



**Centrum voor Wiskunde en Informatica**  
Centre for Mathematics and Computer Science

---

L.J.M. Geurts

Computer programming for beginners  
Introducing the  $\mathcal{B}$  language, Part I

Department of Computer Science

Note CS-N8402

May

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

COMPUTER PROGRAMMING FOR BEGINNERS  
INTRODUCING THE B LANGUAGE, PART I

L.J.M. GEURTS

*Centre for Mathematics and Computer Science, Amsterdam*

This report contains part 1 of a first course in programming based on the new programming language B.

B is a programming language designed specifically with the beginner, and all other 'non-professional' computer users, in mind. Most elementary programming techniques and most of the features of B are presented. The focus is on designing and writing programs, not actually entering programs in the computer, changing those programs, etc.. Many short programs are shown, or asked to be written by the reader as exercises. The text is self-contained, and may be used in courses or for self-study.

1982 CR. CATEGORIES: D.3.3, D.1.0, K.3.2.

KEY WORDS & PHRASES: programming, programming languages, B.

Note CS-N8402

Centre for Mathematics and Computer Science

P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

## Table of contents

About this book .....	3
1. Arithmetic: the WRITE command .....	4
2. Memory: the PUT command .....	8
3. Repetition: the WHILE command .....	12
4. User-defined commands: HOW'TO .....	18
5. HOW'TO revisited .....	22
6. Texts .....	26
7. Types .....	30
8. Input: the READ command .....	32
9. Conditional execution: the IF and SELECT commands .....	36
10. Lists .....	40
11. Another kind of repetition: the FOR command .....	44
12. Tables .....	48
13. Random choice: the CHOOSE command .....	54
14. Compounds .....	56
Solutions to exercises .....	59

## About this book

This book is a first course in programming based on the new programming language *B*. *B* is a programming language designed specifically with the beginner, and all other 'non-professional' computer users, in mind. Although the skill of programming does not depend completely on the choice of a particular language, it is certainly helpful to use a good language to learn programming with. The programming languages used most often today are from 15 to 25 years old, from the days that ease for the programmer counted less than computer time. These languages are not suitable for a programming course aimed at the beginner, because they are too difficult to learn and too low-level. *B*, on the other hand, has been designed for the generation of very powerful personal computers just appearing, and thus allows you to make the most of such machines.

However, programming isn't easy. The basic difficulty of programming stems from the fact that computers are machines that can, in principle, do anything. Telling the machine which of the infinitely many possibilities it is to perform, cannot be expected to be a very easy task. What *B* does is minimize the difficulties of programming by allowing you to concentrate on the problem at hand, rather than the vagaries of the particular computer you have to use, and its limitations.

### Organization of the book

The book is self-contained, and may be used in courses or for self-study. The focus is on designing and writing programs, not on actually entering programs in the computer, changing those programs, etc. In Part 1, most elementary programming techniques and most of the features of *B* are presented. Many short programs are shown, or asked to be written by the reader as exercises. In Part 2, more advanced techniques and the remaining features of the language are treated. Also, more realistic and interesting examples are given there, and the development of larger programs is studied.

### Exercises

The exercises in this book are an important part. The reader is urged to give as much attention to them as to the rest of the text, since learning to program is quite different to reading about programs and programming. It is the same as learning a foreign (natural) language: speaking and writing a new language is not learnt by reading *about* it, and not really by reading texts written *in* it. It is only by trying to speak and write it, with many errors in the beginning, that one masters a foreign tongue.

### The *B* system

When you actually use a *B* system, you will find out that it does much more for you than executing the *B* commands you give. It helps you type commands, change previously typed commands, etc. Because some of these features are still under development, and can be learnt more easily when you actually try them out, no attention is paid to them in this book. This makes it equally useful for those who learn *B* without having a *B* system at hand.

The only parts of a *B* system that are important in this book are the *keyboard* and the *screen*. You type in commands on the keyboard, and everything you type is shown on the screen. The computer writes the results of the commands on the screen, too. All the letters, digits and other signs used in this book are indeed present on the keyboard. Very long commands, or very long results to be written on the screen, which do not fit on one line of the screen, are simply continued on the next line.

## 1. Arithmetic: the WRITE command

It is well known that computers are very good at arithmetic. Later, we will use them for more interesting things too, but we will start with some simple calculations. You can make the computer do arithmetic by typing the command `WRITE`, followed by whatever formula you want computed.

If, for instance, you want to know the area of a rectangle 8 cm wide and 12.5 cm long, you type:  
and the result comes on the next line:

```
WRITE 8*12.5
□ 100.0
```

As you see, `*` is used for multiplication. A formula such as `8*12.5` is called an *expression*. The sign `*` is called the *operator* of the expression, and 8 and 12.5 are its *operands*.

To find out the perimeter of the same rectangle, you type:  
and you get this *output* (result on the screen):

```
WRITE (8+12.5)*2
□ 41.0
```

The brackets are needed to indicate that the addition should be done before the multiplication. If you omit them, you get:

```
WRITE 8+12.5*2
□ 33.0
```

This is because `*` is done before `+`.

To know the volume of a box with the rectangle above as a base, and with height 1.75 cm, you type:

```
WRITE 8*12.5*1.75
□ 175.000
```

☞ 1. Give a command to calculate the total combined length of the edges of the same box.

Here is how you can find the area of the three different rectangular faces of the same box. As you see, `WRITE` can handle several expressions separated by commas. Note that, in the output, the results are separated by spaces, rather than by commas.

```
WRITE 8*12.5, 12.5*1.75, 1.75*8
□ 100.0 21.875 14.00
```

So far, we used only `+` and `*` as operators. Other possibilities are `-` for subtraction and `/` (pronounced *over*) for division. Here they are used to calculate how many degrees Celsius correspond to 50 degrees Fahrenheit.

```
WRITE "50 F =", (50-32)*5/9, "C"
□ 50 F = 10 C
```

Here we also see that `WRITE` can deal with texts, which we have to type between `"` and `"` and are then used literally in the output.

☞ 2. Change the command above in such a way that it will do the same for 100 degrees Fahrenheit.

As a general rule, remember that `*` and `/` are done before (have higher priority than) `+` and `-`. If there are several `+` and/or `-` signs, they are handled from left to right.

```
WRITE 7*3+10/3-24
□ 0.3333333333333333
WRITE 8-4+3, 8-(4+3)
□ 7 1
WRITE -1+3, -(1+3)
□ 2 -4
```

If you want to use more than one `*` and/or `/` operator, the order in which they are done can make a difference:

```
WRITE (24/6)/2, 24/(6/2)
□ 2 8
WRITE (24/6)*2, 24/(6*2)
□ 8 2
```

In such cases, you must use brackets; but you may omit the brackets if the order of executing `*` and `/` does *not* make a difference (as we already saw in the volume example and the Fahrenheit example):

```
WRITE (18*5)/10, 18*(5/10), 18*5/10
□ 9 9 9
```

For a detailed treatment of priorities, see Part 2.

If you omit the brackets in a combination of \* and / where the order of execution *would* make a difference, then the computer will not execute the command, but will write a message about the error. That happens in general when a command does not follow the rules.

If, for instance, you give this command (I use the X sign to make clear to you that this is wrong):

X WRITE 7/(18-3-15)

you will get an error message looking like this:

☐ \*\*\* Division by 0 is not allowed.

If you ever are in doubt about what is done before what, you can always use extra brackets to make sure, because they do no harm:

WRITE 2+3\*4, 2+(3\*4), ((2)+(((3)\*4)))  
☐ 14 14 14

☞ 3. What is the output of these commands?

- a. WRITE "1+1 is", 1+1
- b. WRITE -1/4, -1/40, -1/400
- c. WRITE 3+9/3, (3+9)/3

☞ 4. Give a command to the computer to write:

- a. the average of the numbers -17, 14.157 and 500.
- b. the thickness of a pile of 142 pages, if each page is 0.013 cm thick.
- c. the average growth per year of a tree that was 1.85 m high in 1837 and 35.30 m in 1984.
- d. 7% of 103.12.
- e. 15% less than 157.

There are some other operators you may use in expressions. I will treat only a small selection here. The rest are in Part 2.

The operator round gives the nearest whole number:

WRITE round 2.4, round 2.5, round 2.6  
☐ 2 3 3  
 WRITE round (1-5/3), 2 \* round 2  
☐ -1 4

The brackets are needed here, because round 1-5/3 could have two different meanings, the other meaning being (round 1)-5/3.

If you want to round to a certain number of digits after the decimal point, you can use round with two operands instead of one. The left operand indicates to how many digits after the decimal point the rounding is done:

WRITE 3 round (2/3), 6 round (800/16)  
☐ 0.667 50.000000

There are two other ways to have a number rounded: floor for rounding down to the nearest whole number that is smaller (or equal), ceiling for rounding up to the nearest greater (or equal) whole number:

WRITE floor (-10/3), floor (91/7)  
☐ -4 13  
 WRITE ceiling (-10/3), ceiling (91/7)  
☐ -3 13

To calculate the remainder after division, there is the operator mod, which needs two operands:

WRITE floor (7/3), 7 mod 3  
☐ 2 1  
 WRITE floor (5.3/1.1), 5.3 mod 1.1  
☐ 4 0.9  
 WRITE floor (60/20), 60 mod 20  
☐ 3 0  
 WRITE floor (3.1/8), 3.1 mod 8  
☐ 0 3.1

5. For each of the following problems, think of a command to make the computer help solve it.

- a. Which whole number is the nearest to the square of 12.7?
- b. How many people can take their *full* share of 1.31 pounds from a supply of 6.7 pounds of chocolate?
- c. How much chocolate will be left?
- d. At least how many baskets are needed to hold 141 eggs, if one basket can hold 8 eggs?
- e. Is 1351 divisible by 7?
- f. How many minutes and seconds correspond to 3859 seconds? The output should look like:  
150 seconds = 2 minutes 30 seconds
- g. How many metres, rounded to the nearest whole centimetre, is a one seventh part of 200 metres?
- h. Imagine 7 children standing in a circle. They have numbers on their backs, and they are standing in the order of their numbers, child 7 standing between child 6 and child 1. If we count to 5, starting at child 1, we end at child 5. If we count to 9, we end at child 2. Where would we end if we counted to 3528?

### Rules

- The operators \* and / go before + and -.
- A sequence of + and/or - operators is handled from left to right.
- A sequence of \* and/or / operators needs brackets, unless the order in which they are handled does not make a difference.





## 2. Memory: the PUT command

If we want to know the areas of the three different rectangular faces of a box, as well as the total outer area, we have to type several expressions twice.

There is a way to avoid this kind of repetition. We can give a name to the result of a calculation, and use that name to indicate the result from then on, as is shown in this *program*, i.e., series of commands.

```
WRITE 22*33.3, 33.3*44.44, 44.44*22
□ 732.6 1479.852 977.68
WRITE 2*(22*33.3+33.3*44.44+44.44*22)
□ 6380.264

PUT 22*33.3 IN a1
PUT 33.3*44.44 IN a2
PUT 44.44*22 IN a3
WRITE a1, a2, a3, (a1+a2+a3)*2
□ 732.6 1479.852 977.68 6380.264
```

What is going on here? Obeying the command `PUT 22*33.3 IN a1`, the computer calculates 732.6 as the result of the multiplication, and stores this result in its memory with a tag `a1` attached to it. One may picture the computer as having a memory in the form of a large sheet of paper, which only the computer can see and change. To obey the command `PUT 22*33.3 IN a1`, the computer writes `a1 = 732.6` in its memory. If, later on, it executes a command such as `WRITE a1+1`, it will look up `a1` in its memory, find 732.6 there, compute 732.6+1 and write 733.6 as output (not in memory, only on the screen). Let us follow a series of commands, and see how it changes the computer's memory. The shaded parts below represent the memory.

Originally, the memory is blank:

After the command:

```
PUT 3+4 IN p
```

the memory holds one *target*, with *tag* `p` and *value* 7:

```
p = 7
```

We can create a new target simply by `PUTting` something `IN` it:

```
PUT 2*3 IN q
```

Now, the memory contains two targets:

```
p = 7 q = 6
```

The value of a target is not fixed for eternity, it can be changed by another `PUT` command:

```
PUT 4 IN p
```

which does not cause a new target to be created, but changes the value of the existing target `p`:

```
p = 4 q = 6
```

We may have the value of a target written on the screen with a `WRITE` command:

```
WRITE p
□ 4
```

which leaves the memory unchanged:

```
p = 4 q = 6
```

Instead of explicitly written numbers, we may use tags in an expression:

```
PUT p+q IN sum
```

resulting in:

```
p = 4 q = 6 sum = 10
```

To copy the value of `p` to `q`, we can use:

```
PUT p IN q
```

which does not change the values of `sum` and `p`:

```
p = 4 q = 4 sum = 10
```

To increase the value of the target `p`, whatever it is, by 2, we can use:

```
PUT p+2 IN p
```

Now the memory contains:

```
p = 6 q = 4 sum = 10
```

It is also possible to PUT several values at once IN several targets. The computer does this by first computing all expressions between PUT and IN, and then putting each result in the corresponding target.

This mechanism may be used to swap the values of two targets in an elegant way.

```
PUT 9/3, 2*2 IN p, q
```

```
p = 3      q = 4      sum = 10
```

```
PUT p*q, p/q IN p, q
```

```
p = 12     q = 0.75   sum = 10
```

```
PUT p, q IN q, p
```

```
p = 0.75   q = 12     sum = 10
```

## Rules

- Targets may have any tag composed of *lower case* letters, digits and the single quote ' , but its first symbol must be a letter.

So, some examples of correct tags are:

```
a a1 a'plus'1 y' difference
```

and some incorrect tags are:

```
X 1'plus'a pH 'yes'
```

Since a tag cannot contain space signs, tags containing several words should either be formed by joining those words without space signs:

```
longedge
```

or better, by using the sign ' to separate the words:

```
long'edge
```

- Of course, you can only PUT something IN a target, not IN a number. So this is wrong:

```
X PUT p-1 IN 5
```

- In a program, each command is typed on a separate line.

☞ 1. What is written on the screen by this little program?

```
PUT 1.001, 19.513 IN d, x
WRITE x-d, x, x+d
```

☞ 2. Give a command to increase the value of target **a** by 50%.

☞ 3. How can you give the target **average** the value that is half way between the values of the targets **p** and **q**?

☞ 4. If **a** and **b** are targets both containing a value, give one or more PUT commands to change the values of **a** and **b** in such a way that both will be the sum of the old values.

☞ 5. The target **distance** contains the number of km a train travels, the target **time** contains the number of minutes it takes. Put the speed of the train, expressed in km/h, in the target **speed**.

☞ 6. Given the number 34, the nearest multiple of 9 is 36. What is the nearest multiple of 9, given the value of the target **n**?

☞ 7. If 1 mile = 1760 yard, 1 yard = 3 feet and 1 foot = 12 inches:

- Give values to the targets **mile**, **yard** and **foot** so that **mile** indicates how many inches are in 1 mile, **yard** how many inches are in 1 yard, and **foot** how many inches are in 1 foot.
- Use these targets to write how long 1 mile, 3 yards, 2 feet and 2.5 inches is in inches.
- Use these targets to write how many miles, yards, feet and inches correspond to 1234567 inches. Give output looking something like:

```
9999999 inch = 13 mile 268 yrd 5 ft 3 inch
```

☞ 8. We have seen that the values of two targets may be interchanged with a 'double' PUT command such as PUT **tar1**, **tar2** IN **tar2**, **tar1**. Try to get the same result without using such a double PUT command. (Hint: feel free to introduce any new target you want, giving it any tag you want.)

☞ 9. Given a target `b` containing a positive whole number (say 341), find a way to add a 4 at the end of the number (so that it becomes 3414). See to it that the program works correctly for any positive whole number that `b` may contain.

☞ 10. The target `pnr` contains a positive number (say 4.13 or 2). Have it changed in such a way that only the part following the decimal point is left (in this case 0.13 or 0). (Hint: use `mod.`)

☞ 11. Find a shorter way to do the same as the following program. Make sure that the result of your solution is the same as the result of these three commands, whatever the original values of `a` and `b` are.

```
PUT a-b IN b
PUT a-b IN a
PUT a+b IN b
```

☞ 12. Do the same for:

```
PUT y, -x IN x, y
PUT y, -x IN x, y
PUT y, -x IN x, y
```



### 3. Repetition: the WHILE command

With the WRITE and PUT commands you can do only rather simple things. A very useful way of achieving more interesting results is having one or more commands repeated several times.

With a WHILE command you can have a piece of program repeated as long as a certain condition is true. Here is a little program that writes all multiples of the number 142857 that are smaller than 500000.

```
PUT 142857 IN mult
WHILE mult < 500000:
    WRITE mult
    PUT mult+142857 IN mult
□ 142857 285714 428571
```

Now, how does this work?

As soon as the computer reaches the WHILE line, the following happens. The computer checks if `mult` is smaller than 500000. (That is what `<` means.) If not so, the *whole* WHILE command, which includes the two indented lines following the WHILE line, is skipped, so the execution of the program has come to an end. But if `mult` is smaller than 500000, then first the indented part is executed, and then the computer goes back to the WHILE line to start all over again.

So, it is hard to tell how many times the indented part is going to be executed, but we know that, once the whole WHILE command has come to an end, `mult` will no longer have a value smaller than 500000.

☞ 1. Change the program above to have it write all multiples of 7 that are smaller than 100.

Expressions such as `mult < 500000` are called *conditions*. Here is a list of the special signs that may be used for building conditions:

`a < b` for: *a is smaller than b*;  
`a > b` for: *a is greater than b*;  
`a = b` for: *a is equal to b*;  
`a <= b` for: *a is smaller than or equal to b*;  
`a >= b` for: *a is greater than or equal to b*;  
`a <> b` for: *a is not equal to b*.

☞ 2. Use the description given above to find out what is written by these three programs:

```
PUT 2 IN a
WHILE a < 3:
    WRITE a
    PUT a+2 IN a
```

```
PUT 2 IN a
WHILE a <= 6:
    WRITE a
    PUT a+2 IN a
```

```
PUT 2 IN a
WHILE a >= 3:
    WRITE a
    PUT a+2 IN a
```

☞ 3. If `n` contains the number 1703, what will it contain after this program has been executed?

```
WHILE n > 17:
    PUT n-17 IN n
```

☞ 4. If `n` contains some positive number, what will be written by this program? (Note that only one number will be written, because the WRITE command is not indented, so it does not belong to the part that is going to be repeated.)

```
PUT 0 IN i
WHILE i < n:
    PUT i+1 IN i
    WRITE i
```

We will now change the 142857 program so that it writes every multiple on a new line, preceded by the number that indicates which multiple it is (so the third line should be 3 428571). The program should stop as soon as one million is reached. The sign / at the end of the WRITE command indicates that further output should be written on the next line. Study this example carefully, making sure that you understand that it will indeed produce the output shown.

```
PUT 1, 142857 IN nr, mult
WHILE mult < 1000000:
  WRITE nr, mult /
  PUT nr+1, mult+142857 IN nr, mult
□ 1 142857
□ 2 285714
□ 3 428571
□ 4 571428
□ 5 714285
□ 6 857142
□ 7 999999
```

Let us analyse the roles of the targets that are crucial to this WHILE command: the targets *nr* and *mult*. Loosely speaking, *nr* plays the role of the serial number of the next output line, and *mult* is the multiple of 142857 whose turn it is to be written. More compactly, *nr* is the line number and  $\text{mult} = \text{nr} * 142857$ . In general, it is important to keep the roles of the crucial targets of a WHILE command firmly in mind. In this context, a target is *crucial* if its value may be changed in the WHILE command, and if that value is still important for the next time the indented part is executed. (In this example, all targets are crucial in this sense, but later on we will see WHILE commands containing other targets too.)

The roles of the crucial targets are best formulated as conditions that are true each time the indented part of the WHILE command is going to be executed. Here are some versions of the same program. Pay special attention to the descriptions of the roles of the crucial targets.

Here is a version of the same program based on the same roles for the crucial targets, so again: *nr* is the serial number of the next output line, and  $\text{mult} = \text{nr} * 142857$ .

```
PUT 1, 142857 IN nr, mult
WHILE mult < 1000000:
  WRITE nr, mult /
  PUT nr+1 IN nr
  PUT nr*142857 IN mult
```

The first time the computer arrives *at the start of the indented part*, no output has yet been written and the values of the crucial targets fit their descriptions, since they are:

```
nr = 1  mult = 142857
```

The next time the computer passes this point, one output line has been written, and the crucial targets have these values, which fit their roles again:

```
nr = 2  mult = 285714
```

The next time, after two output lines, the values fit their description again, and so it will be every time:

```
nr = 3  mult = 428571
```

It is also possible to write a version of this program based on these slightly different roles for the crucial targets:

*nr* is the number of lines written so far, and again  $\text{mult} = \text{nr} * 142857$ .

```
PUT 0, 0 IN nr, mult
WHILE mult+142857 < 1000000:
  PUT nr+1, mult+142857 IN nr, mult
  WRITE nr, mult /
```

Here is a version with only one crucial target: *nr*, which indicates the serial number of the next line to be written. It is a matter of taste which version is preferable. An important criterion is: which version is the easiest to understand, so that it is as clear as possible that there are no errors in it. Having fewer targets in the program often helps.

```
PUT 1 IN nr
WHILE nr*142857 < 1000000:
  WRITE nr, nr*142857 /
  PUT nr+1 IN nr
```

☞ 5. Design still another version of the same program, based on one crucial target: *nr*, indicating the number of lines written so far.

Here is a program to show how a capital of 1000 units of currency grows if every year 13 percent is added to it. The program stops as soon as the capital has doubled. I use the target `sum` to hold the sum of money, and the target `y` to keep track of the year. Note that the bank rounds the sum to the nearest hundredth every year.

```
PUT 1984, 1000 IN y, sum
WHILE sum < 2000:
    PUT y+1 IN y
    PUT 2 round (sum*1.13) IN sum
    WRITE y, sum /
□ 1985 1130.00
□ 1986 1276.90
□ 1987 1442.90
□ 1988 1630.48
□ 1989 1842.44
□ 1990 2081.96
```

☞ 6. Change the program above in such a way that the bank calculates the sum exactly every year, the rounding being done in the output only.

Keeping a close eye at the roles of the crucial targets is not our only burden when programming a `WHILE` command. We must also see to it that it will come to an end.

Take this little program for writing odd numbers. It is easy to see that it will happily start to write odd numbers, but the problem is that it will never stop doing so.

```
PUT 1 IN a
WHILE a > 0:
    WRITE a
    PUT a+2 IN a
```

And here is another program that will never stop. (A practical solution if you make a mistake like this is to use the special stop key to stop the execution of the program.)

```
PUT 1 IN a
WHILE a<>10:
    WRITE a
    PUT a+2 IN a
```

☞ 7. Even though both previous programs are hardly correct, it is possible to describe the role of the crucial target `a`. What is this role?

We may build a condition from other conditions by combining them with `AND`, as shown in this program for writing the multiples of 7 under 1000 until the first multiple ending in 33 has been written. In this example with the word `AND`, the repetition goes on as long as both `m+7 < 1000` and `m mod 100 <> 33`.

```
PUT 0, 0 IN nr, m
WHILE m+7 < 1000 AND m mod 100 <> 33:
    PUT nr+1 IN nr
    PUT nr*7 IN m
    WRITE nr, m /
□ 1 7
□ 2 14
□ 3 21
□ 4 28
□ 5 35
□ 6 42
□ 7 49
□ 8 56
□ 9 63
□ 10 70
□ 11 77
□ 12 84
□ 13 91
□ 14 98
□ 15 105
□ 16 112
□ 17 119
□ 18 126
□ 19 133
```

In some combinations of conditions, `AND` is not necessary. Instead of `WHILE -3 < p AND p <= 7` we may write `WHILE -3 < p <= 7`. Similarly, `WHILE a = b < 4` means the same as `a = b AND b < 4`.



Conditions may also be combined with OR. Here is an example where *a* is halved as long as it is greater than 10 or smaller than -10:

```
WHILE a > 10 OR a < -10:
    PUT a/2 IN a
```

If we want repetitions to go on as long as a certain condition is *not* true, we may use NOT, as shown in this program, which is equivalent to the program above:

```
WHILE NOT -10 <= a <= 10:
    PUT a/2 IN a
```

## Rules

- In a WHILE command, the commands to be executed repeatedly have to be typed with indentation. Any following commands not to be repeated should be lined up with the WHILE line again.
- If AND, OR and NOT are combined in a condition, brackets are needed:
- In an AND combination, the computer stops checking at the first condition that is false; in an OR combination, it stops at the first true condition.

```
WHILE (a > 1 AND b > 2*a) OR a = b:
```

## Advice

- When writing the indented part of a WHILE command, you should imagine the situation when the process is somewhere halfway, rather than imagining the first repetition, which is often rather special.

For instance, when designing a program to write the sequence 2, 4, 8, etc., you might be tempted to start this way:

```
PUT 1 IN nr
WHILE nr < 1000:
    WRITE 2
    etc.
```

This kind of mistake stems from focusing on the first repetition, instead of the general case some time later on. When you imagine such a later moment in the process, it is clear that the program needs a target such as *nr* to remember the number most recently written. In every repetition *nr* should be doubled, and the result should be written:

```
PUT 1 IN nr
WHILE nr < 1000:
    PUT 2*nr IN nr
    WRITE nr
```

On the other hand, after writing a WHILE command, you should always check that what happens during the first repetition, and during the last, is correct, because it is here that most errors occur.

- A WHILE command should be designed with the roles of the crucial targets in mind, which should be true each time the indented part is going to be executed.
- In the examples above, we have seen a general pattern in the way the descriptions of the roles of the crucial targets are used. Here is a demonstration of this pattern applied to one of the programs for writing multiples of 142857, where the target *nr* indicated the serial number of the next output line to be written.

1. Before the WHILE command, we give each crucial target a starting value fitting its role:

```
PUT 1 IN nr
```

2. In the WHILE line, we use one or more of the crucial targets to indicate how long the repetition should go on:

```
WHILE nr*142857 < 1000000:
```

3. In the indented part we take some step towards the goal of the program. In this case, such a step is writing the next line of output:

```
WRITE nr, nr*142857 /
```

4. After such a step has been made, the description of the roles will, in general, not fit any more, so now we change the values of the crucial targets to make them fit the description again:

```
PUT nr+1 IN nr
```

8. Write programs that write those numbers of the following sequences that are under 1000. For each program, state the roles of the crucial targets.

a. 1 2 4 8 ...

b. 1 4 9 16 ...

c. 1 2 6 24 ...

(Hint:  $1*2*3*4*...$ )

d. 111 111 222 222 333 333 444 ...

e. 0 1 1 2 3 5 8 13 ...

(Hint:  $13 = 5+8$ )

9. The target `start` contains a positive whole number. Write a program that writes this number, then gives `start` twice that number as value and writes the new value, doubles it again, and so on until a number has been written that ends in 00, 08 or 80.

10. The targets `length` and `width` contain values indicating the size of a rectangular sheet of paper, e.g., `length` = 15, `width` = 10. Write a program to write the size of the sheet after it has been folded in half (dividing `length` in two), then after it has been folded in half again perpendicularly to the first fold, then again perpendicularly to the second fold, and so on. So, the first two output lines for the example would look like:

```
10 7.5
```

```
7.5 5
```

Let the folding stop as soon as the side to be folded is smaller than 1. Also, formulate the roles of the crucial targets in your program.

Also in cases where only one item of output is wanted, a `WHILE` command may be useful. If we want to know the smallest positive whole number `a` for which `a*a+a` is greater than 100, we may proceed by first trying `a` = 1, then if it is not yet true, `a` = 2, etc., until we hit on a value where it is true:

```
PUT 1 IN a
WHILE NOT a*a+a > 100:
    PUT a+1 IN a
WRITE a
```

The role of `a` here is: all positive whole numbers `i` smaller than `a` have: `NOT i*i+i > 100`. You can easily check that this holds at the beginning, and once it is true, it stays that way until the `WHILE` command is over. Once it is over, we know both that `a*a+a > 100` and that for smaller positive whole numbers this is not so. From these two facts, we may safely conclude that `a` contains the wanted number.

11. Write a program along the lines of the program above, to find the smallest positive even number which has a square greater than or equal to the value of `s`.

Another example of a `WHILE` command that may be executed many times to give only one item of output is this program to find the sum of the numbers 1, 2, 6, 24, ... that are under 1000. (Note that the fourth number is 4 times the third number.) The roles of the crucial targets are:

`i` is the serial number in the sequence of the next number to be multiplied,  
`p` is the `i-1`th number of the sequence, and  
`s` is the sum of the first `i-2` numbers of the sequence (so `p` is the next number to be added if it is smaller than 1000).

```
PUT 0, 1, 2 IN s, p, i
WHILE p < 1000:
    PUT s+p, p*i, i+1 IN s, p, i
WRITE s
□ 873
```

12. Write a new version of the program above, basing it on the following roles for the crucial targets:  
`s` is the sum of the first `i` numbers of the sequence,  
`p` is the `i`-th number of the sequence, so  
`i` is the serial number of the number at hand.

13. Write a program to find the first number over 1000 in the sequence 0 1 1 2 3 5 8 13 ...

☞ 14. Write a program that gives as output the smallest *factor* of the whole number in target  $n$ , which is greater than 1. A factor is a whole number greater than 1, which exactly divides  $n$ , so  $n$  has at least one factor:  $n$  itself.

We have seen two advantages offered by targets:

1. They enable us to avoid tedious repetition of expressions.
2. They play the crucial roles in WHILE commands. In fact, it is impossible to write a sensible WHILE command not involving targets.

#### 4. User-defined commands: HOW'TO

We have seen how we can get rid of some types of tedious repetition in our program texts by using targets. There are, however, forms of repetition that cannot be avoided in this way.

If we want to see all multiples of 142857 as well as all multiples of 123456 under 500000, we have to write this program.

```

PUT 142857 IN mult
WHILE mult < 500000:
  WRITE mult
  PUT mult+142857 IN mult
□ 142857 285714 428571
PUT 123456 IN mult
WHILE mult < 500000:
  WRITE mult
  PUT mult+123456 IN mult
□ 123456 246912 370368 493824

```

Now, if only *B* had the special command MULTIPLY  $\times$  UNDER  $y$  with just the right meaning, the program above could be shortened to only two commands:

```

MULTIPLY 142857 UNDER 500000
MULTIPLY 123456 UNDER 500000

```

Well, *B* does not have this as a built-in command, but there is a way to define new commands yourself. We need only tell the computer how to execute such a command. Here you see how we can do that for our own MULTIPLY command. (WRITE / means: go to the next line of the screen, so that any further output will start there.)

```

HOW'TO MULTIPLY single UNDER limit:
  PUT single IN mult
  WHILE mult < limit:
    WRITE mult
    PUT mult+single IN mult
  WRITE /

```

In this definition, MULTIPLY and UNDER are called the *keywords*, single and limit are the *slots*. Once we have given this definition, it is not executed, but stored in the computer's *definition memory*. From now on, we may use the new command as freely as PUT IN or any other command belonging to the language.

Here is an application of the newly defined MULTIPLY command.

```

MULTIPLY 142857 UNDER 500000
□ 142857 285714 428571

```

In this application, MULTIPLY and UNDER are the *keywords*, equal to those of the command definition; 142857 and 500000 are the *parameters*, matching the slots of the command definition.

☞ 1. Use the MULTIPLY command to have the computer write the positive even numbers up to and including 50.

Let us now follow the steps the computer goes through in executing the following little program:

```

PUT 50 IN maximum
MULTIPLY 7 UNDER maximum-1

```

1. The computer gives the target maximum a value. Then the *target memory* (as opposed to the definition memory mentioned above) will contain:

```

maximum = 50

```

2. The computer now searches the definition memory for a definition beginning with HOW'TO MULTIPLY and checks that the second keyword is UNDER. If it is not there, or the second keyword is wrong, the computer reports an error; otherwise, it copies the definition in a temporary piece of extra memory:

```

HOW'TO MULTIPLY single UNDER limit:
  PUT single IN mult
  WHILE mult < limit:
    WRITE mult
    PUT mult+single IN mult
  WRITE /

```

3. In this copy, the computer changes every occurrence of the slots `single` and `limit` into `(7)` and `(maximum-1)` respectively. You will see the reason for the brackets later.

4. The computer discards the top line of the copy, and executes the resulting program.

This gives as output on the screen:

5. Finally, the computer throws away the temporary piece of extra memory, leaving the original command definition in the definition memory. The target memory is now still as follows. (Notice that `mult` does not remain. This will be explained later.)

Note that the computer goes through the steps above without showing anything, except of course the results of any `WRITE` commands it encounters. So, neither the target memory, nor the temporary piece of extra memory is shown on the screen.

2. What happens when this command is given:

```
HOW'TO MULTIPLY (7) UNDER (maximum-1):
  PUT (7) IN mult
  WHILE mult < (maximum-1):
    WRITE mult
    PUT mult+(7) IN mult
  WRITE /
```

```
PUT (7) IN mult
WHILE mult < (maximum-1):
  WRITE mult
  PUT mult+(7) IN mult
WRITE /
```

□ 7 14 21 28 35 42

maximum = 50

MULTIPLY -3 UNDER 10

We can make a command definition out of any program we write. It may then be used over and over again, with only very little effort.

Normally, if we want a program to split off the rightmost digit of the numerical value of target `a`, and keep this digit in the target `rightmost`, we would write the following little program. (Remember that `mod` calculates the remainder after division.)

If we give this program the form of a command definition:

```
PUT a mod 10 IN rightmost
PUT (a-rightmost)/10 IN a
```

```
HOW'TO SPLIT'RIGHT number TO digit:
  PUT number mod 10 IN digit
  PUT (number-digit)/10 IN number
```

we can get the same result as above by typing:

SPLIT'RIGHT a TO rightmost

Furthermore, we can use this new command every time we want to isolate the rightmost digit of a number, also when defining another command, such as this one for finding the sum of the digits of a number. (In the `WHILE` command, `sum` contains the sum of those digits of number that are no longer in `n`.)

```
HOW'TO SUM'DIGITS number:
  PUT number IN n
  PUT 0 IN sum
  WHILE n > 0:
    SPLIT'RIGHT n TO rightmost
    PUT sum+rightmost IN sum
  WRITE sum
```

Using the new command:

SUM'DIGITS 13\*13

gives as output:

□ 16

This is correct, since  $13*13=169$ , and the sum of the digits of 169 is 16. To see what the computer has to do to find this answer, look at what is left after inserting the parameter in the slot of the command definition:

```
PUT (13*13) IN n
PUT 0 IN sum
WHILE n > 0:
  SPLIT'RIGHT n TO rightmost
  PUT sum+rightmost IN sum
WRITE sum
```

To see in more detail what happens, look at what becomes of `SPLIT'RIGHT n TO rightmost` after its definition has been changed according to the parameters `n` and `rightmost`, and after the resulting text has been put in the place of the `SPLIT'RIGHT` command.

```
PUT (13*13) IN n
PUT 0 IN sum
WHILE n > 0:
    PUT (n) mod 10 IN (rightmost)
    PUT ((n)-(rightmost))/10 IN (n)
    PUT sum+rightmost IN sum
WRITE sum
```

The elaborate demonstration above of how user-defined commands work is only for you to get acquainted with the mechanism, not as an encouragement to try out by hand every application of a self-defined command, although that may be useful occasionally when the computer produces unexpected results.

### Rules

- Command definitions are stored in the computer's definition memory, and remain there until we take action to throw them away (using the editor). It is always possible to inspect these definitions, to change them or give them different names.
- Apart from the first line, a command definition should be typed with indentation. This may give rise to double indentation if a `WHILE` command occurs in the definition.
- Keywords of user-defined commands look the same as tags, only they have *upper case* letters instead of lower case. So, examples of correct keywords are: `PUSH`, `GIVE'2ND'NUMBER`, `ADD'`.
- Slots look the same as tags.

- Two different command definitions should have different first keywords. So, if we already have this command definition:

```
HOW'TO ADD d TO a:
    PUT a+d IN a
```

we cannot add this definition:

```
X HOW'TO ADD a AND b:
    WRITE a, "+", b, "=", a+b /
```

- The parameters in an application of a user-defined command can be any expression, including of course single targets (like `p`) and single numbers (like `3.14`).

### Advice

- When defining a new command (say, for writing both the sum and the product of two numbers), one should first decide which items of information one may want to specify differently every time the command will be used (in this case the two numbers), and how one wants an application of the new command to look (e.g., `ADD'AND'MULT 17 AND 36`). Once that has been decided, it is easy to write the first line of the definition (`HOW'TO ADD'AND'MULT a AND b:`).

- It is wise to choose a verb as the first keyword, so that an application of the command really reads as a command. In general, command definitions are easier to understand if the keywords, slots and tags of internal targets are chosen in such a way that they suggest their roles to the human reader.

☞ 3. Describe the meaning of this command definition, and suggest a better wording.

```
HOW'TO WUZZLE word:
    PUT word*word IN word
```

☞ 4. Do the same for:

```
HOW'TO QRZ pppp M ppp:
    PUT ppp, pppp IN pppp, ppp
```

☞ 5. Define a command for writing both the square and the cube of a given number. It should be possible to use it in this way:

```
SQUARE'N'CUBE 9
□ 81 729
```

☞ 6. If someone were not happy with *B*'s `PUT` command, preferring `LET a BE 3` to `PUT 3 IN a`, how could they define their own `LET` command?

7. Here is a definition of a command that shows how interest is added to a sum of money until it has doubled. Change this definition in such a way that the interest percentage is not fixed at 15, but has to be specified every time the command is used.

8. With the help of the following command definition, it is possible to generate the sequence 1, 1, 2, 3, 5, 8, 13 ... ( $13 = 5 + 8$ ) as long as it stays under 1000.

a. How would you use it to generate the sequence 2, 5, 7, 12, 19 ... ( $19 = 7 + 12$ ) as long as it stays under 500?

b. How would you change the definition in order to allow sequences such as

1, 1, 5, 13, 41, 121 ... ( $121 = 3 \cdot 13 + 2 \cdot 41$ ) and  
1, 3, 8, 21, 55, 144 ... ( $144 = -21 + 3 \cdot 55$ )?

c. How do you use your new definition to generate each of the four sequences mentioned?

HOW TO DOUBLE sum:

```
PUT 1984, sum IN year, start
WHILE sum < 2*start:
  PUT year+1 IN year
  PUT 2 round (sum*1.15) IN sum
  WRITE year, sum /
```

HOW TO ADD FROM p AND q UNDER limit:

```
PUT p, q IN a, b
WRITE a
WHILE b < limit:
  WRITE b
  PUT b, a+b IN a, b
```

## 5. HOW'TO revisited

Looking more closely at user-defined commands, we will observe some complications that may arise.

### 5.1. An expression used where a target is needed

Here is the definition of a command that increases the value of a target by 1.

It may be used in this way:

```
HOW'TO INCR a:
  PUT a+1 IN a
```

```
PUT 3 IN number'of'occurrences'so'far
INCR number'of'occurrences'so'far
WRITE number'of'occurrences'so'far
□ 4
```

but not in this way:

```
X INCR 3
```

Why is this wrong? Loosely speaking, because changing the value of 3 into 4 is meaningless. Looking more closely, we see that INCR 3 boils down to PUT (3)+1 IN (3), which is clearly wrong, because the second parameter of a PUT command must be a target.

Another example of the same thing is this program that keeps doubling some initial value as long as it stays under some limit.

```
HOW'TO DOUBLE n UNTIL limit:
  WHILE n < limit:
    WRITE n
    PUT n+n IN n
```

This command may be used in this way:

```
PUT 3 IN p
DOUBLE p UNTIL 100
□ 3 6 12 24 48 96
```

but not in this way:

```
X DOUBLE 3 UNTIL 100
```

This is wrong, because insertion of the parameters in the slots turns out to lead to a wrong PUT command:

```
WHILE (3) < (100):
  WRITE (3)
  X PUT (3)+(3) IN (3)
```

If we want to allow DOUBLE 3 UNTIL 100, we should introduce an extra target into the definition. This extra target, which will only play a role inside the DOUBLE command, is called an *internal* target, as opposed to the usual *external* targets. (We have seen internal targets earlier; the most recent example was *year* in the definition of the DOUBLE command of exercise 4.7.)

```
HOW'TO DOUBLE n UNTIL limit:
  PUT n IN a
  WHILE a < limit:
    WRITE a
    PUT a+a IN a
```

☞ 1. How does this definition look after parameter insertion, when used with:

```
DOUBLE 3 UNTIL 100
```

☞ 2. Why is it not possible to change the definition of INCR in such a way that it is no longer obligatory to use it with a target as parameter?

By looking inside a command definition, it is always possible to tell whether a slot must be filled with a parameter which is a *target*, or may be filled with *any expression*.

☞ 3. Determine this for:

```
HOW'TO REDUCE k BELOW n:
  WHILE k >= n:
    PUT k/2 IN k
  WRITE k
```



## 5.2. Internal targets and external targets

One of the roles of user-defined commands is to make programs easier to understand by dividing them into meaningful chunks. Practical programs tend to be larger than the examples shown so far. It is difficult to avoid programming errors in such large programs, and therefore it is important to keep programs as readable to oneself as possible. The advantage of user-defined commands for this purpose is that we need to understand the details of such a command only once: when we write it. Later on, it is sufficient to remember the keywords of the command and *what* is achieved by the command, not *how* it is done. In particular, we don't want to have to remember the names of the slots and the tags of the internal targets.

Take this (admittedly clumsy) definition of a command to interchange the values of two targets. Here, *z* is an internal target, and *a* and *b* are slots.

If we use this command, we don't want it to spoil the target *z* we may already have as an *external* target, so we want it to work this way:

Similarly, when we use the SWAP command, the value of any external target *a* or *b* should remain unchanged, unless it is used as a parameter:

HOW'TO SWAP *a* AND *b*:

```
PUT a IN z
PUT b IN a
PUT z IN b
```

```
PUT 4 IN z
PUT 8, 5 IN p, q
SWAP p AND q
WRITE p, q, z
□ 5 8 4
```

```
PUT 3, 1, 5 IN a, b, z
SWAP a AND z
WRITE a, b, z
□ 5 1 3
```

In order to guarantee that these wishes are fulfilled, the computer gives each internal target a unique tag by adding ' signs to the tag until it differs from every other target. At the end of this chapter, you will see a complete description of the copying mechanism.

## 5.3. Careful substitution

If we have this definition of a command to divide a target's value by a given number:

then it may be used in this way:

because after substitution, the command definition becomes:

Naturally, the result should be the same if DIVIDE is used in this very slightly different way:

because substitution in brackets gives:

If substitution took place without brackets, we would get the wrong result, since *s* would first be divided by 2 before 1 is added:

In cases where these brackets are not necessary, they do no harm either.

HOW'TO DIVIDE *a* BY *b*:

```
PUT a/b IN a
```

```
PUT 12 IN s
DIVIDE s BY 3
WRITE s
□ 4
```

```
PUT (s)/(3) IN (s)
```

```
PUT 12 IN s
DIVIDE s BY 2+1
WRITE s
□ 4
```

```
PUT (s)/(2+1) IN (s)
```

```
PUT s/2+1 IN s
```

### 5.4. Shared targets

So far, the only path for information into and out of a command is via its slots, because all other tags occurring in a command definition are internal targets. In some cases, this is a nuisance.

If, for instance, we want to process a list like:

5 bottles of wine at 1.95 each  
2 hams at 30.50 each  
1 bar of soap at 1.35

To find out the total cost and the total number of items, we could proceed like this:

```
PUT 0, 0 IN its, cost
PUT its+5, cost+5*1.95 IN its, cost
PUT its+2, cost+2*30.50 IN its, cost
PUT its+1, cost+1.35 IN its, cost
WRITE its, cost
```

For a long list, one would like to have a special command for the purpose:

```
HOW'TO ADD nr AT price TO tnr AND tc:
  PUT nr*price IN c
  PUT tnr+nr, tc+c IN tnr, tc
```

However, the application of this definition is as tedious as the original approach:

```
PUT 0, 0 IN its, cost
ADD 5 AT 1.95 TO its AND cost
ADD 2 AT 30.50 TO its AND cost
ADD 1 AT 1.35 TO its AND cost
□ 8 72.10
```

The problem is that the command definition needs so many slots. It is, however, possible to do away with the two slots *its* and *cost*, using instead a *SHARE* command to specify that the *external* targets *its* and *cost* should always be used by the command:

```
HOW'TO ADD nr AT price:
  SHARE its, cost
  PUT nr*price IN c
  PUT its+nr, cost+c IN its, cost
```

Using the new version is more attractive:

```
PUT 0, 0 IN its, cost
ADD 5 AT 1.95
ADD 2 AT 30.50
ADD 1 AT 1.35
WRITE its, cost
□ 8 72.10
```

☞ 4. Define a command that would help you to add a list of numbers, giving a subtotal every time a number has been added. It should be possible to use it this way:

```
A 45
□ 45
A 13
□ 58
A -4/2
□ 56
```

### Rules

- Any *SHARE* commands in a command definition should follow the *HOW'TO* line immediately, preceding the other commands.

- Here follows a complete description of the procedure the computer follows when executing a user-defined command.

To make the description easier to understand, the results are shown for a simple example, in which the target memory originally is:

```
cost = 9.75  its = 5  nr = 4
c = 3  c' = 1
```

and the command at hand is this application of the definition we have just seen:

```
ADD 2 AT 30.50
```

1. The memory is extended temporarily (an extra piece of paper is pasted to it), and the command definition is copied there.

2. Each internal target (i.e. tag in the copy that is neither a slot nor a SHARED target) is made unique, i.e. changed systematically by adding as many ' symbols at the end as necessary to make them different from the other targets.

3. The parameters, put in brackets, are inserted in the corresponding slots.

4. The head line of the copy, and the SHARE lines if any, are discarded.

5. The modified copy is now executed, the extra memory being used for the internal targets. The result for this example shows that the external targets *c* and *c'* are not spoiled by the coincidence that there was also an internal target *c* in the command definition. In the same way, the external target *nr* remains unharmed, although it has a tag which is the same as one of the slots. We also see that the shared targets *its* and *cost* are properly changed.

6. Finally, the extra piece of memory is thrown away.

• External targets are *permanent*, i.e., they remain until we explicitly delete them, in a way you will see later. Internal targets, on the other hand, only exist during the execution of the command in the definition of which they occur. If the same tag is used in another definition, then it belongs to another target.

☞ 5. Given this command definition:

```
cost = 9.75  its = 5  nr = 4
c = 3  c' = 1
```

```
HOW'TO ADD nr AT price:
  SHARE its, cost
  PUT nr*price IN c
  PUT its+nr, cost+c IN its, cost
```

```
cost = 9.75  its = 5  nr = 4
c = 3  c' = 1
```

```
HOW'TO ADD nr AT price:
  SHARE its, cost
  PUT nr*price IN c''
  PUT its+nr, cost+c'' IN its, cost
```

```
cost = 9.75  its = 5  nr = 4
c = 3  c' = 1
```

```
HOW'TO ADD (2) AT (30.50):
  SHARE its, cost
  PUT (2)*(30.50) IN c''
  PUT its+(2), cost+c'' IN its, cost
```

```
cost = 9.75  its = 5  nr = 4
c = 3  c' = 1
```

```
PUT (2)*(30.50) IN c''
PUT its+(2), cost+c'' IN its, cost
```

```
cost = 70.75  its = 7  nr = 4
c = 3  c' = 1
```

```
PUT (2)*(30.50) IN c''
PUT its+(2), cost+c'' IN its, cost

c'' = 71
```

```
cost = 70.75  its = 7  nr = 4
c = 3  c' = 1
```

```
HOW'TO MULT n:
  SHARE pp, f
  PUT n*f IN p
  WRITE p
  PUT pp*p IN pp
```

and this memory situation:

```
n = 4  p = 8  k = 6  pp = 2  f = 1.5
```

- Try to predict what the output and the memory will be after the execution of the command `MULT k+1`.
- Verify your guess by going through the steps given above.
- Describe the effect of an application of `MULT` in general terms.

## 6. Texts

We have already seen the use of texts in WRITE commands:

Using this possibility, we can define a command to write the text `Merry Christmas` framed in a rectangle of asterisks. (This command has only one keyword and no slots, which is perfectly all right.)

Using this command gives:

A new aspect of texts is that they may be stored in targets, just like numbers. This gives the possibility of defining WISH thus:

Of course, the operators we have seen for numbers are of no use for textual values, since there is no obvious meaning to expressions such as:

However, there are some special operators for texts. To glue texts together, obtaining one text, the operator `^` is used.

For repeating a text to obtain one new text, the operator `^^` is used. (Think of repeated gluing.)

For obtaining the number of characters in a text, there is the operator `#` (pronounced *length*). A *character* is a letter or a digit or one of the other things a text is made of, including the space character. (Note the difference between the text `"time"` and the tag `time`.)

1. Define a command `UNDERLINE` which writes a given text followed by an exclamation mark, and on the next line a series of `-` characters of the same length. The command should allow applications such as:

2. The target `answer` contains a text of at most 5 characters. Give a program to extend it to the right with enough `!` characters to make it exactly 10 characters long.

```
PUT 3, 4 IN n, d
WRITE n, "divided by", d, "=", n/d
□ 3 divided by 4 = 0.75
```

HOW/TO WISH:

```
WRITE "*****" /
WRITE "*" /
WRITE "* Merry Christmas *" /
WRITE "*" /
WRITE "*****"
```

WISH

```
□ *****
□ *
□ * Merry Christmas *
□ *
□ *****
```

HOW/TO WISH:

```
PUT "*****" IN stars
PUT "*" IN blank
WRITE stars /
WRITE blank /
WRITE "* Merry Christmas *" /
WRITE blank /
WRITE stars
```

`X ("Merry Christmas"*"yes") mod 3`

```
PUT "now", "here" IN time, place
WRITE time^place
□ nowhere
```

```
WRITE "No!"^^3
□ No!No!No!
```

```
WRITE #time, #"time", #"an ode"
□ 3 4 6
WRITE #("No!"^^3)
□ 9
```

```
UNDERLINE "Shriek"
□ Shriek!
□ -----
```

These new operators may be used for still another version of the WISH command. (The WRITE command gives no space between two text items.)

The new version does not look especially attractive, but it has the advantage that it may easily be generalized to be used to frame *any* text:

such as:

or even the *empty* text:

#### HOW TO WISH:

```
PUT "Merry Christmas" IN words
PUT "*" ^ #words IN stars
PUT " " ^ #words IN blank
WRITE "***", stars, "***" /
WRITE "*" , blank, " " /
WRITE "*" , words, " " /
WRITE "*" , blank, " " /
WRITE "***", stars, "***"
```

#### HOW TO FRAME words:

```
PUT "*" ^ #words IN stars
PUT " " ^ #words IN blank
WRITE "***", stars, "***" /
WRITE "*" , blank, " " /
WRITE "*" , words, " " /
WRITE "*" , blank, " " /
WRITE "***", stars, "***"
```

#### FRAME "BEWARE!"

```
☐ *****
☐ *      *
☐ * BEWARE! *
☐ *      *
☐ *****
```

#### FRAME ""

```
☐ ****
☐ *  *
☐ *  *
☐ *  *
☐ ****
```

### 3. Give a new definition of the WISH command using the FRAME command.

Two further operators on texts are | (pronounced *of length*) for getting the first part of a text, and @ (pronounced *at*) for obtaining the last part:

Any part of a text may be obtained by combining these two operators in one expression:

The operators | and @ must be used with whole numbers that are neither too small nor too large; these five commands all lead to error messages:

The smallest and largest numbers allowed are shown here. (Both the first and the last command have the empty text as output.)

```
WRITE "nowhere"|3
```

```
☐ now
```

```
WRITE "nowhere"@3
```

```
☐ where
```

```
WRITE "nowhere"@4|2
```

```
☐ he
```

```
WRITE "nowhere"|6@4
```

```
☐ her
```

```
X WRITE "nowhere"|2.5
```

```
X WRITE "nowhere"|14
```

```
X WRITE "nowhere"@14
```

```
X WRITE "nowhere"|-2
```

```
X WRITE "nowhere"@-2
```

```
WRITE "nowhere"|0
```

```
☐
```

```
WRITE "nowhere"|7
```

```
☐ nowhere
```

```
WRITE "nowhere"@1
```

```
☐ nowhere
```

```
WRITE "nowhere"@8
```

```
☐
```

4. The target `t` contains a non-empty text. Give commands such that:

- only the first character is left in `t`;
- all but the first character is left;
- only the last character is left;
- all but the last character is left;
- none of the characters is left.

The operators `<`, `<=`, `=`, `>=`, `>` and `<>`, which were introduced for numbers, have a meaning for texts too. One text is smaller than another text if it would come earlier in a dictionary, so, for instance, `"no"` `<` `"now"` `<` `"s"`. The empty text `""` is smaller than any other text.

## Rules

- To decide which of two different texts is the smaller one, the computer looks if one text is shorter and exactly matches the head of the other text. If so, then the shorter text is the smaller one. (For instance, `"art"` `<` `"artificial"`.) Otherwise, the computer compares the first characters of both texts, then the second characters, and so on. The first difference decides which text is smaller. (So, for instance, `"artificial"` `<` `"artist"` because `"f"` comes before `"s"`.) Because not all characters in texts are letters, we need to know the order of the other characters, too.

The order of all characters a text may contain is given in this list. It starts with the space character, which you might overlook. It is not particularly useful to memorize the list, but note that the digits occur in their usual order, as do both capital letters and lower case letters.

```

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

```

- The usual space between two items written on the screen by `WRITE` commands does not appear between two text items.

5. What is the alphabetical order of these 3- and 4-character texts:

```
it1 it2 it12 ITS its "#!< ~@#?
```

6. Define a command `STRIP` which removes all occurrences of the letter `s` at the beginning and end of a text in a target. The text is not empty. `STRIP` should have this effect:

```

PUT "selfishness" IN word
STRIP word
WRITE word
☐ elfishne

```

7. Here is a program to show all 'endings' of the word `sago`. Turn this program into a definition of a command `SHOW'TAILS` that can do the same for any text.

```

PUT "sago" IN w
WHILE w > "":
  WRITE w /
  PUT w@2 IN w
☐ sago
☐ ago
☐ go
☐ o

```

8. Define a similar command `SHOW'HEADS`.

9. Define a command `SHIFT` which does this for any text of at least two characters:

```

PUT "smother" IN w
SHIFT w
WRITE w
☐ mothers

```

10. Define a command that does the same repeatedly and shows the result of every step:

SHOW'SHIFTS "tea"

☐ tea

☐ eat

☐ ate

SHOW'SHIFTS "murmur"

☐ murmur

☐ urmurm

☐ rmurmu

11. Palindromes are words which read the same backwards as forwards, such as **peep** and **rotator**. Define a command to help check if a text is a palindrome. The command should show what, if anything, is left after identical letters at the beginning and the end have been thrown away in pairs. It should have this effect:

CHECK'PALINDROME "remember"

☐ The rest is: memb

CHECK'PALINDROME "rotator"

☐ The rest is: a

CHECK'PALINDROME "peep"

☐ The rest is:

## 7. Types

So far, we have used two types of values in our programs: numbers and texts. Later, some more types will turn up, but this is a good time to look at some of the consequences of having more than one type of values. For one thing, each type has its own specific operators which cannot be used for other types. For instance, `+` only works on two numbers, `^` on two texts, and `^^` on one text and one number.

So, if you gave this command, the computer would not be able to execute it, but would immediately signal the error that `+` was used incorrectly.

```
X WRITE "Hi"+"gh"
```

If we make this kind of error inside a `HOW'TO` definition, it would be nice to be warned immediately when we make it, instead of having to wait until the computer *executes* this new command.

```
HOW'TO EXTEND number WITH digit:
X PUT (10*number)^digit IN number
```

Luckily, the computer does indeed warn us immediately. As soon as we have finished typing the erroneous line `PUT (10*number)^digit IN number`, the computer warns us that we have tried to use `^` on numbers. This has the advantage that we can correct the error immediately, and not have to come back to this problem at some later time, when we have partly forgotten what the line was all about.

Another type of mistake the computer spots early, is the use of a target for a different type than before.

In this way, accidentally using the wrong target may be corrected immediately, as in this definition of a command that counts with how many equal letters a certain text begins:

```
HOW'TO COUNT'EQUALS text:
PUT text|1 IN f
PUT 0 IN c
WHILE rest|1 = f:
X PUT f+1, rest@2 IN f, rest
WRITE c, "letters", f
```

As soon as we have finished typing the erroneous line `PUT f+1, rest@2 IN f, rest`, the computer will tell us that we are now wrongly using the target `f` for a numerical value, having used it until the previous line for a text. Of course, the definition should have been:

```
HOW'TO COUNT'EQUALS text:
PUT text|1 IN f
PUT 0 IN c
WHILE rest|1 = f:
    PUT c+1, rest@2 IN c, rest
WRITE c, "letters", f
```

Now and then we might want to give a target a value of a different type on purpose, hoping to see results like this:

```
PUT 56789 IN a
WRITE a, a*a, a*a*a
□ 56789 3224990521 183143986697069
X PUT "tar" IN a
WRITE a, a^a, a^a^a
□ tar tartar tartartar
```

Here too, it is not allowed to start using the target `a` for storing a number, and to give it a text target at some later time. This means that we must choose a different tag to put `"tar"` in.

So, the rule that a target may only be given values of the same type as the first value we give it, applies not only to the internal targets of a command definition, but also to external targets. This would mean that once we have given the external target `a` a numerical value in 1984, we cannot use it for a text, even in 1994! In order to cancel this eternal commitment, and to avoid a continuous rise in the number of targets used, the special command `DELETE` may be used to delete targets, both their values and their names.

If the target memory looks like this:

```
a = 46  w = "peep"  p = "yes"  nr = 3
```

then the targets `w` and `nr` may be discarded with this command:

```
DELETE w, nr
```

As a result, the target memory will be:

```
a = 46  p = "yes"
```

It is good practice to `DELETE` those external targets that serve no purpose any more, because it makes it easier to remember the roles of the remaining targets that still do have a purpose.



The rule of using a target only for one type of value does not forbid us from using a command such as this sometimes for swapping the values of two numeric targets and sometimes for swapping texts:

```
HOW'TO SWAP a AND b:
  PUT a, b IN b, a
  WRITE a /
  WRITE b /
```

```
PUT 1, 2, "y", "n" IN p, q, pos, neg
SWAP p AND q
□ 2
□ 1
SWAP pos AND neg
□ n
□ y
```

Of course, we cannot use the SWAP command to interchange the values of a numeric target and a text target:

```
X SWAP pos AND q
```

The problem is that this would lead to this command which clearly violates the type rule:

```
X PUT pos, q IN q, pos
```

## Rules

- An existing external target may only be given a new value if that value has the same type as the previous value.
- Every time a certain user-defined command is used, all values given to one internal target should be of the same type. The computer checks this at the time the definition is made.

☞ 1. For all slots of the following command definitions, determine if it should be filled with a numeric or with a textual parameter, or if either is allowed. Also, determine if the parameter should be a target or may be any expression of the correct type.

```
HOW'TO COUNT l IN t GIVING c:
  PUT 0 IN c
  WHILE t > "" AND t|1 = l:
    PUT t@2 IN t
    PUT c+1 IN c
```

```
HOW'TO ENCLOSE word IN sign:
  WRITE sign, word, sign
```

```
HOW'TO YAF p SO q AH r DU s:
  PUT r+1, s, p, q IN p, q, r, s
```

## 8. Input: the READ command

Earlier we saw this definition of a command that helps us adding a list of numbers, writing a subtotal after each new number.

```
HOW' TO A n:
  SHARE total
  PUT total+n IN total
  WRITE total
```

This command may be used in this way:

```
PUT 0 IN total
A 20.80
□ 20.8
A 7.75
□ 28.55
A 9/2
□ 33.05
```

Although the SHARE command in the definition of the A command frees us from the burden of specifying the external target `total` every time we use A, and although the name A is as short as command names come, we might want to type even less: just the numbers that have to be added.

This is possible if we use READ commands in the definition:

```
HOW' TO ADD' IN t:
  PUT 0 IN t
  READ nr EG 0
  WHILE nr <> 0:
    PUT t+nr IN t
    WRITE t /
  READ nr EG 0
```

When the computer now executes a command like `ADD' IN total`, it first gives `total` the value 0. Then, reaching the command `READ nr EG 0`, it prompts you by writing a ? sign on the next line of the screen, and waits until you have typed a number (or some other numeric expression). As soon as you have done so, the computer gives the target `nr` this number as a value, and goes on with the next command. The `EG 0` part of this READ command indicates that the computer has to check that you type an expression of numeric type, not a text.

So, if we give this command:

```
ADD' IN total
```

then this is what happens on the screen. The only things we have to type are the numbers to be added. Because the condition in the WHILE line of the definition is `nr <> 0`, the process stops as soon as we type a 0 as *input*, i.e. as answer to the ? sign. I have chosen the number 0 as a signal that I want the process to stop, because it would be rather silly to want to add 0 to a target.

```
□ ?
250
□ 250
□ ?
5*5
□ 275
□ ?
-15
□ 260
□ ?
0
```

☞ 1. Every time we use the `ADD' IN` command, it starts afresh adding from 0. In some cases that is not what we want. We might want to keep some target *shopping' costs* up to date by adding once every day the shopping expenses made that day. In the meantime, we want to be able to use the computer for other things. What change do you propose in the definition of `ADD' IN` to allow this? How should the new command be used?

Here is another example of a definition with a READ command. This CHECK/PALINDROMES command keeps asking the user for words to help checking if they are palindromes. Of course, we use the CHECK/PALINDROME command of exercise 6.11 in it. Because this case calls for input of texts, not numbers, we use EG "" instead of EG 0 in the READ commands.

Here we see an example of an application of the CHECK/PALINDROMES command. In this case, we use the text "stop" to indicate that we want to stop (stop is not a palindrome anyway). Note that the input is not restricted to literal texts, but that expressions such as "pop"^^3 are all right too.

In many cases it is a nuisance to have to type the two ' signs before and after every input text. In such cases we can use the keyword RAW instead of EG "" in the definition. This means that the input line, taken as a text, should become the value of the target of the READ command.

When we give input for a READ RAW command, we cannot use an expression such as "pop"^^3, because it would be interpreted as the literal text "pop"^^3:

HOW/TO CHECK/PALINDROMES:

```
READ w EG ""
WHILE w <> "stop":
  CHECK/PALINDROME w
  READ w EG ""
```

CHECK/PALINDROMES

```
□ ?
"remember"
□ The rest is: memb
□ ?
"peep"
□ The rest is:
□ ?
"pop"^^3
□ The rest is: o
□ ?
"stop"
```

HOW/TO CHECK/PALINDROMES:

```
READ w RAW
WHILE w <> "stop":
  CHECK/PALINDROME w
  READ w RAW
```

CHECK/PALINDROMES

```
□ ?
remember
□ The rest is: memb
□ ?
"pop"^^3
□ The rest is: "pop"^^3
□ ?
"stop"
□ The rest is: stop
□ ?
stop
```

Here follows a program which uses most of the features of B treated so far. It is the definition of a command which may be used to play a word guessing game. The player is given turns to guess a secret word. At each turn, the computer shows the beginning of the word as far as the guess was right plus one extra letter. The program may be more complex than you would be able to write at this stage. In fact, the program could be written in a clearer way with the help of constructions that you will learn in Part 2. It is not very interesting to play this game by yourself, since you have to put the secret word in yourself, but someone else may take the challenge of guessing your word in as few turns as possible.

**HOW TO WORD GUESS:**

```

PUT "conundrum" IN it
WRITE "Try to guess the secret word." /
PUT 1, 1 IN turn, known
WRITE "It starts with: ", it|known /
READ try RAW
WHILE #try < #it OR try|#it <> it:
  PUT turn+1, known+1 IN turn, known
  WHILE #try >= known AND #it >= known AND try|known = it|known:
    PUT known+1 IN known
    WRITE "It starts with: ", it|known /
    READ try RAW
  WRITE "The word was: ", it /
  WRITE "It took", turn, "turns to guess", #it-1, "letters" /

```

In the definition above, the target known represents the number of letters to be shown to the player, turn contains the number of guesses the human player has made and try is the latest guess. A typical session with the WORD GUESS command looks like this:

**WORD GUESS**

```

☐ Try to guess the secret word.
☐ It starts with: c
☐ ?
clarity
☐ It starts with: co
☐ ?
consonant
☐ It starts with: conu
☐ ?
conundrums
☐ The word was: conundrum
☐ It took 3 turns to guess 8 letters

```

**Rules**

- Each time you give the wrong type of input, or badly formed input, to a READ command, it will ask you for the correct input.
- When a command containing a READ command in its definition asks for input, you can stop the process by pressing the stop key.

**Advice**

- In the program examples of this chapter, much care has been given to the proper termination of programs. Although the stop key may be used in cases of emergency, it is better if commands terminate properly, because of the possibility of using a command in the definition of another command. It would be a nuisance if one of the commands used in a command definition did not terminate, since the rest of the command definition would never be executed.

2. Define a command to ask the user for the three dimensions of a box, and which then shows some statistics:

**BOX**

```

☐ Give 3 dimensions in cm.
☐ ?
2
☐ ?
5
☐ ?
10
☐ Area of the faces: 10 20 50 cm2
☐ Total area: 160 cm2
☐ Volume: 100 cm3
☐ Area per cm3 volume: 1.6 cm2

```

☞ 3a. Change the command definition of the previous exercise in such a way that after the computer has treated one box, it asks for the dimensions of the next box until the user specifies the first dimension of a box as 0, which is taken as an indication to stop.

☞ 3b. Make another version of the same program, now depending on the use of the stop key for termination.

☞ 4. Define a command NUMBER'GUESS which lets players guess the next number in a regular sequence until they are right. Let it use the kind of sequence we used in exercise 4.8. Here is a demonstration of how the input and the output might look:

NUMBER'GUESS

☐ A number sequence starts with: 2 1

☐ Guess the next number.

☐ ?

0

☐ No, 4

☐ ?

10

☐ No, 9

☐ ?

4\*4

☐ No, 22

☐ ?

53

☐ Right.

## 9. Conditional execution: the IF and SELECT commands

A very crude program to produce the plural of an English noun could be (remember WRITE does not put a space between texts):

The DOUBLE command gives correct results in many cases:

but wrong results if, for instance, the noun ends in s:

To make the definition work for such words too, it would be nice if we could WRITE an e only if the noun ends in s, and then proceed writing an s in every case. The IF command allows us to do this:

Now, most words ending in s will be treated correctly too:

```
HOW'TO DOUBLE noun:
    WRITE noun, "s"
```

```
DOUBLE "trap"
□ traps
DOUBLE "size"
□ sizes
```

```
DOUBLE "kiss"
□ kisss
```

```
HOW'TO DOUBLE noun:
    WRITE noun
    IF noun@#noun = "s":
        WRITE "e"
    WRITE "s"
```

```
DOUBLE "lens"
□ lenses
DOUBLE "gas"
□ gases
```

An IF command looks and works like a WHILE command, but it executes its indented part at most once. An IF command is called for in cases where we want one or more commands executed only if a certain condition holds.

In some programs, we want the computer to choose among two or more possibilities. If we want to make the DOUBLE program work for nouns ending in y too, we want the computer to select one of three treatments, depending on the last letter of the noun.

This can be done by using a SELECT command:

```
HOW'TO DOUBLE noun:
    SELECT:
        noun@#noun = "s":
            WRITE noun, "es"
        noun@#noun = "y":
            WRITE noun|(#noun-1), "ies"
    ELSE:
        WRITE noun, "s"
```

A SELECT command typically contains two or more *alternatives*, each of them indented. An alternative starts with a condition and a :, followed by some (more deeply) indented commands. When the computer reaches a SELECT command, it looks at the conditions, starting at the first one. As soon as it finds a condition that is true, it executes the commands in the indented part of that alternative. All following alternatives are skipped. In the last alternative, we may use ELSE instead of the condition, indicating that this alternative should be taken if none of the previous conditions is true. If we do not use ELSE, at least one of the conditions must be true. Otherwise, the computer reports an error.

Here is a version of the **DOUBLE** command that recognizes some other special cases too:

HOW'TO **DOUBLE** noun:

```

PUT noun@#noun IN z
PUT noun@(#noun-1) IN zz
SELECT:
  zz = "us":
    WRITE noun|(#noun-2), "i"
  z = "s" OR z = "z" OR z = "x":
    WRITE noun, "es"
  zz = "ch" OR zz = "sh":
    WRITE noun, "es"
  z = "y":
    WRITE noun|(#noun-1), "ies"
ELSE:
  WRITE noun, "s"

```

☞ 1. Would it make a difference if the third and fourth alternatives were interchanged (condition *and* command, of course)? How about interchanging the first two alternatives?

☞ 2. Change the **DOUBLE** program so that it will also:

- a. add **es** to words ending in **o**;
- b. use **ves** rather than **fes** for words ending in **lf**;
- c. use **men** for words ending in **man**, but only for those starting with a lower-case letter.

You may assume that the nouns are at least three letters long. Some results would look like this:

```

DOUBLE "hero"
☐ heroes
DOUBLE "calf"
☐ calves
DOUBLE "woman"
☐ women
DOUBLE "Roman"
☐ Romans

```

As we have seen, an **IF** command should be used if one group of commands may or may not have to be executed. A **SELECT** command should be used if precisely one of several groups of commands must be executed. For some problems, the solution may be formulated either way, as shown in the following example.

If we want a command that writes the values of its two parameters in the order of their magnitude, we may use **IF** to swap them if they are initially in the wrong order:

HOW'TO **WRITE'UP** a AND b:

```

PUT a, b IN p, q
IF p > q:
  PUT p, q IN q, p
WRITE p, q /

```

Using **SELECT** for the same problem results in this neat solution:

HOW'TO **WRITE'UP** a AND b:

```

SELECT:
  a < b:
    WRITE a, b /
  a >= b:
    WRITE b, a /

```

or, using **ELSE** in the last alternative:

HOW'TO **WRITE'UP** a AND b:

```

SELECT:
  a < b:
    WRITE a, b /
  ELSE:
    WRITE b, a /

```

☞ 3. What will happen with these five applications of **WRITE'UP**?

```

PUT 5, 3 IN s, t
PUT "way", "side" IN x, y
WRITE'UP s AND t
WRITE'UP 3 AND 5
WRITE'UP x AND t
WRITE'UP x AND y
WRITE'UP "side" AND "way"

```

4. Change the CHECK/PALINDROME command of exercise 6.11 in such a way that it will answer in this way:

5. Write the following variation of the WORD/GUESS command of chapter 8. The human player is invited to guess a secret number. At each wrong guess, the computer should tell if it is too low or too high. The 'conversation' should go like this:

```
CHECK/PALINDROME "remember"
☐ No, remember is not a palindrome.
CHECK/PALINDROME "rotator"
☐ Yes, rotator is a palindrome.
```

```
NUMBER/GUESS
☐ Try to guess the secret number.
☐ ?
100
☐ lower
☐ ?
30
☐ higher
☐ ?
70
☐ lower
☐ ?
50
☐ lower
☐ ?
40
☐ higher
☐ ?
44
☐ Right!
```

## Rules

- The SELECT command is used for selecting one out of several alternatives. The IF command is used if there is only one alternative, which may have to be executed or not.
- At least one of the alternatives of a SELECT command must have a condition which is true. If an ELSE alternative is used, it must be the last one.
- Sometimes it is necessary to add an 'empty' alternative to a SELECT command, in order to stick to the rule that at least one of the alternatives must be true:

```
HOW/TO COUNT nr:
  SHARE pos, neg, npos, nneg
  SELECT:
    nr < 0:
      PUT neg+nr IN neg
      PUT nneg+1 IN nneg
    nr = 0:
      \zero need not be counted
    nr > 0:
      PUT pos+nr IN pos
      PUT npos+1 IN npos
```

The second alternative has no commands in this case. Instead I put a *comment* there, which explains the situation to the human reader.

- Comments can be recognized by the \ sign. Everything following a \ sign on a line is ignored by the computer.





## 10. Lists

Until now we have only seen programs that used rather few things from the computer's memory. Using the means we have seen so far, a program can only work with many items of information if there are many targets in the program, each with its own tag. If we wanted the computer to remember every palindrome we have given it, we would have to use one target for each palindrome. To get those palindromes written on the screen, we would need a program using that (growing) number of targets. Instead of using so many targets, we can use one target of the new type *list*. A list is a collection of items, e.g. texts.

Here is a command to store a list of two words in a target:

```
PUT {"pop"; "peep"} IN pals
```

As a result of this command, the target `pals` contains *one* value of the new type list. The list has two *items* (the texts `"pop"` and `"peep"`), occurring in alphabetical order:

```
pals = {"peep"; "pop"}
```

What can we do with this new type of values and targets? First of all, we can do anything we can with the other types we have seen, such as `PUT` and `WRITE`.

Here the value of the list target `pals` is copied into the target `p`, and the value of `p` is written on the screen:

```
PUT pals IN p
WRITE p
□ {"peep"; "pop"}
```

In command definitions, those slots that may be filled with any type of parameter, may also be filled with list targets.

```
PUT {"a"; "I"; "eye"; "you"} IN vw
SWAP p AND vw
WRITE p
□ {"I"; "a"; "eye"; "you"}
```

For lists of consecutive whole numbers or characters, we may use a shorthand notation with `..` (pronounced *up to*):

```
PUT {"n".."q"} IN lrs
WRITE lrs
□ {"n"; "o"; "p"; "q"}
PUT {2*2..3*3} IN nrs
WRITE nrs
□ {4; 5; 6; 7; 8; 9}
```

Just as there are special operators for numbers and texts, there are special operators *and* commands for lists.

If we want to add a new word to the list `pals`, we can use the special `INSERT` command:

```
INSERT "I" IN pals
```

If we now `WRITE` the new value of `pals`, we see that the order of the items is alphabetical:

```
WRITE pals
□ {"I"; "peep"; "pop"}
```

The list remains sorted as we keep `INSERTing` new words (remember that capital letters come alphabetically before lower case letters).

```
INSERT "eye" IN pals
WRITE pals
□ {"I"; "eye"; "peep"; "pop"}
```

We see the same thing happen with lists of numbers:

```
WRITE {8; 3*5; 4; 1; 2*5}
□ {1; 4; 8; 10; 15}
```

It is all right to insert an item that is already present in a list:

```
INSERT "eye" IN pals
WRITE pals
□ {"I"; "eye"; "eye"; "peep"; "pop"}
```

It is also possible to remove an item from a list target. If the item occurs more than once in the list, only one occurrence is removed:

```
REMOVE "eye" FROM pals
WRITE pals
□ {"I"; "eye"; "peep"; "pop"}
```

It is not correct to try and REMOVE an item that does not occur in the list. So, this would lead to an error message:

There is a way to find out if a certain item does occur in a list:

The in operator is used for building conditions, just like < and =. The operator not'in gives the opposite result, as shown in this command for inserting an item only if it is not yet there:

☞ 1. Define a command REMOVE'ALL which removes all occurrences of a certain item in a list. It should be possible to use it in the following way. Note the output {} to indicate an empty list.

☞ 2. Find a shorter way to write the condition in this program:

The operators in and not'in may also be used on texts to see if a certain character occurs in a text:

Here follow some more operators that may be used both on lists and on texts.

The # operator, which we have used to find the number of characters in a text, may also be used to find the number of items in a list:

There is also a version of # with two operands, which finds out how many times a certain item occurs in a list:

Used with a text as right operand, the # operator needs a single character text as left operand:

The smallest item of a list can be found with the operator min:

Used for a text, min gives the alphabetically smallest character:

It is easy to guess what max does:

```
X REMOVE "rotator" FROM pals
```

```
IF "rotator" in pals:
    REMOVE "rotator" FROM pals
```

```
HOW TO INSERT ONE item IN list:
    IF item not'in list:
        INSERT item IN list
```

```
PUT {5; 5; 8} IN nrs
REMOVE'ALL 5 FROM nrs
WRITE nrs
□ {8}
REMOVE'ALL 4 FROM nrs
WRITE nrs
□ {8}
REMOVE'ALL 8 FROM nrs
WRITE nrs
□ {}
```

```
IF z = "s" OR z = "z" OR z = "x":
    WRITE noun, "es"
```

```
PUT "ayouie" IN vowels
IF letter not'in vowels:
    WRITE letter, " is a consonant"
```

```
WRITE #{8; 4; 7}
□ 3
PUT 7 IN n
WRITE #{2..n}
□ 6
```

```
PUT {"s"; "z"; "x"} IN sib
WRITE "z"#sib
□ 1
WRITE "h"#sib
□ 0
WRITE 2#{1; 2; 2; 2; 3; 4; 4}
□ 3
```

```
PUT "mississippi" IN river
WRITE "i"#river
□ 4
```

```
WRITE min {1..7}
□ 1
WRITE min {"u"; "ewe"; "you"}
□ ewe
```

```
WRITE min "typewriter"
□ e
```

```
PUT {13; 5; 8} IN nrs
WRITE min nrs, max nrs
□ 5 13
```

To get the fourth character of a text, or the third item of a list:

WRITE 4 th'of river

☐ s

WRITE 3 th'of sib

☐ z

☞ 3. Write this in a different way:

a. WRITE 1 th'of pals

b. WRITE 1 th'of river

c. WRITE #pals th'of pals

☞ 4. What will be the results of these commands?

a. WRITE 2 th'of {"four"; "five"}

b. WRITE 2 th'of "four five"

c. WRITE 2 th'of 45

d. WRITE "two" th'of {"four"; "five"}

## Rules

- The operators < etc. work on every type, including lists. To decide which of two lists is the smaller one, the computer checks which one has the smaller first item. If these are equal, the next item will determine which is smaller, and so on.

If all items of one list are equal to the first so many items of a different list, than the first list is the smaller one:

{1; 2; 9} < {1; 3; 4} < {2}

- All items of a list must have the same type. That means that this is wrong:

PUT {2..5} IN nrs  
X INSERT "one" IN nrs

- The type we choose for the items of a certain list may be *any* type. That includes other lists:

PUT {} IN s3  
INSERT {1; 2} IN s3  
INSERT {1; 1; 1} IN s3  
INSERT {3} IN s3  
WRITE s3  
☐ {{1; 1; 1}; {1; 2}; {3}}  
WRITE 2 th'of s3  
☐ {1; 2}

In fact, there are many list types. For instance, {"s"; "z"; "x"} has the same type as {"yes"; "no"} (list of texts), but {9; 11} has a different type (list of numbers). This means that these different types of lists cannot occur as items of one list.

So, this is wrong:

X PUT {{0; 0}; {"one"; "one"}} IN nn

- The operators in, not'in, #, min, max and th'of may be used on texts as well as on lists. They do the same with the *characters* of a text as with the *items* of a list. The commands INSERT and REMOVE only work on lists.

- It is an error to use the operators min, max and th'of on the empty list {}.

- Likewise, th'of should only be used with whole numbers in the correct range, so these applications of th'of are wrong:

X PUT 2.5 th'of {2; 5; 7; 7; 9} IN s  
X WRITE 3 th'of {"yes"; "no"}

☞ 5. What will happen with the following commands?

a. WRITE min {"3"; "22"; "111"}

b. WRITE min {{}}

c. WRITE min min {{5; 2}; {5; 2; 1}}

d. WRITE #{}

☞ 6. The targets a, b and c contain different numbers. Give a command that will write the middle one (that is, neither the smallest, nor the greatest).

☞ 7. Define a command that lets the user type texts as input until a text is repeated. The command should then write how many different words were given as input.

SPOT'DOUBLE

☐ ?

yes

☐ ?

yup

☐ ?

affirmative

☐ ?

sure

☐ ?

certainly

☐ ?

yup

☐ There were 5 different texts.

☞ 8. If `l` contains a lower case letter, how would you describe the value of `u` after the execution of this command:

PUT (#{"a"..l}) th'of {"A"..Z"} IN u

## 11. Another kind of repetition: the FOR command

Working with lists, we often want to treat all items of a list in the same way, such as write them on separate lines of the screen.

We may use the WHILE command to program such a repetition:

```
HOW'TO PRINT'LIST l:
  PUT l IN list
  WHILE list > {}:
    PUT min list IN item
    REMOVE item FROM list
    WRITE item /
```

Because it happens so often that we want to visit each item of a list in turn, there is a special repetition command that may be used for lists:

```
PUT {"eye"; "I"; "u"; "ewe"} IN pals
FOR word IN pals:
  WRITE word, " is a palindrome." /
☐ I is a palindrome.
☐ ewe is a palindrome.
☐ eye is a palindrome.
☐ u is a palindrome.
```

It is easy to tell in advance how many repetitions will take place in this FOR command: exactly as many as there are items in the list pals. The first time the indented part is executed, the target word contains the first (=smallest) item of the list; the second time, word contains the second item, and so on.


The following command definition uses a FOR command to INSERT all items of one list IN another list:

```
HOW'TO MERGE a INTO b:
  FOR item IN a:
    INSERT item IN b
```

This MERGE command may be used this way:


```
PUT {-3..3} IN nrs
MERGE {5; 2} INTO nrs
WRITE nrs
☐ {-3; -2; -1; 0; 1; 2; 2; 3; 5}
```


 1. Rewrite PRINT'LIST above so that it uses a FOR command.

 2. The targets a, b, c and d each contain a numeric value. Use the property of lists that their items are stored in ascending order to have the values of these four targets written on the screen in ascending order.

Here is a program that determines the length of the longest text in the list pals. The crucial target longest'seen contains the length of the longest text word has contained so far, excluding the present value of word.

```
PUT 0 IN longest'seen
FOR word IN pals:
  IF #word > longest'seen:
    PUT #word IN longest'seen
  WRITE "Longest was:", longest'seen
☐ Longest was: 3
```

 3. Write a program that calculates and writes the total length of all texts that are items of the list pals. Formulate the roles of any crucial targets you may need.

 4. Write a command FAHRENHEIT that will show how many degrees Celsius correspond to some temperature in degrees Fahrenheit given as a parameter. The formula can be found in an example in chapter 1. Let the output look like this:

```
FAHRENHEIT 451
☐ 451 F = 232.78 C
```

5. Use the FAHRENHEIT command to define a command that will do this:

```
LIST'FAHRENHEIT {40; 45}
□ 40 F = 4.44 C
□ 45 F = 7.22 C
LIST'FAHRENHEIT {40..45}
□ 40 F = 4.44 C
□ 41 F = 5.00 C
□ 42 F = 5.56 C
□ 43 F = 6.11 C
□ 44 F = 6.67 C
□ 45 F = 7.22 C
```

A FOR command may be used nicely whenever we know in advance how many repetitions we want, even when there is no list around. Here is a way to have the text **yes** written a growing number of times on four lines:

```
FOR lnr IN {1..4}:
  WRITE "yes"^^lnr /
□ yes
□ yesyes
□ yesyesyes
□ yesyesyesyes
```

The same scheme may be used in many situations where a target is needed containing a value which grows by 1 with each repetition.

In situations where we want a target with a value which decreases by 1 with each repetition, we may use the method shown here:

```
FOR diff IN {44..51}:
  WRITE 44+51-diff
□ 51 50 49 48 47 46 45 44
```

6. Define a command that will show an arrow-like shape formed out of \* characters. In this example, the longest line consists of 4 times 3 \* characters:

```
ARROW 4
□ ***
□ *****
□ *********
□ *********
□ *****
□ *****
□ ***
```

## Rules

• Starting to execute a FOR command, the computer calculates the list occurring in the FOR line, and stores the result in a temporary piece of extra memory, which it will throw away after having finished the whole FOR command. During the execution of the FOR command, the computer uses the copy to determine the sequence of values it has to give to the target in the FOR line. This target itself is also stored in the temporary piece of memory.

So, this program may be used to add an extra ! at the end of those words in the list **words** that already end in !:

```
PUT {"ho!"; "oh?"; "viva!!"} IN words
FOR word IN words:
  IF word@#word = "!":
    REMOVE word FROM words
    INSERT word^"! " IN words
WRITE words
□ {"ho!!"; "oh?"; "viva!!!"}

```

• A FOR command may also be used for going through the characters of a text. The characters are visited in the order they occur in the text, *not* in alphabetical order.

```
FOR char IN "yes":
  WRITE char /
□ y
□ e
□ s
```

- If the list or text in a FOR line is empty, the indented part is not executed at all. So, these commands give no output:

```

PUT {} IN texts
FOR c IN min texts:
    WRITE "Hello!" /
REMOVE max texts FROM texts
FOR text IN texts:
    WRITE "Hurrah!" /

```

### Advice

- Both FOR and WHILE commands are repetition commands. A FOR command should be chosen in cases where the number of repetitions is known before; otherwise a WHILE command is in order. A FOR is preferable when you have the choice, because you don't have to worry about its termination.
- Just as with WHILE commands, it is important to keep in mind the roles of the crucial targets of FOR commands. When programming either kind of repetition, one should not imagine the first time the repetition is being executed, but some time half way through the process.

☞ 7. Define an INVERT command that writes backwards the text parameter given to it. Use a FOR command.

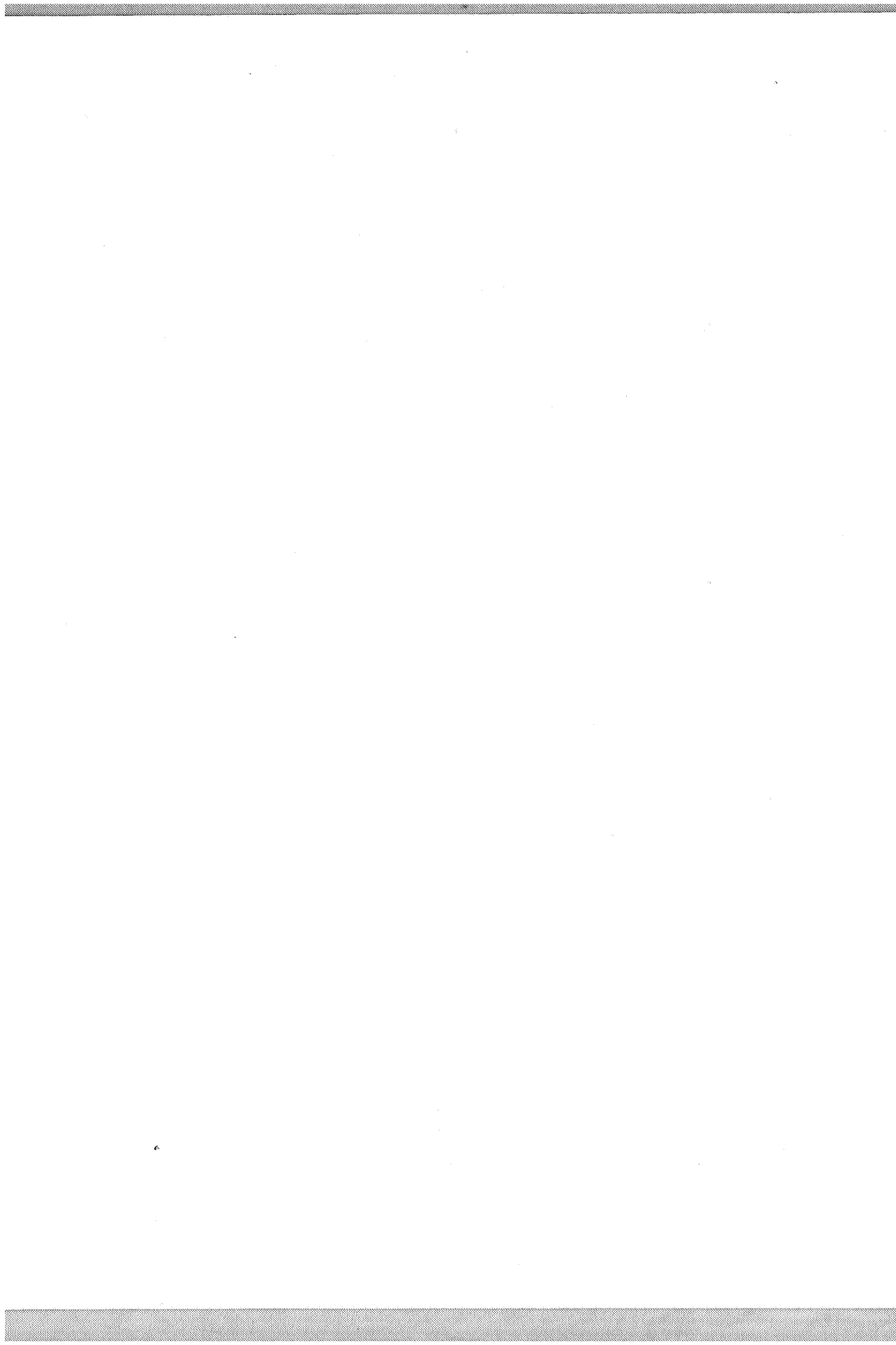
```

INVERT "evil"
□ live

```

☞ 8. Write a program that will write all multiples of 11 and 13 under 100. All numbers in the output should be ordered from small to large.





## 12. Tables

We have seen how lists are useful for storing a collection of items, such as English nouns. However, lists are not good enough if we want to have something like a telephone directory in the computer's memory, to permit us to look up what number is associated with a certain name.

Or we might want to build a dictionary storing a number of nouns coupled to their plurals, to ease finding the plural of such a noun. This can be done by using the fourth type in *B*: a table.

This creates a new target `pl` of type table. Its value is stored in memory in this way:

If we now give the command:

we get as result:

We can extend this very short table:

Writing the new value of `pl` on the screen gives:

The singulars "man" and "ox" are called the *keys* of the table; "men" and "oxen" are the *associates* of `pl`. A pair such as ["man"]: "men" is called an *entry* of the table.

We can add more entries to the table:

We can change one of the associates:

In fact, the table may be seen as a collection of targets, each entry acting as a target: `pl["base"]`, `pl["louse"]`, etc. These special targets may be used in any way that normal targets are used:

Of course, `pl` as a whole is also a normal target:

☞ 1. Give a command to add an entry in the table `plural` for the word `foot`, so that its correct plural will be stored. Afterwards, this should work:

The computer will report an error if we try to use an entry with a non-existing key:

If we want to find out if a certain key is present, we may use the special operator `keys`, which delivers a list of all the keys of a table:

☞ 2. Change the `DOUBLE` command, defined in exercise 9.2, so that it will consult the table `plural` to see if it provides the plural looked for. (Hint: use `SHARE`.)

```
PUT [{"ox"}: "oxen"} IN pl
```

```
pl = [{"ox"}: "oxen"]
```

```
WRITE pl["ox"]
```

```
□ oxen
```

```
PUT "men" IN pl["man"]
```

```
□ [{"man"}: "men"; {"ox"}: "oxen"]
```

```
PUT "bases" IN pl["base"]
PUT "leece" IN pl["louse"]
```

```
PUT "lice" IN pl["louse"]
WRITE pl["louse"]
□ lice
```

```
PUT pl["base"] IN pl["basis"]
DELETE pl["base"]
```

```
PUT pl IN plural
DELETE pl
WRITE plural["ox"]
□ oxen
```

```
WRITE plural["foot"]
□ feet
```

```
0 WRITE plural["pennill"]
```

```
WRITE keys plural
□ {"basis"; "louse"; "man"; "ox"}
IF "pennill" in keys plural:
    WRITE plural["pennill"]
```

Here is a program that uses a table to count how often individual characters occur in a list of texts. The {} in the second command is the empty table; it looks the same as an empty list. This command is needed, because PUTting something IN an entry of a non-existent table is forbidden.

```

PUT {"yes"; "no"; "maybe"} IN texts
PUT {} IN freq
FOR word IN texts:
  FOR c IN word:
    SELECT:
      c not'in keys freq:
        PUT 1 IN freq[c]
      ELSE:
        PUT freq[c]+1 IN freq[c]
FOR c IN keys freq:
  WRITE c, freq[c] /
□ a 1
□ b 1
□ e 2
□ m 1
□ n 1
□ o 1
□ s 1
□ y 2

```

If we know that the texts to be counted contain only lower case letters, for example, we could also use this version. With this approach, letters that do not occur in the texts will also be written, followed by 0.

```

PUT {"yes"; "no"; "maybe"} IN texts
PUT {} IN freq
FOR c IN {"a".."z"}:
  PUT 0 IN freq[c]
FOR word IN texts:
  FOR c IN word:
    PUT freq[c]+1 IN freq[c]
FOR c IN keys freq:
  WRITE c, freq[c] /

```

In both programs we just saw, the roles of the entries of freq can be described as: for all characters c seen so far, freq[c] contains the number of times c was seen. Put differently, in the first program freq contains the number of times each character was seen *if* it was seen at all. In the second program, freq contains the number of times each of the letters of the alphabet was seen, including 0 for letters not (yet) seen.

## Rules

- All keys of a table must have the same type. All associates of a table must be of the same type too, but not necessarily of the same type as the keys.

- Removing an entry from a table is done with DELETE:

```
DELETE freq["x"]
```

It cannot be done with REMOVE, because you can only REMOVE FROM a list target, which neither freq nor keys freq is. So, this is wrong:

```
0 REMOVE "x" FROM keys freq
```

Of course, the whole table can be discarded with DELETE, too:

```
DELETE freq
```

- Before something may be PUT in an entry of a table, the table must have been given some value, however small:

```

PUT {} IN freq
PUT [{"axis"}: "axes"] IN plur

```

- The entries of a table are ordered by their keys, before the table is stored in memory or written on the screen.

- The operators in, not'in, #, min, max and th'of may also be used on tables. The same thing they do with the *characters* of a text and the *items* of a list, they also do with the *associates* of a table.

So, if we have:

then the following conditions are all true:

```
t = {[ -3]: 9; [1]: 1; [2]: 4; [3]: 9}
```

```
4 in t
5 not in t
2 not in t
9#t = 2
min t = 1
3 th'of t = 4
```

• A FOR command may also be used on a table. In the same way as FOR visits the characters of a text or the items of a list from left to right, it visits the *associates* of a table. So, using the table *t* above, we get this output:

```
FOR n IN t:
  WRITE n
□ 9 1 4 9
```

### Advice

• Lists or tables may be useful when we want to work with collections of data. Typically, a list should be chosen if the order of the items does not matter (or if the items should be ordered in the usual way). A table is called for when each item represents a link from one thing (e.g., a person's name) to another (e.g., a telephone number).

• Also in cases where we want to keep a collection of items in a special order, we may use a table:

For instance, in this table three names are stored in an order different from the alphabetical one:

```
{[1]: "John"; [2]: "Al"; [5]: "Joe"}
```

• It is easy to find the associate belonging to a certain key, but the other way around is less direct. Therefore, it is important to decide what to choose as the keys of a table and what as the associates.

If, for instance, we will often need to find a telephone number given a name, we do well to take the names as keys and the telephone numbers as associates:

```
tel = {[ "Al": 4367; [ "Leo": 4141}
```

If, on the other hand, we rely on the normal telephone directory for this purpose, we may want to use the computer for the reverse: to find a name given a certain telephone number. In that case, we will choose the numbers for keys and the names for associates:

```
name = {[4141]: "Leo"; [4367]: "Al"}
```

• Some cases may seem to call for a list or a table, but in fact can be programmed more simply without. Take this program to write a table converting inches to centimetres:

In this case, we can directly write the results without storing them in a table target:

• Although the tag of a list target is typically a plural, or another word referring to a collection:

it is generally better to choose a singular as tag for a table:

```
PUT {} IN cms
FOR i IN {1..12}:
  PUT i*2.54 IN cms[i]
FOR i IN {1..12}:
  WRITE i, "inches=", cms[i], "cm" /

FOR i IN {1..12}:
  WRITE i, "inches=", i*2.54, "cm" /

FOR letter IN vowels:
  REMOVE letter FROM alphabet
INSERT word IN words

WRITE address["Mary"]
IF capital[country] = "Amsterdam":
  WRITE population[country]
```

3. Here is a clumsy way for someone in London to get the time it is at a certain moment in several other cities in the world. Suppose you have a table `diff` containing the time differences, such as `diff["New York"] = -5`. How can you use this table to define a simpler version of the `TIME` command?

4. Define a command `GET/SERIES` that reads a series of input words and puts them in a table in the order they were given. In this example, the input given after the fourth `?` is an empty text.

5. For filling a table, it would be nice to type less than all this:

Define a command `FILL/TABLE` that may be used in this way for filling such a table with text keys and text associates (an empty line is typed as answer to the last `?` mark):

6. Suppose we have a target `tel` giving telephone numbers as in the example above. Define a command `INVERT` which can be used to produce the opposite table `name` as in the same example. You may assume that all telephone numbers are different. So, this should work:

HOW/TO TIME hour:

```
WRITE "Amsterdam:", hour+1, "h" /
WRITE "Cairo:", hour+2, "h" /
WRITE "London:", hour, "h" /
WRITE "Madrid:", hour, "h" /
WRITE "New York:", hour-5, "h" /
WRITE "Sydney:", hour+10, "h" /
WRITE "Tokyo:", hour+9, "h" /
```

GET/SERIES friend

```
☐ ?
John
☐ ?
Al
☐ ?
Joe
☐ ?
```

FOR name IN friend:

```
WRITE name /
```

```
☐ John
☐ Al
☐ Joe
```

PUT {} IN capital

```
PUT "Paris" IN capital["France"]
PUT "La Paz" IN capital["Bolivia"]
PUT "Tokyo" IN capital["Japan"]
```

FILL/TABLE capital

```
☐ key:
☐ ?
France
☐ associate:
☐ ?
Paris
☐ key:
☐ ?
Bolivia
☐ associate:
☐ ?
La Paz
☐ key:
☐ ?
Japan
☐ associate:
☐ ?
Tokyo
☐ key:
☐ ?
```

WRITE tel

```
☐ [{"Al": 4367; "Ann": 4223}]
INVERT tel GIVING name
WRITE name
☐ [{"4223": "Ann"; 4367: "Al"}]
```

☞ 7. Now do the same if all names are different, but the same telephone number may be shared by several people. The new INVERT command should be able to deliver a table `names`, containing a *list of* names for each telephone number:

```
WRITE tel
□ {[ "Oz": 74; [ "Pa": 78]
PUT 78 IN tel[ "Ma"]
INVERT tel GIVING names
WRITE names
□ {[74]: { "Oz"; [78]: { "Ma"; "Pa" }}
WRITE names[78]
□ { "Ma"; "Pa" }
```



### 13. Random choice: the CHOOSE command

In the guessing games we have seen, it was awkward that the programmer had to specify the word or number to be guessed. It would be much nicer if the computer could pick some word or number. This can be done with the CHOOSE command.

After this command has been executed:

```
CHOOSE nr FROM {1..100}
```

the target `nr` will contain one of the positive whole numbers up to and including 100, but you don't know which of the 100 possibilities has been chosen.

The CHOOSE command may also be used on texts. In this way we can choose an arbitrary character from the text `mississippi`:

```
CHOOSE letter FROM "mississippi"
```

The CHOOSE command gives all possible choices equal chances, so the chance of the `i` being chosen is four times the chance that the `m` is chosen.

It will not come as a surprise that we may also CHOOSE an arbitrary associate in a table. If `tel` still has the value it had in exercise 12.7, this is a way to choose a random number from it:

```
CHOOSE no FROM tel
WRITE no
□ 78
```

Here is an example where the CHOOSE command is used to build a text consisting of characters which are chosen at random from a small collection:

```
HOW'TO CURSE n:
  FOR i IN {1..n}:
    CHOOSE sign FROM "@#$$%&*!?"
    WRITE sign
```

☞ 1. Use the CURSE command to have 5 curses written, the length of each curse being a random choice among 3, 5 and 8. The curses should be written on separate lines, as in this example:

```
□ ?*&
□ #?*&!
□ @*#
□ *!#@!!!@
□ $?*##
```

#### Rules

- The CHOOSE command works on the characters of a text, the items of a list or the associates of a table.
- The CHOOSE command gives an equal chance to each element to be chosen.
- The CHOOSE command does *not* remove the chosen element from the text, list or table.

☞ 2. Give a program to write either `yes` or `no`, the text `yes` having twice as much chance of being chosen as `no`.

☞ 3. Define a command PICK that does the same on lists as the CHOOSE command, but also takes out the chosen item:

```
PUT {1..6} IN s
PICK choice FROM s
WRITE choice
□ 3
WRITE s
□ {1; 2; 4; 5; 6}
```

☞ 4. Design a command that writes the items of a list in a random order. (Hint: use the PICK command.) This could be the output:

```
SCRAMBLE {"a".. "e"}
□ e
□ b
□ c
□ a
□ d
```

☞ 5. What should be changed in the definition of the NUMBER'GUESS command of exercise 9.5 in order to make it decide on a secret number itself?



☞ 6. Define a command **THROW** that imitates a throw with a specified number of dice. Note the use of an extra keyword at the end of these examples of the **THROW** command, which is not followed by a parameter. This is all right, if the command definition looks the same.

```
THROW 1 DICE
□ 4
THROW 3 DICE
□ 13
```

☞ 7. Test how randomly the **CHOOSE** command works by counting how often a dice shows a 1, how often a 2, etc. Count 120 rolls of the dice. (Hint: use a table.)

☞ 8. Suppose you have a table **capital** as made in exercise 12.5. Design a command that tests a user by asking what the capitals of the countries are. Let the program select the countries at random, but without repeating countries for which the right answer was given. If the target **capital** only contains **Bolivia**, **France** and **Japan** as keys, the conversation might go along these lines:

```
QUIZ capital
□ Japan
□ ?
Kyoto
□ No, Tokyo
□ France
□ ?
Paris
□ OK
□ Japan
□ ?
Tokyo
□ OK
□ Bolivia
□ ?
La Paz
□ OK
□ That was all.
```

## 14. Compounds

The last type of *B* is one that you have already used without knowing it: the *compound*.

Every time we wrote things like this we used compounds:

```
WRITE n, "is divisible by", d
PUT a, b IN c, d
```

Because compounds are a genuine type of the language, it is also possible to PUT one in a target:

```
PUT "John", 3 IN name1
```

The result of this command in memory looks like this:

```
name1 = ("John", 3)
```

In this example, "John" and 3 are called the *fields* of the compound.

Unlike the other types, there are no special operators or commands for compounds. Also, the operators such as *min* and *th'of*, which work on texts, lists and tables, may *not* be used on compounds. All we ever do with compounds that we would not do with other types is:

1. Packing several values into one compound:

```
PUT "John", #"John" IN name1
PUT -p, p IN b
PUT 1984, "December", 14 IN date
```

2. Unpacking one compound into several targets:

```
PUT name1 IN name, length
PUT b IN minus, plus
PUT date IN year, month, day
```

3. Rearranging several targets into equally many targets:

```
PUT a, b, c IN b, c, a
```

In fact, there are many different compound types, just as there are different list types (list of numbers, list of texts, etc.) and different table types (table with numeric keys and text associates, or the other way around, etc.).

The following compounds are all of different types:

```
(13, 15)
(10, 12, 17)
("a", 1)
(1, "a")
```

So, this is wrong because the types are mixed up:

```
PUT 1900, "January", 1 IN date1
PUT "December", 31, 1899 IN date2
X PUT date1 IN date2
```

There is no way to change the number of fields of a compound target. (The only way would be to DELETE it altogether. From then on the target may be used for any type.)

As for all other types, the operators *<* etc. may also be used on two compounds *of the same type*. To find out which of two compounds is the smaller one, the same procedure is followed as for texts and lists: it depends on the first different field. So these conditions are all true:

```
(0, 5) < (1, 0)
(4, "John") < (6, "Albert")
(1, 0, 0) < (1, 0, 1)
```

Although the items of a list are ordered from small to large, there is a trick involving compounds to bring about other orderings than the usual one.

If, for instance, we have a list of words which we want to be written not in alphabetical order, but in the order of their lengths, we may use this scheme:

```
PUT {} IN new'list
FOR word IN words:
  INSERT #word, word IN new'list
FOR pair IN new'list:
  PUT pair IN length, word
  WRITE word /
```

☞ 1. Use the same scheme to have a list of numbers written starting at the large end. (Hint: if  $a < b$ , then  $-a > -b$ .)

Another use we can make of compounds is for keys of a table. Take this two-dimensional layout showing the sales figures of three salesmen in four years:

	Al	Joe	John
81	65	68	88
82	67	74	96
83	61	74	90
84	65	71	72

Using compounds as keys, we could represent this table in this way:

```
PUT {} IN sales
PUT 65 IN sales[81, "Al"]
PUT 68 IN sales[81, "Joe"]
PUT 88 IN sales[81, "John"]
PUT 67 IN sales[82, "Al"]
etc.
```

To find the total sales in 1982, we can now write:

```
PUT 0 IN t82
FOR person IN {"Al"; "Joe"; "John"}:
  PUT t82+sales[82, person] IN t82
```

If, on the other hand, we want the total sales of Joe since 1981, we can do this:

```
PUT 0 IN t'joe
FOR y IN {81..84}:
  PUT t'joe+sales[y, "Joe"] IN t'joe
```

Finding the total of all sales since 1981 goes like this:

```
PUT 0 IN sum
FOR key IN keys sales:
  PUT sum+sales[key] IN sum
```

A different approach to representing the sales table shown above is to consider it as a table with tables as associates:

```
PUT {} IN s
PUT {[81]: 65; [82]: 67} IN s["Al"]
PUT {[81]: 68; [82]: 74} IN s["Joe"]
etc.
```

Calculating the total sum of  $s$  is slightly more complex now:

```
PUT 0 IN sum
FOR person IN keys s:
  FOR y IN {81..82}:
    PUT sum+s[person][y] IN sum
```

On the other hand, finding the total for one year or one person is easier:

```
PUT 0 IN t82
FOR person IN keys s:
  PUT t82+s[person][82] IN t82
PUT 0 IN t'joe
FOR y IN keys s["Joe"]:
  PUT t'joe+s["Joe"][y] IN t'joe
```

☞ 2. Find a short way to create a table  $t$  with 25 entries, the associates all being 0 and the keys being different compounds with two numeric fields. The fields should range from  $-2$  to  $2$ , so, for instance,  $t[1, -2]$  and  $t[0, 0]$  should be among the targets that get the value 0.

☞ 3. Write a program for finding out how often, e.g., a 3 is followed by, e.g., a 5, in a sequence of rolls with a dice. At the end  $\text{freq}[3, 5]$  (or  $\text{freq}[3][5]$ ) should contain the number of times a roll of 3 has been followed by a roll of 5, etc. So, if the rolls are: 5 3 6 4 3 6 4 4 3, then  $\text{freq}[3, 6] = 2$  (or  $\text{freq}[3][6] = 2$ ). Let the experiment continue until 120 pairs have been registered.

## Rules

- There are no compounds with only one field, because the same things can be done with simple targets anyway.



## Solutions to exercises

### 1. Arithmetic: the WRITE command

1.

Since the edges of a box come in parallel foursomes, this should do it:

WRITE 4\*(8+12.5+1.75)

☐ 89.00

2.

Both occurrences of 50 should be changed into 100:

WRITE "100 F =", (100-32)\*5/9, "C"

☐ 100 F = 37.77777777777778 C

3a.

☐ 1+1 is 2

3b.

☐ -0.25 -0.025 -0.0025

3c.

☐ 6 4

4a.

WRITE (-17+14.157+500)/3

☐ 165.719

4b.

WRITE 142\*0.013

☐ 1.846

4c.

WRITE (35.30-1.85)/(1984-1837)

☐ 0.2275510204081633

4d.

WRITE 0.07\*103.12

☐ 7.2184

or

WRITE (7/100)\*103.12

4e.

WRITE (1-0.15)\*157

☐ 133.45

or

WRITE 157-0.15\*157

5a.

WRITE round (12.7\*12.7)

☐ 161

5b.

WRITE floor (6.7/1.31)

☐ 5

5c.

WRITE 6.7 mod 1.31

☐ 0.15

5d.

`WRITE ceiling (141/8)`☐ 18

5e.

`WRITE 1351 mod 7`☐ 0

The remainder after division turns out to be 0, so 1351 is divisible by 7. We can also use the command:

`WRITE 1351/7`☐ 193

Now divisibility follows from the fact that the result of division is a whole number.

5f.

`WRITE "3859 seconds =", floor (3859/60), "minutes", 3859 mod 60, "seconds"`☐ 3859 seconds = 64 minutes 19 seconds

5g.

`WRITE 2 round (200/7)`☐ 28.57

5h.

`WRITE (3527 mod 7)+1`☐ 7

The following solution is not quite right:

`X WRITE 3528 mod 7`☐ 0

## 2. Memory: the PUT command

1.  
☐ 18.512 19.513 20.514

2.  
 PUT 1.5\*a IN a

3.  
 PUT (p+q)/2 IN average

4.  
 PUT a+b, a+b IN a, b

or

PUT a+b IN a

PUT a IN b

The following is wrong:

PUT a+b IN a

X PUT a+b IN b

because the first command changes the value of a, the new value being used in the second command.

5.  
 PUT (distance/time)\*60 IN speed

or

PUT distance/(time/60) IN speed

In both versions, the brackets are needed because omitting them would leave an expression with / and \*, where the order in which these operators are handled makes a difference for the result.

The \*60 and /60 are there because there are 60 minutes to the hour.

6.  
 WRITE 9\*round(n/9)

7a.  
 PUT 12 IN foot  
 PUT 3\*foot IN yard  
 PUT 1760\*yard IN mile

7b.  
 WRITE mile+3\*yard+2\*foot+2.5  
☐ 761642.5

7c.  
 PUT floor (1234567/mile), 1234567 mod mile IN miles, inches  
 PUT floor (inches/yard), inches mod yard IN yards, inches  
 PUT floor (inches/foot), inches mod foot IN feet, inches  
 WRITE "1234567 inch =", miles, "mile", yards, "yrd", feet, "ft", inches, "inch"  
☐ 1234567 inch = 1 mile 1097 yrd 28 ft 7 inch

Without using mod, it can be done this way:

PUT floor (1234567/mile) IN miles

PUT 1234567-miles\*mile IN inches

PUT floor (inches/yard) IN yards

PUT inches-yards\*yard IN inches

PUT floor (inches/foot) IN feet

PUT inches-feet\*foot IN inches

WRITE "1234567 inch =", miles, "mile", yards, "yrd", feet, "ft", inches, "inch"

☐ 1234567 inch = 1 mile 1097 yrd 28 ft 7 inch

Here is still another way, which does not use floor:

```

PUT 1234567 mod mile IN rest
PUT (1234567-rest)/mile IN miles
PUT rest IN inches
PUT inches mod yard IN rest
PUT (inches-rest)/yard IN yards
PUT rest IN inches
PUT inches mod foot IN rest
PUT (inches-rest)/foot IN feet
PUT rest IN inches
WRITE "1234567 inch =", miles, "mile", yards, "yrd", feet, "ft", inches, "inch"
□ 1234567 inch = 1 mile 1097 yrd 28 ft 7 inch

```

8.

```

PUT tar1 IN third
PUT tar2 IN tar1
PUT third IN tar2

```

9.

```

PUT 10*b+4 IN b

```

10.

```

PUT pnr mod 1 IN pnr
or
PUT pnr - floor pnr IN pnr

```

11.

```

PUT a, b IN b, a

```

To see that this really has the same result as the original three commands, suppose the original values of  $a$  and  $b$  were  $A$  and  $B$ . Then, after the first command the values of  $a$  and  $b$  are  $A$  and  $A - B$ , after the second they are  $B$  and  $A - B$ , and after the third they are  $B$  and  $A$ .

12.

If originally  $x = X$  and  $y = Y$ , then after the first command  $x = Y$  and  $y = -X$ , after the second command  $x = -X$  and  $y = -Y$ , and finally  $x = -Y$  and  $y = X$ . This result can be achieved equally well by:

```

PUT -y, x IN x, y

```



### 3. Repetition: the WHILE command

1.

The two occurrences in the original program of 142857 should be changed into 7, and 500000 into 100:

```
PUT 7 IN mult
```

```
WHILE mult < 100:
```

```
    WRITE mult
```

```
    PUT mult+7 IN mult
```

☐ 7 14 21 28 35 42 49 56 63 70 77 84 91 98

2a.

☐ 2

2b.

☐ 2 4 6

2c.

Nothing will be written.

3.

☐ 3

The same result could have been achieved with:

```
PUT n mod 17 IN n
```

4.

The number written will be the whole number nearest to  $n$  that is greater than or equal to  $n$ . The same result can be obtained by:

```
WRITE ceiling n
```

5.

```
PUT 0 IN nr
```

```
WHILE (nr+1)*142857 < 1000000:
```

```
    PUT nr+1 IN nr
```

```
    WRITE nr, nr*142857 /
```

or

```
PUT 0 IN nr
```

```
WHILE (nr+1)*142857 < 1000000:
```

```
    WRITE nr+1, (nr+1)*142857 /
```

```
    PUT nr+1 IN nr
```

6.

```
PUT 1984, 1000 IN y, sum
```

```
WHILE sum < 2000:
```

```
    PUT y+1, sum*1.13 IN y, sum
```

```
    WRITE y, 2 round sum /
```

☐ 1985 1130.00

☐ 1986 1276.90

☐ 1987 1442.90

☐ 1988 1630.47

☐ 1989 1842.44

☐ 1990 2081.95

7.

In both programs,  $a$  is the smallest odd number not yet written.

8a.

The numbers in this sequence are the powers of 2. In the following program, *n* is the smallest power of 2 that has not yet been written.

```

PUT 1 IN n
WHILE n < 1000:
    WRITE n
    PUT 2*n IN n
□ 1 2 4 8 16 32 64 128 256 512

```

8b.

The numbers in this sequence are the squares of the whole numbers 1, 2, ... . In the program, *i* is the smallest positive whole number whose square has not yet been written.

```

PUT 1 IN i
WHILE i*i < 1000:
    WRITE i*i
    PUT i+1 IN i

```

Here is a version which uses an extra target *i2* which contains the square of *i*:

```

PUT 1, 1 IN i, i2
WHILE i2 < 1000:
    WRITE i2
    PUT i+1 IN i
    PUT i*i IN i2

```

Another way to look at the sequence is: the difference between two consecutive numbers of the sequence is 2 greater than the previous difference, the first difference being 3, and the first number being 1. In the following program, *n* is the next number to be written, and *diff* is the next difference to be used.

```

PUT 1, 3 IN n, diff
WHILE n < 1000:
    WRITE n
    PUT n+diff IN n
    PUT diff+2 IN diff

```

8c.

In this sequence, the *j*-th number is *j* times the previous number, the first number being 1. In the following program, *n* is the next number to be written, and *i* is its serial number.

```

PUT 1, 1 IN n, i
WHILE n < 1000:
    WRITE n
    PUT i+1 IN i
    PUT n*i IN n
□ 1 2 6 24 120 720

```

8d.

This sequence can be thought of as a sequence of equal pairs, each pair containing numbers 111 greater than the previous pair, and the first pair containing 111 and 111. In the following program, *n* contains the number which will occur twice in the next pair to be written.

```

PUT 111 IN n
WHILE n < 1000:
    WRITE n, n
    PUT n+111 IN n

```

8e.

In this sequence, each number is the sum of the previous two numbers, the first two numbers being 0 and 1. In the following two programs, *n* is the next number to be written and *m* is the number just written.

```

PUT 0, 1 IN m, n
WRITE m
WHILE n < 1000:
    WRITE n
    PUT n, m+n IN m, n
□ 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

```

Without multiple PUT commands, it looks like this:

```
PUT 0, 1 IN m, n
WRITE m
WHILE n < 1000:
  WRITE n
  PUT m+n IN next
  PUT n IN m
  PUT next IN n
```

In the next version, a and b are the two numbers just written.

```
WRITE 0, 1
PUT 0, 1 IN a, b
WHILE a+b < 1000:
  PUT b, a+b IN a, b
  WRITE b
```

9.

```
WRITE start
WHILE start mod 100 <> 0 AND start mod 100 <> 8 AND start mod 100 <> 80:
  PUT 2*start IN start
  WRITE start
```

The target start plays the role of the number just written.

In the following version tail contains the number consisting of the last two digits of start:

```
WRITE start
PUT start mod 100 IN tail
WHILE tail <> 0 AND tail <> 8 AND tail <> 80:
  PUT 2*start IN start
  WRITE start
  PUT start mod 100 IN tail
```

10.

In the following program, the target to'be'folded contains the length of the side to be folded, other contains the length of the other side.

```
PUT length, width IN to'be'folded, other
WHILE to'be'folded >= 1:
  PUT to'be'folded/2, other IN other, to'be'folded
  WRITE to'be'folded, other /
```

11.

```
PUT 2 IN a
WHILE NOT a*a >= s:
  PUT a+2 IN a
WRITE a
```

The role of a can be described as: a contains a positive even number, all positive even numbers smaller than a having squares smaller than s.

The program may also be written without NOT:

```
PUT 2 IN a
WHILE a*a < s:
  PUT a+2 IN a
WRITE a
```

12.

```
PUT 1, 1, 1 IN s, p, i
WHILE p*(i+1) < 1000:
  PUT i+1 IN i
  PUT p*i IN p
  PUT s+p IN s
WRITE s
```

□ 873

13.

In this program,  $m$  and  $n$  are two consecutive numbers in the sequence, all previous numbers including  $m$  being smaller than 1000.

PUT 0, 1 IN  $m, n$

WHILE NOT  $n > 1000$ :

    PUT  $n, m+n$  IN  $m, n$

WRITE  $n$

□ 1597

14.

In this program  $factor$  contains a whole number greater than 1, all smaller such numbers not being factors of  $n$ .

PUT 2 IN  $factor$

WHILE  $n \bmod factor \neq 0$ :

    PUT  $factor+1$  IN  $factor$

WRITE  $factor$

#### 4. User-defined commands: HOW'TO

1.

MULTIPLY 2 UNDER 51

2. MULTIPLY -3 UNDER 10

The computer starts typing

□ -3 -6 -9 -12 -15 -18 -21 -24 -27 -30 -33 -36 -42 -45 -48 -51 -54 -57 -60 -63  
and will never stop.

3.

The command replaces the value of the target which is its parameter by the square of that value, so a better wording would be:

HOW'TO SQUARE n:

PUT n\*n IN n

4.

This command interchanges the values of the two targets that are its parameters, so:

HOW'TO SWAP a AND b:

PUT a, b IN b, a

5.

HOW'TO SQUARE'N'CUBE n:

WRITE n\*n, n\*n\*n

6.

HOW'TO LET target BE value:

PUT value IN target

7.

If we want to be able to specify a rate of 9 percent in this way:

DOUBLE 1000 AT 9

then the new version of the command definition should look like this:

HOW'TO DOUBLE start AT perc:

PUT 1984, start IN year, sum

WHILE sum < 2\*start:

PUT year+1 IN year

PUT 2 round (sum\*(1+perc/100)) IN sum

WRITE year, sum /

Of course, this command should only be used with a positive second parameter, since otherwise the value of sum would never grow beyond 2\*start and the command would never end. In fact, for *very* small positive values of perc, the command would not end either, because sum might keep its original value, the rounding operator destroying the very slight growth of sum each time. The same would happen for a very small initial sum.

8a.

ADD'FROM 2 AND 5 UNDER 500

8b.

HOW'TO ADD'FROM p AND q TIMES f1 AND f2 UNDER limit:

PUT p, q IN a, b

WRITE a

WHILE b < limit:

WRITE b

PUT b, f1\*a+f2\*b IN a, b

8c.

ADD'FROM 1 AND 1 TIMES 1 AND 1 UNDER 1000

ADD'FROM 2 AND 5 TIMES 1 AND 1 UNDER 1000

ADD'FROM 1 AND 1 TIMES 3 AND 2 UNDER 1000

ADD'FROM 1 AND 3 TIMES -1 AND 3 UNDER 1000

## 5. HOW'TO revisited

1.  
HOW'TO DOUBLE (3) UNTIL (100):  
    PUT (3) IN a  
    WHILE a < (100):  
        WRITE a  
        PUT a+a IN a

2.  
Since it is the very essence of the INCR command to change the value of a *target*, a target should be given as its parameter.

3.  
The slot *n* may be filled by any expression as a parameter, because no value is ever PUT IN *n*. The slot *k*, on the other hand, should be filled with a target, because *k/2* is PUT IN it.

4.  
HOW'TO A *n*:  
    SHARE total  
    PUT total+n IN total  
    WRITE total

Important: before the A command is used for the first number to be added, the target *total* must be given the value 0.

5b.  
The definition is copied:

*n* = 4   *p* = 8   *k* = 6   *pp* = 2   *f* = 1.5

```
HOW'TO MULT n:
  SHARE pp, f
  PUT n*f IN p
  WRITE p
  PUT pp*p IN pp
```

The internal targets are made unique:

*n* = 4   *p* = 8   *k* = 6   *pp* = 2   *f* = 1.5

```
HOW'TO MULT n:
  SHARE pp, f
  PUT n*f IN p'
  WRITE p'
  PUT pp*p' IN pp
```

The parameters are inserted in the slots, enclosed in brackets:

*n* = 4   *p* = 8   *k* = 6   *pp* = 2   *f* = 1.5

```
HOW'TO MULT (k+1):
  SHARE pp, f
  PUT (k+1)*f IN p'
  WRITE p'
  PUT pp*p' IN pp
```

The head line and the SHARE line are discarded:

*n* = 4   *p* = 8   *k* = 6   *pp* = 2   *f* = 1.5

```
PUT (k+1)*f IN p'
WRITE p'
PUT pp*p' IN pp
```

The resulting program is executed, using the extra memory for the internal target  $p'$ :

$n = 4 \quad p = 8 \quad k = 6 \quad pp = 21 \quad f = 1.5$

PUT  $(k+1)*f$  IN  $p'$

WRITE  $p'$

PUT  $pp*p'$  IN  $pp$

$p' = 10.5$

□ 10.5

Finally, the extra piece of memory is thrown away:

$n = 4 \quad p = 8 \quad k = 6 \quad pp = 21 \quad f = 1.5$

5c.

The product of the external target  $f$  and the parameter of the MULT command is written, and then the external target  $pp$  is multiplied by that product. The rest of the memory remains unchanged.

## 6. Texts

1.  
HOW'TO UNDERLINE text:  
WRITE text^"!"/  
WRITE "-^^(#text+1)  
The +1 takes care of the extra - character below the ! character.
2.  
WHILE #answer < 10:  
PUT answer^"!"/ IN answer  
or  
PUT answer^("!"^^((10-#answer))) IN answer
3.  
HOW'TO WISH:  
FRAME "Merry Christmas"
4.  
PUT t|1 IN t  
PUT t@2 IN t  
PUT t@#t IN t  
PUT t|(#t-1) IN t  
PUT "" IN t
5.  
The alphabetical order is: "#!< ITS it1 it12 it2 its ~@#?
6.  
HOW'TO STRIP word:  
WHILE word|1 = "s":  
PUT word@2 IN word  
WHILE word@#word = "s":  
PUT word|(#word-1) IN word
7.  
HOW'TO SHOW'TAILS t:  
PUT t IN w  
WHILE w > "":  
WRITE w /  
PUT w@2 IN w
8.  
HOW'TO SHOW'HEADS t:  
PUT t IN w  
WHILE w > "":  
WRITE w /  
PUT w|(#w-1) IN w
9.  
HOW'TO SHIFT text:  
PUT text@2^text|1 IN text
10.  
HOW'TO SHOW'SHIFTS start:  
PUT start IN t  
WRITE t /  
SHIFT t  
WHILE t <> start:  
WRITE t /  
SHIFT t



11.

HOW'TO CHECK'PALINDROME text:

PUT text IN t

WHILE #t &gt; 1 AND t|1 = t@#t:

PUT t@2|(#t-2) IN t

WRITE "The rest is: ", t

In this definition, the internal target *t* plays the role of a middle part of *text* with some letters omitted at the beginning of the word and the same letters, in reverse order, omitted at the end. The extra *#t > 1* in the condition of the WHILE command is needed to cover cases such as CHECK'PALINDROME "rotator". After three repetitions this word would have shrunk to the one letter text *a*, which would lead to problems with *t@2* in the next line of the definition, because it is impossible to take a text of length 1 from the second character. Note that the following solution without internal target is not correct:

HOW'TO CHECK'PALINDROME text:

WHILE #text &gt; 1 AND text|1 = text@#text:

PUT text@2|(#text-2) IN text

WRITE "The rest is: ", text

The problem with this shorter version is that it can only be used with a *target* as parameter, so an application such as:

X CHECK'PALINDROME "peep"

would be incorrect, since it would lead to the incorrect command:

X PUT ("peep")@2|(#("peep")-2) IN ("peep")

## 7. Types

1a.

The slot  $l$  should be filled with a parameter of type text, because of the condition  $t|1 = l$ , which shows that  $l$  must have the same type as  $t|1$ , which clearly has type text.

The slot  $t$  must be filled with a text too, because of  $t|1$  and  $t@2$ .

The slot  $c$  should be filled with a numeric parameter because of  $PUT\ 0\ IN\ c$  and  $PUT\ c+1\ IN\ c$ .

Furthermore, the slots  $t$  and  $c$  should be filled with targets, because something is  $PUT\ IN$  both, whereas  $l$  may be filled with any expression of type text.

1b.

Both slots may be filled with any expression of any type, since nothing is  $PUT\ IN$  either slot, and no special operators are used.

1c.

All slots must be filled with targets, since  $IN$  every slot something is  $PUT$ . The slot  $r$  must be filled with a numeric parameter, because of  $r+1$ . The type of  $p$  must be the same, because  $p$  is  $PUT\ IN\ r$ , and  $r+1\ IN\ p$ . For similar reasons the slots  $q$  and  $s$  must be filled with parameters of equal types, but in one application that type may be numeric and in another application textual.

## 8. Input: the READ command

1.

We must remove the command `PUT 0 IN t` from the definition. That means that, before using the command `ADD'IN shopping'costs` the first time, we should give the command:

```
PUT 0 IN shopping'costs
```

2.

HOW'TO BOX:

```
WRITE "Give 3 dimensions in cm." /
READ h EG 0
READ w EG 0
READ l EG 0
PUT h*w, h*l, w*l IN a1, a2, a3
WRITE "Area of the faces: ", a1, a2, a3, "cm2" /
PUT 2*(a1+a2+a3) IN area
WRITE "Total area: ", area, "cm2" /
PUT h*w*l IN volume
WRITE "Volume: ", volume, "cm3" /
WRITE "Area per cm3 volume:", area/volume, "cm2" /
```

3a.

HOW'TO BOXES:

```
WRITE "Give 3 dimensions in cm." /
READ h EG 0
WHILE h <> 0:
  READ w EG 0
  READ l EG 0
  PUT h*w, h*l, w*l IN a1, a2, a3
  WRITE "Area of the faces: ", a1, a2, a3, "cm2" /
  PUT 2*(a1+a2+a3) IN area
  WRITE "Total area: ", area, "cm2" /
  PUT h*w*l IN volume
  WRITE "Volume: ", volume, "cm3" /
  WRITE "Area per cm3 volume:", area/volume, "cm2" /
  WRITE "Give 3 dimensions of next box. Type 0 if you want to stop." /
  READ h EG 0
```

Note that when the user of this program wants to stop, only a single 0 need be typed as the first of the 3 dimensions. The other two don't need to be specified.

3b.

HOW'TO BOXES:

```
WHILE 1 = 1:
  WRITE "'Give 3 dimensions in cm." /
  READ h EG 0
  READ w EG 0
  READ l EG 0
  PUT h*w, h*l, w*l IN a1, a2, a3
  WRITE "Area of the faces: ", a1, a2, a3, "cm2" /
  PUT 2*(a1+a2+a3) IN area
  WRITE "Total area: ", area, "cm2" /
  PUT h*w*l IN volume
  WRITE "Volume: ", volume, "cm3" /
  WRITE "Area per cm3 volume:", area/volume, "cm2" /
```

Using the BOX command of exercise 2 gives a much shorter program:

HOW'TO BOXES:

```
    WHILE 1 = 1:
```

```
        BOX
```

The condition  $1 = 1$  may look rather strange, but it serves its role very well: it is always true, so the WHILE command will be repeated until the stop key is used.

4.

HOW'TO NUMBER'GUESS:

```
    PUT 1, 2 IN p, q
```

```
    PUT 2, 1 IN a, b
```

```
    WRITE "A number sequence starts with:", a, b /
```

```
    WRITE "Guess the next number." /
```

```
    PUT p*a+q*b IN new
```

```
    READ guess EG 0
```

```
    WHILE guess <> new:
```

```
        WRITE "No,", new /
```

```
        PUT b, new IN a, b
```

```
        PUT p*a+q*b IN new
```

```
        READ guess EG 0
```

```
    WRITE "Right."
```

## 9. Conditional execution: the IF and SELECT commands

1.

It would not make a difference if the third and the fourth alternatives were interchanged, but it *would* make a difference if we interchanged the first two alternatives. In this case, a word such as *cactus* would lead to *cactuses* instead of *cacti*, because the first alternative, with condition  $z = "s"$ , would be chosen.

2.

HOW'TO DOUBLE noun:

```

PUT noun@#noun IN z
PUT noun@(#noun-1) IN zz
PUT noun@(#noun-2) IN zzz
SELECT:
  zz = "us":
    WRITE noun|(#noun-2), "i"
  z = "s" OR z = "z" OR z = "x" OR z = "o":
    WRITE noun, "es"
  zz = "ch" OR zz = "sh":
    WRITE noun, "es"
  z = "y":
    WRITE noun|(#noun-1), "ies"
  zz = "lf":
    WRITE noun|(#noun-1), "ves"
  zzz = "man" AND "a" <= noun|1 <= "z":
    WRITE noun|(#noun-2), "en"
ELSE:
  WRITE noun, "s"

```

3.

The first two applications are all right; their output is:

☐ 3 5

☐ 3 5

The third application is wrong, and the computer will report the error that the types are mixed up.

The last two applications are correct:

☐ sideways

☐ sideways

4.

HOW'TO CHECK/PALINDROME text:

```

PUT text IN t
WHILE #t > 1 AND t|1 = t@#t:
  PUT t@2|(#t-2) IN t
SELECT:
  #t <= 1:
    WRITE "Yes, ", text, " is a palindrome."
  ELSE:
    WRITE "No, ", text, " is not a palindrome."

```

Note that the cases  $\#t = 0$  and  $\#t = 1$  are treated in one alternative.

5.

HOW TO NUMBER GUESS:

WRITE "Try to guess the secret number." /

PUT 44 IN it

READ guess EG 0

WHILE guess &lt;&gt; it:

SELECT:

guess &lt; it:

WRITE "higher" /

guess &gt; it:

WRITE "lower" /

READ guess EG 0

WRITE "Right!"

## 10. Lists

1.  
HOW TO REMOVE ALL item FROM list:

```
WHILE item in list:
    REMOVE item FROM list
```

2.  
IF z in {"s"; "z"; "x"}:  
 WRITE noun, "es"

3a.  
WRITE min pals

3b.  
WRITE river|1

3c.  
WRITE max pals

4a.  
The list {"four"; "five"} is used in alphabetical order, so the computer treats it as {"five"; "four"}. As a result, the computer will write:

☐ four

4b.  
In this case the second item of a text is asked for, resulting in the second character of the text:

☐ o

4c.  
The operator th'of is used incorrectly here, because the right-hand operand must be a list or a text, not a number.

4d.  
Here again the th'of operator is misused: the left operand must be a (whole) number, not a text.

5a.  
Put in its proper order, the list will look like {"111"; "22"; "3"}, so the output is:

☐ 111

5b.  
This list contains just one item, which, of course, is also the smallest item:

☐ {}

5c.  
The command:

```
WRITE min min {{5; 2}; {5; 2; 1}}
```

is equivalent to:

```
WRITE min min {{2; 5}; {1; 2; 5}}
```

which is equivalent to:

```
WRITE min min {{1; 2; 5}; {2; 5}}
```

After execution of the (right-most) min operator, this is left:

```
WRITE min {1; 2; 5}
```

So, the output will be:

☐ 1

5d.  
The list {} contains 0 items, so the output is:

☐ 0

6.  
WRITE 2 th'of {a; b; c}

7.

HOW TO SPOT DOUBLE:

PUT {} IN seen

READ text RAW

WHILE text not in seen:

INSERT text IN seen

READ text RAW

WRITE "There were", #seen, "different texts." /

8.

The target *u* will contain the upper case letter corresponding to the lower case letter in *l*. For example, if *l* = "d", the computer executes:

PUT (#{"a".."d"}) th'of {"A".."Z"} IN *u*

The list {"a".."d"} is shorthand for {"a"; "b"; "c"; "d"}, which contains four items. So, #{"a".."d"} = 4. This means that the computer executes:

PUT 4 th'of {"A".."Z"} IN *u*

The fourth item of {"A".."Z"}, which contains capital letters in alphabetical order, is "D". So, the computer executes:

PUT "D" IN *u*



## 11. Another kind of repetition: the FOR command

1.

HOW'TO PRINT'LIST list:

FOR item IN list:

WRITE item /

2.

FOR nr IN {a; b; c; d}:

WRITE nr

3.

In this program `total` plays the role of the total length of all values the target word has had in the preceding repetitions of the FOR command. Note that the first command is really necessary, because without it the command `PUT total+#word IN total` could not be executed, `total` having no value.

PUT 0 IN total

FOR word IN pals:

PUT total+#word IN total

WRITE total

4.

HOW'TO FAHRENHEIT f:

WRITE f, "F =", 2 round ((f-32)\*5/9), "C" /

5.

HOW'TO LIST'FAHRENHEIT list:

FOR f IN list:

FAHRENHEIT f

6.

HOW'TO ARROW n:

FOR i IN {1..n}:

WRITE "\*\*\*"^^i /

FOR diff IN {1..n-1}:

WRITE "\*\*\*"^^(n-diff) /

or

HOW'TO ARROW n:

FOR i IN {1..n}:

WRITE "\*"^^(3\*i) /

FOR diff IN {1..n-1}:

WRITE "\*"^^(3\*(n-diff)) /

7.

In this program the target `result` contains the letters of `text` seen so far in the FOR command. It contains them in reverse order.

HOW'TO INVERT text:

PUT "" IN result

FOR letter IN text:

PUT letter^result IN result

WRITE result

8.

```

PUT {} IN multiples
PUT 11 IN mult
WHILE mult < 100:
    INSERT mult IN multiples
    PUT mult+11 IN mult
PUT 13 IN mult
WHILE mult < 100:
    INSERT mult IN multiples
    PUT mult+13 IN mult
FOR nr IN multiples:
    WRITE nr
□ 11 13 22 26 33 39 44 52 55 65 66 77 78 88 91 99

```

or

```

PUT {} IN multiples
FOR single IN {11; 13}:
    PUT single IN mult
    WHILE mult < 100:
        INSERT mult IN multiples
        PUT mult+single IN mult
FOR nr IN multiples:
    WRITE nr

```

A fully different approach is followed in this program, where m11 contains the smallest multiple of 11 not yet written, and m13 the smallest multiple of 13 not yet written:

```

PUT 11, 13 IN m11, m13
WHILE min {m11; m13} < 100:
    PUT min {m11; m13} IN mm
    WRITE mm
    SELECT:
        m11 = mm:
            PUT m11+11 IN m11
        m13 = mm:
            PUT m13+13 IN m13

```

## 12. Tables

1.  
 PUT 'feet' IN plural['foot']

2.  
 We only have to add a SHARE line containing plural, and put a new alternative in front of the existing ones:

```
HOW'TO DOUBLE noun:
  SHARE plural
  PUT noun@#noun IN z
  PUT noun@(#noun-1) IN zz
  PUT noun@(#noun-2) IN zzz
  SELECT:
    noun in keys plural:
      WRITE plural[noun]
    zz = 'us':
      WRITE noun|(#noun-2), 'i'
    z = 's' OR z = 'z' OR z = 'x' OR z = 'o':
      WRITE noun, 'es'
    zz = 'ch' OR zz = 'sh':
      WRITE noun, 'es'
    z = 'y':
      WRITE noun|(#noun-1), 'ies'
    zz = 'lf':
      WRITE noun|(#noun-1), 'ves'
    zzz = 'man' AND 'a' <= noun|1 <= 'z':
      WRITE noun|(#noun-2), 'en'
  ELSE:
    WRITE noun, 's'
```

3.  
 HOW'TO TIME hour:
 SHARE diff
 FOR city IN keys diff:
 WRITE city, ':', hour+diff[city], 'h' /

4.  
 HOW'TO GET'SERIES s:
 PUT {} IN s
 READ text RAW
 WHILE text > '':
 PUT text IN s[1+#s]
 READ text RAW

In the next version, the target nr is used to keep the number of entries already present in s:

```
HOW'TO GET'SERIES s:
  PUT {} IN s
  PUT 0 IN nr
  WHILE text > '':
    \nr = #s
    PUT nr+1 IN nr
    PUT text IN s[nr]
  READ text RAW
```

5.

HOW'TO FILL'TABLE table:

```

PUT {} IN table
WRITE 'key: '
READ key RAW
WHILE key <> '':
    WRITE 'associate: '
    READ associate RAW
    PUT associate IN table[key]
    WRITE 'key: '
    READ key RAW

```

or, without the internal target associate:

HOW'TO FILL'TABLE table:

```

PUT {} IN table
WRITE 'key: '
READ key RAW
WHILE key <> '':
    WRITE 'associate: '
    READ table[key] RAW
    WRITE 'key: '
    READ key RAW

```

6.

HOW'TO INVERT table:

```

PUT {} IN inv
FOR key IN keys table:
    PUT key IN inv[table[key]]

```

or

HOW'TO INVERT table:

```

PUT {} IN inv
FOR left IN keys table:
    PUT table[key] IN right
    PUT left IN inv[right]

```

7.

HOW'TO INVERT table:

```

PUT {} IN inv
FOR key IN keys table:
    SELECT:
        table[key] in keys inv:
            INSERT key IN inv[table[key]]
    ELSE:
        PUT {key} IN inv[table[key]]

```

### 13. Random choice: the CHOOSE command

1.  

```
FOR i IN {1..5}:
  CHOOSE length FROM {3; 5; 8}
  CURSE length
  WRITE /
```
2.  

```
CHOOSE answer FROM {"yes"; "yes"; "no"}
WRITE answer
```
3.  

```
HOW'TO PICK item FROM list:
  CHOOSE item FROM list
  REMOVE item FROM list
```
4.  

```
HOW'TO SCRAMBLE l:
  PUT l IN list
  WHILE list > {}:
    PICK item FROM list
    WRITE item /
```
5.  

We only need to change the command: PUT 44 IN it

into, for instance: CHOOSE it FROM {30..300}

resulting in:

```
HOW'TO NUMBER'GUESS:
  WRITE "Try to guess the secret number." /
  CHOOSE it FROM {30..300}
  READ guess EG 0
  WHILE guess <> it:
    SELECT:
      guess < it:
        WRITE "higher" /
      guess > it:
        WRITE "lower" /
  READ guess EG 0
  WRITE "Right!"
```
6.  

```
HOW'TO THROW nr DICE:
  PUT 0 IN total
  FOR d IN {1..nr}:
    CHOOSE eyes FROM {1..6}
    PUT total+eyes IN total
  WRITE total /
```

7.

We will use the table `times` to contain the number of times each outcome of a throw has occurred so far:

```

PUT {} IN times
FOR eyes IN {1..6}:
  PUT 0 IN times[eyes]
FOR i IN {1..120}:
  CHOOSE eyes FROM {1..6}
  PUT times[eyes]+1 IN times[eyes]
FOR eyes IN {1..6}:
  WRITE eyes, ":", times[eyes] /

```

8.

In this program, the internal target `copy` contains that part of the table for which no correct answer has been given yet. As soon as a correct answer *is* given for a certain key, the corresponding entry is removed from `copy`. It is better not to use the original table for that purpose, because it would become empty at the end of the quiz, so we could not use it any more.

HOW TO QUIZ table:

```

PUT table IN copy
WHILE copy > {}:
  CHOOSE key FROM keys copy
  WRITE key /
  READ answer RAW
  SELECT:
    answer = copy[key]:
      WRITE "OK" /
      DELETE copy[key]
  ELSE:
    WRITE "Wrong, ", copy[key] /
WRITE "That was all."

```

## 14. Compounds

1.

```
PUT {} IN neg'pos
FOR number IN numbers:
  INSERT -number, number IN neg'pos
FOR pair IN neg'pos:
  PUT pair IN neg, pos
  WRITE pos /
```

In fact, we can use a simpler method in this case:

```
PUT {} IN negs
FOR pos IN numbers:
  INSERT -pos IN negs
FOR neg IN negs:
  WRITE -neg /
```

2.

```
PUT {} IN t
FOR i IN {-2..2}:
  FOR j IN {-2..2}:
    PUT 0 IN t[i, j]
```

3.

In the program, the target `old'roll` contains the most recent roll:

```
PUT {} IN freq
FOR i IN {1..6}:
  FOR j IN {1..6}:
    PUT 0 IN freq[i, j]
CHOOSE old'roll FROM {1..6}
FOR i IN {1..120}:
  CHOOSE new'roll FROM {1..6}
  PUT freq[old'roll, new'roll]+1 IN freq[old'roll, new'roll]
  PUT new'roll IN old'roll
```

ONTVANGEN 5 JUNI 1984