E +9
+0.0000000E +0	+1.0000000E+10	+1.0000000E +0	+1.0000000E+10
+0.0000000E +0	+1.0000000E+11	+1.0000000E +0	+1.0000000E+11
+0.0000000E +0	+1.0000000E+12	+1.0000000E +0	+1.0000000E+12
+0.0000000E +0	+1.0000000E+13	+1.0000000E +0	+1.0000000E+13
+0.0000000E +0	+1.0000000E+14	+1.0000000E +0	+1.0000000E+14
+0.0000000E +0	+1.0000000E+15	+1.0000000E +0	+1.0000000E+15
+0.0000000E +0	+1.0000000E+16	+1.0000000E +0	+1.0000000E+16
+0.0000000E +0	+1.0000000E+17	+1.0000000E +0	+1.0000000E+17
+0.0000000E +0	+1.0000000E+18	+1.0000000E +0	+1.0000000E+18
+0.0000000E +0	+1.0000000E+19	+1.0000000E +0	+1.0000000E+19

Table 4.6: Computation of the roots of a quadratic equation

Upon closer consideration it is not necessary to keep all values W_i: we only want to compute the sum of the values W_i and the sum of the squares W_i^2. For that we have to take into account only one W_i at a time.

During these summations, overflow can occur. This would happen with virtual certainty if we were to compute both sums as integers. We shall therefore compute the sums as reals, trusting that the real range of our computer is large enough.

We come to the following program:

```
   read number of measurements;
   compute sum and squaresum;
   compute mean and variance;
   print values.


read number of measurements:
   put ("Batch size = ");
   INT VAR n;
   get(n);
   line(2).


compute sum and squaresum:
   INT VAR number:: 0;
   REAL VAR sum:: 0.0, squaresum:: 0.0;
   WHILE not last value
   REP
     read next value;
     adjust sum and squaresum
   ENDREP.


not last value:
   number < n.

read next value:
   REAL VAR value;
   put("Next value, please: ");
   get(value);
   line;
   number INCR 1.

adjust sum and squaresum:
   sum:= sum + value;
   squaresum:= squaresum + value ** 2.

compute mean and variance:
   REAL CONST mean:: sum / real(n);
   REAL CONST variance::
     sqrt((squaresum - real(n) * average
**2) / real(n-1)).
```

Notice that we have to convert `n` explicitly into a real in order to use the real division and multiplication. The refinement can be simplified somewhat to

```
compute mean and variance:
   REAL CONST mean:: sum / real(n);
   REAL CONST variance::
     sqrt((squaresum - sum * average) /
real(n-1)).

print values:
   line;
   put ("Number of measurements= ");
put(n); line;
   put ("Average= "); put(mean); line;
   put ("Variance= "); put(variance);
line.
```

We install this little program in the computer on the work floor of the factory, give the necessary instructions to the personnel and, after some time, conclude with pride that the quality control is now so much better that the clients find on the average fewer noodles in their soup than before.

But one day the foreman, his face ash-grey, runs into our office with the cry: "It doesn't work ...". We follow him to the computer and read upon the screen the message

```
    FATAL ERROR AT ADDRESS 000247 IN MODULE
EC00D3E4
    OVERFLOW IN REAL DIVISION
    SYMBOLIC DUMP FOLLOWS
```

followed by a load of drivel. After some searching we find that the operator by mistake had indicated a batch size one. The program could not cope with this input. After some thought we modify one refinement

```
read number of measurements:
   INT VAR n;
   REP put ("Batch size = ");
       get(n);
       line(2)
   UNTIL n > 1
   ENDREP.
```

thus making the program somewhat more robust.

4.6 NTO of the reals

The NTO for the type REAL has to be composed of a number of parts:

- the monadic +, -;
- the dyadic +, -, *, /, MOD and **;
- the comparison operators =, <>, <, <=, > and >=;
- the output procedure put;
- the input procedure get;
- the real constants maxreal, smallreal and pi;
- the real functions sqrt, abs, sin, cos, tan, arcsin, arccos, arctan, ln and exp;
- the conversion functions real, round and trunc.

All these parts can be found back in the scheme of Fig. 4.4. This NTO in its turn forms part of a larger framework together with other NTO's which we have not included in this picture.

For a complete set of operations on reals see appendix B.

4.7 Exercises

1. (Square root) One of the eldest algorithms is the following determination of the square root of a number. If x is an approximation of the square root of a then a better approximation can be obtained by the formula $x_{i+1} := 0.5(x_i + a/x_i)$.

   Write a program to compute the square root of some number. Compare the results with those gained by the built-in `sqrt(x)` function of Elan.

2. Explore the behaviour of real arithmetics by performing the following computations
   - $1 - 1/3 - 1/3 - 1/3$,
   - $1 - 1/6 - 1/6 - 1/6 - 1/6 - 1/6 - 1/6$,
   - $x - \sqrt{x^2}$ for some values of x,
   - $x - (\sqrt{x})^2$,
   - $x - \tan(\arctan x)$.

3. Compute e and $1/e$ by means of the sequences

   $$e = 1 + 1/1! + 1/2! + 1/3! + \ldots,$$

   $$1/e = 1 - 1/1! + 1/2! - 1/3! + \ldots$$

   After every step, the product of the approximations should be computed as a check.

4. (Minimization) Write an algorithm that finds a minimum of a concave function in one variable by repeated halving of the interval in which this minimum should lie. Find, with the aid of this algorithm, that value in $[3.0 : 4.0]$ for which $\cos x$ is minimal.

5. Compute $\pi/4$ by means of the following sequence

   $$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - \ldots$$

   After every step, show the value of the approximation and compare it to the value the computer gives for $\pi/4$.

6. Compute $\ln 2$ by means of the sequence

   $$\ln 2 = 1 - 1/2 + 1/3 - 1/4 + 1/5 - \ldots$$

   Compute the power of e with your approximations as a check.

7. Determine experimentally the values of maxreal and smallreal on your computer.

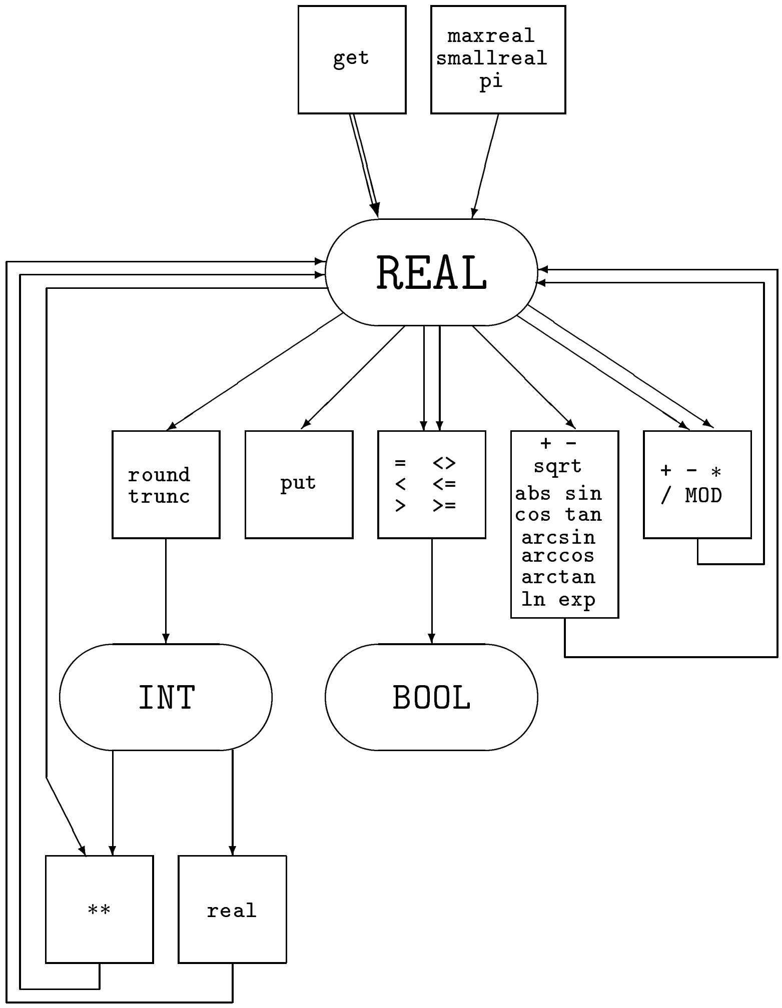


Figure 4.4: NTO of reals

Chapter 5

Truth values

In Elan two truth values are distinguished: *true* and *false*. In honour of George Boole (1815–1864) who invented the logical algebra or "switching algebra", these values are called Boolean values; in computer jargon: *booleans*. Their type has the name `BOOL`.

Truth values can be regarded in two ways. In the first place they appear as values of objects that can be manipulated by algorithms. In this respect they behave similarly to integers and reals. On the other hand, truth values can also control the execution of algorithms. In the choice and repetition, the execution of the algorithm is controlled by the result of a condition.

In Elan, no distinction is made between those two aspects of truth values, but experience shows it is difficult for the beginner to reconcile those two different uses.

5.1 Boolean denotations

Since there are only two truth values, the syntax of the boolean-denotation consists of an enumeration of the possibilities.

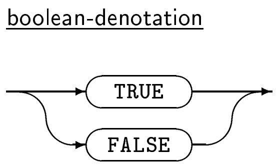


Figure 5.1: Boolean denotation

The yes-value is represented as `TRUE` and the no-value is represented as `FALSE`. In the standard packets two boolean constants, `true` and `false`, are predefined for the use of those people who (like this author) do not like conspicuous uppercase letters. Therefore we can write `TRUE` and `FALSE` in small letters instead if we prefer.

Also for this type we have to distinguish between a number of concepts:

truth value	the concepts *true* and *false*
boolean	the representations of those truth values in the programming language
`BOOL`	the name of the type of the booleans
boolean-denotation	the notation for boolean values.

5.2 Elementary algorithms for booleans

Instead by way of a boolean denotation, a truth value can be introduced by means of an algorithm yielding a boolean value (a condition or "test"). This is the usual way to obtain such a value.

5.2.1 Comparison operators

In Elan for all elementary types, including the booleans, there exist two comparison operators that compare the values of their operands for equality. These operators are shown in Table 5.1. The two

operator	meaning
=	equal to
<>	not equal to

Table 5.1: Comparison operators for equality

operands must be of the same type, be it `INT` or `REAL` or `BOOL` (or, as we will see in chapter 6, `TEXT`). The result of the comparison is a truth value. For operands of type `BOOL` these are also the only comparison operators. In addition, for those elementary types for which an ordering relationship exists, such as `INT` and `REAL`, but also `TEXT` (see chapter 6), there exist four more comparison operators. They are shown in Table 5.2.

operator	meaning
<	smaller than
>	greater than
>=	at least
<=	at most

Table 5.2: Comparison operators for ordering

These also take two operands of one same type and yield a truth value. Just like the arithmetic operators, these comparison operators are not defined for combinations of integer and real operands; the types of their operands have to be the same.

The priority of comparison operators is lower than the priority of the arithmetic operators, so that a formula like:

```
b ** 2 < 4 * a * c
```

means the comparison of `b ** 2` with `4 * a * c`; no brackets are needed here.

The two different aspects of truth values — elementary object as well as control of the execution — is reflected in the use of formulae such as the one above. This formula might for instance be used in a choice like:

```
IF
  b ** 2 < 4 * a * c
THEN
  compute complex roots
ELSE
  compute real roots
FI
```

Here the result of the test is used immediately to choose between the computation of the complex or the real roots of a quadratic equation. The result of a test is, however, a truth value. Therefore the test can also be used in an assignment or an initialization, e.g.

```
BOOL CONST
   discriminant is negative :: b ** 2 < 4
* a * c
```

The result of the test is now kept as the value of a constant and can be used later in a choice like:

```
IF
  discriminant is negative
THEN
  compute complex roots
ELSE
  compute real roots
FI
```

Of course this constant declaration makes the most sense if its value is used more than once. Observe, by the way, that the value of this constant of course does not change if we happen to change the values of `a`, `b` and `c` afterwards. Its value will not be recomputed, in distinction to, for example, a refinement

```
   discriminant is negative: b ** 2 < 4 *
a * c.
```

whose value is recomputed afresh each time the refinement is invoked.

5.2.2 Logical operators

For the manipulation of truth values as elementary objects, the following operators are given: `NOT` (negation), `AND` (conjunction), `OR` (disjunction, inclusive OR), and `XOR` (exclusive OR).

The result of these operators for various values of their operands follows from Table 5.3. The operator `AND` has a higher priority than the operators `OR` and `XOR`, but a lower priority than the comparison operators. The monadic operator `NOT` has, like all other monadic operators, the highest priority.

With the help of these operators and the boolean comparison operators `=` and `<>`, the sixteen operators

operands		operations			
p	q	NOT p	p AND q	p OR q	p XOR q
TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
TRUE	FALSE		FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE		FALSE	FALSE	FALSE

Table 5.3: Boolean operators

of the proposition calculus can be realized simply. As an example, both

```
p = q OR q
```

and

```
NOT p OR q
```

are formulae computing the *implication* $p \rightarrow q$.

5.2.3 Combined conditions

Some care has to be exercised in the use of comparison operators. The formula

```
p = q = r
```

does not mean what you hope. Because the priority of the operators is equal, this formula is executed from left to right, and therefore it is equivalent to

```
(p = q) = r
```

i.e. the boolean result of the comparison `p = q` is compared to the (supposedly logical) value of `r`. If we wish to ensure that we get the result `TRUE` exactly when `p`, `q` and `r` all have the same value, we must write

```
p = q  AND  q = r
```

In testing on an ordering, a formula of the form

```
a <= b <= c
```

is syntactically wrong, because the result of the leftmost operation is a value of the type `BOOL` which does not allow comparison with the integer `c`. Therefore we have to write

```
a <= b  AND  b <= c
```

if we wish to achieve our goal.

Note that both operands of a dyadic logical operator are computed even when, after the computation of the first operand, the result is already decided. As an example,

```
a <> 0 AND b DIV a >= 10
```

is not a suitable way to prevent a division by zero. Instead one has to write

```
IF
  a <> 0
THEN
  b DIV a >= 10
ELSE
  false
FI
```

For the convenient handling of such cases, some programming languages and systems, e.g. EUMEL, offer the so called conditional AND and conditional OR operators (shortly CAND and COR). Here, the second operand will not be computed if the result may already be determined from the first one. Then you can write

```
a <> 0 CAND b DIV a >= 10
```

5.2.4 Input and output of booleans

For the input and output of truth values, Elan does not provide algorithms. Instead of these truth values, one will have to read and write a *text,* as is shown in chapter 6.

5.3 Trusting the booleans

The twofold character of booleans (value and control) leads many people to strange circumlocutions and errors of style that can all be explained as a lack of trust in the booleans. Assume for example that `a` is a boolean variable. Many programmers write

```
IF a = true THEN ...
```

where simply

```
IF a THEN ...
```

would suffice. Obviously they do not realize that the expression `a` by itself already yields a boolean and try to turn it into a control by means of a test. It must be conceded that the last line looks a bit naked, but not if you use a meaningful identifier like

```
IF it is raining THEN ...
```

A comparable distrust of the integers would lead to the absurd

```
put (i + 0)
```

in order to ensure that the value of the integer variable `i` is really a whole number. Strange contortions like

```
BOOL VAR t;
t := IF a > b THEN true ELSE false FI;
IF (t = true) = true THEN t := false FI
```

show a lack of understanding of the booleans and a lack of trust in their dual character.

It is instructive to invent shorter versions for the following boolean expressions:

```
IF a > 0 THEN true ELSE false FI

IF a THEN NOT a ELSE true FI

IF a THEN false ELSE a FI
```

5.4 Example: Prime numbers (1)

Consider the problem of determining whether a given positive whole number is a prime number, i.e. whether it has no divisors except 1 and itself. We start with a rough formulation

```
read the number;
determine whether the number is prime;
print the answer.
```

Reading the whole number gives no problems.

```
read the number:
   INT VAR number;
   put("Number, please: ");
   get(number).
```

The easiest way to find out whether a number is prime is to look it up in a table of primes, but in order to keep the problem interesting we shall assume that such a table is not available.

We can try to divide the number by all prime numbers that are smaller than that number and check every time whether the rest is zero, but for that we again need a table! On the other hand, we might just as well simply divide by all numbers that are smaller than the number, including the non-prime ones. This costs more work but leads to the same result. After all, that's what we have computers for.

The answer to the primality question we shall record in a boolean variable `no divisors`, that initially is `TRUE` and is made `FALSE` as soon as we find a divisor. The first `candidate` we try is 2.

```
determine whether the number is prime:
   BOOL VAR no divisors :: TRUE;
   INT VAR candidate :: 2;
   WHILE yet candidates to try
   REP
     look if candidate fits;
     take the next candidate
   ENDREP.

look if candidate fits:
   IF number divisible by candidate
   THEN no divisors := FALSE
   FI.
```

How does one decide whether a number is divisible by another number? One way to do it is the following. We first divide the number by the candidate and ignore the remainder. Then we multiply the quotient obtained by the candidate and compare the result with the original number. The result is only equal to the original number if the first division had zero as remainder.

```
number divisible by candidate:
   (number DIV candidate) * candidate =
number.
```

Strictly speaking, the brackets are superfluous here, but we leave them in to make this refinement more transparent.

Somewhat simpler is the use of the operator `MOD` that, for positive operands, yields the remainder of the division.

```
number divisible by candidate:
   number MOD candidate = 0.
```

As candidates we can try all numbers smaller than the number itself

```
yet candidates to try:
   candidate < number.

take the next candidate:
   candidate INCR 1.
```

For the last refinement, `print the result`, we steal a leaf from chapter 6 and print one of two texts.

```
print the answer:
   IF no divisors
   THEN put ("prime number")
   ELSE put ("not a prime number")
   FI.
```

The program is now complete, but it relies very much on the brute force of computers. Let us try to make it somewhat cleverer.

A first criticism of the program is that we try far too many divisors. It is sufficient to try only those candidates that are smaller than the square root of `n`. For if `k` is a divisor larger than that square root, then also `n DIV k` is a divisor, smaller than that square root! For large values of `n` this makes a tremendous difference, e.g. having to try 100 divisors rather than 10000.

```
yet candidates to try:
   candidate * candidate <= number.
```

A second improvement follows from the observation that, once a divisor has been found, there is no reason to continue the repetition. In the current formulation the process inexorably continues until all candidates have been tried. We can improve upon this by insisting in the condition of the repetition that `no divisors` is true.

```
yet candidates to try:
   candidate * candidate <= number AND no
divisors.
```

We can now also simplify the refinement `look if candidate fits` somewhat:

```
look if candidate fits:
   no divisors := division does not fit.

division does not fit:
   NOT(number MOD candidate = 0).
```

or as

```
division does not fit:
   number MOD candidate <> 0.
```

Notice that the brackets in the first version of `division does not fit` are necessary because of the high priority of the operator `NOT`. We prefer the second version.

Because of the role played by the variable `no divisors` in assuring fast termination of this repetition, we call this form a *repetition with a boolean auxiliary variable.*

The repetition can end in one of two ways: either the number turns out to be prime, after all candidates have been tried, or one of the divisions fits and the variable `no divisors` obtains the value `FALSE`.

5.5 The LEAVE-construct

The use of such a boolean auxiliary variable is a trick to obtain two things in one stroke. The problem is that at a place deep in the algorithm we have a certain knowledge (viz. that a divisor has been found) with which we do not know what to do at that place (in the very interior of `look if candidate fits`). Another refinement `determine whether the number is prime` can be completed due to use of this knowledge.

For this purpose, Elan knows a specific construct, the `LEAVE`-construct (called terminator in the syntax; see Fig. 5.2).

terminator

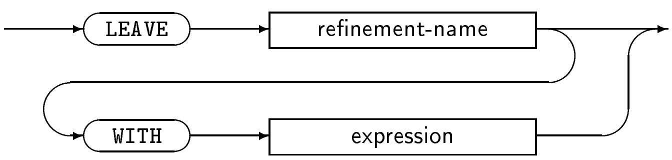


Figure 5.2: Terminator

This construct causes the present execution to be disrupted; and instead of it, the refinement mentioned is completed from within in one fell swoop. Obviously, only such an algorithm can be named in a `LEAVE`-construct whose execution led to it. The `WITH`-part, which is optional, serves to complete an algorithm which yields a result. It will be discussed in more detail later on.

Using this construct, we can write

```
look if candidate fits:
   IF division fits
   THEN no divisors := FALSE;
     LEAVE determine whether the number is
prime
   FI.

division fits:
   number MOD candidate = 0.
```

Now we can simplify another refinement

```
yet candidates to try:
  candidate * candidate <= number.
```

The resulting program is somewhat more efficient but especially more perspicacious. In languages without the LEAVE-construct we have to make use of boolean auxiliary variables instead.

5.6 Example: Prime numbers (2)

The program that we have obtained now is already an enormous improvement over the previous version. Still a number of superfluous divisions are made. Let us try another improvement.

After having tried the divisor 2 we do not have to try any other even number and can restrict ourselves to the odd numbers as candidates. We can achieve this by, after the test whether the number is not even, starting with the candidate 3 and computing the next candidate by increasing the divisor 2.

However, after the candidate 3 we need not try any other multiples of 3 so that, having found that the number is not divisible by 7, we can continue with the candidate 11. We will now compute the next candidate by increasing the candidate alternately by 2 and 4. Why? Here is the explanation.

The number p is either divisible by 3 or $p \bmod 3 = 1$ or $p \bmod 3 = 2$. If $p \bmod 3 = 1$ then $(p+2) \bmod 3 = 0$ and $(p+4) \bmod 3 = 2$, i.e. p should be increased by 4 if we want to skip the numbers divisible by 3. Similarly, if $p \bmod 3 = 2$ then $(p+4) \bmod 3 = 0$ and $(p+2) \bmod 3 = 1$, i.e. now p should be incremented by 2 in order to skip the unwanted numbers. Hence treating 2 and 3 as special cases, we then try the divisors 5, 7, 11, 13, 17, 19, 23, ...

In this way we have to try at most $\lfloor\sqrt{n}/3\rfloor + 2$ candidates, including 2 and 3, rather than $n-2$ and $\lfloor\sqrt{n}\rfloor - 1$ candidates in the first and second versions, respectively. Even for fairly large values of n, for instance $n < 10^8$, this algorithm can still be used.

The following program works according to this idea. The alternating addition of 2 and 4 to the value of `candidate` is achieved by increasing its value with the value of the variable `increment`, which thereupon obtains the value `6 - increment`. The variable `increment` in this way assumes the values 2, 4, 2, ...

Of course we could go still farther and omit also the multiples of larger prime numbers as candidates. The values that `increment` then has to take become rather difficult to compute and the returns are diminishing, partly because of this additional computing. We therefore leave such an amelioration to the reader.

```
determine whether the number is prime:
   BOOL VAR no divisors :: FALSE;
   IF number MOD 2 = 0 OR number MOD 3 = 0
   THEN
     it was not prime
   ELSE
     INT VAR candidate :: 5, increment ::
2;
     WHILE yet candidates to try
     REP
       look if candidate fits;
       take the next candidate
     ENDREP
   FI;
   it was prime.
```

All refinements are as in the previous example, except:

```
take the next candidate:
  candidate INCR increment;
  increment := 6 - increment.

it was not prime:
  LEAVE determine whether the number is
prime.

it was prime:
  no divisors: = TRUE.
```

For yet larger values we have to use quite different methods. These are often based on the theorem of Fermat. We shall not discuss them as they require a firm knowledge of higher mathematics. But we can, at least, draw the lesson that when the amount of data to be processed increases significantly we have to apply more efficient algorithms based on higher mathematics. It may even be better if we turn to an expert.

5.7 Comments

We will discuss the construct called a *comment* at this place, although properly speaking it does not belong here. Comments are pieces of text which are not directed at the computer but at the human reading the program. They are not a functional part of the program and therefore do not fit into the systematic scheme of the programming language. For that reason we might just as well discuss them here as anywhere else.

In Elan, comments can appear anywhere between symbols, denotations and identifiers, and have no effect at all on the meaning or execution of the program. A comment starts with a comment-open-symbol and ends with a comment-close-symbol. Between those symbols, all characters may appear that can not be confused with a comment-close-symbol. There are two representations for these symbols:

comment-open-symbol	{	(*
comment-close-symbol	}	*)

Some examples of comments:

```
  version 7.3 of 23 October 1976, H.F.
(* now the fun starts! *)
```

Comments serve such purposes as:

- Indicating name, version and author of a program.
- Giving a short characterization, limitations and preconditions for the use of a program or some part of a program, e.g.:

```
{ Solution of the equation system AX = B
and computation of the determinant as a
check
on the precision, according to the method
of
Gauss-Jordan.
A, B and X must have the same size.
The algorithm should not be used for large
systems because of instabilities. }
```

- Asserting invariant properties for the benefit of the human reader and for a proof of correctness, e.g.:

```
find maximum of a sequence of positive
integers:
   INT VAR max :: 0;
  { maximum of an empty sequence }
   FOR i FROM 1 UPTO n
   REP
     IF max < row[i]
     THEN max := row[i] FI
     { max is the maximum of row[j] for j
= 1 .. i }
   ENDREP
   { max is the maximum of row[j] for j =
1 .. n } .
```

- Giving only absolutely necessary explanations, e.g.:

```
(* In the interest of efficiency we have
   omitted the test on overflow. *)
```

Comments are not intended to conserve stupid remarks for posterity as in the second example `(* now the fun starts! *)` or as in

```
x := 0;     x is set to zero
```

It is much more preferable to include abstractions functionally in the program, by the use of refinements for the algorithms and by the choice of meaningful names for the objects, rather than to add to the program (mostly in hindsight) comments that try to make it understandable.

A refinement might, of course, have a misleading name, but that can usually be noticed by inspection of a small part of the program. A comment on the other hand is not functionally part of the program so nobody cares whether the comments remain up to date in any modifications to the program.

In this book we lay so much stress on abstraction that the use of comments turns out to be largely superfluous. In a more industrial environment, the careful and formalized use of comments is absolutely necessary, due to restrictions of the programming languages used that do not allow the retention of the abstractions that occurred in the programming process. Due to the lack of refinements, the program by itself does not give enough documentation.

We end this section with a pearl of wisdom:

> *A badly structured program cannot be saved*
> *by the addition of any number of comments.*
> *Its chaotic origin will always remain obvious.*

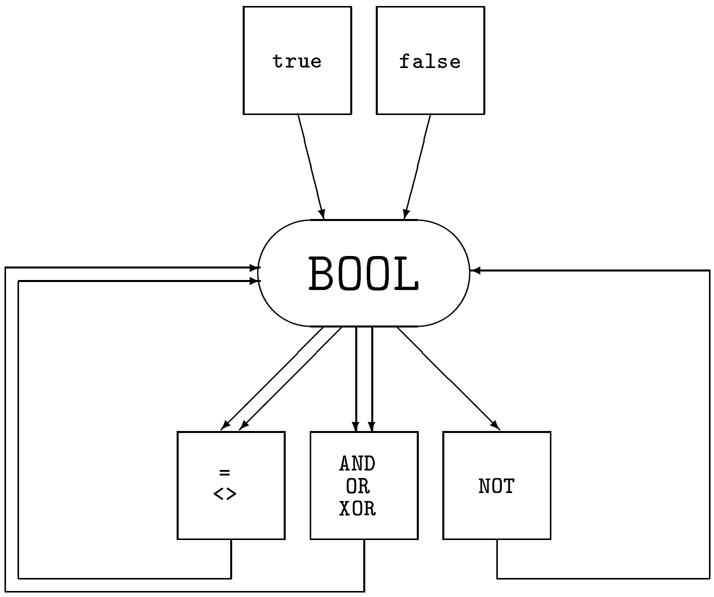


Figure 5.3: NTO of booleans

5.8 NTO of the booleans

The NTO for the type `BOOL` has to be composed of a number of parts:

- the monadic `NOT`;
- the dyadic `AND`, `OR` and `XOR`;
- the comparison operators `=` and `<>`;
- the boolean constants `true` and `false`.

All these parts can be found back in the scheme of Fig. 5.3. This NTO in its turn forms part of a larger framework together with other NTO's which we have not included in this picture.

Chapter 6

Texts

The algorithmic manipulation of *texts* is the key to a whole world of non-numeric applications:

- the presentation of the results of computations in an attractive form,
- translators and interpreters for programming languages,
- linguistic research and other research with a linguistic component,
- the processing of texts in newspapers and in the office,
- various forms of office administration.

The applications mentioned last may be the most prosaic, but economically they are the most important. With the advent of machines speaking and understanding human speech, the scope for non-numerical applications will certainly grow.

For the manipulation of texts, special programming languages have been designed (such as SNOBOL and all kinds of macro processors) that allow a concise formulation of complicated text manipulations. The standard library of Elan offers a whole collection of cutting and pasting instruments for dealing with texts. The language mechanisms of the subset Elan-0 are in this respect more primitive but still adequate.

6.1 Denotation of texts

Texts are composed of *characters,* chosen from a specific alphabet. The type of such a text is `TEXT`. A text can be thought of as a row of characters, each of them representable in the computer (Fig.6.1). What characters are representable in the computer depends on the implementation used, but this alphabet in any case comprises the signs with which Elan programs are written (lower case and capital letters, digits, punctuation marks, operator tokens, spaces, etc.).

text-denotation

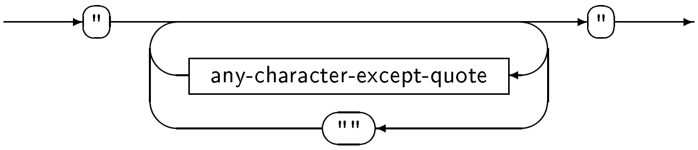


Figure 6.1: text denotation

A text is denoted by putting it between *quotes,* e.g.:

```
"this is a text"
```

The value of the denotation is the sequence of characters obtained by omitting the enclosing quotes.

In order to represent a quote within a text, the convention is used to double such a quote sign (a typical trick from informatics, used over and over again). Example:

```
" ""Silence!"" spoke Gandalf, ""Hear
Thorin's speech""."
```

In particular the text consisting of a single quote is denoted as follows:

```
""""
```

The empty text (a row of zero characters) is denoted as

```
""
```

To avoid confusion, in this chapter we will indicate the space within a text denotation by the sign `#` in order to make it easier to count how many spaces are meant in, for example:

```
"###"
```

This character does not appear on your computer, where the space will be represented by an empty position.

Although, in principle, texts of any length can be denoted, a particular implementation may impose an upper limit on the length of representable texts. As an example, the Elan-compiler in the EUMEL system limits the length of texts to 255 characters (the value of the `INT`-constant `max text length`). Larger texts have to be treated in that system as rows of lines (see chapter 8 on rows).

6.2 Operations on texts

For the manipulation of texts, Elan offers a number of standard operations. Unfortunately only a small part of those is also available within Elan-0. This is one of the places where the language had to be severely reduced. In their stead, Elan-0 has a simpler set of operations. We shall first describe the operations that Elan-0 has in common with Elan and afterwards those which are particular to Elan-0. The remaining Elan operations are described in section 6.6.

6.2.1 Common text operations

1. The operator `+` concatenates two texts, e.g.:

   ```
   "abc" + "def" = "abcdef"
   ```

2. The operator `LENGTH` yields the number of characters in the text, an integer greater than or equal to zero. As an example

   ```
   LENGTH "abc"
   ```

   is equal to 3, and

   ```
   LENGTH """"
   ```

   yields 1.

3. The operator `*` with a non-negative integer `n` as its left operand and a text as its right operand yields the `n`-fold concatenation:

   ```
   3 * "abc" = "abcabcabc"
   ```

4. The characters of a text `x` can be considered to be numbered starting from 1. In order to select the `k`th character from the text we have an operator `SUB` (its left operand a text `x`, its right operand an integer `k`, yields a text composed out of one character, viz. the `k`th character of `x`). Example:

   ```
   ("abc" SUB 2) = "b"
   ```

   If the right operand of `SUB` is smaller than one or larger than the number of characters in the left operand then the result is an empty text, for example in

   ```
   "abc" SUB 37
   ```

5. The dyadic operator `CAT` (with a text variable as its left operand and a text as its right operand) combines a concatenation with an assignment, e.g.:

   ```
   t CAT "abc"
   ```

   means the same as

   ```
   t := t + "abc"
   ```

   Remark: The notation `CAT`, from concatenation, is somewhat inconsistent with the generally accepted terminology. In text processing, the term *concatenation* usually means the previously introduced `+` operation of Elan.

The operator `SUB` has a blemish: it has an extremely low priority, so that expressions involving `SUB` in many contexts have to be enclosed between brackets. If we omit the brackets we may discover to our surprise that

```
"abc" SUB 2 = "b"
```

means:

```
"abc" SUB (2 = "b")
```

which is rejected by the Elan implementation with a rather bizarre error message. Namely, an integer can not be compared to a text, and a boolean value can not be the operand of `SUB`.

It is rather bewildering for that the seemingly similar expression `LENGTH s - 1` is correct and has the same meaning as `(LENGTH s) - 1`. The difference is caused by the fact that `SUB` is a dyadic operator with low precedence while `LENGTH` is a monadic operator and therefore has the highest possible priority.

For this reason it is advisable to include in expressions involving the operator `SUB` a sufficient number of brackets to avoid unintended interpretations, e.g.

```
3* ("abc" SUB i)
```

6.2.2 Comparison of texts

The comparison operators for texts are the same as those for integers and reals:

```
=    <=    <    >    >=    <>
```

The ordering relation meant is the alphabetic lexicographic ordering. This implies that

```
"" < "a"
"a" < "b"
"a" < "ab"
"aa" < "b"
"aa" < "ab"
```

and so on. You may know that this ordering has the property that, if some text `x` is the same as an initial segment of a text `y` (i.e. `y` consists of `x` possibly followed by some more characters), we then know for sure that `y >= x`.

Each character has a *code*, a small positive whole number (mostly < 128 or < 256) that serves as internal representation. The individual characters of a text are compared on the basis of their codes. Since different implementations may each use their own code, apart from the ordering of letters and the ordering of digits not much is certain (does the space come before or after the small letter "a"? Do the small letters come before or after the capital letters? Does the dollar sign ($) come before or after the "at-sign" (@)? To such questions no general answer can be given: it depends on the code used).

The two codes that are most widely used are

EBCDIC (Extended Binary Coded Decimal Interchange Code) on IBM main frames,

ASCII (American Standard Code for Information Interchange) on nearly all others

but the situation is further complicated by the fact that ASCII is actually the American variant of ISO, which has a number of other national variants containing such signs as ä and ç. If you do not know in what code you are working you should avoid making difficult comparisons.

6.2.3 Subtexts in Elan-0

The sublanguage Elan-0 has a number of algorithms for working with parts of a text (the complete language Elan has a whole collection as described a few sections down, in section 6.6).

The monadic operator `HEAD` yields a text consisting of the first character of its operand, for example:

```
HEAD "abc" = "a"
```

and the `HEAD` of an empty text is the empty text. The monadic operator `TAIL` yields a copy of the value of its operand but with its first character removed, for example,

```
TAIL "abc" = "bc"
```

The `TAIL` of an empty text is again empty, just like the `TAIL` of a text of length 1. In all cases

```
HEAD s + TAIL s = s
```

and

```
HEAD s = (s SUB 1)
```

6.2.4 Input and output of texts

A text can be written onto the screen with the output algorithm `put`. Where it crosses the end of the line, it is continued on the next line. In this way the layout can never become a total mess, but it is still advisable to keep the layout in hand by means of the procedure `line`. Example:

```
line; put ("Sum#=#"); put (sum); line
```

The algorithm `get`, called with a `TEXT`-variable as parameter, reads the input until the end of the line (marked by RETURN, ENTER, CR, etc.). An empty line is skipped and the next non-empty line is taken. The resulting text is assigned to its parameter. Example:

```
TEXT VAR message; get (message)
```

After this, the value of `message` will be a non-empty text.

6.2.5 First summary

The elan-0 subset comprizes the following text operations:

- the monadic `HEAD`, `TAIL` and `LENGTH`;
- the dyadic `+`, `CAT`, `SUB` and `*`;
- the comparison operators `=`, `<>`, `<`, `<=`, `>` and `>=`;
- the output procedure `put`;
- the input procedure `get`;
- the conversion procedure `ascii`.

The NTO in figure 6.2 gives an overview.

6.3 Example: Making crossed paper

We want to make a grid of squares according to the pattern given in Fig. 6.3.

```
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
```

Figure 6.3: Crossed paper

We program:

```
draw grid:
   INT VAR row number :: 1;
   REP
     draw row of squarecaps;
     row number INCR 1
   UNTIL row number = 5
   ENDREP;
   draw horizontal stripe.

draw row of squarecaps:
   draw horizontal stripe;
   draw vertical lines;
   draw vertical lines.

draw horizontal stripe;
   put(4 * (cross + 5 * horizontal) +
cross);
   line.

draw vertical lines:
   put(4 * (vertical + 5 * blank) +
vertical);
   line.
```

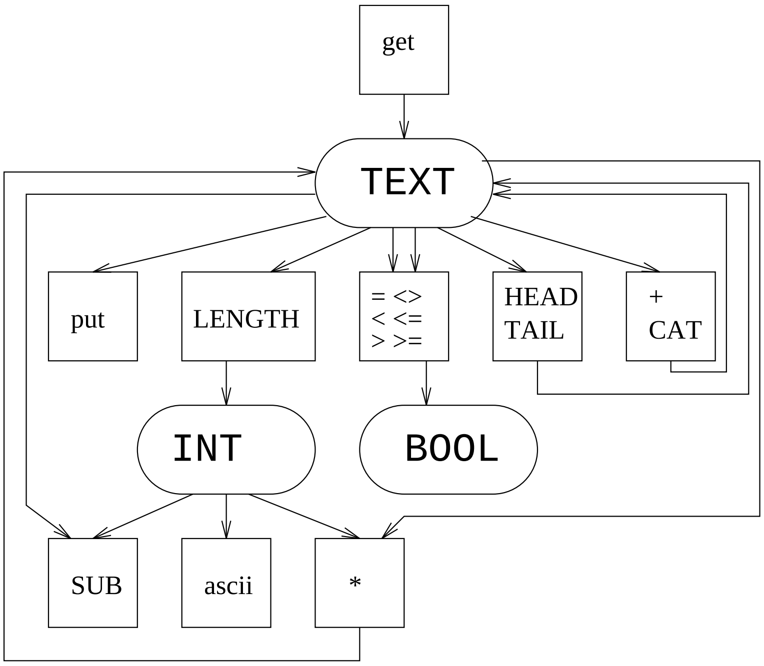


Figure 6.2: Text operations in Elan-0

```
horizontal: "-".

vertical:   "|".

cross:      "+".

blank:      "#".
```

Notice that, instead of the refinements `horizontal`, `vertical`, `cross` and `blank`, we might just as well have declared text constants. Refinements yielding a value are often an alternative to the declaration of a constant.

Notice also that the priority of `+` and `*` for texts is the same as that for integers or reals. You may remember that the priority is a property of the operator and independent of the type of the operands.

6.4 Example: Converting integers to a given radix

We construct a program that reads a sequence of whole numbers. The first is taken as the radix of some number system, to which the other numbers have to be converted.

We are used to the decimal number system, with radix 10: a number is represented decimally as a weighted sum of powers of 10, using the digits of the number as weights. For example

$$\text{decimal } 4711 = 4*10^3 + 7*10^2 + 1*10^1 + 1*10^0$$

Each of those digits is a positive whole number smaller than the radix (so $0 \leq digit < radix$).

We can generalize this system to radices other than 10; for example we can take radix 2

$$\text{decimal } 23 = 1{*}2^4{+}0{*}2^3{+}1{*}2^2{+}1{*}2^1{+}1{*}2^0 = \text{binary } 10111$$

We can also take radices larger than 10 where A,B,... will be used to represent the curious "digits" ten, eleven, etc.

A rough formulation of such a program can be:

```
radix conversion:
   ask for a radix;
   WHILE another number to convert
   REP
     represent the number to this radix;
     print the representation
   ENDREP.

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

another number to convert:
   line;
   put ("A#number,#please:#");
   INT VAR number;
   get (number);
   number > 0.
```

We determine the representation of the number in this radix by, from back to front, splitting off digits in the radix number system.

```
represent the number to this radix:
   TEXT VAR representation :: "";
   REP
     split off the rightmost digit;
     append it to the representation
   UNTIL no digits left
   ENDREP.
```

```
split off the rightmost digit:
   INT CONST digit :: number MOD radix;
   number := number DIV radix.
```

We append a text, corresponding to the digit just split off, at the front of the representation.

```
append it to the representation:
   representation :=
            representation of this digit +
representation.
```

We shall introduce a representation for the digits up to fifteen; higher digits will be printed as a question-mark.

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

We have to add one to the value of the `digit`, because the numbering of characters in a text does not start at zero but at one.

```
no digits left:
   number = 0.

print the representation:
   line (1);
   put ("Representation#is#");
   put (representation);
   line (2).
```

This example is particularly useful because it deals with the relationship between texts (that may contain digits) and numbers (composed of digits). The reader is urged to try out this algorithm for some values of radix and number.

6.5 Example: Circular shifting of texts

The next example is intended to clarify the use of the operators HEAD and TAIL. By "circularly shifting the text to the left" we mean a cyclic movement as in

```
ABCDE
BCDEA
CDEAB
...
```

where each element is moved one place to the left, except for the first, which is moved to the far right.

We shall construct a program that shifts a string circularly to the left until the original is obtained again.

```
circular shift:
   declare string;
   REP
     shift string circularly one place to
the left;
     display string
   UNTIL same as original
   ENDREP;
   line.

declare string:
   TEXT VAR string;
   put("String,#please:#");
   get(string);
   TEXT CONST original :: string.

shift string circularly one place to the
left:
   string := TAIL string + HEAD string.

display string:
   line;
   put (string).

same as original:
   string = original.
```

This program shifts a string like AAAAAA only once to the left, and a string like ABABAB only twice since after one, or two, steps `string` will be the same as `original`. In this example we do not make a difference between similar characters at different places. However, the description of the problem is ambiguous and allows various interpretations: we may consider even the same characters different if they are at different places. In this case the algorithm `circular shift` must be modified:

```
circular shift:
   declare string;
   UPTO LENGTH string
   REP
     shift string circularly one place to
the left;
     display string
   ENDREP;
   line.
```

Now the `same as original` refinement is not necessary.

The example illuminates that the description of a problem, i.e. its *specification* is often ambiguous. Beginners as well as professional programmers frequently misunderstand the problems to be solved. We shall return to this question later, in chapter 11.

6.6 Subtexts in Elan

Besides +, *, SUB, CAT and LENGTH, the full Elan language knows a whole collection of special algorithms for cutting and pasting texts. Although none of these algorithms is absolutely necessary (with the operators + and SUB and some trouble any effect could already be achieved), they do make the manipulation of texts a lot easier than in Elan-0.

In many applications it is nice to have more powerful algorithms at your disposal, as described in the following sections.

6.6.1 Standard constants

In the standard environment, a number of TEXT-constants are predefined:

```
TEXT CONST niltext :: "",
           blank :: "#",
           quote :: """"
```

These can be used without further declaration.

Programmers often use the word `blank` as a synonym for *space* but even more frequently as a synonym for `niltext`. This latter seems to carry the day. By tradition Elan-0 uses the name `blank` as given above but, if you like, you may declare another constant with the name `space`.

6.6.2 Subtexts: the algorithm `text`

A text `t` can be turned into one of length `len` (by cutting it if `len < LENGTH t` and by padding it on the right with blanks if `len > LENGTH t`) by means of the call

```
text (t, len)
```

In all following examples we will assume that `t` has the value `abc`.

```
text (t, 2) = "ab"
text (t, 5) = "abc##"
```

We can also cut a text at a specific position `p` by the aid of a call of the form

```
text (t, len, p)
```

Its meaning is: a text of length `len` is formed from the text `t` but starting at the character with index `p`. Examples:

```
text (t, 2, 2) = "bc"
text (t, 5, 2) = "bc###"
```

It is easy to see that generally

```
text (t, 1, p) = (t SUB p)
text (t, len, 1) = text (t, len)
```

A more complicated example (cyclic shifting):

```
text (t, LENGTH t - 1, 2) + (t SUB 1)
="bca"
```

Strictly speaking there are two different algorithms defined in the standard environment, both with the name `text`, but with different number and type of parameters. (We shall later on meet more algorithms with this same name.) They all have in common that they turn "something" into a text, as the name tries to suggest. (The careful reader might notice that so far we have introduced three different `get` and three different `put` algorithms although we have not emphasized it. These `get` and `put` procedures differ in the *type* of their parameter.) Such a collection of algorithms with the same name, different number and/or types of parameters and a comparable function is called *generic.*

6.6.3 Subtexts: the algorithm `subtext`

The generic algorithm `subtext` is a practical pair of scissors. In its simplest form

```
subtext (t, from)
```

it yields the text obtained from `t` by starting at the position `from`. Example:

```
subtext (t, 2) = "bc"
```

In general

```
subtext (t, i) = text (t, LENGTH t - i +
1, i)
```

The call

```
subtext (t, from, to)
```

yields the subtext from `t` starting at the position `from` and ending at the position `to`, for example:

```
subtext (t, 1, 2) = "ab"
```

In case `from` > `to` the result is the empty text.

```
subtext (t, 2, 1) = ""
```

Generally

```
subtext (t, p, p) = (t SUB p)
```

As an example we will once more formulate the cyclic shift:

```
subtext (t, 2) + (t SUB 1) = "bca"
```

Since it has more than one definition with the same name, but different parametrization, the algorithm `subtext` is also generic.

6.6.4 Searching a text: the algorithm `pos`

It is possible to look in a text `t` for a text `x` by means of the call

```
pos (t, x)
```

This call yields the index of the leftmost occurrence of `x` in `t`, provided `x` occurs in it, and zero otherwise. Examples (assuming `x = "a"`):

```
pos (t, "b")  = 0
pos (t, "ab") = 1
pos (t, "ba") = 2
```

Its generic brother

```
pos (t, x, from)
```

yields the index of the first occurrence of `x` in `t` starting from the position `from` (and zero if `x` does not occur). Thus the second occurrence of `x` in `t` can be found by the call

```
pos (t, x, pos (t, x) + 1)
```

6.6.5 Replacing a subtext: the algorithm `change`

A subtext of a particular text can be replaced by another text by a call of

```
change (text variable, subtext, other
text)
```

in which the leftmost occurrence of `subtext` in the `text variable` is replaced by the `other text`. If the `subtext` does not occur in the value of `text variable` nothing happens. Since the length of `subtext` and `other text` may differ, it is possible that this call changes the length of the value of the `text variable`.

We can, as an example, change every occurrence of `John` in a text into `Jim` by means of

```
WHILE
  pos (my text, "John") > 0
REP
  change (my text, "John", "Jim")
ENDREP
```

Remark: This short algorithm shows nicely the effect of a call of `change` but it hides dangerous traps, too. E.g. if the loop body is modified to `change (my text, "John", "Johnson")` the repetition continues forever. Be cautious!

6.6.6 Reading and writing: the algorithms `get` and `put`

As in Elan-0, a text is written by means of the algorithm `put`.

For the reading of a text, in full Elan a number of algorithms are available. Let `x` be a `TEXT`-variable. The simplest one

```
get(x)
```

has the following effect:

1. spaces and new lines are skipped in the input until a non-space is found;
2. from the current position until the next space or end of line, characters are read and combined into a text;
3. this text is assigned to the variable `x`.

The difference with the `get` of Elan-0 is that spaces are considered as separators between texts.

Notice that the text read in this way can not contain spaces and that the new lines in the input are practically invisible. This form of input is evidently particularly suited for more or less linguistic applications.

A generic variant of this one, with a whole number as its second parameter

```
get(x, length)
```

assigns to `x` a text of at most `length` characters, read from the current position without skipping initial spaces. There is no automatic skip over the end of line and spaces at the end of a text are disregarded. Therefore a blank line is read as an empty text.

If one wishes to achieve in Elan the same as the `get` in Elan-0, then one has to use this last version with `length` equal to the line length of the input.

6.6.7 Converting texts: the algorithms `text`, `int` and `real`

There are a number of algorithms for converting values of other types into texts and converting texts into values of other types.

To begin with there is an algorithm with the name `text` (in lower case letters) that converts a `REAL` or `INT` value into a text resembling a denotation for it. An example:

```
TEXT VAR t :: text(123)
```

The value of the integer 123 is converted to a text consisting of some spaces followed by the characters `1`, `2` and `3`, in that order, for example `####123`.

In the other direction (`TEXT` to `INT`), the procedure `int` can be used with, as parameter, a text resembling an integer denotation, possibly preceded by a `+` or `-` sign. For example:

```
put (7 + int(t))
```

now will print the number 130. With the procedure `real` we can convert texts resembling real denotations to `REAL` values:

```
REAL VAR x :: real(t + ".0")
```

assigns to `x` the value 123.0.

Conversely, the procedure `text` can be called with a real parameter and then yields a text resembling a denotation in “floating point” form. As an example,

```
text(0.1234)
```

yields a value like `1.2340000E-1`.

In using those conversion algorithms many things can go wrong, especially when starting out from a text. After all, one wrong character in the text is enough to make conversion impossible, as in

```
int("123a")
```

In order to inquire whether the conversion performed last was successful, a boolean standard procedure, `last conversion ok`, exists that can be used to program a reaction to errors:

```
IF NOT last conversion ok
THEN put ("conversion#failed");
     bang
FI
```

Exercise: Refine the algorithm `bang` using the various cutting, pasting, and converting algorithms you learned about in this chapter.

6.7 Example: A desk calculating machine

In this example we will try to imitate a pocket calculator of the kind that is available in every department store at bargain prices. For simplicity we confine ourselves to the whole numbers and their four basic operations.

The calculator holds a kind of dialogue with us. When it is ready to start computing it gives a dollar sign (as a "prompt"). The input consists of an expression. As an answer, the machine prints the value of that expression and then gives another prompt to indicate that more input is expected. If we want to stop calculating, we give as input the text `halt`. (Of course we would never get rich selling such rubbishy calculators, but we leave it to the reader to design a more realistic one.)

The dialogue between calculator and user could look like:

```
$4*3
      12
$5+8
      13
$halt
```

First we implement the dialogue

```
desk calculator:
   ask for first input;
   WHILE further computation desired
   REP
     print value of formula;
     ask for next input
   ENDREP.

ask for first input:
   TEXT VAR input;
   ask for next input.

ask for next input:
   line;
   put ("$");
   get (input).
```

Using Elan-1 and this version of the generic `get` we cannot put spaces between operands and operators for a space terminates a text. But that is not required.

```
further computation desired:
   input <> "halt".
```

The input will have to be scanned from left to right, in order to determine what operators and operands it contains. We will do this by inspecting the first character of the input and, whenever it has been recognized, removing the first character of the input. We will call that first character the "head" of the input.

```
head:
   input SUB 1.
```

```
skip head:
   input := subtext (input, 2).
```

The rest of the program is based on a number of syntax diagrams, expressing the difference in priority between operations (Fig. 6.4).

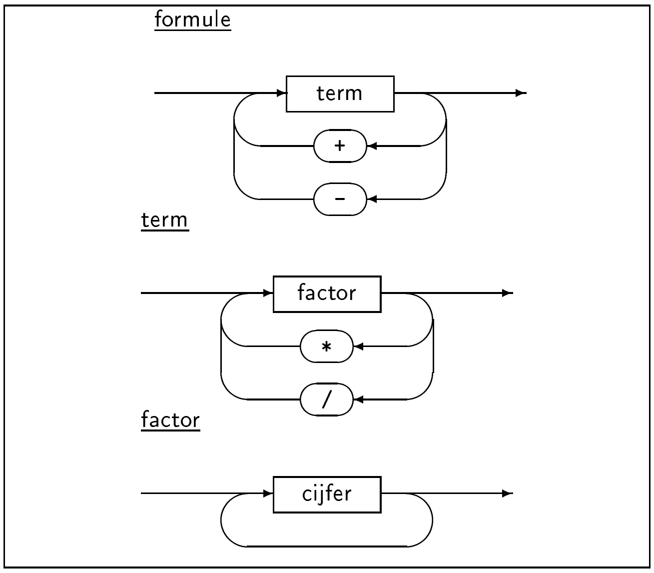


Figure 6.4: Syntax diagrams for desk calculator

For each of the the syntactic notions we introduce a refinement that tries to recognize this notion in the input and yields as value the result of the corresponding expression.

```
formula:
   INT VAR result formula :: term;
   WHILE head = "+" OR head = "-"
   REP
     IF head = "+"
     THEN
       skip head;
       result formula INCR term
     ELSE
       skip head;
       result formula DECR term
     FI
   ENDREP;
   result formula.
```

```
term:
   INT VAR result term :: factor;
   WHILE head = "*" OR head = "/"
   REP
     IF head = "*"
     THEN
       skip head;
       result term := result term * factor
     ELSE
       skip head;
       result term := result term DIV
factor
     FI
   ENDREP;
   result term.


factor:
   IF head = "+"
   THEN
     skip head;
     number
   ELIF head = "-"
   THEN
     skip head;
     - number
   ELSE
     number
   FI.
```

We will recognize numbers in the same way. A number must consist of one or more digits. Notice that two numbers in a formula will always be separated from one another by at least one operator, so that a sequence of digits always forms one number.

```
number:
   IF head is digit
   THEN
     INT VAR value :: 0;
     WHILE head is digit
     REP
       skip head;
       value := 10 * value + digit
     ENDREP;
     value
   ELSE
     error;
     0
   FI.


head is digit:
   INT CONST digit :: pos ("01234567879",
head) - 1;
   digit >= 0.
```

We make explicit use of the fact (deducible from the diagrams) that a number is always followed by an operator. We complete the example

```
print value of formula:
   INT CONST result :: formula;
   IF head = ""
   THEN
     line;
     put (result)
   ELSE
     error
   FI.

error:
   line;
   put ("error#in#formula");
   LEAVE print value of formula.
```

Notice that our desk calculator does not allow spaces in the input. This is realistic in so far as a pocket calculator does not even have a space key, but on a (micro)computer the space key (and a lot of other spurious keys) are present. A more intelligent version of the desk calculator could take this into account.

For all its shortness, the example is not simple. The use of syntax diagrams as guidance in programming is a very powerful technique, which however needs some knowledge of grammars.

The solution given can easily be expressed in Elan-0. In doing so, we have to substitute the standard procedures `subtext` and `pos` by rewriting the refinements that call them:

```
skip head:
   input := TAIL input.


head is digit:
   INT CONST digit :: pos of head - 1;
   digit >= 0.


pos of head:
   INT VAR i :: 1;
   WHILE i <= 10
   REP
     IF ("0123456789" SUB i) = head
     THEN
       LEAVE pos of head WITH i
     ELSE
       i INCR 1
     FI
   ENDREP;
   0.
```

Notice that the name `pos` has also been changed to `pos of head`. `pos` is the name of a number of standard generic procedures and if we redefined it as a refinement then those procedures would become hidden i.e. they could not be called any longer. This is not what we want as it could lead to various problems. In chapter 10 you will learn more about the *scope rules* of procedures and refinements.

A nicer version should skip the spaces in the input stream. But this is left as an exercise to the reader.

6.8 NTO of the texts

An overwiew of the text-operations in the Elan0 subset was given in figure 6.2. In the Elan-1 level we find a number of further operations:

- the text constants `niltext`, `blank` and `quote`;
- the "scissor" procedures `text`, `subtext`, `pos` and `change`;
- the conversion procedures `text`, `int` and `real`;
- another version of the input procedure `get`.

These are depicted in the form of an NTO in Fig. 6.5.

The NTO's of text in their turn form part of a larger framework together with other NTO's which we have not included in this picture.

For a complete set of operations on texts see appendix B.

6.9 Exercises

1. (Palindromes) A "palindrome" is a text that, read from left to right, is the same as read from right to left. In order to obtain interesting palindromes, it is customary to admit, besides the letters, spaces and other punctuation marks, but they play no role in the comparison. E.g. a palindrome from the last century is: *a man, a plan, a canal: panama.*

   Write an algorithm that reads a line of text and reports whether the letters occurring in it form a palindrome.

2. (Cryptology) In a simple form of secret code, all letters and the space are shifted circularly by some number of places. For example, shifting the alphabet by 3 places to the left we write *d* instead of *a*, *e* instead of *b*, *b* instead of *z*, *c* instead of space, etc.

   Write a program that reads a text and then prints it with circular shift 1,2,...,26.

   Use that program to read the following (partly distorted) messages: "zerdidomdidrizixdriqdmwqivm" and "txvwidradrzxbiwtvpgstc".

3. (Arabic to Roman) Write a program that reads a whole number and prints it out in Roman numerals.

4. (Roman to Arabic) Write a program that reads a number in Roman numerals and prints it out in the decimal system.

5. (Calculator) Modify the `ask for next input` refinement (and also others if needed) in order to skip possible spaces in the input. Recall that the procedure `get` with a single text parameter has a slightly different meaning in Elan-0 and Elan-1.

6. (Children's language) Children like "secret" speech. In one of their favourite languages each vowel is doubled and the consonant *b* is inserted between them. For example, the word *today* will be spelled as *to*bo*da*bay [KLI85].

   Write a program that reads a sentence and then converts it into this language.

7. (Ticket vendor machine I) Program a simple ticket vendor machine that endlessly

   - informs about the available tickets (for the sake of simplicity, use a number of price categories),
   - asks for the ticket required,
   - requests the money until the paid sum reaches or exceeds the price,
   - returns the excess money, if any, and issues the ticket.

8. (Ticket vendor machine II) Program a more realistic ticket vendor machine, performing the same task as the previous one, but working with a limited set of coins, say, 1, 2 and 5 ducats. Initially, the machine has a limited supply of coins. If, later on, there will be any possibility that the excess money can not be returned, it should display the message `Exact payment, please!` and refuse excess coins. You may also program a cancel button.

9. (Supermarket bill) Program a cash-register that, with a limited choice of articles,

   - accepts (abbreviated) names and amounts of purchased goods;
   - prints bills containing names, unit prices, subtotals and total;
   - tracks sales and stocks, and gives warning if a stock becomes too low;
   - sums up the daily takings and calculates the (article-dependent) tax.

   If you want your program to be realistic explore a near-by supermarket.

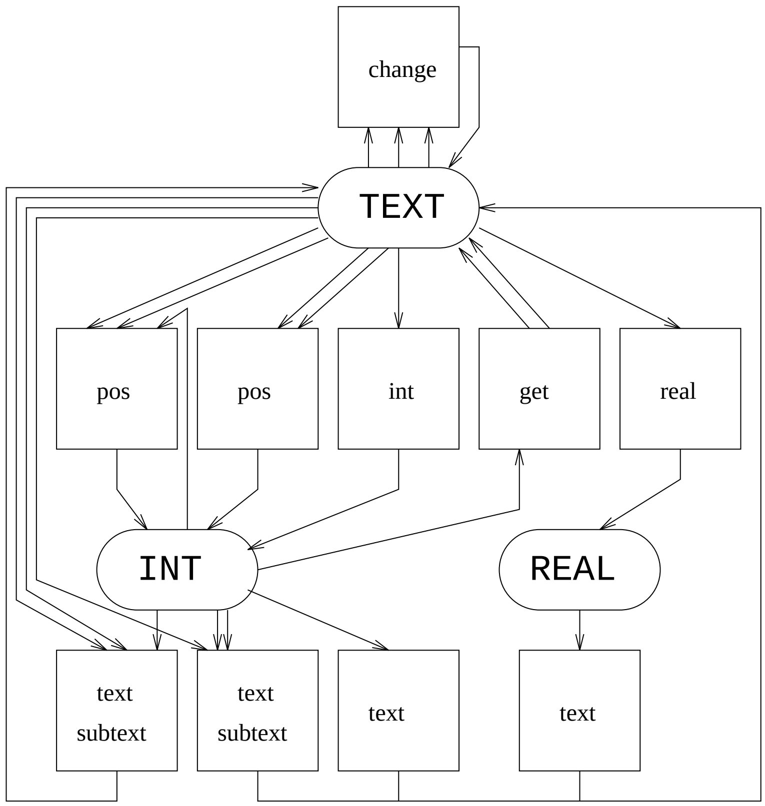


Figure 6.5: Text operations in Elan-1

Chapter 7

Control structures

The control structures are amongst the most important stylistic properties of a programming language and to a large extent determine its appearance for the programmer. They serve for the construction of composed algorithms, just as data structures are means for the construction of composed objects. Syntactically, control structures indicate how a number of algorithms can be composed to make one algorithm that in its turn can be used in the composition process. They are the cement of the programming language. The control structures also decide how the meaning of the composed algorithm is to be expressed in terms of the meaning of its parts, i.e. how its value and its effect are to be expressed in terms of the value and the effect of its subalgorithms. Control structures control the execution of the program, which gives them their name.

We have already described most of the control structures of Elan. In this chapter we shall give some additions and clarifications.

7.1 Paragraphs and units

One of the most fundamental control structures in algorithmic languages is the *sequence,* realized in Elan by the *paragraph* (Fig. 7.1).

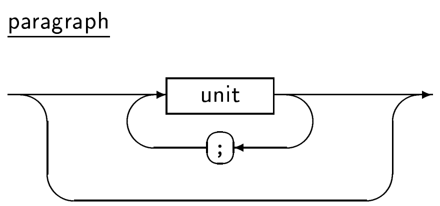


Figure 7.1: Paragraph

A paragraph consists of a number of units, separated from one another by *semicolons.* Notice that a paragraph may even be empty.

Such a unit (Fig. 7.2) in its turn can be either a simple form of *declaration* or an expression or a composed-unit (a control structure, see Fig. 7.3).

In reality (see appendix A) the syntax is more complicated, but these diagrams provide sufficient detail.

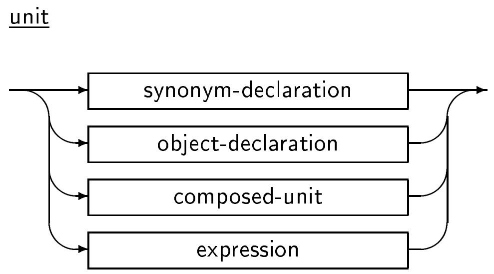


Figure 7.2: Unit

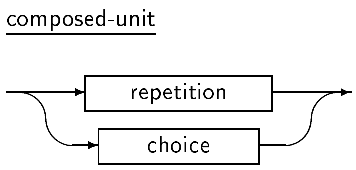


Figure 7.3: composed unit

7.1.1 Effect and value of a paragraph

The effect and value of a paragraph can be deduced from the value and effect of its parts.

The execution of a paragraph consists of the execution in the given order of its units (subalgorithms). Therefore the effect of that execution also consists of the serially composed effect of its subalgorithms. The value of a paragraph ending on an expression is the value of that expression; paragraphs not ending on an expression are actions and yield no value.

Observe that every unit in a paragraph can itself be a composed-unit.

7.2 Expressions

Expressions are composed of operands and operators (Fig. 7.4 and 7.5).

We have already dealt with most forms of operand; the others will follow later.

7.2.1 Priority of operators

Observe that the syntax diagram for expression is ambiguous for expressions with more than one dyadic operator. In order to give to expressions their (unambiguous) meaning we have to take into account the *priority*

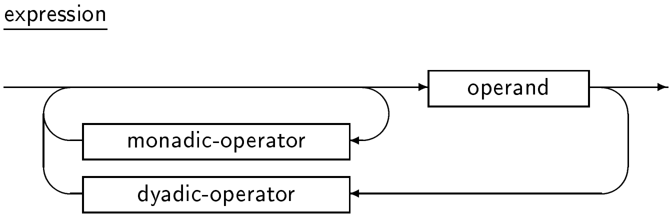


Figure 7.4: Expression

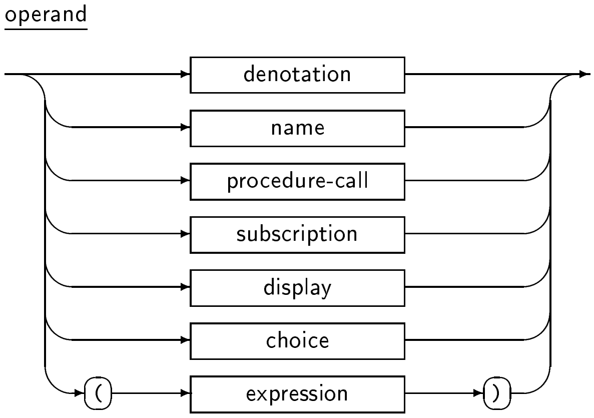


Figure 7.5: Operand

of the operators. These priorities (from low to high) can be deduced from Table 7.1.

The operators with the highest priority bind most strongly. Operators with the same priority bind from left to right. As an example

```
2 + x DIV 2 DIV a + 3 - c
```

means the same as

```
(((2 + ((x DIV 2) DIV a )) + 3) - c)
```

In evaluating an expression, the operands of the operator with the lowest priority are evaluated, after which the operator is applied to their values. The same holds for the evaluation of those operands which are, in their turn, expressions. The effect is, loosely speaking, that the operations with the highest priority are performed first.

7.2.2 Side effects

Both operands of a dyadic operator are evaluated collaterally, i.e. nothing can be said about the order of their computation. In a program like

```
INT VAR x;
x:= 0;
put (a + b + x).

a:
  x:= x + 4;
  1.

b:
  x:= 2 * x;
  2.
```

1. The assignment and the operators combined with an assignment

   `:= ::  INCR DECR CAT`

2. All abstract dyadic operators (declared by the user, using a form of declaration to be introduced later) as well as `SUB`

3. `OR XOR`

4. `AND`

5. `= <>`

6. `<= < >= >`

7. The dyadic operators

   `+ -`

8. `* / DIV MOD`

9. `**`

10. All monadic operators, such as

    `+ - LENGTH ABS SIGN`

Table 7.1: Priority of operators

you can not assume that `a`, `b`, and `x` are computed in that order in expression `a + b + x`: any other order is also correct! Therefore you do not know whether the result printed is 11 (first `a`, then `b`, finally `x`) or 7 (first `b`, then `a`, then `x`) or even 3 (first `x`, then `a`, then `b`).

A program like this, in which the execution of one operand influences the value of another operand (a *side effect)* must therefore be avoided. Keep in mind that the textual order of the operands and of the operations need not at all be the order of execution.

7.3 Object declarations

Declarations in general serve to define an object by binding a name to a value. The simplest declarations are the two kinds of object declaration (see Fig. 7.6).

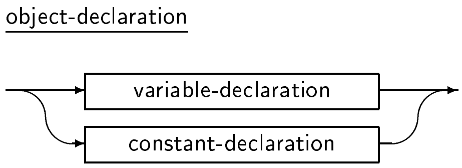


Figure 7.6: Object declaration

You are already familiar with both forms of object declaration (see Fig. 3.2.1 and Fig. 3.6. These differ only in their access attribute and in the fact that a variable need not be initialized whereas a constant must be.

7.3.1 Effect of an object declaration

The execution of an object declaration with a unit as initialization has as effect that this unit is executed and an object is introduced with the given type, access attribute, name, and the computed value. If the initialization is missing, an (as yet) undefined value is taken. A declaration does not yield a value.

7.3.2 Scope of declarations

An object declaration is valid in a specific *scope,* i.e. within a specific part of the program a specific object is indicated with that name.

In the kind of programs we have been writing until now, consisting of a paragraph and some refinements, every declaration is valid in the whole program. In Elan constructs to be introduced later, more limited scopes may appear.

Notice that an object may not be used before its declaration has been executed: it does not have a value yet. It is of course possible to mention the name of an object earlier in the program, for example in a refinement which textually precedes the refinement in which the declaration occurs. This does no harm as long as the object has obtained a value at the moment it is first used.

Within the scope of a declaration, no other declaration may occur with the same name. For every object in the program there has to be exactly one declaration. Of course this one declaration may appear within a repetition, for example

```
WHILE i <= LENGTH message
REP
   TEXT CONST sign:: message SUB i;
   put (3 * sign);
   put (".");
   i INCR 1
ENDREP
```

The object `sign` is a constant that in every turn of the repetition has as value the next sign of the `message`. Yet it is a constant, because it is impossible to assign to it. It is a constant whose value is different at different moments (admittedly a strange kind of constancy — it resembles a person who falls in love time and again, and each time is sure that true happiness has now been found).

7.4 The choice

A *choice* is made between a number of paragraphs depending on either a truth value or on a number (Fig. 7.7).

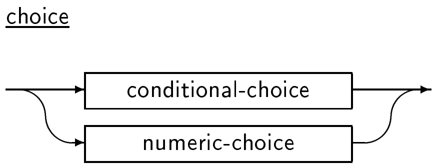


Figure 7.7: Choice

7.4.1 Conditional choice

The choice based upon a condition has already been introduced. Its syntax diagram is shown in Fig. 7.8.

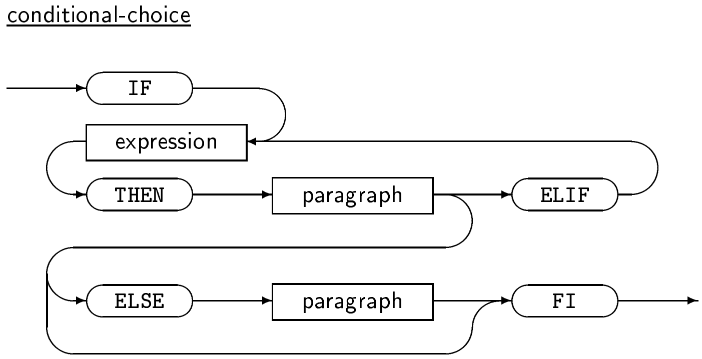


Figure 7.8: Conditional choice

There are many paths through this syntax diagram. In particular the `ELSE`-part may be absent and there may be any number of `ELIF`-parts. An alternative representation for `FI` is `ENDIF`.

The meaning (value and effect) of a *conditional choice* is the meaning of the paragraph that is executed on the basis of its condition. We will not describe this meaning any further but draw the attention to some details.

The choice can be made between paragraphs that do not yield a value, so that the choice itself also does not yield a value. Only in that case the `ELSE`-part may be omitted. If all paragraphs deliver a value of one specific type then this is also the type of the result of the choice. In this case, the `ELSE`-part is obligatory.

The `ELIF`-construction serves to distinguish between a larger number of possibilities without incurring an avalanche of `FI`-brackets (this form is called also *multiple choice*):

```
IF   x > 0.0 AND y > 0.0
THEN put ("point lies in first quadrant")
ELIF x < 0.0 AND y > 0.0
THEN PUT ("point lies in second quadrant")
ELIF x < 0.0 AND y < 0.0
THEN put ("point lies in third quadrant")
ELIF x > 0.0 and y < 0.0
THEN put ("point lies in fourth quadrant")
ELSE put ("point lies on an axis")
FI
```

For a declaration appearing in the `THEN`- or `ELSE`-part, the scope is the whole program, as said above. The object declared in this fashion can not be used to bring a result to the outside, because there also exists a way through the program on which this declaration is not executed.

```
IF the first case
THEN
  INT VAR result;
  compute result in the first way
ELSE
  compute result in the other way
FI;
put (result)
```

is wrong, because it is possible to follow a way through the program in which the declaration of `result` is not executed. It does not help to put a declaration for `result` in both the THEN-part and ELSE-part: in that case the program contains two declarations with the same name, which is forbidden:

```
IF the first case
THEN
  INT VAR result;
  compute result in the first way
ELSE INT VAR result;
  compute result in the other way
FI;
put (result)
```

The declaration will have to take place before the choice.

```
INT VAR result;
IF the first case
THEN
  compute result in the first way
ELSE
  compute result in the other way
FI;
put (result)
```

7.4.2 Numerical choice

Elan knows another form of choice, the *numerical choice* or SELECT-construct, that serves to choose a specific paragraph on the basis of a whole number. In other languages, this control structure may be called *case clause*, *switch* or *computed goto*. This construct is not included in Elan 0. Its syntax is depicted in Fig. 7.9.

This is quite complicated, therefore we first give an example:

```
weekday:
   SELECT day OF
     CASE 1: "monday"
     CASE 2: "tuesday"
     CASE 3: "wednesday"
     CASE 4: "thursday"
     CASE 5: "friday"
     CASE 6: "saturday"
     CASE 7: "sunday"
     OTHERWISE "no day at all"
   ENDSELECT.
```

We assume here that the days of the week have been encoded in the obvious fashion as the whole numbers 1 to 7 and the numerical choice maps these codes onto texts. This solution is somewhat shorter and may be somewhat faster than through a cascade of ELIFs.

The CASE-parts in this example were very simple; of course, they may be much more complicated. Observe that the cases need not be ordered, and that two cases with the same paragraph can be taken together, as in

```
is the digit even:
   SELECT digit OF
     CASE 0, 2, 4, 6, 8: true
     OTHERWISE false
   ENDSELECT.
```

(This might be programmed in a simpler fashion using MOD).

The SELECT-construct is of most use in mapping whole numbers onto a choice between algorithms. As we have seen in the previous examples, it can also yield a value but then each case must yield the same type of value; even the — otherwise optional — OTHERWISE-part.

7.5 The repetition

For *repetition*, Elan has a number of notations that all are variants of one same basic form. The repetition is a control structure that does not yield a value.

7.5.1 The basic form

The simplest forms of the repetition are the *prechecked loop*

```
WHILE there is something to do
REP
  do it
ENDREP
```

and the *postchecked loop*

```
REP
  whatever there is to do
UNTIL ready
ENDREP
```

If we omit both the WHILE- and UNTIL-part, we obtain a repetition which is in principle *unending,* like the following loop which appears in many operating systems:

```
REP
  read a program text;
  translate it and check for errors;
  IF the text contains no errors
  THEN
    execute the program;
    print the results
  ELSE
    print the errors found
  FI
ENDREP
```

Only a disaster (or operator action) can stop this repetition.

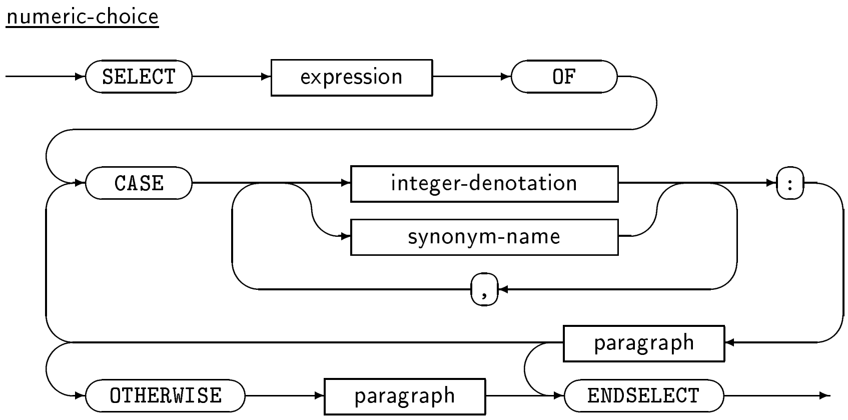


Figure 7.9: Numerical choice

Many non-sequential algorithms, that are not covered in this book, make use of indefinitely repeating processes.

7.5.2 The controlled variable

In many repetitions, a variable appears whose value is changed incrementally until a specific situation is reached. In most cases this is an integer variable, the *controlled variable,* whose value is increased by one as long as a specific condition holds or until a specific limit is reached, e.g.:

```
INT VAR i:: 1;
WHILE i <= limit
REP
  repeated action;
  i INCR 1
ENDREP
```

This example shows the following characteristic elements:

- the *controlled variable* `i`;
- the *initialization* `:: 1`;
- the *test* `i <= limit`;
- the *repeated action* depending on `i`;
- the *incrementing* of the controlled variable `i INCR 1`.

In the order of these elements, there are a number of degrees of freedom.

It is, for example, possible to exchange the incrementing and testing, which then must be done slightly differently. Where we are assured that `limit > 0` we can write

```
INT VAR i:: 1;
REP
  repeated action;
  i INCR 1
UNTIL i > limit
ENDREP
```

and also

```
INT VAR i:: 0;
WHILE i < limit
REP
  i INCR 1;
  repeated action
ENDREP
```

In proving the correctness and termination of an algorithm the controlled variable often plays a central role. After completing the text of a program, it is good practice to check whether all controlled variables will start and end with the intended values, before executing the program for the first time.

7.5.3 The limited repetition

The use of the prechecked loop carries the danger that one forgets to initialize the controlled variable. For the normal case, that is when this is an integer variable that has to step by one through a specific range, the programmer can express his intentions by

```
FOR index FROM start value UPTO limit
value
REP
  repeated action
ENDREP
```

or

```
FOR index FROM start value DOWNTO limit
value
REP
  repeated action
ENDREP
```

This construct, the *limited repetition* or `FOR`-loop, has the following properties:

- `index` must be an integer variable that has been declared before. Its value before the repetition is irrelevant. After the repetition it has the first value rejected.

- `start value` and `limit value` have to be integer expressions. They are *computed collaterally once,* at the beginning of the repetition.

  > Programming languages usually evaluate *first* the `start value` and *then* the `limit value` as the latter may depend upon the first. The designers of Elan preferred the collateral execution for it allows, in principle, the parallel evaluation of the two expressions provided more than one arithmetic units are available. The parallel evaluation of expressions may speed up programs a lot therefore this topic has great significance in modern programming languages.

- The controlled variable `index` runs through a specific *repetition range* with step size 1 in case of `UPTO` and -1 in case of `DOWNTO`, starting at the `start value`, until the `limit value` is passed. For each of the (zero or more) values in this range, the `repeated action` is performed once.

Examples (what results are displayed?):

```
FOR t FROM 1 UPTO 10 REP put (t) ENDREP
```

Displays the result 1 2 3 4 5 6 7 8 9 10.

```
FOR t FROM -10 UPTO 0 REP put (t) ENDREP
```

Displays -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0.

```
FOR t FROM 10 DOWNTO 1 REP put (t) ENDREP
```

Displays 10 9 8 7 6 5 4 3 2 1.

```
FOR t FROM 10 DOWNTO 1 REP put (11 - t)
ENDREP
```

The result 1 2 3 4 5 6 7 8 9 10 is displayed.

```
FOR t FROM 10 UPTO 10 REP put (t) ENDREP
```

Displays 10.

```
FOR t FROM 1 UPTO -1 REP put (t) ENDREP
```

Nothing is displayed: since the step size is positive, 1 is already beyond -1.

7.5.4 Abbreviated forms

This form of repetition admits a number of *abbreviations*:

- `FOR index` may be left out if the controlled variable does not occur in the `repeated action`.
- `FROM 1` may be left out.
- `UPTO maxint` may be left out. (Remember: `maxint` is the largest integer representable in Elan.)

Example:

```
UPTO 3 REP line ENDREP
```

has the same effect as

```
line (3)
```

7.5.5 Limited repetition with conditions

Combining the limited repetition with the conditional repetition, one obtains the most general form of repetition (Fig. 7.10).

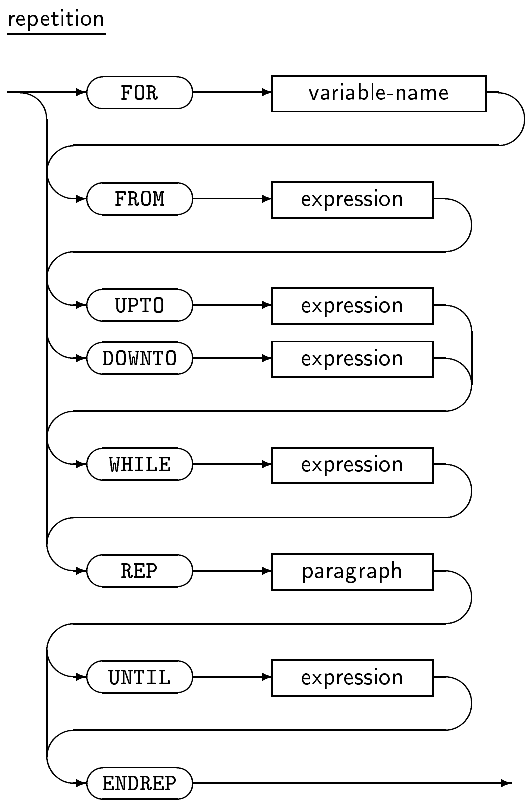


Figure 7.10: Repetition

This construct is reminiscent of a Swiss army knife with large and small blades, a small screwdriver, bottle opener and corkscrew and even a very small hammer and tongs. In spite of its overwhelming multi-purpose character it is still a practical instrument, because one can leave out the parts one doesn't need.

7.5.6 About the controlled variable

In a limited repetition, it is intended that the counting variable runs through a specific range. If the `repeated action` itself modifies the value of `i`, the program can be very hard to understand. It is better to avoid tricks and to let the controlled variable go its way without side effects. In order to convince the reader we do not answer the question of what the following piece of program means. Is it the counting mechanism of the repetition or the assignment in its body that carries the day?

```
INT VAR i;
FOR i FROM 1 UPTO n
REP
  put (i);
  i:= 2 * i
ENDREP
```

If the controlled variable of the limited repetition is allowed to run through its range, after the repetition it has the first value rejected, i.e. the first value it obtained that does no longer lie in the range.

7.5.7 Example: Trailing blanks

We give an example of the interplay between a limited repetition range and a condition in a WHILE-part. We want to eliminate from a text the blanks at its end. To this end we have to determine the position of the first trailing blank, i.e. of the first space in the line which is followed only by spaces.

```
first trailing blank:
   INT VAR place;
   FOR place FROM LENGTH text DOWNTO 1
   WHILE another space
   REP
   ENDREP;
   place + 1.


another space:
   (text SUB place) = " ".
```

When there are no trailing blanks, this yields the value `1 + LENGTH text` and if the whole text consists of spaces it yields the value `1`.

7.6 Conclusion

This was a somewhat tough chapter, full of syntax diagrams and nitpicking, with few convincing examples, but it was necessary to take a closer look at a number of constructions we have been using for some time rather loosely, as a preparation to the introduction of composed objects.

7.7 Exercises

1. (Morse code) Consider the problem of converting a text to morse code. In the morse alphabet every sign is represented by a sequence of short and long signals with pauses in between. A short signal is called a *dot,* a long signal a *dash.* The morse alphabet looks as follows:

A	. –	K	– . –	U	. . –	1	. – – – –
B	– . . .	L	. – . .	V	. . . –	2	. . – – –
C	– . – .	M	– –	W	. – –	3	. . . – –
D	– . .	N	– .	X	– . . –	4	. . . . –
E	.	O	– – –	Y	– . – –	5	. . . . .
F	.. – .	P	. – – .	Z	– – . .	6	– . . . .
G	– – .	Q	– – . –	Ä	. – . –	7	– – . . .
H	. . . .	R	. – .	CH	– – – –	8	– – – . .
I	. .	S	. . .	Ö	– – – .	9	– – – – .
J	. – – –	T	–	Ü	. . – –	0	– – – – –

Period	. – . – . –
Error	. . . .
SOS	. . . – – – . . .
Start of message	– . – . –
End of message	. – . – .

Between two letters we give one space, between two words we give 3 spaces. Write a program to convert a message consisting of one line to morse. You may make use of a numerical choice.

2. (Decoding morse) Write a program that reads a line of morse code and deciphers it. Try it out, together with the previous program.

Chapter 8

Composed objects: Rows

In the previous chapters we have dealt with the elementary types of Elan. In this chapter we introduce a mechanism for the composition of types from elementary types and discuss the meaning and use of composed objects.

8.1 Values, objects and types

A composed object is an object that can assume values of a composed type. Before discussing composed types, we shall recapitulate a number of important concepts and terms that have been introduced in the course of the preceding chapters.

8.1.1 Recapitulation of terminology

To begin with, there are ideas and physical objects, facts and figments of the mind. In writing a program, we may devise for these some representations in the computer, in order to perform some computations on those representations. By a *value* we mean an element of some abstract set for which there exists a representation in the computer, as well as that representation itself.

When we speak of “the value three” we may mean both a specific familiar mathematical concept, the follower of two, and some particular representation for it in the computer. In a wider context, we can even speak of “the value of the dollar” or “the values of Western Civilization”. The word value has a multiple meaning. Usually we are not bothered about such ambiguities in the meaning of words, since the context will make clear which particular meaning is intended.

We cannot manipulate values directly, but instead write algorithms that achieve the intended result. In such an algorithm we may denote a value directly, by a *denotation*, a notation in the programming language that serves to denote a specific value. We may also give a *name* to a value, by means of a declaration or an assignment. The name is our handle on that value. The combination of a name and a value (bound to it in the course of the execution of the algorithm) we call an *object.*

The binding of a value to a name is temporary. In the course of time, to one name different values may be bound, for instance by the repeated execution of a declaration with different values for its initialization. Some objects (the *variables)* allow the explicit replacement of their value by another, by means of an assignment. Others (the *constants)* do not allow such an assignment.

The values can be distinguished into different classes, the *types.* Each type has a name and a collection of operations, applicable to values of that type. The set of values belonging to a specific type is called its *domain.*

The type of an object is the type its values may assume. The type of an object may be elementary (given by the name of that type) or composed (constructed out of named types by means of a *data structure).* In this chapter we shall be concerned with objects of a specific composed type, the `rows.`

8.1.2 Type declarers

In a declaration for a variable or constant, its type is indicated by a *type declarer* (Fig. 8.1).

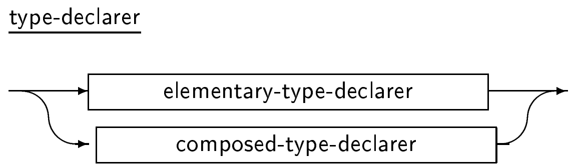


Figure 8.1: Type declarer

The simplest example of a type declarer is the name of an elementary type (Fig. 8.2).

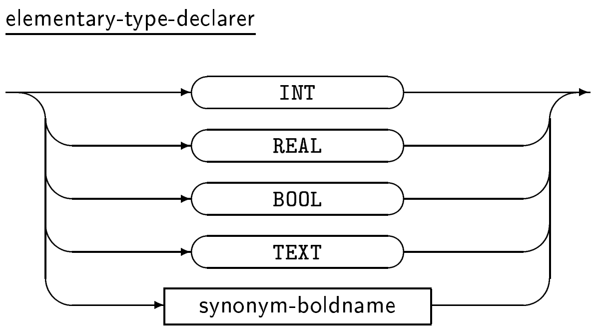


Figure 8.2: Elementary type declarer

We already know the four concrete types. The possibility of introducing abstract types by means of a synonym-boldname is introduced later.

The *rows* and the *structures* are the data structures of Elan. (Fig. 8.3). In this chapter we deal only with rows. Structures will be described in the second volume of this book.

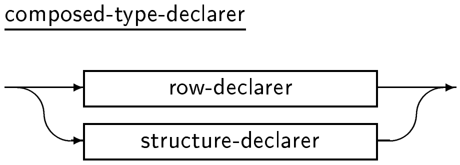


Figure 8.3: Composed type declarer

8.2 Rows

By a (one-dimensional) row we mean a collection of a limited number of objects of one same type, the *elements* of the row. These elements are numbered contiguously from a lower bound 1 upto some *upper bound,* equal to the number of elements ("cardinality"). The position of an element in the row is called its *index.*

The type of a row is indicated by a *row declarer* in which the number and the type of the elements is stated. The syntax for the row declarer is given in Fig. 8.4.

The representation of the row-symbol is `ROW`. After the row-symbol follows the number of elements. Later, in the context of synonym-declarations, we shall discuss the possibility of indicating the number of elements by a synonym-name. Examples:

- `ROW 10 REAL` A row of ten reals numbered from 1 to 10.
- `ROW 200 ROW 60 TEXT` A likely representation for a book consisting of 200 pages of 60 lines each.

Just like the declarers `INT`, `REAL`, etc. introduced before, the row declarers are used to indicate types in object declarations. Example:

```
ROW 10 REAL VAR old temperature, new
temperature
```

We can consider a row as one single object and assign it in one fell swoop to a suitable variable:

```
old temperature := new temperature
```

After the assignment the value of `old temperature` will be a copy of the value of `new temperature`. Apart from assignment, there are no concrete operations on a row as a whole.

8.2.1 Access to the elements of a row

The elements of a row can be manipulated separately. One can take the value of an element ("read access") or give another value to the element ("write access"). The latter is possible only for the elements of a *row-variable.*

In the context of the declaration of the row-variable

```
ROW 10 REAL VAR temperature
```

the value of the `ith` temperature is indicated by

```
temperature[i]
```

Its value can be printed, for example, by

```
put (temperature[i])
```

and can be changed by the assignment

```
temperature[i] := 0.0
```

This construct is called a *subscription* (Fig. 8.5).

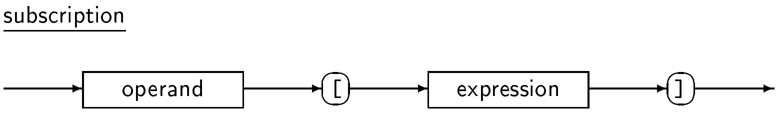


Figure 8.5: Subscription

The value of the *integer expression* is called the index. The index must have a value between the lower bound 1 and the upper bound of the row; otherwise, the subscription is undefined and the execution of the program is terminated.

Assigning a value to one of the elements of a row leaves the other elements unchanged. The program fragment

```
FOR i UPTO 10
REP
  old temperature[i] := new temperature[i]
ENDREP
```

has the same effect as the assignment

```
old temperature := new temperature
```

in the previous section.

8.2.2 Inheritance of the access attribute

The subscription `temperature[i]` in all aspects behaves like a real variable: it inherits the access attribute `VAR` from the row which is subscribed. Letting EL stand for the type of the elements,

> Subscription of an EL-row-variable yields an EL-variable.
> Subscription of an EL-row-constant yields an EL-constant.

The elements of a composed constant are also constants. They can not be modified by an assignment.

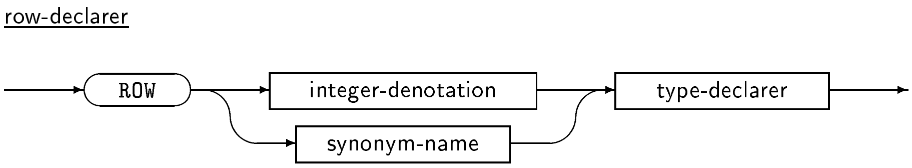


Figure 8.4: Row declarer

8.2.3 Denotation of one-dimensional rows

In distinction to the elementary types, INT, REAL, etc., the row-constants have no proper denotation. In its stead comes another more general control structure that constructs a composed object from the values for its elements, the *display.* This construct plays the role of denotation for rows. Its syntax is shown in Fig. 8.6.

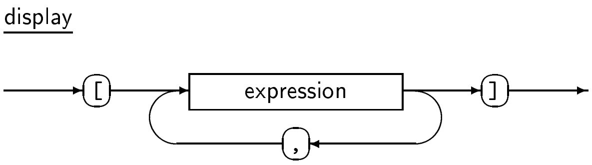


Figure 8.6: Display

Displays have the following constraints:

- The expressions in a display, used to denote a row, all must have one same type, the type of the elements of the intended row.
- Their number determines the upper bound of the row obtained.

Examples:

`[0, 1, 3 DIV 2, 7]`	is a	`ROW 4 INT`-display
`[0.0, 1.0, 3.0/2.0, 7.0]`	is a	`ROW 4 REAL`-display
`["I", "you", "he"]`	is a	`ROW 3 TEXT`-display

We can give a name to the value of a display by means of a constant-declaration.

```
ROW 10 INT CONST first 10 primes ::
                 [2, 3, 5, 7, 11, 13, 17,
19, 23, 29]
```

8.3 Example: Counting words

As an example of the use of rows we shall consider the problem of determining from an input, consisting of words, the frequency of each word, i.e. the number of times it appears. We use one row each to remember the words and their frequencies.

```
counting words:
   start with an empty list;
   WHILE another word follows
   REP
     count that word
   ENDREP;
   print the list.
```

We reserve space for at most 100 words.

```
start with an empty list:
   ROW 100 TEXT VAR word list;
   ROW 100 INT VAR frequencies;
   INT VAR first free place :: 1.
```

The end of the input we shall indicate by a word consisting of nine letters "z".

```
another word follows:
   TEXT VAR word;
   put("Next word, please: ");
   get(word);
   line;
   word <> "zzzzzzzzz".
```

A word is entered into the list only once.

```
count that word:
   IF the word already appears in the list
   THEN
     increment its count
   ELSE
     enter it
   FI.
```

The word list is not ordered, so we have to scan through it all in order to look for a particular word.

```
the word already appears in the list:
   INT VAR place;
   FOR place FROM 1 UPTO first free place
- 1
   REP
     IF word list [place] = word
     THEN
       LEAVE the word already appears in
the list WITH true
     FI
   ENDREP;
   false.
```

The counting is simple.

```
increment its count:
   frequencies [place] INCR 1.
```

```
enter it:
   IF first free place > 100
   THEN
     no place left
   ELSE
     word list [first free place] := word;
     frequencies [first free place] := 1;
     first free place INCR 1
   FI.
```

What do we do when the word list is full, i.e. a hundred different words with their frequencies have already been entered and any further word we have now cannot be added? One possibility is to stop altogether

```
no place left:
   put ("Word list is full.");
   print the list;
   LEAVE counting words.
```

Another possibility is to continue counting the occurrences of the one hundred words already entered and ignore any additional words. Then this last refinement becomes yet simpler.

```
no place left: .
```

We can print the words in any order.

```
print the list:
   line(2);
   put("Frequencies of words:");
   line;
   INT VAR w;
   FOR w FROM 1 UPTO first free place -1
   REP
     put(frequencies[w]);
     put("  " + word list[w]);
     line
   ENDREP.
```

In order to obtain two regular columns, we put the frequencies first (for an integer uses a fixed number of character positions on the screen).

8.4 Nested rows

A row in its turn is an object with a specific type and therefore can occur as an element in another row. We can use a composed type declarer in a declaration like

$$\underbrace{\texttt{ROW 200 }\underbrace{\texttt{ROW 60 TEXT}}_{\text{declarer}}}_{\text{declarer}}\texttt{ VAR book}$$

and can continue with declarations like

```
ROW 10000 ROW 200 ROW 60 TEXT VAR library;
ROW 200 ROW 60 TEXT VAR book;
ROW 60 TEXT VAR page
```

Duplicating a `book` in the `library` can be done by an assignment

```
library[4711] := library[9999]
```

Obviously this is easier said (written) than done.

A composed row has a composed subscription,

```
library[12][14][8]
```

is an indication of the 8th line on the 14th page of the 12th book of the `library`.

A display for a row of rows of course is a *nest of displays.* The outermost brackets of this nest display belong to the leftmost row, and the innermost brackets belong to the rightmost row, while the depth of nesting has to be equal to the depth of nesting of the rows. Examples:

```
ROW 15 ROW 2 TEXT CONST translation from
Dutch ::
  [
   ["aap",       "monkey"],
   ["noot",      "nut"],
   ["Mies",      "Mary"],
   ["Wim",       "Bill"],
   ["zus",       "sister"],
   ["Jet",       "Harriet"],
   ["Teun",      "Tony"],
   ["vuur",      "fire"],
   ["Gijs",      "Gilbert"],
   ["lam",       "lamb"],
   ["Kees",      "Cornelius"],
   ["bok",       "he-goat"],
   ["weide",     "meadow"],
   ["does",      "poodle"],
   ["schapen",   "sheep"]
  ]

ROW 3 ROW 3 INT CONST magic square ::
 [[8, 3, 4,], [1, 5, 9], [6, 7, 2]]
```

8.5 On the bounds of rows

The lower bound of a row is always 1. The upper bound must be a whole number that cannot depend on the execution of the program. It must be a denotation, not be the result of a computation. The reason for this highly restrictive rule is that the number of elements determines the type of the row:

```
ROW 3 INT
```

and

```
ROW 4 INT
```

are different types, and assigning a value of the one type to a variable of the other type is impossible.

Usually the upper bound is repeated in more than one place of the algorithm (in the row declaration, as the limit for a repetition, etc.). If I want to change this upper bound then I have to make changes in a number of places of the program. This is highly annoying and for that reason a special mechanism has been introduced in Elan, the synonym-declaration, that gives a name to a denotation. We give the syntax diagram for this synonym-declaration in Fig. 8.7.

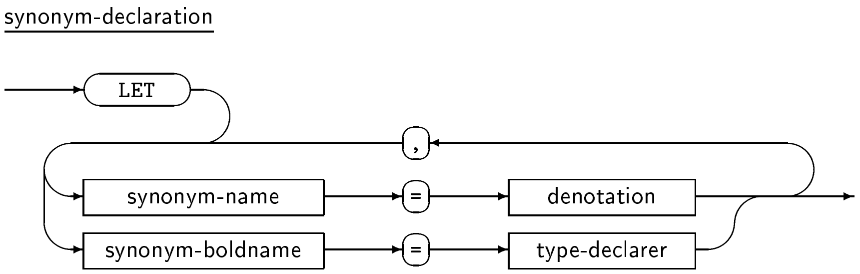


Figure 8.7: Synonym-declaration

We will discuss here only the first form. An example is:

```
LET max = 200
```

This declaration introduces a synonym `max` for the denotation 200 that can be used everywhere where 200 may appear, e.g.:

```
ROW max REAL VAR temperature;
FOR i UPTO max
REP
  temperature[i] := 0.0
ENDREP
```

In this way we can abstract from the exact number of temperatures. The synonym-declaration is an aid in abstraction.

It can not be denied that Elan has a confusingly large number of ways to bind a name to a value. The differences between the following constructions are rather subtle.

- `INT VAR max :: 200`

  the name `max` gets the value of 200. It can obtain another value by an assignment.

- `INT CONST max :: 200`

  The name `max` gets the value of 200 and this value can not be changed by an assignment.

- `max : 200.`

  Every time it is invoked, the refinement `max` yields the value of 200.

- `LET max = 200`

  The name `max` becomes a synonym for the denotation 200.

Only when declared in a synonym-declaration can a name be used as the upper bound of a row.

8.6 Sorting

We shall now discuss a number of simple algorithms for the problem of sorting a row. Assume we have a row-variable with n elements $x_1, x_2, \ldots, x_n$ and an ordering relation $\leq$ between the elements. We say that this row is sorted if, for any i and j, $i < j \rightarrow x_i \leq x_j$. By sorting we mean a shuffling of the elements of the row-variable until they satisfy the ordering relation, without gaining or losing elements. In terms of the programming language we have to achieve this by assigning the values of the elements to other elements of the row-variable x. We call this also *in situ* sorting, in distinction to the possibility of leaving the original row undisturbed and building up the sorted row in another row-variable.

For each of the following algorithms, we assume the following declarations:

```
LET n = 1000;
ROW n EL VAR x
```

Here we mean by `EL` the type of the elements (you may fill in `INT` or `TEXT` or some such), `n` is the number of elements and therefore the index of the last element.

8.6.1 Example: Selection Sort

The idea behind this algorithm is as follows.

> We split the row into a sorted part (initially empty) and an unsorted part. In the unsorted part we repeatedly look for the smallest element which we then glue behind the sorted part, that therefore grows by one element (see Fig. 8.8).

```
selection sort:
   initially the sorted part is empty;
   WHILE not all elements in place
   REP
     take the smallest from the unsorted
part;
     glue this behind the sorted part
   ENDREP.

initially the sorted part is empty:
   INT VAR i :: 0.
```

Observe that here, as in so many examples, an initialized declaration has a clear abstract meaning that can easily be verbalized.

```
not all elements in place:
   i < n - 1.
```

We do not have to sort the end-element, because as soon as the first `n-1` elements are in their place the `n`th element has to be also.

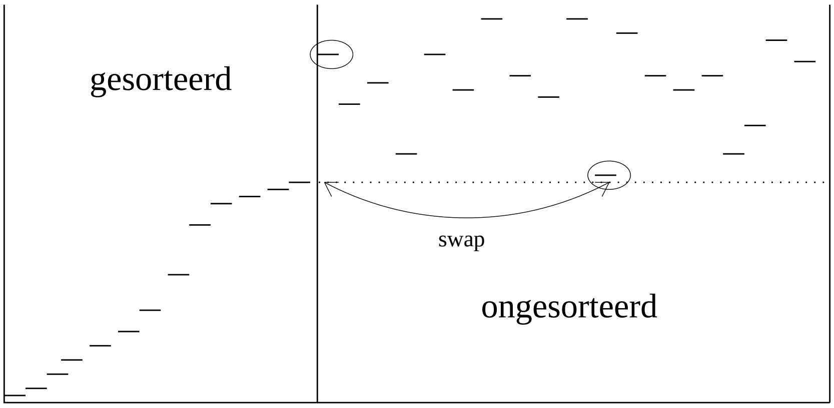


Figure 8.8: Selection Sort

```
take the smallest from the unsorted part:
   start at the first;
   WHILE not all considered
   REP
     look whether you have got a smaller
one
   ENDREP.

start at the first :
   INT VAR j :: i + 1;
   EL VAR smallest :: x[j];
   INT VAR index smallest :: j.

not all considered:
   j INCR 1;
   j <= n.

look whether you have got a smaller one:
   IF x[j] < smallest
   THEN
     smallest := x[j];
     index smallest := j
   FI.

glue this behind the sorted part:
   EL CONST set aside :: x[i+1];
   x[i+1] := smallest;
   x[index smallest] := set aside;
   i INCR 1.
```

How does the execution time of this algorithm depend on n, the number of elements? An interval of diminishing size has to be searched repeatedly, in total $n-1$ times, where the time to search it is practically proportional to the length of that interval. Therefore the time is of the order of $\sum_{i=1}^{n-1}(n-i)$ which for large n is proportional to n^2.

8.6.2 Example: Insertion Sort

Again the first part of the row is sorted, the second is unsorted. This time we repeatedly take the next unsorted element and insert it in the right place in the sorted part, according to the picture in Fig. 8.9.

```
insertion sort:
   INT VAR j;
   FOR j FROM 2 UPTO n
   REP
     insert jth element in the right place
   ENDREP.

insert jth element in the right place:
   determine the place where it belongs;
   put it in that place.
```

We determine the place by *linear search.* Later in this chapter we will describe a more intelligent way of searching.

```
determine the place where it belongs:
   INT VAR place :: 1;
   WHILE x[place] < x[j]
   REP
     place INCR 1
   ENDREP.
```

Observe that this repetition always terminates: in the worst case all indices from `1` to `j` are tried, but on the average half this range.

```
put it in that place:
   take the jth element;
   shift the others to the right;
   drop the jth element.

take the jth element:
   EL CONST element :: x[j].
```

Shifting the other elements to the right is necessary in order to make room for the element. The sorted part is enlarged by one element, overwriting the original `j`th element. Observe that the shifting has to proceed from higher to lower indices (what happens if we go the other way?).

```
shift the others to the right:
   INT VAR k;
   FOR k FROM j DOWNTO place + 1
   REP
     x[k] := x[k-1]
  ENDREP.
```

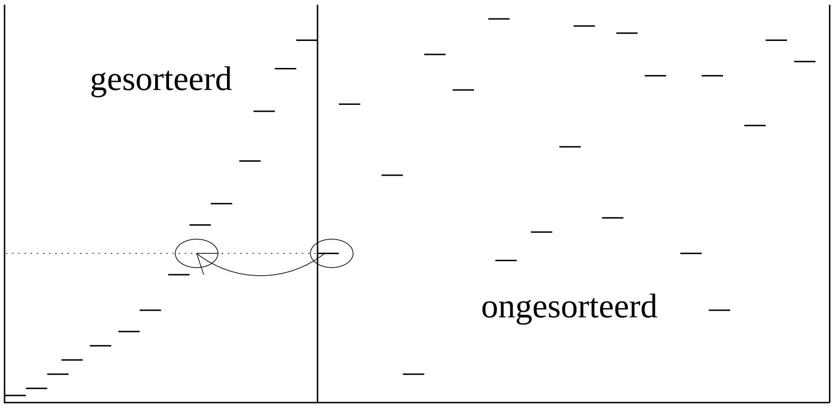


Figure 8.9: Insertion Sort

```
drop the jth element:
   x[place] := element.
```

8.6.3 Example: Bubble Sort

It is well known, at least for those who are familiar with combinatorics, that every permutation can be achieved by a number of permutations of 2 consecutive elements. (Permutation means a particular enumeration of all elements of a sequence.) This brings to mind the idea of going repeatedly through the row while sorting each consecutive pair. We do this until there is nothing left to do.

The algorithm thanks its name to the similarity that exists between the motion of the "lighter" elements of the row, while it is sorted, towards the end of the row and the upwards motion of the bubbles in a glass of soda water.

```
bubble sort:
   REP
    sweep through the row
   UNTIL nothing out of order
   ENDREP.

sweep through the row;
   INT VAR i;
   BOOL VAR exchange performed :: FALSE;
   FOR i FROM 1 UPTO n - 1
   REP
     may exchange pair
   ENDREP.

may exchange pair:
   IF   x[i] > x[i+1]
   THEN
     EL CONST set aside :: x[i];
     x[i] := x[i+1];
     x[i+1] := set aside;
     exchange performed := TRUE
  FI.

nothing out of order:
   NOT exchange performed.
```

If the row is "nearly" sorted the algorithm Bubble Sort will need only a small number of sweeps. Only in this special case it is noticeably more efficient than the two previous ones. (The word "nearly" needs some explanation. It does not mean what you would suppose at first. As an example, consider a row where the smallest element happens to be at the opposite end of the row. Now it does not matter how the other elements are sorted, the algorithm must execute each one step. Would you consider the row `2 3 4 5 1` "nearly" or "far from" sorted?)

An improvement follows from the observation that after a sweep the "greatest" element is guaranteed to have been moved up. For this reason the next sweep can be one shorter.

8.6.4 Should we really sort this way?

How does the *complexity* (in particular the execution time) of these three algorithms depend on n, the number of elements? To begin with, this of course depends on speed of the computer — a microcomputer may be a factor of 1000 slower than a "real" computer. For a comparable architecture of computers we can assume that they have to perform more or less the same actions. In general we can express the execution time as the number of actions times a factor indicating the machine speed. We abstract from this last factor by looking only at the dependence on n.

The execution time depends not only on the number but also to some degree on the values of the elements. We shall therefore examine how many actions are to be performed

- in the worst case,
- in the best case,
- in the average case (but what do we mean by "average"?)

as a function of n.

Finally the execution time also depends on the character of the actions to be performed and on details of the programming. For sufficiently large n all other actions can be neglected with respect to those repeated most often. All those aspects can be subsumed under the factor already mentioned.

In the worst and in the average case for the given algorithms the execution time depends quadratically on n, i.e. the execution time $= O(n^2)$, as argued for the algorithm Selection Sort. Sorting twice as many elements therefore takes four times as long!

As an example we take the sorting of a list of names with an $O(n^2)$ algorithm on some rather slow computer.

school class:	30 names,	1 sec
sports club:	300 names,	100 sec
Automobile Club:	300000 names,	10^8 sec
USA:	$300 * 10^6$ names,	10^{14} sec

Bear in mind that the times mentioned are given only for reasons of comparison.

The better algorithms, such as Quick Sort which we describe later in this book, are of the order $O(n \log n)$. For sufficiently large n they are arbitrarily faster. In spite of that, their sorting time still increases more than linearly with the number of elements. The administration of the citizens of the USA will certainly not be sorted anew every day, but a file of that size will be kept on background memory using special data structures.

8.7 Example: Binary search

To conclude this chapter we describe the *binary search method,* a classical halving algorithm for finding the index of a given element *el* in a row x. For the row $x_1, x_2, \ldots, x_n$, where $x_i \leq x_j$ and $1 \leq i < j \leq n$, we search an index k such that $x_{k-1} < el \leq x_k$. In other words: if there is an element equal to *el* in the row, then k is the smallest index of such an element, and if there is not such an element, k is the index that *el* would have if it were inserted in the right place of the row.

Consider an arbitrary element of x with index $t \in [1 : n]$ and compare it with *el.* Obviously one of the relations

$$x_t > el \quad \text{or} \quad x_t = el \quad \text{or} \quad x_t < el$$

must hold.

Looking at the value of the element with index t allows us to say something about the index k:

$$\begin{array}{lll} x_t \geq el & \rightarrow & k \in [1 : t] \\ x_t < el & \rightarrow & k \in [t+1 : n] \end{array}$$

Choosing the index t in the middle of the row allows us to eliminate half of the elements of the row at one time. We repeat this until we have only one element left.

```
binary search:
  start with the whole interval;
  REP
    choose an index t;
    IF x [t] >= el
    THEN
      continue in left half
    ELSE
     continue in right half
    FI
  UNTIL only one element left
  ENDREP.
```

```
start with the whole interval:
   INT VAR lower :: 1, upper :: n.
```

Observe that `lower` < `upper` because `1` < `n`. We choose `t` to be in the middle of the interval [`lower` : `upper`].

```
choose an index t:
  INT CONST t :: (lower  + upper) DIV 2.
```

For the termination of this algorithm we have to assure that both halves are smaller than the whole.

By the special meaning of `DIV` (division dropping the remainder) we can deduce `lower` < `upper` → `lower` ≤ `t` < `upper`.

```
continue in left half:
  upper := t.
```

In this case the interval gets smaller because `t` < `upper`.

```
continue in right half:
  lower := t + 1.
```

In this case also the interval gets smaller, because `t` ≥ `lower`, so `t+1` > `lower`.

All what we still have to do is to give the terminating condition. Since the part yet to be sorted is between `lower` and `upper` we write:

```
only one element left:
  lower = upper.
```

Observe that in this way a sorted row of 2^k elements is searched in k steps, and in general n elements in about $\log n$ steps. The algorithm is of the order $\log n$.

A row of one million elements is searched in about 20 steps — much faster than linear search. We can use this algorithm to speed up the algorithm Insertion Sort, although the effect is not dramatic because the insertion of the element of the right place remains quadratic and takes most of the time.

The above algorithm works well until the required element, `el`, is not greater than the greatest element of the sorted row. If greater this algorithm still delivers the index of the greatest element of the row instead of the index that is larger by one. There is a number of solutions. We can, for example, check at the beginning whether the required element is greater than the greatest element and stop if it is. We get a more elegant — and shorter — algorithm if we use a *guard.* In our case the guard will be the `(n+1)`th element of the row and its value must be the greatest possible value. (In case of a row with `INT`-elements, for example, `maxint` would be an appropriate guard.) Now, when the required element is greater than the greatest element of the (original) row the modified algorithm delivers the index of the guard.

8.8 Exercises

1. (Shaker Sort) A variant of Bubble Sort is obtained by alternating the directions of the sweep so that alternately the next largest element is moved to the right and the next smallest element is moved to the left. The unsorted area is enclosed between two growing sorted areas, so that the sweeps get shorter and shorter.

2. (Symmetrical binary search) In the given formulation of the binary search algorithm we have exploited only two of the three possibilities (because we did not distinguish the case that `t` happens to be the index of the right element). As a consequence the algorithm is not completely symmetrical in dealing with the left and right half. Write a variant of the algorithm that does exploit all three possibilities and is symmetrical. Make use of the `LEAVE`-instruction.

3. (Comparison of sorting methods) Make an experimental comparison between the sorting methods given (and possibly others). Distinguish between the best, average and worst case. Try to explain the differences found.

4. (Calendar) Write a program that can print a calendar for any year between 1901 and 2100. Make use of the knowledge that the first of January 1901 was a Tuesday. The calendar should make an appetizing impression, which means that the layout of the output is very important. (Hint: it may be simpler to build up the calendar in a row of texts before printing it.)

5. Write a program that reads in a text of several lines, computes a frequency table of the letters occurring in that text and then displays this table in the form of a bar diagram with vertical bars. Choose the vertical scale such that the longest bar fits exactly on the screen.

6. (Text formatter) Write a program that reads a text (consisting of words with punctuation marks and layout, a number of lines and ending on a dollar sign) and prints it out in a minimum number of justified lines. A line is called justified if (like most lines in this book) the first word starts on the leftmost position of the line and the last word ends at the rightmost position of the line. The words are separated by one or more spaces and the spaces are distributed as evenly as possible between the words on the line. The last line need not be right-justified. A line consisting of one overly long word can also not be justified.

7. (Game of life) Take a big board with squares and an appropriate number of stones (e.g. a GO-board with its stones). Put some stones as the first generation on the board. Each stone may have up to eight neighbours. Apply the following rules (first described by Conway in 1970) to all stones of the board configuration to determine the next configuration:

   - a stone with two or three neighbours survives,
   - a stone with less than two neighbours dies in loneliness,
   - a stone with more than three neighbours dies in overpopulation,
   - a new stone will born if an empty field is surrounded by precisely three neighbours.

   Establish the initial configuration by means of some display. Show the sequence of configurations on the screen.

Chapter 9

Files

Computers owe their fast development not so much to the fact that they can calculate as to the fact that they provide an economic solution for important problems of administration. It is the electronic data processing that brings in the big money. The pure calculating applications, even for the military, would not have justified the enormous investments over the last thirty years. The storage and processing of large amounts of data needs other techniques than we have described so far.

Consider for example the administration of spare parts in a large production plant. For thousands of parts, a certain amount of information has to be stored, such as the name of the article, the part number, the number in stock and the retail price. The collection and the processing of the data generally takes place by different programs at different times, while over a number of terminals various programs must act in sequence or even simultaneously on the same collection of data.

This demands a quite different form of data storage than we have met until now. We shall not describe at this place the physical storage methods and the consequent techniques for administration and programming that have been developed. We only give a number of essential concepts and some simple programming means by which a model can be made of such a large administration.

9.1 Some concepts

We introduce only some of the most general concepts, and do not concern ourselves with the detailed terminology that has arisen in this branch of Informatics over the years.

A *file* is a stored collection of data that has a *name*. The concept of file allows us to abstract from the physical properties of specific storage means (such as in the past punched card or punch tape, nowadays magnetic tape, magnetic drum or disk and shortly also the compact disk).

The data are stored in the form of *records* that each contain a line of text. Those records we consider as split up in some way into a number of *fields* each containing a number or text (for a spare part: the item's part number, number in stock and retail price followed by a text of at most forty characters, the name). Of course different programs can look upon the records of one same file in different ways, but this has to be done with great circumspection. A record that contains 3 numbers can also be looked upon as a record containing one text, but then it may be rather more difficult to extract the 3 numbers.

By means of input procedures, the fields of a record are read as values. Similarly, values can be written to a file by means of output procedures. Files can be kept or deleted, they can obtain another name, and their contents can be modified repeatedly.

9.2 File operations in Elan-0

Elan has a concrete type of file (`FILE`) with a complete set of operations that are representative of typical electronic data processing. We introduce here instead a somewhat simpler set of operations that are adequate for an introduction to file handling and for understanding the most essential aspects. These operations are the only ones available in Elan-0, and they can easily be expressed by the language means of the full language. For a description of the standard packet for file management we refer to the Elan language definition or to chapter 11 of [HOM83].

9.2.1 Opening and closing files

At any moment the computer knows of a number of files. Every file has a text as its name. In order to access a file from a program we first have to *open* it, indicating what kind of file we need (new or old, for reading or writing) and what its name is. Finally we must make the file available to other programs by *closing* it.

In simple Elan-0 systems, only one file at a time can be open. This is a severe restriction compared to the practical situation where quite a large number of files can be open simultaneously. This restriction is motivated by the desire to use Elan-0 also on microcomputers that have only a cassette recorder for file storage. A consequence of this restriction is that algorithms have to be oriented more towards internal data structures (rows) than to data structures on files. Such file structures are interesting but will not be dealt with until the second volume of this book.

For file handling we have a number of concrete algorithms, of which we shall indicate the name and the parameters by means of a *procedure-heading.* We hope that the notation used here to give particulars about these procedures will be self-evident although procedures are described in chapter 10.

a) `PROC new file (TEXT CONST name):`

   An empty file with the name `name` is opened for writing, which means that we can write on it but can not read from it. Writing starts at the first record. If there already exists a file with that name, it is erased (beware!).

b) `PROC old file (TEXT CONST name):`

   If there already exists a file with the name `name` then this is opened for reading, starting at the first record. If such a file does not exist, the execution of the program is halted.

c) `PROC close file:`

   Any file that was open is closed. After being closed, a file can be opened by one of the two preceding procedures. The closed file is kept until it is erased.

d) `PROC erase file (TEXT CONST name):`

   If there already exists a file with the name `name`, then this is erased, otherwise nothing happens. In this way we can limit the number of files that the computer has to keep.

If you have opened a file and forget to close it then at the end of the program it is closed automatically. (But better when a programmer is not absent-minded.)

9.2.2 Writing to a file

Writing is possible only to a file that has been opened for writing. The following algorithms write on the current line of the file starting at the current position.

e) `PROC write (INT CONST x):`

   A denotation of the value of `x` is written, possibly preceded by a negative sign.

f) `PROC write (TEXT CONST x):`

   The characters of the value of `x` are written one by one. As with the procedure `put`, no quotes are given surrounding this denotation, which leads to a nice layout but may make reading back somewhat complicated.

g) `PROC writeline:`

   After a call of `writeline`, writing continues at the beginning of the next record of the output file.

Many machines have a special file name for the printer, for example, `prn`. It is the name of a file that can only be written to. Writing to this file causes the output to appear on the printing device of the computer (if it has one).

9.2.3 Reading from a file

Reading is possible only from a file that has been opened for reading. In reading from a file, it is possible to read back in the form of a text something that has been written as a number and vice versa, so that some care is necessary. Also it is not allowed to read beyond the end of the file; in trying to read beyond the end of file, the execution of the program is stopped.

h) `PROC read (INT VAR x):`

   An integer denotation with eventual preceding sign and layout (new lines, spaces) is read. Its value is assigned to `x`.

i) `PROC read (TEXT VAR x):`

   The rest of the record is read and assigned as a text to `x`. After that, reading continues with the next record of the file. There is no separate procedure for going to the next input record.

j) `BOOL PROC file ended:`

   Answers the question whether the end of the input file has been reached. No other convention to test for the end of file is needed, provided this test is invoked in time.

Observe that a record in general contains zero or more numbers followed by a text. If we reverse this order (have a text followed by a number) then, in reading, this number is assumed to be part of the text, so that it cannot be read separately, which may lead to unforeseen behaviour of the program.

9.3 Editing

Besides the reading of data, its modification occurs so often in all kinds of information systems that we shall introduce for that purpose a special operation that can collaborate very well with the file manipulation operations. It is somewhat complicated:

k) `PROC edit (TEXT VAR t, INT CONST p):`

   The first parameter of `edit` is a text variable; its value is displayed on the screen and then the opportunity is offered for modifying the text on the screen. Finally the thus-modified text is assigned to the actual parameter corresponding with `t`.

   The constant `p` (with $1 \leq \texttt{p} \leq \texttt{LENGTH t}$) gives a threshold position. The cursor cannot be moved before it. If one chooses $\texttt{p} = 1$, the whole text can be modified.

With the help of `edit` it is simple to make information systems such as the following example.

9.4 Example: Keeping an address list

I have an address list containing the names, addresses and phone numbers of a growing number of people.

Now and then I add a person or change his address or phone number. In the course of time, these modifications make such a muddle of my address list that I take an evening out for copying the address list into a fresh book, with the names alphabetically ordered and without insertions.

We shall write a program to administer an address list. We assume that the address list is written on a file and that we may modify it by the following program.

9.4.1 User interface

We shall first plan the behaviour of the program from the point of view of the user and the commands to be given by the user. It is highly important to design this user interface well, so that the user can easily use the system, has a good mental picture of its workings and cannot be suddenly confronted with surprises.

9.4.1.1 Phases

In the execution of the program we distinguish three phases. At the beginning of the program, the current address list is read from its file. During the second phase of the execution of the program, a copy of the whole address list is in memory, and we have the opportunity to make all manner of modifications. Finally the thus-modified address list is written out as a new file. Since this new file has the same name as the old one, the old file thereby becomes inaccessible.

The modifications are not made directly to the file. As a consequence, the old address list remains available in unmodified form for the duration of the program, so that it is possible to stop the execution of the program prematurely before the new address list is written to a file.

9.4.1.2 The information to be stored

We use a very simple file structure: three records per person, containing their name, the first part of their address and the second part of their address. The detailed contents of those records we choose not to specify. The second line of the address might contain, for example, the phone number, but the precise contents of the records do not concern us.

9.4.1.3 Commands

The user interface is modelled somewhat on that of the Elan-0 programming environment. The commands consist of one single letter, followed by pressing the RETURN-key and possibly some further information. After the file has been read initially, the following commands can be given:

- `a` adding a person. Should be followed by lines containing the name, adr1 and adr2. If a person with that name is already known, then his adr1 and adr2 fields are updated, otherwise the new person is entered at the right place in the alphabetic list. Notice we cannot have more than one acquaintance with the same name.
- `d` deleting a person. Demands a known name. If a person with that name occurs in the list, this person is removed, otherwise nothing happens.
- `s` showing a person. Demands a known name. If a person with that name is known then its entries are shown, otherwise nothing happens.
- `e` modifying entries. Demands a known name, looks up the corresponding adr1 and adr2 and gives the opportunity to modify these. If the name given is unknown nothing happens.
- `l` listing the names. Gives on the screen an alphabetic list of the known names.
- `q` quitting. The dialogue is ended and the program goes over to the third phase, in which the resulting alphabetic address list is written, overwriting the input file of the same name.

9.4.2 Program

The three phases of execution can be found back in the main program.

```
program:
   define all commands;
   read old address list;
   show all names;
   REP
     ask for command;
     execute command
   UNTIL command = quit command
   ENDREP;
   write new address list.
```

The main program is a simple interpreter for the commands.

```
define all commands:
   TEXT CONST add command :: "a";
   TEXT CONST delete command :: "d";
   TEXT CONST show command :: "s";
   TEXT CONST edit command :: "e";
   TEXT CONST list command :: "l";
   TEXT CONST quit command :: "q".
```

The various commands are introduced here as text constants, making it simple to choose a different user interface.

9.4.2.1 Reading the old file

We read the old file by means of the operations introduced in this chapter and store their records in the form of three rows.

```
read old address list:
   LET list max = 100;
   INT VAR max index :: 0;
   ROW list max TEXT VAR namerow;
   ROW list max TEXT VAR adr1row;
   ROW list max TEXT VAR adr2row;
   open old address list;
   REP
     read person;
     insert person
   UNTIL file ended
   ENDREP;
   close file.
```

By modifying the value of `list max` we can try to cope with a larger address list, but this can only succeed if our computer has sufficient memory.

```
open old address list:
   ask address list name;
   old file (file);
   read next nonempty line.
```

In order to keep the program flexible and independent of the underlying operating system we prompt the user for a file name.

```
ask address list name:
   TEXT VAR file;
   put ("Name of address list, please: ");
   get(file).
```

We must be careful in reading the lines from the file. On the one hand, if we tried to read after the end of the file we would get an error message and our program would be stopped. On the other hand, we may wish to insert empty lines in our address list to enhance readability, but we do not want these empty lines to be taken as names or addresses, so we have to skip them. This is again a nice example for the interplay of the `WHILE`- and the `UNTIL`-part.

```
read next nonempty line:
   TEXT VAR textline :: "";
   WHILE NOT file ended
   REP
     read (textline)
   UNTIL textline <> ""
   ENDREP.
```

After so much preparation we can read the old address list.

```
read person:
   read name;
   read address1;
   read address2.
```

The entries for one person consist of three records that we read separately. Notice that we always read the next line in advance. This is necessary if we want to detect the end of file condition while skipping the empty lines. It is a variant of the often used *look-ahead* technique.

```
read name:
   TEXT VAR person name :: textline;
   read next nonempty line.

read address1:
   TEXT VAR address1 :: textline;
   read next nonempty line.

read address2:
   TEXT VAR address2 :: textline;
   read next nonempty line.
```

9.4.2.2 Adding a person

We keep the list of names sorted in memory. Therefore we cannot just append a new person at the end of the list but have to insert it somewhere in the list (cf. Insertion Sort, Sec. 8.6.2). First we compute its index, i.e. the position in the name table that either contains a person with that name or is where a person with that name should fit according to the alphabetic-lexicographic ordering.

```
insert person:
   find index;
   IF already known
   THEN
     overwrite
   ELSE
     shift up from index;
     overwrite
   FI.

find index:
   INT VAR index :: 1;
   WHILE index <= max index
   REP
     IF namerow [index] >= person name
     THEN
       LEAVE find index
     FI;
     index INCR 1
   ENDREP.
```

We use here the simplest form of linear search. To increase efficiency, the reader might prefer a form of binary search (see Sec. 8.7).

```
already known:
   IF index > max index
   THEN
     false
   ELSE
     namerow [index] = person name
   FI.
```

This condition is rather “tricky”, because a person might have to go after the last element of the list.

```
overwrite:
   namerow [index] := person name;
   adr1row [index] := address1;
   adr2row [index] := address2.
```

```
shift up from index:
   IF max index = list max
   THEN
     address list full
   ELSE
     INT VAR i;
     FOR i FROM max index DOWNTO index
     REP
       shift one up
     ENDREP
   FI;
   max index INCR 1.

shift one up:
   namerow [i+1] := namerow [i];
   adr1row [i+1] := adr1row [i];
   adr2row [i+1] := adr2row [i].

address list full:
   line;
   put ("Address list full ...");
   line;
   LEAVE program.
```

The algorithm presented here is of poor efficiency. Suppose we have n entries and want to add m more entries. Then this algorithm will execute `shift one up` in the order of $n*m$-times. A better solution would be to first store the new entries in separate rows and then, when the program writes the entries into a file, merge the data from the different rows. Of course, if we have only 100 entries it makes no difference but, again, with large amounts of data we must be careful.

9.4.2.3 Processing a command

```
ask for command:
   TEXT VAR command;
   REP
     line;
     put ("Command = ");
     TEXT VAR comline;
     get (comline);
     command := comline SUB 1
   UNTIL command >= "a" AND command <= "z"
   ENDREP.

execute command:
   IF command = add command
   THEN
     ask person;
     insert person
   ELIF command = delete command
   THEN
     ask known name;
     remove person
   ELIF command = show command
   THEN
     ask known name;
     show person
   ELIF command = edit command
   THEN
     ask known name;
     edit address1;
     edit address2
   ELIF command = list command
   THEN
     show all names
   ELIF command = quit command
   THEN
   ELSE
     line;
     put ("Admissible commands: a d e l s
q");
     line
   FI.
```

9.4.2.4 Asking for information

We shall use names like "`ask...`" for those algorithms that read information interactively (i.e. not from file, but via keyboard and screen directly from the user of the program).

```
ask person:
   ask name;
   ask address1;
   ask address2.
```

Remember that the `get` algorithm with a single text parameter behaves differently under the various Elan implementations. Therefore, it is better if we hide the call of `get` into one refinement thus making the modification easier. Here is the Elan-0 version:

```
nonempty textline:
   get (textline);
   textline.
```

And here is one of the possible Elan-1 versions:

```
nonempty textline:
   get (textline,80);
   textline.

ask name:
   line;
   put ("Name:     ");
   person name := nonempty textline.

ask address1:
   line;
   put ("Address1: ");
   address1 := nonempty textline.

ask address2:
   line;
   put ("Address2: ");
   address2 := nonempty textline.
```

When the requested name is not found we shall kindly warn the user: `Unknown`.

```
ask known name:
   ask name;
   find index;
   IF NOT already known
   THEN
     line;
     put ("Unknown.");
     line;
     LEAVE execute command
   FI.
```

9.4.2.5 Modifying entries

For modifying entries we make use of the procedure `edit`.

```
edit address1:
   line;
   put ("Address1: ");
   edit (adr1row [index], 1).

edit address2:
   line;
   put ("Address2: ");
   edit (adr2row [index], 1).
```

We might similarly permit also the name of a person to be modified, but then we must take care to delete the old person and insert the new person.

9.4.2.6 Removing persons

In order to remove a person the `execute command` algorithm asks for a known name. If, for any reason, our answer is wrong the `ask known name` refinement gives a warning message (`Unknown.`) and prematurely terminates the execution of the command. On the other hand, if the name is known `index` will contain the proper value and the entry can be deleted. Now only the prompt will indicate that the command has been executed.

```
remove person:
   shift down until index.

shift down until index:
   INT VAR j;
   FOR j FROM index UPTO max index - 1
   REP
     shift one down
   ENDREP;
   max index DECR 1.

shift one down:
   namerow [j] := namerow [j + 1];
   adr1row [j] := adr1row [j + 1];
   adr2row [j] := adr2row [j + 1].
```

9.4.2.7 Showing entries

```
show all names:
   FOR index FROM 1 UPTO max index
   REP
     show name
   ENDREP.

show name:
   line;
   tab;
   put (namerow [index]).

show person:
   line;
   tab;
   put (namerow [index]);
   line;
   tab;
   put (adr1row [index]);
   line;
   tab;
   put (adr2row [index]).

tab:
   put (8 * " ").
```

9.4.2.8 Writing the new file

Before we write the address list into a file we kindly ask the user to name the output file: the old file may be rewritten or a new file may be created. The `write end of file mark` refinement is necessary if we want to read in the same file later. Most Elan implementations makes the `file ended` condition `true` when the operating system indicates that the file, opened for reading, has reached its end, others need an explicit mark. It is safe if we always append this mark to the end of the file.

```
write new address list:
   ask address list name;
   new file (file);
   write all persons;
   write end of file mark;
   close file.

write all persons:
   FOR index FROM 1 UPTO max index
   REP
     write person
   ENDREP.

write person:
   write (namerow [index]);
   writeline;
   write (adr1row [index]);
   writeline;
   write (adr2row [index]);
   writeline.

write end of file mark:
   write (ascii (4)).
```

9.4.3 Our first address list

We are proud of having developed such a beautiful program, we laboriously type it in — and then suddenly realize that it has one curious property: it presupposes that the file we want to modify is already present. This file must exist before the execution of the program and may not be empty. Where do we get it from?

The simplest way is to make another program that writes an initial file with at least one person. Once we have got that file, the initializing program is no longer necessary. We choose one of the people we want to have in the list, and write the following program:

```
start data base:
  ask address list name;
  new file (file);
  write ("Aardvark, Anthony A.");
  writeline;
  write ("17, Hampstead Road");
  writeline;
  write ("Weston-Super-Mare");
  writeline;
  write end of file mark;
  closefile.
```

It would have been simpler to make an empty file (we would have needed only the first and the last unit), but the program we have developed needs at least one person.

9.4.4 The use of the program

We start the program and after some time see the *prompt* appear

```
Command:
```

Immediately behind this we type an `a` and finish the line with the RETURN-key. A new prompt appears

```
Name:
```

whereupon we type in

```
Cowznowsky, Melvin S.
```

Now we are prompted for the first part of the address. We enter also the two address fields and again get the prompt

```
Command:
```

Curious to see the result of our work, we give an `l` and see

```
Aardvark, Anthony A.
Cowznowsky, Melvin S.
```

By means of a number of `a` commands we introduce the other persons and give again an `l` whereupon we obtain an alphabetic list of all persons.

Now Anthony Aardvark comes in and asks what data we have stored about him. We give an `s` followed by

```
Aardvark, Anthony A.
```

(It is rather long. Maybe we had better find a convention for searching for abbreviated names, such as: `Aard*`.) and see his entries appear. We give a `q`, whereupon the new address list is written away and the program is finished.

We shall not describe the use of the other commands; you should rather try them yourself.

The program given is rather long, but, it is hoped, quite comprehensible. It is one of the largest Top-Down programs appearing in this book. All kinds of information systems with a similar structure can be modelled after it.

9.5 Exercises

1. (Flexible searching) Modify the `ask known name` refinement in the address list program allowing
   - abbreviated names, such as `Aard*`,
   - case-insensitive searching, such as `aardVARK, anTHONy a.`,
   - meaningless punctuation characters, such as `Aardvark Anthony A`,
   - combinations of the above cases.
2. (Membership administration) Are you a member of some club? Try to imagine how its administration of members should be organized. Per member the following entries are of importance: address, membership number, last year paid, and other entries that are dependent on the nature of the club. Take care that at least the following lists can be obtained:
   - complete list of members in alphabetic order;
   - same ordered by number, with addresses, in the form of address labels;
   - list of tardy payers.
3. (School administration) Design an administration for a school, that captures
   - for every pupil the courses and results;
   - for every class the pupils.

   What kind of modifications to this file must be possible? What questions should be asked of it?
4. (Giro system) Design and realize a simple banking system that keeps the name, address, account number and balance of the clients, takes care of deposits, withdrawals and transfers, sends daily statements and periodically computes interest.
5. (Booking system) Design and realize a simple booking system for an airline, that per plane administers the customers and the empty places and can make booking for (groups of) customers. This system can be made as realistic as you wish by the addition of all kinds of aspects (like smoking/non-smoking, waiting list, alternative routing, different time zones ...).

Chapter 10

Procedures

As the means of defining abstract algorithms, we have until now used refinements. This mechanism is of major importance in learning a systematic way of programming, but in most programming languages it is not available in its pure form. Instead, these languages have, as a means of abstraction for algorithms, the *procedure* ("subroutine", "subprogram", "function"). Elan distinguishes both refinements and procedures, each associated with a specific programming style.

Just as for a refinement, a procedure is a means of giving a name to a piece of program text, so that that name can be used in place of that piece of text. The purpose of this mechanism is, to begin with, simplification of programming by shortening the program text and reduction of the opportunity to introduce errors in repeating pieces of text. Both mechanisms can be used as a means of abstraction for algorithms in Top-Down programming. Procedures additionally have a possibility of communicating with their environment through *parameters* given at their call. Therefore they lend themselves also to a different programming style than refinements.

From this chapter on, we leave the sublanguage Elan-0, which is intended purely for Top-Down programming. Although in this book we do not yet introduce the Bottom-Up programming style, the following chapters are a preparation for it.

10.1 Refinements: a look back

Human beings have great trouble in keeping an overview of complicated matters. In order to compensate for this human weakness, we do not try to solve difficult problems in one stroke, but are satisfied as a first approximation to take the problem apart into smaller parts, each of which seems to be easier to solve. Then we repeat this decomposition process on each of the subtasks until we arrive at tasks that are small enough to be resolved immediately.

We use the same technique in programming: we want to construct an abstract algorithm that precisely solves our problem. To that end we imagine a number of suitable abstract algorithms to solve parts of the problem, which we then stick together by means of control structures. After that, we have to realize each of those abstract algorithms in terms of (other) abstract algorithms and, finally, the concrete algorithms of the programming language used.

For defining those abstract algorithms that arise as a fleeting intermediate stage in this process we make use of refinements.

10.1.1 Example: description of a manual task

We want to construct an algorithm that describes how to fix a hole in the front tube of a bicycle. We assume that we have a simple but highly specialized processor that can manipulate parts of bicycles. The example is realistic to the extent that it can be seen as a program for a kind of robot.

At the highest level of abstraction we consider the bicycle as consisting of a few large parts, such as the wheels. Those are the objects that we can talk about. At this level we can indicate what we mean by `fix front tube of bicycle`.

```
fix front tube of bicycle:
   turn the bicycle about;
   remove the front wheel;
   fix the front tube;
   attach the front wheel;
   turn the bicycle about.
```

This is a complete algorithm, albeit as yet not very detailed. It is so far not incorrect, in that it does not say anything wrong as yet. The "only" thing we have to do now is to realize the abstract algorithms mentioned.

We call such a formulation, at the highest level of abstraction, the *rough formulation* of the algorithm. It is important that this rough formulation captures the essence of the solution, otherwise nothing has been achieved. We do however have the freedom to introduce useful abstract algorithms at will, even if they do not belong to the concrete repertoire of one or other processor. Afterwards we shall make them concrete by means of refinements.

Consider for example the abstract algorithm with the suggestive name `remove the front wheel`. The name more or less adequately expresses *what* has to be done. In the refinement we now have to indicate *how* that must be done. This realization we express in terms of a lower level of abstraction, where we conceive of the front wheel as in its turn composed of a hub with nuts, spokes, nave, inner and outer tube.

```
remove the front wheel:
    remove the left front wheel nut;
    remove the right front wheel nut;
    take the front wheel from the fork.

attach the front wheel:
    fit the front wheel into the fork;
    attach the right front wheel nut;
    attach the left front wheel nut.
```

Again we abstracted from a number of things. We have for instance not mentioned at all where to keep the nuts safely. The decision about that is part of the refinement of `remove the left front wheel nut`.

A decision of detail that we have already made in the realization given concerns the question of in which order the nuts have to be loosened and tightened. We indicated that first the left and only afterwards the right front wheel nut had to be removed. Of course the order might just as well have been the other way around, because the result does not depend on it. But this decision has a consequence for other details. We might for instance deduce that the processor between the removal of the nuts and their tightening is at the right side of the bicycle. The decision "first left, then right" is an *overspecification.* In order to keep generality as great as possible, we should have specified collateral removal, so that some kind of octopus might even have removed both nuts simultaneously. Since we restrict ourselves to sequential algorithms we have no notation to describe collateral execution. Overspecifications like this one occur in programming all the time: time and again we see ourselves forced to make a choice that excludes whole classes of just as good or possibly better realizations.

We might have delayed the decision another refinement step by introducing an abstract algorithm `remove both nuts`; but that would merely have delayed the issue.

In programming, one has to be aware all the time of such phenomena, and must take care to make concrete choices for a specific strategy as late as possible, in order to minimize remorse about hasty decisions. On the other hand, when taking a decision can no longer be avoided, we should cut through the knot courageously and accept the consequences of our decision. It is advisable in this process to define those abstract algorithms that belong to the same level of abstraction first before embarking on the next, lower, level of abstraction.

We continue refining:

```
fix the front tube:
    loosen the outer tube;
    take the inner tube out;
    fix the inner tube;
    stuff the inner tube in;
    mount the outer tube.
```

In fixing the inner tube we have to take care of yet more details, so that now the hole in the tube also belongs to our level of abstraction. It may be that there is more than one hole, or that we have made an error and cannot find any hole at all.

```
fix the inner tube:
    WHILE there is another hole
    REP fix that hole
    ENDREP.
```

And so we must continue refining until we reach (we hope) the level of directly executable concrete algorithms.

The method that we have applied here is known as the *Top-Down method,* hierarchical decomposition with the aid of refinements.

By the systematic use of refinements with well-chosen names we strive to retain a large part of the passing thoughts, ideas and insights that play a role in the design process. In a later modification of the program the meaning of the maker need not be reconstructed painfully from his deeds (see [MEE77], [DAH72]).

Refinements serve to support the programming process, but they do not perform wonders. They serve to retain the fleeting thoughts arising during the programming process. They give a certain rhythm to a program. They reduce the distance in abstraction that has to be bridged at one time. They restrict the horizon that the programmer has to oversee at one time. They allow him to distinguish between the "what" and the "how". They allow another person at a later time to follow the kinky thoughts of the author of a program.

Once again — they serve as a means for capturing thoughts. But they cannot suggest the thoughts.

10.2 Procedures as building blocks

Top-Down programming, in principle, proceeds until the level of the concrete expression means of the programming language has been reached.

In many cases, and especially for larger programs, this level is definitely not the most convenient one to end at. Much work can be saved by defining beforehand a few well-chosen elementary algorithms and objects, intended to serve as the lowest level of detail for this particular program.

We shall introduce for this purpose another form of abstract algorithm, the *procedure,* that we use whenever we feel the wish to define an algorithm that is to be used in more than one place in the program.

In that case we no longer consider it as an intermediate stage in the thinking process but as a supplement to the concrete algorithms of the language, as a new elementary building block. As an example, turning the bicycle around, manipulating the left and right nuts of the front and rear wheel, and also fixing of a hole obviously belong to the basic capabilities of anyone who is to repair his bicycle.

In somewhat larger programming exercises, one proceeds by first carefully defining a packet of elementary algorithms, objects and types in terms of which the solution of the problem can be expressed more conveniently, rather than by attempting to close the whole distance between the given problem and the given concrete language in one great step.

The consequent application of this idea leads to another programming style, *Bottom-Up programming* ("The Method of stepwise Synthesis") (e.g. [KLE81]), for which Elan has a number of special constructions (procedure declarations, type declarations, operator declarations, packet mechanism with interfaces). We shall go deeper into this subject in volume 2.

Of course there is a spectrum of possibilities between Top-Down and Bottom-Up programming style, in which procedures play an important role.

Those abstract algorithms that are worthy to be used more than once as elementary algorithms in the program, we shall realize as *procedures.*

Abstract algorithms that play only a unique, passing role in the design process of the algorithm we realize by means of *refinements.*

As a matter of fact in the previous chapters we have already silently made our acquaintance with procedures, because many of the concrete algorithms in Elan are procedures. Take for example the standard procedures `put`, `sin` and `cos`. In this chapter we shall indicate how to declare procedures, how to use them and how to classify them into various sorts.

10.3 Procedure declarations

Just like other objects (variables and constants), procedures have to be declared before they can be used. Such a declaration looks as given in Fig. 10.1.

An example of a procedure-declaration is:

```
REAL PROC average (REAL CONST a, b):
          (a + b) / 2.0
ENDPROC average
```

This declaration introduces a function of two arguments with the name `average`. (We usually say that operators have operands, functions have arguments and procedures have parameters. Their role is essentially the same.) Each of its parts (heading, body and tail) stands here on a line of its own. The first line is the heading and the second one the body.

The execution of a procedure declaration has as effect that a *routine* (a value which serves as the internal representation for the procedure) is bound to the name of the procedure. As you see, a procedure is again an object with a name and a value. The scope of a procedure declaration is determined in the same way as for other declarations.

10.3.1 The procedure-heading

In the heading of a procedure the correspondence is indicated between the environment where the procedure is executed and the body of the procedure. This correspondence is effected, as it will be described later, by binding values and variables from the environment *(actual parameters)* to names that are known only within the procedure *(formal parameters).* As an example, in the call `sin (0.5 * x)` the expression `0.5 * x` is the actual parameter of `sin`. The name of the corresponding formal parameter we have to look up in the declaration for `sin`. The syntax of the procedure-head is shown in Fig. 10.2.

The formal-parameter-pack gives declarations for the *formal parameters,* objects for which, in calling the procedure, a value will be given (Fig. 10.3. The pack of formal parameters is omitted if the procedure has no parameters.

A procedure may deliver as its result a value of the type indicated in its head. The type of the eventual result of the procedure can be deduced from the type declarer that precedes the keyword `PROC`. There are also procedures that have no value and only an effect. In the last case the type-declarer is empty.

The result of the procedure in our example is of type `REAL`.

The list of formal parameters `REAL CONST a, b` defines two real constants `a` and `b` for use in the body of `average`.

In the head of the procedure we find the names of all formal parameters with their type and access. In calling the procedure, for every formal parameter a corresponding actual parameter has to be supplied, i.e. an object or expression whose value will be bound to the formal parameter. The formal-parameter-pack indicates the number, types and order of the parameters demanded and gives them a name for use in the body of the procedure.

From the syntax diagram given above we omitted the possibility of passing a procedure as a parameter, which will be described in the next book.

10.3.2 Examples of headings

The extent to which a declared procedure is understandable depends on the procedure-name chosen, the names of the formal parameters and their types. Through the use of short catchy names, readability can be improved. A well chosen procedure-head indicates clearly *what* the procedure does without telling *how* it does it.

In order for this story not to get too dry, we shall first give a number of examples of procedure-headings.

```
REAL PROC sin (REAL CONST x):

REAL PROC max (REAL CONST a, b):

BOOL PROC even (INT CONST n):

TEXT PROC multiply (INT CONST n, TEXT
CONST s):

     PROC put (INT CONST x):

     PROC get (INT VAR x):
```

Anticipating the possibility of declaring abstract types, we will also give some examples of procedure-headings with formal parameters of various abstract types.

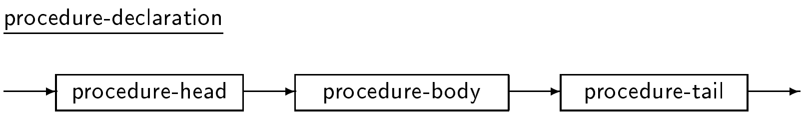


Figure 10.1: Procedure declaration

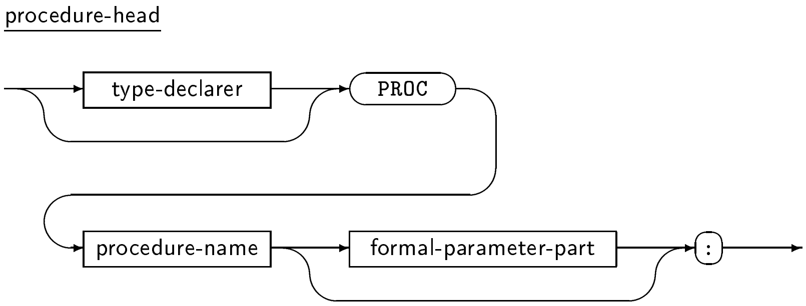


Figure 10.2: Procedure head

These examples illustrate well the usefulness of meaningful names: such names suggest the reader of the program text what is to be expected without telling him how it can be achieved.

```
REAL PROC max value (FUNCTION CONST f,
INTERVAL CONST i):

     PROC invert (MATRIX VAR a):

BOOL PROC are equal (ELEMENT CONST a, b):

BOOL PROC is in (SET CONST m, ELEMENT
CONST x):

     PROC perform transfer (ACCOUNT VAR
debtor, creditor,
                            REAL CONST
sum):

     PROC add to (ELEMENT CONST x,
COLLECTION VAR m):

     PROC signal (TEXT CONST message):

     PROC stop:
```

In all cases the head of procedure can easily be recognized by the magic word PROC.

10.3.3 The procedure-body

The body of a procedure consists of a paragraph which may be followed by some refinements (Fig. 10.4). The execution of the body consists of the execution of its paragraph.

In the previous example the body of the procedure is very simple: the paragraph consists of one single unit. The body may also contain one or more refinements as in the (rather contrived) example

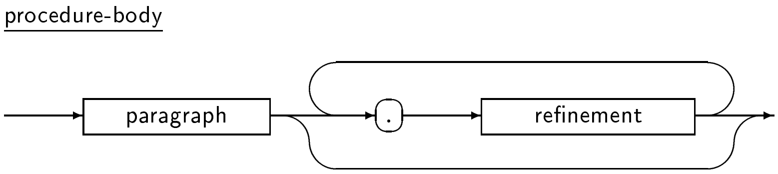


Figure 10.4: Procedure body

```
REAL PROC average (REAL CONST a, b):
   sum of values / sum of weights.
   sum of values:
      a + b.
   sum of weights:
      1.0 + 1.0.
ENDPROC average;
```

10.3.4 The procedure-tail

The name of the procedure may be repeated in its tail, a redundancy that makes it simpler to signal structural errors (such as lost ENDPROCs) adequately. The delimiter ENDPROC may also be written as END followed by PROC (Fig. 10.5).

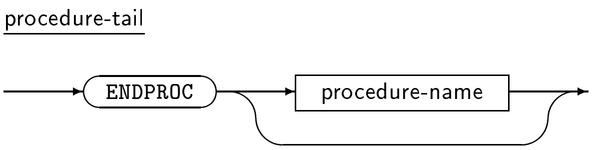


Figure 10.5: Procedure tail

In the examples, we will usually give a semicolon after the procedure-tail, but strictly speaking this does not belong to the procedure declaration: it acts as a separator from the next declaration.

10.3.5 The procedure-call

Once a procedure has been declared, it can be called from different places of the program. In a procedure-

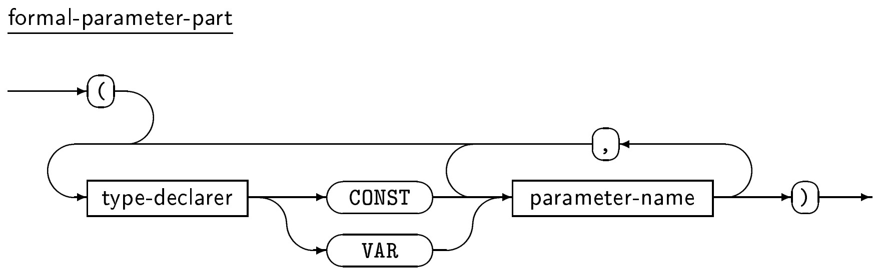


Figure 10.3: Formal parameter pack

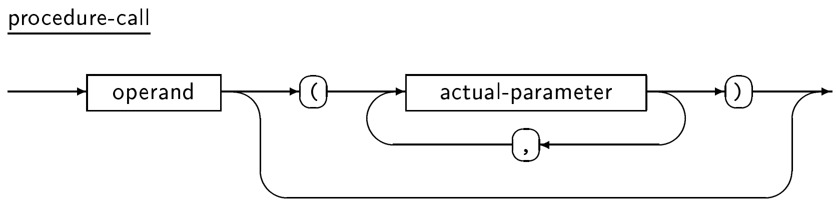


Figure 10.6: Procedure call

call, the values of objects from the environment of the call are bound to the formal parameters of the procedure. The syntax of the procedure-call is shown in Fig. 10.6 and 10.7.

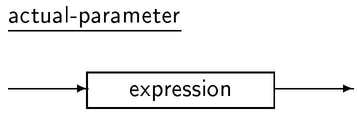


Figure 10.7: Actual parameter

The syntax diagram for actual-parameter has been simplified by omitting the possibility of passing a procedure as a parameter.

The value and the effect of the call of a procedure are the value and the effect of the execution of its body, taking into account the binding of parameters *(parameter mechanism)* as described in the next sections. A call is executed by first binding the values of the actual parameters to the corresponding formal parameters, and then executing the body of the procedure.

The types of the formal parameters and the corresponding actual parameters have to be the same. If a formal parameter has the access right `VAR` then the corresponding actual parameter also must be a `VAR`, so that assignment to it is possible.

If the body of the procedure yields a value then this value after returning from the call acts as the result of that call. For example the call `sin (0.2)` yields as result the real that is computed as the value of the body of the procedure `sin`.

We shall now discuss the parameter mechanism.

10.3.5.1 `CONST`-parameters

In calling a procedure, the formal parameters with a `CONST` access attribute (the *formal constants)* are initialized to the values of the corresponding actual parameters.

Consider the call `average (1.3, 1.7)` of the procedure already described. It yields the same value as the paragraph

```
REAL CONST a :: 1.3, b :: 1.7;
(a + b) / 2.0
```

consisting of two declarations, one for each formal constant, with the actual parameters as initializations, followed by the body of the procedure. Therefore the call

```
put ( average(1.3, 1.7) )
```

writes the real number one-and-a-half.

10.3.5.2 `VAR`-parameters

For `VAR`-parameters *(formal variables)* the correspondence between formal and actual parameters is even tighter. Any assignment to the formal parameter is also an assignment to the actual parameter: the formal parameter is nothing other than an *alias* for the actual parameter.

Consider the example:

```
PROC increment (INT VAR x, INT CONST y):
   x := x + y
ENDPROC increment;

INT VAR number :: 0;
increment (number, 3)
```

For the duration of the call, the formal variable `x` is an alias for the actual variable `number`. Any assignment to `x` assigns also to `number` and vice versa. The call has the same effect as the execution of the paragraph

```
{ Let x be an alias for number }
INT CONST y :: 3;
x := x + y
```

After the call, `x` is no longer an alias for `number`, but the assignment to `x` has had its effect on `number`. Therefore the call `increment (number, 3)` has as net effect that the variable `number` is incremented by `3`.

10.3.6 Scope of local declarations

A procedure declaration makes the name of the procedure visible in the whole scope of the declaration, in our case the whole program. The formal parameters of the procedure and all objects declared within the body of the procedure *(local objects)* have as scope the procedure declaration itself, and are not visible outside it. This means that the names of objects, declared in the environment of the procedure *(global objects),* can be re-used within the procedure as names of formal parameters, local objects, etc. The local meaning then holds only within the procedure and the global meaning holds outside it. It is as if the local objects automatically and invisibly obtain another unique name if their name happens to be the same of that of a global object.

In the context `INT VAR y :: 13, x :: 4` consider the execution of the call `increment (y, x)`. To avoid name conflicts, the local names `x` and `y` in the procedure `increment` are, as it were, changed into names `xx` and `yy`, so that `increment (y, x)` is executed as

```
{ Let xx be an alias of y }
INT CONST yy :: x;
xx := xx + yy
```

The fact that a procedure can have local objects is one of the essential differences with refinements. On the other hand, the declarations occurring outside a procedure have as scope the whole program!

The syntax of Elan does not allow a procedure to be declared local to a procedure, so all procedures stand side by side in a global environment.

10.3.7 Communication with the environment

Procedures communicate via their formal parameters with the environment in which they are called. Another form of communication is possible via a *global variable* that is visible both from the call and from the body of the procedure itself. Just like refinements, procedures can communicate via variables in their environment. The environment can influence the behaviour of a procedure through such a variable.

The following example illustrates how we might control the behaviour of a procedure through a global variable.

```
INT CONST is sine :: 1, is cosine :: 2;
INT VAR type :: is sine;

REAL PROC sine or cosine (REAL CONST x):
   IF type = is sine
   THEN sin (x)
   ELIF type = is cosine
   THEN cos (x)
   ELSE
     put ("Unknown function type");
     line;
     0.0
   FI
ENDPROC sine or cosine;
```

The chimaera `sine or cosine` behaves as the sine or as the cosine, dependent on the value of the global variable `type`.

It is an excellent example of the *bad* programming style that should be, at any rate, avoided: the internal behaviour, i.e. the control, of a procedure is influenced through global variables (so called flags). If an error occurs in the environment it may cause an error also in the procedure, far from the eventual place where the error has been made. Such an error is extremely difficult to localize. Through global data objects the procedures should exchange data and no control information! This bad programming style roots in assembly-level programming, and makes precisely that protection mechanism ineffective that makes the distinction between high- and low-level programming languages.

There is an important difference between communication via parameters and communication via global variables. In the first case, the communication via parameters, we can see at the call explicitly which objects may be modified by the procedure. We can call the procedure with different sets of actual parameters. In case of communication via global variables, the variables that can be modified by the procedure are implicitly given by the procedure declaration. At the call of the procedure we cannot see which global variables may be modified.

Both techniques for communication have their own applications. Which is preferable depends on circumstances. When a procedure is to be called in different environments this may be a reason to prefer the use of parameters.

When we have a collection of collaborating procedures that together realize some abstraction, and for that purpose need some common memory, it is desirable to let the communication between those procedures run via global variables. Communication in this case happens behind the scenes as it were, because it is not relevant to the environment in which the procedure is called; it is abstracted away from.

10.3.7.1 Alias problems

As we have seen, a global variable may, via a `VAR`-parameter, be known within a procedure under two different names. It is not advisable to have assignments to both, as in:

```
INT VAR total :: 10;
PROC riddle (INT VAR t):
   total INCR 1;
   t INCR total
ENDPROC riddle;
```

The meaning of the call `riddle (total)` is not easy to deduce without the help of a tableau (Fig. 10.8).

total, t
~~10~~ ~~11~~ 22

Figure 10.8: Alias

Indisciplined use of an alias opens the door to dangerous side effects. We advise you to avoid alias problems and not to write programs whose meaning depends on subtle details of the parameter mechanism.

10.4 Classification of procedures

Procedures can be classified in various ways. One criterion is to distinguish whether the procedure does or does not yield a result, i.e. whether its execution yields a value.

Whenever a procedure (directly or via a formal parameter) modifies the value of a global variable one says that that procedure has an *effect.* Note that this effect need not be observable within the program that uses the procedure, but may also exist in the printing of a text on a line printer or the screen of the terminal (a kind of external variable). Strictly speaking, a procedure always has an effect, because the call of the procedure at least uses up some computer time. However we will disregard the aspect of time. (The *time* as a variable has an important role in various application areas, e.g. in controlling industrial processes, rockets, space ships etc.)

The observation that a procedure may or may not have an effect and may or may not yield a value leads to a classification of procedures in four kinds, dependent on their behaviour with respect to their environment.

A procedure may:

1. have an effect and yield a value *(function with effect),*
2. have an effect and yield no value *(action),*
3. have no effect but yield a value *(function),*
4. have no effect and yield no value *(dummy).*

The first sort is a mixture between the second and the third. These procedures are difficult to use, because their ambiguous behaviour may be a source of errors in programming. The fourth sort is the most innocent and has as an important parameterless representative

```
PROC dummy:
ENDPROC dummy;
```

Procedures of this last sort can be used during the testing of programs. By taking a "dummy" procedure for an action that has not yet been programmed, with a suitable heading and an empty body, often other parts of the program may be tested at an early stage.

10.4.1 Functions

An example of a function is:

```
REAL PROC max (REAL CONST a, b):
   IF a > b THEN a ELSE b FI
ENDPROC max;
```

that computes the maximum value of the parameters `a` and `b`. By the aid of `max` we can define

```
REAL PROC max of 4 (REAL CONST a, b, c,
d):
   max (max (a, b), max (c, d))
ENDPROC max of 4;
```

which computes the maximum of four reals. A *pure function* is a mapping from its parameters to a result that depends only on the value of those parameters. Well-known examples of pure functions are the conventional trigonometric functions such as `sin` and `cos`.

Programming languages also allow *impure functions,* whose result depends not only on the parameters but also on global variables. The examples in this section all are pure functions.

10.4.2 Actions

An action has an effect on its environment but yields no result. Well-known examples of actions are the procedures `get` and `put` for reading and writing respectively. Other examples are:

```
PROC count (INT VAR number):
   number INCR 1
ENDPROC count;
```

that increments the value of its parameter by 1.

```
PROC exchange (INT VAR x, y):
   INT CONST aux :: x;
   x := y;
   y := aux
ENDPROC exchange;
```

This action exchanges the value of its parameters. Strictly speaking, functions are not necessary because we can achieve the same effect using only actions (cf. "statement languages" like COBOL). As an example, instead of the standard function `sin` we could define an action `compute sine`:

```
PROC compute sine (REAL CONST x, REAL VAR
sine):
   sine := the sine of x
ENDPROC compute sine;
```

In order to compute the value of the formula `1 - sin(x) ** 2` with the aid of the procedure `compute sine`, we would then have to write:

```
REAL VAR s, result;
compute sine (x, s);
result := 1 - s ** 2
```

The example makes clear that the notation as a function is much more palatable.

It is also possible to go all out for functions (as in pure "expression languages" such as LISP), but in general it is most convenient to have a choice between both forms of expression.

When seeing a procedure purely as a mapping from its (eventual) parameters to a value, the function form will be preferable. In other cases, e.g. when no result or rather more than one result is to be yielded, the procedure will be declared as an action. An example of this last case is:

```
PROC division with remainder
   (INT CONST dividend, divisor, INT VAR
quot, rem):
   quot := dividend DIV divisor;
   rem := dividend - quot * divisor
ENDPROC divide with remainder;
```

10.4.3 Functions with side effect

When the execution of a function has also an effect (on a global variable) we speak of a *side effect.* This name already makes clear that the purist frowns upon side effects. A function is supposed to yield a value, not to modify the global environment!

Upon closer consideration, legitimate examples of the meaningful use of side effects are easy to find. They have in common that the global environment implicitly acts as a memory for the function and influences its result. Examples:

```
INT VAR n :: 0;

INT PROC next client:
   n INCR 1; n
ENDPROC next client;
```

Obviously a global memory is essential to this function.

```
BOOL VAR a;

BOOL PROC flipflop:
   a := NOT a; a
ENDPROC flipflop;

PROC set:
   a := TRUE
ENDPROC set;

PROC reset:
   a := FALSE
ENDPROC reset;
```

For anybody conversant with electronics this set of procedures has a familiar behaviour. Notice the way in which `flipflop` yields its value.

Another function with a side effect, again without parameters, is:

```
INT PROC next nat:
   INT VAR numb;
   REP get(numb)
   UNTIL numb >= 0
   ENDREP;
   numb
ENDPROC next nat;
```

which yields as a value the next non-negative number of the input and has a side effect on that input.

10.4.4 Genericity

Procedures are *generic*: different procedures with the same name may occur alongside one another as long as they differ in the number or type of formal parameters. In a procedure call, the number, the order and the type of the actual parameters must agree with the number, order and type of the formal parameters of one of the procedures declared with that name. That procedure will then be identified by this procedure call. In the second part of this book we shall introduce the use of genericity as an abstraction mechanism.

10.4.5 Encapsulation

A secure use of procedures using global variables as memory demands a facility to protect the global objects that are necessary for the correct execution of the procedures from the environment in which those procedures are called. The programming language Elan to this purpose contains a mechanism, the *packet mechanism,* which we shall describe in the second volume.

For the implementation of large program systems, on which many people will be at work, such an explicit encapsulation mechanism is essential.

10.5 Structure of programs

At this point we can discuss the structure of Elan programs. We shall not yet introduce the complete syntax (which can be found in appendix A) but describe two subsets of Elan that we shall call Elan-0 and Elan-1 respectively.

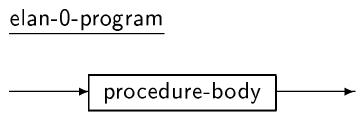


Figure 10.9: Elan-0 program

Is Fig. 10.9 a surprise? The language mechanisms that we introduced in the first 8 chapters turn out to be precisely those that can be used in the body of a procedure.

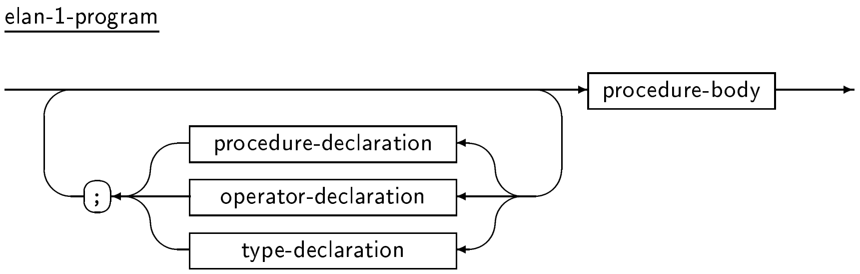


Figure 10.10: Elan-1 program

One conclusion from that is that we may take any complete Elan-0 program and enclose it between the lines

```
PROC program:
ENDPROC program
```

thus turning it into a procedure, an action without parameters.

By allowing the closed-declaration for procedures (and operators and types), we obtain a second, wider subset, Elan-1, i.e. Elan without packets (Fig. 10.10). It can be seen that an Elan-1 program consists of a Bottom-Up part, in which some abstract algorithms and types are defined, followed by a Top-Down part in which they are used.

For the rest of this book we shall employ the subset Elan-1.

10.6 Procedures and refinements revisited

Refinements and procedures have much in common. Why then do we distinguish two mechanisms?

The differences between procedures and refinements are subtle but have rather far-reaching consequences.

- A procedure has as its environment (i.e. as the scope of its declaration) the program. The same holds for refinements of the program but not for refinements of procedures. Those have as environment the procedure itself, and therefore can be used for local Top-Down programming.

- Communication between refinements can occur only implicitly via effects on common global objects. Communication between procedures takes place explicitly, by means of parameters.

- Declarations within a procedure are local to that procedure. Declarations in a refinement are visible in the whole program; thus it is possible to refine a declaration! Many examples in this book show that especially an initialized declaration (which introduces a variable and at the same time leads to a well-defined state of the program) can be verbalized well.

- A procedure only obtains its meaning at its declaration; before that it may not be called. A refinement can be invoked everywhere in its scope without the necessity for executing a declaration.

In other programming languages, refinements do not appear as separate constructs. One can then use (parameterless) procedures as refinements. This does lead to a number of problems:

- Some programming languages (like standard BASIC) have such severe limitations on the choice of names of procedures that these can no longer be seen as abstraction means.

- The fact that procedures can only be called after execution of their declaration means that such a declaration has textually to precede its first call. This conflicts with the order of the Top-Down programming process: at any moment we may postulate some specific abstract algorithm that we have to declare only later. In using procedures, the order of invention and the textual order can no longer be the same.

- Writing down a procedure declaration with a detailed heading adds so much "overhead" to the design process that it restrains the programmer from writing many short abstract algorithms. Implicit communication raises much less overhead and fits better into the Top-Down programming method.

- The necessity in many programming languages (like PASCAL) to declare all variables before the algorithmic part of a procedure of program strongly impedes a natural Top-Down style. Elan on the other hand is very liberal as regards the placement of declarations.

- The block structure that arises through the possible nesting of procedure declarations causes great trouble to beginners. The very simple scope rules of Elan are easier to follow.

There is no doubt that procedures may be used strictly as refinements, but the problems mentioned justify the introduction of refinements as a separate syntactic mechanism. Also from the point of view of systematization of language concepts, it is advantageous to distinguish between the means for Top-Down programming (refinements, control structures) and Bottom-Up programming (packets, procedures, operators, types).

Refinements are above all a didactic aid in learning systematic programming. Once their use has been fully understood it is possible to program systematically even without such an aid.

Refinements are for the programmer what the navel is for the Zen Buddhist.

10.7 Exercises

1. (Roman numerals) Write a procedure

```
    INT PROC value of (TEXT CONST roman
numeral)
```

   that converts a Roman numeral to an integer. Make use of an auxiliary procedure to obtain the values of Roman digits.

2. (Converting to Roman numerals) Write a procedure

```
    TEXT PROC romanize (INT CONST number)
```

   that represents a natural number in the Roman numbering system. Test it with the previous procedure.

3. (Standard library from chapter 6) Realize the procedures `pos`, `text`, `subtext` and `replace` in terms of the operations `+`, `LENGTH` and `SUB`.

Chapter 11

Languages and grammars

In this chapter we want to take a look at languages and grammars. A linguistic application of Informatics will be presented: the generation and analysis of sentences according to a grammar. In passing, we shall have to deal with the subject of random numbers.

11.1 On the description of languages

The English language can be seen as the collection of all its sentences. Likewise, the language Elan can be seen as the collection of all its programs.

A natural language is a living organism, part of an intricate social, cultural and economic system. The boundaries of a natural language are vague and subject to shifting in time. Expressions, constructions and words will slowly be adopted and become commonplace, whereas others become obsolete. There are dialects and language variants. Pronunciation and spelling will change over time.

For an artificial language like our programming language, explicitly constructed by human beings, the boundaries are much sharper. But there we have a problem in sharing it with others: We are not immersed in the language from our birth, and cannot take recourse to native speakers of the language. The inventor of an artificial language will have to describe it.

How do you describe a language? To enumerate a complete listing of all English sentences is obviously impossible, because there are too many. There is not even a strict bound to the number of sentences: I can take any sentence and make it two words longer by embedding it in

> He said: " ... ".

Furthermore, such an enumeration of all sentences is not very enlightening, because sentences have meaning only in relation to a specific context.

In the description of a language, be it natural or artificial, three different aspects can be distinguished: syntax, semantics and pragmatics.

The *syntax* indicates which sequences of symbols form the sentences of the language and what syntactic structure they possess. The *semantics* assigns a meaning to syntactically correct sentences. The *pragmatics* is concerned with the relationship between the language and the human being who uses it, in other words: the heart of the matter.

We shall not concern ourselves with semantic and pragmatic description, but shall introduce one particular formalism for syntactic description from mathematical linguistics, the *context-free grammar*. This formalism is related to the syntax diagrams that we introduced earlier in this book, but it may lead to a more concise notation with a somewhat higher degree of abstraction.

Most applications of Informatics in the human sciences, and especially those in linguistics, can be classified as follows:

- *Text storage.*

  The input, storage, modification and retrieval of texts ("word processing"), their preparation for printing and automatic typesetting. You will have noticed that the text of this book has been prepared with the aid of the computer (and incidentally quite a number of human beings). As the output devices of the computers get more perfect and more professional, the computer turns more and more into a super-typewriter.

- *Calculating with texts.*

  The computer with its long patience and unwavering conscientiousness is a very reliable aid for counting the occurrences of certain phenomena in texts (language statistics), for screening and sorting linguistic data and for compiling indices and concordances. More and more the computer is replacing the shoebox stuffed full of indexing cards on which the old-fashioned linguist kept his notes.

- *Applications of syntactic techniques to texts.*

  The linguist and the informatician share a common interest in the description of languages and the automatic processing of sentences. This interest is reflected in the examples in this chapter.

11.2 A syntactic notation

A context-free grammar ("CF grammar", "CFG")
is a formal notational system, in which linguistic concepts are defined in terms of other concepts and, finally, in terms of specific words or classes of words.

As an example we verbally express a part of English grammar:

1. a sentence consists of a subject, followed by a verb, followed by an object;
2. a subject is either a personal pronoun, or an article followed by a noun;
3. an article is one of the symbols 'the', 'a' or 'an'.

In a context-free grammar we express this same information by means of the three rules

```
sentence:
  subject, verb, object.

subject:
  personal pronoun;
  article, noun.

article:
  a-symbol;
  an-symbol;
  the-symbol.
```

The punctuation marks in this grammar can be pronounced as follows:

:	as "is a",
,	as "followed by a",
;	as "or a",
.	as a short pause.

The resemblance of this notation to the Elan notation for refinements is no accident. Refinements have a similar descriptive character to a grammar and we might well call the refinements the "special grammar" of the program.

In order to simplify the recognition of symbols we may use the convention that their name always ends in -symbol.

11.2.1 Context-free grammar

In a context-free grammar we distinguish between *concepts* and *symbols*, each represented by their name, consisting of lower case letters and possibly digits. We may use spaces and hyphens ("-") to enhance readability.

Words like sentence, subject and verb are names of concepts, the so-called *non-terminal symbols* or *non-terminals*. The grammar has for every non-terminal a *rule,* that defines precisely which sequences of words are comprised by this concept.

Words like a-symbol, the-symbol etc. are names for the symbols of the language, the so-called (*terminal*) *symbols* or *terminals*. For these terminals there is no production rule, but instead one or more *representations* are given. Take for example the the-symbol. It can be pronounced, written by hand or printed, all the time remaining the same symbol, just like 2 or two always retains its twoness.

A rule consists of a *left-hand side* ("definiendum"), a colon, a *right-hand side* ("definition") and a period. The left-hand side consists of the non-terminal to be defined. The right-hand side consists of one or more *alternatives,* separated by semicolons. An alternative in the definition of a non-terminal we also call one of the *direct productions* of this non-terminal.

An alternative consists of a *list* of zero or more words, i.e. terminals or non-terminals, separated from one another by commas. An alternative may also be empty.

Based on the notion of direct production we shall now define the notions of production, terminal production, sentence and language.

A *production* of a non-terminal x is either a direct production of x, or a list of words, obtained by replacing in a production of x some non-terminal y by a direct production of y.

Some productions of sentence:

```
sentence
subject, verb, object
personal pronoun, verb, object
article, noun, verb, object
a-symbol, noun, verb, object
the-symbol, noun, verb, object
...
```

By a *terminal production* of x we mean a production that contains no further non-terminals and therefore consists solely of a list of terminals. The *representation* of a terminal production is the sequence of representations of its terminals.

A *sentence* is a representation of a terminal production of the syntactic notion sentence. The collection of all sentences, described by a grammar, is the *language* of that grammar.

In the back of this book can be found a context-free syntax of Elan (appendix A). This is the syntax which is used by the Elan implementation to determine whether a specific sequence of symbols belongs to Elan.

With this grammar in hand, it is possible to find answers to questions about the syntax of Elan, like:

- Is `marilyn` an identifier?
- Can the body of a refinement end in a `LEAVE`-construct (the terminator)?
- Can the body of a refinement consist of an object-declaration?

without trying out examples on the computers.

11.2.2 Other formalisms

The notation for context-free grammars that we have used is not one of the various conventional notations from linguistics, but a notation from Informatics named after A. van Wijngaarden [ALGOL68].

To be precise, the so-called *two-level grammars* have been named after van Wijngaarden. In fact we have used a simplified version of the van Wijngaarden notation, using only one context-free level. Another notation used in Informatics is the so-called *Backus-Naur Form.* All these different notational systems are however interchangeable.

11.2.3 Using a grammar

Suppose we have obtained a context-free grammar of a substantial part of the English language. How can we determine how good this grammar is?

One technique would be to *analyse* a large number of sentences by the aid of the grammar and to determine whether a sufficiently large fraction of those sentences can be analysed according to the grammar. The linguist who works in this fashion is called a *corpus linguist,* named after the (huge) collections of text ("corpora") with which he works.

Another technique consists in *generating* sentences: producing examples of sentences according to the grammar, which are then checked for grammaticality by human beings. A grammar suitable for generating sentences is called a *generative grammar.*

Observe that there is little sense in generating, in some order, "all" sentences of the language: their number is usually infinite. That would be extremely boring and not very instructive. It is more interesting to generate "by chance" examples of English sentences.

We shall first indicate how you might program a generative grammar. Assume that one of the rules says:

```
statement:
  subject, verb, object.
```

In order to generate an example of a statement, we have to generate an example of a subject, followed by an example of a verb and then an object. This sounds remarkably like the calling of algorithms. For every notion, we introduce a procedure written in Elan, e.g.:

```
PROC statement:
   subject;
   verb;
   object
ENDPROC;
```

There is a striking resemblance between this notation in the programming language and the syntax rules, but unfortunately the punctuation marks ";" and "," have a different meaning in the two systems. We can continue with the syntax rule

```
subject:
  personal pronoun;
  noun group.
```

As an example of a subject I have to generate either a personal pronoun or a noun group, with equal probability:

```
PROC subject:
   IF fifty fifty
   THEN personal pronoun
   ELSE substantive group
   FI
ENDPROC;
```

The condition `fifty fifty` has to be a boolean expression that yields true with 50% probability and false with 50% probability. How do we achieve this?

11.3 Excursion: about chance

A six-sided dice is an instrument for generating numbers by chance. From such a dice we expect:

- even knowing the whole of history, it is *unpredictable* what the next outcome will be;
- the six possible outcomes are *homogeneously distributed,* i.e. they all have an equal probability.

Every model that has those two properties can serve instead of a dice. We can try to deduce such *random numbers* from a natural process, like a white noise generator, or from the "random" contents of some part of the memory of the computer, but the first demands extra hardware and the second will not be very satisfactory.

We can also try to determine algorithmically a sequence of real random numbers in the range 0.0 to 1.0 as

$$\begin{aligned} R_0 = &\quad \text{initial value,} \\ R_1 = &\quad \text{obtained from } R_0, \\ R_2 = &\quad \text{obtained from } R_1, \\ \ldots \end{aligned}$$

in such a way that the sequence R_i is unpredictable and homogeneously distributed. For practical reasons we prefer to compute a sequence of non-negative integers I_i and then let

$$\mathtt{R_i} = \mathtt{real}(\mathtt{I_i})/\mathtt{real}(\mathtt{maxint})$$

in which $I_i > 0$, and of course $I_i \leq$ `maxint`. The simple computation scheme

$$I_{i+1} = (a * I_i + b) \bmod c$$

turns out to have surprisingly good properties provided a, b and c are chosen with care. In order to simplify the computation we choose $c =$ `maxint`, on most machines a power of two minus 1. Observe that the sequence is in every case periodical: there are only a finite number of integers, so that after a certain number of terms an element of the sequence must be repeated. A number of I_i occurring for the second time in the sequence as I_{i+k} will have the same successor value $I_{i+1} = I_{i+k+1}$ and so on.

We must obviously try to keep the period as long as possible. It can be at most equal to `maxint`. Also the sequence has to be unpredictable. We have to choose for a and b prime numbers, such that a, b and c have no divisors in common. For well-chosen values of a and b the elements of the sequence are homogeneously distributed, and also stronger statistical tests do not point to regularities. In distinction to real chance, this sequence is reproducible. This allows us to replay a "random process". Such a reproducible random sequence is called *pseudo-random.*

Due to the way these pseudo random numbers are generated, the lowest bits of consecutive random numbers are highly correlated. The higher bits are "more random".

11.3.1 Random numbers in Elan

We do not ourselves have to search for suitable values of a, b and c for our computer. Somewhere in the standard packets of Elan there is a hidden integer variable `last random` declared as

```
INT VAR last random
```

and an algorithm with a real result, similar to

```
REAL PROC random:
   last random := (a * last random + b)
MOD maxint;
   abs ( real(last random) / real(maxint)
)
ENDPROC random;
```

Using this algorithm, the expression

```
random < p
```

has a probability `p` to yield `TRUE`.

At the start of the execution of a program the variable `last random` is automatically and mysteriously set to some initial value, which is probably based on the constellation of the stars or the built-in clock of the computer. This makes the random sequence truly unpredictable for us.

It is possible to choose your own random sequence, initializing the hidden variable by a call of `initialize random (x)` in which `x` is a real expression, whose value has to lie between 0.0 and 1.0. By doing this again the same random sequence follows, thus making it possible to repeat history.

We can simulate a dice as

```
make a dice:
   INT CONST dice :: trunc(random * 6.0) +
1.
```

This results in a beautiful dice: it is unpredictable, homogeneously distributed and, if desired, also reproducible.

In order to facilitate the production of integer random numbers, there is a variant of `random` with two integer parameters. It yields an integer that lies within those two bounds. The call `random (1,6)` is the equivalent of a dice.

The algorithm `fifty fifty` can now be written as:

```
fifty fifty:
   random < 0.5.
```

11.3.2 Random numbers in Elan-0

In the standard library of Elan-0, the algorithms just mentioned do not occur, but instead an algorithm `choose128` is given. The value it yields is the same as that of `random (0,127)`.

To simulate with this algorithm a dice with six equiprobable results is somewhat more complicated. One method is to take only the values from the interval $[0 : 125]$, and these modulo 6:

```
make a dice:
   INT VAR random, dice;
   REP
     random := choose128
   UNTIL random <= 125
   ENDREP;
   dice := 1 + random MOD 6.
```

With the aid of `choose128` we can realize the algorithm `fifty fifty` as follows:

```
fifty fifty:
   choose128 MOD 2 = 0.
```

But this is not a very satisfactory solution, since its value then depends on the lowest bit of a pseudo random number which, for various reasons, is not the most unpredictable. It is preferable to use the higher bits:

```
fifty fifty:
   choose128 > 63.
```

11.4 Example: A generative grammar

We now give a context-free grammar for a part of the English language, in which we assign a probability to each of the alternatives (a *probabilistic CF grammar).* According to this grammar we will write a program to generate random sentences. We can control only the form of the sentences produced, their meaning escapes our analysis. In order to make our sentences not too nonsensical, we will restrict our vocabulary to semantically strongly loaded words from a specific area. This particular vocabulary was inspired by the verse "Mary had a little lamb". The symbols appearing in the grammar we shall indicate by their representation between quotes.

sentence: subject, predicate.

subject:
  [.7] substantive group;
    personal pronoun 1.

The notation with the square brackets serves to indicate that the probability of the first alternative of this rule is equal to 0.7 and of the second alternative is 1.0 - 0.7 = 0.3.

We program this rule as follows:

```
PROC subject:
   IF random < 0.7
   THEN substantive group
   ELSE personal pronoun 1
   FI
ENDPROC;
```

We continue the grammar:

substantive group:
    article, noun phrase.

article:
  [.45] "a";
        "the".

We output the terminal symbol to the screen, taking care to follow each symbol by a space.

```
PROC article:
   IF random < 0.45
   THEN put ("a ")
   ELSE put ("the ")
   FI
ENDPROC;
```

noun phrase:
  [.25] adjective noun phrase;
        noun part.

Notice that the concept noun phrase is defined here in terms of itself. Such a concept is *recursive.* The intention of this rule is to indicate that there is no *a priori* upper limit to the number of adjectives in front of a noun part. Of course the probability of having a large number of adjectives is rather small. We realize this effect with the aid of a conditional repetition with a 25% chance of continuation.

```
PROC noun phrase:
   WHILE random < 0.25
   REP adjective
   ENDREP;
   noun part
ENDPROC;
```

We have a choice of adjectives.

adjective:
  [.30] "little";
  [.60] "meek";
  [.90] "big";
        "bad".

In order to generate an example of an adjective we compute one random number and then, on the basis of its value, choose one of the alternatives.

```
PROC adjective:
   REAL VAR r :: random;
   IF   r < 0.30
   THEN put ("little ")
   ELIF r < 0.60
   THEN put ("meek ")
   ELIF r < 0.90
   THEN put ("big ")
   ELSE put ("bad ")
   FI
ENDPROC;
```

The rest of the grammar should be self-explanatory. We do not give an Elan procedure for each individual rule — the correspondence should by now be clear.

noun part:
  noun, rel clause option.

rel clause option:
  [.25] rel clause;
        .

noun:
  [.20] "boy";
  [.40] "girl";
  [.60] "lamb";
  [.80] "bear";
        "tree".

rel clause:
        "that", predicate.

predicate:
        adverbial option, verb, object.

adverbial option:
  [.20] modifier; .

modifier:
  [.33] "always";
  [.67] "often";
        "never".

verb:
  [.20] "had";
  [.40] "sees";
  [.60] "likes";
  [.80] "eats";
        "dreams about".

object:
  [.80] substantive group;
        personal pronoun 4.

personal pronoun 1:
  [.25] "he";
  [.50] "she";
  [.75] "Mary";
        "Jim".

personal pronoun 4:
  [.25] "him";
  [.50] "her";
  [.75] "Mary";
        "Jim".

The syntax serves as a blueprint for a program generating random sentences.

11.4.1 Results

Some example sentences generated by the aid of this program are

```
the bear that sees the bear likes a bad
bear
the bear likes him
the boy dreams about the bear
a girl never sees the girl
Jim sees Mary
a boy that sees the big tree always likes
a boy that always
    eats a tree
Mary eats a little bear
```

It is obvious that a sentence appears more meaningful if it is short. This is no wonder, since all meaning is accidental. The longer a sentence is, the more opportunity it gets to contradict itself.

11.5 Syntax Analysis

Let us now turn to the subject of syntax analysis, which comprises two closely related activities: sentence recognition and parsing.

A *recognizer* for a (context-free) language G is an algorithm that, for a given sequence of symbols s, determines whether s is one of the sentences the language of G. It gives a yes/no answer.

$$recognize(G, s) = \begin{cases} s \in L(G) \rightarrow true \\ s \notin L(G) \rightarrow false \end{cases}$$

A *parser* is an algorithm that, gives a sentence $s\epsilon L(G)$, determines its structure (in the form of a parse tree).

$$parse(G, s) = \begin{cases} s \in L(G) \rightarrow parse\ tree\ for\ s \\ s \notin L(G) \rightarrow undefined. \end{cases}$$

The two activities can profitably be combined: A parser should also signal non sentences in a proper fashion, and once a recognizer has been constructed it can easely be extended to a parser — infact the best proof that a sequence of symbols forms a sentence is to construct a parse tree for it!

We shall investigate some aspects of syntax analysis, attempting to construct a recognizer for the fragment of english just described.

11.5.1 Recursive Descent

We want to construct a recognizer for sentences, that is, a boolean procedure `is sentence` that will yield `true` if and only if the input contains a terminal production of sentence. A sequence of symbols is a terminal production of *sentence* precisely if it consists of a terminal production of subject followed by one of predicate.

Basically we proceed in the same way as we did for constructing a generator, by writing a boolean recognition procedure for each of the non-terminal symbols in terms of recognizers for the other non-terminal and for the terminal symbols. Very simple, but we will meet a number of complications.

Recognizing a terminal symbol amounts to snipping off the head of the input provided it is the symbol we want to recognize. We hold the sequence of symbols as a text in the variable `input` and use an index `inptr` to remember how far the recognition process has proceeded.

```
TEXT VAR input :: the input we want to
recognize;
INT VAR inptr :: 1; { the position of the
next symbol }
```

We provide one boolean procedure to recognize a symbol, which also takes care of the blanks between symbols.

```
BOOL PROC is (TEXT CONST s):
  WHILE (input SUB inptr) = " "
  REP inptr INCR 1
  ENDREP;
  IF subtext (input, inptr, inptr + LENGTH
s - 1) = s
  THEN
    inptr INCR LENGTH s;
    true
  ELSE false
  FI
ENDPROC is;
```

How should we compose a recognition procedure for

```
sentence: subject, predicate.
```

out of recognition procedures for its constituents? We might first think of

```
BOOL PROC is sentence:
  is subject AND is predicate
ENDPROC is sentence;
```

but this is wrong: rather than trying the second recognizer only in case the first one succeeds, it always tries the second — a dyadic operator (in this case `AND`) always evaluates both its operands. And what is more, they are evaluated collaterally, so we are not even sure of the order in which the two recognizers are called! We have to be much more careful.

```
BOOL PROC is sentence:
  IF is subject
  THEN is predicate
  ELSE false
  FI
ENDPROC is sentence;
```

The next sytax rule

```
subject:
   substantive group;
   personal pronoun 1.
```

has two alternatives. We try them in order and if any of them succeeds, a subject was found.

```
BOOL PROC is subject:
  IF is substantive group
  THEN true
  ELSE is personal pronoun 1
  FI
ENDPROC is subject;
```

substantive group:
article, noun phrase.

The next rule leads to a choice between terminal symbols

article:
"a";
"the".

```
BOOL PROC is article:
  IF is ("a")
  THEN true
  ELSE is ("the")
  FI
ENDPROC is article;
```

The remaining rules can be transcribed to recognition procedures without further adventures, apart from the treatment of the adverbial option. Since it is optional, it cannot fail to be recognized. Rather than transcribing it into a boolean procedure that always yields `true`, we make it an action.

predicate:
adverbial option, verb, object.

adverbial option:
modifier; .

```
BOOL PROC is predicate:
  adverbial option;
  IF is verb
  THEN is object
  ELSE false
  FI
ENDPROC is predicate;

PROC adverbial option:
  IF is modifier
  THEN
  ELSE
  FI
ENDPROC adverbial option;
```

The name of the resulting procedure is not prefixed with `is ...`, as we did in constructing other recognition procedures, in order to show that it cannot fail.

Finally we add a small driver program to read an input line. It tries to recognize a sentence from the input, and continues to offer the input for editing until it is recognizable.

```
simple parser:
  start with empty input;
  REP ask further input
  UNTIL is sentence
  ENDREP;
  congratulations.

start with empty input:
  TEXT VAR input :: "";
  INT VAR inptr :: 1; { index of next
input character }
  put ("Input, please ...").

ask further input:
  line;
  edit (input, inptr);
  inptr := 1.

congratulations:
  line;
  put (inptr * "-");
  line;
  put ("Success!");
  line.
```

Upon recognizing a sentence from the input, the program stops after underlining the recognized portion of the input text. This serves to cope with the problem that the recognizer works from left to right and has the property that, once it recognizes a sentence, it does not care whether it covers the whole input or is followed by some non-sense. It is easy enought to impose the additional condition that a sentence must cover the whole input, but this problem gives a hint that there are some problems in combining recognizers.

11.5.2 Some complications

This particular recognizer will work correctly for the grammar from which it was derived, but that does not mean that we can construct in this fashion a recognizer for any context-free grammar.

To begin with, there is the matter of *left-recursion*: a rule may very well be left-recursive, like

noungroup:
noun;
noungroup, postmodifier.

which expresses the fact that a noun can have any number of postmodifiers. Turning this rule into a recognizer will lead to an un-ending program execution: when the input does not start with a noun, the procedure `is noungroup` will call itself, and so on until the end of the world or until the memory of the computer is exhausted (whichever comes first).

Furthermore, the situation where two alternatives start with the same (terminal or non-terminal) symbol, like

nounpart:
noun, rel clause;
noun.

leads to complete confusion of the recognizer, and to an altogether different language being recognized than is generated by the grammar (think of a noun followed by another noun instead of a rel clause). That is why this rule has to be left-factored, taking out the common part noun.

nounpart: noun, rel clause option.

rel clause option: rel clause; .

These and other problems make that a given CF grammar only under very restrictive conditions can be recognized by recursive descent. The grammar presented here cleverly adheres to these so-called LL(1)-conditions. There is a rich literature about syntax analysis, to which we refer for further reading [WAI84] [AHO86].

The recursive descent analysis technique is much too restrictive for linguistic purposes, since it can not cope with *ambiguity* . A sentence is ambiguous if it can be produced from the grammar in more than one way, like the famous sentences

> They are flying planes

and

> Time flies like an arrow

Do you see the 2 respectively 4 different analyses?

Analysis techniques that can deal with ambiguity do exist — but this is not the place to pursue this subject.

You may also have noted that the grammar studiously avoids the problems of coordination between pronouns, substantives and verb forms, by having only the 3d person singular forms. It is possible to extend the grammar to deal with other persons and with plural forms, but then it grows tremendously in size. Context free grammars are (as the name indicates) not a convenient formalism to express context dependency, and although linguistics makes wide use of context free grammars under various guises, they are mostly extended with some mechanism for dealing with context — like Augmented Transition Networks [WOO70].

In Informatics a number of extensions to context-free grammars have been invented, like van Wijngaarden Grammars, Attribute Grammars [KNU68] and Affix Grammars [KOS70]. Again, it would lead too far to pursue this highly interesting subject further in this textbook.

11.6 Beyond analysis

Translating sentences from one language to another can be seen as the problem of first analysing a sentence according to the syntax and semantics of the first language and then generating an equivalent sentence in the other language. This process is wrought with all the problems just outlined, plus a few more: how to extract and represent the meaning of a sentence in such a way that an equivalent sentence can be produced and how to produce the translation from the semantic representation.

"Machine Translation", as this problem used to be called, is amongst the oldest applications of computers (then often called "electronic brains") in the very early fifties. After some initial successes, Bar-Hillel showed the inadequacy of the available theory and methods in the late fifties [BAR60], and a soberer period followed, in which linguistic fashion turned to pragmatics — syntax being considered too limited and semantics too difficult to be of interest. In the sixties and seventies powerful analysis and translation techniques were invented in Informatics and computers became so much larger and faster that a fresh interest in the syntax of natural languages and, in particular, in machine translation was raised.

All kinds of applications need a linguistic interface. With the advent of the speaking chip, the analysis and synthesis of human speech and (written) language will get tremendous importance. The present chapter is meant to provide a modest initial introduction to the linguistic application of computers — and at the same time to present a wonderful opportunity to see procedures in action in a larger real example.

11.7 Exercises

1. Describe, by way of a CF grammar, the structure of

   - a train (of waggons, locomotive, coal tender, brake car)
   - a division of the regular army (consisting of regiments, companies, etc.), for as far as you can obtain the necessary information
   - a context-free grammar.

2. Write a generative grammar for the production of free poetry, full of strongly evocative substantives, adjectives and verbs (animals, sea, colours, persons, sorrow, happiness).

3. Write a generative grammar to produce a nicely structured letter full of insults. You can give your opinion of the pedigree of the addressee, his habits, psychological stability and future.

4. (Roulette) Smith and Jones are playing roulette. One plays only on red, the other on black. Their goal is to double their initial capital. Both start with the same initial capital and a bet of one. Jones reacts to a loss by doubling his bet. In case of a win he returns to the basic bet 1. Smith believes himself to be even cleverer. He raises his bet by one in case of loss and upon winning returns to the basic bet. The probability for red and black is of course equal, but not quite fifty percent because with a probability 1/37 the ball ends up on the 0, which is neither red nor black.

   Write a program to simulate an interesting evening at the casino. Try it out with initial capitals 10 and 100. The bank has an unlimited supply of money.

5. (One-armed bandit) Simulate a one-armed bandit. In order to raise the level of verisimilitude, some field work (in pubs and gambling halls where those things can be found) may turn out to be unavoidable.

6. (Shuffling I) Shuffling is the inverse of sorting. Write a program to shuffle a sorted pack of 52 cards by randomly taking a card either from the bottom or from the top of the pack, until the whole pack has been taken, and repeating the process a number of times. Display the shuffled cards during the process and observe the effect of a longer shuffling.

7. (Shuffling II) The shuffling is more effective if we take the cards at random from the sorted pack. Write a program to shuffle a pack of 52 cards and show the result.

   Hint: After choosing a card from the pack remove it indeed. If you simply mark it the random selection might result in fruitless moves which decrease the effectiveness of the algorithm.

8. (Selection game) Manfred Eigen described a simple game simulating natural selection [EIG81].

   In the beginning 10 individuals each of 4 different species live in a small world. They die and are born as other mortals. But whenever one of them dies exactly one other comes into the world, which is a duplicate of one of the remaining individuals. Surprisingly, after a short period of coexistence only one of the species will survive.

   Write the program and study natural selection.

9. (Wasps) There are two rooms separated by a door. One of the rooms is full of flying wasps. When we open the door some of them will find the opening and fly into the other room. After some time the distribution of wasps will be more or less the same in both rooms.

   Write a program to simulate the behaviour of the wasps.

10. (Ten marksmen, ten pigeons) If ten marksmen shoot at ten pigeons simultaneously some pigeons will survive even if all the marksmen are sharpshooters. Show by simulation how many pigeons remain alive.

11. (Chi-square test) The most often used method of examining random distributions is the chi-square (χ^2) test [KNU69].

    Assume we want to test a die simulated on our computer. Then we have to "throw" the die n times and record how often every single result was obtained; let us denote their number by Y_s. Hereupon, the formula

    $$V = \frac{1}{n}\sum_{s=1}^{k}\left(\frac{Y_s^2}{p_s}\right) - n$$

    is to be computed where k is the number of possible outcomes and p_s is their probability; in our case $k = 6$ and the probabilities are equal, i.e. $p_s = \frac{1}{6}$. The value V is then to be compared against the table below so that the quality of our "die" can be judged.

	$k-1=5$	
		not random
$p = 99\%$	0.5543	
		suspect
$p = 95\%$	1.1455	
		almost suspect
$p = 75\%$	2.675	
$p = 50\%$	4.351	acceptable
$p = 25\%$	6.626	
		almost suspect
$p = 5\%$	11.07	
		suspect
$p = 1\%$	15.09	
		not random

    Write a program to test the random number generator of your Elan Programming Environment or test a random number generator of your own making.

12. (Birthdays) 23 persons are celebrating the birthday of a friend. One of them suggests a bet that there are at least two persons present whose birthday falls on the same day of the calendar. Determine by simulation whether this is a fair proposal.

13. (A telephone directory, a needle and the number π) It is possible to determine the number π with a telephone directory and a needle — and with the help of probability theory. The columns in the directory form a raster; let its distance be denoted by d. A needle of length l, where $l \leq d$, will repeatedly be dropped on this raster. The probability that the needle crosses a raster line is $p = 2 * l/(\pi * d)$. With $d = 2$ and $l = 1$, $p = 1/\pi$.

    Write a program to determine the approximate value of π by experiments. *Hint*: use relative frequency instead of probability.

Chapter 12

Recursive algorithms

Quite often an obvious formulation of an algorithm contains in its turn a call of that algorithm: the problem is reduced to a simpler version of the same problem.

In mathematics it is a standard technique to describe certain sequences by means of recurrence equations like $t_n = t_{n-1} * x/n$.

When an algorithm calls itself, we use the term *recursion*. We shall illustrate the use of recursion as a programming technique by means of a number of examples.

12.1 Recursion

In order to give a recursive solution to a problem we look for

1. A simplest case, in which the solution is trivial.
2. A way of reduction the complicated version of the problem to a simpler version of the same problem. In doing so we suppose in each step that the simpler solution has already been found so it can be used in solving the more complex problem.

It should be obvious that this method is soundly based on mathematical induction.

We say a procedure is *directly recursive* if it contains a call of itself, and *indirectly recursive* if it calls a procedure that in its turn calls it (directly or indirectly). We say an algorithm is *simply recursive* if it contains only one directly recursive call of itself which is not contained in a repetition.

We shall first give an example of simple direct recursion.

12.1.1 Example: Printing a number

We want to write a procedure `print number` such that `print number(x)` prints the integer `x` in a minimal number of positions. We assume the procedure `print digit`, that prints one single digit, to be given.

A positive number less than 10 we can print directly by means of `print digit`; that is a trivially simple case.

From a number consisting of `n` digits, where `n` > 1, we can separate out one digit so that we are left with `n` -1 digits; a simpler case.

We could try to remove this digit at the front of the number but then would have to go through a whole rigmarole with powers of 10 in order to compute `n`, with all kinds of possibilities to obtain overflow; once we have determined `n`, we can obtain the consecutive digits by division by 10^{n-1}, 10^{n-2}, and so on — all in all not very appetizing.

We can also take the digit from the back of the number, by simply taking the number modulo 10. So there we sit, holding in our hand a digit that we can only print after all preceding digits have been printed. How do we print all those preceding digits? Well, in the same way: we now have one digit less than we had initially, and therefore a simpler case of the same problem.

A small complication is the fact that the integer may be negative. In this case we simply print a minus sign, followed by the opposite of the number, that therefore will be positive.

The solution sketched looks as follows:

```
PROC print number (INT CONST number):
   IF number is negative
   THEN
     print minus sign;
     print opposite of number
   ELIF number consists of 1 digit
   THEN
     print that digit
   ELSE
     print the preceding digits;
     print the last digit
   FI.

number is negative:
   number < 0.

print minus sign:
   put("-").

print opposite of number:
   print number(- number).

number consists of 1 digit:
   number < 10.

print that digit:
   print digit(number).

print the preceding digits:
   print number(number DIV 10).
```

```
  print the last digit:
     print digit(number MOD 10).

  ENDPROC print number;
```

Finally we shall give a possible definition for `print digit`:

```
  PROC print digit (INT CONST value):
    IF value < 0 OR value > 9
    THEN put ("?")
    ELSE put ("0123456789" SUB (value + 1))
    FI
  ENDPROC print digit;
```

We now have a procedure for printing an integer in a compact way. The solution found is possibly not very efficient, because the test `number is negative` may be performed unnecessarily often, but we hope its principle is clear. Observe that in the whole procedure not a single variable or assignment appears. This contributes to the clarity of its structure.

In order to get an insight in the working of this procedure, we will trace a call of `print number(123)`.

```
print number(123)
|
|  INT CONST number ::  123
|  print number(123 DIV 10)
|  |
|  |  INT CONST number ::  12
|  |  print number(12 DIV 10)
|  |  |
|  |  |  INT CONST number ::  1
|  |  |  print digit(1)                   prints a "1"
|  |  |
|  |  print digit(12 MOD 10)             prints a "2"
|  |
|  print digit(123 MOD 10)               prints a "3"
|
```

We see that at the high point of activity three instances of `print number` are active. Such an "active instance" of an algorithm, with its own local data, is called an *incarnation* of that algorithm.

A non-recursive formulation of an algorithm, in which repetition is used rather than recursion, is called *iterative.* If we try to give an iterative version of this algorithm — a sketch has already been given — we find out that this is much more cumbersome in its formulation and as a consequence will in the first instance be likely to contain more errors.

The condition `number consists of 1 digit` plays a central role in the termination of the algorithm: when this condition is satisfied no further recursive call is made. We call it the *termination condition* of the algorithm.

Finding a suitable termination condition (the "simplest case") is usually the key to finding a recursive solution.

At the end of this chapter you will find a number of small exercises that have a simply recursive solution, for which you are requested to find the termination condition.

Figure 12.1: Initial situation

12.2 Multiple recursion

We shall now look at a number of algorithms that are *multiply recursive,* in the sense that each incarnation can make more than one recursive call. The first example *(Towers of Hanoi)* is introduced because of its high didactic value, not for its practical importance. In later chapters we will give examples like Quicksort that do show that with the aid of recursion short, clear and efficient solutions for practical problems can be obtained.

12.2.1 Example: The towers of Hanoi

One of the oldest stories in informatics is the legend of the towers of Hanoi, as it was told to the author by Leo Geurts, many years ago.

According to this legend there once stood, long ago, in front of a temple in Hanoi three columns; the first one was made of copper, the second of silver and the third of gold. On the copper column one hundred disks were stacked, which were made of porphyry. From the largest one at the bottom to the smallest one at the top the disks were decreasing in size (Fig. 12.1).

An old monk had set himself the task of carrying the tower of porphyry disks from the copper column to the golden column, one disk at the time, by repeatedly taking the topmost disk from one column and putting it at the top of another column, taking care that a larger disk never landed on top of a smaller one. According to the legend, once the monk had finished his work the end of the world would be at hand.

Soon the monk understood that in this work he had also to make some use of the silver column. He sat down in front of his desk in order to make a plan.

He meditated and thought, thought and meditated, and suddenly obtained enlightenment: he could solve his problem in three steps.

(Step 1)

Transport the tower, consisting of the topmost 99 disks from the copper column to the silver one.

(Step 2)

Transport the last, greatest disk from the copper to the golden column.

(Step 3)

Transport finally the tower of 99 disks from the silver column to the golden one.

In considering this scheme the monk noticed that Steps 1 and 3 would be hardest to perform; and because he was not only an old but also a wise monk, he decided to have these steps performed by his eldest disciple. When the disciple finished the first step, our monk would take it upon himself to carry the largest disk from the copper to the golden column; and then he would once more invoke the services of his eldest disciple.

In order not to overly tax his eldest disciple, he decided to communicate this plan to him in order to simplify his work.

The algorithm that the monk on the next day nailed to the temple door we here translate from the ancient Vietnamese.

Method and way to transport a tower of n disks from one column to another making use of a third column:

In the case where the tower consists of more than one disk, request your eldest disciple to move a tower of the top $n-1$ disks from the first to the third column making use of the other column.

Personally carry first disk from the one to the other column.

In the case where the tower consists of more than one disk, request your eldest disciple to move a tower of $n-1$ disks from the third to the other column making use of the first column.

After the monk had completed the nailing of this document he rested somewhat; and upon waking up, he asked himself what he had to do now: oh yes, he had to move a tower of one hundred disks from the copper column to the golden column making use of the silver column. Because he was somewhat tired after the heavy thinking of the past day he did not remember in detail how to do something like that; but seeing a large knot of people in front of the temple door reading something, he knew what he had to do. He pushed his way up to the temple door and started reading.

And thus he called his eldest disciple to him and requested him to transport the tower of 99 disks from the copper column to the silver one making use of the golden column and upon completion report to him.

Questions:

- What is the first thing that this monk does?
- How many monks will have been put to work before the first disk is actually moved?
- What does our old monk do when his eldest disciple finally reports?

We will now formulate the monk's algorithm in Elan. The columns will be indicated by their names in the form of texts, the disks by their numbers (the biggest disk having the highest number).

```
PROC transport tower
     (INT CONST n, TEXT CONST first,
second, third):
  IF n > 1
  THEN transport tower (n-1, first, third,
second)
  FI;
  move disk (n, first, second);
  IF n > 1
  THEN transport tower (n-1, third,
second, first)
  FI
ENDPROC transport tower;
```

The movement of a disk we will indicate by printing which disk is carried from what column to what column.

```
PROC move disk (INT CONST n, TEXT CONST
from, to):
     line;
     put ("Carry disk"); put (n);
     put (" from the "); put (from);
     put (" column to the "); put (to);
put (" column.")
ENDPROC move disk;
```

We can now solve the original problem by the call:

```
transport tower (100, "copper", "golden",
silver")
```

The procedure `transport tower` contains two directly recursive calls and therefore is multiply recursive. The condition `n > 1` is the negation of the termination condition. A variant in which the termination condition is more explicit but which follows the old Vietnamese original less faithfully is as follows:

```
PROC transport tower
     (INT CONST n, TEXT CONST first,
second, third):
  IF n > 0
  THEN
    transport tower (n-1, first, third,
second);
    move disk (n, first, second);
    transport tower (n-1, third, second,
first)
  FI
ENDPROC transport tower;
```

The recursion now ends at the level $\mathtt{n} = 0$, where nothing remains to be done.

12.2.1.1 Complexity of the algorithm

How often must a disk be moved in order to transport a tower of height n? We call this number S_n.

$$\begin{array}{rrcll} & S_1 & = & 1 & \\ & S_2 & = & 1+2*S_1=3 & \\ \text{in general} & S_n & = & 1+2*S_{n-1} & \text{for} \quad n>1 \end{array}$$

Let us test the hypothesis:

$$\begin{array}{lcl} S_i & \stackrel{?}{=} & 2^i-1 \\ 1+2*S_{i-1} & \stackrel{?}{=} & 2^i-1 \\ 1+2*(2^{i-1}) & \stackrel{?}{=} & 2^i-1 \\ 1+(2^i-2) & \stackrel{?}{=} & 2^i-1 \\ 2^i-1 & \stackrel{?}{=} & 2^i-1 \quad Q.E.D. \end{array}$$

The amount of work therefore grows exponentially with n, the number of disks.

Question: Assuming that all the monks work very hard, so that one disk is moved every second, how long will it take until the end of the world?

12.2.1.2 Example of output

We will complete this example with some output of the algorithm ($\mathtt{n} = 4$):

```
Carry disk 1 from the copper column to the
silver column.
Carry disk 2 from the copper column to the
golden column.
Carry disk 1 from the silver column to the
golden column.
Carry disk 3 from the copper column to the
silver column.
Carry disk 1 from the golden column to the
copper column.
Carry disk 2 from the golden column to the
silver column.
Carry disk 1 from the copper column to the
silver column.
Carry disk 4 from the copper column to the
golden column.
Carry disk 1 from the silver column to the
golden column.
Carry disk 2 from the silver column to the
copper column.
Carry disk 1 from the golden column to the
copper column.
Carry disk 3 from the silver column to the
golden column.
Carry disk 1 from the copper column to the
silver column.
Carry disk 2 from the copper column to the
golden column.
Carry disk 1 from the silver column to the
golden column.
```

In the numbers of the disks moved, a regularity can be found which has a nice relationship to counting in the binary number system.

Figure 12.2: Mouse in maze

12.3 Example: The mouse in the maze

In the previous example we have made use of the fact that every monk can instruct another, namely his eldest disciple, to perform a task that can be done in the same way.

This is an excellent way to understand recursion and not very far from the execution of the algorithm on a machine: every incarnation of the procedure `transport tower` can call upon further incarnations of the same procedure. If, in our model, four monks are busy simultaneously in fulfilling their task, and all but one of them are waiting for another, then there also exist four incarnations of the procedure, of which only the youngest is active.

A difference lies in the fact that the number of monks with disciples, and also the measure of their patience, is in reality severely limited. The number of possible incarnations of a procedure, on the other hand, is practically unlimited: the processor can arbitrarily make new incarnations and forget old ones, as long as some maximum number of simultaneous incarnations is not exceeded (depending on the machine used, the memory space available and details of the implementation of the programming language used), of the order of a few hundreds to many thousands of incarnations.

It is very enlightening to compare the recursive invocation of a procedure to the giving of tasks by one person to another. We will now give a second example, in which again such an anthropomorphic image aids in understanding.

A mouse wants to find a shortest path through a maze, starting at a given point A and ending at another point where lies the cheese. (Notice that there may be more than one shortest path of one same length. Which of the shortest paths we choose is immaterial.)

We shall represent the *maze* as a rectangle of $n*m$ fields (Fig. 12.2). Every field, except for those on which the mouse sits or the cheese lies, is either blocked by a wall or free. We shall indicate the fields by their position (x, y) with $1 \leq x \leq n$ and $1 \leq y \leq m$.

By a *path* of length k we mean a sequence of k fields

Figure 12.3: Mouse in compass

$F_i = (x_i, y_i)$, such that for $1 < i \leq k$ it holds that F_i has exactly one side in common with F_{i-1}:

$$1 < i \leq k \rightarrow$$

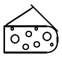
$$x_{i-1} = x_i \text{ AND } y_{i-1} = y_i \pm 1 \text{ OR } x_{i-1} = x_i \pm 1 \text{ AND } y_{i-1} = y_i.$$

These relations describe formally that every field of a path must be connected with its sides to the neighbour fields.

A *loopfree* path is a path such that $i \neq j \rightarrow x_i \neq x_j \; OR \; y_i \neq y_j$.

A *successful* path of length k is a path such that

$$\begin{array}{lcl} (x_1, y_1) & = & \text{A}, \\ (x_i, y_i) & = & \textit{free} \quad (2 \leq i \leq k-1), \\ (x_k, y_k) & = & \end{array}$$



12.3.1 The length of a shortest path

The problem is to find the length of a shortest successful path. Notice that such a path is always loopfree.

One could try to compute the collection of all paths, from those eliminate all paths that are not successful and then choose the shortest one, but this is not very simple to formulate iteratively and costs a forbidding amount of work from the processor.

Idea of a solution: the length of a shortest path starting at a specific field is one more than the minimum of the lengths of the shortest path starting at its neighbour fields.

A mouse, sitting on a specific field (Fig. 12.3), sends from each of its free neighbour fields other mice, one form each field, with the task of finding the length of a shortest path starting at that field. The mouse itself remains waiting at the field (x, y) until the others have completed their task. No other mouse may therefore pass over this field. This guarantees that every path considered is loopfree.

Upon the return of the other mice, the mouse computes the minimum of the values they found, and adds 1 to that (for the distance from the field on which it is sitting), reports that value to the mouse that sent it and leaves the field. Very simple, but it does need a large number of dedicated mice (albeit at most $n * m$).

As termination condition we make use of the fact that, if the cheese lies on a specific field, the path from that field to the cheese has length zero. In the form of an algorithm:

```
IF the field contains cheese
THEN
  0
ELSE
  send from every free neighbour field
  a new mouse with the task of finding
  the length of a shortest path to the
  cheese and take the minimum of the
  lengths thus obtained;
  1 + the minimum
FI
```

The status of a field (`free`, `wall`, `mouse` or `cheese`) we will encode as an integer. For that purpose we declare:

```
LET free   = 1,
    wall   = 2,
    mouse  = 3,
    cheese = 4;
```

The maze will be represented by a row of rows:

```
ROW n ROW m INT VAR maze;
```

The behaviour of one mouse we realize as a procedure that computes the length of a shortest path from `(x, y)` to the goal:

```
INT PROC length of shortest path from (INT
CONST x, y):
   IF already at cheese
   THEN
     0
   ELSE
     take the field;
     find the minimum of four directions;
     release the field;
     1 + minimum
   FI.
```

The elementary operations can be refined:

```
already at the cheese:
   maze[x][y] = cheese.

take the field:
   maze[x][y] := mouse.

release the field:
   maze[x][y] := free.
```

Observe that the procedure given is not robust against a call with a field that already contains a mouse or wall; we shall have to test for that before calling the procedure.

We shall have to compute a path length even in the case when the mouse gets into a blind alley. In such a case we choose as path length "infinite", that is to say: greater than any meaningful path length. For that we do not take `maxint` (why not?), but for example:

```
INT CONST infinite :: n * m;
```

If there is no path from the mouse to the cheese, the length of the path will be at least `infinite`. We can now refine

```
find the minimum of four directions:
   INT VAR minimum :: infinite;
   a mouse to the north;
   a mouse to the south;
   a mouse to the west;
   a mouse to the east.
```

(By the way: any resemblance to T.S. Eliot's "Choruses from The Rock" is purely accidental.)

The order of the last four units in fact plays no role, but our sequential programming language forces us to fix a specific order. (Observe that allowing the mice to proceed in parallel would open the door to endless confusion.)

We define two auxiliary procedures that obviate the need for many similar refinements

```
PROC shortest (INT VAR min, INT CONST
term):
     IF min > term THEN min := term FI
ENDPROC shortest;

BOOL PROC can go to (INT CONST x, y):
     IF within maze
     THEN maze[x][y] = free OR maze[x][y]
= cheese
     ELSE false
     FI.

within maze:
     1 <= x AND x <= n AND 1 <= y AND y <=
m.
ENDPROC can go to;
```

Observe that we can only test whether a field is free or contains cheese if we are sure that it lies within the maze, so we cannot just write

```
within maze AND (maze[x][y] = free OR
maze[x][y] = cheese)
```

because the subscription `maze[x][y]` is meaningless for indices `x` and `y` outside the maze.

We program very careful mice, e.g.:

```
a mouse to the north:
   IF can go to (x, y+1)
   THEN
     shortest(minimum,
              length of shortest path
from(x, y+1))
   FI.
```

and analogously for other directions. We finish the procedure with

```
ENDPROC length of shortest path from;
```

The resulting program is somewhat boring because of its repetitive character but not badly structured. Still it has a shortcoming.

12.3.2 Shortest path — an alternative approach

What are the preconditions under which `length of shortest path from (x,y)` can be called? The following must be fulfilled:

- the procedure may only be called with $x \in [1:n]$ and $y \in [1:m]$, otherwise a subscription error occurs;
- the field `maze[x][y]` must contain `cheese` or be `free`. We may for instance not start in a `wall`.

These conditions have to be tested within the procedure at every recursive call. But the same holds for the initial call that occurs outside the procedure. Of course it is all too easy to forget such tests.

It is much wiser to perform the test immediately upon entry of the procedure, thereby making it robust against misuse. All the tests before sending a new mouse are now superfluous: the mice are clever enough to perform the testing as necessary.

With these modifications (and making use of a numerical choice — see section 7.4.2) we obtain:

```
INT PROC length of shortest path via (INT
CONST x, y):
  IF NOT within maze
  THEN infinite
  ELSE
    SELECT maze[x][y] OF
    CASE free  :
            take the field;
            find the minimum of the four
directions;
            release the field;
            1 + minimum
    CASE wall  : infinite
    CASE mouse : infinite
    CASE cheese: 0
    ENDSELECT
  FI.

take the field:
     maze[x][y] := mouse.

find the minimum of the four directions:
   INT VAR min :: length of shortest path
via (x+1, y);
   shortest (min, length of shortest path
via (x-1, y));
   shortest (min, length of shortest path
via (x, y+1));
   shortest (min, length of shortest path
via (x, y-1)).

release the field:
     maze[x][y] := free.

within maze:
   1 <= x AND x <= n AND 1 <= y AND y <=
m.
```

The remaining procedures stay the same. Again we finish the procedure with:

```
ENDPROC length of shortest path via;
```

Observe that the procedures given above may make temporary modifications to the maze by means of `take the field`, but that upon their return the maze is always left in its original state. They remove, as it were, their own garbage.

The two approaches `length of shortest path via` and `length of shortest path from` differ in the sense that the first one only works subject to stringent preconditions, whereas the second one is robust. It is good practice to prefer testing at the entry of a procedure to testing at each of its calls.

Of course we should have programmed a robust version of this algorithm right from the start. The second version is shorter, clearer, and gives less opportunity for errors in programming. But beginners especially have a curious preference for solutions like the first one. It is very hard for human beings to find obvious and simple solutions to their problems.

12.4 Conclusion

In a number of ways, a good understanding of recursion leads to a deeper insight into programming.

- Many problems naturally admit a constructive recursive definition that can be used as a first rough formulation of the algorithm.
- Recursion is the most economical form of refinement: a recursive algorithm is also its own refinement.
- Recursion leads to an economical form of thinking. A large problem (for example the analysis of a maze) is reduced to a smaller local problem (the analysis of a square in the maze) plus the original problem in a simplified form. Once the local problem has been solved correctly, a correct induction scheme leads to a correct solution for the global problem.
- Recursion leads to an economic form of proof of correctness and termination, because only the correctness of the solution of the local problem and the induction step have to be proved.
- Recursion leads to a simple form of administration of intermediate results, that can be kept in local variables and manipulated without interfering with global objects.
- Recursion leads to short concise programs, that are easier to overview than equivalent iterative solutions and therefore lead to fewer errors.

Of course one pays a price for these advantages: in most implementations a recursive program is executed somewhat slower than an equivalent iterative program. Furthermore, some old-fashioned programming languages do not admit recursion.

In a following chapter we shall take a look at techniques for deducing, starting from a correct recursive algorithm, an equivalent iterative algorithm, while retaining correctness and generally raising efficiency. A number of important algorithms have been found in this way. A simple-minded iterative thinker would probably never have found them.

12.5 Exercises

Find termination conditions for the following formulations and program the suggested simply recursive procedures.

1. Let there be given `ROW n INT CONST row`. The maximum of the n elements of the `row` is the maximum of the nth element and the maximum of the preceding elements.

2. Let be given a `ROW n INT CONST d`. The value of a decimal number, represented by $d_n d_{n-1} \ldots d_1 d_0$ (with $0 \leq d_i \leq 9$), is the value of the last digit d_0 plus ten times the value of the decimal number $d_n d_{n-1} \ldots d_1$.

3. Again `ROW n INT CONST d` is given. The value of a decimal fraction, represented by $0.d_0 d_1 \ldots d_n$, is one tenth times the sum of the values of the leading digit d_0 and the decimal fraction $0.d_1 \ldots d_n$.

4. Given is the `TEXT CONST a`. The row of characters $a_1 a_2 \ldots a_{n-1} a_n$ is symmetric, if
   - a_1 is equal to a_n and
   - the row $a_2 \ldots a_{n-1}$ is symmetric.

5. a to the power n is a times (a to the power $(n-1)$).

6. Given is a `ROW n BOOL CONST row`. The parity of the `row` is by definition `TRUE` if the `row` contains an even number of `TRUE` elements, and `FALSE` if the `row` contains an odd number of `TRUE` elements. Write a procedure

   ```
   BOOL PROC parity (INT CONST k)
   ```

   that recursively computes the parity of the first `k` elements of `row`.

Now follow a number of multiply recursive algorithms.

7. (Subdividing a line) In how many ways can a line of length n be subdivided into pieces of length 1 and 2? (Hint: there are two kinds of divisions, those starting with a piece of length 1 and those starting with a piece of length 2).

8. (Mouse in maze, again) The test whether we are still within the maze can be omitted if we surround the maze by a wall.

```
ROW n ROW m INT VAR maze;
initialize border as wall
```

Rewrite the algorithm accordingly.

9. Write a procedure with the heading

```
PROC print all paths from (INT CONST x,
y):
```

that prints all loopfree paths, starting at `(x, y)` and ending at the cheese. (Hint: At the moment the cheese is reached a loopfree chain of mice lies in the maze from the starting point to the cheese).

10. (Tabular) Print a 10-column table containing the natural numbers up to 100. Numbers divisible by 7 and numbers in which the sum of the digits is divisible by 7 must be replaced by `*****`.

    Hint: In order to print 10 numbers in a single line the conversion procedure `text(INT CONST i, width)` should be used. The new-line-required and divisible-by-7 tests can be accomplished by using the `MOD`-operator. You have also to separate the digits: give first an iterative and then a recursive solution.

Chapter 13

Computer graphics

In this chapter we will first introduce some concepts and terminology from Computer Graphics and a set of algoritms for Comuter Graphics in Elan. Then we will discuss some recursive picture-drawing algorithms.

Computer graphics means the processing and presentation of visual information by means of the computer. The technology for processing and presenting data in graphical form is developing fast. It plays an increasingly important role in realising user-friendly man-machine interfaces.

Drawing pictures on the computer screen is an attractive exercise in systematic programming. In particular there exist beautiful families of recursive drawings that provide good examples of the design of recursive algorithms. For a more comprehensive study of computer graphics various textbooks are available (eg. [FOD84]). Graphics software packages are also good examples to investigate the modular and layerwise structure of large programs, as we shall see in volume 2.

13.1 The physical layer

The assortment of equipment for graphical input and output is very large. We only enumerate a number of graphical output devices: plotters to make drawings on paper with a mechanically controlled pen; cathode ray tube (CRT) screens applying the TV principle: pictures are composed of horizontal lines which, in their turn, are composed of either black-and-white or coloured points — this method is called "raster graphics"; CRT screens controlled by separate processor(s) that change the pictures autonomously, with lightning speed, or their miserly elder brothers in which a single processor supports both picture generation and computing, with the consequence that the picture disappears during computing. There is a plethora of devices for sale — only the 3-dimensional holographic colour plotter is still music of the future.

We shall restrict ourselves to the graphics display with which most microcomputers are equipped. The *screen* is a rectangle composed of points, so called *pixels* (= picture elements). Each pixel can take on a colour from a fixed palette (black or white; a shade of grey; or perhaps one of 100000 Japanese colours). We restrict ourselves further to monochrome displays that allow black and white only.

The number of pixels in a horizontal (vertical) line is called the horizontal (resp. vertical) *resolution.* For microcomputers, a common resolution is 720x348, but large variations can be observed. With the resolution mentioned the structure of the lines composed of pixels can still be distinguished from quite near but from some distance the dots can not be recognised.

On such a display it is very easy to draw horizontal or vertical lines. However, in order to draw sloping lines (see Fig. 13.1) some smart approximation algorithm (e.g. that of Bresenham, see [FOD84]) will be required.

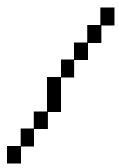


Figure 13.1: Sloping line as drawn by the Bresenham algorithm

The situation is often complicated by the fact that many CRT screens use special hardware support (a "character generator") to display texts. Therefore text output can not be alternated with graphics — first the screen must be switched into another operating mode. It is as though you had two screens, a text screen and a graphics screen, between which you have to switch.

A further complication may be that the height of a pixel usually differs from its breadth. A square, composed of the same number of pixels in both directions, may appear flattened on the screen. The ratio between the vertical and horizontal unit of lenght is called the *aspect.* In order to avoid distortions the aspect must be taken into account.

13.2 Integer graphics

The lowest layer of graphics operations is called *integer graphics* because the screen is addressed by integer coordinates. It forms a uniform interface to the graphics hardware, which itself varies from machine to machine, and thus consistent behaviour of programs is guaranteed.

Integer graphics is a (machine dependent) packet which is part of the standard library of each Elan Programming Environment. We describe here what it delivers to the user without telling how it is implemented.

Before using any of its operations you should initialise some hidden variables and prepare the screen by calling the procedure

```
PROC enter graphics mode:
   { clear the screen and switch to
graphics mode }
ENDPROC enter graphics mode;
```

Similarly, you should finally leave graphics mode by calling

```
PROC enter text mode:
   { clear the screen and switch to text
mode }
ENDPROC enter text mode;
```

The drawing always occurs in a given position inside the area

$$[1 : \texttt{graphics x limit}, 1 : \texttt{graphics y limit}]$$

which represents the graphics screen.

```
LET graphics x limit = 720,
    graphics y limit = 348,
    aspect = 1.35;
```

These values, characterising the Hercules board of the IBM PC, are machine dependent; the aspect 1.35 means the pixels are higher than wide.

The current screen position may be modified through the effect of the various graphics actions. When the graphics mode is switched on, the initial value of the current screen position (the so-called *reference point)* is $(1, 1)$, which corresponds to the *upper left corner.*

The current position can be changed by the call

```
PROC move (INT CONST x, y):
   { go to position (x, y) }
ENDPROC move;
```

Whenever either `x` or `y`, or both, are outside the machine dependent boundaries of the screen, a beep is sounded and the coordinate concerned takes on one of the boundary values given above.

It is possible to draw something on the screen by moving the screen position as desired and calling the procedure

```
PROC plot pixel:
   { plot the pixel in the current
position }
ENDPROC plot pixel;
```

The plotting of a pixel has an effect which depends on the colour of the "brush". We may choose a colour by calling the procedure

```
PROC color (INT CONST c):
   { use, until further notice, colour
number 'c' }
ENDPROC color;
```

The numbering of colours is highly machine dependent. If you do not have a handbook describing this correspondence for your machine perhaps you better keep yourself away. After the first call to `enter graphics mode` the brush is white (or amber or light green) and the background colour of the screen is black.

By means of these procedures all possible capers can be performed on the screen although their usage is not at all simple because of the tedious details. Therefore, another procedure is added for drawing lines. It plots all the necessary pixels according to the Bresenham algorithm.

```
PROC draw (INT CONST x1, y1):
   { draw a line as straight as possible }
   { from the current position to (x1, y1)
}
   { and plot all pixels en route }
ENDPROC draw;
```

The usual procedures `get`, `put` and `line` for input and output work also in graphics mode, but the type font used may differ from that in text mode: In graphics mode the characters are also drawn laboriously pixel by pixel, instead of by a "character generator".

As you might imagine, each character is composed of pixels arranged as a matrix (see Fig. 13.2). Both horizontally and vertically a gap is needed between two neighbouring letters in order to maintain readability. The space used to display any character, together with its surrounding gaps, is constant but machine dependent; its height is given by `line height`, its width by `character width`. (On the IBM PC, for example, `line height` is 11 and `character width` is 8.) The reference point of a character is its *upper left corner.*

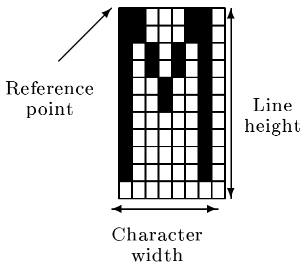


Figure 13.2: Letter M in a character matrix

Of course, as a side effect, the current position is always updated by `draw`. It is stored in a hidden variable (or, more probable, in a pair of hidden variables) in the packet realising the graphics interface.

The procedure `line` can also be called in graphics mode in order to perform a transition to the beginning of the next "line" on the graphics screen, going down by `line height` pixels and to the leftmost pixel in the line.

Starting in position `(x, y)`, the effect of this procedure is the same as that of `move (1, y + line height)`.

In the sequel, a few example programs will be discussed that draw on the graphics screen. In order to write portable programs we try to be careful in using relative coordinates in terms of `graphics x limit` and `graphics y limit`.

13.2.1 Useful auxiliary procedures

A call to `enter text mode`, or its alternative form `leave graphics mode` declared as

```
PROC leave graphics mode:
   enter text mode
ENDPROC leave graphics mode;
```

immediately clears the graphics screen. We may introduce a procedure `wait for confirmation` which waits for the user to hit the space bar, thus giving him the necessary time to study the picture.

This procedure has two parameters: the integer coordinates of a place on the screen where you want the warning `Hit space!` to appear.

```
PROC wait for confirmation (INT CONST x,
y):
   move (x, y);
   put ("Hit space!");
   TEXT CONST t:: inchar
ENDPROC wait for confirmation;
```

Notice that any character is accepted, not only the space.

A group of procedures which are very useful in the input dialogue for numbers and texts can be declared as follows:

```
INT PROC ask int (TEXT CONST message):
   INT VAR x;
   put (message);
   get (x);
   x
ENDPROC ask int;

REAL PROC ask real (TEXT CONST message):
   REAL VAR x;
   put (message);
   get (x);
   x
ENDPROC ask real;

TEXT PROC ask text (TEXT CONST message):
   TEXT VAR t;
   put (message);
   get (t);
   t
ENDPROC ask text;
```

The trigonometric functions of the standard library assume the angle as a real parameter given in radians. In integer graphics it is more natural to compute the angle as an integer given in degrees. To that end we declare:

```
REAL PROC sin (INT CONST a):
   sin (pi * real (a) / 180.0)
ENDPROC sin;

REAL PROC cos (INT CONST a):
   cos (pi * real (a) / 180.0)
ENDPROC cos;
```

13.2.2 Example: Radar

The first program draws a number of lines of one same length, springing from the approximate centre of the screen. In the circular pattern created in this way the effect of the line drawing algorithm applied is clearly recognisable. Notice the nice interference of lines in Fig. 13.3.

```
radar:
   enter graphics mode;
   print heading;
   determine parameters;
   draw radar;
   wait for confirmation (2 * graphics x
limit DIV 3, 1);
   leave graphics mode.

print heading:
   put ("Radar");
   line;
   put ("=====").

determine parameters:
   REAL CONST radius::
     real (min (graphics x limit, graphics
y limit)) / 2.1;
   INT CONST centre x:: graphics x limit
DIV 2,
             centre y:: graphics y limit
DIV 2.

draw radar:
   INT VAR i;
   FOR i FROM 0 UPTO 359
   REP
     move (centre x, centre y);
     draw (centre x + round (aspect * sin
(i) * radius),
           centre y + round (cos (i) *
radius))
   ENDREP.
```

13.2.3 Example: Moving radar

The next program example draws similar lines, but this time springing from a moving centre.

```
moving radar:
   enter graphics mode;
   print heading;
   determine parameters;
   draw moving radar;
   wait for confirmation (2 * graphics x
limit DIV 3, 1);
   leave graphics mode.
```

Figure 13.3: Radar

```
print heading:
   put ("Moving radar");
   line;
   put ("============");
   line.

determine parameters:
   REAL VAR radius:: ask real ("Radius?
");
   radius:= min (radius,
                  real (min (graphics x
limit,
                             graphics y
limit)) / 2.1);
   put ("Radius used: ");
   put (text (int (radius), 3));
   REAL CONST step x::
      min (1.0, real (graphics x limit) /
360.0);
   REAL VAR centre x:: max (0.0, radius -
260.0);
   INT CONST centre y:: graphics y limit -
int (radius).

draw moving radar:
   INT VAR i;
   FOR i FROM 0 UPTO 360
   REP
     move (round (centre x), centre y);
     draw (round (centre x + aspect * sin
(i) * radius),
           centre y + round (cos (i) *
radius));
     centre x INCR step x
   ENDREP.
```

In order to allow the studying of various interference patterns the program asks the user for the radius to be

Figure 13.4: Moving radar

used. The program limits this value so that the drawing always remains within the graphics screen. The experimental value 2.1 is used to leave enough space for messages.

Like in the previous example we want to make a whole turn of 360 degrees, which means that we also have to take 360 steps horizontally, across the screen. On some screens — and also in the laser printer that produced this book — `graphics x limit` is less than 360. To that end, `step x` is introduced. It has the value 1.0, or less if necessary.

The coordinates of the centre are stored in `centre x` and `centre y`. The former is a *real variable* since the centre must drift horizontally, in steps which may be less than 1. Initially, `centre x` contains either 0.0 or — for large radii — a value that keeps the drawing within the graphics screen. The somewhat arbitrary, experimental value of 260.0 is motivated by the fact that — for large radii — the leftmost point is drawn cca. in step 260. The initial value of `centre y` leaves enough space for the messages at the top, even in case of large radii.

The loop body is similar to the previous one. Notice the upper limit of the loop variable is 360 in the present case.

13.2.4 Example: Mondriaan

As a last example, let us see how pictures reminiscent of the paintings of Mondriaan can be generated by means of integer graphics.

Please do not consider this as a serious contribution

Figure 13.5: A Mondriaan?

to computer art, although ithelps you to imagine how some artists use the computer as a working tool. Computer art makes extensive use of random numbers, allowing the artist to study the fringe area between chaos and order.

```
make a famous painting:
   prepare the canvas;
   stain it diligently;
   painting is done.
```

```
prepare the canvas:
   enter graphics mode;
   INT CONST xmin:: graphics x limit DIV
10,
             ymin:: graphics y limit DIV
10,
             xmax:: graphics x limit -
xmin,
             ymax:: graphics y limit -
ymin;
   move (xmin, ymin);
   draw (xmin, ymax);
   draw (xmax, ymax);
   draw (xmax, ymin);
   draw (xmin, ymin).
```

```
stain it diligently:
   INT VAR k, l, x, y;
   FOR k FROM 1 UPTO 150
   REP
     x:= random (2 * xmin, xmax - xmin);
     y:= random (2 * ymin, ymax - ymin);
     l:= int (real (ymin) * (1.0 - sqrt
(random)));
     IF random > 0.5
     THEN
       move (x - l, y);
       draw (x + l, y)
     ELSE
       move (x, y - l);
       draw (x, y + l)
     FI
   ENDREP.
```

```
painting is finished:
   wait for confirmation (5, 5);
   leave graphics mode.
```

In conjunction with the resulting picture, this simple program should need no further explanation.

13.3 Turtle graphics

Another packet in the standard library builds a layer on top of the integer graphics that simplifies graphics programming. The packet is inspired by Karel the Robot (see chapter 2) and the turtle graphics of Logo.

In the first place, relative coordinates will be used instead of absolute ones. After all, we want mostly to draw regiments of lines. A figure is regarded as a sequence of (visible or invisible) lines connected to each other. The notion of *current position* (the "graphics cursor") and *current angle* are also introduced. A piece of line always starts in the current position and extends in the direction determined by the current angle; only the length must be given explicitly (see Fig. 13.6).

In the second place, integer coordinates are replaced by real ones. It is much easier to round off an accurately computed real position to an integer afterwards than to hold it continuously as an integer when (because of skew slope directions) all kinds of rounding errors may occur.

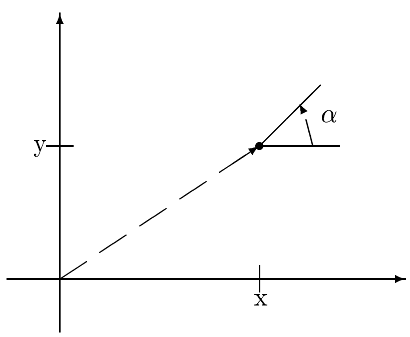


Figure 13.6: Current position (x, y) and current angle α

13.3.1 The turtle graphics interface

Two line drawing algorithms are introduced: `move` for invisible lines and `draw` for visible ones.

```
PROC move (REAL CONST l):
   { draw an invisible line of length 'l'
in the }
   { direction determined by the current
angle }
   { and starting in the current position
}
ENDPROC move;

PROC draw (REAL CONST l):
   { draw a visible line of length 'l' in
the }
   { direction determined by the current
angle }
   { and starting in the current position
}
ENDPROC draw;
```

After a call of these, the endpoint of the line server as a new current position. In order to steer the lines we must be able to change the value of the current angle, for example, by

```
PROC turn (INT CONST angle):
   { add 'angle' degrees to the current
angle }
ENDPROC turn;
```

Recall from mathematics that the counter-clockwise direction is the positive one. For the special cases of turning 90 degrees to the left and right two faster procedures are introduced:

```
PROC turn left:
   { add 90 degrees to the current angle }
ENDPROC turn left;

PROC turn right:
   { subtract 90 degrees from the current
angle }
ENDPROC turn right;
```

In addition to the relative moves (similar to that of Karel the Robot going after his nose) two more line drawing procedures are defined that use absolute coordinates:

```
PROC move (REAL CONST x, y):
   { go to position (x, y) }
ENDPROC move;

PROC draw (REAL CONST x, y):
   { draw a line from the current position
to (x, y) }
ENDPROC draw;
```

We may arbitrarily choose the units to be used with distances. To that end, two constants `turtle x limit` and `turtle y limit` are declared within the packet, the value of which may be changed by the user.

To switch the graphics screen to and from turtle graphics mode two actions are declared: `enter turtle graphics`, `leave turtle graphics`.

13.3.2 Example: Drawing a rosette

Let us give one (not very ambitious) example of turtle graphics: we shall draw a rosette.

Figure 13.7: Rosette

The rosette consists of six equilateral triangles, with each subsequent triangle turned by 60 degrees. The drawing starts in the centre of the screen and the edges are equal to one third of the screen length.

```
draw a rosette:
   enter turtle graphics;
   go to the centre;
   UPTO 6
   REP draw equilateral hook;
       turn (60)
   ENDREP;
   wait for confirmation (10, 10);
   leave turtle graphics.

go to the centre:
   REAL CONST d:: turtle y limit / 3.0;
   move (turtle x limit / 2.0, turtle y
limit / 2.0).

draw equilateral hook:
   draw (d); turn (120);
   draw (d); turn (120);
   move (d); turn (120).
```

In the next section you will find some more complicated applications of turtle graphics.

13.4 Recursive drawing

The following examples of recursive drawings, which involve multiple recursion, are certainly not simple, but they serve well to demonstrate the power and elegance of recursion.

13.4.1 Example: Peano curves

The number theorist Peano not only gave his name to the axiomatic treatment of the natural numbers, but he was also concerned with the continuum, the real numbers. He considered, for instance, continuous mappings from the line $[0:1]$ to the square $[0:1, 0:1]$.

For this purpose he defined families of continuous curves, composed of straight lines, with the property that the nth member of such a family passes each point of the square at a distance of at most 2^{-n}. A trivial example of such a curve is shown in Fig. 13.8.

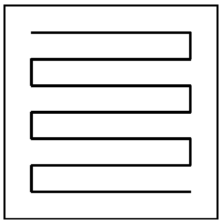


Figure 13.8: A simple Peano curve

There exist much more beautiful families of such curves. In this section we introduce an algorithm by Aad van Wijngaarden that computes and draws such a family of Peano approximations (which was first described by Hilbert). We define a family of Peano-approximahts, as follows. The zeroth member of the family has to pass each point of the unit square at a distance of at most 2^{-0}. This condition is met by a point in the middle of the square (Fig. 13.9).

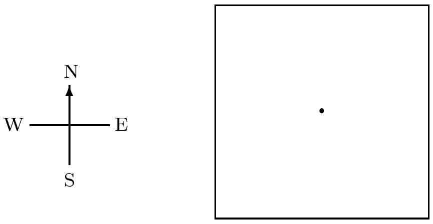


Figure 13.9: Peano approximation 0

As a next approximation (the first member of the family) we divide the square into four squares with a side of 1/2 each. In these smaller squares we use the previous approximation and join the small drawings by means of three line pieces of length 1/2, the first one in an easternly direction, the second to the north and the third to the west (Fig. 13.10).
This is an approximation with orientation north, because the net movement is to the north. The four possible orientations for $n = 1$ are named as indicated in Fig. 13.11.

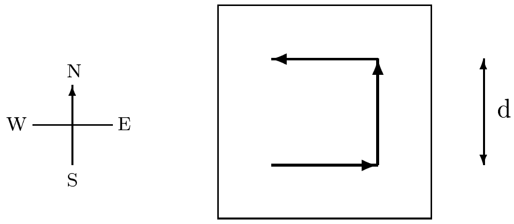


Figure 13.10: Peano approximation 1

Our algorithm is based on the idea of constructing the nth approximation from four smaller $(n-1)$th approximations, joined together by means of three pieces of length 2^{-n}, with carefully chosen orientations. We show, as an example, an approximation of order 2 with orientation north in Fig. 13.12.

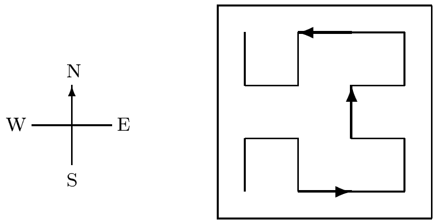


Figure 13.12: Peano approximation 2

A Peano approximation of order n with orientation north starts at the point $(2^{-(n+1)}, 2^{-(n+1)})$ and ends, after a sweep through the unit square, in the point $(2^{-(n+1)}, 1-2^{-(n+1)})$.

For example, the approximation of order $n = 2$, as shown in Fig. 13.12, consists of:

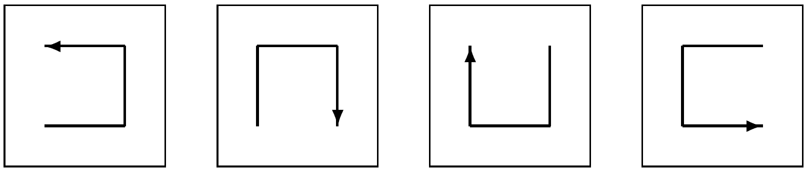
```
approximation east (1);
line segment east;
approximation north (1);
line segment north;
approximation north (1);
line segment west;
approximation west (1)
```

We can now express the drawing of an approximation of order n in the form of four mutually recursive procedures:

```
PROC approximation north (INT CONST n):
     IF n <> 0
     THEN approximation east (n-1);
          line segment east;
          approximation north(n-1);
          line segment north;
          approximation north (n-1);
          line segment west;
          approximation west (n-1)
     FI
ENDPROC approximation north;
```

In the case where `n` $= 0$ only a point has to be drawn; but because such a point always lies in the crossing of two line segments nothing has to be done.

All line segments occurring in the drawing have one same length, $d = 2^{-n}$.



N E W S

Figure 13.11: Four orientations

```
PROC approximation east (INT CONST n):
     IF n <> 0
     THEN approximation north (n-1);
          line segment north;
          approximation east (n-1);
          line segment east;
          approximation east (n-1);
          line segment south;
          approximation south (n-1)
     FI
ENDPROC approximation east;

PROC approximation south (INT CONST n):
     IF n <> 0
     THEN approximation west (n-1);
          line segment west;
          approximation south (n-1);
          line segment south;
          approximation south (n-1);
          line segment east;
          approximation east (n-1)
     FI
ENDPROC approximation south;

PROC approximation west (INT CONST n):
     IF n <> 0
     THEN approximation south (n-1);
          line segment south;
          approximation west (n-1);
          line segment west;
          approximation west (n-1);
          line segment north;
          approximation north (n-1)
     FI
ENDPROC approximation west;
```

All line segments occurring in the drawing have one same length, $d = 2^{-n}$.

The current position (in the unit square $[0:1, 0:1]$) we shall administer in a couple of variables `REAL VAR x, y`. In order to draw line segments of length `d` we introduce four procedures:

```
PROC line segment north:
   draw (x, y + d)
ENDPROC line segment north;

PROC line segment south:
   draw (x, y - d)
ENDPROC line segment south;

PROC line segment east:
   draw (x + d, y)
ENDPROC line segment east;

PROC line segment west:
   draw (x - d, y)
ENDPROC line segment west;
```

To put this whole machinery into movement, the graphics cursor has to be positioned at the starting point, e.g. `(d/2.0, d/2.0)`, on the screen. The main program can be:

```
program:
   ask order;
   draw curve;
   end program.

ask order:
   enter graphics mode;
   INT VAR n:: ask int ("Peano curve of
order? ").
```

So far, we have concerned only with unit length but now the real size of the graphics screen must be taken into account. The solution is based on *integer graphics.* Let us, then, consider the shorter side of the integer graphics screen to be of unit length; for this purpose a multiplying factor, `scale`, is introduced .

```
draw curve:
   REAL VAR x, y;
   REAL CONST
      d:: 1.0 / 2.0 ** n,
      scale:: real (min (graphics x limit,
                         graphics y limit
- line height));
   move (d / 2.0, d / 2.0);
   approximation east (n).

end program:
   wait for confirmation (graphics x limit
DIV 2, 1);
   leave graphics mode.
```

In the program two procedures, `draw` and `move`, expecting real parameters are used. We shall define them in terms of their integer counterparts. The scaling and the aspect of the pixels will be taken account here, too. And one more thing: the starting point of the curves. If position $(0.0, 0.0)$ has to denote the lower left corner the drawing must be mirrored vertically.

```
PROC draw (REAL CONST xp, yp):
   draw (round (scale * xp * aspect),
         graphics y limit - round (scale *
yp));
   x:= xp;
   y:= yp
ENDPROC draw;

PROC move (REAL CONST xp, yp):
   move (round (scale * xp * aspect),
         graphics y limit - round (scale *
yp));
   x:= xp;
   y:= yp
ENDPROC move;
```

Finally, in Fig. 13.13 we show two drawings that were generated in this way.

Some questions:

- Why must the approximations start and end in the corners of the squares?

- The approximation north and south we have chosen to rotate counter clockwise, whereas west and east rotate clockwise. Is it possible to find a solution where one approximation can be obtained from another by rotation?

- How does the complexity, expressed as the total number of procedure calls or draw calls, depend on n?

- Notice that the recursion in this algorithm ends at the level where nothing remains to be done; this style is typical for Van Wijngaarden. It is more usual to end the recursion at the level where the implementation is sufficiently trivial to do it at one stroke. What do we have to change in order to let the recursion end at $n = 1$? What effect does this have on the number of procedure calls?

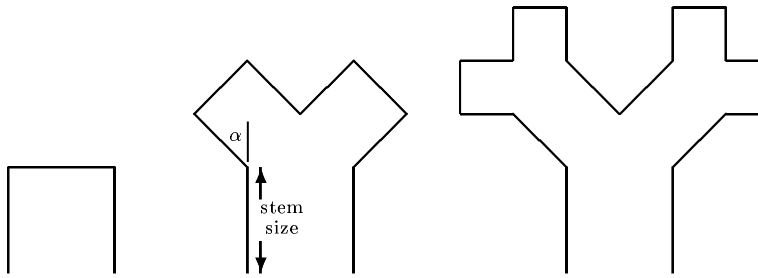
13.4.2 Example: Pythagoras tree

A Pythagoras tree (of order n) is a recursive drawing containing two Pythagoras trees (of order $n-1$). Fig. 13.14 shows three Pythagoras trees of order 0, 1 and 2. A Pythagoras tree has a number of parameters: the stem size, the angle α of its left subtree and the depth of recursion determining the shape of the tree.

It is quite a puzzle to derive a specific algorithm out of this general scheme. Moreover, it is worth the trouble to keep the number of calls of `turn` low since on a microcomputer the computation of sine and cosine consumes a lot of time. Where possible the (faster) `turn left` and `turn right` will be used.

The following solution, based on *turtle graphics*, assumes that the current angle initially points in the growth direction of the tree and after termination in the opposite direction. With these conventions the number of rotations remains restricted.

Figure 13.13: Peano curves



Order 0 Order 1 Order 2

Figure 13.14: Low-order Pythagoras trees ($\alpha = 45^o$)

Figure 13.15: Pythagoras tree of order 8

```
PROC pythagoras tree (REAL CONST size,
                      INT CONST depth,
angle):
   draw (size);
   IF resolution achieved
   THEN
     turn right;
     draw (size);
     turn right
   ELSE
     turn (angle);
     pythagoras tree (size * cos(angle),
depth - 1, angle);
     turn left;
     pythagoras tree (size * sin(angle),
depth - 1, angle);
     turn (90 - a)
   FI;
   draw (size).


   resolution achieved:
   depth = 0.


ENDPROC pythagoras tree;


program:
   enter turtle graphics;
   move (graphics x limit DIV 2 - 60, line
height);
   put ("Pythagoras tree");
   line;
   ask parameters;
   draw frame;
   draw pythagoras tree;
   wait for confirmation (1,
                          graphics y limit
- line height);
   leave turtle graphics.

  draw frame:
    move (0.0, 0.0);
    draw (0.0, 100.0);
    draw (100.0, 100.0);
    draw (100.0, 0.0);
    draw (0.0, 0.0).

  ask parameters:
     INT CONST tree depth:: ask int ("Depth?
");
     INT CONST tree angle:: ask int ("Angle?
");
     REAL CONST tree size:: ask real ("Size
? ");
     REAL CONST tree xpos:: ask real ("Xpos
? ");
     REAL CONST tree ypos:: ask real ("Ypos
? ").
```

`tree xpos` and `tree ypos` determine the position of the lower left corner of the stem. By assigning appropriate values to these constants you may choose a nice layout for your drawing.

```
  draw pythagoras tree:
     move (tree xpos, tree ypos);
     pythagoras tree (tree size, tree depth,
  tree angle).
```

It is quite interesting to try different values for the angle α and the recursion depth. Sometimes a whole cauliflower grows out of the screen as e.g. in Fig. 13.15. In other cases, especially with angles not far from 90^o, a large value must be supplied for `depth` otherwise the drawing will be uninteresting. However, as you know, the deeper the recursion the slower the execution. Fortunately, deeply in the recursion the stem size becomes so small that it can no more be displayed — and therefore we can use it as an alternative stop condition.

```
    resolution achieved:
      depth = 0 OR size < 0.1 * tree size.
```

This restriction ensures that no tree can be smaller than the pixel size. This will result in a better balanced tree with regards to the stem size of the subtrees.

Fig. 13.16 shows two distorted Pythagoras trees where the second variant of the test `resolution achieved` was used.

13.4.3 Example: Peano curves revisited

Based on turtle graphics, the program drawing Peano curves (see section 13.4.1) can be made shorter. The following version exploits the observation that Peano curves may have only two essentially different orientations: turned to the left or to the right. Each case will be covered by a recursive procedure. Notice that a Peano curve is constructed entirely from small, straight pieces of lines of a fixed length `d`. This piece will be drawn by the procedure `connect`.

Figure 13.16: Distorted Pythagoras trees

```
PROC connect:
   draw (side)
ENDPROC connect;

PROC peano right (INT CONST n):
   IF n <> 0
   THEN
     turn left;
     peano left (n - 1);
     connect;
     turn right;
     peano right (n - 1);
     connect;
     peano right (n - 1);
     turn right;
     connect;
     peano left (n - 1);
     turn left
   FI
ENDPROC peano right;

PROC peano left (INT CONST n):
   IF n <> 0
   THEN
     turn right;
     peano right (n - 1);
     connect;
     turn left;
     peano left (n - 1);
     connect;
     peano left (n - 1);
     turn left;
     connect;
     peano right (n - 1);
     turn right
   FI
ENDPROC peano left;

program:
   ask order;
   draw curve;
   end program.

ask order:
   enter turtle graphics;
   move (1, 1);
   INT VAR order:: ask int ("Peano curve
of order? ").

draw curve:
   REAL CONST limit:: min(turtle x limit,
turtle y limit);
   REAL CONST side:: limit * 2.0 ** -
order;
   turn right;
   peano right (order).

end program:
   wait for confirmation (graphics x limit
DIV 2, 1);
   leave turtle graphics.
```

13.5 Exercises

1. (trembling line) Write an algorithm two draw a trembling line between two points.
2. (snow) Draw a number of snow-flakes. Exploit the fact that a snow-flake is either 6- or 12-fold symmetric. Use reflection and rotation.
3. (cube) Draw a perspective picture of a cube, using turtle graphics, in which the near lines are twofold shifted or turned (in order to gain virtual depth).

Chapter 14

Recursive sorting

By sorting a given row $f(i)$,$1 \leq i \leq n$, with respect to a given ordering relation $\leq$ we mean the problem of finding a row $f'(i)$ consisting of the same elements as the row $f(i)$ such that:

$$1 \leq i < j \leq n \rightarrow f'(i) \leq f'(j)$$

The row f' is a permutation of the row f; no elements have been added or lost.

Sorting can be performed by many different algorithms, with different complexities in time (the number of elementary algorithms to be performed) and in space (the number of elementary variables necessary for the performance of the algorithm). Of all computing time used every day a large part is spent in sorting all kinds of files according to various criteria. It is therefore of great economic importance to look for sorting algorithms that are efficient in time and space.

Efficient in space: the amount of space needed depends at least linearly on n, the number of elements to be sorted. We shall investigate only those sorting algorithms that, apart from the elements themselves, need only a small number of elementary variables, if possible independent of n. We therefore restrict ourselves to *in situ* sorting, in which f' is built up in f. The permutation process takes place in the variable f itself.

Efficient in time: if we program an *in situ* sorting algorithm in a straightforward fashion we obtain a formulation as given in chapter 8. The time complexities of these obvious sorting algorithms are quadratic in the number of elements to be sorted. This means that in for example doubling the number of elements, the amount of work goes up by a factor of four. Such an algorithm may well work adequately for the sorting of the data of a thousand employees; but for sorting the data of 200 million inhabitants the amount of work goes up prohibitively. It is obvious that such methods are useless for sorting really large files.

14.1 Merge Sort

One way out is not sorting all the elements simultaneously but sorting the two halves of the row separately and then in some way combining the two sorted parts. This is another application of the "divide and conquer" technique, a halving method.

The reason for expecting that this is faster is the following: if the elements to be sorted are split into two equal groups, each of those groups can be sorted according to a quadratic algorithm in one fourth of the time needed for the whole row. After that the two sorted groups have to be mixed in some way. If we now manage to perform this mixing fast enough we shall gain in total time.

We shall now design such a Merge Sort algorithm.

```
PROC merge sort (INT CONST lwb, upb):
   IF lwb < upb
   THEN
     INT CONST middle:: (lwb + upb) DIV 2;
     merge sort (lwb, middle);
     merge sort (middle + 1, upb);
     in situ merge process
   FI.

in situ merge process:
   INT VAR down:: lwb;
   INT VAR up:: middle + 1;
   REP
     IF the smallest is up
     THEN
       set smallest aside;
       shift lower group up by 1;
       drop smallest in free slot
     ELSE
       pass down element
     FI
   UNTIL lower group is up or empty
   ENDREP.

the smallest is up:
   f[up] < f[down].

set smallest aside:
   ELEMENT CONST smallest:: f[up].

shift lower group up by 1:
   INT VAR i;
   FOR i FROM up - 1 DOWNTO down
   REP
     f[i+1]:= f[i]
   ENDREP;
   down INCR 1;
   up INCR 1.
```

```
drop smallest in free slot:
   f[down-1]:= smallest.

pass down element:
   down INCR 1.

lower group is up or empty:
   upb < up OR up <= down.

ENDPROC merge sort;
```

Now the sorting of the two halves will decidely go faster, but the `in situ merge process` spoils everything: large sub-rows have to be shifted about more than once, so the algorithm again has a quadratic worst case behaviour. It is easy to see that by making use of a second row variable as an auxiliary the merging can be performed in linear time. But then the algorithm needs too much auxiliary memory to be called *in situ.* Variants of the Merge Sort algorithm, that do not work *in situ* but make use of large background memories, are indeed in daily use. We shall not deal with them here.

Question: How does the time complexity of this version of Merge Sort depend on the number of elements to be sorted?

14.2 Quicksort

In Merge Sort the two halves are sorted separately and then merged. We might also try to first unmerge the row into two parts and then sort those two parts separately.

For this, C.A.R. Hoare in 1961 proposed a recursive sorting algorithm that cannot but be called elegant. He first splits the row into two parts $[1 : m]$ and $[m + 1 : n]$ by moving elements until all elements from the first part according to the given ordering relationship precede all elements of the second part, i.e.:

$$(1)\ k \in [1 : m], j \in [m + 1 : n] \rightarrow f_k \leq f_j.$$

The number of elements to be moved for this purpose is linear in n. Once the table has been split up in this fashion, the two parts can be sorted separately *in situ* (by the aid of the same algorithm Quicksort) so that we obtain:

$$(2)\ k < j \in [1 : m] \rightarrow f_k \leq f_j,$$

$$(3)\ k < j \in [m + 1 : n] \rightarrow f_k \leq f_j.$$

From (1), (2) and (3) we can deduce immediately:

$$(4)\ k < j \in [1 : n] \rightarrow f_k \leq f_j,$$

that is, without any further merging the whole row has been sorted *in situ.*

The tricky part is the splitting process in which (just as in Merge Sort) an element may be moved more than once. Let us look more closely at this process.

14.2.1 Splitting the row

What is the ideal splitting of the row? In the ideal case the row is split into two equal parts. Then we have most advantage from the splitting. We therefore have to find an element x such that as many elements precede x as follow it. Such an element is called the *median*: it is that element which after sorting of the row would occur in the middle. But this has not brought us any nearer to a solution:

- elements may be equal so that the median is not uniquely defined;
- when the number of elements is even, there is no middle element: the middle lies between two elements;
- the obvious algorithm for finding the median is: sorting the table and then taking the middle element.

We shall therefore be satisfied with an approximation to the median; if this approximation is reasonable, we shall not be far from optimal behaviour of the algorithm; we shall later investigate the behaviour for a bad approximation. There are still two different ways in which we may proceed:

- choosing a specific element from the row;
- choosing a value (not necessarily an element) as near as possible to the median value.

Proceeding according to the first method, we may in principle take any element of the row. Let us take the middle element as the approximate median. Then we are in luck if the whole row happens to be (nearly) sorted.

What is the worst possible split? That is one in which one of the two parts is empty and the other the whole original row. With this split we do not advance at all, because sorting the greatest part then consists of sorting the original row — if we are not careful the algorithm may even get into a never-ending loop. We shall therefore see to it that each of the parts is really smaller than the original row.

We will now split according to the following idea. Let the row be non-empty. We choose an element in the middle as the approximate median. We let a pointer p go upwards through the row and a pointer q go downwards through the row. Whenever we find under p an element greater than the approximate median and under q an element smaller than the approximate median we exchange those two elements. In other words, if we find on both sides an element that is out of position we exchange those elements.

More precisely: let there be given a global row of elements

```
ROW n ELEMENT VAR f
```

in which `ELEMENT` is one or another type. Let $1 \leq i < j \leq n$. We choose m approximately in the middle of $[i:j]$. Let $x = f_m$ (the approximate median). We want to order $f_i, \ldots, f_j$ *in situ* and choose p and q from $[i-1:j+1]$ in such a way that for all k in $[i:j]$:

$$\begin{array}{llll} k < p & \rightarrow & f_k \leq f_m & \text{and} \\ k = p & \rightarrow & f_k > f_m & \text{and} \\ k > q & \rightarrow & f_k \geq f_m & \text{and} \\ k = q & \rightarrow & f_k < f_m & \text{and} \\ q \leq p & & & \end{array}$$

In this process, the variable p gets as value the index of the first element greater than f_m, and q points in the same way to the last element smaller than f_m. If the left part is empty, q obtains the value $i-1$; if the right part is empty, p gets the value $j+1$. In a picture:

$$\begin{array}{ll} \text{elements} \leq f_m & i \longrightarrow p \\ \text{elements} \geq f_m & q \longleftarrow j \\ \text{elements} = f_m & q \longleftrightarrow p \end{array}$$

We have now split the row into three parts, of which the middle part is non-empty because it contains at least the approximate median. If the median element does not end up automatically between p and q, we have to shift it to this middle part. The middle part may also contain some elements other than the median and equal to it, but need not necessarily contain all elements equal to the median. We need not sort the middle part any more. The left part or the right part may be empty. Since, however, both parts are smaller than the original row, a never-ending repetition is excluded.

```
PROC split (INT CONST i, j, INT VAR p, q):
   initialize;
   WHILE p and q suitable candidates
   REP exchange p and q element
   ENDREP;
   if necessary bring median to middle.


initialize:
   p:= i; q := j;
   INT CONST m:: (i + j) DIV 2;
   ELEMENT CONST med:: f[m].

p and q suitable candidates:
   shift p upwards as far as possible;
   shift q downwards as far as possible;
   p < q.

shift p upwards as far as possible:
   WHILE may shift p upwards
   REP p INCR 1
   ENDREP.

may shift p upwards:
   IF p > j THEN false ELSE f[p] <= med
FI.

shift q downwards as far as possible:
   WHILE may shift q downwards
   REP q DECR 1
   ENDREP.

may shift q downwards:
   IF q < i THEN false ELSE f[q] >= med
FI.

exchange p and q element:
   ELEMENT CONST l:: f[p];
   f[p]:= f[q]; f[q] := l;
   p INCR 1; q DECR 1.

if necessary bring median to middle:
   IF m > p
   THEN exchange m and p element
   ELIF m < q
   THEN exchange m and q element
   FI.

exchange m and p element:
   f[m]:= f[p];
   f[p]:= med;
   p INCR 1.

exchange m and q element:
   f[m]:= f[q];
   f[q]:= med;
   q DECR 1.

ENDPROC split;
```

The splitting process is linear in n because p and q together will traverse at most $2n$ elements. The middle part is guaranteed to be non-empty. It may even contain any number of elements, e.g. in sorting a row consisting of only equal elements.

Observe that the order indicated in the body of the procedure `split` is an overspecification: we do not really wish *first* to move `p` up *and then* to move `q` down. These two actions should be undertaken collaterally, or possibly even in parallel.

14.2.2 Sorting with the aid of `split`

Once the row has been split we can sort it recursively:

```
PROC quicksort (INT CONST lwb, upb):
     IF the row contains more than 1
element
     THEN
       INT VAR p, q;
       split (lwb, upb, p, q);
       quicksort (lwb, q);
       quicksort (p, upb)
     FI.

the row contains more than 1 element:
     lwb < upb.
ENDPROC quicksort;
```

Do you see another overspecification in this procedure? What guarantees the termination of this procedure?

The splitting process gives the worst result when accidentally either the highest or the lowest element happens to be in the middle. What complexity does the algorithm have in this worst case? And in the best case?

The "average" complexity depends strongly on the distribution of the values of the elements. In [KNU73] it is proved that the time complexity of the algorithm, under the condition that the elements have been chosen at random from a homogeneous distribution, is of the order $O(n \log n)$, as in the best case, but with a different multiplying factor. For sufficiently large values of n this is arbitrarily better than $O(n^2)$.

We end by tracing an application of Quicksort (see figure 14.1). The elements which are still to be sorted are underlined. The approximate median in the splitting process is surrounded by a box. We indicate the situation only at some critical moments.

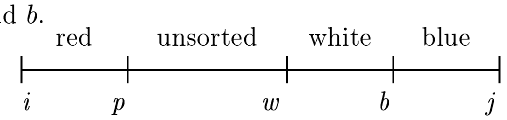
14.2.3 Splitting the row: an alternative method

In [DIJ76] the problem of the *Dutch National Flag* is posed and solved. We may use this solution directly for splitting the row in Quicksort. The resulting algorithm is somewhat more elegant than Hoare's original splitting algorithm but needs a few more movements.

Consider a row of elements each of which has one of the colours red, white or blue. Order them *in situ* until they make up a Dutch flag (red followed by white followed by blue). In other words, we want to permute the original row *in situ* until we obtain three separate areas, from left to right: all red elements (elements smaller than the median), all white elements (elements equal to the median) and all blue elements (elements greater than the median).

Half-way through the execution of the algorithm, we have four areas: one containing red elements, one containing white, one containing blue and one containing elements yet to be considered. The coloured areas we keep in the intended order, that much is clear; but where do we put the fourth area? We are completely free in this choice: initially the whole row is yet to be considered; at the end that area is empty, so its place does not matter. We shall make a specific choice and leave it to the reader to reason out why other choices in any case do not lead to better algorithms.

We choose the fourth area to be between red and white and mark the areas by the aid of the indices r, w and b.



therefore

$$\begin{array}{lcll} i \le k < r & \rightarrow & f_k < & med \text{ (red)}, \\ w < k \le b & \rightarrow & f_k = & med \text{ (white)}, \\ b < k \le j & \rightarrow & f_k > & med \text{ (blue)}. \end{array}$$

We now repeatedly look at the element f_w (the reader may convince himself that to look at f_r is less advantageous). If f_w is white, we simply move w by 1 to the left. If f_w is red, we exchange that element with f_r and shift r by 1 to the right. If f_w is blue, we exchange it with f_b and shift both b and w by 1 to the left.

It is easy to see the correctness of this algorithm. The red, white and blue areas remain red, white and blue. In each step the fourth area is diminished by 1 element, so the algorithm certainly terminates. If we have chosen as *med* an element of the row, the white area will be at least 1 long, so that the red and blue areas at the end of the split will certainly be shorter than the whole row, thus guaranteeing termination of the algorithm.

Initially the red and blue areas will be empty. If we choose f_j as *med,* the white area will initially have length 1. (Can we also allow the white area to be initially empty?) The resulting procedure `split` looks as follows:

```
PROC split (INT CONST i, j, INT VAR r, w):
   initialize;
   WHILE not all elements considered
   REP
     inspect element w;
     IF element is white
     THEN
       shift w by 1 to the left
     ELIF
       element is red
     THEN
       exchange w and r element;
       shift r by 1 to the right
     ELSE
       exchange and b element w;
       shift b and w by 1 to the left
     FI
   ENDREP.

initialize:
   ELEMENT CONST med:: f[j];
   r:= i;
   w:= j - 1;
   INT VAR b:: j.

not all elements considered:
   r <= w.

inspect element w:
   ELEMENT CONST this:: f[w].

element is white:
   this = med.

shift w by 1 to the left:
   w DECR 1.

element is red:
   this < med.

exchange w and r element:
   f[w]:= f[r];
   f[r]:= this.

shift r by 1 to the right:
   r INCR 1.
```

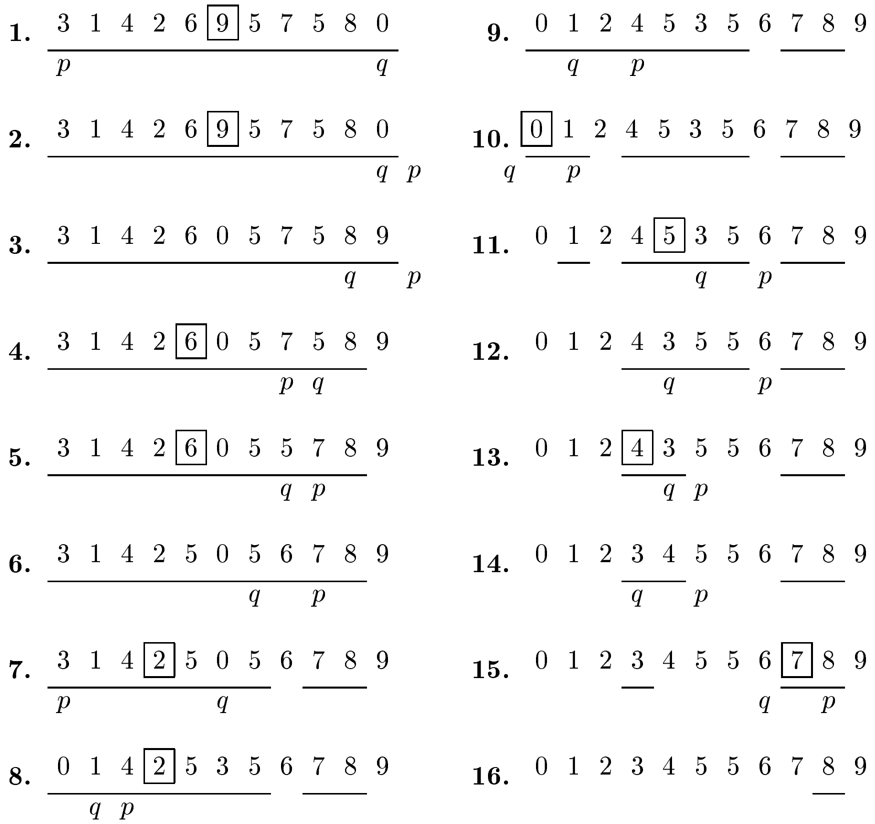


Figure 14.1: Trace of Quicksort

```
exchange w and b element:
   f[w]:= f[b];
   f[b]:= this.

shift b and w by 1 to the left:
   w DECR 1;
   b DECR 1.

ENDPROC split;
```

Whenever f_r is red and so is `this`, this element is first moved to f_w; in the next iteration, it will be brought back to the red area. This is a weakness of the algorithm, a sacrifice to simplicity. We can modify the algorithm in such a way that the index r is first moved to the right as long as there are red elements. The algorithm then becomes less clear-cut, but comes nearer to the splitting according to Hoare. Investigate this modification.

14.3 Exercises

1. Try to find an iterative solution for sorting in time $O(n \log n)$. (Hint: try Merge Sort with an auxiliary row, or look up the matter in [KNU73].)
2. Design a splitting algorithm in which not one element of the row but the average of three elements is taken as median [EMB70].

Chapter 15

Backtrack programming

The mouse in the maze was a good example of an algorithm based on systematically first doing and then undoing steps in the direction of the solution. The mouse as it were retraces its steps. There are important classes of problems that can be solved according to such a *backtrack* strategy.

15.1 The enumeration problem

The problem of *the mouse in the maze* can be reformulated as follows: find all ways to make a loopfree chain of mice from the initial position to the cheese. For every field of the maze that is not filled with `wall` or `cheese` we have to decide whether we cover it with a mouse or leave it free. We can therefore look upon the, at most, `n * m` free fields of the maze as variables, each of which has to take one of the values `free` and `mouse`.

We shall call the collection of the values that the ith variable (the ith free field) can assume the *selection space* for that variable. We have here an instance of the **enumeration problem**:

> find all sequences $(x_1, x_2, \ldots, x_n)$ with x_i from a finite set X_i, $1 \leq i \leq n$, that satisfy predicate $P(x_1, x_2, \ldots, x_n)$ [KNU75].

Let k be the number of free fields in the maze ($1 \leq k \leq n * m$). In our example the desired property $pathtocheese(x_i, x_2, \ldots, x_k)$ is the property that the fields $1, 2, \ldots, k$ form a loopfree chain of mice from the initial position to the cheese, while the remaining fields are free.

There are, for example, 8 free fields in the maze of Fig. 15.1. These free fields can be enumerated from the top to the bottom by rows and in each row from the left to the right.

The three possible solutions are given as sequences of field values satisfying the property *path to cheese.* Here f stands for *free* and m for *mouse.*

1. *path to cheese*(f, f, m, m, m, f, f, f),
2. *path to cheese*(m, m, m, m, m, f, f, f),
3. *path to cheese*(f, f, f, f, f, m, m, m).

We obtain an alternative formulation by taking as the ith variable the position of the ith mouse, $x_i = (r, c)$

Figure 15.1: Paths to cheese

with $1 \leq r \leq n, 1 \leq c \leq m$, which leads to another enumeration problem with the same result, i.e. the same collection of loopfree paths. Possibly the second formulation is somewhat more obvious and in any case more according to the spirit of the program already obtained.

The three solutions shown in Fig. 15.1 can now be formulated as

1. *path to cheese*
   $((3,1),(2,1),(2,2),--,--,--,--,--)$,
2. *path to cheese*
   $((3,1),(2,1),(2,2),(1,2),(1,3),--,--,--)$,
3. *path to cheese*
   $((4,2),(4,3),(3,3),--,--,--,--,--)$.

Now it suffices to deal with subsequences that yield a solution. As the remaining elements of the sequences are unimportant we denoted these arbitrary positions by "$--$".

We shall pursue the first formulation, however, in order to have a fresh look at the problem.

We shall investigate some general methods for solution of the enumeration problem and in this way discover a natural application of recursion.

15.1.1 The brute force method

The enumeration problem is a finite combinatorial problem and as such can in principle be programmed straightforwardly: we need only go systematically through all combinations of variable values.

```
WHILE not all combinations tried
REP
  generate next combination;
  test whether it satisfies P
ENDREP
```

It is not immediately obvious how to generate the next combination but with some thought all kind of schemes can be found.

In our case, according to the first formulation there are 2^k, and according to the second formulation there are even at most k^k combinations (in general: $q_1 * q_2 * \ldots * q_k$, in which q_i is the number of elements of the selection space X_i). This may be somewhat overdone. There is no sense in going blindly through the whole selection space $X_1 * X_2 * \ldots * X_k$ and we have to find a more clever approach.

15.1.2 The selection tree

Consider the collection of all possible subsequences over the selection space (see fig. 15.2). For the first variable x_1 there are q_1 different possibilities. For each of those, the second variable x_2 admits q_2 possibilities, and so on.

Without loss of generality we can represent these subsequences as a **tree** with branching, the *selection tree.* In our case there are k variables: the selection tree has depth k. Schematically we can depict the tree as shown in Fig. 15.3.

Figure 15.3: Selection tree with levels 1 . . . k

According to the *brute force* method we could walk through the selection tree systematically, for example (in bastard Elan):

```
FOR v1 IN (free, mouse)
REP FOR v2 IN (free, mouse)
    REP    ...
               FOR vk IN (free, mouse)
               REP IF path to cheese(v1,
v2, ..., vk)
                   THEN report a solution
                   FI
               ENDREP
           ...
    ENDREP
ENDREP
```

But, as we have already established, this makes no sense: k tree is too large.

Upon closer analysis we observe that in walking through the tree it is often possible to decide early (i.e. before we reach level k) that some specific branch cannot lead to a solution, because, for example, there is a loop in the path or the mice get into a dead end. In this case we may discard the whole branch. In this way we may (if we are lucky) have to visit only a fraction of the tree (see Fig. 15.4).

Figure 15.4: Selection tree, partially discarded

We must now somehow generate the possible combinations in such an order that unproductive branches of the selection tree are found as early as possible.

15.1.3 Heuristics

Heuristics is a methodology for solving new problems by means of known methods.

In deciding the order in which the selection tree is going to be traversed we have a number of degrees of freedom:

- The *choice of the order* of the variables: we want to go through the variables in such an order that unproductive branches are found as early as possible, so that the selection tree is most effectively reduced by discarding branches early in the search process. In the case of *the mouse in the maze,* for example, we must not go through the fields in random order or in a fixed order like “from left to right”, “from top to bottom”: it is always preferable to extend a path at the end, so that the possible loops and dead ends are found early.

- *Reduction of the selection space* per variable: often the values of the variables already filled restrict the possible values of subsequent variables.

- The use of more or less strong *rejection criteria* for parts of the tree: in our case, loops in the path and dead ends. It might for example be advantageous first to close off all dead ends in the maze since they cannot lead to the cheese.

- The exploitation of *symmetry arguments* in order to obtain more solutions at a time — not so evident in this case but in many examples simple and often necessary in order to keep a grip on the complexity of the algorithm.

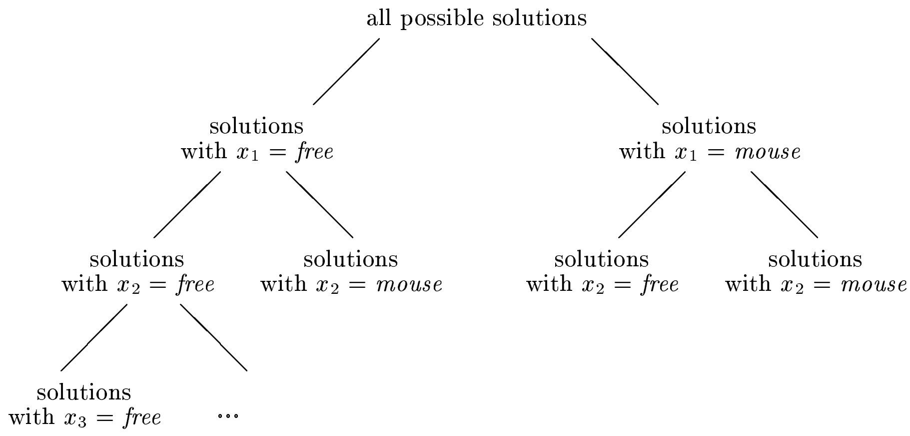


Figure 15.2: The selection space

The way we can make use of these and other degrees of freedom in solving a given problem depends on the properties of that problem and on our own imagination: we have to invent something.

In honour of Archimedes, who gave us the saying "Eureka" and who was the first known user of heuristic methods, the word *heuristic* is used for any strategy, method, rule of thumb or trick that serves to reduce the complexity of a search process without impairing the correctness of the solutions found [SLA71]. (In some scientific fields this word is also used for a method to reduce the complexity of a search process that yields "sufficiently good" solutions; in such a case we will speak of an *approximate heuristic* rather than the exact heuristic meant here.)

15.1.4 The backtrack method

We want to find all sequences $(x_1, x_2, \ldots, x_n)$ with x_i from X_i, $1 \leq i \leq n$ that satisfy a property $P_n(x_1, x_2, \ldots, x_n)$.

To solve this problem we try to find intermediate properties $P_k(x_1, x_2, \ldots, x_k)$ such that for $0 \leq k < n$ we can prove that property $P_{k+1}(x_1, x_2, \ldots, x_k, x_{k+1})$ implies property $P_k(x_1, x_2, \ldots, x_k)$.

In other words, if P_k does *not* hold for $(x_1, x_2, \ldots, x_k)$ then P_{k+1} *can not* hold for any choice for x_{k+1}. Continuing this argument inductively: the sequence $(x_1, x_2, \ldots, x_k)$ can not be extended to $(x_1, x_2, \ldots, x_k, \ldots, x_n)$ that satisfies P_n.

We give each P_k in the form of an algorithm to decide whether a sequence satisfies that property.

In abstracto we can search according to the following algorithm:

```
ROW n EL VAR x; INT VAR k ::  0;
PROC generate all continuations:
   {P_k(x_1,...,x_k) ∧ 0 ≤ k ≤ n}
   IF k = n
   THEN
     {P_n(x_1,...,x_n)}
     report a successful sequence
   ELSE
     {P_k(x_1,...,x_k) ∧ 0 ≤ k < n}
     EL VAR y;
     FOR y := each element from X_{k+1}
     REP
       IF P_{k+1}(x_1,...,x_k,y)
       THEN
         x[k+1] := y;
         k := k+1;
         {P_k(x_1,...,x_k) ∧ 0 < k ≤ n}
         generate all continuations;
         k := k-1
         FI
     ENDREP
   FI
ENDPROC generate all continuations;
```

The expressions between curly brackets are *invariants,* properties that invariably hold at that particular point of the program. They serve here to clarify the program but are not part of the program and therefore are merely comments.

Observe that after dealing with one branch of the selection tree the algorithm returns to a previous point in that selection tree; from this property the name *backtrack algorithm* is derived. After each recursive call the algorithm comes back (by means of `k := k-1`) upon a choice made before (`x[k+1] := y; k := k+1`). In the maze problem this backtrack consisted in explicitly removing a mouse from the field.

The backtrack algorithm traverses the selection tree in the "depth first" order: from left to right but going in such a way that every branch is traversed completely before its right neighbour is traversed, omitting branches that are recognized to be non-productive (see Fig. 15.5).

15.1.5 On the choice of the P_ks

The predicates P_k form the embodiment of the heuristics mentioned a few pages ago. In choosing them quite some knowledge of the problem and a good deal of inventivity is needed. Furthermore it must be easy to formulate them as algorithms that can be computed

Figure 15.5: Selection tree, order of traversal

efficiently. How do we choose them?

By choosing TRUE for P_k for $k < n$ we again obtain the brute force method; obviously these P_ks are too weak. In the ideal case it should hold that

> $P_k(x_1, \ldots, x_k)$ exactly when there exist $x_{k+1}, \ldots, x_n$ such that $P_n(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$.

That is, P_k indicates precisely whether there exists some continuation. But such an algorithm may be difficult to find and may be just as time-consuming in its execution as the original problem. (Namely, checking the truth of such a P_k assumes the original problem has been solved.)

In general we shall have to make a compromise between restrictive but time-consuming, and weak but easily computed P_ks. If we choose them too weak, we find the correct solutions but have to go through too much of the selection tree. When we choose them too strong, they do not lead to any gain in efficiency.

15.2 Example: 8 Queens on a chessboard

A classical example of a backtrack algorithm is the solution of the following problem. Place 8 Queens on a chessboard in such a way that no Queen can attack any other Queen (horizontally, vertically, on a right diagonal or left diagonal).

Choice of the variables. As variables we choose the positions of the Queens, i.e. 8 coordinates $\in [1 : 8, 1 : 8]$.

Reduction of the selection space. In each row only one Queen may stand, because it could attack a second Queen in the same row. Furthermore there must be exactly one Queen in a row, otherwise we will never put eight Queens on a chessboard.

Order of the variables. We (arbitrarily) number the variables from 1 to 8 depending on the row in which they occur.

```
PROC place queens from (INT CONST this
row):
   IF this row > 8
   THEN
     report success
   ELSE
     INT VAR i;
     FOR i UPTO 8
     REP
       IF position i is not threatened
       THEN
         take position i;
         place queens from(this row + 1);
         free position i
       FI
     ENDREP
   FI.
```

The place of the ith Queen is kept in a global row

```
ROW 8 INT VAR column queen;
```

We refine

```
take position i:
   column queen[this row] := i.
```

In order to free this position there is nothing we have to do explicitly — the return from the procedure is sufficient.

```
free position i:  .

position i is not threatened:
   INT VAR j;
   FOR j UPTO this row -1
   REP
     IF jth queen threatens i
     THEN
       LEAVE position i is not threatened
WITH false
     FI
   ENDREP;
   true.

jth queen threatens i:
   same column OR above to the right OR
above to the left.

same column:
   column queen[j] - i = 0.

above to the right:
   column queen[j] - i = this row - j.

above to the left:
   column queen[j] - i = j - this row.

report success:
   put("Solution of 8 queens problem");
   line;
   print the chessboard.
ENDPROC place queens from;
```

```
PROC print the chessboard:
   INT VAR k;
   FOR k UPTO 8
   REP
     print one line;
     line
   ENDREP.

print one line:
   INT VAR l;
   FOR l UPTO 8
   REP
     print position
   ENDREP.

print position:
   IF l = column queen[k]
   THEN put("Q ")
   ELSE put(". ")
   FI.

ENDPROC print the chessboard;
```

We need not initialize the row `column queen`. As main program we can take:

```
ROW 8 INT VAR column queen;
place queens from(1).
```

Without too much work this program yields all 92 solutions for the 8 Queens problem. By making a clever use of symmetry arguments, the amount of computation could be reduced further, because only 12 of those are essentially different.

15.3 The optimization problem

A generalization of the enumeration problem is the discrete optimization problem [GOL65].

> Determine that sequence $(x_1, x_2, \ldots, x_n)$ with x_i from a finite set $X_i, 1 \leq i \leq n$, for which some specific *criterion function* $F_n(x_1, x_2, \ldots, x_n)$ takes a minimal (or a maximal) value. Compute that value of F_n.

When there is more than one solution, we may choose one of them or the problem may be formulated such that we have to give a list of all solutions.

Obviously this problem can be seen as an enumeration problem, in which we remember the best of all sequences found up to now. Conversely an enumeration problem can be seen as a discrete optimization problem with a two-valued criterion function (useful/useless) for which we are asked to give all useful solutions. Observe that we restrict ourselves to finite discrete selection spaces X_i. If X_i is for example a segment of the real numbers and F_n a partially differentiable function, completely different methods may be applied. In operations research and cryptography many examples of discrete optimization problems can be found.

Figure 15.6: Distorted selection tree

Now consider such a discrete optimization problem. We shall derive from the problem formulation a backtrack algorithm by constructing a collection of partial criterion functions

$$F_k(x_1, x_2, \ldots, x_k), 0 \leq k < n$$

that satisfy

$$F_k(x_1, \ldots, x_k) \leq F_{k+1}(x_1, \ldots, x_k, x_{k+1}), 0 \leq k < n$$

The partial criterion function F_k can be seen to provide a lower limit for the minimal value that the criterion function F_n can take on for any argument starting with $(x_1, \ldots, x_k)$.

In other words: F_k is a pessimistic guesser for F_n; it may underestimate but never overestimate. If the value of F_k tells us that a branch can be discarded, we are sure that this is never unjustified.

15.3.1 Branch-and-Bound

In a certain sense more information can be expressed in these functions F_k than in the two-valued functions that we constructed for the enumeration problem. We can use this information systematically: as soon as we have constructed one complete sequence $(x_1, \ldots, x_n)$, we know a **bound** for the minimal value of the solution, that we can use to reject those partial sequences $(x_1, \ldots, x_k)$ that can never lead to a better result. As we find better and better solutions we can therefore reject larger and larger parts of the selection tree for which we find $F_k \geq$ *bound*.

Symbolically we can indicate this in the form of a selection tree that has been distorted in such a way that the length of a path from the root is equal to the value that the criterion function F_n assumes for it (Fig. 15.6.

This variant of the backtrack algorithm is called the *Branch-and-Bound* algorithm — a name that captures both the branching of the selection tree and the bounding by solutions already found.

15.4 Example: Shortest route

Between a collection of cities, numbered consecutively from 1 to n, exists a system of roads, so that between each pair of cities i and j there may be either no connection at all or a *direct connection* of some specific length dw_{ij} (a real number). See the example in Fig. 15.7.

Figure 15.7: Road system with distances

We are looking for an algorithm `shortest route(i, j)` that computes the length of a shortest route from `i` to `j`. Of course there may be more than one shortest route, but in that case they all have the same length. How do we indicate that there is no direct way from `i` to `j`? We shall encode this by giving a sufficiently large value to `dw[i][j]`, for example `infinite`, so that if there is a shortest route, it will have a length smaller than `infinite`.

The matrix `dw` is symmetric (unless we introduce one way roads).

```
LET number of cities = ...;
ROW number of cities ROW number of cities
REAL VAR dw;
REAL CONST infinite :: maxreal /
real(number of cities);
learn system of roads;
ask for i and j;
compute shortest route from i to j.
```

We consider a route consisting of sequences of connected direct ways and choose as variables the cities over which they go. In terms of the definition of the optimization problem: find that sequence of direct connections (starting in i, and forming a loopfree connected path ending in j) for which the sum of the lengths is minimal, i.e. minimize

$$F_n(x_1, \ldots, x_n) = dw(x_1, x_2) + dw(x_2, x_3) + \ldots + dw(x_{n-1}, x_n)$$

with $x_1 = i$ and $x_n = j$, $n \geq 1$.

As partial criterion functions F_k we obviously must choose the length of the path from i to k. This choice is meaningful, because for every route from i to j that starts with the segment $i, \ldots, k$ must hold:

$$F_n(x_1, \ldots, x_k, \ldots, x_n) \geq F_k(x_1, \ldots, x_k).$$

The essential part of the backtrack algorithm is

```
continuations from p:
   INT VAR q;
   FOR q UPTO number of cities
   REP IF there is a direct way from p to
q
       THEN increase partial length with
that direct way;
            try continuations via q;
            reduce partial length with
that direct way
       FI
   ENDREP.
```

As soon as one route has been found, its length will be a bound for the partial criterion functions. If the length of some partial route exceeds the bound we need not look at any of its continuations, they will all exceed that bound. Every time we find a new route whose length is less than the bound, we adjust the value of the bound.

```
try continuations via q:
     via q to j
     IF partial length smaller than the
bound
     THEN IF q=j
          THEN improve bound
          ELSE on the path[q] := TRUE;
               continuations from q;
               on the path[q] := FALSE
          FI
     FI.
```

In order to guarantee termination we will see to it that the path is loopfree. To that end we remember which of the cities are on the path, keeping a `BOOL` variable for each city. Observe that this information by itself is not enough to reconstruct the path.

The algorithm is started by taking a closer look at the start point (since i might be equal to j). Initially the path length is zero and the bound is infinite.

We now can concretize this algorithm:

```
compute shortest route from i to j:
   REAL VAR bound :: infinite;
   ROW number of cities BOOL VAR on the
path;
   no city is on the path;
   find shortest route(i, j, 0.0).
```

```
no city is on the path:
   INT VAR c;
   FOR c UPTO number of cities
   REP on the path[c] := false
   ENDREP.
```

```
PROC find shortest route (INT CONST i, j,
                          REAL CONST path
length):
   IF no sense in continuing
   THEN
   ELIF target reached
   THEN improve bound
   ELSE try all continuations
   FI.
```

The parameter `path length` contains the accumulated length of the route started at i; it plays the role of the F_k.

```
no sense in continuing:
   path length >= bound.

target reached:
   i = j.

improve bound:
   bound := path length.
```

Observe that this is always an improvement.

```
try all continuations:
   INT VAR k;
   FOR k UPTO number of cities
   REP
     try continuation via k
   ENDREP.

try continuation via k:
   IF NOT on the path[k] AND
      there is a direct way from i to k
   THEN
     on the path[k] := TRUE;
     find shortest route(k, j, path length
+ dw[i][k]);
     on the path[k] := FALSE
   FI.

there is a direct way from i to k:
   dw[i][k] < infinite.

ENDPROC find shortest route;
```

This optimization algorithm can be turned into an enumeration algorithm by simply modifying one refinement

```
no sense in continuing:
   FALSE.
```

We shall investigate the effect of the *Branch-and-Bound* heuristics by way of an example. As a measure for the amount of work to be performed we take the number of times that the procedure `find shortest route` is called. This number depends on the number of cities, the number of direct ways in the matrix `dw` and on their distribution and ratios.

We shall therefore generate repeatedly a "random" matrix with a specific *density,* i.e. of which a given percentage of the elements is not infinite. We shall exclude the direct way from the first to the last city (i.e. with indices 1 and `number of cities`, respectively).

```
fill the matrix:
   INT VAR i, j;
   FOR i UPTO number of cities
   REP
     dw[i][i] := infinite;
     FOR j FROM i+1 UPTO number of cities
     REP
       REAL CONST distance :: between 0
and 100;
         dw[i][j] := distance;
         dw[j][i] := distance
     ENDREP
   ENDREP;
   dw[1][number of cities] := infinite;
   dw[number of cities][1] := infinite.
```

We make use of the procedure `random` that, for each call, yields a pseudo-random real number between 0.0 and 1.0.

```
between 0 and 100:
   IF random < density
   THEN random * 100.0
   ELSE infinite
   FI.
```

By inserting a counter we now count the number of calls in `find shortest route (1, number of cities, 0.0)`. Since backtrack algorithms show a strongly fluctuating behaviour we average the result over ten times. Still the numbers measured show wild jumps, and give no more than an idea of the behaviour of both algorithms. Table 15.1 shows the results of such a simulation. Especially for higher densities, these numbers show a great advantage for the *Branch-and-Bound* method, but for large numbers of cities this method too will presumably demand a prohibitive amount of calculation.

density	number of cities	enumeration algorithm	branch-and-bound algorithm
.1	5	7	6
	8	12	10
	11	27	20
	14	52	37
	17	127	68
.3	5	11	12
	8	102	40
	11	1867	99
	14	4728	305
	17	> 10.000	1090
.5	5	32	18
	8	356	77
	11	> 10.000	280
	14		471
	17		1661

Table 15.1: Number of calls of `find shortest route`

It should be pointed out that the "random" choice of the `dw[i][j]` was not quite realistic: for a larger number of cities, each city will be connected to a rather small proportion of neighbouring cities, whereas direct connections between distant cities will hardly ever occur. With the density as the only parameter, we cannot model this behaviour satisfactorily. It might therefore be that some version of the *Branch-and-Bound* method may be applicable to this problem even in realistic situations. But for this simple "shortest distance" problem there exist much more efficient algorithms.

15.5 Exercises

1. (Permutations) Print all permutations of the numbers $1 : n$.

2. (Cutting up) From a sheet of n by n a number of squares with different whole-numbered sizes between 1 and $n - 1$ have to be cut, such that the largest possible part of the area of the sheet is used.

3. (Partitions) Under a partition of an integer $n > 0$ we understand a sequence $x_1, x_2, \ldots, x_m$ of integers with $x_i > 0$ and $x_1 + x_2 + \ldots + x_m = n$. Write a recursive program that finds all different partitions of an integer which is read in.

4. (Knight's tour) Write a recursive program that finds one tour of a Knight over a chessboard of $5 * 5$, starting at a given starting point, such that every field is reached exactly once.

5. (Paying in coins) A sum of money can be composed in different ways out of a collection of coins. Write a recursive program that, given an amount and a system of coins, decides in how many different ways this amount can be formed in that system of coins. Try it out for various amounts < 1000 and the following coin systems:

(H)	10, 20, 50, 100, 200, 500,
(NL)	1, 5, 10, 25, 100, 250,
(FRG)	1, 2, 5, 10, 20, 50, 100, 200, 500,
(SU)	1, 2, 3, 5, 10, 15, 20, 50, 100.

6. (Knight's jumps) On an — otherwise empty — chessboard stands a Knight. Write a recursive algorithm that, given the position of the Knight, marks all fields of the chessboard with the minimum number of jumps in which the Knight can reach that field.

7. (Camels) In a tunnel two caravans meet, each consisting of n camels, the nose of one camel touching the tail of his predecessor. Between the caravans is a free position exactly the size of a camel. The initial situation for $n = 4$ may, for example, be represented as **pppp qqqq**.

   A camel standing before that empty place can walk one camel length forward. A camel which is separated from that empty place by one other camel can jump over that other camel.

   Give a shortest sequence of camel movements that exchanges the positions of the two caravans.

8. (Spiral) Write a program that prints a narrowest right-going spiral of 100 fields, starting to the north in the origin $(0, 0)$. Each field of the spiral is a unit square which must have one side in common with its direct predecessor and its direct successor, and may not have a side or a corner in common with any other part of the spiral. Some of the fields of the board may contain a barricade which may also not be touched by the spiral.

Backtrack problems galore can be found in games and puzzles or as optimization problems in Operations Research.

Chapter 16

Transforming recursion to iteration

Consider an iterative algorithm with a form like

```
a;
WHILE b REP c ENDREP;
d
```

Obviously, it can be rewritten into an equivalent recursive form

```
PROC bc:
   IF b THEN c; bc FI
ENDPROC bc;
a; bc; d
```

In this chapter we shall inquire into the possibility of turning a recursive algorithm into an equivalent iterative version, which is not always simple.

The most important reason for which we discuss this subject is that it throws an illuminating light on the choices and the degrees of freedom in programming.

In this chapter we shall restrict ourselves to procedures with one single parameter, of the form

```
PROC p (INFO CONST x):
   IF c(x)
   THEN
     n(x); p(f(x)); m(x)
   ELSE
     d(x)
   FI
ENDPROC p;
```

in which `c(x)`, `d(x)`, `n(x)`, `f(x)` and `m(x)` stand for pieces of program in which `x` may appear, but which do not contain local declarations that have an application in one of the other pieces. The parameter `x` is therefore the only local object. For the abstract type `INFO` any concrete type may be assumed.

16.1 Eliminating right recursion

We speak about *right recursion* if the recursive call is executed immediately before the procedure terminated. Consider the following example of a right-recursive procedure

```
PROC p (INFO CONST x):
   IF c(x)
   THEN
     n(x); p(f(x))
   ELSE
     d(x)
   FI
ENDPROC p;
```

As usual, `c(x)`, `d(x)`, `n(x)` and `f(x)` are pieces of program in which `x` may occur and `p(f(x))` is a recursive call of `p`.

We can eliminate the right recursion by the introduction of an auxiliary variable `x1` and a `WHILE`-form

```
PROC p (INFO CONST x):
   INFO VAR x1:: x;
   WHILE c(x1)
   REP
     n(x1);
     x1:= f(x1)
   ENDREP;
   d(x1)
ENDPROC p;
```

The equivalence of both procedures can be proved by induction. Notice that both programs generate the sequence

$$n(x); n(f(x)); \ldots; n(\underbrace{f(f(\ldots f(x)\ldots))}_{(n-1)\text{ times}}); d(\underbrace{f(f(\ldots f(x)\ldots))}_{n\text{ times}});$$

provided all elements of the sequence

$$c(x); c(f(x)); \ldots; c(\underbrace{f(f(\ldots f(x)\ldots))}_{(n-1)\text{ times}})$$

are `true` and the call

$$c(\underbrace{f(f(\ldots f(x)\ldots))}_{n\text{ times}})$$

yields `false`.

Let us take as an example a procedure to calculate the sum of the first `n` of a given row of reals:

```
REAL VAR total:: 0.0;
```

```
PROC summate (INT CONST n):
   IF n >= 1
   THEN
     total INCR row[n];
     summate(n-1)
   FI
ENDPROC summate;
```

After transformation and simplification we obtain

```
REAL VAR total:: 0.0;

PROC summate (INT CONST n):
   INT VAR n1:: n;
   WHILE n1 >= 1
   REP
     total INCR row[n1];
     n1 DECR 1
   ENDREP
ENDPROC summate;
```

16.2 Application to Quicksort

In applying this transformation to

```
PROC quicksort (INT CONST lwb, upb):
   IF  upb-lwb > 0
   THEN
     INT VAR p, q;
     split(lwb, upb, p, q);
     quicksort(lwb, q);
     quicksort(p, upb)
   FI
ENDPROC quicksort;
```

we can eliminate only the last recursive call. The transformation is also complicated by the fact that the local variables `p` and `q` occur in the calls of `quicksort`. However, the declaration of `p` and `q` can without any problems be shifted outside the repetition. Some puzzling leads to

```
PROC quicksort (INT CONST lwb, upb):
   INT VAR lwb1:: lwb, upb1:: upb, p, q;
   WHILE upb1 - lwb1 > 0
   REP
     split(lwb1, upb1, p, q);
     quicksort(lwb1, q);
     lwb1:= p
   ENDREP
ENDPROC quicksort;
```

The assignment `lwb1:= p` is crucial. It corresponds to `x1:= f(x1)` of the transformation scheme. The procedure obtained in this way is somewhat faster than the original and uses somewhat less auxiliary memory. It is no longer right-recursive but still contains a *middle recursion.*

A further improvement can be achieved by exploiting the fact that the intervals `[lwb1:q]` and `[p:upb1]` will in all probability not have the same length. If we go into recursion only for the shorter interval, we may materially reduce the recursion depth. This idea leads to

```
PROC quicksort (INT CONST lwb, upb):
   INT VAR lwb1:: lwb, upb1:: upb, p, q;
   WHILE upb1-lwb1 > 0
   REP
     split(lwb1, upb1, p, q);
     IF q-lwb1 < upb1-p
     THEN
       quicksort(lwb1, q);
       lwb1:= p
     ELSE
       quicksort(p, upb1);
       upb1:= q
     FI
   ENDREP
ENDPROC quicksort;
```

Observe that we make here a creative use of the overspecification signalled before in the order of sorting of both halves.

Right recursion can be eliminated so easily that some compilers (for example for LISP but also for the language CDL2 in which ELAN was implemented) remove all right recursions automatically.

16.3 Eliminating middle recursion

If recursion no longer takes place "at the right", things get more difficult. Let us investigate the following scheme:

```
PROC p (INFO CONST x):
   IF c(x)
   THEN
     n(x); p(f(x)); m(x)
   ELSE
     d(x)
   FI
ENDPROC p;
```

The problem is that `m(x)` has to happen as often as `n(x)` and for the same sequence of arguments `x`, but in reverse order.

The execution of `p`, as it depends on the number of times that `c(x)` yields `true`, can be depicted as follows (by `ĉ` we indicate the first call of `c` that yields `false`):

```
 none  ĉ(x); d(x)
 once  c(x); n(x); ĉ(f(x)); d(f(x)); m(x)
twice  c(x); n(x); c(f(x)); n(f(x));
       ĉ(f(f(x)));
       d(f(f(x))); m(f(x)); m(x)
```

and so on.

We must try to rewrite `p` to an iterative procedure in such a way that in its execution the same sequence of calls of `c`, `n`, `p`, `f` and `m` occurs with the same arguments as in executing the original procedure `p`.

A possible solution makes use of a stack with values of type INFO.

16.3.1 The stack

By a *stack* we mean a data structure for the storage of elements that later must be fetched back in the reverse order of storing ("last-in-first-out", "LIFO-stack", "LIFO-list"). A physical model for such a stack can be found in most cafeterias: a stack of plates from which only at the top a plate can be removed or added (unless difficult contortions are performed).

A stack at any moment contains some number of elements. If that number is zero, the stack is called empty.

Typical operations on a stack are:

push add an element at the top. The number of elements thereby increases by one.

pop remove the element added last. The number of elements decreases by one.

Furthermore it can be asked (by means of a condition) whether the stack is empty. A stack may have a limitation on the number of elements, e.g. because a fixed amount of space has been reserved for it. In that case it must also be possible to find out whether the stack is full, so that no further element can be added. (Of course in real life every stack is limited in some way, because cafeterias and computers are not infinitely large. There may however be stacks whose maximum number of elements cannot be computed in advance).

A straightforward realization of a stack of elements (each of the abstract type EL) uses a row of elements. Such a row has a fixed number of elements, so that the upper limit is also fixed.

With this representation we can realize the stack as follows:

```
make an empty stack:
   LET max = ...;
   ROW max EL VAR stack;
   INT VAR top:: 0.
```

The integer variable top gives the index of the last element pushed, which is equal to the number of elements in the stack. This variable should not assume a value greater than max. The further operations can be indicated schematically

```
push x:
   IF stack is full
   THEN
     explosion
   ELSE
     top INCR 1;
     stack[top]:= x
   FI.

stack is full:
   top >= max.
```

```
pop x:
   IF stack is empty
   THEN
     implosion
   ELSE
     x:= stack[top];
     top DECR 1
   FI.

stack is empty:
   top < 1.
```

In case of implosion or explosion there is probably nothing better to do than to report the problem and terminate the execution of the program.

With the language mechanisms introduced in the next volume of this book we can realize a stack somewhat more elegantly (for example through a separate packet that defines an abstract type STACK). Also it is possible to choose such a representation (as a "linear list") that the necessity to give an *a priori* limit to the stack size disappears. We shall then return to the subject of the stack.

By the aid of a stack we can rewrite the procedure to:

```
PROC p (INFO CONST x):
   make an empty stack;
   INFO VAR x1:: x;
   WHILE c(x1)
   REP
     n(x1);
     push x1;
     x1:= f(x1)
   ENDREP;
   d(x1);
   WHILE NOT stack is empty
   REP
     pop x1;
     m(x1)
   ENDREP
ENDPROC p;
```

It is easy to see that the stack contains the arguments for the calls of m to be performed later.

This piece of program closely resembles the code that a compiler makes from the recursive version. We are doing the work that a good compiler would do by itself.

16.3.2 Example: Printing a natural number

Let us consider a recursive procedure print number for the paper-saving printing of natural numbers, assuming the availability of a procedure print digit that can print one digit.

The recursive solution given in section 12.1.1 is essentially:

```
PROC print number (INT CONST n):
   IF n >= 10
   THEN
     print number(n DIV 10);
     print digit(n MOD 10)
   ELSE
     print digit(n)
   FI
ENDPROC print number;
```

Rewriting this with the use of a stack yields:

```
PROC print number (INT CONST n):
   INT VAR n1:: n;
   make an empty stack;
   WHILE n1 >= 10
   REP
     push n1;
     n1:= n1 DIV 10
   ENDREP;
   print digit(n1 MOD 10);
   WHILE NOT stack is empty
   REP
     pop n1;
     print digit(n1 MOD 10)
   ENDREP.

make an empty stack:
   ROW 12 INT VAR stack;
   INT VAR top:: 0.

stack is empty:
   top < 1.
```

We have assumed here that the integer range is such that the number has at most 12 digits. We therefore do not have to check explicitly that `top` does not become too large.

```
push n1:
   top INCR 1;
   stack[top]:= n1.

pop n1:
   n1:= stack[top];
   top DECR 1.

ENDPROC print number;
```

16.3.3 Using an inverse function

A simpler solution without the use of a stack is possible when the following two conditions are met:

- Besides the algorithm `f(x)` we also have its *inverse,* `g(y)`, such that for *any* `x` in the range `g(f(x)) = x`.
- The function `f(x)` is strictly monoton (which is the usual case in programming practice).

If these conditions are met we can transform the recursive procedure to:

```
PROC p (INFO CONST x):
   INFO VAR x1:: x;
   WHILE c(x1)
   REP
     n(x1);
     x1:= f(x1)
   ENDREP;
   d(x1);
   WHILE x1 <> x
   REP
     x1:= g(x1);
     m(x1)
   ENDREP
ENDPROC p;
```

This transformation is preferable when the computation of the inverse of `f` is “cheaper” than the use of a stack.

We can see the formulation with a stack as a general technique to memorize the inverse rather than to compute it.

In the case of the procedure `print number` the first condition is not met since in `x1:= x1 DIV 10` information (namely, the remainder) is lost that cannot be recovered. Of course it must be possible to find an iterative version, based on division by suitable powers of ten, but this cannot be obtained in an evident fashion by rewriting the procedure in the form given. This is caused by the fact that in a sense we have not given enough information explicitly as a parameter.

16.3.4 Complete parametrization

The `INFO` parameter in the transformation schemes given is conceptually one single object that however may often have to be realized as more than one Elan-object. In this case the procedure obtains more than one parameter (or a composed object as a parameter). These parameters together must contain all information that plays a role in the recursion. In the previous example this can be achieved as follows.

Rather than dividing `n` repeatedly by 10, we introduce an extra parameter, the power of ten by which we want to divide, and leave `n` unchanged. To this end we declare a completely parametrized recursive auxiliary procedure to which we shall give the tremendous name `print number1`.

```
PROC print number (INT CONST n):
   print number1(n, 1)
ENDPROC print number;
```

```
PROC print number1 (INT CONST n, power of
ten):
   IF n DIV power of ten >= 10
   THEN
     print number1(n, 10 * power of ten);
     print digit(n DIV power of ten MOD
10)
   ELSE
     print digit(n DIV power of ten)
   FI
ENDPROC print number1;
```

In comparison to the scheme given above, the parameter `INFO CONST x` has been split into two objects `INT CONST n, power of ten`. The (symbolic) assignment `x1 := f(x1)` of the pattern must therefore be realized as two assignments, one for each of the components. Also, `n(x1)` turns out to be empty, so that only this assignment ends up in the first repetition.

Applying this transformation to `print number1` leads to:

```
PROC print number1 (INT CONST n, power of
ten):
   INT VAR n1:: n, power of ten1:: power
of ten;
   WHILE n1 DIV power of ten1 >= 10
   REP
     n1:= n1;
     power of ten1:= 10 * power of ten1
   ENDREP;
   print digit(n1 DIV power of ten1 MOD
10);
   WHILE n1 <> n OR power of ten1 <> power
of ten
   REP
     n1:= n1;
     power of ten1:= power of ten1 DIV 10;
     print digit(n1 DIV power of ten1 MOD
10)
   ENDREP
ENDPROC print number1;
```

The procedure `print number1` is now iterative and is called only once. We substitute its body for the call in `print number`, remove the spurious variable `n1` and make some further small simplifications with as result:

```
PROC print number (INT CONST n):
   INT VAR power of ten:: 1;
   WHILE n DIV power of ten >= 10
   REP
     power of ten:= power of ten * 10
   ENDREP;
   print digit(n DIV power of ten MOD 10);
   WHILE power of ten <> 1
   REP
     power of ten:= power of ten DIV 10;
     print digit(n DIV power of ten MOD
10)
   ENDREP
ENDPROC print number;
```

A further simplification is possible but this is already a quite respectable iterative program.

16.3.5 Other methods to remove recursion

In closing we want to mention two other points of view from which a transformation of recursion to iteration with the aid of a stack can be performed.

The first point of view is the *incarnation stack*: associate with every incarnation of the procedure one element in the stack, that comprises both the values of the parameters and its eventual local variables. A call of the procedure is turned into a repetition of the body of that procedure after the element belonging to the current incarnation has been stacked. After each repetition of the body one element is popped.

Yet another point of view is possible: looking upon the stack as a set of *tasks to be performed.* As an example, in a iterative version of Quicksort the stack will contain a number of intervals yet to be sorted.

We will not elaborate on these ideas because the result resembles greatly the use of the stack already mentioned, only the motivation is different.

16.4 Conclusion

It is not the intention of the transformations given to serve as a complete recipe, but to show the essence of such transformations. In every concrete case some brainracking and legwork will be necessary before such a transformation succeeds. Some further transformation rules can be found in [BIR77].

As long as our translators do not become appreciably smarter and as long as computing time is more expensive than human time, there may be situations in which it may be worthwhile to rewrite a recursive algorithm in an iterative version.

The elimination of recursion from an algorithm has a very limited effect on its speed. Often however it is possible to eliminate simultaneously the (implicit) use of a stack for the recursion (which concludes in using less memory).

The correctness of the resulting iterative algorithm is generally difficult to see, but can be guaranteed by starting out with a correct recursive version and transforming this while retaining correctness. A number of important algorithms have had such a development history.

Increasingly a specific school of programming methodology is developing, that looks upon programming as a repeated transformation process (from idea via formal specification via correct but inefficient realization to correct and also efficient realization). An exposition of this attitude can be found in [BAU76].

16.5 Exercises

1. Eliminate also the mid-

dle recursion from `quicksort` by making use of a stack. This can most simply be done by temporarily introducing an auxiliary procedure `PROC quicksort1 (INT VAR p, q)` which is explicitly parametrized with the boundaries `p` and `q` and has global access to the row `f`.

2. Eliminate the middle recursion from the program for the 8 Queens. A separate stack turns out to be unnecessary because the row `column queen` can play that role. Then compare the result with that in [WIR71].

3. Eliminate for as far as possible the recursion from a procedure computing the Ackermann function

$$
ack(m,n) = \begin{array}{lcl} m = 0 & \rightarrow & n+1 \\ m \neq 0, n = 0 & \rightarrow & ack(m-1,1) \\ m \neq 0, n \neq 0 & \rightarrow & ack(m-1, ack(m, n-1)) \end{array}
$$

This exercise demands quite a lot of inventiveness and perseverance. Further information about this function can be found in, for example, [SUN71].

Appendix A

Grammar of Elan

The following context-free grammar of Elan is a paraphrase of the official syntactic description [HOM79], in which the terminology at a number of places is chosen differently, and including a few revisions of the syntax that have later been agreed upon by the Elan community.

We give a complete context-free grammar in the notation of chapter 11. It can be seen as a recapitulation of the syntax diagrams that occur here and there in the text, but it also contains those constructs of Elan that have not yet been introduced.

There is a good reason why we do not simply reprint the syntax diagrams: the mechanism of syntax diagrams has the advantage of being immediately obvious to the beginner, but it has the danger that great amounts of information can be introduced in one picture, while the abstraction (in the form of useful intermediate concepts) is lost. The use of a context-free grammar is much more conducive to a careful "refining" of the concepts than the use of pictures, whilst retaining the overall view. Furthermore there exist many people (like the author) who can better memorize a hierarchical system of carefully formalized definitions than collections of two-dimensional pictures.

A.1 Syntactic abbreviations

In order to keep this syntax short and concise, we make use of a number of conventions for omitting redundant rules. For example, in the grammar we shall need the concept of a list of identifiers, with a syntax rule like

```
identifier—list:
   identifier;
   identifier, comma—token, identifier—list.
```

But we shall also have unit—list, with a similar rule; and there will be still more forms of lists. We therefore introduce the convention that for any notion of the form N—list (in which N stands for some word) we can assume

a1) N—list: N; N, comma—token, N—list.

Applied to the word unit, this leads to

```
unit—list:
  unit;
  unit, comma—token, unit—list.
```

Of course this is yet another device to increase the abstraction and enhance the readability of the grammar.

Further abbreviations are:

a2) N—option: N; .

Something that is optional may be left out.

a3) N—sequence: N; N, N—sequence.

A sequence of things consists of one or more of those things, one after another. Notice the difference: in an N—list there is a separator, the comma—token, between two consecutive elements, but there is no separator in an N—sequence.

a4) N—pack: open—token, N—list, close—token.

A pack is a list enclosed between parentheses.

a5) N—token: comment—option, N-symbol.

This rule expresses the fact that a comment is allowed before each symbol — although most symbols will not be preceded by a comment. At the end of this chapter, the representations of symbols are listed.

A.2 Programs and packets

```
elan-program:
  packet—sequence—option, main-packet.
```

In the sublanguage Elan-0 packets have not been implemented.

```
packet:
  packet-head, packet-body, packet-tail.

packet-head:
  packet—token, packet-name, packet-interface, colon—token.

packet-interface:
  export-interface.
```

Elan has no explicit import-interfaces. A packet implicitly imports everything that preceding packets have exported.

```
export-interface:
  defines—token, export-name—list.
```

The export-interface contains the names of those entities, defined in the packet, that are made visible in subsequent packets. Refinements and variables can not be exported.

packet-body:
packet-paragraph, refinement-train—option;
packet-paragraph, refinement-train—option,
period—token, packet-body—option.

packet-paragraph:
packet-unit;
packet-unit, semicolon—token, packet-paragraph.

packet-unit:
basic-declaration;
closed-declaration;
expression.

Closed-declarations can only occur in a packet-paragraph. They form the Bottom-Up part of the packet.

packet-tail:
end-packet—token, packet-name—option, semicolon—token.

main-packet:
packet-body.

An Elan-1 program has the form of a packet-body.

A.3 Procedures and operators

procedure-declaration:
procedure-head, procedure-body, procedure-tail.

procedure-head:
result—option, proc—token, procedure-name,
formal-parameter-specification—pack—option, colon—token.

result:
type-declarer.

formal-parameter-specification:
formal-declarer, formal-parameter-name—list.

procedure-body:
paragraph, refinement-train—option.

An Elan-0 program has the form of a procedure-body.

paragraph:
unit;
unit, semicolon—token, paragraph;
.

A paragraph obtains the type and yields the value of its last unit. Notice that a paragraph may be empty, in which case it yields no value and is of the (hypothetical) type VOID.

unit:
basic-declaration;
expression.

procedure-tail:
end-proc—token, procedure-name—option.

operator-declaration:
operator-head, operator-body, operator-tail.

operator-head:
result—option, op—token, operator-name,
formal-parameter-specification—pack, colon—token.

operator-body:
procedure-body.

operator-tail:
endop—token, operator-name—option.

A.4 Refinements

refinement-train:
period—token, refinement, refinement-train—option.

refinement:
refinement-head, refinement-body.

refinement-head:
refinement-name, colon—token.

refinement-body:
paragraph.

A.5 Declarations

The scope of a declaration occurring in a procedure-body is that procedure-body; the scope of a declaration occurring outside a procedure-body is the directly surrounding packet and (where the object-declarer is exported) all following packets.

basic-declaration:
object-declaration;
synonym-declaration.

A basic-declaration is a VOID unit.

object-declaration:
object-declarer, object-association—list.

object-association:
object-name, object-initialization—option.

A variable may have an initialization, but a constant must have one.

object-initialization:
initial—token, expression.

synonym-declaration:
let—token, synonym-association—list.

synonym-association:
synonym-value-association;
synonym-type-association.

synonym-value-association:
synonym-value-name, equal—token, denoter.

synonym-type-association:
synonym-type-name, equal—token, type-declarer.

closed-declaration:
procedure-declaration;
operator-declaration;
type-declaration.

The closed-declaration is not implemented in Elan-0.

type-declaration:
  type—token, type-association—list.

type-association:
  type-name, equal—token, type-declarer.

A.6 Declarers

object-declarer:
  type-declarer, access-declarer—option.

access-declarer:
  const—token;
  var—token.

A missing access-declarer is assumed to be `CONST`.

type-declarer:
  elementary-type-declarer;
  composed-type-declarer.

elementary-type-declarer:
  concrete-type-declarer;
  abstract-type-declarer.

concrete-type-declarer:
  int—token;
  real—token;
  bool—token;
  text—token.

abstract-type-declarer:
  type-name.

composed-type-declarer:
  row-declarer;
  struct-declarer.

row-declarer:
  row—token, cardinality, type-declarer.

cardinality:
  denoter;
  synonym-value-name.

The cardinality of a row must be of type `INT`, and either a denoter or a synonym for one.

struct-declarer:
  struct—token, field-specification—pack.

field-specification:
  type-declarer, field-name—list.

formal-declarer:
  object-declarer;
  procedure-declarer.

procedure-declarer:
  result—option, proc—token, parameter-declarer—pack—option.

parameter-declarer:
  formal-declarer.

A.7 Expressions

An expression is a formula, which is either a primary, or is composed of other formulae by means of operators, according to their priorities. In appendix B, the available operators are listed, together with the types of their operands and results.

expression:
  priority-i-formula.

priority-i-formula:
  priority-ii-formula, rest-priority-i-formula.

rest-priority-i-formula:
  priority-i-operator, priority-i-formula;
  .

priority-ii-formula:
  priority-iii-formula, rest-priority-ii-formula.

rest-priority-ii-formula:
  priority-ii-operator, priority-ii-formula;
  .

priority-iii-formula:
  priority-iii-i-formula, rest-priority-iii-formula.

rest-priority-iii-formula:
  priority-iii-operator, priority-iii-formula;
  .

priority-iii-i-formula:
  priority-iii-ii-formula, rest-priority-iii-i-formula.

rest-priority-iii-i-formula:
  priority-iii-i-operator, priority-iii-i-formula;
  .

priority-iii-ii-formula:
  priority-iii-iii-formula, rest-priority-iii-ii-formula.

rest-priority-iii-ii-formula:
  priority-iii-ii-operator, priority-iii-ii-formula;
  .

priority-iii-iii-formula:
  priority-iii-iii-i-formula, rest-priority-iii-iii-formula.

rest-priority-iii-iii-formula:
  priority-iii-iii-operator, priority-iii-iii-formula;
  .

priority-iii-iii-i-formula:
  priority-iii-iii-ii-formula, rest-priority-iii-iii-i-formula.

rest-priority-iii-iii-i-formula:
  priority-iii-iii-i-operator, priority-iii-iii-i-formula;
  .

priority-iii-iii-ii-formula:
  priority-iii-iii-iii-formula, rest-priority-iii-iii-ii-formula.

rest-priority-iii-iii-ii-formula:
  priority-iii-iii-ii-operator, priority-iii-iii-ii-formula;
  .

priority-iii-iii-iii-formula:
  monadic-operator, priority-iii-iii-iii-formula;
  primary.

A.8 Primaries

primary:
compact-primary;
open-primary.

compact-primary:
simple-primary;
closed-primary.

simple-primary:
denoter;
entity-name.

closed-primary:
choice;
repetition;
display;
open—token, expression, close—token.

choice:
conditional-choice;
numerical-choice.

conditional-choice:
if—token, condition, then-part, else-part—option, end-if—token.

condition:
expression.

A condition must have type BOOL.

then-part:
then—token, paragraph.

else-part:
else—token, paragraph;
elif—token, condition, then-part, else-part—option.

numerical-choice:
select—token, expression, of—token,
case-part—sequence, otherwise-part—option, end-select—token.

The expression after the select—token must have type INT.

case-part:
case—token, case-label—list, colon—token, paragraph.

case-label:
denoter;
synonym-value-name.

otherwise-part:
otherwise—token, paragraph.

repetition:
for-part—option, while-part—option, repeat—token,
repetition-body, until-part—option, end-repeat—token.

for-part:
for—token, variable-name, from-part—option, direction-part—option.

from-part:
from—token, expression.

direction-part:
upto—token, expression;
downto—token, expression.

while-part:
while—token, condition.

repetition-body:
paragraph.

until-part:
until—token, condition.

display:
sub—token, expression—list, bus—token.

open-primary:
call;
subscription;
selection;
abstractor;
concretizer;
terminator.

call:
primary, open—token, actual-parameter—list, close—token.

The primary must be a procedure, to which the actual parameters agree in number and type.

actual-parameter:
expression;
procedure-declarer, procedure-name.

subscription:
primary, sub—token, expression, bus—token.

The primary must be (or yield) a row. The expression must be of type INT.

selection:
primary, period—token, field-name.

The primary must be (or yield) a structure, and the field name must be the name of one of its fields.

abstractor:
type-name, colon—token, compact-primary.

The abstractor serves to denote abstract values.

concretizer:
concr—token, compact-primary.

The concretizer serves to break the abstraction of a value and to obtain its realization.

terminator:
leave—token, algorithm-name, premature-result—option.

A refinement x may be left either from within itself, or from within one of the refinements that x directly or indirectly invokes. The premature-result yields the value, if any, that is yielded by the refinement left.

premature-result:
with—token, compact-primary.

A.9 Names

export-name:
  constant-name;
  procedure-name;
  operator-name;
  type-name.

formal-parameter-name:
  variable-name;
  constant-name;
  procedure-name.

entity-name:
  object-name;
  procedure-name;
  refinement-name;
  synonym-value-name.

algorithm-name:
  procedure-name;
  operator-name;
  refinement-name.

object-name:
  constant-name;
  variable-name.

constant-name:
  identifier.

variable-name:
  identifier.

procedure-name:
  identifier.

refinement-name:
  identifier.

synonym-value-name:
  identifier.

field-name:
  identifier.

packet-name:
  identifier.

type-name:
  bold-identifier.

synonym-type-name:
  bold-identifier.

bold-operator-name:
  bold-identifier.

operator-name:
  bold-identifier;
  special-identifier.

identifier:
  letter, letgit—sequence—option.

letgit:
  letter;
  digit.

bold-identifier:
  bold-letter, bold-letter—sequence—option.

special-identifier:
  equal—token;
  not-equal—token;
  less—token;
  less-equal—token;
  greater—token;
  greater-equal—token;
  plus—token;
  minus—token;
  asterix—token;
  divide—token;
  int-div—token;
  obelix—token;
  becomes—token;
  initial—token.

A.10 Operators

priority-i-operator:
  becomes—token.

priority-ii-operator:
  bold-operator-name.

priority-iii-operator:
  or—token;
  xor—token.

priority-iii-i-operator:
  and—token.

priority-iii-ii-operator:
  equal—token;
  not-equal—token;
  less—token;
  less-equal—token;
  greater—token;
  greater-equal—token.

priority-iii-iii-operator:
  plus—token;
  minus—token.

priority-iii-iii-i-operator:
  asterix—token;
  divide—token;
  int-div—token;
  modulo—token.

priority-iii-iii-ii-operator:
  obelix—token.

Because the asterix—token happens to be an asterisk, the unwieldy sign for exponentiation obtained the name obelix—token.

monadic-operator:
  not—token;
  plus—token;
  minus—token;
  bold-operator-name.

A.11 Denoters

denoter:
  comment—sequence—option, denotation.

denotation:
  int-denotation;
  real-denotation;
  bool-denotation;
  text-denotation.

int-denotation:
  digit—sequence.

real-denotation:
  digit—sequence, period-symbol,
    digit—sequence, exponent-part—option.

exponent-part:
  exponent-symbol, sign—option, digit—sequence.

sign:
  plus-symbol;
  minus-symbol.

bool-denotation:
  true-symbol;
  false-symbol.

text-denotation:
  quote-symbol, text-item—sequence—option, quote-symbol.

text-item:
  quote-image;
  character-image;
  any-character-except-quote-symbol.

quote-image:
  quote-symbol, quote-symbol.

character-image:
  quote-symbol, digit—sequence, quote-symbol.

The character-image serves to denote a character with a specific code (indicated by the digit—sequence).

A.12 Comments

According to the abbreviation rule **a5)**, any symbol may be preceded by a comment. A comment has no meaning within the language (but it may be useful for other reasons).

comment:
  comment-open-symbol, comment-item—sequence—option,
    comment-close-symbol.

comment-item:
  (* any symbol other than a comment-open-symbol or comment-close-symbol *).

A.13 Representations

The representations of the symbols mentioned are as follows:

packet-symbol	PACKET		
endpacket-symbol	ENDPACKET	END PACKET	
defines-symbol	DEFINES		
colon-symbol	:		
comma-symbol	,		
semicolon-symbol	;		
period-symbol	.		
proc-symbol	PROC		
end-proc-symbol	ENDPROC	END PROC	
op-symbol	OP		
end-op-symbol	ENDOP	END OP	
open-symbol	(		
close-symbol	)		
sub-symbol	[		
bus-symbol	]		
initial-symbol	::		
equal-symbol	=		
let-symbol	LET		

type-symbol	TYPE
int-symbol	INT
real-symbol	REAL
bool-symbol	BOOL
text-symbol	TEXT
row-symbol	ROW
struct-symbol	STRUCT
const-symbol	CONST
var-symbol	VAR

if-symbol	IF		
end-if-symbol	FI	ENDIF	END IF
then-symbol	THEN		
else-symbol	ELSE		
elif-symbol	ELIF		
select-symbol	SELECT		
of-symbol	OF		
end-select-symbol	ENDSELECT	END SELECT	
case-symbol	CASE		
otherwise-symbol	OTHERWISE		

repeat-symbol	REP	REPEAT	
end-repeat-symbol	ENDREP	ENDREPEAT	
	END REP	END REPEAT	PER
for-symbol	FOR		
from-symbol	FROM		
upto-symbol	UPTO		
downto-symbol	DOWNTO		
while-symbol	WHILE		
until-symbol	UNTIL		

concr-symbol	CONCR
leave-symbol	LEAVE
with-symbol	WITH
becomes-symbol	:=
or-symbol	OR
xor-symbol	XOR
and-symbol	AND

not-equal-symbol	`<>`	
less-symbol	`<`	
less-equal-symbol	`<=`	
greater-symbol	`>`	
greater-equal-symbol	`>=`	
plus-symbol	`+`	
minus-symbol	`-`	
divide-symbol	`/`	
int-div-symbol	`DIV`	`%`
modulo-symbol	`MOD`	
asterix-symbol	`*`	
obelix-symbol	`**`	
not-symbol	`NOT`	
quote-symbol	`"`	
comment-open-symbol	{	`(*`
comment-close-symbol	}	`*)`

Appendix B

Standard packets

A number of concrete algorithms, objects and types is available in Elan thanks to the *standard packets,* which (are supposed to) precede the program itself and whose exports are therefore available in the program. In particular, this is the way in which the four concrete types and their operations are introduced.

The following overview shows the most important standard algorithms, objects and types of Elan in the forms of declarations or of (procedure or operator) headings. From the headings can be deduced for what types of arguments the various operations are defined, and what their resulting type is.

By means of a dot in the left margin we shall indicate that the particular declaration is also part of Elan-0. The Elan-1 and EUMEL implementations include the full standard prelude. Similarly, a plus in the margin will indicate a (non-standard) extension, available in Elan-0 and Elan-1.

B.1 Priorities of operators

The following list shows the priorities of all operators of the ELAN standard packets, using 10 as the highest level of priority and 1 as the lowest.

1. The assignment `:=` (including the initialization `::`).
2. All abstract dyadic operators (i.e. those defined by the user) as well as `SUB` and the assigning operations `INCR`, `DECR` and `CAT`.
3. `OR XOR`
4. `AND`
5. `= <>`
6. `<= < >= >`
7. Dyadic `+` and `-`.
8. `* / DIV MOD`
9. `**`
10. All monadic operators, such as `+`, `-`, `LENGTH`, `ABS`, `SIGN` and `NOT`.

B.2 Integer

```
. TYPE INT

. PROC get (INT VAR x):
. PROC put (INT CONST x):

. BOOL OP = (INT CONST x, y):
. BOOL OP <> (INT CONST x, y):
. BOOL OP < (INT CONST x, y):
. BOOL OP <= (INT CONST x, y):
. BOOL OP > (INT CONST x, y):
. BOOL OP >= (INT CONST x, y):

. INT OP + (INT CONST x, y):
. INT OP - (INT CONST (x, y):
. INT OP * (INT CONST x, y):
. INT OP DIV (INT CONST x, y):
. INT OP MOD (INT CONST x, y):
  INT OP ** (INT CONST x, y):

. INT OP + (INT CONST x):
. INT OP - (INT CONST x):

. OP INCR (INT VAR x, INT CONST y):
. OP DECR (INT VAR x, INT CONST y):

  INT OP SIGN (INT CONST x):
  INT OP ABS (INT CONST x):
  INT PROC sign (INT CONST x):
  INT PROC abs (INT CONST x):

  BOOL PROC even (INT CONST x):
  BOOL PROC odd  (INT CONST x):

  INT PROC max (INT CONST a, b):
  INT PROC min (INT CONST a, b):

  INT PROC trunc (REAL CONST x):
  INT PROC round (REAL CONST x):
  INT PROC int (TEXT CONST x):

  INT CONST maxint
```

B.3 Real

In Elan-0, the type `REAL` with its operators is not implemented.

```
  TYPE REAL
```

```
PROC get (REAL VAR x):
PROC put (REAL CONST x):

BOOL OP = (REAL CONST x, y):
BOOL OP <> (REAL CONST x, y):
BOOL OP < (REAL CONST x, y):
BOOL OP <= (REAL CONST x, y):
BOOL OP > (REAL CONST x, y):
BOOL OP >= (REAL CONST x, y):

REAL OP + (REAL CONST x, y):
REAL OP - (REAL CONST x, y):
REAL OP * (REAL CONST x, y):
REAL OP / (REAL CONST x, y):
REAL OP / (INT CONST x, y):
REAL OP MOD (REAL CONST x, y):
REAL OP ** (REAL CONST x, INT CONST y):
REAL OP ** (REAL CONST x, y):

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

OP INCR (REAL VAR x, REAL CONST y):
OP DECR (REAL VAR x, REAL CONST y):

INT OP SIGN (REAL CONST x):
REAL OP ABS (REAL CONST x):
INT PROC sign (REAL CONST x):
REAL PROC abs (REAL CONST x):

REAL PROC max (REAL CONST a, b):
REAL PROC min (REAL CONST a, b):

REAL PROC real (INT CONST x):
REAL PROC real (TEXT CONST x):

REAL CONST smallreal, maxreal

REAL CONST pi

REAL PROC sqrt (REAL CONST x):
REAL PROC sin (REAL CONST x):
REAL PROC cos (REAL CONST x):
REAL PROC tan (REAL CONST x):
REAL PROC arctan (REAL CONST x):
REAL PROC arccos (REAL CONST x):
REAL PROC arcsin (REAL CONST x):

REAL CONST e

REAL PROC exp (REAL CONST x):
REAL PROC ln (REAL CONST x):
REAL PROC log10 (REAL CONST x):
REAL PROC log2  (REAL CONST x):
```

B.4 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 x, y):
 . BOOL OP <> (TEXT CONST x, y):
 . BOOL OP < (TEXT CONST x, y):
 . BOOL OP <= (TEXT CONST x, y):
 . BOOL OP > (TEXT CONST x, y):
 . BOOL OP >= (TEXT CONST x, y):


 . INT OP LENGTH (TEXT CONST t):
   INT PROC length (TEXT CONST t):


 . TEXT OP + (TEXT CONST x, y):
 .      OP CAT (TEXT VAR x, TEXT CONST y):
 . 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
p):
        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 t1, t2):
   INT PROC pos (TEXT CONST t1, t2, INT
CONST from):


 + TEXT OP HEAD (TEXT CONST t):
 + TEXT OP TAIL (TEXT CONST t):
 + TEXT PROC ascii (INT CONST code):


   TEXT PROC text (INT CONST i):
   TEXT PROC text (REAL CONST r):


   TEXT CONST niltext :: ""
   TEXT CONST blank :: " "
   TEXT CONST quote :: """"
```

B.5 Boolean

```
 . TYPE BOOL
 . BOOL CONST false, true
```

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

B.6 File handling

In standard Elan, a rather comprehensive file handling system is available, whereas the file handling in Elan-0 is much simpler but not according to the standard.

B.6.1 Standard file handling

The standard file handling is available in the Elan-1 and EUMEL implementations.

```
   TYPE FILE, DIRFILE

   FILE PROC sequential file
(TRANSPUTDIRECTION CONST d,
                               TEXT CONST
ident):
   FILE PROC direct file
(TRANSPUTDIRECTION CONST d,
                           TEXT CONST
ident):
   TRANSPUTDIRECTION CONST input, output,
update

   PROC close (FILE CONST f):
   PROC close (DIRFILE CONST f):

   PROC erase (FILE CONST f):
   PROC erase (DIRFILE CONST f):

   PROC putline (FILE CONST f, TEXT CONST
t):
   PROC getline (FILE CONST f, TEXT VAR
t):

   INT PROC maxlinelength (FILE CONST f):
   INT PROC maxlinelength (DIRFILE CONST
f):
   INT PROC maxpagelength (FILE CONST f):

   PROC line (FILE CONST f):
   PROC line (FILE CONST f, INT CONST i):
   PROC page (FILE CONST f):
   PROC reset (FILE CONST f):

   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 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, TEXT CONST
delimiter):
   PROC get (FILE CONST f, TEXT VAR t, INT
CONST maxlen):

   PROC putline (DIRFILE CONST f, TEXT
CONST key, t):
   PROC getline (DIRFILE CONST f,
                 TEXT CONST key, TEXT VAR
t):

   BOOL PROC opened (FILE CONST f):
   BOOL PROC opened (DIRFILE CONST f):

   BOOL PROC new (FILE CONST f):
   BOOL PROC new (DIRFILE CONST f):
   BOOL PROC eof (FILE CONST f):
```

B.6.2 File handling in Elan-0

The file handling in Elan-0, as described in chapter 9 of this book, is small and simple, but not according to the standard. It is also available in the Elan-1 implementation.

```
+ 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 x):
+ PROC write (TEXT CONST x):
+ PROC writeline :

+      PROC read (INT VAR x):
+      PROC read (TEXT VAR x):
+ BOOL PROC file ended:
```

B.7 Screen handling

```
. PROC line:
. PROC line (INT CONST i):
  PROC page:
+ PROC cursor (INT CONST x,y):
+ PROC get cursor (INT VAR x, y):

  INT CONST max line length

+ PROC edit (TEXT VAR t, INT CONST p):
```

B.8 Generating random numbers

```
  PROC initialize random (INT CONST a):
  PROC initialize random (REAL CONST a):
  INT PROC random (INT CONST a, b):
  REAL PROC random:

+ INT PROC choose128:
```

Bibliography

References chapter *Algorithms.*

[EXN70] Eva Exner: *Braten und Schmoren in Römertopf.* Eduard Bay, D 5412 Ransbach, 1970.

[HER71] H. Hermes: *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit.* Heidelberger Taschenbücher, 1971.
A more mathematical treatment of the concept of algorithm can be found, for example, in this book, giving an introduction to the theory of algorithms.

[HOM79] G. Hommel et al.: *ELAN Sprachbeschreibung.* Akademische Verlagsgesellschaft, Wiesbaden 1979.
The official description of Elan.

[SAM76] J.E. Sammet: *Roster of Programming Languages for 1974-1975.* Communications of the ACM, Vol. 19, No. 12, pp. 655–699, Dec. 1976.
This article reviews the Babylonian confusion in the area of programming languages.

References chapter *Notation of algorithms.*

[PAP80] S. Papert: *Mindstorms.* Children, Computers, and Powerful Ideas. Basic Books, Inc., Harper Colophon Books, 1980.
Gives an account of the philosophy underlying the programming language LOGO and also, in a much more modest form, underlying our use of Karel as a stepping stone to algorithmic thought.

[PAT81] Richard E. Pattis: *Karel the Robot: a gentle Introduction to the Art of Programming.* John Wiley and Sons, 1981.
An introduction to algorithmics, centered wholly around Karel.

References chapter *The whole numbers.*

[ADA72] Richard Adams: *Watership Down.* Penguin Books Ltd, Harmondsworth, England 1972.
The life of a number of rabbits in this turbulent world.

[CRA87] D. Craemer: *Abstrakte Maschinen und modulares Programmieren in ELAN.* GMD ITF Bericht, Bonn 1987.

[KLI85] L.H. Klingen, J. Liedtke: *ELAN in 100 Beispielen.* MikroComputer—Praxis. B.G. Teubner Stuttgart, 1985.

[KNU72] D.E. Knuth: *The Art of Computer Programming.* Vol. 1, Fundamental Algorithms. Addison-Wesley Publishing Company, 1972.
The limit we quoted in this chapter can be found on page 613.

References chapter *The real numbers.*

[DAH74] G. Dahlquist and A. Björk: *Numerical Methods.* Prentice Hall, 1974.
A more advanced textbook, containing a collection of important numerical algorithms.

[GRA78] J. van der Graft: *Introduction to Numerical Computations.* Academic Press, 1978.
Two elementary textbooks on Numerical Analysis.

[RAL65] A. Ralston: *A first Course in Numerical Analysis.* McGraw-Hill, 1965.

References chapter *Truth values.*

[KNU69] Donald E. Knuth: *The Art of Computer Programming.* Vol. 2, Seminumerical Algorithms. Addison-Wesley Publishing Company, 1969.
Section 4.5.4 of this book gives an extensive overview of methods to find prime factors of integers and to decide primality.

References chapter *Composed objects: Rows.*

[KNU73] D.E. Knuth: *The Art of Computer Programming.* Vol. 3: Sorting and Searching. Addison-Wesley Publishing Company, 1973.
The standard work on searching and sorting. One of the two chapters in part 3 of Knuth's *magnum opus* contains a thorough analysis of a number of sorting algorithms.

[MEH81] K. Mehlhorn: *Data Structures and Algorithms 1:* Sorting and Searching. Springer EATCS Monographs on Theoretical Computer Science, 1981.
A solid textbook including much more recent material.

References chapter *Files.*

[HOM83] Hommel, Jähnichen, Koster: *Methodisches Programmieren.* DeGruyter Lehrbuch, Berlin, 1983.
Chapter 11 of this book describes the file operations of Elan and gives an extensive example.

References chapter *Procedures.*

[DAH72] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare: *Structured Programming.* Academic Press, 1972.

[KLE81] K. Kleine, S. Jähnichen, W. Koch, G. Hommel: *Program construction with abstract notions in ELAN.* **in:** Proceedings 3rd World Conference on Computers in Education, Lausanne, North-Holland, 1981.
Deals with Bottom-Up programming.

[MEE77] L.G.L.T. Meertens: *Program text and program structure.* **in:** Constructing quality software, P.G. Hibbard, S.A. Schuman (eds.). IFIP WG2.1/WG2.4 Conference, Novosibirsk, May 1977. North-Holland, 1978, pp. 271–283.
An article about the place of refinements in the development of programs.

References chapter *Languages and grammars.*

[ALGOL68] A. van Wijngaarden (ed.): *Revised Report on the Algorithmic Language ALGOL68.*
The complete formal definition of a programming language, employing a syntactic formalism from which the notation used here was derived.

[BAR60] Y. Bar-Hillel: *The present status of automatic translation of languages.* **in:** Advances in Computers I, 1960.

[WAI84] W.M. Waite, G. Goos: *Compiler Construction.* Springer-Verlag, 1984.

[AHO86] A.V. Aho, R.Sethi, J.D. Ullman: *Compilers, Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1984.

[WOO70] W.A. Woods: *Transition network grammars for natural language analysis.* **in:** Communications of the ACM, Vol. 13, 1970.

[KNU68] D.E. Knuth: *Semantics of context-free languages.* Mathematical Systems Theory 2, 1986.

[KOS70] C.H.A. Koster: *Affix-Grammars.* in: ALGOL68 Implementation, North-Holland, 1971.

[EIG81] M. Eigen, R. Winkler: *Das Spiel.* Naturgesetze steuern den Zufall. München und Zürich, 1981.

References chapter *Recursive algorithms.*

[BUR75] W.H. Burge: *Recursive Programming Techniques.* Addison-Wesley Publishing Company, 1975.
A general introduction to recursive algorithms and their application areas.

[FLE65] J.G. Fletcher: *A Program to Solve the Pentomino Problem by the Recursive Use of Macros.* Communications of the ACM, Vol. 8, No 10, October 1965.
The pentomino problem is a classic. Generations of computer scientists have tried with more or less success to solve this puzzle algorithmically.

[WIR76] N. Wirth: *Algorithms + Data = Program.* Prentice-Hall, 1976.
Contains, amongst others, a chapter about recursive algorithms.

References chapter *Computer graphics.*

[FOD84] J.D.Foley, A.van Dam: *Fundamentals of interactive computer graphics.* Addison-Wesley publ. cy.,1984.

[PEA90] G. Peano: *Sur une courbe, qui remplit toute une aire plaine.* Math. Annln., Vol. 36, pp. 157-160, 1890.

References chapter *Recursive sorting.*

[AHO74] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.
Treats a number of classical algorithms and analyses their time and space complexity.

[DIJ76] E.W. Dijkstra: *A Discipline of Programming.* Prentice-Hall, 1976.
On the stepwise construction of algorithms from their specifications. Contains a large number of inspiring examples.

[EMB70] H.M. van Embden: *Increasing the Efficiency of Quick Sort.* Communications of the ACM, Vol. 13, pp. 563-567, 693-694, 1970.
Gives a mathematical analysis and a fundamental improvement of the Quicksort algorithm.

[HOA62] C.A.R. Hoare: *Quicksort.* Computer Journal, Vol. 5, No 1, pp. 10-15, 1962.
The original article.

References chapter *Backtrack programming.*

[GAR69] Martin Gardner's *New Mathematical Diversions* from Scientific American, Allen and Unwin, London 1969.

[GOL65] S.W. Golomb, L.D. Baumert: *Backtrack Programming.* Journal of the ACM, Vol. 12, No 4, pp. 516-524, October 1965.
A classical article on backtrack programming.

[HOR78] E. Horowitz, S. Sahni: *Fundamentals of Computer Algorithms.* Pitman Publishers, 1978.
Contains (amongst other things) an extended discussion of backtrack programming and Branch-and-Bound methods.

[KNU75] D.E. Knuth: *Estimating the Efficiency of Backtrack Programs.* Mathematics of Computation. Vol. 29, pp. 121-136, 1975.
An article by the master.

[LAW66] E.L. Lawler, D.E. Wood: *Branch-and-Bound Methods: A Survey.* Operations Research. Vol. 14, pp. 699-719, 1966.
A classical article on Branch-and-Bound methods.

[SLA71] J.R. Slagle: *Artificial Intelligence, The Heuristic Programming Approach.* McGraw-Hill Series in Systems Science, 1971.
On the relation between heuristics and problem-solving in the context of Artificial Intelligence.

References chapter *Transforming recursion to iteration.*

[BAU76] F.L. Bauer: *Programming as an Evolutionary Process.* Proceedings of the 2nd International Conference on Software Engineering, IEEE, 1976.
On the meaning of program transformations and their place in the programming process.

[BIR77] R.S. Bird: *Notes on Recursion Elimination.* Communications of the ACM, Vol. 20, No 6, June 1977.
A good starting point; the transformation recursion to iteration lies at the heart of much modern research in informatics.

[SUN71] Y. Sundblad: *The Ackermann function, a theoretical, computational and formula manipulative study.* BIT, Vol. 11, pp. 107-119, 1971.

[WIR71] N. Wirth: *Program Development by Stepwise Refinement.* Communications of the ACM, Vol. 14, No 4, April 1971.
Gives, among other things, a direct derivation of an iterative solution for the 8-Queen problem.

Index