Aristo Tacoma

Art of Thinking

Vol. I: G15 PMN Programming for Kids

```
*====================================*
|                                    |
|    mousefun=            ck         |
|                                    |
|    ll:1                 se         |
|    q1                              |
|    60                   ex         |
|    activepause                     |
|                                    |
|    starspeople          lo.        |
|                                    |
*====================================*
```

****
Publication info: [[[MANUSCRIPT STILL BEING WRITTEN!!!]]]
All the 5 volumes in Art of Thinking by Aristo Tacoma to
be available for free on the net. This is the 1st volume.
See links at norskesites.org/fic3/fic3inf3.htm
This first volume is particularly oriented towards the
youngest readers. Forthcoming volumes build on this.
This is also published in paper, in 2018, by Yoga4d:VRGM,
Avenuege Library with ISBN number for National Library
of Norway assigned to Volume 1: ISBN 978-82-996977-9-8.
You can freely copy and distribute this text as long as
the text is whole and unedited and unshortened and
includes this notice and it is done so as to further
G15 PMN use. Copyright the author. Author's pen name
is: Aristo Tacoma. Publisher info: www.avenuege.com
Stein Henning Braten Reusch alias S.R. Weber asserts the
copyright for publications in his artist name Aristo
Tacoma. Yoga4d Avenuege Library, Yoga4d:VRGM, Oslo, 2018.
****

PRELUDE

According to Alice in "Alice in Wonderland" by Lewis Carroll, what's the use of a book if it doesn't have pictures? There are lots of images in this book--in your mind, when you read the easy text here. And if you want more images, type the examples into your PC and it'll make images. Does it satisfy Alice? I don't know. But for a book to teach a programming language, I think it is the right way entirely. Hope you like it, too!

In the upcoming volumes in this Art of Thinking series we will build on what is here, but slowly, gradually, coming to think about thinking in the purest sense; and hopefully also clarify how we are absolutely not machines and how we can learn from work with machines about how to enable our intuition.

Spelling takes are by the angels.
--Anonymous

Note to the youngest readers:
I have tried to avoid the longest words but sometimes I have forgot myself and they are there! But then, just read on. Usually everything that is important is mentioned more than once. And as you read it a second time, you'll find that words that made little sense to you the first time, makes more sense then, as you get used to them, and see how the words are used. Play with your thoughts around that which you want to learn--play and experiment, try out things, and you'll pick it up soon enough. And good luck!

Note to the not youngest readers
I say 'for kids' because I mean to write to anyone who can read English texts, also slightly more complicated texts, but who do not have lots of experience with computers; and I try to use a language that is simple. But of course this may just happen to make it interesting to adults, as well. Wasn't it the physicist Albert Einstein who once said that only if you can express it simply, have you understood it? So this is not merely for kids,--it is about making computer programming simple.

Note to those who want to learn also other programming languages but who are thinking of starting with G15 PMN
In today's world of also more or less 'corporate' programming languages--more or less owned by, or driven by, and modified by, companies that in turn also offer such as 'programming certificates' and possibly also jobs, certainly there are people who, perhaps having hardly ever touched a programming language before, wonder whether G15 PMN could be a gentle way to begin at the beginning.

There is no gentler way to begin at the beginning than G15 PMN! So go ahead! But to be fair, open and honest, I wish it to be stated strongly, right now: when G15 PMN was made, it wasn't exactly because the other programming languages seemed to be very ideal! It was made also so as to make up for a lot of the troubles one easily run into with other programming languages. Once you know this, then you can go ahead with G15 PMN with this idea central in your mind: that once you know what it means 'to code' a function, an algorithm, then you have something that--although clothed in perhaps very different words--you can build on as a core expertise no matter what programming language you take up next. Most other programming languages hides things which are here up front (such as 'the stack', and 'warps')--but there is no harm in knowing a little more of what goes on under the lid when you then take up a language which clothes things in impressive-sounding words like 'objects' and 'threads' and 'classes', or whatever words they do clothe coding in.

To give you a little more motivation for the pathway of starting out with G15 PMN before taking up other programming languages, I will suggest that it is easy to build self-confidence in interacting with a PC that has G15 PMN on it, while experimenting with new programs. This self-confidence is something that you can have with you, as a permanent factor, also when you go into wholly different approaches to programming, which (unlike a few nobler examples, like Wall's Perl or Wirth's Pascal) might feel as though they are taking the programmer through a bureacratic process in order to get anything done (and sometimes literally so, when a company is so controlling in their attitudes that they insist on looking at a program before allowing it to be performed on a computer of their make).

Whatever you choose as pathway to programming, be sure that you work with a PC that doesn't have dense programs pretending to have smartness hanging in the background, trying to 'assist'. Programming is fundamentally a relationship to your own thinking process where the machine is no smarter than a mirror. It is the almost delicious stupidy of the machine that makes programming create an intensification of your own mind when you do good coding.

FOREWORD

As you know, most forewords to books (when written by the author of the book) are written long after the book itself was written, and contain various excuses for why the book is the way it is and hope you like it anyway and so on.

This foreword ain't so. (Besides the prelude and the notes before this already has that type of stuff in it.)

The day, today, as I'm writing this, is sunny, and I'm sitting in a park and crowds, tiny crowds of people are around and children are playing and screaming and shouting. And, chewing on my pen, I'm wondering how to write this book.

I'm asking myself, what could possibly be the point of suggesting to these children and youngsters and young adults--and any adult--to spend time with something as technical and abstract as a "programming language"?

Why not just spend time with the concrete? And with natural language--such as getting good at English, develop humour and wittiness there? And to exercise physically, and to train elegant dance motions and postures and draw and paint and indeed do what's life is all about? So, to repeat my own question, why should anyone bother about programming?

Okay--some might answer--it could be a bit useful. To program probably means, for instance, you get better about handling numbers. Which again may help you to think better about money and money-making and that's helping everything else.

Yet--I would object--most other fields of learning of any kind are more truly useful than programming, in that most other fields somehow leads to concrete skills in dealing with life and opening up to life, perhaps.

But hang on a moment. What was that phrase we just used? "Opening up to life"--that phrase!

I will now suggest a real answer to why you--yes, you--should learn G15 PMN programming if you haven't already done so. And indeed any one. And I will use that phrase--"opening up to life". Here is what I claim--and please consider it:

To learn programming, and to work on making programs--and looking at them--opens your mind and heart to a greater experience of the orders of nature and life, rather as learning to draw can help you to look at the beauty of another's face.

The sun is shining very brightly now--after days of clouds and rain--and I forgot to bring my sunglasses. If I try to look straight at the sun, I have to close my eyes four-fifth or so. And as I do it, bursts of radiant golden bows as if emerging from the golden-yellow-slightly bluish disk of the hot sun flicker in synchrony with any slight motion of my eyelids. It's quite a display. Life is full of harmonies, orders, patterns, and experiences, which can be read like a symphony of coincidences, synchronicities. And, like waves on the ocean, nothing in experience stands still but the music, the orders, flow on and on.

In all this vast movement, we have to relate to, and often change, our own expectations, plans, hopes, desires --and yet when the day is suddenly very beautiful, the mind as if orders itself. Intuition--your capacity to get things right without quite knowing how or why--comes more easily then, in what we can call "moments of harmony" or "moments of meditation".

"Meditation", the word, comes from an ancient greek word relating to good and proper "measures", as does "medicine" and "moderate".

So, in ancient traditions, words of deep meaning and with nice sounds to them, and patterns--called "mandalas", for instance--have been used to calm the mind of devotees of many religions. And in all these traditions, the experience of the connection between inner peace and silence, on the one hand, and a sense of beauty and love, on the other, has been commonly reported.

The simple pattern, the good sound--gently, effortlessly repeated--or the ritualistic action, done with attention for the sake of the energy of attention--these things, then, are part of a life where experience goes deeper, where the sense of art of living becomes possible.

And so, can it not be, then, that in the nice little repetitive patterns and words involved in a great programming language like G15 PMN (it is great, although, as creator of it, it is immodest that I claim it)--can lead to a more meditative life? A life in which you are constantly more open to the beauty of all experience? The sharpened sense of numbers you get by working with what is called "32 bit numbers" in this language is also no impediment to deep experience of the order of nature. Inside numbers there are beautiful patterns--they are not just heaps, but highly organised.

To conclude this foreword, I would suggest that whatever usefulness we may or may not find connected to programming, even a little bit regular touch, weekly or perhaps daily, with something like G15 PMN is a connection to something of the inner workings of the very fabric of existence. If that sounds too grand, let us bear in mind that notable thinkers in ancient Greece thought the universe was "made up of numbers". Add to that a little (or a lot) movement, and you get algorithms, functions. And yet, the universe may have this inside its core, and yet be beyond it: for the universe is no machine, but alive, like you and me.

As motivator, then, to look into G15 PMN code, and do your little experiments with it--or make full programs on your own--is the idea that you are sort of cleansing your mind of the confusions of past experience and readying your mind to experience life anew. As if for the first time, and yet with a sense of constantly learning at your soul and spirit level.

A.T. April 10, 2018

P.S. Note that it is very possible that some slight mistakes on occasion have crept into the program code included in this book. At some point, the manuscript is finalized, even if not all those issues have been discovered and correct; pls be forgiving about this! :)

FOR ADVANCED STUDENTS INTERESTED IN THE ENTIRE FIVE-VOLUME SERIES 'ART OF THINKING'--AN INTRODUCTION TO ALL OF THEM

How can we best think about thinking when we are, as it were, wrapped into thinking? There is such a vast quantity of studies into thinking that begin with drawing up a map of thinking,--a map that is then fought for and which, eventually, has to be dropped. Perhaps this is because thinking is infinitely complex, and because, when we think about thinking, we must find in ourselves a vast humility for the subject matter and be cautious about 'maps'.

I also think we have to be careful about drawing too many deductions from our natural language about the area of thinking. Natural language serves many other purposes than mapping the nature of the mind. While a natural language with a great and flexible number of general concepts like English is a great thing to have during such an enquiry, it can also mislead and so we have to apply great care in language use.

Therefore,--and to some, this might seem surprising at first--I decide not to talk any much about thinking in the first volume, but rather sketch a working programming language from the very start. An artificial computer language is (when designed, as G15 PMN has been, to be as simple, beautiful and elegant as possible) a great advantage in our study. For then we have something that is related to English yet not fully dependent on it, something which is like a natural language in some ways but very different in other ways. So, by enquiring into thinking while having both instruments at hand, we can do a better job than if we have only one of them.

A computer language, like the G15 PMN I've made, has a great advantage compared to a discipline like 'geometry', 'algebra', or the like, in that it doesn't rely on much of 'intellectual agreement' between scholars, but rather shows its worth by the integrity of its technological functionality. The dissolution (in practise) of the area of 'foundational mathematics' in the 20th century after Goedel's 2nd Incompleteness Theorem is a case in point.

Just as it is the task of the first volumes in this series to create an as vivid understanding of the G15 PMN formal language as possible, also at more advanced levels, so it is the task of the completing volumes in this five-volume series to be able to call on both this and natural language while not being trapped in either, as we explore thinking. By necessity, this enquiry into mind, thinking and intuition must attend to the greatest context, ie, cosmos, and so we will naturally benefit from the theory of cosmos that I believe is the most simple interpretation of the core results of the past hundred and fifty years of studies of energy processes, atomic & subatomic processes, namely the Super-Model Theory. Inside the Third Foundation app in G15 PMN there's a text that describes the theory. That text has been published as part of the printed book "The Beauty of Ballerinas" by me alias SRW, Avenuege 2017. In that text there is a full set of acknowledgements which is relevant also to the development of the programming language G15 PMN.

The study of an art of thinking should naturally lead to enhanced skills in the art of making decisions, and contribute to several related quests, like dialogue, meditation, harmony in relationships. In that manner, it is my sense that these volumes can contribute harmoniously to society and not just practically to handle computers and robots, or philosophically as a theory of mind.

In this regard, it is worth noting that G15 PMN has been so designed that any technology no less than that which evolved at the beginning of the 21st century AD should be able to peform G15 PMN. It is in that sense a formal language that can be spoken about regardless of any future state of technology and science, as long as the natural minimum of pesonal computing technology does exist, as I'm sure it always will. My bet is also that English in its present inter-cultural form always will be much in use.

PART A: GETTING ACQUAINTED WITH G15 PMN

Part A, chapter 1: SAY "HI" TO G15 PMN

Why is it called "G15 PMN"? It's just a name, of course. Ideally, the name of the language should tell you something about it, and has a nice and unique sound to it as well.

First, "G15". It is a name for the core. The core is meant to be part of something generous, graceful, etc. "G" on its own sounds rather like "gee", and has a bit of rhyme with "fifteen". This core, nearest the electronics that makes up a personal computer, is about numbers in movement. So a number being part of its name makes sense.

Fifteen is a neat number; it is, as you know, 3 times 5, and 3 together with 5 is a combination we come back to in discussing "the golden ratio". And here we have a pair of numbers that can help you in drawing and painting. It's also possible to say much more about how 15 is meaningful, --it being small, yet big enough so that when we count from 0 to 15 we have what we will discuss later as "four bits". And more so.

Then "PMN". Mostly, it's just a good, short phrase, pee-emm-enn, again a rhyme within itself and a bit same type of sound as G15.

Where G15 on its own is mostly about numbers, PMN is a sort of layer on top or around G15 to make it more easy for us, more natural, more meaningful. Perhaps the "MN" is for "meaning", and "P" for "primary" or first. Perhaps M is for "matrix"--something we discuss later--or for "match" or for a big word about all existence, "multiverse". Then "N" may be as in "net"--something we'll briefly touch on when we say something in a chapter about robots and then "FCM"--or by a word like "noetics", meaning mind or the like.

But it is not the machine that has mind when you use G15 PMN--you have mind, and you get more activated in your mind as you use it. In a way, it is a language for the mind to speak to itself in a different way than what a natural language like English can offer.

In having a keyboard and a mouse and a monitor, all connected to an open personal computer, or PC, with G15 PMN in it, you have a kind of tool for thinking at your fingertips. The machine won't do any thinking for you, but like a camera in a way, it can give valuable impulses--what we call "feedback". (Some people have sometimes claimed that they can make machines think--don't believe these people.)

Most things in life are a bit gradual, on the move, not quite one way or another way. A big word for this is "analog". In doing something with the G15 PMN programming language, we find that things typically are either one way or another way. A word for this is "digital".

So we also say, "digital computer", "digital electronics". (Technically, we can say that the digital electronics of a G15 PMN computer has in it a G15 PMN CPU, or Central Processing Unit, of a 32-bit kind, which is not made by means of a 'microprocessor' but as a combination of many things we call "intraplates". But this is not necessary to know in order to learn and enjoy this language. By the way, if your computer has a key on the keyboard named "DEL" or "DELETE" it is a different type of CPU in it, and G15 PMN is running on top of it; the DEL key is then used to connect to this other CPU.)

Sometimes there are bridges between the analog and the digital--fortunately! For instance, we can set up a digital computer to work together with machinery that is not digital but more analog, so as to produce analog, normal music we can listen to and dance to or relax to.

In each of our meetings with the programming language G15 PMN, we will either type something into the language itself (and we always use the keyboard, so it is time well spent to learn to touch-type on it without having to look at the letters on the keys all the time!)--or we will put it into bright green cards withy the two columns of large, easily redable letter. And then, after putting the cards in order, we make the PC go through one or more of them.

Then, if we have typed something wrong, we can go back to the card (the PC will tell us which number it has, so we'll find it easily, each time). There, we'll fix it and we can try again. This is called "program correction". It can take even more time than making the first form of the program--and all in all this is what we call "programming". Even the best programmers set aside lots of time for program correction. Sometimes, though, there's hardly anything to correct, and the program just works. That's fun, but it's not a goal to do away with program correction. Program correction is part of our thinking about what we really want the computer to do. Often we discover that we haven't entirely thought it through in this phase, and must lean back or go for a walk or come back to it next day with some fresh ideas. So program correction isn't only about correcting spellings and such: it is as often to look at our own thoughts and think even better thoughts--and then to change the program and see if the PC does what we thought it would do after this.

So, we either type something straight into the programming language--into what we also call "the terminal"--or we type it into cards that we save, one by one.

The terminal is a fun place to begin and it's also very useful in checking how a program works. The terminal we use, normally, in doing whether simple or very advanced G15 PMN programming, is also called "the TF terminal". Sometimes we just say "TF" or "tf". This is short for a long phrase which sums up stages in how PMN was shaped. The full phrase is "The Third Foundation" (it isn't necessary to understand just why it is called that, just remember that when we say "terminal" or "tf" we usually mean the same type of working with the language as if somebody says "third foundation").

Have you ever played a game, such as TexasStars, on a PC with G15 PMN?

Then you have possibly done it this way--which is very similar to how we start the TF terminal--so I suggest you start up TexasStars this way if you aren't already familiar with this way of starting up that game of ours.

1. You switch on the G15 PMN PC (and if it is already up and running with G15 PMN, you take it off first).

2. You type the three letters

```
     mnt
```

and you press lineshift. Some call this large button to the right of the letters for the ENTER-button, and we sometimes say it one way, sometimes the other way. The old way of saying it--from the times of the mechanical type-writers--was "carriage return", so on occasion you'll see the two letters CR to mean just the same type of thing. (G15 PMN often wants you to press that lineshift button, so when it appears to be waiting for some input and you don't know what to press, it is often correct to press that button.)

The three letters mnt are short for "mount", and it means, let's load some stuff into the PC.

3. The menu shows and you press the digit one:

```
       1
```

4. Then you type the number of the program--also called "application", or "app", for short, which in the case of TexasStars shooting game are these seven digits:

```
       101010
```

As you press lineshift after these then it should say something about SUCCESS. (If it doesn't, get somebody to help you set up the computer so that it has both this TexasStars app and the TF app in it.)

Press lineshift again and the menu will open.

5. At the menu, press the CTR button (also sometimes named CTRL)--which means 'control'--and hold it in, and, while holding it in, also give the W-key a quick press. Then release your hold on the CTR button. This we can write as:

```
       CTR-W
```

This activates the mouse, so you can use the mouse to start programs. (A click on the right of the mouse makes switches off the menu mode again. Then the menues can be changed, what we call the 'edit' mode. To get the mouse back to work if you hit the right side of it, press CTR-W again.)

6. Now you can use the mouse to start the program. Look for an arrow or flower-like type of thing, in this case after a big F and before a big number 1. Find this place with the mouse pointer, and click on the left side of the mouse to start the program. The whole app starts now, and a press on the large lineshift button is all it should take to get you going with the game! You shoot at the mysterious objects in the sky by a click on the mouse, and move by a push on left and right arrows. ESC to leave the game.

7. When done playing, type

```
       qu
```

and press lineshift to get to the menu.

8. To leave the menu and the other cards--we also call this the CAR editor--press

```
       CTR-Q
```

in the same way you pressed CTR-W above. Then type

```
       reb
```

and press lineshift, and that allows the G15 PC to reboot and freshen itself and its socalled "memory" or RAM, so you can start it anew.

To start the G15 PMN language in its TF terminal, all you have to do is to go through exactly the same motions again, only choose to type, not 1010101 as app number, but 3333333 (you can just type as many 3's as the field allows, so it fills up to seven digits). Do the CTR-W and the mouse-click to start it.
  Done it? Great!!!

Now that you have the Third Foundation, or TF, terminal, clear and present on your Personal Computer's display, you see there a sort of greeting graphics--a square made of a lot of lines and a bit of a triangle made out of tiny squares, and so on. As you press lineshift several times, you kind of carve a bit of black in that bright green graphics each time. If you press lineshift many times, you make the whole column black.
  Let's connect to G15 PMN, say hello to it. Type the two letters:

```
       hi
```

and press lineshift. You might enjoy the greeting it gives you. It was actually pretty much the first thing I got PMN to do, when I made it. It wishes you good luck with large bright green letters on black. You find that this bright green makes the world around it look quite rosy--it is something called 'afterglow'--just like a yellowish sun makes the sky around it blue, so does a bright green computer monitor makes any white near it look a bit pink or red or something like that. It's called "complementary colors". But bright green, or yellow-green, is also found to be both relaxing and positive whether you work with it early in the day or late in the day or at night.
  Some might ask, "Isn't it best to have a color PC? For then you can look at such as photos more like things are in real life." But think about it: do you really wish the PC to go so near real life it looks like it? Is that the point of computing machines? Or is it the point that they are going to be there for this and that and the other task --including calming our minds--but then we want to go out and leave the PC and experience the full colors of real, analog life. We don't want to make the PC into something we get addicted to, so we get lazy and do nothing but steer at it all day. So the PC is a good thing only if we don't allow every desire to have its say but keep back a

little, keep it back a little, how much we develop it. Nobody who is sane want the machines to take over life or to melt with the machine somehow, and so the machines should not, must not get over-developed. And if they have been over-developed in the past--as they have--let us not contribute further to this over-development, but go beyond it and focus on what is good.

So you have typed in "hi" and you can do that again and again and you get the same greeting over and over again. Let's have the PC type your name in nice letters. It's time to use some of the other things on the keyboard than letters and numbers. For instance, above the digits you find many other things--we call them "characters" or "special characters". Over the digit 6 you get to ^ and over the digit 7 you get to &, if you press SHIFT key and hold it in while you also give the digit a quick press. The SHIFT key is sometimes written as "SHIFT" on the keyboard, at other times it is a sort of very broad arrow pointing up. It is the same key you use for getting big letters in when typing a text. (At some keyboards, what is above the digits may be moved around a little bit, but experiment and you'll find all you need.)

But first, let's clear the screen. Type

```
       ce
```

and the screen gets all black. You can think of it as c as in "clear" and "e" as in "entire screen" or "everything". This happens when you press lineshift.

Type it in using small letters. (In the TF terminal, the big and small letters are rather the same, though, only that a dot appears on top of the letters that are meant to be thought of as big.) Don't put any blanks before these little commands. But if you do, just type it in once more.

So, what's your first name? Type this into G15 PMN:

```
       &yourname&
```

only that you type whatever your name is inside the & and the &. Then press lineshift, and if you did it right, there is no message from G15 PMN. It just stores that name and you can get it out again, this way:

```
       b9
```

Nice, right? It gives you your name, but it uses more smooth letters than what it usually uses for programs, because you said "b9". Try the same but slightly different --like this:

```
       &yourname&
       pp
```

So you type your name again, with the & around them, and then you use pp instead of b9. The "p" means "print", and "pp" simply means that it is a very simple and often used print command, and two p's are easy to type in. "B9" sounds like "benign" which means "nice, good, healthy", when you read it out. So we call the type of pp letters one type of "font" and the b9 type of letters for another type of "font". And so you have a B9-font and you have another font, that we sometimes call "robot-font", because --like robots--it is a bit square and machine-like. It is also easier to use when you do much programming in G15 PMN than any other font, this robot-font--because it is shaped so that this language is easy to work with using it.

Ready for another experiment with G15 PMN? The computer is nothing if not repetition! So let's get it to repeat your name ten times. Ready for it? A bit of typing we shall do. Can you manage to type this exactly? Try it. Then do it again if it didn't work. If all gets funny, type 'qu' and exit the G15 PMN terminal, press CTR-Q, type the 'reb' command to let G15 PMN refresh itself. As you start it up again, you'll find that it is entirely clear and it has no scars or memories of the last session--which is one of the wonderful things about programming--you get infinitely many new chances!

So, let's type in something that allows us to create a new word in the language. That word will exist just for this session, for we do not store it on a card. So almost any name will do. Shall we call it "myname"? So we are going to say myname= something something. This "=" is usually high up on the keyboard, around the area where the lineshift button is. It is the "equal sign". We type in some stuff--at least one line, maybe several--and then at the last line in this little program, we put a dot, ".". Okay? So what we will type in is what we did above, but we are going to say--do this ten times--and the way we say it to the PC is this way: LL:10. Or with small letters, ll:10. This means, let's have a "loop". A loop is another word for "repetition". It wants to know what--how many lines to repeat--and so each time we type in LL: something then it wants, on a lower line, to have the letters LO, or --with small letters to be exact, "lo". So here we go, type in this as exactly as you can (just be sure to type your first name where it says 'yourname'):

```
       myname=
       ll:10
       &yourname&
       b9
       lo.
```

If you typed this exactly--being sure to type the letter "l" rather than the digit one where it says "ll" and where it says "lo"--it will say nothing at all. The principle is "no news is good news"!

If it says nothing, it means that it perfectly handled what you typed. It means you succeeded, most likely, in making a new word in G15 PMN for this session. (If not, just restart the PC the way we said, and try over again. This is a very normal thing to do for any professional, advanced programmer--get to love the restart of the G15 PMN PC!)

So to check out your new command, your new word, type this:

```
       myname
```

and as you press lineshift, you should get the PC to give you ten copies of your own first name on the computer display. I'd say that's enough for a first session, and let's complete the session while the game is going good. So type

```
       qu
```

and press lineshift. At the menu, press

```
       CTR-Q
```

and then type

```
       reb
```

and at lineshift, the G15 PMN will refresh itself. Next time, we will see how we can use cards, so that it can store something between our sessions. For what we type in during one session in the TF terminal isn't there the next time we take that terminal up, but that which we store in cards is more easy to get up the next time on the same PC. Of course, it may be that you are using a PC where the cards are constantly cleansed in order to keep the PC in top shape for the use of many people--but even so you can learn to use cards while you sit at the PC. Then later on you can get to work with a PC that is more your own, and use the same skills you build up here.

Part A, chapter 2: CONTROL YOUR G15 CARDS

The Personal Computer, at its best, is a sort of greatly advanced paper-and-pen type of thing. Similar to paper and pen work, you can gather yourself, compose your thoughts, by work with a Personal Computer. And it is calm and it is easy to get the same responses each time, so you get a nice sense of overview. How important isn't that in a world where sometimes our meetings with each other are full of openness and lovely--and sometimes not so lovely--uncertainties! Just like sleep is in contrast to wakeful-ness, and a starry night is in contrast to sunlight, so is working with pen and paper, or with a Personal Computer, or with reading a book, or making a drawing or making a painting something that provides the right type of calm in between our social life with friends. We can't be social all the time without getting exhausted--and it's much harder to be friendly when we are exhausted, especially if we are much exhausted. So a healthy good life needs these contrasts.

So, the Personal Computer is not supposed to be fluid and unpredictable and moody and someone you have to talk convincingly too. No, it is supposed to be just like pen and paper but with this wonderful capacity it has to provide some feedback, some machine-movement, in relation to what you type into the machine.

And then there are some things the machine allows us to do easy--like sending information across distances, and keeping tracks of numbers if you run a shop, and so on: this is all part of the Personal Computer.

But, as you perhaps know, the Personal Computer doesn't do a thing unless it is given a program. And that program it carries out to the letter.

If you put one letter wrong, it can't do that program. Is that stupidty? It is neither exactly stupidity nor its opposite, smart,--for a machine isn't ever smart. We are not trying to make it smart--at least, not if we have a little education and know a little bit about what's what in life.

So, I'm not saying that these are the opinions of every one who has ever worked with computers. Some people try to define words in a different way, and argue that some machines, or programs, somehow can be smart or intelligent --and I know their arguments fairly well, I think, and I also think I understand their arguments, perhaps even better than they do. And I think they are just plainly wrong about it, but it's something that requires lots of time to think clearly about. Years, even. So I mention this in order for you to be aware that I am talking you through a view of Personal Computers that I think is right and scientific and true but I'm not claiming that this is, or always has been, the universal opinion.

Right? So I give you my take on the computer, and I think you can go a long, long way with it, and gets lots of fun with computers.

Let's get going, then. Last time I think we got the PC to display your name ten times. We typed some stuff into the terminal and when we rebooted the PC--if you rebooted it afterwards--it would have 'forgotten' all about it. Let's this time do something that stays in cards even if we reboot the PC, turn it off and on again.

So, perhaps you start up the PC just like last time, all the way into the terminal, into the TF terminal. Once you have done it, you see that bright green square and so on. It's a good way to check that you have control over the machine.

So let's do that.

Then, instead of typing in anything at once to the terminal, we'll leave it, store something to a card or two, then get back into the terminal and tell it to 'have a look' at these cards.

Now, when you read this, you may notice that when I said 'forgot' just above, and 'have a look at' here, I put some quote signs (' or ", single or double) around these phrases. Why do you think that is?

Because otherwise we might start talking about machines as if they were humans and alive too easily. It's nice to be able to say, oh, the machine 'forgot' the program. That has a nice easy tone about it. But to be precise, we can say, that program wasn't stored on the harddisk. Such precise talk is a little boring sometimes, especially if we do it all the time. So to loosen up and be a little playful about it, we use the quotes, and when we use quotes around a phrase, we remind us that we aren't quite saying it precisely--such as when we talk of letting the machine 'have a look at' something. Even if we put a camera to the PC and makes a program go through the camera input, it doesn't mean that the PC is looking at anything at all. It is never 'looking'. It is perhaps matching patterns against patterns.

So did you quit the terminal, after you started it up? This you probably did this way. You typed

```
       qu
```

and pressed lineshift. Right?

Next press a key--when we write something like <CTR> or <PGUP> or <HOME> we usually mean, there is a button on the keyboard with such letters on it, find that key and press it. So now we say, press

```
       <HOME>
```

And that gets you to a card that has the menu for G15 PMN. It is, of course, card 15 in disk G, which we can also write as G15 or G:15 or, with lowercase letters, g15.
  Inside texts like this, it is a good thing to write it with that : colon inside, because then we'll quickly know what type of thing it is. I'll try to remember to write such as g:15 and i:1 in this book, even though when you are typing it into the PC, then almost always you must write it without colon. But the text gets more readable this way. In some places where we have long listings, we might add something around these, like <i:1>.
  Anyhow!

We are next going to put some stuff to card i:1 and i:2.
  I hope that's okay with your PC? Or has somebody else, or you earlier on, stored something on i:1 & i:2? If so, I'm sure you'll find out how to put it on another place; but if you have a 'clean' machine, set up just to learn G15 PMN programming, with the TF terminal app put into it, then the i:1 and i:2 cards are usually ready for anything you like them to have. As are i:1000 and i:1000000 for that matter. And so also k:1, k:2, l:1 and l:2. Just to mention some of the many cards you can use for various things from disk C to disk L.
  There are disks A and B also, but they are not available except when the PC starts up and fetches its startup

stuff. So C to L are available, and disk i, j, k, l are very often used during programming and writing, and often you find that a finished program is put to disk f, right after the Terminal, which often may occupy a couple of thousand cards on disk f.

Not that you have to remember all this.

So let's go and have a look at i:1. Press

```
       <CTR>-L
```

that is to say, press <CTR> button and, while holding it in, give the L button a quick push and then let go of the hold on the <CTR> button.

Then the PC should ask you about which card, or "card-id" you wish to see. "Card-id" is sometimes written just "CID", by the way, if you see those letters somewhere --you probably will, sooner or later, when you learn G15 PMN programming. Card-id means a letter and then a number, and it should be typed in with lowercase letters (even if it shows quite like capital letters in robotfont; remember that robotfont puts a dot above and to the right of a letter it considers to be a capital or big letter).

So you type into the PC, usually without colon as said,

```
       i1
```

and press lineshift. Now the screen should shift and at the lower left part of it, it says i:1 or so, doesn't it? Right before that it says MENU or EDIT mode. That has to do with the <CTR>-W stuff.

Now it may be that the i:1 card is perfectly blank, but suppose it isn't--or suppose you put something there now, and you want to put something different there the next time, so you should want to know of a really quick way to cleanse--not just card i:1, but also several more, i:2, and so on, let's say up to i:15. Right? So this is a

===>USEFUL THING TO KNOW: CLEANSE CARDS FAST
To cleanse a number of cards that you wish to program on. Here's how to cleanse 15 cards starting with i:1, but you can use it to cleanse any quantity of cards, anywhere.

Click

```
       <HOME>
```

key and get to the main g:15 menu card. If it says {EDIT} at the lower left part of it, click

```
       <CTR>-W
```

so it changes to {MENU}. Then click on the :colon where it says D:10. You get up a different menu. (The colon is used to jump quickly between cards. The arrow- or flower-like /

type of thing is used to start programs.)

Then find the place it says E:5 and click on that colon.

Then find the place where it says E/5000 and click on the arrow or flower-like thingy between E and 5000.

You get up a cleanser program. Be careful to type right next, or you may remove more stuff from the PC than you want to (so that the PC has to be 'reinstalled' with G15 PMN). All right? So, with attention, type

```
       15
```

for that's the number of cards. Press lineshift.

```
       i
```

for that's the disk. Press lineshift. Type

```
       1
```

for that's the start card on that disk. Press lineshift.

It should then say DONE! Press lineshift and you can press

```
       <HOME>
```

to get back to the main G15 menu card.

So such "Useful things to know" are meant to stand out in the chapters where they occur, so you can fairly easily find them by leafing back and forth in the book. That's why there is an arrow in front of the words and the words are all in capital letter. Sometimes there will be even more than one such "Useful things" in a chapter.

To cleanse cards doesn't mean that they become totally empty. The cards are there, and there has gotta be something on them. Even a blank, a " " type of thing, is something rather than nothing, right? You type it in and although a blank isn't shown as anything, it is still a blank. And yet, when we cleanse cards the way we just did we don't put blanks into them, but something slightly more mysterious, which is called a "nil". Think of it as more or less the same as "null", but even less of a thing than null or zero. A nil (sometimes called a "nil character" or a "nilchar") is a highly useful thing. When you write in a program on cards, it may take more than one card. But how is the PC to know that you don't want it to 'look at' more than just a handful of cards, or however many are in your program? The answer is that it stops 'looking' for cards when it comes across this guy "nil".

So let's see. Let's go to the card we cleansed and see what it looks like. Press

```
       <CTR>-L
```

and type, without any colon, and in lowercase as usual,

```
       i1
```

and press lineshift.

So you see these funny signs--not quite letters, not quite numbers, more like a drawing of a square, or a piece of paper, or perhaps a folder with papers.

Now we are going to get these nil-chaps, these nil-chars completely away and fill up that card i:1 just with blanks In that way, we can begin to put our program there, the program that displays your name ten times.

So, do a

    [RIGHTCLICK]

on the mouse. When we want the mouse to be used, we can put the [ ] signs around, but when we want a particular key like <HOME> on the keyboard, we use the < > signs around, okay?

The [RIGHTCLICK], that is to say, the click on the right side of the mouse, allows you to change the card--also called EDIT the card. So the word EDIT appears on the screen somewhere.

Now you can erase all of the screen quickly. First use

    <ARROWS>

to get to the upper left corner, if that half-dark line, the text marker, isn't already on the upper left corner. (When we say <ARROWS> we mean any one of the four slim arrows that are together on the keyboard, often to the right of it--these we can also call <ARROW-UP>, <ARROW-RIGHT>, <ARROW-BENEATH> and <ARROW-LEFT>.)

Then press

    <TABL>

also called "tabulator" or "indent" key, the key on the left on the keyboard--usually to the left of "Q", where it says eg TAB or TABL or -->| or something like that. I like to call it just <TABL>.

When you press it, you'll see that it clears away a handful or two of those peculiar nil-chappies. Press it several times. Then press

    <ENTER>

or what we call lineshift (I will sometimes call it <ENTER>).

You see that <ENTER> is really practical here, it has the same effect as <ARROW-BENEATH> followed by many <LEFT-ARROW>. Continue to use <TABL> to cleanse that line. See how fast you can cleanse every bit of the card!

Two other keys it is very handy to know about when you do anything with these cards.

The first is <END>, which moves to the end of the line. Try it.

The second is the combination <CTR>-R, which moves to the right column--about the middle, perhaps a touch to the right of the middle. This is used all the time when we program because we have room for both a left column and a right column when we put our programs on the cards, since each line is quite slim.

What else? Oh, I almost forgot. We should now save the blank card. It's ready to be used. So, when you are satisfied that there are blanks everywhere on the card, press

```
       <CTR>-S
```

and type

```
       i1
```

and press lineshift. The card is now saved with blanks instead of the nil's. Try press

```
       <PGDN>
```

to go to the next card, card i:2. Then press

```
       <PGUP>
```

to go one 'page' or card up, that is to say, to the card just before i:2, namely the i:1 that we were working at. If you used <CTR>-S correctly, you should now see a perfectly blank card there (apart from the usual text at the bottom of it which tells which card it is and so on).

So now it's time to put our program there. I suppose learning is also about repetition, so I think that we should begin by putting exactly the same program as we made in chapter 1 on the card here, and add no fancy stuff at all. Then we can adjust it a little bit and see what more can be produced with little effort.

Just one little thing we'll add. Remember that Clear Entire screen command, those two letters CE? That's really handy to put in first. So here's what I suggest we'll do next. Press <PGDN> or <PGUP> to get back to i:1.

In case the card says MENU mode rather than EDIT mode, do a

```
       [RIGHTCLICK]
```

and in case the text marker line isn't at the beginning of the first line, press a lineshift then <ARROW-UP> to get it right at the beginning of the first line. Ready? Then let's type! With lineshift after each line:

```
       sayit=
       ce
       ll:10
       &yourname&
       lo.
```

Be sure it's all lowercase, even though, when you type it into the card, it looks more like LL and LO than ll and lo. If it is uppercase, these letters get a dot above them. That's how uppercase looks in the robotfont!

This time we call our new word 'sayit', if that's fine with you. The rule for choosing new such words is that it is short, more than two letters, lowercase, without funny signs, and that it isn't already been given a meaning in the TF terminal (we'll look into a way to check that fast). But you can call your new word abla or difta or progeny or what you want, as long as you remember what it is when you are going to start it up. Note that digits are okay in these little program names, when not in the first position. So hi3 is fine, 3hi isn't.

Be sure now to save the program card back to the i:1 position very beautifully. Here's how:

```
       <CTR-S>
```

then, as it asks where to save it, type, without colon,

```
       i1
```

and that's it. Ready to try the program? Then we must, the way we have set it up now (we'll look at ways to set it up for even faster starts), first tell TF where the program starts--at i:1--then we must write 'sayit' or whatever you called the program. Ready? Super. Then press

```
       <HOME>
```

and click

```
       <CTR>-W
```

and click on the :colon between H and 1 on the top there. That gets you to the h:1 menu, where I think is where you start the TF terminal, though we can set it up in many other ways. At the menu H:1, click on the F/1 thingy as you by now have done several times if you have done all what we have said so far in this book, and the terminal starts up with its little graphics squares and so on.

Now you have the terminal up, and we're now going to tell the terminal to go through the cards--or the card, in our case, since it's just one card. You must then find, on the keyboard, the ^ sign which is usually found by the combination <SHIFT>-6, since it is usually above the digit 6. If you press other things, just press the <== key, which we also call <BACKSPACE> or just <BSPACE>. Ready? Then type:

```
       ^i1
       cc
```

with lineshift after each line. If you have done it right, it will say a large YES! In case it says something else, just go back and have a look at the program and try again.

(Then you press lineshift, type qu and press lineshift, and do a <CTR>-L where you type i1 and press lineshift. Check the program, go back to the terminal and type it carefully once more. You'll get it right! And keep spirits up no matter what the PC says. It forgives instantly, and forgets all mistakes completely and forever.)

If you called the program gringring or propanol or slartibart type it in, otherwise, if you called it just "sayit" as I suggested, type it in:

```
       sayit
```

and the screen should get completely blank and the name you love the most of all appears ten times in a nice font.

Restart the PC completely and go straight into the terminal and type the ^i1 and the cc again, and when you type 'sayit' or whatever you called it, it does it again.

And unless this is a PC that somebody cleanses afterwards, it will work tomorrow also: for cards don't require electricity to keep on their data.

And in getting the PC to do this, you have learned loads of things--loads of them. It's time to congratulate yourself and be in a mood of celebration and take a break and feel the joy of the full command you are having over your personal computers, your G15 PMN computers.

Part A, chapter 3: CONTROLLING THE DOTS ON THE DISPLAY

Before each programming lesson--if 'lesson' is the right word for these chapters--let's open our minds a little bit and just relax and take in the broader picture.

Sometimes, when you program, you may feel that all the world is in order--because you find that the computer responds perfectly to what you type into it, and what you type into it reflects perfectly what you think. And maybe you get a kick out of that. A bit of self-confidence. And when you then leave the machine and go out and enjoy yourself with people, you may feel that you have a glow inside, a kind of 'carrier wave' or inner force. Yes, programming can give you this.

Now there is a really interesting type of study done by an austrian named Asberger a long time ago,--he found out, put very simply, that some people aren't much tuned into other people's intentions--what they want, you know, and why they move about the way they do, why they gesticulate the way they do--and that just these people may be great at doing something which requires great concentration to detail. That's a simplification and I blend in some later studies in my summary there.

Instead of--as maybe Mr. Asberger did--thinking of this as some kind of trouble, I would call it a personality mode, or a person mode,--just as we can switch between Edit mode and Menu mode with the cards. Sometimes you just ignore other people's intentions and at other times you are really tuned into other people's intentions--you seem to know what they want in a flash. So, the question I think is cool is this: what if somebody is a little bit stuck in one mode and find trouble switching to the other mode? You follow? And how can programming help?

First, let's imagine somebody who is a little bit stuck in not tuning to other people. That person will probably have an easy time concentrating on the computer. How can he, or she--let's say "she", it's more positive--how can she learn to understand other people's behaviour more, by programming?

Well, just think of what it is to program: you put in a program and if it is a bit long, chances are great that the PC will do something other than what you planned. And to fix it, you must look at the behaviour of the PC, go back to the program and see and look and understand and then you fix it. So the PC has a behaviour and you get to learn to think through what lies behind that behaviour. As you get good at programming, you get self-confidence, and that--as said--helps you to be great social, too. So that takes care of that. Admittedly, it's a long way from thinking about the behaviour--or activity, to be more correct in language--of a machine, and that of a human being with a living mind. But it can be seen as a beginning--not all things about why people do what they do is so complicated, right?

The other way, those who are great at understanding other people's behaviour--they catch their plans without having to have them explained, all that--they may however find difficulty concentrating. So when the PC is in front of them, demanding nothing, it can help produce calmness. And their understanding of other's motives, plans, goals, intentions, means that they are good at thinking through this kind of stuff--and then all they have to do to learn programming is to translate plans into a series of steps that fits what the PC can 'make sense' of. So he, or she, --again, let's say "she"--she will learn to program by admitting to herself that she's already thinking through quite complex things on behalf of other people. The PC is much more simple. And the value of being patient with the PC and pick up its language is the great sense of calmness and the self-confidence and relaxation it can produce, among other things.

Then, for both types of persons--those who are mostly social, and those who are more, shall we say, self-centered--work around a PC can very easily become social. It can become a bridge, to sit near the PC and help each other and sometimes show off to one another.

Anyway, let's begin on some work, to get on with the next step. Last time we actually got a program put to cards--some would say to 'file'--and we got the PC to 'have a look' at these cards and follow the instructions on them.

Some would call this 'to compile' cards. We got the PC to 'compile' the program. That's a way to put it. If you like, the two 'c's we used--remember we wrote, into the terminal, something like ^i1 and then 'cc' on the next line? We could maybe think of this as Compile Cards.

These two-letter, and sometimes one-letter, words or what we should call them--commands, maybe--are also called pre-defined words, or PD words. "Pre-" means 'already'. And "define" is what we do when we tell the meaning of a word. It isn't as though we cannot make new PD words. We can--but that's hardly necessary before we are doing really huge programming projects, like an entirely new type of game that nobody has thought about before. You can do huge amounts of programming without making new two-letter commands. But to make new words of three or more letters, that's what you have already begun doing.

Let's keep it freshly in our minds what we've just been through--when it comes to putting stuff to cards and getting the PC to 'have a look' at it. So could you get up card i:1 again? You start up the PC, mount (with mnt) the 3333333 thingy--if it is not already mounted in the way it is set up for you--and you press <CTR>-L and type in

```
       i1
```

Got it? And, to be sure it is in Edit mode--so we can change what's on the card--do a [RIGHTCLICK] on the mouse. (In case somebody else have been messing about with your cards, go back and do the cleanser thing that we explained in the previous chapter--'useful thing to know' was the heading inside that chapter, right?)

So let's take away what's on that card, use <ARROWS> and put the text marker line on the uppermost left position, and then press <TABL> key to blank out a lot of them at a time. (Remember that key? The -->| type of button on the left side of the keyboard.)

I have in mind that we first get some numbers out of the machine, then do something with those numbers to change them into bright little green stars on the screen. How about that? So let's have a look on this first--just numbers--let's call it 'ournums' or something:

```
       ournums=
       ll:15
       i1
       nn
       lo.
```

Now before you type this in, let's see how similar it is to our earlier program. It begins with a new word, which then has an equal-sign (=) in it. The last word finishes with a dot after it (.). The dot, when typed into the cards, is a whole little block of black on the screen, but when inside a book it's typically just a little spot. So you see the cards editor--also called CAR or CAR editor (because we drive the cards a little like we can drive a car)--is really friendly towards our programming. It will be very clear at once whether we have typed in, or forgotten about, such a thing as the final dot in each little program or, as we also call them, our 'functions'.

Anyway. More similarities? Yes, you can see that again we have the two letters ll in the first line after the new word, and lo on the last line. That means we're into repetition in this little program also. How many reptitions? 15 this time, not just 10.

Actually, last time we had a 'ce' to clear the screen on the first line, just before the ll: stuff. But it isn't really necessary here. Let's keep it all, this time, as simple as we can.

So we'll first get this program to produce the numbers 1 to 15. Then we'll calculate them up to become 10 to 150. Having done that, we'll produce some bright-green stars on the screen using these numbers--just a very simple line of very tiny dots, nothing fancy or varied this time (we'll come back to how we can vary it). We're gonna make a vertical line of bright star-like dots.

You see there is a two-letter word, or command there--that we haven't used before--nn. Just as both pp and b9 can be used to print tiny texts on the screen, so can nn be used to print any number. We'll soon look into ways of printing also long texts. By the way, we say "print" as a way to talk about putting something to the PC screen. Why we sometimes say print is, I suppose, because in the first days of computing there weren't any displays, just printers making a lot of noise putting some digits and letters to paper each time the computer was going to give some results. So 'to print' means the same as 'to show', when we speak compu-lingo (unless we are really talking about connecting a printer to the computer and using it).

Just before the nn, by the way, is a new little command which has nothing to do with cards and all to do with the ll: and lo type of looping or repeating something, and that's i1. This means, let's have the count, or what is also called the "index"--for the first repetition. (So if we have several on top of each other, we use i2, i3 and so on up to i9 and then, the tenth, ix. We'll come back to that.)

So the 'i1' on one line puts a number 'on the table' as it were--also called, 'on the stack'--and then the 'nn' picks it up again and shows it on the screen. We'll see enough examples of this soon, so you'll understand it fast enough, don't worry about it. It will become simple.

Alright. Do you mind typing this in? It would be nice if you can train your fingers to get used to this type of stuff more and more; that's part of programming--not just to think with the head and your heart, but let your fingers learn to dance on the keyboard.

For the moment, just type it in from the top left of the card. We'll look into ways of typing with blank lines and the use of the second column soon enough, so that it gets

easier to change the program.
  When you're satisfied, save the card as last time: Click

```
       <CTR-S>
```

and, with a lineshift after, type in

```
       i1
```

and then, to be sure, press

```
       <PGDN>
```

to have a look at i:2. That card should be empty. (If not, get it empty by the way we showed last time. There are other ways we can do that also, we'll come to that.) And it's almost certainly empty if it's full of these small squares with a sort of 'thumbs up' on its right side (some stuff can 'hide' inside such signs, though, and we'll come to a way to check that out).
  Then you could press

```
       <PGUP>
```

and have a look at i:1 again. Does it look fine? No extra blanks? No sudden uppercases? Letters where there should be letters--the small l (L, not the digit 1), for instance --before the colon : and before the digits 1 and 5, on the second line?
  When you're satisfied with how the program looks--the compu-lingua word for it--that is to say, the computer technical word for it--is "syntax"--a word used also in grammars when it comes to spellings and such--when you're satisified, then, go into the TF terminal and start it up. So, press

```
       <HOME>
```

and press

```
       <CTR>-W
```

to get to Menu mode. Click on the colon inside H:1 on top there. In that h:1 card, click on the line in F/1 there. That starts it up, as you know. In the terminal, type

```
       ^i1
       cc
```

How did it work? If you typed it perfectly and saved it, the computer will say a big bright YES! in upper left corner (something we didn't clear away with a 'ce'). Then you can type whatever we called our little program this time, ournums. So type:

```
       ournums
```

and press lineshift. What the screen should show, then, is

```
  1
  2
  3
```

and up to 15.

Does the program work? If not, you have the fortune of learning some program correction. For all I know the PC might have stopped completely because something was typed in differently. Suppose you left out the dot. Suppose you didn't put in the 'lo'. Stuff like that. Perhaps you have to type some rubbish and press lineshift many times to get any response from the PC again. Maybe you even have to push the power button (perhaps even hold it in for a good many seconds) in order to restart the PC. That's something you should do in excellent spirits and with good humour.

The golden rule of thumb of program correction is this:
THE MORE ABSURDLY WRONG THE PROGRAM BeHAVES, THEN, USUALLY, THE MORE EASY IT IS TO FIX IT.

Whatever is the case, you should at some stage be able to type qu or so, and get back to i:1 card and fix it up and get it to run. Have you done it? Then let's multiply the number by ten. For this we just put in the number '10' and the multiply command, which is--you can just about guess it from our earlier commands, pp and nn,--yes, it is 'mm'.

So, get back to card i:1 again if you don't mind, and rewrite it to this,--and we'll soon enough produce some star-like dots--or 'pixels' as they are called--on the screen from these numbers. Here we go:

```
       ournums=
       ll:15
       i1
       10
       mm
       nn
       lo.
```

Save it to i:1, start up the terminal again in its h:1 card, and do the ^i1 and the cc and type

```
      ournums
```

and now we get:

```
  10
  20
  30
```

and all the way up to 150. Note that you could write, if you like, a million instead of 10 there. And you could write ll:100 instead of ll:15, and the PC is equally happy. (I should say, 'happy', sometimes I forget to put in the quote-signs.) But take care once a number gets larger than twice a thousand million--that is to say, take care when numbers get bigger than two billion. That's about the size of what the PC 'likes' to handle. Up to 2,000,000,000 (usually without those commas although we can instruct it to put in those commas), and a little more, and the numbers can go into the minus region as well.

So though we begin learning programming with really small and neat numbers, it doesn't mean that we aren't also learning how to put the PC to hard and complex work with digits that are thousands or even millions of times bigger. When you learn to handle inputs from keyboard and mouse and output to the screen, and storage to cards and getting stuff from cards into a program again, and also sometimes connect to other machinery from a PC, then you can, by equally simple commands--though usually very many of them indeed--you can get a PC to do anything that a PC can do. These things are the nuts and bolts, the alphabet, we might say, that we need handle so as to build up our programming language skills.

There are many ways to program, and there are many ways also to make programming languages. A guy named Charles "Chuck" Moore worked with simple computers in the 1960s to control motors connected to such as telescopes to look at the stars. And he, more than anyone else I know of in computer history, pioneered the work on using a 'stack' of numbers for the simple commands of a programming language to 'speak together'. This enormously elegant way shines all the way back to the work he did in the 1960s, although many things had to be made better for a really good and useful and easy programming language to result form it in the long run. In the making of G15 PMN we have learned from many places, but the heart is with the impulse from Mr Moore.

Let's next remake the program so it puts dots on the screen. Now I'm typing this into the B9edit editor, and I have the benefit of just copying what we have already programmed earlier as 'somenums' and putting it next here, and modifying it. Have a look at this:

```
       ourstars=
       ll:50
       800
       i1
       10
       mm
       px
       lo.
```

So I changed the ll:15 to ll:50. That's exactly the same type of command, just 50 repetitions instead of 15. So we get some more of these star-like dots we want to produce.

Then there is the number 800. That can be any number you like from about 0 to about 1023, and it goes from the entire left of the screen on the G15 PMN machine to the entire right. 800 means much to the right but not at all on the edge of it. In fact, it's about where we usually type stuff into the terminal.

What else is new? The name, 'ourstars'. (Call it what you want, of course, as usual, as long as it is unused.)

And then the 'px'. Now the 'px' command means 'pixel'--in other words, a pixel should be put on the screen at the place where we want it. It expects that 'on the table', as it were, --or on the 'stack' of numbers, we have two numbers. The first tells the left-right position and the second--the one which we calculate by means of our '10' and 'mm'--the multiply by ten thingy--is the vertical position. Left-right numbers go from 0 to about 1023. The vertical numbers go from 0 to about 767. That's how a G15 PMN screen is defined. Once you know that, you can compose the screen as you like it. Some programming languages try to handle any type of screen but it is my experience that the stage of the 'dance' of the program must be known for our programs to be looking good and easy to think with and work with.

With 1024 times 768 small dots, you can get a full book-width of beautiful small letters and there is still room for margins. You can have two dozens of lines of text, with good spacing in between them. It is a size found to be great for lots of good work, that type of 1024x768 display. It can be enlarged so big posters can show on the wall, or made rather small to fit in robots. Wider displays tend to go together with a kind of work that flips between several tasks in a distracting way. So you can rest assured that 1024x768 is a great type of display. (The reason we say 1024 'times' 768 is that by multiplying the number of dots horisontally with the number of dots vertically we get the total number of dots on the screen--somewhere above 750,000.) Each of the dots can vary in 64 tones of brightness and that is more than enough to present beautiful photos with great detail. The bright spring green, or yellow-green, or computer-green, is ideal to emphasize an optimistic outlook, health and beauty, and studies have shown the eyes like that color more than most other colors when it comes to a working environment.

When we count from 0 up to 3, we get four numbers, right? 0, 1, 2, 3. So that's why 0 to 1023 means 1024 dots--or 'pixels' as we should get used to call them. A picture-bit--that's what a 'pixels' is about.

So also when we count from 0 up to 7, we get eight numbers: 0, 1, 2, 3, 4, 5, 6, 7 is eight because the zero counts as a number. So 0 to 767 pixels means 768 pixels in all. It is of great value to know what the maximum and the minimum number is when we are going to use a command or something in programming: then the machine won't protest.

So px expects two numbers. We give it 800--that means on the right side, about--and then we give it a number that varies from 10, via 20, 30 and so forth, up to 500. So both of these numbers make sense to give to px. Try the program!

First, you put it to card i:1 the way you know--get up the card by <CTR>-L and then type i1 and press lineshift. Press [RIGHTCLICK] on the mouse, wipe out what you want to whipe out with such as <TABL> key, and type it in. Then press <HOME> and <CTR>-W and get to card h:1 and click on the place you click there to start the terminal. In the terminal, type

```
       ^i1
       cc
```

and then type

```
       ourstars
```

with lineshift after each line. With luck, you get a row of bright little dots, vaguely like shining stars, right where you wrote 'ourstars' and underneath it, in that column.

Fix the program if it needs to get fixed. And now you have entered the world of graphics programming and you deserve to have that glowing feeling which comes from having set a goal and achieved it.

Part A, chapter 4: PUTTING YOUR THINKING TO 0'S AND 1'S

So how is it going with your programming? We could make a list of what we've been through now, but one of the really nice things about programming is that if you have some idea of what you're looking for, you can usually get the PC to look up some texts that will tell you just what you want. There are texts--some of them more readable than others--inside your PC in this very moment that tells something about every command and word in the G15 PMN language. A few of them--the earliest, smallest set of two-letter commands--are briefly explained when you click on the right side of the g:15 menu page, the d:70910 thing (press <PgDn> after it to see list, click again--it is a bit tough to read at first). More of them can be found in other ways, including by the use of a socalled 'scan' program inside the terminal--that we'll look into later.

Also, the things you need most often when you do G15 PMN programming come up very often, so why make lists? After all, there are far fewer core words in G15 PMN than there are in English and every one of them can be explained very clearly, whereas in English, words float over and into each other and meanings also shift with the occasion. So English is wonderfully suited for thinking about all life just as G15 PMN is wonderfully suited to concentrate on numbers and getting the PC to do what you want. And, by such concentration, you also sharpen your logic and that, in turn, can help your English. Good sharp logic is also a good thing to have as background for making humorous comments (a teacher of university logic that I used to spend much time with had a reputation as a considerable humorist).

I have in mind that we take up the work on making more varied type of 'stars', or dots on the screen, soon enough --so please keep it in mind, what we worked with in the previous chapter. But we'll wait a little bit. I thought it could be as well, in this chapter, that we do a little more work with the terminal and pick up a few things about how the PC likes to work with 1's and 0's and how it likes to stack numbers on top of each other so we can pick them from there again.

Have you heard the phrase 'boolean logic'? If not, you have heard it now. Never mind why it is called that way, it has to do with some ancient person who worked with that type of stuff long before computers came around. But what you'll work with in this chapter will teach you something which can be called just that--boolean logic.

When you take decisions in daily life, you sometimes use logic--as when you talk things over with others and then you decide what to do--and at other times, you just think things through, and then in addition, you make guesses, and dream up solutions in some way that is or isn't quite like talking things through,--sometimes it may be very right to do it this way--then we call it 'intuition'.

When you talk and reason, you use many forms of logic. "Logic" means, put simply, that we use language, we talk, so as to find things out and reason one thing after another.

The machine logic--the PC logic of 1's and 0's--is very simple and it is wonderful that you yourself are not that simple. So be careful not to be too enthusiastic about what we will study in this chapter! Don't fall in love with machine logic, but keep thinking like a human! Those who have fallen too much in love with machine logic we think of as a bit strange, we might call them 'nerds'--and so that's why I give these warnings, against getting too stuck on this way of thinking that we'll do in this chapter. Having said that, it's wonderful to be able to also do it that way, when it is right. If many people have to talk things over, it's a great thing if you can lift out some simple arguments and show how they belong together. Or if somebody claims that they have perfect logic, and they don't, you may be better able to spot it if you have learned to handle the 0'1 and 1's in the way we begin to study here, when we talk of boolean logic.

So, let's start up the terminal. If you don't remember how to do it, go back to the first chapter.

Then we'll do some thinking about 1's and 0's--which is a way in which the electronics, the machinery of the computer, can handle really easily. Think of '1' as a way to tell the PC 'yes' and '0' as another way of saying 'no'.

In real life, we often don't quite know whether some-thing is fully one way or the other way. So real life is often more 'analog'--more flowing, fluid, dancing. PCs are, in contrast, more 'digital'--a word which comes from an ancient latin word which refers to the toes of your feet, if I remember correctly. Today, when we say that something is digital, we typically mean that it has a lot to do with 0's and 1's, or with computers.

As a first little exercise, let's type in some 0's and some 1's to the computer, with lineshift after. For instance, you might type something like this:

```
       1
       0
       0
       1
       1
```

That's five numbers--five 'bits', we might say, for a 'bit' is another word for dealing with 0's and 1's. So we have five bits. Where did they go? Are they still there? Of course. And now let's get them out. Type

```
       nn
```

and you get the last bit, 1. Type

```
       nn
```

again, and as you press lineshift, you get the second to last bit, also a 1. Type a third

```
       nn
```

and you get a 0 on the screen. Then another 0. Then a 1.

So this teaches us something about the socalled 'stack', or the table, as it were, where we can imagine that the PC puts numbers just as we can put things or papers on top of each other. And that is that the last thingy is picked first. Some want to call this for a "LIFO" stack: it is a "last in, first out" type of stack. This is opposite to how it is in real life if a group of people are queueing up in front of a shop or cafe door about to open: then it is obviously so that the first who came along to the door is the first to get through--a bit like 'FIFO', "first in first out". Of course, numbers we type into the terminal are more like papers or flat things we put on top of each other, where it is convenient to pick the last thing first.

Let's try something more. Let us imagine that the first number we put to the stack is an answer to a question, such as whether it is sunny today or not; and the second number is answer to another, related question, as to whether it is raining today or not. For some days it could be both and just those days it could be a rainbow, right?

So let's type:

```
       rainbow=
       an.
```

Now this is a really short little program, just one line after the = line. We call the little program 'rainbow', but we really mean a longer type of thing--namely, 'does it make sense to look for a rainbow on such a day?'. All this long question is shortened into that word, 'rainbow'.

And then there is that lone word, 'an'. It sounds like 'and' but it is just two letters long, like so many of the other little commands or words have seen so far. Let's try it. Try to type in these four lines next:

```
       1
       1
       rainbow
       nn
```

The first '1' says: 'yes, it is sunny today'. The second says: 'yes, it is raining today'. And after the PC does the little program 'rainbow' and then 'nn' to show what's on top of the stack, out comes '1'.

In other words, it makes sense to look for a rainbow in such a case. Try now, if you have the patience,

```
       0
       1
       rainbow
       nn
```

and it says '0'. Try also,

```
       1
       0
       rainbow
       nn
```

and again it says '0'. And it will say '0' also if you try

```
       0
       0
       rainbow
       nn
```

So now you have learned something about what we can call, using big words, for the "Boolean AND"--which is just 'an' in G15 PMN--and that is that it gives 1 as result when both its inputs are 1. In all other cases it gives '0'.

The most important words, or 'functions', to use that fine name, in Boolean logic are AND, OR and NOT, and in G15 PMN they are 'an', 'or' and 'n?'. They are really very simple and as a group extremely handy, as we will see over and over again. So it's worth paying attention to. They are part of the motor of perhaps all programs there are, at least all programs of any size.

Let's continue with the idea that we give our programs two numbers, two bits, and the first is the answer as to whether it is sunny, and the second is the answer as to whether it rains. When the first answer is 'yes' and the second is 'no', it makes sense to consider whether to go to the beach, right? Let's say it does. And let's say we want a tiny little program to hold this type of logic. It isn't very useful on its own, but by grasping how the PC juggles these 0'1 and 1's around you are learning to steer it really well.

So let's put that 'n?' to use. It's really unusual that a second sign in a command is any other than a letter, but 'n?' is a very important little command. It tells a 1 to become a 0 and a 0 to become a 1. You can think of it as a question, ie, "is it no?". Yes, it is a no. No, it isn't a no. So it converts a yes to a no and a no to a yes. That's one way of putting it.

Then, let's type in this, and experiment a little:

```
beach=
n?
an.
```

You remember that we type in, first, is the sun shining?, second, is it raining? We want to go to the beach, maybe, if it is so that the sun is shining and it is not so that it is raining. Before we study how it works, let's try it:

```
1
0
beach
nn
```

Type in that, with lineshifts, and it says '1'. But that's precisely it: 1=it is sunny, 0=it isn't raining. So let's consider the beach! In every other case it gives 0. Just try such as:

```
0
0
beach
nn
```

and it gives a '0'. Or,

```
0
1
beach
nn
```

and it also gives a '0'. So how does the PC know which of the two numbers it is going to apply the 'n?' to? The answer is that it does it quite automatically with the most recent number. So since the first question is, is it sunny, and the second is, is it raining, the 'n?' will work on the second answer, because it is the last one--the most recent--it is the answer on the top of the stack.

But what if we had the answers in the opposite sequence? What if we first answered, 'is it raining', then answered, 'is it sunny?'. Could we make a function to handle that as well? Yes, we could. We just throw in the one-letter command 'w', which tells the PC that we want the opposite sequence of the two most recent numbers on the stack. It would be something like,

```
       beachday=
       w
       n?
       an.
```

Be sure, when you make different versions of a little program, to name them two different things,--unless you quit the terminal (the command 'qu' will do this) and restart it. The PC 'wants' one word for one program, not one word for two programs, in each run.

So, with beachday, we could answer the questions the other way around: first, is it raining, then, is it sunny:

```
       0
       1
       beachday
       nn
```

and it will say '1'. And so on.

We have covered some ground so far--we have got working with the Boolean AND and the Boolean NOT--the 'an' and the 'n?'. The third really important one is, as said, the Boolean OR--what is simply 'or'. Don't think of it as any complex at all, though the word 'Boolean' sounds perhaps a bit fancy. The reason I use the word 'Boolean' at all is chiefly to remind ourselves that our natural English language is so much much richer, also in how we use these little words. They may mean a whole lot of other things, and, unlike a programming language, the meanings shifts around with the circumstances--what we also call the 'context'. A programming language is very independent from circumstances--from 'context'--and that is both the reason it is great for concentrating and meditating and the reason we must be careful not to get, shall we say, 'intoxicated' with it or infatuated with it too much. In other words, don't take the song away from your everyday natural language just because you have learned to program. Rather, you know more: you are learning a supplement to natural language,--not something that is meant to eat away

the finer dance of English.
  So let's have a look at 'or', and then we have done our first excursion into Boolean logic, and also learned a lot more about how stacks work.
  When both the bits are '1', 'an' gives us a '1'. In all other cases it gives us '0'. So we might say, the 'an' is a really demanding little chap. Right?
  Then we have 'or', and it gives '1' far more often. It gives '1' when any one of the bits are '1'--including also both of them. Only when both bits given to 'or' is '0', then it gives '0'.
  So, to use another example of a pair of questions to which we provide a '1' or a '0' as answers, let's imagine that the first question is this: do you like meeting friends? Let the second question be: do you like meeting new people? So you have a pair of numbers, of bits here. And then we make a simplistic little program:

```
       gotocafe=
       or.
```

Let's test it. You like meeting friends? Eg, '1'. You like meeting new people? Eg, '1'. Then:

```
       1
       1
       gotocafe
       nn
```

and the PC will respond with '1'. So in this case, try going to a cafe. But it will also suggest this when we type

```
       0
       1
       gotocafe
       nn
```

or when we type

```
       1
       0
       gotocafe
       nn
```

The only time it won't produce a '1', or a 'yes', is when we give it two '0's as input. So that's all there is to the simple, but eminently useful, little commmand 'or'.
  Enough of machine logic for this time, and, as we begun by saying: remember that natural language is much, much richer when it comes to logic, and nevertheless it is greatly valuable to know such more shall we say 'mechanical' logic, or machine-like logic. Knowing both natural language reasoning, and Boolean logic, you know a

lot lot more than those who stick to just one of the things.

Part A, chapter 5: YOU'RE INTO GRAPHICS PROGRAMMING!

Have you ever thought of what you are 'made of', inside your mind? I mean, how would you summarize yourself--not your body, merely, but your mind? You can dream, think, etc, but what are your capabilities? You follow?

If you find it a bit complicated to answer such a question, you can relax: since the time of Aristoteles, who lived some hundred of years before the time of Christ--and before him, I suppose, also--and in the following thousands of years--nobody has ever totally agreed on any way to describe the mind.

Yet, if we think of dreams, let us remember, or put words on, some capabilities we have: to visualize, to see images when none are present before our eyes. Perhaps some do this more clearly than others but I'm sure that everybody does it.

And we can also dream up music, new sounds, voices, right?

And we can certainly think. And respond to rhythm of music. And, whether we like it or not, we respond also by various feelings and emotions, mild or strong, around in the body or mostly felt in the mind--or a feeling may show as a clear blush in the face. And all sorts of things like that.

We can also think before we act--imagine that you are about to open the door but you also have something in your hand and it suddenly occurs to you that it might slip when you shift the grip of it unless you are careful. So that's a reasoning that may take place very swiftly, within the second, and it is perhaps a reasoning where you don't have to use words--you rather 'think with the body'. And in dancing, you are 'thinking with the body'--especially when you train and build up patterns.

So thinking, sometimes, is a kind of action that you don't quite do yet, but you do it in the mind--and then you may carry it out or not, after having a feeling over how well it sits with you.

This is all enormously complicated and yet we are born with it. When we want a robot to do a little bit of what a human like you or me can do so naturally, we must put in a lot of information and even then the robot is likely to be awkward and do it clumsily or wrongly where a human being with a living mind may do it really nicely and fluidly. For we are not just machines, we have intuition as well--something very lively, which enables us to really see and really feel and really think, not just--as a robot would do--go through lists of rules all the time. A robot can be cleverly set up so that it seems to us that it may be 'learning' but it is not really: for there is nothing in between the zeroes and the ones in it. And so we don't train the robots,--we should rather use other words than 'train' or 'learn' about them--and a word is 'entrain'. It sounds more technical and more like the type of thing a machine can do. So the machine don't 'recognise' you or me even though it may say our names in one way or another: rather, it may have rules that produce a name, perhaps through a 'match' on texts or camera input or whatever. So we say 'match' not 'recognise' when it comes to robots. If we are good about language when it comes to robots, we can have much fun with them and also sometimes program them. But we won't think us as stupid as them nor will we think them as smart as us--ever. And that is important: that we respect ourselves as full human beings and that we --living people--have something about ourselves that no machine, no thing we make in our factories and workshops--can ever have. Perhaps it also has to do with the word 'infinite'. You know that word?

So, when something is 'infinite' it has no end. And so, perhaps, one of the reasons it is so complicated to describe our minds is that our minds are somehow not finite,--that we are somehow 'infinite'. Can't you feel it sometimes? Perhaps you're dancing or talking with people and suddenly your mind feels all over the place--and it is perhaps not an illusion. You feel as your nerves have no ends to them. That sense of 'being everwhere' is a kind of deep feeling and you can come more easily to it some times --when you are in harmony with yourself, and happy with who you are--than at other times. But it is awesomely important that you don't think of yourself as a machine. I'm not saying that machines can't act smarter than people are acting--sometimes--but that's just because of some lucky programming, not because the machines have anything smart or intelligent about them at all.

They have just programs. And we can pretend to be like programs, sometimes, but we are so much much more inside. Let's bring that into our feeling of ourselves when we work with programming. It must be in every programming book, in every class on programming. So that's what we call 'philosophy'--to think about the grand things. The word 'philo' refers to a word that at the time of this chap Aristoteles (ie, in ancient Greece) means 'love'. And then 'sophy' means wisdom, or truth, or understanding.

Alright, enough big words, let's do some small numbers. And by these small numbers we can make a program that produces both more varied dots on the screen than that which we have done earlier in the book, and some text as well, and do so without an entirely clear-cut pattern-- as if by chance or coincidence. For this we have the two letters 'af', which we may try and remember as 'A Free number'. Let's begin by a little experiment in the terminal, then we put a short program to the i:1 card-- short, yet longer than that which we have tried before-- and then we try it out. This will give you more command over the display, and not just its dots or 'pixels' but also how you can put texts all around it.

So, let's start up the terminal the way we usually do.

Have you done it? Then type, with lineshift after,

```
500
af
nn
```

And now, what comes on your screen is any number between 1 and the number you wrote on the line above af, namely 500. Try to type in those three lines a couple of times. Each time you get various numbers--23, 84, 123, whatever. (You can try to put a higher number than 500 to it, but if you go into the tens of thousands, there is another word you can use, that works well with really big numbers, and that word is rffg).

Are these numbers entirely coincidental? No, absolutely not! They are just a lot of funny calculation where one of the inputs to the calculation comes from the little clock inside the PC. This clock doesn't merely count seconds, but it counts tiny bits of seconds. And so, each time you start up the PC, there will be tiny little differences in timing, and these produce enormously different numbers when you use 'af' (or 'rffg'). So this is a 'free' number only in some ways--in other ways, it is bound up to calculations as all numbers a machine comes up with are.

In the last chapter, we saw how the PC really 'adores' 0's and 1's and how, by putting something of our thinking into bits of 0's and 1's, we can produce programs. I have in mind that we are going to compare some numbers in what we are going to type in as a program to card i:1 soon. So how do we compare numbers? I mean, how do we get the PC to check whether a number is greater than, or less than, another number? That may be handy when we get the PC to produce numbers with 'af'.

So, to check whether numbers are bigger than, or smaller than others, we have such as 'gt' and 'lt'. How about trying out this:

```
       5
       3
       gt
       nn
```

The PC will say '1', which we have learned to think of as the answer 'yes'. So, the way to read it is this: 'we have a five, we have a three, now is the first number greater than the second?' And the answer is '1', or, 'yes'. So, if we switch the sequence of the numbers, we will get '0':

```
       3
       5
       gt
       nn
```

For this time, the question is, 'is 3 greater than 5?' And since the answer is no, the PC produces '0'. But we can put 'lt' instead of 'gt', and check whether '3 is less than 5', and we will get a '1':

```
       3
       5
       lt
       nn
```

And we have some more of this sort. We can check whether two numbers are equal by 'eq':

```
       8
       8
       eq
       nn
```

And it will say '1' to tell that they are equal. We can also combine the 'eq' with 'lt' and then we get 'le'; we can combine the 'eq' with 'gt' and then we get 'ge'. So, for instance, is 9 'greater than or equal to' 8?

```
       9
       8
       ge
       nn
```

Here the PC says '1'. Enough of this. Let's get into the card i:1 and type in a little program.

In case your PC has been used for other things in that card area, would you mind going back to the chapter where we said 'useful thing to know', and wipe ten or twenty cards clean? (That was in chapter 2, higher up in this part, part A.)

So let's put the program to card i:1.

You remember how to do that? You press <CTR>-L and type in

```
       i1
```

and press lineshift. And do a [RIGHTCLICK] on the mouse if you have to to get it to Edit mode. Put blanks to it by many uses of <TABL> on the left on the keyboard, the key that perhaps has something like ==>| on it. Be sure to keep things lowercase. Do you know that if you happen to hit <SHIFT>-<TABL> you get uppercase all the time? Press <SHIFT>-<TABL> to switch off this uppercase again. Good thing to know in case you wonder why it may be suddenly all uppercases after you've clicked here and there.

After pressing <TABL> press lineshift and press many <TABL> again, and use <ARROWS> also, if need be, until you're satisfied it has spaces all over itself.

Now the next program has four cards. That's all it takes --merely one, two, three, four cards, to produce stars that look more like stars, for they vary in intensity; to show them so that the same pattern doesn't quite occur twice; and add to that the word 'people' spread around that region of stars, quite a few number of times--and also in a new way each time we run the program.

Notice how many sentences in English it took to describe that program. So we must put up with the fact that it has several parts to it.

But I promise I will explain each part very clearly and apart from one or two things, it builds directly on all we've been through so far. All you have to do is to apply it--and be willing to think a little bit about numbers, not difficult numbers--just simple numbers. Okay? Let's go --and I will explain one card at a time, as we type it in, before we run the program.

Here's the first card, which you can put to card i:1. Be sure to press <CTR>-R to get to the second column. I have put in extra many blanks in this textbook so there is no doubt in the world that the second column really is the second column. So this is the first. It makes a new word called 'freeplace'. It uses 'af', which makes a free number, and 'ad', which adds something to it:

```
       freeplace=
       100                   100
       500                   500
       af                    af
       ad                    ad.
```

So, amazingly, though the two columns are entirely the same--apart from the 'freeplace=' on top of the left one, they usually produce two different numbers. Now which numbers are that? They are supposed to be roughly between 100 and 600, both of them. So you see '100' is written first. Then '500', and the '500' goes to 'af', which makes a free number, anywhere between 1 and 500, varying each time we run it. Then 'ad' expects two numbers into itself and adds them up. The '100' is then added to the free number, so we get something like 101 up to 601 or there-abouts. You see, what we want is it to be well within the limits of the screen. Remember we talked about the screen having 768 dots vertically and 1024 dots horisontally? So we want to be very well within both of these numbers. In that way, we can do anything we like--we can put dots there, and even texts,--the word 'people'--which can go quite a bit to the right without going outside of the screen.

So that's why it is called 'freeplace'. The little word produces two numbers which is a kind of free place on the screen. Let's save the card when you have put it the way you think is right. (Whether you have some blank lines on top, in between, or after the lines don't matter at all; it's the sequence in each column that matters.)

So, to save it, you press <CTR>-S and type in

```
       i1
```

and press lineshift, right? When this is done, let's go to card i:2 and put spaces to it all with <TABL> and line-shift and such.

The next card is this, it's a short one, and it uses the 'af' also, and in addition the 'le'--'less than or equal to'. It goes like this:

```
       usually=
       3000
       af
       2990
       le.
```

So '3000' goes to 'af' and that makes a free number anywhere between 1 and 3000. Now usually that number is less than 2990, right? It's just very occasionally it is in the uppermost 10 places of 2991 to 3000. So we're going to use this little program inside our programs to make stars and put the word 'people' on the screen, so that while it usually does make stars, it doesn't that often produce the word 'people': otherwise there would be only the word 'people' all over the screen and we wouldn't get to see the nice little bright green pixel-stars.

So 'usually' will produce '1' to mean, 'yes, this free number is less than or equal to 2990' most of the time; but some of the time it will produce '0' to mean, 'no, this is one of the unusual cases where the free number is in fact greater than 2990 and smaller than 3000'. A handy little program or function or what we call it that we will use in the next card.

When you have typed in the i:2 card above, save it the same way, by <CTR>-S and you type

```
       i2
```

and press lineshift.

Ready for the next card? This is building up a program that has many small programs in it. 'Usually' is one such small program. We can call these small programs for under-programs or under-functions--or, shorter, sub-programs, or sub-functions. If you want a big word for these chappies, you can call them 'subroutines'--under-routines, in a way. (Since a 'program' can also be called a 'routine').

Ready for the next card, i:3? Here we put the previous little programs--or 'subroutines' if you like--to use. I've put in many blank lines here--I usually do that when I use a word that is called 'se'--it's a new word, and I'll explain it as soon as you've had a look on the little program named 'saypeople'. So this is i:3:

```
        saypeople=           &people&

        usually              freeplace

        se
        ex                    bx.
```

Now this card, as all cards with two columns, is something you read 'first left column, then right column'. That's the way to read them. This program is the part of the top program that we get to in next card. So we're soon finished! We're over half-done with it!

So, the program 'saypeople' has on its left column the word 'usually'. Most of the time--as we said about the word 'usually'--it will produce '1'. And most of the time we want this program to exit, so that it doesn't clutter up all the screen with the word 'people'. And 'ex' does actually exit the program. Therefore, 'se'--which you can perhaps remember as 'see' (but remember computers don't really see anything, they just match numbers)--this is a funny type of command, because it says: the next command, after it, will be done only when 'se' gets a '1' into itself. In case it gets a '0' into itself--and 'usually' does sometimes, but not often, produce a '0'--it will skip doing that which is the next command. And the next command is 'ex'--that is, exit this bit, this part. It doesn't mean exit the terminal or anything like that. It just means exit this little word in this particular run of it; and when we call this word many times, sometimes it won't do that exit.

So you see that 'se' is quite a grand type of command. That's why I typically put blank lines both before and after it--I really want it to stand out, because it is so influential. Be sure that the next thingy after 'se', whether it is on the same card or (if there are many blank lines) on the next card, is a command--two or more letters (not a number nor a &..& type of thing or anything such as another 'se'--the 'se' has been made so that we must keep it simple, and that has turned out to be great when we make big programs!).

Next column--which is then only rarely done--yet sometimes, if we call 'saypeople' very often--has in it the word &people&. The &..& we have used before, in the first chapters, when we got the PC to say your name, right? And here we want the PC to just say 'people'.

Where do we want it to say that text? Answer: any free place, so we say 'freeplace'--the word we first made just now, on card i:1. So the cards that are one after another are always so that they can use that which is earlier on, quite freely.

That last command is 'bx'. Remember 'b9', which we used to put your name very neatly in nice B9font on the screen? We promised we would come around to show a way to put texts, also long texts, anywhere on the screen. So 'bx' is this variation. It does the same as 'b9' but you must tell which dot horisontally and which dot vertically it is going to begin on. And 'freeplace' takes care of this--by giving it two numbers between 100 and 600, about.
Save the card, when done, to i:3, by the usual <CTR>-S and you type in i3 and press lineshift.

If I have jumped into too much here by coming with a program that is suddenly four cards long, then bear over with me: we'll get the program to perform whether we fully understand it or not the first times; then, as you work on with the book, you can come back to this program and it will suddenly be much more understandable. The first times, I know, it can be a bit overwhelming. Then, just trust that you're learning--and sometimes much more than what you might think. And what you pick up during day-time you work more on when you sleep, and so you are more ready to quickly understand it the next day.
The next card is very much like how we have been doing things before, when we got the PC to put a column of dots on the screen--only here we use 'p3' instead of 'px', so we can vary the intensity of brightness of the dot freely. This 'p3' chap expects three numbers--the horisontal dot, the vertical dot (ie, positions), and then any number between 0 and 255, where 0 is coal black and 255 is as bright as the computer screen can make its green.
Get ready to try it! Just type in the program 'starspeople' here, which calls on all the previous three cards, and does so without much complications:

```
        starspeople=        255
        ll:10000            af
                            p3
        saypeople




        freeplace           lo.
```

So, where you have the blank lines don't matter as long as you feel it is clear enough. The left column, then, says: our program 'starspeople' has a repetition 10,000 times--that's a lot of sweet stars. The 'll' (two lowercase L's) is matched with a 'lo' at the bottom right column (lowercase LO). So that's our loop to make stars and also sometimes, but only sometimes, show the word 'people'. And we simply write 'saypeople' to get the text in there. Don't have to think through that one again.

Then, we use 'freeplace', to get a place for the star; and on top of right column we have '255' and then 'af', which means that we get any number between 1 and 255 for intensity of the green. These three numbers go into 'p3' which, as said, produces a dot just there, and of that intensity. And that's it!

So save the card, when done, to i:4 by the <CTR-S> and that you type in i4 and press lineshift. Then get into the terminal--you know how, I think, or check with chapter 1 if your memory need refreshment--and there we type

```
        ^i1
        cc
```

If there are no 'complaints' by the PC, then the program was correctly typed in. And to get the effects, you can type

```
        starspeople
```

and get a field of 500x500 on the computer monitor full of star-like dots of varying intensity, and a bunch of times, all over that place and sometimes a little to the right of it with some letters, the word 'people'.

I say, you're into graphics programming!

Part A, chapter 6: YOUR MOUSE PRODUCES STARS

Once upon a time--so begin fairy tales, but also, sometimes, can a tale of history begin--it was thought, by a German thinker, that if only there was a perfect language of logic, then that would end all conflicts in the world. For if people had any conflicts anywhere, they could just sit down around a table and say, "Let's calculate!" And so, by using this perfect language of logic, they would work out what's fair and each would stick to it and there would be no conflicts of any sort.

Well, do we laugh of this idea? Let us laugh a little, at least. Conflicts cannot always be solved via calculation--although, perhaps, the thinker, the philosopher (his name was Gottfried Leibniz), had a certain point: when we think clearer, we can sometimes find ways to avoid trouble, and in thinking, it does help if we can compose our thoughts really nicely. And for that we do need language, and perhaps also sometimes a formal language--a language like a programming language--can be of some help. It can help produce clarity. At least for those who spend time with it.

Yet, the world is infinitely more complex than that which any human being in it can think out. Right? The world, especially the world of people, is just so enormously complex--and sometimes you must move really fast and with fantastic precision, as of a dancer, to capture a moment and swing it if something is about to take a turn that it shouldn't take. Then you must rely on judgements inside yourself that goes deeper than mere thinking. That's when we talk of intuition.

Somehow, and this is a theme I have also noticed with other people--those who know a bit of how to program computers are often more willing than most not only to sort out their own thoughts but also listen to deeper feelings and to intuitions. Clarity in one place helps clarity in another place. So, in the upcoming volumes--and I hope, like me, you are considering the Art of Thinking a journey, a journey where we can make discoveries--in the upcoming volumes we'll not only tackle programming that builds on what's in this volume, but also see if we can sort out a bit about how to tune our own intuition in all areas of life. For thinking is also about that--not just about computers, of course.

Shall we then do something as down to earth as to work a little with our cards and prepare for a next program? Up until now, when we have worked with cards, we have, if you have followed my suggestions exactly, used <CTR>-L to load cards. We'll now set it up so we can move between cards a

little faster.
  Then, we have started out making new programs each time. How about going in and 'hacking' into the program we made in the last chapter? This you can do really fast if the program is still on your PC, if you are using the same PC and it hasn't had its G15 PMN cards re-installed or any such thing.
  And one of the things we might want to do is to figure out a way we can combine mouse with something of what we did last time. So you can use your mouse to produce stars.

But first, it's time to put your mark on the program. Every program with respect for itself begins with some comments, which tells something of the name and usually also who has made it and perhaps also when.
  A comment, then, is the absolutely most humane part of a program. It begins with a bar or what we call it, the | type of sign, which is found on the keyboard perhaps to the left of Z when you also click <SHIFT> (the uppercase kind of <SHIFT>), or perhaps just to the lower left of the <ENTER> key (ie, the lineshift button), also requiring a <SHIFT>, usually.
  So let's do some work--I assume the cards you put into the PC at i:1, i:2, i:3 and i:4 in the previous chapter are still there. If not, put them in.
  And the first bit of work is to push all these cards one step to the right, so they become i:2 to i:5 and that we get i:1 free for the use of commentaries. Some of you may already know how to do this--it involves <CTR>-C. And, as promised, we'll soon enough set up something--a card--to point to the i:1 card so we don't have to do <CTR>-L each time we want to get to that card.
  But for now, if you are ready, do a <CTR>-L and type in

```
       i1
```

and then let's move these cards exactly one step to the right, or forward, of what we call it. This is how:

===>USEFUL THING TO KNOW: MOVE CARDS TO THE RIGHT/FORWARD
So this is a fast way to make room for one or more cards. This is most relaxing to do when there are no cards above the cards you're working on, that need to be in the same place. (If there are, you must watch out for how big number you type in next, and be sure it isn't too big.)
  So, number one, get to the card which is the first in a series, where you want this series to be moved forward. For this, you might use <CTR>-L for instance.

Do a [RIGHTCLICK] so the CAR editor is in Edit mode. Then, press <CTR>-C and type in

```
      33
```

or any number a little bigger than the amount of cards you are working with. (I usually type a number like 33, 333, 555 or 9999 or so, anything easy to type in and that's covering the amount of cards that are going to be moved.)

Be sure this number isn't so big it touches something that lies above the area you are working with, and that should be in place.

Next: if just want to insert room for one new card, press <PGDN>. To make room for more cards, navigate to the card above it e.g. by <CTR>-L.

Then press <CTR>-T. It will ask you whether you really want to copy To this place. When you are sure, press <SPACE> (ie, the long space bar underneath all the letters on the keyboard). If you are uncertain--such as if it mentions an amount that surprises you--press <ESC> instead and try again.

You can now go back, for instance with <PGUP>, to the card you started with. It will have a copy of itself with a higher cardnumber, so you can just wipe it clean with <TABL> and begin to use it.

***The same as we did just now, you can do in the reverse, e.g. by selecting PGUP instead of PGDN after you have done the <CTR>-C and before you do <CTR>-T. In that way, you have a method to remove cards also.

All right. The explanation was a little bit longer than it has to be, just so that has a form that you can look up later if you need to have memory refreshed about this, and use it to handle some other card-moving situations.

Have you done it? Great. Now let's put in some comments. Just remember to begin each line in each column that has some comments in them with a | and, so that we create clear good code and don't 'confuse' the PC, avoid using a dot (.) or ending a line with a &-sign (for . and & are used to signal things to the PC--the dot or . is used to signal that a particular program is finished; the & or 'ampersand' is used, as you know, when we wish to quote something so as to display it on the monitor or the like).

Here's what the comment card, i:1, could look like. Get to the i:1 card and be sure it is in Edit mode and type in your name where it says YOURNAME so you get it in:

```
      |studying           |A program
```

```
        |the mouse           |by
                             |YOURNAME
```

In case your name goes over more lines, be sure to put a | before the next line in the column. Remember that <CTR>-R brings you to the right column.
  When satisifed, save the new i:1 card by <CTR>-S as usual. Then, as you press <PGDN>, you should see that the program starts nicely on i:2 and continues.

We should check that the comments have been put in right, in the sense that the PC doesn't 'protest'. But can you wait a moment before you check it? Then we can set up a quick way to get to i:1 and to get to the startup of the terminal, which normally starts at card f1 (at least when you mount the app as I indicated in chapter 1 in this part, part A).
  So here's a way that is highly practical:
  Press <HOME> then <PGUP>. That gets you to card g:14. Why can't we set this up with a couple of things we can just click at when we need to? (Another card, equally easy to get at, is g:16, by <HOME> then <PGDN>.)
  So either it is plenty of space already on your g:14 card, or you can make it that way: do a [RIGHTCLICK] then use <ARROWS> and <TABL> and <ENTER> to put in blanks on the card where you want it. Then, of course, type in the first card of the program--and more, if you like to. Perhaps with an explanation, like this:

```
        Programming now: i:1 i:5
```

And let's add another line also:

```
        To start the terminal: f/1
```

and here you don't put in a slash / but rather the type of flower or slash-like sign that on a G15 PMN you usually find by a <SHIFT>-5 press (ie, the <SHIFT> button for uppercase, combined with the digit 5). (I should say that this sign rather looked like 0/0 before we redrew it this way, and is the only strong change of the G15 PMN keyboard compared to how it has typically been done on other key-boards; the name of that was 'percent', but we don't need to have such a sign for that word, which easily can be overused in any case--'permille' is a more important word, I would suggest--both of these can be shortened easily. But, then, as G15 PMN programmer, you can put any shape of any sign you like--there are apps for character drawings.)
  So why don't you save the g:14 card now? Press <CTR>-S and type in

```
        g14
```

and then you can try out how it works. We call this, "to set up links". There's a link to the program we're working on, and a link that starts the terminal.

If you like, have a look at the normal place you start the terminal--I suppose that's h:1--and see that you have written the f/1 thingy in exactly the same way as there. Then press <HOME> and then <PGUP> and you're back to your link card.

Now do a <CTR>-W to enable the mouse, and try them. We can try the i:1 link first. Click on the colon : there. It should jump right to i:1 card. Then click on <HOME> and then <PGUP> and you can try to start the terminal from there, by a click between the 'f' and the '1' there. Did it work? If not, just go back and have a look at how the h:1 card does it, or wherever you start the terminal from, and make sure it is equal, and a blank before and after.

Does it work now? Excellent! Then, inside the terminal, let's check that our program still works, with the new comments, and with all cards moved one step up. Type

```
        ^i1
        cc
        starspeople
```

and the program should start--and perform just as it did in the last chapter (in the next part, we'll see how a program can be set up to start more and more directly, with less typing needed at the terminal).

When all this works, type

```
        qu
```

and we are ready to change the program so that we can try out the mouse--just very simply--not bothering about whether it is clicked or not, but so that we get a tiny 'starfield' just where we move the mouse when the program is running. We also need a way to keep the program to run for a while, then, and a way to get it to exit when we click on the lineshift button or something.

So we're going to put in a little bit stuff, and learn some new commands. We'll learn that 'su' is substract, just as 'ad' is add; we'll learn that when we want to find out whether keyboard is touched when something else goes on, we can use 'ck'--Check Keyboard; we'll learn that 'cm' is one way to Call on Mouse; and we'll see that it is possible to get a loop to go on for as long as we please, until we press lineshift, simply by putting in the mysterious little command 'q1' at the right place. We'll also learn that 'sh' is a command to take away unnecessary numbers on the stack (and one of the commands, 'cm', puts some more numbers on the stack than we need, so 'sh' will come very handy to 'shuck' away anything unnecessary.

With a little patience, then, we'll rework the program from the previous chapter to do something a bit unusual with the mouse. We'll even learn a new, longer command, or function, or what we call these things--called 'activepause'. This we are going to use because one PC may be frightfully fast and another may be slow and when we make a program drawing things on the screen by the mouse, this is a way to make the PC do roughly the same kind of thing no matter how fast the PC works in the background.

These are many new things but it is not that much typing, and, as pay-off, the program can be played with more than any program we've made before in this book.

So roll up your sleeves, stretch your fingers, and let's start to change the cards!

And let me say it clearly what I have indicated in the beginning: programming G15 PMN is fun to learn not only because you get results quickly, by typing in just a little, but also because it's easy to change a program a and see what happens then. Just be ready, when you change things you aren't certain about, to restart the PC! And, as a good rule of thumb, restart it often anyway, when you do hefty programming--and hefty programming is exactly what you are doing.

This is the new i:2 card. It has 'sh' in it and that 'shucks' away--deletes, removes numbers we don't need. And the word 'cm' tells us a lot about the mouse--not only the position, but a 1 or 0 as to whether the left button is clicked, and a 1 or 0 as to whether the right button is clicked. So there are two 'sh's after the 'cm':

```
        freeplace=          w
        cm
        sh                  40
        sh                  af
        40                  su
        af
        ad                  w.
```

Now that you are such a pro you will get this right without me having to spell out that you use <ARROWS> and <CTR>-S and so on. So each card that should be changed I will simply mention here, and if you want to try this little gem of a mouse program, be sure to get the program cards stored right.

You see that the number before 'af' is much smaller--so there will be stars just around where we move the mouse. And as the mouse moves around, the word 'cm' tells us where,--it leaves the horisontal position and then the vertical position on the stack. If you look this up in some overview, it will say 'x' and 'y'. This is the usual quick way to talk about position horisontally and position vertically. A way to remember it is that when you look at the letter 'X' and the letter 'Y', then 'Y' has something more vertical about it! And a way to remember what sequence they come in is that, of the two, x is first in the alphabet, then y comes along. (Incidentally, have you noticed that 'z' in the cards has its line falling like backslash \ rather than as slash /? This is because we want to use 'z' in programming to indicate that something is finished. So 'z' in robotfont is kind of the mirror image of 'z' in b9font.)

Our friend, the single-letter command 'w', we have seen before--remember when we were doing the '0' and '1' type of thinking--about rain, sun, rainbow, beach, cafe and so on? Then we used 'w' to 'switch around' something. Here we are doing it so that we get to add something to the mouse position. The mouse position is given as the two numbers, x and y. So there is a w and that means that the sequence becomes y and x. So we can add something, then, to that which is now top of stack--x is top of stack. Having done that, we do another w and that switches it back again.

So, by a little bit of thinking--draw up the stack of x and y and how this routine makes a free number by 'af', adds it, uses 'w' to switch, makes another free number, adds it, uses 'w' to switch back. Now why does it matter why it switched back, so we have x and y instead of y and x? This you can, if you wish, try afterwards--when you get the program to run well. Just drop the last 'w', put blanks on that line, and suddenly everything you try to draw gets some other place on the screen than where the mouse it.

The i:3 card has 'usually' on it, we don't need to change that. This is the new i:4 card--exactly as before, only that we must have a shorter word to give the mouse freedom to move also towards the edges. Let's make it say 'hi' instead of 'people':

```
        saypeople=          &hi&

        usually             freeplace
        se



        ex                  bx.
```

The i:5 card has just a tiny change--it's a much smaller loop, it says ll:50 now, because we want the mouse to sprinkle stars just very near around itself, and so it shouldn't be so many. So here's the new i:5:

```
        starspeople=        255
        ll:50               af
                            p3
        saypeople




        freeplace           lo.
```

We could call it 'starshi' and saypeople for 'sayhi' or the like, but we do it the easy way. The point isn't what we call the function, but what it does--as long as we have a clear idea about it all, any good-sounding name will do.

Then, we add a new card, i:6. Here we go:

```
        mousefun=           ck

        ll:1                se
        q1
        60                  ex
        activepause

        starspeople         lo.
```

Okay, this is really quick to explain if you look at it, even though it has several new things. First, the loop is amazingly just ll:1, not ll:2 or ll:3 or ll:10000. How can that work out? Just one? But then, at the next line, is the mysterious 'q1'. It does a trick--we might say--but it is also perfectly in order. It changes the counter by substracting one from it. So the counter will be 0, after q1 has done its work. This means that the loop will next increase the value to 1, and again q1 makes it 0, so it goes back and forth between 0 and 1. It doesn't exit unless it has something like 'ex' in it--and, yes, it does have an 'ex' in it. Remember this little command? It quits a little program--not quitting the terminal--just the little program that is running. And see that 'se' before it? That 'se'--which reminds us of 'See', to look (although the PC doesn't really 'look' as you know)--it does the next line when, and only when, it gets a '1' to itself. And so, before the 'se' is a 'ck'--Check Keyboard --so you can press such as a lineshift and the program will exit!

The one new thing to explain, then, is 'activepause', with the number 60 before it. Why 60? Just try different, higher numbers, if you like--500 is half a second, 50 is a tenth of half a second--1000 is one second. 60 seems to be okay if you want to create a burst of stars where you move the mouse on a pretty broad area ('cm' is one way to read the mouse--it works on the Curveart area, if you have tried the ART command; you can try 'md' for a somewhat larger area if you like, instead of 'cm'). The higher the number the more it will pause before it bothers to look at where the mouse is and make stars there. So there will be fewer stars the higher the number is, as given to this 'activepause' (if you want to look up more about this word, most of the longer-than-two-letters word in the TF terminal are at least mentioned somewhere in the text that comes along with the 3rd Foundation app 3,333,333).

Now, before we start it, notice that the new card calls many many times on the previous program we made--only that we have changed the previous program so it responds to your mouse. And notice how easy it is--just to mention the name of the previous program, and, as long as the program is there--in the cards before it--it all works out. As we'll see.

Ready to test it? Into terminal, then

```
        ^i1
        cc
        mousefun
```

And when the cards are exactly as above (hopefully I have typed correctly here and everywhere when it is the program cards themselves, but be forgiving if I haven't!)--you will find this amazing thing: until you press <ENTER>, the program will make a 'burst' of stars--bright dots of varying intensity, different each time--just where the mouse is--and sometimes it will write 'hi' there!

Go on and change the program--one thing at a time--and change it back, then change another. Restart the G15 PMN each time through a 'reb' command, just in case you do something really wild that has effects that must be 'cleansed' by the 'reb' action. Poke into the program and see how things belong together. You might find ways of making it even more fascinating by this or that tiny or not-so-tiny change.

In the next part, part B, we'll have some chapters where we will slowly build on what we have worked with in this part, so that we get familiar with even more commands and and more capabilities in making varied programs.

PART B: BECOMING FRIENDS WITH MANY NUMBERS

Part B, chapter 1: THE ART OF ARITHMETIC

Perhaps now you feel that you have a sense of what it is to go in and out of the terminal for G15 PMN; how it is to put a program to cards and run it and then change it; and how G15 PMN 'likes' to put numbers in a stack--and then 'pops' them off again. You may or may not be one who likes to think about numbers, but G15 PMN is gentle if you don't like it too much--and will gently increase your ease with which you handle numbers, as you work with more and more programs. It isn't too much to say that G15 PMN could be just about all you need to learn to handle numbers at any level you like--and in that way, it is an education that may have a core value for all. For instance, we can learn to think about curves--the socalled 'sine' and 'cosine' curves, for instance--and other things like that, just as we have already handled a bit of what we called 'Boolean Logic', reasoning by means of 1's and 0's and such.

To set aside some time--perhaps quite often for some-- to work with growing programming competence, and using it, and also make and change programs, is a work with that part of your mind that thinks about large and fine orders, --great structures, and details as well. There is a calm-ness in doing that. You build a sense of beauty inside. The more beautiful program it is, in your eyes, to more easily you will also spot anything in it that has to be changed to make the program do what you want it to do. Beauty makes it easy to, as it were, 'heal' the program.

And this beauty you build inside, you can bring out to the individuals you meet as a smile within your body, a sense of something--order, gratitude, that things are good--which other people sometimes, but not always, bring to you spontaneously. So it isn't too much to say that, in order for you to have a more steady experience of harmony, and to be more robust--if that's the word I want--in your generosity and radiance--it helps greatly to have a bit of discipline in a study of something like G15 PMN. Add to this more artistic disciplines like drawing and dance, and you'll find that all your body can radiate more of its best capacities.

So let's look a bit into numbers, and how easy it gets when we type our commands into the terminal. Start up the PC and get into the terminal, as explained in chapter 1 of Part A, and let's explore a few things.

Try to type

```
        f
        nn
```

and see what happens. Out comes '123456', doesn't it? This is how the Third Foundation terminal tells you that the stack is in its starting-point. So if you run a program, and your program puts things on the stack and takes the same number of things away from the stack before it exits, you can type

```
        f
        nn
```

to get some kind of test that your program behaved as your thought. If your program took more off the stack than it put to it, or left some extra on it, normally there would be another number here. (What does 'f' do? It makes, or, if you like, 'forges', a copy of what's on top of the stack. So you can do 'f' and 'nn' many times over and it will not change the stack; but 'nn' alone takes one number off the stack.)

In part A we saw that 'ad' does addition of numbers, and 'su' substracts numbers. When you use 'su', it matters which way you put numbers to it. The rule of thumb is that su is 'the first number minus the second number'. For instance,

```
        8
        5
        su
        nn
```

gives '3', but

```
        5
        8
        su
        nn
```

gives '!3', which is a way in which G15 PMN can tell you that a number is negative. Another way, which is, for instance, used in the G15 PMN Spreadsheet app, is a dash, a '-' type of sign. But when you program, it is of great value that something as enormously different as a positive and a negative number is shown by something that stands out--even when you do many things and perhaps are a little bit exhausted. So often you'll find that G15 PMN invites you to type in any signed number with an exclamation mark, a '!' sign. For instance,

```
        !5
        8
        ad
        nn
```

gives 3. Turning to 'mm', which is 'one number times another number', it is as 'ad' in that the sequence doesn't matter:

```
        3
        5
        mm
        nn
```

gives '15' as does

```
        5
        3
        mm
        nn
```

But when we divide one number on the other, again (as with 'su'), the sequence matters. The command 'di' works this way, as a rule of thumb: 'the first number divided on the second number'. So, for instance,

```
        15
        5
        di
        nn
```

gives 3, but if we switch the sequence of 15 and 5 we get just 0. For 'di' handles whole numbers. We can go a long long way with programming without bothering about decimal dots; but when we need it, we can put it in, in one way or another. For not too-long numbers, one of the places we find functions to show, add, divide etc such numbers is in the G15 PMN "Spreadsheet" app. For long numbers, these functions can use, instead of our normal numbers, a type of extra long number called "60-bit number", which is also in the G15 PMN Open Robotics app. These longer numbers are built up by our normal numbers: and by looking into these functions, and thinking a bit, you can double the length, and get up to 120-bit numbers. (But you already have 15 or 16 digits with 60-bit numbers, so I doubt you'll often need anything more.)

I mention this to show that G15 PMN, while it starts with whole numbers up to two billion, it doesn't mean that it can't handle decimal numbers and even bigger numbers.

Yet how important are the decimal dotted numbers anyway? Let's give it a think. Very often, numbers with a "decimal dot" in them are simply a way of writing whole numbers. In some cases, such as when it comes to converting money, the 'small digits' matter and decimal dots become really handy --and that's the sort of thing the Spreadsheet app is made for, so it makes sense that this has decimal dots in it.

Take two numbers, say, 55 and 34, for instance. If we put it to the PC as they stand, the PC will say that when one is divided on the other, it is 1. Now let's work on this, and get more insight into it. First, let's try it in the terminal:

```
        55
        34
        di
        nn
```

And here we get 1. One of the rules of arithmetic with addition, substraction, multiplication and division is, as I believe you already know, that when the numbers divide ' neatly--like 15 and 3--then you can multiply the result, in that case 5, with the second number, 3, to get back the first, in this case 15. Right? So since 15 divided at 3 is 5, then 5 multiplied by 3 is 15. That's when the numbers divide neatly. When they don't, there is a way to talk about how much 'off' the result is, and that's given a long name--'remainder'--or a difficult name, 'modulus'-- and why it has these names I don't know. It could be called 'apple' or 'orange'. In G15 PMN we just call it 'mo'. So let's try. The 'mo' of 15 and 3 ought to be 0, since they fit so neatly, but the 'mo' of 55 and 34 ought to be quite something other than zero. So 'mo' will tell us how much 'off' the result is, if you multiply it back. So it ought to be something like 21, because that's how much 'off' the answer is, if we 'calculate it back'. (Do you think this is a boring number lesson? Cheer up! We're soon done, and even if you don't normally have much use for any such arithmetic, then suddenly--perhaps you are figuring out a clever way to calculate scores or something for a game program--then it may be good to know that there's a chapter explaining these little things.)

So, when we do

```
        55
        34
        mo
        nn
```

the PC tells us '21'. Just one more thing about 'mo', and that's a thing that sometimes interesting when you want to play with very free numbers. And that is this: type any number--up to a billion--and then any smaller number, above 0 and 1, and then 'mo', and what you get is, for sure, a number that's between 0 and that number you gave. It will never be as big as the number you gave (and never bigger). And so you have a way to, as it were, 'shape', free numbers really fast. Try, for instance,

```
        123999555
        17
        mo
```

and the PC will respond with something or other between 0 and 16. I have no idea beyond that what the answer is. So I'll go into the terminal and check and type it in here. The terminal said: '8'. All right. Now you try any other number up to about 999,999,999 and then '17' and then 'mo', and you'll see that the number is 'converted' into something with such a pleasant range.

So now you have a way of remembering 'mo': 'mo' is short for 'MOdest'!

Now, then, let's look into what 55 divided at 34 is, in case we want to see more of the result. Try,

```
        55000
        34
        di
        nn
```

and we get '1617'. So, if we did it with decimal numbers, we could have said this: 55.000 divided on 34 is 1.617. So you see that whole numbers 'carry' a lot of information,--and so in a certain sense, a decimal number is no more than a way to write whole numbers.

So the rule is, multiply by 1,000 or 10,000 or 100,000 or so--as for the first number--when you wish to get a more exact result than whole number division can give. The result you will then read as having as many decimals as the number of 0's in the 1,000 or whatever you multiplied the first number up with.

A comment for specially interested:

How far can we go with this idea of only using whole numbers? Very far--just remember one thing: when you try and divide a number on a decimal number, it's a funny and different thing altogether--just try it out in the Spreadsheet (app# 3555558) if you have the time. If you divide, say, 5, on 0.25, it becomes a larger number: and so inside the Spreadsheet there is a function that figures out the right way to divide on decimal numbers. To calculate such division means getting into big numbers --and bigger the more decimal digits there are--and so, that's where 60-bit numbers become interesting, to do serious division work with decimal numbers.

Another 'famous' use of decimal number is when we speak of a circle, and we know something of its 'radius' (that's half of how long circle it is if you cut it straight over). So if it has radius 10,000 meters or something, how long pathway around the circle? Type

```
        tp
        nn
```

and it says '62832'. You can think of 'tp' as 'To walk a Path around the circle' or as 'two pi' ('pi' being the Greek letter used for that in ancient philosophy). Why do we have to write 1.0000 and 6.2832 when it is just as clear to write 10 thousand and 62832?

Half of tp is called 'pi', which you can get if you type

```
        pi
        nn
```

and then it says '31416'. So these numbers aren't meant to be altogether exact--there's a lot of thinking into this by philosophers in ancient times. It was found that no matter how many additional digits one adds, it is still not absolutely exact, but always rather a circa thing.

So with this way of thinking about circles, we can get to such as the 'sine' and the 'cosine' curves, that--really nicely--have the same type of curves that we see that a circle has, but more like waves. And to calculate around with this means also getting into such as multiplying a number with itself (also called 'squaring') and extracting the number that has been multiplied with itself (also called 'finding the square root'). For this sort of thing, G15 PMN has inbuilt functions. And there are apps that can make use of the longer numbers, the 60-bit numbers, if we need more decimals after the 'dot'.

So when you work with the four plus one ways of doing arithmetic -- addition, substraction, multiplication, division, and what we called 'mo' (or remainder)--then you are looking into, in a way, something that can be called--to use a grand type of word--'order'. You are studying something that is, in a way, both the most complex and the most simple there is. And by using these four plus one ways of arithmetic on numbers, especially numbers of just one, two or three digits, or so, you can get to become so familiar with some numbers they just about can be called 'friends'. And this friendship, if that's the right word, with numbers, can help you along in --and I'm not over-stating it--absolutely every field of life, from dancing and art to swimming.

This "art of arithmetic" goes along with a sense of the power of your own, natural, human mind. The machine is there to help us: but we have shaped it to make it more easy for us to go through our thinking, also about numbers --and, occasionally,--also when we think about the infinite, and how the infinite and these numbers somehow belong together, in exciting and also mysterious ways.

Part B, chapter 2: DISCOVERING OUR GOLDEN RATIOS

Imagine a world composed only of circles, squares, triangles, and perfect rectangles and such. How would it be to live in it? I suppose it would be a little bit like being extremely tiny and living in a bowl of salt. Rather boring, except possibly for the first minutes.

Then imagine living in a world which has nothing of circles, squares, triangles or rectangles. Rather like a living in a soup, a soup in which there is nothing that stands out except in some kind of vague, foggy way--no clear lines, nothing geometrical. Again, I would think that we agree that it would certainly be rather boring.

In between these two rather puzzling worlds--we can call them 'extremes', for they are extremely just one thing or another--we have the world in which we live in, in which there are lots and lots of forms that are not quite the perfect circles or squares nor just completely soup-like or blurry but they are, excitingly, standing out as both having something to do with geometry and with what we can call 'floating out'.

Think of somebody who has trained her shoulders rather much, perhaps in dance, or martial arts, or just exercise of some kind. And so her shoulders become a little more rectangular. Right? A little more of the right angles, when seen at a distance. And yet there is not just more of that, but also more of a curve--the muscles are, as we say --at least a little 'bulging'.

So that can be very attractive. But somebody who is but rectangular or but bulging--or who has nothing of the rectangular at all nor of anything bulging anywhere--you follow? The extremes, again, may not be so attractive. So you are combining things, taking something from geometry and the 'perfect forms', and something from a more fluid world, and living beings sort of exist in between all that --as does the world of clouds, the Sun, the stars, the oceans, and so on--if you think about it that way.

In order to learn more about how we understand beauty and the attractive, it would be interesting to find if there are some rather 'perfect forms', like a rectangle, that nevertheless is just so shaped that it connects to the organic, rich life more than most such forms. And some of you may already have heard of what it takes to make such a form,--at least the phrase. And the phrase is: The Golden Ratio.

Have you heard about it? What do you know about it? If, like me, you know something about it, I think you will also agree that there is a sense of there being always more to find out about The Golden Ratio. It keeps on being exciting to me--when I paint, or meet people, whatever I do, The Golden Ratio keeps coming up as an idea, or as a

question, or as a possibility, and it is always, I find, something creative about this. So let's explore this--just very simply--and connect it to one of the simplest possible programs we can make in G15 PMN. (Then we'll take it up more later on as we explore art of thinking.)
  We'll call our little program for 'fibo', and that's because one of these ancient thinkers again left his mark on some good work about this (called "Fibonacci" but his name was really something like "Leonardo Bonacci").

So I'll give a first glimpse into all this, and I can guarantee you that if you look into it once in a while, you'll find some way or other to make good use of it. It's a little bit involved to explain, so try and be patient with me in this chapter, and perhaps go back and read parts of it later. The Golden Ratio is best explained in a bit jumpy way, so we get a real sense of its wonder and don't merely think of it as yet another number.

Any rectangle, no matter what size it has, may have the Golden Ratio about it, when the length of the longest line divide in a way to the shorter. How long the longest side is compared to how long the shortest side is something we can call "proportions". That word we typically use when we do such as comparing lengths and widths. And a related word--not so typical in daily life language--is "ratio". And so, 'the golden ratio' is something that every artist must know about--and has talked about and used for ages, even though perhaps nobody has exact understanding of what it is all about. I think it's a bit of mystery to all.
  While I talk so warmly about the golden ratio, let's bear in mind that the world of beauty, art, seeing etc is infinite--and also infinitely more than the golden ratio. Let's not get stuck on it. It's immensely useful to know about, though, if we don't get into greedy over-use of it.

Now we're going to come back to the Golden Ratio after the next program. You know the ingredients of what comes up--every bit of it--except one or two small things. And you even know the Golden Ratio number, because in the previous chapter we showed that 55 divided at 34 is between one and two--we got a result that told us that it is about 1.617. So it's more than one and a half, but not much more. And, with that number of digits, it's pretty close to the number that people talk about when they talk about the Golden Ratio--because that's 1.618! Like the numbers we talked about in connection to the circle--the 'tp' and the 'pi'--there are studies that show that no matter how many digits one keeps on adding as decimals, one won't get it exactly right. But 'exactly right' is for atoms. An artist can go far with 1.618, or 55 to 34, or 8 to 5 or even 5 to 3. So even if I say that the Golden Ratio 'is' 1.618, it's good if you remember that these numbers are circa.

Just as we can put the first letter of the Sun in capital when we please, so also can we put the first letters of the Golden Ratio in capital--but in such cases we must also be forgiven for dropping capital letters when we feel that is easier. All right, so the golden ratio is about 55 to 34, or, with decimal numbers, about 1.618.

Next, let's just do some programming, so that we produce the numbers 55 and 34, and many more numbers that all somehow talk to us about the golden ratio. It's a little bit involved, but the program is just one card.

So, where does '55' and '34' come in? That's what we're going to show next. And here, as said, you know every bit of what comes--except a few thingies. You already know that:-

'f' makes a copy ('forges' a copy) of number on top
    of stack
'w' switches the two numbers on top of stack ('sWitch')
'sh' takes away ('shucks' away) the number that is on
    top of stack

and you'll only need one more of these, namely

'd' which copies the second number on stack and puts it to the top (imagine that it makes a 'briDge' over the number that is on top of stack, to the second number)

And we already know the other things we need--that 'ad' adds numbers, that 'nn' takes a number away from the stack and shows it on the display, and that LL: .. LO stuff makes a loop. So here we go! Study this little piece and I'll promise I'll explain everything about it:

```
        fibo=               d
        1                   ad
        2                   f
                            nn
        ll:17               lo

                            sh
        w                   sh.
```

This will produce a series of numbers that more and more have the golden ratio in between them. How that is, we'll talk about as soon as we have talked about how the program works. At some point, it would be lovely if you get it into card i:1. If you have followed the chapters step by step on the same PC there's probably stuff on i:2 and so on also. In one of the earliest chapters in Part A we talk--in a 'Useful To Know' part--how to cleanse a series of cards. Perhaps you should do that now? And when you type in the card, you use your skills--including <CTR>-R to get to the second column and <CTR>-S to save it to the right place, i:1, by typing in

```
        i1
```

when it asks and press lineshift after. You'll notice that when you type in a program in the CAR (as we call it)--the card editor, that is to say--you will see things about the program--new things. They suddenly stand out. And that's no coincidence but exactly how it is meant to be. That font and that screen size and so on were made together with the shaping of the PMN part of G15 platform.

So, now let's look at the program and discuss it and we'll understand it perfectly.

First, notice that '1' and '2' are put on the stack right at the beginning. Then there is a LO: .. LL type of thing--which, as we know, means that the PC repeats something, in this case, seventeen times. Right? Then, after the LL (I write it in uppercase sometimes so it is easier to read in normal text, although it is meant to be in lowercase when you type it in, as always)--after that LL you find 'sh' and again 'sh'. So, if the program is correctly made, there will be two numbers on the stack when the loop--the repetition part--starts, and there will still be two numbers on the stack when it's done (but not necessarily the same two!).

Right, so now let's look at--we can say, 'analyze', the LL: .. LO part, the loop. What goes on here? First, the 'w', then the 'd', then an 'ad'. Hm. Let's draw that up: On the stack is 1 and 2. The most last number is on the top, so 2 is on the top, even if we type it in underneath the 1. That's a bit confusing until you get used to it.

Then 'w' is done and 1 is suddenly on the top, because 'w' switches those two around.

Then 'd' is done and we have 2 and 1 and 2 on the stack.

Then 'ad' is done and we have 2 and 3 on the stack.

So 1 and 2 was 'magically' changed into 2 and 3. What's that all about? Let's see, let's see.

The rest of the loop changes nothing--except on the screen, there is the 'f' and the 'nn' which simply shows the top on the stack. Right? The 'f' copies the top of the stack and 'nn' picks it off the stack and prints it on your PC's monitor. So we ought to see 3 there!

Now, let's have a look, but a briefer look, what happens when the next round of the loop begins, and the same series of things--w, d, ad are done again:

First, 2 and 3 becomes 3 and 2, then 3 and 2 becomes, through 'd', 3 and 2 and 3, then, after ad, we have 3 and 5. So 2 and 3 'magically' gets transformed into 3 and 5.

And 5 is put to the screen!

So, a way to say, in less complicated words, what happens, is this: the two numbers are added, and the sum is shown on the screen. But we have kept a copy of the previous sum we had. So each time, the two most recent numbers are added.

Notice this, then: the two most recent numbers are added. Try it in your mind when you look at these numbers:

```
  3    5    8    13    21    34    55
```

and there we have our pair of numbers that we already have worked on in the previous chapter, 55 and 34. And the list goes on--I think we put 17 after the LL: and so we get really cool numbers like 377 and 233 as well.

How easy is this program? It is short, and we can say it rather easily--'the two most recent are added'--but it does require a bit of thinking. I'm not saying it is the most obvious thing in the world. If we want it to look more simple, but accept that it gets a little longer, we can use some commands we haven't used so far in the book, so that there is less need for 'f' and 'w' and 'd'. I'll just mention these extra commands, because it is high time I do mention them--and because they are wondeful to know about when we do other programs. They also are ways in which this programming language does have something in its core that never was in such early languages as the "Forth" that I mentioned in an earlier chapter, and that is a kind of 'ancestor' of G15 PMN. These simple commands are:

```
  s1 s2 s3 s4 s5 s6 s7 s8 s9  and  sx
```

as well as

```
  t1 t2 t3 t4 t5 t6 t7 t8 t9  and  tx
```

What these commands do is that they store numbers in what we could call a 'private' place for each function. So each function can store as much as it likes in these places, without bothering about how other functions are made. But when it comes to the stack, that's shared between all

functions.
  To get out the numbers that s1 .. s9 and sx sets, use:
  i1 i2 i3 i4 i5 i6 i7 i8 i9   and ix
  And to get out the numbers set by t1 .. t9 and tx, use:
  j1 j2 j3 j4 j5 j6 j7 j8 j9   and jx
A way to remember them, then, could go like this: "Use S to InSert into the i's", and, "Use T to adJusT the J's."
  And the 'x' is used by the ancient Romans to mean 'ten'.
  Now, earlier on, in a chapter or two, we have used, inside an LL: .. LO type of repetition, one of these already: we have used 'i1', haven't we? Because that's where the counting takes places, when we have one LL:..LO. When we have two on top of each other, it uses 'i1' and 'i2'. (You might remember this by the 'i' in 'countIng'.)
  So we could write the 'fibo' above for instance by using t1 and t2 to set the two most recent numbers. Then, to add them up, we would write,

```
        j1
        j2
        ad
```

and so we could build it up without that much use of the stack. And so you'll very often find that there are many ways of doing anything. (That's something that a teacher on programming languages named Larry Wall has often pointed out).
  Sometimes, then, a slightly longer program is so much more simple to understand that, unless the PC is pressed hard to do many things fast, we would prefer to put in the longer program.

Alright, now let's connect the numbers to the description of the world that we began with. How does it all tie up? We spoke of finding some proportions of a rectangle that is a bit more organic, not like the 'bowl of salt' of only perfect circles and triangles and rectangles and such, but which has something to do with life--at least a hint of it. Now take any rectangle which is about 3 times 5,--it can be meters or centimeters or feet or whatever way you prefer. You notice, if you start up the program on i:1 in the terminal, like this:

```
        ^i1
        cc
        fibo
```

that the first numbers are 3 and 5. That's already very near the 1.618, which we have talked about as circa the golden ratio. To see this, we can divide the numbers on each other, just as we did with 34 and 55 in last chapter. Then we took the largest number, multiplied by for instance 1,000, then the smaller, and then we used 'di'.

And so each of these numbers are formed by adding the two most recent. In drawing up a rectangle, this somehow, and I admit it is a bit mysterious, but it works--it somehow becomes this: that each rectangle as it were contains an infinite number of itself within itself, both smaller and larger versions. We get some kind of spiral effect and somehow we notice this in our minds, our eyes pick it up, back there in the depths of ourselves, and it automatically becomes something more interesting about this rectangle compared to other rectangles.

Let's see how that can be. If you have a rectangle that is 5 times 8 in size, you can chop off a square that is 5 times 5 and that leaves you with a smaller rectangle that is 5 times 3. You can do it with pieces of paper. Make the largest perfect square you can, clip it off, and what remains is a rectangle that, although it is smaller, is about 5 times 3. Now 5 to 3 isn't far from 8 to 5,--in fact, these are all numbers that the fibo program made!

Right? It made these numbers: 3 5 8 13 21 34 55 and some more like that. So the nearer we have this perfect golden ratio that these numbers work up towards, better and better, the more it is the case that each time we chop off a square, we have the same rectangle all over again, just smaller!

The more perfect the golden ratio of our rectangle, the more it becomes possible for your eyes, your mind, your brain, to get a swirling kind of spiral through the edges of this infinitely many equal but different sized rectangles, all contained within one another.

To actually draw a spiral is possible if we take a line that swirls through the corners of the rectangles, which are inside one another. And this spiral turns out to have a shape that is often found in Nature.

Now, we can go the other way as well. Suppose we start with a 3 times 5 rectangle. Take the longest side of it, which is 5, and make a square just this size--5 x 5--and put it beside that rectangle on its longest side. Then we have a 5 x 8 rectangle. You see the numbers here? 3, 5, 8.

How is it that your eyes, your mind, can do this sort of thing in a blink? Because shaping a full square, or taking away a full square, is one of the easiest and most natural thing for your mind to do when it studies anything at all, from infancy and up. It is just such a natural thing. If a circle isn't full, then maybe one just has to look a little more, and it becomes full. If something of a square is there, perhaps it's just to look a little more, and one can see the rest of the square--or something like a square --or like a circle. You see? There is a word, a long and complicated word for this way of looking at things, and it is called 'Gestalt Psychology'. We naturally look for simple patterns to fulfill themselves. We're born to do it. Our minds do it all the time. When we study this sort of thing in science, we have another word for this--a long and complicated but very fine word, and that is 'perception'. So we can talk about 'seeing Gestalts' when we mean to see such simple patterns, and when we don't just talk about seeing but also listening and feeling and and so on, we can say 'Gestalt perception'. Yes, in music and all the other senses, the golden ratio stuff comes in wonderfully.

So, when our minds, armed with this amazing human skill, which works fast as lightening and often without us noticing it very much, meet with the golden ratio, some kind of natural inner activity raises up. This natural inner activity becomes an excitement. Things happen. And for instance in dance, when you see the movement of the body, you can imagine how many times each minute some kind of golden ratio here and there is touched on--even if only for a split second--but it adds up, and contributes, to a sense of beauty. It's not the only thing about beauty, but when we explore beauty, think about beauty, then we understand more when we also think about the golden ratio. Keep it as a glowing question, always alive: what is the golden ratio? And is it somehow hidden here, or there? Is it part of this or that intense experience?

Part B, chapter 3: GETTING WILD WITH WARPS

In the previous chapter we talked about golden ratios, as part of the experience, also intense experience, including of such things as beauty. One of the most obvious places to go if you wish to think about beauty is to somewhere flowers are growing. There's such a balance there, between symmetry and the lively, mild breaking of symmetry, that the sight alone can be breath-taking; then add to all that the actual magic of scents flowing to our nostrils from a vast variety of flowers; and the delight in seeing how they stretch towards light and require not much more for their fantastic work than a bit of water and soil and some tender care.

Yet study the flowers a little deeper, and you'll notice that, for all their similarities within a type, the exact way each flower grows is quite often unpredictable. Will a new stem sprout off here, or there? Will it be this number or that number of buds? Will they open simultaneously or one by one or will some buds just drop off?

And there's a lot of fine words in the English language that somehow ties in with the human fascination for flowers. One of these is 'stamina', which not only is something we talk about when we have a lot of energy and motivation to train or swim long distances or whatever, but it is also having to do with the core of the flower and how it may reproduce itself by an almost sexual contact between itself and other flowers--a process that may be helped by human instruments also.

Another word, used also to talk of just this way in which a new stem may suddenly arise out of an existing stem is 'warp'--we might say that "the new stem warps out". The idea of warping carries over to different parts of human living, where it becomes a word that we can, perhaps for fun--also in stories about outer space--and sometimes for real (in one way or another)--use to mean something like a great sudden change. It can be a change that is perhaps surprising.

An example where we could use the word 'warp' is when we play together, and then something changes. Perhaps the play, being a mild fight, becomes a real fight. Perhaps the play, being of one kind, changes into something, still harmonious, but an altogether different type of play--and not in a way that is exactly following what happened before. We could say, "the play sort of warped into a different thing".

A mild play-fight can become a serious fight--or a mild talking together can become angry shouting. Is such a change always a bad thing? But if we are always mild, always entirely mild-mannered, we may find ourselves so that when it's good that somebody else have got to change, we are not going to help produce that change. So a warping can be dramatic, or it can be just a warp from one easy light-hearted experience to another, or a warp from something experienced as bad to something experienced as good. The word "warp" covers all these cases.

Now I have softened up the word 'warp', and in this chapter I wish to say that it's a most central word in G15 PMN and one of the things that somehow stands out a bit compared to what I know of the history of programming languages. I know of other words, used in other programming languages, for something of the same--but the very choice of words show that it isn't quite taken to mean the same thing (the most common alternative is this: "pointer"--and I think you agree that it doesn't have the same flavour of direct change at all--and so it leads to a different type of thinking when it is called that).

In using just about anything in G15 PMN, you have already used warps. I will prove the point. Get up the terminal, as an experiment, and type in what day it is, with & before and & after also. Like this, perhaps:

```
        &thursday&
```

Have you done it? Great. Now next, you might want to something like 'pp' or like 'b9', both these displays the word once more--but please hold back a moment. We are going to do it different this time. What we will now do is to have a look at what goes on behind the scenes as it were, or 'under the lid' of the machine. I want you to 'get a feel' for warps, and I tell you: there's a warp on the stack right now. To prove the point to you, please type in (the first 'f' command makes an extra copy of the number, so we can have a look at it):

```
        f
        nn
```

and you get some kind of big number. It could be 23043232 or 798034823 or 5654459--I have no idea which it is. And that's one of the things we can know about a warp: it's a type of number that can change between each PC and even change from one run to another on the same PC. What can we use that number for? It turns out that, even if it is a bit strange, it has many uses, many uses indeed. One use is this: we can--especially if we store it somewhere--show the same text several times, or do other work on it.

So, assuming you are still using that terminal, type

```
         sx
```

What now happens is that we can use the command 'ix' to get that peculiar number out several times over, as long as we use this terminal anyway. (You remember 'sx' and 'ix' from the overview in last chapter? We had s1, s2, s3 and up to s9 and then sx, remember? And then i1, i2, i3 and up to i9 and then ix to get it out.)

So let's try to make use of the warp! Try, for instance,

```
         ix
         pp
```

and the name of the day comes out in robotfont. Try then,

```
         ix
         b9
```

and the name of the day comes out in b9font. There are all sorts of things we can do with a warp like this,--one more example is this: how many letters does it have? Then we use the command 'lk'--which is a shortened form of "LooK" (or "luck", if you like)--and it will look where the warp says it should look, and the first thing it comes around to find is the number of letters. Try this, if you like:

```
        ix
        lk
        nn
```

and it will say how many letters of the name of this day. In case of us entering &thursday&, for instance, it should say 8. Had we written & thursday& it would have said 9, for it counts blanks also.

In a moment we'll look into an even more dramatic type of warp. But first, let's summarize something of what we said in a 'useful thing to know':

===>USEFUL THING TO KNOW: A WARP TO A TEXT CAN SAY LENGTH
In G15 PMN, the word 'warp' can mean several things, but they are all related. Usually, they are really big numbers and in by far most cases the warps are different between each program runs. So the smart thing to do is to find out way to 'get warps' out of the machine and then store them somewhere--and be sure that the next time you run the program, it gets entirely fresh and new warps. When we

talk of a 'warp to a text', we typically may mean a number that is so that--at least if the text is tiny--we can give it to 'pp' or such. And whenever we use &..& there is such a warp on the stack. So, for instance,

```
        &The Sun&
        sx
```

stores the text "The Sun" somewhere in the PC's memory, and the number--the warp--can be fetched by 'ix', so that this will show it again:

```
        ix
        pp
```

and in these cases we can find the length of the text by using the command 'lk':

```
        ix
        lk
        nn
```

will tell '7' since "The Sun" has seven places (characters --we can say--and a 'character' includes also space).

Let's at once now jump--or 'warp'--to another way of using that word, "warp": not just texts, but also the little programs we are making and giving names by = and finshing with a dot . are having warps. This is some of the funniest ways of programming there are--but to see the fun of it, we have got to learn a few more things, also about something called 'arrays' (which is a way we store many numbers in a series).

But it's easy to learn warp in this way--and it is more truly living up to its name when we see what we're going to do next. See here, let's have a very tiny program typed in on the terminal, something like this:

```
        sayhi=
        ll:15
        &You=the best&
        pp
        lo.
```

So each time you type in 'sayhi' it will say, fifteen times, some little funny text like that--You==the best. Test it just once and then we get on to finding its warp! This is how (the ^ sign is often found by <SHIFT>-6):

```
        ^sayhi
        ff
```

and now there is a mystery number on the stack. A warp. Or a warp number. The warp is "to the function" 'sayhi'. You could have written &sayhi& instead of ^sayhi, but the use of the ^ sign is enough when there are no blanks in the text. We have to put some kind of quote-sign, like ^ or &..& around the 'sayhi', otherwise it will just perform. This time, we didn't want it to perform; we wanted to find its warp. Store the number! Type,

```
        s1
```

and as you type this, it will show the number:

```
        i1
        nn
```

What is it showing? 2339939 or 89564895 or 7543844 or anything between 1 and a billion or so. I have no idea, nor does it matter exactly what it is, as long as we have it stored with its exact value that it has in just this run on the terminal on just the PC you are using right now. Your neighbour's PC may have a different value. It doesn't matter. The clue is that it is now the right value --the right warp--for this run of the terminal. Let's put it to test! Type

```
        i1
        pf
```

and look what happens! The function starts, the little fun loop shows the text, you=the best or whatever it was,-- but completely without us having named the function. So 'ff' means 'find function warp' and 'pf' means 'perform function warp'.

  If you don't believe me, try printing out the number on the screen again, by

```
        i1
        nn
```

and, for the sake of the argument, let's imagine that the number you found was 8590299. It's going to be different, but if it was, then you could type

```
        8590299
        pf
```

and, lo and behold!, again runs your program. Instead, you type the number you found--exactly right--and then pf. And you find that your fun little program starts.

  In case you wish to upset the PC so that you practically have to cut the power wire--no, I don't mean that--but you may certainly have to reboot it, type any number you like and then

```
        pf
```

and do it with your hand at the power off-button! This is not recommended practise, but you should have a go at it as a budding forceful programmer! You may also find that nothing at all happens. If so, try another number, but very quickly type

```
        qu
```
and get out of the terminal and then do the normal CTR-Q and type REB for reboot. For something about the PC may have gotten wierd. Because you warped its 'memory', its RAM,--quite possibly. And to be moralistic, never do this with a PC that has something like a robot connected to it, because the robot may do dangerous movements. So, now you have it--you see what power you have at your fingertips, literally! Let's sum this up:

===>USEFUL THING TO KNOW: WARPS CAN BE TO PROGRAMS
If you have any program, any function, of three or more letters, you can find its warp number by 'ff'--'find function warp'. For instance, if you have made 'myprogram'
```
        ^myprogram
        ff
```
and then, once you have it, you may want to store it somewhere. If you store it, for instance, by 'tx', you can get it out again by 'jx' and then you can perform it by 'pf'. Here's how: first store it:
```
        tx
```
then, each time you wish to run that 'myprogram' you can write
```
        jx
        pf
```
And there are still more types of warps--also to arrays, where we can store warps to many programs and keep them available in between many functions in your program.

Let me just add a note about how we talk about these things--it may be easier in the beginning to say, 'a warp to something'. But you'll often encounter other ways of saying it, such as 'a warp of something'--such as, 'the warp of a function', or 'the warp of an array'--or even yet shorter--'a function warp', 'an array warp'. These are all the same types of things that we discussed in this chapter. And instead of saying, 'let's perform a function via this warp', we could say something like, 'let's warp to this function!'.

Part B, chapter 4: THE DEEPER MEANING OF ETCETERA

You have perhaps noticed how almost everything--if not everything--we have done at the computer so far somehow have had to do with numbers. And yet they have given rise to other things, letters, dots, even star-like dots, mouse like movements, calculation, thoughts about the golden ratio and so on. And of course, when we discussed warps, we discussed still more numbers. So we have numbers, and they can do certain things, it seems, when they are in a particular place in the computer.

Even before the time of computers, many thinkers, or, as we call them, philosophers--seekers of truth, wisdom, lovers of the mind, the world--were enthusiastic about numbers and how they somehow may partake in everything--even music.

One of the things--and it's fair to say that the controversy still exists at the time of writing this--that has created disagreement between thinkers has to do with how numbers like 1, 2 and 3, connect--or don't connect at all, with the idea of the infinite. In one way, they seem like a big opposite: the numbers are very much not infinite--we can call them 'finite'--and then we can imagine some kind of infinite landscape without borders even a million billion kilometers away,--a landscape in which there is no point of trying to apply any number for it is forever beyond even the largest number possible to write.

For instance, let's imagine that we have some kind of gigantic group, or bag, of numbers. Some thinkers have wanted to use the word 'set'--they say, let's imagine that we have a 'set' of numbers. A big collection. And let's imagine that it is so big that every number from 1, 2 and 3 and up is in it.

They have tried to think through this and they have called such a set for N. And they say: it is so big that no matter how big a number you think of, N has numbers that are even bigger. So N must be infinite in size.

Now I'm going to give you a bit of my own take on this, and perhaps, at the moment of writing this, my take on it isn't very common--just so you know. But I think it belongs in a book that teaches you much about numbers, also in programming--and I have thought about this when shaping G15 PMN. It isn't very practical, but a sort of day-dream type of thinking about infinity--but then, later we may find that it gets a little practical after all.

Let's see, then. All right, let's think about this collection N. What type of numbers are in it? I'm going to suggest that some very funny type of numbers sort of jumped into it, even though we didn't say it should be so.

Just think again of its size: N is a group of all numbers there are, from 1, 2, 3 and up, and so the size of it is infinite. Very well, it is gigantic. Greater than any number we can say, when it comes to how many are in it.

Let's ask this: is the size of it some kind of mystery number? Well, it's infinity. But is infinity a number? Some have said--perhaps infinity is a kind, a very special kind of number. But we put only ordinary numbers inside N, so even if its size is this special kind of number--let's call that special kind of infinite number for i--we would perhaps not expect it to be inside N. After all, we begin with 1, 2, and 3, and go up in easy steps.

So this or these mysteries, 'i' for infinite, isn't in the set N--that's the idea most thinkers in both past and present have. My take on it is that somehow the i's have gotten into N after all. And I think I can show you how my argument may be right. Just let's build up the idea of N a little slowly, and we'll see it together.

We start with 1, then we add the number 2 to it, then we add the number 3 to it. So, in building up the idea of N, we first see that it has size 1, then it has size 2, then it has size 3. Right? So onwards with 4, 5, 6. And up we go. At every step, we find that the size of it is in the bag. Right?

Then, how do we imagine, in our fantasy world where all sorts of things that make clear ideas can be seen and felt and thought about, that we complete the building of N? We must imagine that it stretches out, as we add numbers in the thousands, in the millions, in the zintillions or googleplexions or whatever we wish to call such fantasy numbers--and onwards and onwards, like clouds in the distant horison. Further and further and further. At every step, then, it's natural and a clear idea to say: the size of it, even if it is very very big, is still some kind of number and this number is in the bag N. Okay?

Then let's will forth the sense that we sweep forward infinitely fast and see that N is complete. We have lifted our gaze a little above the furtherst cloud, and it is now complete. It is done. N is infinite. N is infinite in size, something like 'i'. When we did lift our gaze, and peeked into the horizon and above it, we never said: we are going to disconnect the size of N as we build it from the numbers we put into it. So, when we lift our gaze and let the size of it go to infinite, we also are letting the numbers inside it go exactly together with this. That's the clearest idea. And clear ideas are all we can ask for when we talk about the infinite. We can't get the machines to help us out very much in these things, for machines are only about the finite. Right?

So, to repeat the last steps: we keep on adding larger and larger numbers to N and the size of N keeps on becoming bigger and bigger--and all the time, the number that is its size is also inside N. The number 'outside' of N is also inside N, if you like. Right? When it had just three members, three was its size and three was also inside it. When it has three thousand members, three thousand was its size and three thousand was also inside it. And when we lift our gaze and sweep through the universe or multiverse or cosmos or whatever we call it, in our imagination, then both the size of it and the content changes, from the finite, into the infinite. How can it not? We don't have any instrument, any way, in which we can prevent the infinite from somehow getting into the bag!

(The advanced students may object: can't we filter out the non-finite numbers afterwards, by checking whether they can be made by 1+1+1...+1? But how can you hover over an infinite process with such finite tools of calculation? That isn't a clear idea: it is clearer to say that infinity changes the situation fully. Adding such features means adding something that contains the infinity questions all over again, and we can use the same way of picking such added features apart.)

So the puzzling thing is that when we write something as simple as 1, 2, 3 and up, we are writing something that really takes us into the infinite in more than one ways. This mystery number i,--perhaps it is not just one, but many such mystery numbers. So it's more like, in my take on it, like 1, 2, 3, ... i ...

Another way of writing this is: 1, 2, 3, etc, i, etc.

And this has room in for the sense that i somehow is in movement, isn't just one thing, etc.

You might think that after a hundred or a thousand years, philosophers might have figured this one out so that nobody can disagree. But they haven't. When it comes to how infinity is understood, in number thinking, only a few has disagreed. I should mention one name here, a very respected name in logic--L.E.J. Brouwer, a Dutchman who lived in the 20th century. He would have nodded to my doubts of how the mainstream has handled infinity, even though he didn't say the same things as I do.

To be a little more concrete, in mostly all writing--but not so much in the writing of L.E.J. Brouwer--those who have been thinking deep about numbers and infinity have thought it fairly much a clear idea that we can make, in our minds, a set N such that it has all finite numbers and only finite numbers. After all, it's easy to say.

But when we look at it in our imagined world together, and that's something everyone can do--and children are often great at thinking in free and also philosophical ways and about universes and such--this I know for I regularly teach children--we saw that when we build up the set N, it's a big leap of imagination to get to the point that it is infinite. And when it is, we cannot be sure that it doesn't have some kind of infinite mystery numbers inside it. Rather the opposite, it seems almost certain that it does have some infinite mystery numbers inside it.

For advanced students:-
Just a brief comment on language, I use 'i' here just to talk about some kinds of infinity, not that 'i' hasn't or cannot be used in entirely different ways. And when I say such as 'infinity number', I do so well aware that to some the whole idea of talking about 'infinity' and 'number' in the same breath is patently nonsensical. (Let me also add that there are some thinkers, including G.J. Chaitin, who has used a symbol--omega--to stand for something which has some similarities with our 'i' here, but if you read this chapter throughout, you'll see that this isn't his take on it--and although I find flaws with Mr Chaitin's approach, I like that people are calling out disagreement with the present dull mainstream as for thinking about these things.)

All right. I have told you my take on it, and how most have been talking about it, and I have told you about my disagreement. If I'm so lucky that you go along with believing in me, then let's together ask a question: what does it tell us about numbers or about the infinite? What can we say now, that wasn't that easy to say before? What cwe see now, that is new, and that comes out of our imaginations and musings over the infinite cosmos?

Well, perhaps this: that when we began talking about the mystery infinite numbers, we came to them by a lot of movement in our imagination, and we also, perhaps, came to a sense that there may be many more of these infinite numbers. So these are some words:- movement, more of them. If there are more of them, there may be an infinity of them. We can use the word 'diversity'. There may be a lot lot, a diversity of infinite numbers i. And somehow they arise because it isn't clear how we can 'close them out'. So they are naturally there.

From this, then, we can then suggest: perhaps it's more natural, more easy, to talk about an infinite world in movement, where numbers are naturally somehow part of it; and that when something stands out as a more still and fixed and finite number like 1, 2 or 3, it shouldn't be taken to mean that it doesn't somehow connect to all these moving infinities.

And this leads us to suggest: the finite numbers are somehow 'created by' infinite numbers in some kind of movement. An image that could suggest this: when two giant waves on the ocean meet--if they meet in a very direct way and are really huge--they may create a new series of waves --perhaps as smaller ripples. And so perhaps the finite numbers are created with infinite numbers meet and dance together, as it were.

It's just an image, of course, but it is a type of a philosophy of numbers and the infinite that just may just happen to be more right than many of the things that have been said about cosmos and numbers by those who perhaps haven't looked into this puzzling giant collection which we named 'N'.

(For advanced students: it is quite common to talk about many different infinities in 20th century literature on sets. But these have usually taken for granted that 'N' in the narrow way mentioned--that it is easy to ensure that it has only finite members--is a starting-point. In the case that their idea of 'N' is unclear, most talk about 'many infinities' or the like in 20th century logic becomes meaningless and one must start all over again.)

So where do we go from here, to come back to our G15 PMN programming? We are going to work, in fact, with something much like a 'set' or a 'collection'--and in G15 PMN we call this 'array'. And we have learned that in our minds, we can say 'etc'--which comes from ancient Latin, 'et cetera', which means again, 'and so further'. But on the computer, we must always tell how big something is going to be before we make it, and it must not only be finite but clearly fit within the sizes that the computer can handle. So that's what we are going to do next!

So let's roll up our sleeves, if we do have sleeves, and work out something on card i:1. Before you do so, you should check that i:2 and some cards up are all clear--that they have those squarish images that tell you that the card is what we in one of the first chapters in this book called 'nil'--which is to say, nothing. And in case some cards need cleansing, you have a 'useful thing to know' in just those early chapters that tell you how to clean cards, you know--so they become 'nilcards'.
  Let's imagine that we are on our way to build a game, which has to do with money--this kind of dollar, that kind of dollar, you follow? Then we would want a place to store how much money the player has, so that any function we make can change that sum or read that sum quite freely.
  Something like this can do the trick--and it's called a 'variable'--but wait a moment before we put it to card i:1 because we are going to add a little more:

```
        money=
        ^.
```

So what's all this about? The new word 'money' is created, with the normal '=' at the start and the normal '.' at the end--but there is nothing there! Except this ^ (which is often found at <SHIFT>-6, ie, above the digit 6). We have seen it briefly before, when we talked about warps, in the connection with 'ff' or 'Find Function'. We said it was a way to quote something when it doesn't have any blanks. Well, that's right. There are no blanks here! But also there is no size to the quote! But that's fine, for all we need is to store a number, that has to do with money. And a quote always begin with a number that tells its length. The fact is that this doesn't have to be the length, as long as we stick to a certain way of using it. There is

room for just one number whenever you see that '^.' thing.
  So, in short, when there is something like
```
        blablah=
        ^.
```
then you can think of ^ as a 'v' in Variable upside-down.
For that's just how we make a variable in G15 PMN!
  So here's how to put a value to it:
```
        money
        10000000
        kl
```
and here's how to get the value out of it:
```
        money
        lk
```
so that we can show it eg by
```
        nn
```
You see that 'kl' and 'lk' are the same letters, just the opposite way. The command 'lk' you can think of eg as 'LooK'. When you turn it around, you might think of it as, for instance, 'KiLL the earlier value there and put in this instead!' or, if you are a pacifist and like to spell things with 'k' instead of 'c': 'KouLd you please put this value to this variable!'

A thing to remember: when we speak of a variable, we mean a way to store one number. When we speak of an array, we mean a way to store many numbers. In the next part we will also work out a clever way to arrange arrays so that they have both columns and rows (like the dots on the screen), and this type of array we call 'matrix'. Matrices can have even more about them than just columns and rows, too.

So these things you could test at a terminal if you like, but I understand that you are now rapidly learning G15 PMN and merely by glancing at the above you comprehend it fully. So let's raze on! For we said, didn't we, that we wanted money in more than one currency. Several types of dollars, for instance. So we have to tweak, to fix on, how we use this money variable a little. This is one way to write it (and, as always, there are more ways, and some of the other ways are much shorter, but this is the way you can do with most arrays of any size)--here's card i:1. Type it in and we'll talk about all parts of it. Then in i:2 and i:3 we'll put it to use. This is a way to make an array, whether it is as short as here, with only three numbers, or a million times as big:
```
        money=              &&
        ^.

        3                    money
        sz                   kl
```

As always, it doesn't matter where you put the blank lines as long as you like it that way. And we read it so that it is first the left column, then the right one (and the start of the right one you reach by <CTR>-R, when you are in the Edit mode of the card, which you get to by doing a [RIGHTCLICK] with the mouse).

And now the explanation!

First, we make the variable called 'money'. Fine, we have just talked about that.

Then we put '3' to the stack. That is the size of the array we are going to somehow connect to 'money', so that we can have three types of money currencies in it.

Then, 'sz'. You guessed it, it means 'SiZe'. So this 'sz' command, which we use before or in between or after making functions, tells the PC that the next quote, no matter how small it is--it can even be entirely empty, like when we type &&--that quote is going to be roomy, so it can vary in size however much we like up to what we tell 'sz'. And the quote can be an array of numbers.

You got that? You put a number to the stack, in this case '3'. Then you write 'sz'. Then you type in '&&' or something like that--a quote--and this quote doesn't have to be used as a quote of letters, it can be an array of numbers. That a quote also can be an array of numbers isn't all that strange when we remember that everything in the PC, including letters, is stored as numbers (as we'll in the next part, big A has the number 65, small a 97).

And, yes, indeed, after the 'sz' we do see a '&&', right on top of the start of the right column. When we have done one 'sz' and then a '&&' then we have sort of 'used up' that 'sz' command. We can make many arrays after one another but then we put in many cards like this after one another, and every one of them can have a 'sz' and a '&&' in them. (Or we can cram several arrays into one card, or spread them over even more cards--that's just a matter of how we like to put in blank lines and divide things up.)

The last two lines in our i:1 card says: 'money' and 'kl'. This tells the PC that the array we just made is going to be stored, somehow, in the variable money. If that sounds funny--yes, I agree!--it sounds funny. So let's say it in a more precise way: the warp to the array is stored in the variable. Or we can say, 'the warp of the array is stored in the variable.'

Exactly how we say it doesn't matter so much--because we'll do this thing many times over and then you'll get entirely used to it. The clue is that this type of card you got above is how to 'set up' an array of any size--and though there are shorter ways of setting up especially small arrays, this is the way that is enough to use for most purposes.

The next thing we have got to learn about arrays is how to put things into them and how to put things out of them. The rule of thumb is: you tell which position, which place --which position number--you want to put it in, or take it from. For this we have two simple commands, 'ay' and 'ya'. The 'ay' is as ArraY and 'ya' is, of course, the opposite sequence of the letters. So 'ay' is to get it out of the array and 'ya' to 'yapp it in'!

If we really were programming a game right now, I suppose it would make sense to give the player some start money. Let's imagine that we give the player a 1000 dollars of the first type, 2000 dollars of the second type, and 3000 dollars of the third type. Then we could do something like this, in i:2:

```
        1000                lk
        1                   ya
        money               3000
        lk                  3
        ya                  money
        2000                lk
        2                   ya
        money
```

This should perhaps have been typed in over several cards, but just read it in sequence: 1000 goes into position 1. 2000 goes into position 2. 3000 goes into position 2.
A clue is to read 'money' followed by 'lk' as one thingy. The word 'money' is a way to get to the beginning of the array, but we need 'lk' to bring it about.

Is it confusing that the 'lk' appears on top of the column to the right? But sometimes it is good to train oneself to read it as one flow--regardless of layout on the card. The computer goes through one step at a time. So can we do when we analyze what we type in.

So we 'yapp' in a 1000 to the first position in array money, 2000 to the 2nd position, and 3000 to the 3rd position. This we can do just like that--we don't have to make everything into a function with = and a dot, when we want it to be the start-up stuff. Do you mind getting this into card i:2?

What we have just done is, if you like, to make a 'set', the set of some money, which we could write like, {1000, 2000, 3000}. We began this chapter by talking about how complicated it gets when such innocent-looking sets get something like three dots in them: {1, 2, 3, ...}. Then we must bring in our imagination, and imagine it as well as we can, and when must not only imagine it, but also try and find out things that we can never check with our senses--in other words, we must use 'intuition'. And that's something a machine obviously doesn't have.

So when we work on an array on a PC, it is as finite as just about anything can be finite. When we make it, we start out by telling its maximal size. Then we must live up to the maximum size when it comes to putting things into it and getting things from it or the PC will protest.

The PC has memory--what really should be called 'memory' with quotes--and this is also called RAM (technically, that's because it was often called 'random access memory' to indicate that one can both read and write to it and, and that the sequence is free and 'random'). The RAM of the G15 PMN PC is organised so that when you have the warp number of it, you can at once go and get the stuff that's there, or change it. The warp numbers are, like all the core numbers in G15 PMN, never higher than about 2 billions.

Most of the time, as we have said, the warps work so that you don't have to actually view those numbers. But when we typed this into the PC,

```
        3000
        3
        money
        lk
```

then, just so you know,--at this point, there is a warp right there on top of the stack, which points straight at the first free position in 'money'. So when the next command is entered,

```
        ya
```

it takes it for granted that the warp of the array is on top of the stack, and right underneath it, position, and right underneath that again, the value we want into it. So to think of stacks is a sort of 'vertical' thing--and that which is on top of the stack is the last thing you typed in. We should get used to changing things around in our minds a little bit, then, to think of stacks clearly: we type

```
        3000
        3
        money
        lk
```

(again, remember to think of 'money' then 'lk' as a thing) and what's on stack is then the opposite way: with money on top, then 3, then 3000. Yet another way we can turn this around in our minds is to write it from left to right so that we can put it into a comment line. Remember the comment lines? They begin with a vertical bar, |, and can look like this:

```
| A fun program
| Updated today
```

If we wanted to make comments to explain how 'ya' works, we could say something like this:

```
| ya expects:
| n pos warp
| on stack,
| with warp
| to array
| on top
```

You'll very often see 'pos' where we mean 'position' in comments, so it gets short and snappy, and just 'n' to mean any suitable value or number. You'll also often, in comments to functions in G15 PMN, also see this type of writing--"n pos warp", or, "n, pos, warp"--or some letters like "a, b, c" written. This is to be read so that the rightmost thingy is on top of the stack. And there may be one comment line for the things that should be on the stack just before a function starts, and one comment line for the things that will be on the stack after it has done its job. In the case of 'ya', it just stores, so we don't expect it to give anything extra on the stack. In the case of 'ay', we expect it to give the value of that which is in the position on the stack.

We have gone a long long way in imagination--in a sense, we couldn't have gone any longer--we have been to infinity and back!--and we are working with a lot of new things here. So we're going to finish it quickly, by getting the quickest, easiest feedback from the computer as to how much money we have of each type. Just two cards--and we'll try it--and that's it! A brief explanation I'll give after we've tried it--to add on what's already said.

So, here's i:3, and i:4 completes the function 'ourmoney':

```
        ourmoney=           &Dollar #2&
        &Dollar #1&         pp
        pp                  2
        1                   money
        money               lk
        lk                  ay
        ay                  nn
        nn
```

Then, i:4, so identical you could copy i:3 if you like, and put it in by <CTR-T and fix on it (hint: you can press just <ENTER> after <CTR>-C when it's just one card):

```
                            &Dollar #3&
                            pp
                            3
                            money
                            lk
                            ay
                            nn.
```

This card i:4 can of course have a different spacing--you can use left column or both left and right columns. This is just how it might look if you space away the first column in card i:3, if you copy it over to i:4, and then modify the number '2' there to number '3'. In addition, put in that dot . on the last line.

When ready to see if you have typed it right, start up the terminal, type

```
        ^i1
        cc
```

and when it is right, try type

```
        ourmoney
```

and it should tell the sums we put in first, just very very simply.

All right, so how does it work? The function begins by printing "Dollar #1" to the screen. Then several things are put on the stack and 'ay' is done. The command 'ay' fetches a value. Which value? That's the '1' there, in the first column of card i:3. So that's the money in position #1. Right. We then see the phrase 'money' followed by 'lk'. As said above, but we repeat it: it's a kind of unit or package. The 'money' is the place where we have the array warp and 'lk' brings it up front. So we'll very often see

```
        arrayname
```

lk

together, whether we use 'ay', 'ya', or some more exotic words, which we'll encounter in the next part, such as 'ww' and 'yy' (for matrices).

And once 'ay' has produced the money of type #1, it shows this number on the screen. And onwards to column 2, and then the last column, in card i:4. They do exactly the same except for money type 2 and type 3.

All this was a lot of learning and you can now relax--we are going to make use of this array type of stuff in various places so that you get more and more used to it. And we'll see that it helps programming tremendously that we have these things, arrays in their various forms. It is quite something to learn to work with.

To finish this part off, before we're on to the next: we are seeing how the machine expects absolutely every little detail to be spelled out or else it won't do anything. In that sense, we are perfectly right when we say that the machine is 'stupid'--in another sense, we don't really have to call it that, for it is neither 'smart' nor 'stupid' for both of those words we use about living human minds and a machine isn't even on the starting-line as to having a mind, right?

And then, when we think about the larger questions--such as infinity--the machines can't help us, not directly anyhow. They can give us some confidence that we know how to handle some numbers, but they can't say whether we have the right guesses or intuitions when we speculate about what goes on in the imagined infinite landscapes of possible infinite gatherings.

In a sense, to simplify enormously, we have three fields of activities when we work with G15 PMN this way, from one week to the next: we work with finite numbers; we allow our minds to touch questions of infinity; and thirdly, all sorts of things--work, dance, eat, connecting to friends, new and old, and people who are strangers but whom we may want to know. So here we may have experiences of love and dialogue, sometimes conflicts, sometimes misunderstandings and apologies, and sometimes great fascinating new plans are made in collaboration with others.

In all these fields of activity--the finite, the infinite, and the social/private/interpersonal etc, we can have a lot of fun, and energies may flow between these avenues of unfoldment--'synergy', we can say. Yet, about infinity--plese don't overdo speculation about the infinite. Touch it--because that's part of the whole speculation about cosmos which ignites something of our inner flames--but don't get burned by trying to wrestle 'control' over the infinite. It can't be done. Be a visitor, but gently, as for the landscapes of infinity, and seek to leave without a trace each time.

PART C:

Part C, chapter 1: THE WORLD OF ENGLISH LETTERS: ASCII

In Part A you got acquainted with the G15 PMN computer and the programming language (unless you already knew it and have jump-started into the book, eg here!). In Part B we did some pretty tough experiments with numbers, large and small, both in our imagination and at the PC in various ways. Hopefully, you got the sense that the numbers aren't fighting you but that you can make friends with them. By them, we can rule over the computers and make them ours. And all the while, we have sprinkled in some big thinking on the wonders of life and how we mustn't think of ourselves as machines. When we have sorted out these things, the computers are there to help us, and they will, indeed, be truly helpful.

Most of the time, we put some stuff to the computer as a program, and the program produced some results on the screen, and we mused over why. But in one case, we got the mouse to be useful while a program was running and we could use it to display star-like fields here and there.

In this part, Part C, we will have more such 'C' for "Communication" in which a program, while it is running, makes use of keyboard input and also mouse input. Perhaps the 'C' is also for "Contact", at least a contact with ourselves--you with yourself--while you use the PC. That's a very important form of contact. When you are good at having contact with your own thoughts and ideas and images then you become better at having contact with other people as well.

Of course, we can also wire the computers up--or use a radio wave, wireless communication or something like that, and via the computers connect to other people. That makes sense when there are things that ought to happen very fast and involves perhaps such as money, booking a place, or voting. But as an alternative to actual meeting with real other people, do we really want a machine as filter, to stand between us?

I can think of only one reason why we might want a machine to filter the communication--and that's when we don't want any much communication. But when we do want communication, going through a machine is a poor substitute.

Pardon all these long words, if you are--as the title of this book suggests--a child, reading this. But I think you know what I'm talking about: that real life has so much more to offer than machines, and we need to talk about this often, even when we learn how to handle machines.

Can you bear over with some more long words? The world is 'analog', life is 'analog'--like waves reflecting the sparkling Sun a mild afternoon--and communication is also analog. The computer is digital--either/or. Right?

And, yet more long words: the real life meeting between friends, between people who chat together, walk together, dance together, cook together, think together, run together, feel things together, talk over life together--that is all something 'immediate'--a word which means both that it happens just now, and that it happens without anything in between. You know the word? It's immediate! --that means, "It is now!". But it also can mean: it isn't 'mediated'--in other words, there is nothing that is used as a bridge, as a mediator, as a wire, as a filter. So, the 'immediate' stands together with 'the analog'. A computer is always digital and always 'mediate'. That's a heavy way of putting it, but if you remember it, you will know something of the limitations of the computer, no matter how people talk about it. They may say it is 'So immediate! Real contact! Absolutely analog and all!' but then you can think, "That's just hype! The real world, real life, is immediate and analogue and the computer is just a little part of it."

So, we can write "analogue" as well as "analog"--both ways of writing this word is correct.

Now, when you make a program, and you then start up the program and somehow communicates with yourself--that is a blending between the analog and the digital, because you are analog and you are now and immediate and then you are using the digital to focus, to concentrate--to meditate, if you like. So you may feel an intensity through that, a calmness--that's very useful when you then turn off the machines and go out and meet real people.

Let's now do some hardcore work with the PC. To begin with I'm going to produce a list, and there is no use in trying to remember the whole list--I don't know of any who do--but you may want to know where it is and what it is all about, when you're going to program later on.

I think that in one of the previous parts, I briefly mentioned that capital letter "A" is, when inside the PC, stored as a number, and that number is 65. The small "a" has--paradoxically--a larger number, 97. (There's always 32 between small and capitals in the English alphabet when stored on the computer in the normal way.) The list of all the numbers for all the normal letters and digits and more such on the keyboard, which sort of underlies most or all the work with the programming language, is called--for historical reasons--"ASCII". I think the "A" is for "American" and the "C" is for "Characters". It doesn't really matter so much--that's the name of this list, no matter what they meant to call it. (Or the "C" might have been for "Commitee", for all I know.) In fact, it's better not to know what it originally meant. For this is what it is:

ASCII (which you may pronounce 'ask-key'):

32		45	-	58	:	71	G	84	T	97	a	110	n	123	{
33	!	46	.	59	;	72	H	85	U	98	b	111	o	124	|
34	"	47	/	60	<	73	I	86	V	99	c	112	p	125	}
35	#	48	0	61	=	74	J	87	W	100	d	113	q	126	~
36	$	49	1	62	>	75	K	88	X	101	e	114	r		
37	%	50	2	63	?	76	L	89	Y	102	f	115	s		
38	&	51	3	64	@	77	M	90	Z	103	g	116	t		
39	'	52	4	65	A	78	N	91	[	104	h	117	u		
40	(	53	5	66	B	79	O	92	\	105	i	118	v		
41	)	54	6	67	C	80	P	93	]	106	j	119	w		
42	*	55	7	68	D	81	Q	94	^	107	k	120	x		
43	+	56	8	69	E	82	R	95	_	108	l	121	y		
44	,	57	9	70	F	83	S	96	`	109	m	122	z		

Full list of special numbers: 32 is blank, 13 is <ENTER>, sometimes also 10, 0 is NIL, 8 is <BACKSP>, 9 is <TABL>, 27 is <ESC>. The range 0..126 is called "7-bit Ascii". "Visible 7-bit Ascii" is 32..126 range. G15 PMN has the arrow-or-flower-like symbol at position 37 (not percent). The 'robotfont' of G15 PMN has the 'z' inversed.

So this is just a list of numbers that puts each thingy on the keyboard to one of the numbers between 32 and 126. That is to say, all the visible thingies, and blank, are given such numbers. All the other keys, <BACKSP> and <TABL> and <HOME> and <PGUP> and all those either have numbers smaller than 32 or larger than the ASCII range; there is no ASCII list that tells exactly which number such as the <F12> function key should have, for instance. But in G15 PMN they all have exact numbers, and we'll soon see how we can find out all we need about these numbers.

The ASCII list is a little bit elegant, is it not? Perhaps a little confusing is: why does the digits, such as '1', have their own numbers, such as '49'? Why can't the digits be their own numbers--0 for 0, 1 for 1 and so on? The answer is that all the numbers under 32 had roles, early on, in computing. Now only some of them have roles: <TABL> key is given 9, <BACKSP> is given 8, <ENTER> is given 13, though 10 is sometimes used together with this, or instead of, in texts. <ESC> button is 27. And 0 we use in G15 PMN to keep cards empty--empty even of space! It is what we call 'nil' or 'the nilchar'. When we press <CTR> together with a letter from B and up to Z we will usually produce numbers from 2 and up to 26, by the way.

So '32' is space. A way to remember that--you do perhaps remember the phrase "32-bit", as when we say: the G15 PMN computer is a "32-bit computer". You don't have to understand it exactly at first, but that's a phrase you've met already, isn't it? So 32-bit--that makes the number 32 big. And the biggest button on the keyboard? That's 32 in the ASCII table.

Just get used to the idea that the numbers between 33 and 126 are all visible signs on the keyboard, including the digits. And we'll look into how we can make use of such lists and how you can find out these numbers even without having such a list at hand. And this we can use next when we make a program that uses the keyboard and shows something on the screen each time we click on the keyboard--while the program is running. A most useful thing to know, for instance if you wish to build games.

All right, now that you have the list, right here at the beginning of Part C in this book, we might as well also show some quick way of going from a number to a sign and from a keypress on the keyboard to a number. All this can happen fast and fairly effortlessly by using the terminal you by now know so well--the TF (Third Foundation) terminal, that is such a essential thing in the G15 PMN world, in addition to the cards.

In fact, almost nothing is easier than to show what number--in the list you now know is called "ASCII". Just start up the terminal, and type

```
        ki
```

and as you press <ENTER> or lineshift or what we call it, you'll notice the amazing thing that the usual rectangle, or 'text marker' as we also call it, has disappeared. The PC appears to be waiting. Indeed, it is waiting. It is waiting for a key to be pressed--or 'inputted'. So to remember 'ki', you might remember "Keyboard Input". Press any key, like A, or even lineshift itself, and then type

```
        nn
```

and you'll get it! So you can then check that 1 is 49, and * is 42--as two examples. Or that by a press on <BACKSP>, we get up 8, or a press on <TABL>, we get up 9. You can even get very much higher than the maximum of 126 when you press exotic keys like <F12> and <PGUP>. Type 'ki' many times (or make a loop if you really want to explore).

So that's one way--from keyboard to numbers. Suppose we have some numbers and we wonder what they are in the list and we don't have the list at hand. Let's say, 34 and 94. All right, then you could type something like this:

```
        ^ab
        s3
```

This just creates a little text--"ab" to start with--and you store the warp to the text by using 's3'. If you recall, s1, s2, s3 and up to s9, sx stores in i1, i2, i3 and up to i9, ix. So we can show our little text by this:

```
        i3
        pp
```

In other then, to see what 34 works out to be, and what 94 works out to be, we will modify the text. In the first position of a text is its length (in this case, 2). In the next position is the first letter or digit or whatever it is. So you could use the 'ad' command to add 1. But adding 1 is a very common thing to do so there's an easy simple command for it--'up'. And adding 2, to get to the letter after that, is also very easy--that's 'u2'! So try this stuff and you'll at once get to know what 34 and 94 are:

```
        34
        i3
        up
        kl
        94
        i3
        u2
        kl
```

You remember that 'kl' is to put stuff into something? The 'lk' is to look it up again, 'kl' is the same letters in the opposite way. Now, then, you have modified the text you have the warp to in i3. Let's show it!

```
        i3
        pp
```

And now the text is completely different. What comes out now is this, isn't it: "^

So 34 is the double-quote, the ", and 94 is ^. And the terminal showed you just what's in the list earlier in the chapter. (I often call '^' for 'the hat'.)

Now imagine a programming language in which there is no such thing as a 'terminal'. You always have to store a program somewhere, then leave that area where you can type it in, and start up the whole program, and only then, after all these steps, do you get the PC to do something--but only something--and then the PC won't do more before you next time store something and repeat all the steps. Instead, here you have a willing terminal in which you can not only do one thing, but a number of things--try them out, informally, easy, gaily we might say, and you may or may not have a big program you've made loaded in. If you do have it loaded in, you can do such checks also on bits and pieces of your program and so fine-tune it, get it to be completely right. So you see how important it is with such a terminal. It's hard to believe that most of the programming languages of the 20th century didn't have any such option, but it's just so.

Remember that which we have talked about several chapters ago--namely that there always are many ways of doing something? Well, here's another way putting something, let's again say 94, to position 2, of the text you point to in i3:

```
        94
        2
        i3
        ya
```

And the 'ya' is exactly that which we used with arrays, at the end of the previous part--'ay' for look-up in arrays, and 'ya' to 'yapp' something new into them!

Now that means, doesn't it, that a text, like ^ab or, as we can also write it, &ab& (since we can use ^ when there are no blanks in a quote), is a form of array. And that makes sense, doesn't it? It is one letter after another, and these letters, in the way we now know is called ASCII, are all stored as numbers.

In any case, let's make a program that 'listens' to the keyboard and does something. Ideally, these example programs should sometimes combine things from earlier chapters, so that we get to repeat stuff we've already been through. Remember the stuff about warps to functions? Where we run a function only by having some kind of fancy warp number, that we fetched with 'ff',--and we used 'pf' to perform that function? Why don't we combine something about keyboard and something about running small programs via an array of warps to them. This is a few more cards than that which we are used to, perhaps, but it sort of combines things we've already been through--including earlier in this chapter--and I think we can handle this without introducing more commands--or perhaps, if we do, just one or two.

If, then, you could be so considerate that you cleanse the cards, as you by now probably have done several times, that start at i:1, so that we have a whole bunch of them ready for us, then we'll work through a couple of cards. When we then start the program, and press some of the many possible keys on the keyboard, we'll get a sprinkling of some messages here and there. It will almost feel like the beginning of a game. It's certainly a huge step forward when it comes to making a program that we can somehow 'connect' to!

And I'm going to promise you, that if you follow through all the chapters in this book, you'll see that eventually, in one way or another, we'll 'grow' a little game with scores and all out of the cards we type in next! We'll work out such a game before the book is complete. And you may also find, when done with this book, that you have enough programming power that you can take up some existing G15 PMN games and make new games out of them; or begin projects with entirely new games or other sorts of programs. Having said as much, let's also say that there's no harm having a look into upcoming volumes in this series before you declare youself a Pro Ultra in G15 PMN :)

So, let's make four rather simple functions. Each of them puts a nice little text on the screen just where the mouse is. They are almost identical except for the text, so you can do a <CTR>-C and press <ENTER> and then go to the next card by <PGDN> and do <CTR>-T there to put it there. Then you edit that card and save that and do a <CTR>-S and go on to produce one card after another really easily.

Under the motto, "there are always more ways to do anything", we could separate the text parts out into own functions, and put four more snappier functions together. But sometimes it is more simple to expand it a little bit.

After the four functions we list them up in an array. I'll explain how it works--we have before hinted that there is a snappier way to make short arrays, and already here we are doing it!

The program shows four bright little words in B9font when you press a, b, c, d (all lowercase).

If you click anything other than those four--Ascii 97 and up--program exits. To check the numbers we have here a new word, 'iswithin'--I'll explain all new bits after the program. The program starts automatically--that's a first! It does so by the command 'zz'--also that I'll explain.

I hope you bear over with the fact that it goes over as much as 9 cards, but it isn't much typing if you do it cleverly. And remember this is part of building a game!

So here's i:1:

```
        ashow=              bx.

        ^Top

        md

        sh
        sh
```

And i:2:

```
        bshow=              bx.

        ^Yes

        md
        sh
        sh
```
Here's i:3:
```
        cshow=              bx.

        ^Up

        md

        sh
        sh
```
And i:4:
```
        dshow=              bx.

        ^Ok

        md

        sh
        sh
```
Then some variation, some fresh typing, here is i:5:
```
        ourwarps=           ^bshow
        &1234&.             ff

        ^ashow
        ff
        1                   2
        ourwarps            ourwarps
        ya                  ya
```
I'll do a really fast explanation here of how we can use 'ya' directly after 'ourwarps'. If you leaf back to where we first did 'ay' and 'ya' we also used 'sz' to tell the size of an array. But when it is so short that it is just a handful, we can make a quote--any quote, any content will do. Here we say &1234& but &wxyz& would do just as nice. It has length 4. We want an array of length 4. So we just use a quote! And when we just use a quote, then we don't have to do 'kl' nor 'lk' stuff at all--again, if you find that chapter where we first did arrays, we did 'lk' a lot of times. But with super-small arrays, done as a quote in this way exactly, we get a snappy way of doing it all!

Now bear over with me if what I just said doesn't make much sense. We'll do more arrays of both types in this book and I'll give fresh explations then, and you can also read back. And when you have slept on something--literally

--for the mind/brain works during night with new learning, then suddenly things make more sense when you re-read it.
  Alright, the i:6 has more of the same stuff that i:5 had:

```
        ^cshow              ^dshow
        ff                  ff




        3                   4
        ourwarps            ourwarps
        ya                  ya
```

So now we have done 'ff' on all the four functions. They are all listed in our array 'ourwarps'. The 'ff' is, as you recall, 'find function'. We can do 'pf', or 'perform function', to run any one. And that we'll do when a key is pressed. To press 'a' gives Ascii 97 (see the table above) and so we substract 96 and that means a press of 'a' gives 1. To press 'b', 'c', 'd', gives 2, 3, 4. Anything else leads to program exit. That exit is done here, in i:7:

```
        gamebit=            1
        ce                  4
        ll:10000            iswithin
        ki                  n?
        96
        su                  se
        sx
        ix                  ex
```

And on i:8 the loop finishes. The 'activepause' gives a pause to the PC so that it can work out a few things in the background after you have pressed a key. The number '100' means it pauses just a tenth of a second. In that way, it gets to sort out where the mouse is (possibly it can do that without the 'activepause' as well--it depends on just how the G15 PMN PC is made, and this is the way that works on all of them):

```
        100                 ix
                            ourwarps
                            ay

                            pf



        activepause         lo.
```

And this tiny content on card i:9 autostarts the program. You could of course put it together with other stuff on the previous card, but it is such a big thing--to use this 'zz'--that I think it makes good sense that we often have it on a card on its own.

```
        &gamebit&
        zz
```

This i:9 means that we don't have to type in the name of the program to start it after we've loaded it in via 'cc'.

I have promised explanations, and I'll work through all the cards quickly--but most of it is repetition and is talked about in earlier chapters, okay?

The card i:1 has 'md' in it. That is a command that we can remember as "Let's get some Mouse Data!". On top of stack are two numbers that talk about left and right side of the mouse--whether it has been clicked or not. These we just 'shuck' away with 'sh'. Then there is the place on the screen, what we also can call 'x and y coordinates', to use a fine long expression. The idea of 'x' and of 'y' is that these talk about the dots on the screen--what we also call 'pixels'--and to remember which is horisontal (x) and which is vertical (y), try writing them in uppercase: X Y. You see there is only one vertical line there and that's the bottom part of Y. So Y is vertical!

The 'bx' then shows the little text at that x, y place. It is small so that we don't go too far to the right when the mouse is very much to the right. We can put in longer texts but then we ought to check on the x place, really.

So that's all about i:1. And i:2, i:3 and i:4 are all exactly identical except for the text.

On i:5 we make, then, an array--a list of numbers--of a really short type, only four members. It is called 'ourwarps'. Had it had a hundred thousand members, or even just 15, we would have used 'sz' and set it up neatly as in the chapter which discussed arrays in the previous part. But when it is so short it fits within a quote on a slender column in a card, then we can do it this way: we type &....& and inside the & and the & we put in as many letters or digits or other signs as we want the array to have members. In this case, the amount is 4.

Having set it up in this snappy way, we can just 'ya' something into the array and 'ay' it out again, without having to go through the 'lk' as we did in the chapter where we first talked about arrays.

All right, so what is it we put into the array? That's the first function--we quote it--and then we use 'ff'.

We have to quote it, otherwise it will just run. To find the warp of it, we must look at the name of it, and give that name to 'ff'. And that's what we do and we put it into place 1 on the array via 'ya'. That is, the little function 'ashow', which shows a tiny text on the screen where the mouse is, is put into place 1.

And so for bshow, cshow and dshow. Exactly identical. And that, then, covers both i:5 and i:6.

Then, on i:7, we set it all up--under the name 'gamebit' --because it is a bit of a game!

First 'ce', which is to 'cleanse the entire screen'.

Then there is a loop--here we set it to 1000 repetitions but usually you'll try it a few times and then press <ESC> or something to leave it--after all, it isn't a full game yet!

Then we use the command 'ki' which waits on the key to be pressed. Suppose it is 'b'. If we look up in our Ascii table, we find that it has number 98. But when we do a 'su' or 'substract' with 96, that means we get the number 2 for b. And so 3 for c, 4 for d, and of course 1 for a.

Any other key should lead to exit of program. So we check the input with 'iswithin'--'is it within so and so?'

We tell 'iswithin' that we want it to be within 1 and 4.

Then iswithin does it work--though we could do it with 'lt' and 'gt' and such, that we looked at in one of the earliest chapters in the book (and that is indeed how 'iswithin' is doing it, it is one of the hundreds of very convenient pleasant words that are part of what we call 'The Third Foundation', all available at the terminal).

In case that iswithin didn't find that it was a, b, c, or d, then we want it to exit. So we put in 'n?'--which we have looked at much earlier in the book. It is like a question,--'is it not so?'. And when it is not so, then the command 'se' will allow the next simple command, 'ex', which exists, to run. Otherwise 'se' will jump over it!

One more thing about that card i:6. It stores the 1, 2, 3, or 4, via the command 'sx'. That stuff can then be found--inside that function--via 'ix'. Remember? The s1 has to do with i1, s5 to do with i5, sx to do with ix. Right?

So ix has the number and keeps the number inside that function. In the next card, i:8, we again pick that number up. We now know that the number is right--it isn't out of 'bounds', as we might say. It is 1, 2, 3 or 4. So we can, without making the PC go bananas, use that number with 'ay' and get something out of 'ourwarps'. And having done

so, we run the warp, we warp to the function--using 'pf'.
As said, we also let the PC relax a little bit, using the 'activepause'. That's a really good thing whenever the mouse is being used, also when it is being used together with the keyboard. It allows the timing to work out.
So 'LL:1000' starts the loop, and 'LO' completes it.
Finally, the i:9, as said, allows us to save some time when we want to start it up. Instead of having to type 'gamebit' each time, after we have typed

```
        ^i1
        cc
```

it will start 'gamebit' of its own. That's what the little command 'zz' is doing, which is usually only done at the very last card. And it is done outside of any function, naturally.
On the next card, i:10, it ought to be nothing at all. Only the 'nilchars', remember? And that tells the PC that when we reach that card, there is no more needed as for this program, and it will check whether there is a 'zz' and so start the program up.
In the robotfont, used in cards, you'll notice that the 'z' letter is as if a mirror of the 'z' in B9font. That is because we want to use this letter also to show that something is over, finished--and so the line goes like a backslash, \, rather than up,--like a dwindling sunset.

With nine cards, that's a bit of typing, and don't let it depress you whether you have to correct it once or twice or more before the program works. Remember that rule of thumb we gave much earlier in the book--the more absurd the program behaves, the easier it is to correct it!

So, are you getting it to work? When you start it, it will make the screen completely black. Then you press a, b, c or d and move the mouse and each time it will show a word, tiny and optimistic, in the position you have the mouse--at least within a fair area of the screen. When you then press eg <ESC>, the program exits. You can then type 'qu' to leave the terminal safely.
Suppose now we think of some kind of game like this: at the screen the program puts something or other. If you are able to both place the mouse at the right position and press the right key, a, b, c, d, then a word will show there and you'll score some points! Right? And then if we could let it run e.g. for two minutes so that each person who tries it gets the same time, we could compare score.

Clearly, that's near to our capacity to do that. How fancy should we do it? Let's think about it when we go through more things in this book, and take our time to get it just right.

Part C, chapter 2: ANTI-AI AND GET THE PC TO HANDLE TEXTS

I have seen an enormous number of attempts to make programs, running on a normal computer, that as if are able to 'answer questions' in natural language. Some are fancifully made; some, of course, connected to sound output devices--something which on its own is very easy to do. But the result? If you ask me, I'd say that every program that aims as making English language statements is a stinking program--if programs can, indeed, stink.

Just think of it: here we have our natural, full, living minds, learning, feeling, thinking, sensing our world, each other, ourselves, our bodies, and in this world we have this lovely, rich, naturally evolved language, which we call "English". Then comes along a bunch of programmers who probably don't do any much talking to anyone. They just sit in front of the computer all day and all night and I expect they rarely wash, and eat just junk-food, and have never read any philosophy other than, at most, such that aims putting machines above people. And they analyze and fine-analyze how sentences are built, grammar and all sorts of things. They build databases of sentences extracted from a thousand or a hundred thousand books. And they are able to get a program to 'answer' questions and 'engage in conversation'.

No, they aren't able to do that. They are able to fake language, just as illusionists are able to fake Gandalf magic. They are pretenders, tricksters. And they are taking away the fun of computing, the fun of programming, and they are threatening something of the fun of English by their mockery of English in their socalled 'natural language computation'--to which they add other phrases, which again, I would suggest, stink--such as 'machine learning' and 'artificial intelligence' or 'AI'. It is perfect nonsense, what these 'natural language programmers' are doing. They don't care one whit about what's natural. Their programs are not just clumsy, they are clumsy to nth degree because they make the computer pretend that it isn't a computer anymore, but something almost human or better than the human being--and they have never meditated, never enquired into what lies beyond memory and beyond technique, they have never had any sense of love nor of intuition or communion or actual communication with anyone in their whole professional life.

Yet companies, all over the planet, and governments, too, are hiring people to program computers in this horrible way. Do you react? Do you walk with a poster that says, "Anti-AI"? Some do. Most don't care--they say that 'these developments are inevitable, they can't be fought'. Of course they can be fought. It's just to point out how horrific it is with spoiling natural language by trying to get computers to pretend that they have what they have not, namely mind.

And to avoid using such programs, if they come on the scene, somehow.

And to switch off those news channels that speak happily about such illusionist programmes as "AI".

So, you see, there's a spirit of the rebel here, right in this book, in this writer: and all the more important, then, that this writer also has decided to be entirely constructive--and come up with something that stimulates the human mind and that allows computers to be programmed, also the computers that are parts of robots.

When we ditch the idea of "Artificial Intelligence" then we say that, instead, we can have something else, that doesn't pretend to have mind, but that can express something "mind-like"--and this "mind-like" thing comes from the programmer, that is, from you and me. And there is no mimicking of mind, no mimicking of learning. It is a first-hand thing. We can say that we express our mind or mentality through a certain type of program, when it is so that it steers robots to do types of boring, repetitive work that we don't want to burden humans with. Instead of talking of 'machine learning'--which is a paradox, because a machine is pr definition stiff in its roots, whereas learning requires something alive--we can say that we 'entrain' the robot to do certain things. You see how language is important--neither to try to mimick language inside a program, nor put wrong labels on the programming we do, and to rebel against others who couldn't are less. Because we have to care. We have to protect our minds from destruction by the types of attitudes some people may harbour, attitudes where they hate life so much that they want to put machines above humans. And even if it is attachs a lot of money to joining with gangs of the type that hypes "AI", we might stay noncorrupt, I feel, and reject this type of false money.

So, when I do robot programming, I use a set of ideas that I summarize under the phrase "First-hand Computerised Mentality" or FCM. It is first-hand--not a commitee but something done by a responsible person, directly. It involves expressing our minds, our real minds, so that we get a bit of 'computerised mentality'--which is a much milder phrase than 'artificial intelligence'--it simply says that the programmer has expressed something a bit like mind into some programs, usually as part of a robot. And it is a required part of FCM that we are making the computers, also if they are part of robots, humble and modest in their claims about what they can do. So FCM belongs to the future and is scientific, whereas "AI" is merely the false hopes of advertisement companies and bosses of bogus universities and misguided giant companies and other people who have read some science fiction and mistaken it for a recipe for how to program.

FCM makes sense--in all future--a future in which we as humans handle technology in a first-hand way and do not let technology over-power us. FCM means that we humans are in control and that, as such, living human minds are in control, and that we do not make an 'insect race' of robots that pompeously go around claiming things that are patently false.

FCM can be used for autopiloting a bus within a tunnel built just for an FCM-steered bus--but we do not, when we see the greater perspectives, put a bus or a car steered by computers loose on streets where there are living people or where there are cars driven by living people: because then all sorts of things can happen that computers

can't be programmed to tackle aforehand. FCM means care.
  And when you know all this about FCM, you can also know: there is a lot of functions built into what we call the "terminal" or "The Third Foundation terminal" or the "TF terminal" that enables you to do FCM programming--and also robot programming. Yes, the pathway is right there. Though getting really into it is a topic we only deal with briefly in this Art of Thinking series (in an upcoming volume), and more in other books to come, everything you learn now can also be put into FCM robot programming.
  And once we have spoken out against "AI" and against trying to make programs construct English sentences, we also see why the playful little program "Eliza", which is part of one of the menues of the G15 PMN platform, or "operating system", if we wish,--clearly says that it is "not AI". The program provides some pre-stored responses in a rather coincidental way, just to stimulate your own thinking.

So that was the philosophy lesson of today! But how do we, once we have understood that we don't want programs to mimick real mind, handle long texts anyway? For up until the previous chapter, every quote we looked at was a tiny one--at most a dozen letters or so. Are there easy enough ways of getting G15 PMN to come with long sentences on the screen? And to accept also much typing on the keyboard while a program is running? For sure. Clearly. And we're going to look into it next.
  We can look into it also via the motivation of making a game. For even the simplest of games deserve to have some explanations to begin with. It creates what writers call 'an atmosphere'. Perhaps also the game wishes to ask a question, such as about how fast the game is going to be played--a number can be typed in. That kind of stuff.

Right, so here we go. As long as we only work with some or a dozen cards or so at a time, it's wonderful exercise in programming to do some re-typing, so I expect the best idea is that we do some 'card cleasning' this time also--even if the game we will work out as the book is closing will use something of the cards we had in the previous chapter. (How to cleanse cards is described in one of the first chapters.)
  First, to quote a long sentence when we are going to type it into a card with only slim column means that we must write it a little bit funny--we must let the text

go over many columns, and the words will be divided up strangely when they are at the cards. But when the program is showing, the sentences will be straightened out and appear completely normal, the way you want it, on the screen.

To make a long quote, or a long text, the way G15 PMN does it, is to use this type of word:

```
        longtxt*
```

and then the text begins on the next line in the same column. It can go over several columns--over a handful of cards if you have to--and then, after the last line:

```
        *txtcomplete
```

That will have much the same result as to write something like &Friday& in that something is left on the stack. This will be a 'warp', as we say, to the text. (The clever reader may wonder--what if the text has any of these phrases inside itself--how does that work out? But in the case of the first one, 'longtxt*', that's fine, just put it in; and in the case of the final one, '*txtcomplete', this will work out in one way or another as well. It's only when the line begins with '*txtcomplete' that the text will complete. But it is easy to add a quote that contains this text, eg by ^*txtcomplete, or write one letter different and change it the instant after the whole long quote is complete.)

So how about making some kind of sentence that more or less can be a first one in our upcoming game? Try this, in your i:1 card. We'll make a word, 'gametext1', where we store the quote, after we've made it, using 'kl'. This is very similar to how we deal with arrays in the previous part, by the way, apart from the 'longtxt*' stuff and the word 'cliptrail', which I explain in a moment:

```
        gametext1=          cliptrail
        ^.
        longtxt*
        Welcome to the
         game of giggl
        igaggli! Great
         fun to come!       gametext1
        *txtcomplete        kl
```

So you see how it goes. First we make 'gametext1' and we make it to be the '^.' stuff. Remember that the hat, ^, also looks like a 'v' as in 'Variable' upside-down? And so this is a place where we can put any number, any warp. And the warp we put there is to this text, in between longtxt* and *txtcomplete. It is a sentence or two, almost a full screen-length, if we use our normal-sized fonts to show it. We put in 'cliptrail' and this does the following: it makes sure that there are no blanks at the end of the line. Such blanks tend to get in the way when we do much heavy work with long sentences and switch fonts. They can, for instance, hang onto lines that are so long that they almost touch the right side of the screen. And the PC 'objects' when stuff--even blanks--get over the edge. It's a bit simplistic that way. So a rule of thumb is, just throw in a 'cliptrail' after you have used a *txtcomplete, unless you really want those blanks for something.

Then we use 'kl' after 'gametext1' at the last two lines of the card. You remember that 'kl' has the same two letters as 'lk', but whereas 'lk' is to 'look up' what's there, 'kl' puts stuff in there. Right? So the quote, the long nice quote that could start a game, is stored there.

Anything else we should say about this? Oh yes,--one thing. When we use *longtext then it expects you to use the full length of the slim column,--that is to say, 14 digits or letters or signs of whatever kind you prefer. But how do you get to know whether you're at the 14th position, and you are working on the left column? Simple!

Method 1: click <CTR>-R and it will go to position 16. So there should be one full blank between the last letter in that column and the first letter of the next.

Method 2: click <CTR>-W and go to menu mode and click on the place you're curious about and it will tell you both the position number, the line number, and even the Ascii number of whatever is there! (So that's why, if you have ever wondered, you always see a '37' briefly arise when you start a program by a mouse-click on it--for 37 is the startup sign in the Ascii list, the flower or arrow-like sign you know).

Let's add another line, this time with a question, and we'll use some new words, among others 'prt', to show these things and to get something to be typed at the keyboard by the game-player as well.

This, then, is i:2 (I suppose you now have saved the other card to i:1):

```
        gametext2=          cliptrail
        ^.
        longtxt*
        How fast do yo
        u want to play
        ? 3=fast, 2=me
        dium, 1=slow:        gametext2
        *txtcomplete         kl
```

Since you by now know all about <CTR>-C and so on, when making new cards, I assume that it's unnecessary to point out that since this i:2 is so similar--except for the text--to i:1, the fastest way of making it is to go to card i:1, do <CTR>-C then <ENTER>, then <PGDN> to i:2, then <CTR>-T and <SPACE> to put it in there. Then fix on it and save it by <CTR>-S as usual.

By the way--in case you are writing the text inside the 'longtxt*'..'*txtcomplete' thingy in the right column, the right column is exactly as long as you want it to be for this to work out--fourteen signs.

Let's hurry up and put all this to use. To print such a line-long text, we can use such as 'bx', or 'rp', and then we use a small enough x, such as 10 or 50, rather than a big x, such as 900 or 1000, which means that the command will put it to the left of the screen. (Remember we give some of these commands an 'x' and a 'y', the position on the horisontal line and on the vertical line? And that a way to remember the sequence is that 'x' is before 'y' in the alphabet, and capital 'Y' has a vertical line in it, so the vertical position is the second of the two.)

But we'll make it faster this time, and use 'prt'. The second is a question, and for that we can use 'prtinput'. Before using these, it's a good routine to use 'prtclr', which clears the 'prt' routines so that we are sure that they begin on the top line of the screen. These are made so that they are easy to use. Just look here, at i:3:

```
        prelude=             gametext2
        ce                   lk
        prtclr               prt



        gametext1
        lk                   prtinput
        prt                  tx.
```

You see? The 'ce' is the command we now know so well to clear the screen. The 'prtclr' we have just talked about--good to do, but not necessary always. Then we fetch the long quote, we write 'gametext1' then 'lk'. So here, for long quotes as with big arrays, remember to use the 'lk', right after the place where the warp or pointer or whatever you call it is stored. Then 'prt'. As simple as that. This will show it in robotfont. If you wish it to show in b9font, just put in the word 'lush' before the first 'prt'.

Then the second line, the question about how fast the game is going to run. 'gametext2' then 'lk' then 'prt'. Pretty simple, huh?

Then we use 'prtinput'. Here anything can be typed, really, but if the game-player behaves well, she'll put in the number 1, 2 or 3. That we don't look at here, we just wanted to see how to get text out and text or numbers in, suitable for a game-start. If you like, put in this as the i:4 card, to make it start by itself:

```
        &prelude&
        zz
```

Then try it--go into the TF terminal, type 'i1' then 'cc'. In case it doesn't work, go back to the cards and reread them closely. Some people thought it was fun to call this type of thing 'debugging' because they felt it was right to call a mistake in a program for a 'bug'--but once you've heard that joke, it isn't as funny the second time, and not at all the third time. And no writer in English talks of spelling issues and such as 'bugs'. It is bad language. It's better to say, 'it has got to be corrected' and not call up the image of irritating insects. So in G15 PMN we say 'program correction'. And that's an art.

When you get the program to work, the two lines will come neatly on the screen, starting on top:

```
  Welcome to the game of giggligaggli! Great fun to come!
  How fast do you want to play? 3=fast, 2=medium, 1=slow:
  _
```

And at the third line, you can type in something. When you press lineshift, the program exits. You can then check that we did everything right with the stack in the way we talked about in a previus part--type 'f' then lineshift, then type 'nn', and it should say: 123456.

Before we finish all this, I should mention a couple of 'useful things' to know.

===>USEFUL THING TO KNOW: TO SEE WHETHER A WORD IS IN USE

When you are making new functions of three and more letters, then you want to be sure that they haven't already been given a role in the Third Foundation. And this is how you check it. Let's check, for instance, if the word 'prelude' has already been used--it hasn't--but just to be sure, I checked it this way myself before I used the word just above in this chapter. This way:

```
        ^prelude
        exists
```

And it will, in a somewhat funny way (I hope you think) tell you whether it has been used or not. If not, go ahead and use it. (To check two-letter commands, see the next 'useful thing' in this chapter.)

===>USEFUL THING TO KNOW: INFO ABOUT A TWO-LETTER COMMAND
This also works with one-letter commands, by the way. So let's see we want to see something of how the word 'ad', which adds two numbers and produces a new number, their sum, on the stack, is made. Then be prepared for something which is very cryptic-looking, until you get used to looking at it. It is the underlaying way of commenting and the underlaying way of making PMN in what is called 'Yoga6dorg G15 assembly'--to use a long and technical phrase ('assembly' meaning that it is nearest the machine, the electronics, the machine code). So here's how:

Step 1. Type this word and press lineshift:

```
        scan
```

Step 2. Type a blank and then ad: and press lineshift:

```
         ad:
```

(Be sure you type a blank before the two-letter command or it will find every word that ends with 'ad:' instead)

Step 3. The way we normally start the terminal, it has all its stuff on the F-disk, beginning with card f1. So type this and press lineshift:

```
        f1
```

Step 4. Type any amount of cards that surely is bigger than what you need, but not so big it takes up much time. Type this, and press lineshift:

```
        3333
```

Step 5. It will then search it and when it finds it, type this and press <ENTER>

```
        car
```

and you'll get up that rather cryptic stuff. But you'll see that it talks about a, b => c, meaning that it reads two numbers and produces one number. It will also say something about addition. To view more cards, press <SPACE>, to quit listing more cards, press 'q'.

Step 6. In case you think there are more places where this stuff you have just searched for can be found, type 'mmm' and it will try a search for the next place.

===>USEFUL THING TO KNOW: INFO ABOUT FUNCTION DEFINITIONS
We say 'function definition' usually when we talk of functions we make via a '=' and which have length at least

3 (it must begin with a letter but it can have digits inside it).

What you do is to use the 'scan' function as explained in the 'useful thing' just above, only that you type, for instance,

```
        iswithin=
```

when you want to search up how 'iswithin' is made.

That's all! You can even use the 'scan' function to look up words in texts that are for B9edit. But then you type 'more' instead of 'car' to view the text. And when you want to repeat the search in a text for B9edit, type four m's instead of three m's: 'mmmm' is the command then.

All right. Whether you need these 'useful things' right now or later, it is good to know that they exist. In any case, let's move now towards making a game even more. We have to have some way of showing something neat on the screen and we should also get to talk about the form of arrays called 'matrices'. That's not complicated but we need to get through it to have the ground-learning we need in G15 PMN. So why not combine the two?--and make a little matrix in order to show an outline of a spaceship or something suitable for a little game. In addition, we haven't made a program that goes to the disk cards and picks some stuff out of them. That, too, we can put into the next chapter--meaning that we can sketch the spaceship in a card and get the program to read it from it. That's more or less how games like TexasStars have been made--only that these use a somewhat bigger matrix that we are going to use, and a few two-letter commands to show them real fast. The principle is the same!

Part C, chapter 3: OUTLINE OF A SPACESHIP IN A MATRIX

Again I ask: how is it going with your programming? Are we still talking together? Is this book, if you have been so kind as to work through all the chapters up until here, still a 'friendly' book? Or have the promises of it being simple, at least fairly simple, most of the places, not been upheld? I have looked at the earlier chapters with an earnest willingness to see it as if for the first time, and I think that some parts are a bit challenging--and sometimes the words and the sentences have got too complicated to be fully the childlike flow of thinking that I want it to be. Yet, all in all, it's good with a little bit of challenges and a book doesn't run away from you, does it? You can come back to it and it may have more treasures for you at a later reading, when there are some challenges in it.

We're going to look into matrices and this is stuff out of which science fiction can be made--and yet it is really simple. First, let's bring in a quote--the only quote from that thinker that I remember, and then only faintly--from 19th century writer C.S. Pierce. He was talking about this thing called 'dimensions'. When you have an array, or a line, a series of numbers one after another, that's in a way 'one dimension'. But when you put them up so you have both rows and columns, that's 'two dimensions'. And when you open your eyes to the world around you, then, if you wish to measure the size of such as a box, you need not only its width and breadth but also its height--so that's 'three dimensions'.

So this guy Pierce pointed out that the jump from one to two dimensions is really a huge one, because, as he said, you can get such as a triangle drawn up in two dimensions, and that's a kind of relationship you simply don't get with just one dimensions.

Right? So you put dots one after another--no triangle there! But once you can have both X and Y, both a width and a breadth--or width and height, if you prefer--once you have two dimensions, you can put something near two other points but not quite on the same line. So we can think about relationships in pretty complicated ways, draw them up, once we have two dimensions. So Pierce suggested that the jump from one to two dimensions is one of the biggest jumps there is--and much bigger than when we go, for instance, from two to three dimensions.

Let's think over this for a moment. How do we see the world in front of us with our two eyes? Each eye get a slightly different impression. And this impression is in movement, in all sorts of ways--also because we always move our eyes at least a little bit. So let's imagine we made a kind of matrix of numbers of colors for one eye, and then a matrix of numbers of colors for the other eye, --one number pr color 'dot' or 'neuron' or what we call it --then these numbers have to change all the time to capture the sense of movement. We have two matrices, but there is no rule that says we can't make one grand matrix out of those two matrices--put them beside one another in some fashion.

On a computer, then, we would not be able to change the numbers 'all the time'. They could change, perhaps, 25 times a second. (When people want to use a computer to give an illusion of movement, they program it to change a picture that fast, or at least much faster than just a tenth a second--but when illusions are made via a computer they may create a sleepiness or drowsiness in the mind, so that's an approach at least this writer suggests should be avoided, as a rule of thumb.)

Alright, so a matrix--maybe big, and with big numbers--that changes many times a second: that could have something to do with three dimensions in movement, even though we have just two dimensions here. And one can work with this idea to get feelings of even more dimensions.

So how do you use a matrix to get more effects than just pictures in movement? How does a robot, when it has been programmed with what we call 'FCM matrices', put things in matrices in order to do something that to us makes sense?

Just to give you an idea how it could be done: let all the bits from a camera attach to just some columns. Then, add some rows of big numbers of how these bits fit together--for instance, are they circular? Squarish? Much like a horisontal line? Or wavish? Or a combination?

So we add lots of such numbers after some functions 'crunch' the numbers in the first rows.

Then add numbers for the sorts of things that the robot might 'expect' to have in its little world--books, or, if it makes some food, perhaps the shapes of stuff it might come across.

Then add numbers for how well the expected shapes fit with, or match with, the numbers that somehow came from the camera.

You follow? By simply adding more and more rows and having a cleverly made program around it, we can use a huge matrix for just about everything that the robot might expect to work with, and also how it should work with it. Perhaps you get the idea now--or perhaps you should work a little with matrices in this and upcoming chapters and then re-read this bit, to understand more of FCM and more of matrices.

But if you do understand it--at least a little bit--then you may also understand how some writers, not just of science fiction, but also when trying to understand cosmos as it is--the universe, you know--have suggested that, in some way or another, 'the universe is a matrix'. That's not very different from what the ancient hellenes--the ancient Greek--were saying when they said, 'the world is made of numbers'. If we then add, 'numbers in movement', and get a sense of how programs might be part of it, we do have some kind of idea of cosmos that has to do with what this writer calls 'Super-Model Theory'. A 'super-model', then, is a kind of matrix or a bit of a matrix, with some programs of some sort--but also with some liveliness that a computer (on its own) never can have. Amazingly, we can, using a programming language, get a hint of something of what a computer can't do--and will get a glimpse of that hint in an upcoming chapter, in between the other things we'll do there to get the game completed--the keyword is the name of German thinker named Kurt Goedel, who did some work early in the 20th century about just this--long before the computers came about.

Now to a bit of the game graphics. We want to have some sort of simplistic-looking spaceship of sorts. It should move in a certain way, and when it moves in one direction, then, if you can 'catch it' with your mouse and at the same time click the right letter, you'll get some score, somehow. We'll work it out.

But how are we going to draw the spaceship? Like this: go into a card and make a drawing, sort of, with the stars--the * or 'asterix' sign--and keep it blank where you don't have the stars. Here's a bit what I have in mind, and this time I haven't bothered to divide it up in two columns, so that it is easier to get at how it will look when we re-draw it on the screen by tiny dots or what we call 'pixels'. So save something like this to a card. Let's see, how about j:1? So that we can keep the program in i:1? Then the program in i:1 and up will read of that card. So something like this is what you can store to j:1:

```
                      *
                    *   *
                   *  *  *
                  *   *   *
                 *    *    *
                  *  ***  *
                  *  ***  *
                  *  ***  *
```

And try to keep it rather in the middle of the card, okay? It's easier to make the program work out neatly that way.

So, when we read in this card inside a program, we're going to convert every space there to '0' and every star there to '255'. The zero means black, and the 255 means bright green on the screen. Remember we used 255 as the maximum intensity when we made a star-like field while moving the mouse, early in the book? So 0..255 is the 'range', as we say, of the brightness on the screen--we also call that the 'tones' of the screen.

In this part we listed up all the socalled codes, or 'Ascii' codes, or numbers, for each of the signs on the keyboard. We can consult it (or we can switch to menu mode by <CTR>-W and click on the various parts of our drawing by stars and blanks), and then we find that the star symbol has number 42, and the blank, the space, has number 32.

Alright. There is a way to read a card inside G15 PMN which is called 'rr'--you can remember it as 'Read a caRd' --or, if you like to use a more complicated word for card (but which means much the same), 'Read Record'.

This expects us to tell which disk and which card, and it expects us to use numbers also for the disk. The particular numbers 'rr' should have are simply the place in the alphabet (one for A and so on). Whenever you see anything in G15 PMN that wants a 'disk number', it is a=1, b=2, c=3, and so on, including i=9, j=10, k=11, l=12. In other words 'disk number' is, then, usually 3 to 12, and has nothing to do with the Ascii list.

When 'rr' reads something, it has got to put it somewhere. For that, there is a place that we can call the 'load card', and we find it by the command 'lc'.

When we are going to make a game with a flying spaceship or something like that, we want the ship to move around as fast as can be. When we make the program go to the cards, that's typically a thing that takes more time, perhaps much more time, than when it is inside an array or something like that.

So, then, what we want to do, to check how our spaceship looks, is to read in the card by 'rr', change all the numbers into 0 and 255 and put them in an array or something--and since it has rows and columns it is the array we call a 'matrix'--and then we get the program to show that matrix as some dots on the screen anywhere where we like it. Do you think we can do all that in this chapter? I think so!

For those of you who are young and struggling to learn a certain new field, an area of knowledge--please don't be tempted to rush to a conclusion along the lines that, "This ain't my cup of tea." I'm not saying every field of discipline and knowledge is for everyone: but don't make any haste in forming such conclusions. Rather, consider this: that every field is, in a way, a language of its own,--a culture,--a kind of 'world context' or even 'way of experiencing life'. It takes time to get into it, whether that time is spent during kindergarten age or a little later or much later. It is a question of letting the seasons and the years make something, a field of knowledge or discipline, rather melt with your being--not just so you can 'think about it', but rather so that you can 'think it'. And not just so that you 'deal with it', but so that the knowledge as it were lives in your finger-tips, in your feet, in your body language. To get that familiar with any theme is possible given time and a lot of sense of play and effortless work together with occasional joys, celebrations, senses of achievement,--but above all, time, yet more time, and still more time. And why is it worth it? Because with some fields of knowledge or discipline with which you have, as it were, 'wedded', you are no longer merely flirting with life but you have some kind of depth knowledge--a depth in yourself --a contact with something in the world, not just sex, not just food, not just conversation with other people--but something that goes rather deep in you.

When you have something of depth knowledge, then it works within you so as to create a stronger personality, a bit more self-confidence,--perhaps a lot--and this, in turn, affects, positively, how you look, how you behave, in almost every other field. So it is certainly part of growing up to really get to know something not just on the surface.

And now to the program. First, we need a matrix. We need, that is to say, an array that we set up to be used not just as a line of numbers but with rows and columns. Not all that many, just enough that we can read in that space-ship and put it out again on the screen. So here's an idea of i:1--I suggest you cleanse the cards i:1 and up and then type it in:

```
        spaceship=     29
        ^.             8
        300            spaceship
        sz

        &&
        spaceship
        kl             wwyymatrix
```

Quickly, I'll explain just how this card works out. First of all, let's note that everything in the left column is exactly--absolutely exactly--as we have done it as for arrays, in the chapter in an earlier part of this book where we begun with arrays. The only point worth commenting on here is--where does '300' come in? Answer: we are going to store a card in this, and a card, as you by now know very well, has eight lines. Each line has 29 signs. Multiply 8 by 29 and you get 232. The rule of thumb is that you add a bit more than 50 when you are going to make a matrix rather than an array--and the idea of adding 50 or more is written about in comments near the word you find in the next column--namely, 'wwyymatrix'.

So, 232 plus more than 50 equals--rounded up--300. So just remember, add 50 or more when we are going to make a matrix. This is because a matrix needs to have a place where it is stored the size of the columns and the rows, and with free space for some more extra info if we want to use it. All this comes up if you scan for 'wwyymatrix=' or read about 'wwyymatrix' in the Third Foundation docs, written so that they can be opened by the B9edit editor, and part of the app 3,333,333 which is the TF app.

All right, now the next column.

You see '29' then '8' (x size, y size) and name of the matrix-to-be then the word 'wwyymatrix'. So this is a matrix meant to be used together with 'ww' and 'yy', and they are very similar to 'ay' and 'ya'. You may remember that 'ay' expects the position in the array. 'ww' expects both column and row number, otherwise it is identical. And 'ya' expects new value and position in the array, whereas 'yy' expects new value and column and row number in the matrix. The column number is the same as 'x', and the row number is the same as 'y', and we have made rules earlier on to tell the sequence--x before y since x is before y alphabetically, and 'Y' is vertical--ie, row number--because it has a vertical line in it.

I like to think about 'x' and y' rather than 'row' and 'column', because I find it easier to remember that x is horisontal and y is vertical, so from now on, I'll say 'x' and 'y' more often when we speak about matrices.

So, what the right column in the card i:1 above does is to declare that the variable 'spaceship' from now on is going to be used with 'ww' and 'yy' as a matrix, and not merely be an array. It adjusts the variable and sets up the size of x and y and such things.

Do you want to test it? To see how you can manually put somethin into a matrix and get it out again? Let's go into the TF terminal and type

```
        ^i1
        cc
```

That one card--supposing you have the famous 'nilcard' on i:2--leads to just a big 'YES' on the screen, and nothing more,--a signal from G15 PMN that it did its compilation, or what we call it, successfully.

Let's try to put some stuff into the matrix. First, let's simplify how we can go to it. It's called, as we know, 'spaceship'. And by the two lines 'spaceship' and 'lk', we get to exactly where that matrix is. The warp to it, as you know it is called.

But to get something into it and take something out of it, perhaps done several times over, means that it would be handy if we could find a shortcut, an even quicker way of getting to the matrix than to type 'spaceship' and then

'lk'. So this is one way:
```
        spaceship
        lk
        sx
```
Now it is stored in 'ix'. (We have used some of these commands, the s1..s9 and sx, before, haven't we? And these store in i1..i9 and ix.)

So all we have to do is to use 'ix' each time we want to put something into the matrix or get something out of it. First, let's put something into it. It has the same x-width as a card, and the same y-height. The x-width is 29 numbers--could be letters, when shown on a card--and the y-height is 8 lines. In a matrix, we have numbers--they can be shown as anything we like depending on the program we make around it. A matrix is a very, very flexible thing,--you control it fully via your program! It can have any numbers you like--within the plus-minus two billion range.

So let's put a number to position 5, 5. The number can be anything--1000, let's say. So try to type this into the terminal after you have loaded the card by ^i1 and 'cc', and put the matrix into 'ix' as we did just above:
```
        1000
        5
        5
        ix
        yy
```
That's how easy it is to put anything you want into any place of a matrix. Then we put a million into position 28, 7:
```
        1000000
        28
        7
        yy
```
And let's get out both values:
```
        5
        5
        ix
        ww
```
To show it,--it should say '1000'--type
```
        nn
```
Right? And the million comes out here:
```
        28
        7
        ix
        ww
```

when you type 'nn' after this.
  Pretty neat?
  Now, this is a thing we need to remember when it comes to dealing with matrices: when we say that their size is, as in this case, 29 and 8, then it normally will mean that we use positions x = 0..28, and positions y = 0..7. That is to say, a matrix position usually starts at x = 0 and y = 0 rather than x = 1 and y = 1. And so the top number for x is one less than that which we give as size, and the top number for y is one less than that which give as its size. So the top position in a matrix of size 29, 8, is usually, 28, 7.
  This is something to get used to--that stuff sometimes starts at zero, or, as we call it in G15 PMN, 'basis', rather than at one, or, as we also call the number one in G15 PMN, 'dance'. These nice words, 'basis' and 'dance, we have in G15 PMN simply because it's boring to say 0 and 1 over and over again, and because it tells something of the attitude to these numbers: 0 is 'basis' for such as a matrix, typically. And many things starts with, or has as a 'yes', the number 1--so the dance starts with it!
  And that means that, such as inside a program, you can type 'dance' instead of 1 and 'basis' instead of 0, if you like.
  So we have two 'useful things' here. Please read the second one closely, because it contains a clue as to how you can get a matrix to start at 1,1 if you like:

===>USEFUL THING TO KNOW: BASIS AND DANCE ARE NUMBERS
In G15 PMN, the word 'basis' means zero, 0, and the word 'dance' means one, 1. This is a word usage that we have come with in this programming language. It is also useful to have some numbers that are in the form of normal words, when it comes to use (explained in later chapter) of such as 'd2', 'd3' up to 'd8' and 'dh'. There are some more suchs words for numbers if you look in the Third Foundation documentation.

===>USEFUL THING TO KNOW: MATRICES OFTEN START AT BASIS
In the usual case, when we make a matrix with 'wwyymatrix' we have made it 50 or somewhat more bigger than size x times size y. So, for instance, if the matrix is going to be 500 x 1000, then we make it so that it has size 500050 or at little bigger than that, via 'sz'. And then we use x values from 0 to 499 (one less than size x) and we use y values from 0 to 999 (one less than size y).

To make a matrix that is to be used so that it starts at dance rather than basis, that is fine as long as we make the sizes bigger. For instance, to make a matrix that has x in 1..300 and y 1..500, then we make the size 301x501 plus 50 or more (instead of 300x500 plus 50 or more). And we also tell 'wwyymatrix' that the size is 301 and 501.

If you find that the last part of the last 'useful thing' is hard to remember, just remember that the 'useful thing' do exist right here, and you can just return to it when you need it.

Let's complete this program! We now want your beautiful outline of a spaceship to be put into the matrix and for this matrix to be shown on the screen, as a tiny tiny set of pixels. Let's go at it! We need these cards--and I'll talk us through them after we have them, right here. This, then, you put to i:2:

```
        getstars=           lc
        spaceship           s4
        lk
        tx

        10
        1                   ll:8
        rr                  ll:29
```

Next card, i:3, is here:

```
        i4                  m2
        lk                  m1
        42                  jx
                            yy
        eq
                            h4
        255                 lo
        mm                  lo.
```

This is, then, i:4, and we have put in a comment | there:

```
        showship=           ll:8
        | in: x y           ll:29
        s9
        sx

        spaceship
        lk
        s5
```

And, finally, i:5:

```
        ix                  m2
        m2                  m1
        ad                  i5
                            ww
                            p3
        i9
        m1                  lo
        ad                  lo.
```

So this is the stuff we need to read in the beautiful sketch you saved to j:1, and make use of the matrix you made in i:1, put it to it--that's 'getstars'--and then show it where we like on the screen as a microscopic version of your drawing (we'll see later how we can enlarge such stuff).

Before we try it out, let's go through the cards. The only entirely new thing, I think--no, we have touched it briefly early in the book--is that we use 'm1' and 'm2' instead of 'i1' and 'i2' here and there. The 'm1' and 'm2' is 'm' for 'modest', remember? :-) So 'm1' is one less than 'i1', and 'm2' is one less than 'i2'. So we get the 1..8 count changed into 0..7 count. That's all there is to that.

Let's go through the cards. Before you try them, use the opportunity to check that you typed them in entirely right --no spelling issues, nothing omitted, and such things!

When you are satisfied that it is right, here's how to check that we have got this bit of the game right. In terminal, do as usual

```
        ^i1
        cc
```

then, supposing that worked out right (if not, go back and have a look at the card number it complains about, fix it, and try again), next type

```
        getstars
```

and this will get the outline of the spaceship right into the matrix. To get a tiny tiny version of it as dots on the screen somewhere, type the place as an x number and as a y number and then 'showship', eg like this:

```
        500
        300
        showship
```

It ain't big, but it is a spaceship suitable for a tiny little game! In case the dots don't make sense, go over each and every one of the handful of cards you have typed in and see if you can spot where you have typed it differently.

And now, the explanations! You can dwell over these cards yourself and figure it out without much explanations from me. But you can also go back and look at the cards later on, when you have forgotten something of my explanation and see if you can make sense of it then.

Anyhow, here we go:

The card i:1 just sets up the matrix, and we have already spoken a bit about that. The word 'wwyymatrix' wants the x-width of the matrix, the y-height of the matrix, and then the variable where the warp to the array is stored. The array must have at least 50 more in size than necessary to handle all the x times y numbers. That's the fast summary of i:1. For more info about it, look earlier in this book about arrays, or earlier in this chapter about matrices, and, in addition, there's always some information in the Third Foundation docs.
In addition, you can use the 'scan' to look up examples of use of such as 'wwyymatrix' (eg, by scanning for just that word instead of for 'wwyymatrix=', and then you type 'mmm' for each further search). Finally, you can look up the program--or the 'source code', as we also call it--for any G15 PMN app and you'll find that many of them have matrices and some of these are well commented and have a program that is easy to read.

Let's go to i:2 and i:3, together they construct our new word 'getstars'. If you look at the bottom right part of the i:2 card and the same for the i:3 card, you get a good overview over what it does. You see the LL: twice there, in i:2, and the LO also twice, in i:3. So these are two loops--as we say--and one outside the other one. Before these two loops there is some setting up of some stuff. That's the main outline.

So why are there two loops? How does that work out? Well just think of how you read this bookpage: you spend time with one line of text, read that from left to right, then you go one step further, to the next line, then read that from left to right, and so on. So in going from left to right on one line, you are as if 'increasing the X'. In going to the next line, you are as if 'increasing the Y'. You finish each line before you go to the next. So one line is like an 'inner loop', and when it is finished, the 'outer loop'--that takes you to the next line--can get a chance of getting on.

And so we want this program to 'read' your drawing of a spaceship, which you have stored at card j:1. And it will read it one line at a time--then go to the next line--and so the outer loop, the first loop, is one line after another and that's the LL:8. The inner loop is to take all the blanks and stars that you have put into that line, and there are 29 of them all in all, so that's LL:29. Alright?

If you think this way of formulating it--'inner loop' and 'outer loop' is confusing, then don't focus too hard on it, for when you get on with programming, you'll come across examples of x and y and loops inside loops often enough that you'll figure it out, whatever words you use!

One thing here we can point out as a 'useful thing', and that is this:

===>USEFUL THING TO KNOW: Y-LOOP FIRST, THEN X-LOOP
When you are going to loop through something that has lines of letters, or rows with columns, or a matrix with x and y places, then as a rule of thumb, write LL: for the Y-size (ie, the height) first, then write LL: with the X-size (ie, the width) second.

Then, before you complete the loop-inside-a-loop with 'LO' and then 'LO' again, you can get to the matrix by using 'm1' and 'm2', as these count from basis, from 0, and up, which is often the case for most matrices.

(Otherwise use 'i1' and 'i2'.)

As for the sequence, the 'm1' concerns the FIRST of the LL: and the 'm2' concers the SECOND of the LL:. So 'm1' will have to do with 'y', and 'm2' will have to do with 'x'. As a rule of thumb, then, since you often are going to use x, then y, for most functions that wants both x and y, use then 'm2' then 'm1' (instead of 'm1' then 'm2').

In other words, then, when you have LL: twice, remember that it's isn't the normal count of first 1, then 2, rather it is first 2, then 1 (as in the case of 'm2' and 'm1'). In yet other words: remember that the sequence can be a little bit confusing when two loops are inside one another!

Right? So now we have looked at the overall way that the 'getstars' work. Let's look at the details of it.

The first thing it does is to get hold of the matrix. It has 'spaceship' then 'lk' there, and that puts the matrix warp on the stack. Then it puts it to any of the many possible places--'tx' is used, so 'jx' will bring it back. (We can use lots of other places only that when we have two loops inside one another, they use 'i1' and 'i2' so we won't store anything there.)

Then it uses the command 'rr', which we briefly mentioned earlier on--it 'Reads a caRd' (or a 'Record', if you like). And which is the card it reads? '10' and '1' is given to it. So it is card #1 in disk #10. But what is disk #10? Well, #3 is c, #4 is d, #5 is e, #6 is f, #7 is g, #8 is h, #9 is i, and #10 is, for sure, 'j'. Then #11 would be k and #12 the L-disk.

(The L-disk is sometimes used a bit differently when a G15 PMN PC connects to other G15 PMN PC's--then the L-disk may sometimes be used for these two PCs to connect. But except for such particular programs that use L-disk in such a way, the L-disk is fully free to be used for whatever you want it to be used to.)

Right, so 'rr' gets your outline of a spaceship to somewhere. To where? To 'lc', the 'load card',--or, to use a fine word for it, the 'load buffer'. (We also have a 'save card' or a 'save buffer' to write cards.)

There, the card is like one long array--starting with the first line then, after those 29 signs, comes the next line with another 29 signs, all the way up to 232 signs. So it's only natural that we want to get hold of the beginning of that 'lc' and store it to somewhere, which we can then jerk up, one sign at a time. Again, we can use several places to store this. We use 's4' and that means that it is stored at 'i4'. We also have a special command, which perhaps we haven't talked about before in this book, which is 'h4', and that 'hitches' the value of i4 one up. In other words, the command 'h4' adds one to i1 and stores it back to i4.

And now we have said more than enough about i:2, I should think! Let's quickly deal with the rest. It's going to be simpler, for it is much the same kind of thingy with small variations.

In i:3, to look at the last part of it first, we see the 'h4' as expected, and also the 'lo' twice over. But what goes on in the left column? It's a little trick, easy for a programming language like G15 PMN, but hard for most other programming languages. G15 PMN is made so that you can add and multiply and do such things with just about anything,--very few restrictions, great freedom, and more fun possibilities once you get into it.

So, in the left column of i:3, first it fetches the sign from the j:1 card, the present position. The first time, that'll be the upper left corner. That's either a space, which has Ascii number 32, or it is a star, which has Ascii number 42. Right? (Consult the Ascii table some pages earlier in this book when you want.)

So it does 'i4' and then 'lk', or 'look', at what's there. Is it 42? That is what it does next--'42' goes to the stack, then 'eq' compares and--if you remember how the 'eq' works--it produces a '1' to say 'yes' or a '0' to say 'no'. Then we multiply this value with 255! And so, when it is 42, we get it to become the brightest green possible --namely, a dot of intensity 255. In case it is blank, 0 times 255 is still 0, and so it becomes pitch black.

Satisfied with our result here, we then 'yy' it into the matrix, which is at 'jx'. In which position? Answer: m2 is the x position, m1 is the y position, so it is 'm2' first, then 'm1', as in the 'useful thing to know' just above.

That's a lot of words for very few and rather simple commands. Do you see the power of a programming language also in this way? A good programming language has in it the power of what we can call 'abstraction'--it is 'abstract' in that it just throw numbers around without bothering about meaning. And so, to work with it can for us be a good distraction from sometimes confusing events of life. A distraction that is also a meditation.

Then the next two cards, much simpler still to explain. The i:4 and the i:5 shows the spaceship, which ought to be --at this point--safely stored in that matrix which is in what we call the 'memory' or the RAM of the PC. We will put it to the screen straight from RAM--and that is something that goes very fast. But to get it from disk to RAM is something we only do each time we start up the program, once, for getting anything from the disk isn't nearly as fast as to get it from RAM. And in that way, we save up on the computer's powers so that we get a more interesting and flowing game experience.

The 'showship' has--as you see if you look at both i:4 and i:5--the same type of loop-within-a-loop as we have just discussed. This time it is a question of dots, or 'pixels', as we say, rather than signs in a card, but the numbers are the same, 29 and 8.

At i:4 we have a comment bar, a '|', and it says that it expects something 'in', and what it expects in--as 'input' we can say--is x and y. Alright. So this function doesn't put the spaceship always in the middle of the screen or on top left or something. It can put it anywhere we like. But we have got to tell where. So we tell an x, about up to 950 or so, and a y, about up to 700 or so. In this way we won't go outside of the screen limit of 1024 and 768, even when the little ship has some size.

Such a comment can be dropped and the program still works. But when you read a program with a comment line that talks about what the function has as input, and a comment line that talks, when it does leave something on the stack, about what it gives to the stack--the keyword here is often just that, 'gives'--then it is much easier to read the program later on. So it is a good rule of thumb to have comment lines in the beginning of a function --just after the 'funcname=' part--which briefly tells about what comes 'in' and what it 'gives'. This is far more important when the program goes over more than a handful of cards. Then it becomes really important to have some such map over it all.

But let's focus and get this explanation done with! The spaceship matrix is still found by 'spaceship' and then 'lk'. Just as a variation--not because it is in the slighest important--it is here stored somewhere else. It is stored via 's5' in 'i5'. But before that, the 'x' and the 'y' have got to be stored somewhere--at least, the program becomes easy to read that way. (They could of course just be on the stack if we like but then we have got to clear it up when the program finishes.)

So we use 's9' and then 'sx'. You can say it the other way around if you like, or use some other places. But I'll explain why it can make sense sometimes to use 's9' then 'sx' when you have 'x' then 'y' as input to a function:

When a function has 'x' then 'y' as input, it means that 'y' is on top of the stack. This is because, as we talked about in one of the earliest chapters in this book, that which is put in latest, is on top, and that is what we first get out of the stack. We said 'LIFO'--last in, first out, didn't we? So the 'y' is on top. And we want to put it somewhere nice. The 's9' is chosen because the '9' has a bit of similarity in its lines to something of 'y'. To get the 'y' value out, later in this function, we use 'i9' and we remember this because 'y' and '9' has a little bit similarity, okay?

And then, of course, comes the 'x' value, and what is more natural than to put that in 'ix' via the 'sx'? Right? So sometimes we can use 'sx' then 's9', other times we can use 'tx' then 't9', and at yet other times we use any one of the others, including such as 's3' and 's4'. The only rule is that once you have decided where to put the x and the y, stick to using that place inside that function, inside that word.

Then on i:5, finally, we put it all together. That's where the 'draw-ship-on-the-screen' looping completes. Just before it completes--and it's often a good thing to read a card from the bottom right and up--it calls on 'p3'. Remember that little command? It puts a point--a dot--a pixel--to the screen at place x, y and with intensity as the third thing given to it, a number of value 0 (black) up to 255 (bright green).

Alright, the intensity comes from the matrix. The matrix is in i5. Just before that we give 'm2' and then 'm1'. That's x and y, okay? And we use 'ww' because that is indeed the way to get anything out of a matrix, in the normal case. 'yy' means into the matrix, 'ww' means into it, just as--for a pure array--'ya' means into the array, and 'ay' means out of it.

On the left side of the card i:5 the place on the screen is worked out by means of two uses of 'ad', or addition, plus. The 'ix' and the 'i9' are used--and we add just how far we have come through the matrix. We add, in other words, the 'm2' or 'x' for the matrix to 'ix', and the 'm1' or 'y' for the matrix to 'i9'.

If we didn't do this addition, then either the matrix would be shown always at the top left of the screen, or the whole matrix would be shown in just one dot on the screen and so be in practise invisible. So what this function does is to say that the x and y given by the person who calls this function is to be the top left position of the spaceship. A little bit is added to it, so that each of the little dots get spread out over a 29 x 8 area.

And that's it! So when we run this program, we type in 'getstars', and the matrix get full of the stuff, and then we type in an x and then an y--eg 100 and 100, or 500 and 300, or 600 and 200,--and then the word 'showship'.

If you find all this a little bit complex, I can promise you that it gets simpler as you work with matrices a bit more. For it is really the same kind of thing over and over again--the few additional commands that do exist to handle matrices, such as 'w9', all build on just this what we have talked about here. In the last chapter of this book we'll put the whole game together and use pretty much the cards we have here and we'll very briefly talk about how it works there, too--and you'll also see, and if you type it in, experience the clarity that comes together with typing the matrix program into the CAR editor--so that all in all you'll have a chance of getting much more understanding of it.

Part C, chapter 4: HOW TO SOLVE IT, AND LARGER MATRICES

How do you solve anything in life?

I like the question because it's obviously too giant, too organic, too dancing to allow for any concrete answer, any definite set of steps or method. It is, in other words, a question that appeals to you as a human being and it doesn't relate to you as if you were a machine or something--rather, the question relates to you as a 'flame of philosophy', a question that can have an intensity like a flame, a living question.

And so we can have a dialogue--with ourselves, with each other, when we put a question like that and let it float in the air, let it be suspended, let it hang there and appeal to our intuitions.

In the word 'educate', there are some Latin roots, and they could be translated like this: 'to draw out'. And so, in a classroom, at a school, perhaps you are there to sort of 'remember' what you already somehow know, deep inside. The teacher helps you to 'draw out' this knowlegde of yourself.

To be able to really use your mind means that you are willing to actually keep an open mind and apply fresh thinking to things--which is different than just repeating some words as a 'standard answer', given certain questions. It can also be called, 'to improvise'.

When we program, and the program has to be corrected, how we handle that situation can teach us something about how we can handle other challenges in life,--I think. At least --if we find in ourselves some ways to correct programs, also when the program is hard to correct,--we may get an inspiration about relating to other things in life in more or less a similar way. Of course there are enormous differences between programming and mostly everything else --but when it comes to fixing a program, it may have, after all, great similarities, with how we can solve anything at all.

Take one rule of thumb: when you are going to solve something, and it appears that no matter what you do, it still has got to be solved, then divide things up and go through one thing at time, and find out also how important each thing is.

If a program doesn't work, it may have something in it that is rediculously easy to fix once you spot it, but perhaps you haven't looked at the right place. You may have thought that the first card or cards have been perfectly tested in all ways when they haven't. There may be a tiny little thing there that has a big effect later on. But then there may be more such little things. So you need to divide it up. It is not about kicking the computer. Or say that programming is wrong, or the language is wrong. That 'all or nothing' attitude is rarely right when it comes to solving a program problem, and it is also rarely right when it comes to solving a life problem.

Divide things up! And separate what's most important to look at now, from that which certainly can wait. Don't demand a solution to absolutely everything at once. Something surely is more important than something other. And there may be a handful of solutions ready for you once you begin looking for them. And more will come once you start getting them realised.

Be creative about these solutions. When you look to the side of normal ways of thinking as to solving things, you may find, in those sideways, that new solutions open up.

Yet sometimes it is most right to keep on the narrow path and be patient and work it out from within there, not escaping from the main path.

In working with G15 PMN, you are working with numbers, but these numbers sometimes have names--words for them, and sometimes one sequence of them is right and the opposite sequence of them is wrong. Take, for instance, when you set up card i:1 in the chapter just before this, the one where you make a matrix. Suppose you write '8' and '29' instead of '29' and '8'. Initially, nothing appears to be wrong. The size of the matrix is still fine--8 x 29 plus 50 is the same as 29 x 8 plus 50. Both are within 300. So that's a tiny thing. Aristotle once said, "A small error in the beginning grows to a big error eventually". Now that isn't correct for all small errors in the beginning, but in the case of the spaceship, you'd better get the x and the y right!

In case the matrix gets twisted in its definition, it will produce some graphics, but not the right graphics. G15 PMN is great in giving you freedom to do anything you really want--it doesn't impose lots of heavy-handed restrictions on you as a programmer. It has been made as a stage with a lot of openness in it, but with fixed limits only as for the limits of that stage. On that stage, once you're on it, you can dance as you like, if only you stick to some essential rules.

So, with G15 PMN, you have this enormous freedom in how to structure things, but it also means that you will have to check the little structures you make--say, each couple or handful cards you make for a program, you check as best you can. Not only check that the PC doesn't protest when you do the 'cc' thing and compile the cards, but you check also that the cards seem to really work.

And when they don't, don't be too sure that things were defined right even when you hastily look at a card and assume that 'it has got to be right'. Don't look hastily. Look slowly. When you are going to solve anything, divide up what you are going to look at--then look playfully, dwell, take pauses, ask questions, don't rush on and so by rushing avoid seeing the obvious things. The obvious things may sometimes be hard to see just because they are obvious. They are so obvious they 'melt' into the environment, they are so obvious they 'cannot' be wrong--but they can be wrong, sometimes. And so you need a kind of playful, self-critical, and critical, attention to these things. That includes the earliest cards when you define something that later behaves funnily.

And you go through each card. How do you check whether something like 'getstars' work, if you suspect that it doesn't? You look at it and it looks fine but maybe you're overlooking something. And so you want to test it. How do you do that?

A way to do it is this: first, make extra space inisde the function. To make extra space normally means that you copy a bunch of cards 'to the right', and then erase that which is double, in some clever way. (In some of the earliest chapters in this book, where we discuss <CTR>-C and <CTR>-T, we talk of how to insert space for extra cards in between others. That's the same type of thing we want.)

So, having made extra space--let's say, inside the loop, after the LL: and before the LO,--then you could try and put some numbers to the screen. The clue is that you do this without upsetting the stack. So you could put in, for instance, 'f' then 'nn' at a spot. That would allow you to see a certain part of what goes on. When you do such testing, you are also doing what I like to call 'probing'. So I often insert a comment line, |, with the word, "probe", put in those parts--so I know what to remove when I'm done with the probing, the checking.

When you probe a function, you must sometimes pay attention to not get too many things to the screen. This matters much when it's inside a loop. You may want to have for instance 29 numbers shown--for the first line of the card--rather than all 232 of them. So then you can also modify the LL:8 so it says LL:1 instead. That's one way of doing it. In the name of probing, all such little modifications are okay! But put in a line like "|probe" so you can easily modify it back again.

Well, that's enough about program correction at this point. Do you still have the cards j:1 with your outline of the spaceship, and i:1 to i:5 inside the PC? How about showing the ship in a completely different, and larger way? There is an easy way of doing this, with some new words, but they are easy to explain since you're now quickly becoming such an expert on G15 PMN matrices. So I'll show you how you can make card i:6 and i:7, and you can type it in and try it--it will look really nice, your ship will show in upper left corner in neat squares as part of a matrix that fills the whole screen a bit like when you start up the TF terminal each time. So here's i:6:

```
        showship3=          ll:8
        ce                  ll:29
        freshsketch
        newmatrix

        spaceship
        lk
        s5
```

And here's i:7:

```
        i2                lo
        i1                lo

        m2
        m1
        i5
        ww
        matrfield         approvesketch.
```

And to test it, supposing i:1 to i:5 are all as in the chapter just before this, and your outline of a spaceship is still at j:1, type this, inside the TF terminal:

```
        ^i1
        cc
        getstars
        spaceship3
```

Did it work? If you typed it in precisely as this, it should work, and it looks neat, eh? Here's how it works:

First, the 'showship3' clears the screen--that's the 'ce' you've seen before

Then, the 'freshsketch' is the way to begin to use the way to show things to the screen in this funny way. When you see this word, you'll often spot 'approvesketch' in the same function: you see it in the next card. And that's when it is actually put to the screen. In the middle, after 'freshsketch' and before 'approvesketch', there is some drawing activity. In this case, it is by 'matrfield'. You see that on i:7 also.

The word 'newmatrix' simply makes a big block of bright green over a big part of the screen. You can drop it if you don't want to see the vertical and horisontal lines of the matrix on the screen.

Otherwise it is pretty much exactly as in the chapter before this. We store the matrix in 'i5', via 's5'. Right? That's where we use 'spaceship' then 'lk' then 's5'.

At i:7, we call 'i2' then 'i1' and that is 'x' and 'y', with the normal confusing sequence ;-)

In this case, we just want it to show at the upper left side and so we don't add anything to these numbers.

Then we go to the matrix via 'i5' which is given to 'ww' --just as in the chapter before this. This has 'm2' and 'm1', which is 'x' and then 'y', again in this slightly confusing sequence--but not confusing when you get used to it.

Now when we call 'matrfield', it uses much larger dots on the screen than the small dots that 'p3' uses. So there are fewer of them. They are from 1 to a little over 140 as x, and from 1 to 100 as y. The reason we use 'i2' and 'i1' in connection to 'matrfield', and not 'm2' and 'm1', is because this particular way of putting something to the screen doesn't start at 0, but at 1,--in other words, with 'matrfield', it starts not at basis, but at dance. (You remember now that 'm' is for 'modest', right?--and so 'm1' is one less than 'i1', and 'm2' is one less than 'i2', and in a loop, that means that m1 and m2 are from 0 and up, whereas i1 and i2 are from 1 and up.)

So that's all there is to this way of showing it!

Now which of the two ways of showing the ship did you like the most? Hands up those who liked the second way, in this chapter, the most! All right--I see lots of hands going up there--clearly a majority. It means we should stick to this way, when we make the game. It wasn't entirely clear to me which of the ways would work out more interestingly when we started doing this. This is one of the ways in which programming, while we work on the PC, is 'teaching' us something--even though we make the programs, what the PC comes up with, even when the program works perfectly, is sometimes surprising in one way or another. And so we are in a process where we improvise--that is to say, we try this, try that--but at the same time we apply logic, we apply thinking, we are, indeed, engaging in the art of thinking.

Just one thing--when you use that word 'matrfield', then know that there are at least two big alternative ways of using it. One is to let the PC keep on 'sketching' in the background, not showing anything on the screen before it is finished. The 'approvesketch' tells the PC that what it has worked out is now going to be shown. That's the fastest way when anything complicated, that may fill up a large part of the screen, is going to be shown. But when we have a game like the one we have in mind, we want just a tiny bit of the screen to be updated--and we want it to be updated at once. We don't want to wait until the whole screen is sort of 'rolled out', just to get a little space ship moved from one spot to another. And so, when we are in the final chapter where we make the game itself, you'll see that the word 'sketch' isn't used. Instead, we call a function at the start of the game that makes 'matrfield' to be very active and direct (the word is a complicated-sounding one, but it's only called once: 'nowdirectmatr'; and when it is called in the beginning of a game, then, just before the game completes, another word is used to tell 'matrfield' that it can go back to normal, and that other word is 'restoremdraw'). I'll repeat these things in the final chapter, I just wanted it to get briefly said, so you're a little prepared for it, okay? :)

The previous chapter was a very long one; let this one be a very short one, so we get a sense of well-deserved relief from all this thinking. Time to go out and have lots of fun! --And, knowing, we're progressing towards making a little game with a nice little outline of a flat spaceship that we can throw around the screen in some way. We're getting there, we're getting there. And from that to all sorts of other programs and games.

Part C, chapter 5: IMAGINING THE UNMAKEABLE SMART PROGRAM

In the next chapter, we'll move on with our game again. But in this chapter, we are going to get a glimpse of something, yes, infinite! :-) A keyword is: "Goedel".

Have you ever thought about why sometimes talking something over with someone can give so much more fresh new ideas and sense of understanding and all sorts of things like that, compared to how it is just let the thoughts swirl around in your own head?
  In other words, what is it about some conversations that makes us smarter than we usually are?

I mean, the other person, or people, may perhaps be just as confused as you may feel about the theme--whatever it is, which you want some clarity about. And then you are able to 'make clarity' by discussion, by conversation, by --to use a nice word--'dialogue'.

But why is it so? How come it works so well, when it does work? And sometimes we have to bring clarity to ourselves when we don't have anyone to go to at the very same moment we need it--and how do we make clarity then? You follow?

So whatever there is to the 'magic of dialogue'--a rule of thumb is: better than to just 'let the thoughts swirl' is to express them and look at what you express. To express a thought does something to your mind. You are suddenly able to 'see' the thought, and feel it over, and you get fresh ideas. And that is at least part of what a conversation can do. So can you have a 'conversation with yourself'?--of course you can. Surely you can. And this is not just something for old people: you can do it no matter age, and you can do it elegantly, without it looking strange. I don't mean that talking to oneself, aloud, is something we should make into a habit, and certainly not in public. But there are several things you can do to have a conversation with yourself so as to create clarity in yourself. This may concern how to make a program or any of a zillion other things.

For instance, you can write. And so I would suggest, get to 'tame' the keyboard of the Personal Computer. Let your fingers learn to type without you having to look at the keys all the time. Let your fingers find balance and dance and let them fly. And so you can type into some place-- an editor, a card, into the Eliza program in G15 PMN,-- wherever it is, you get to look at your thoughts that way. You can ask questions. You can divide things up, as we talked about in the previous chapter. You can get to new clarities, new questions, sort things out. Writing is enormously effective--just don't hypnotise yourself by your writing into believing in nonsense. Even if you are very good at writing, it doesn't mean you are getting nearer reality--even if it feels very nice and harmonious. So the writing must be self-critical, not full of emotions and sentimentality (sentimentality, that's a word we use for instance about emotions that we give too much importance to). The sentences mustn't overstate too much, nor understate too much. Rather, find the golden mean of touching reality without leaning too much to any side.

So 'self-critical' doesn't mean obsessed with saying negative things about yourself. It means being a little careful about hypnotising yourself to believe whether in outrageously positive or outrageously negative things,-- and not just about yourself, but about anyone.

So writing is one way to express your thoughts. You can also whisper your thoughts, without necessarily moving your lips--just speak them more clearly to yourself inside your mind--and in that way come into a feeling of giving attention to your thoughts. There's a great meditation in that. You can clear things up.

If you are entirely by yourself and completely upset and unable to feel at ease with thinking either in whisper or as writing, you can of course also express things in a clear voice to yourself, and in that way sort things out a little. Then you can turn to quiet whisper, or writing, as you get calmer.

There are other ways of thinking also: drawing things, painting, dancing, taking free walks without quite knowing why and where to go,--and even using some parts of your body, such as your fingers, to move through certain ideas, take you to questions and possibilities (a philosopher who taught me certain things, David Bohm, used to do a certain 'thinking with his fingers', also, quite often, during conversations with others).

When you are giving attention to yourself, then it is a deeper part of yourself that gives attention to a more superficial part of yourself. Can we put it that way?

This can intensify, especially when you also turn attention to something very beautiful. Beauty, order, harmony--these things can always teach us something new, prepare our minds for whatever we are going to do--and when we give attention deeply, intensely, it can focus itself into a sense of fresh, flowing, dancing feeling, a glowing sense of something really, really good. And so we can come to a feeling of giving attention to the flow of attention itself. Right? A very peculiar thing, we are, as it were, full of awareness, attention--it is energy that flows on and on--not confined to your body or head or any particular place--together with others present,--and flowing on and on. And this we can call 'trance'. We don't really need any drugs or anything like that to get to it, though drugs can hint of what it is all about. But drugs always produce side-effects, enormous drowsiness, or emotional stuff, for days or hours or more afterwards,

depending on how heavy the drug dose is. So drugs, unless you are careful, can be an expensive way to get a trance.
The 'purest' way, the meditative way, to get to a trance is by giving flowing attention--first to your thoughts. Then give attention to something beautiful. Finally, give attention to attention itself, and let it flow out and embrace all the world--as a sense of love. Reunion. Compassion. That's really the stuff religion is about, not the re-reading of some old scriptures as a form of boredom & stupidity: but religion in the sense of a glowing light, a sense of attention, of energy, burning happily within.

These are big words, and when we are moving towards programming--perhaps through the field of 'logic'--is there anything there that we can compare such big thoughts with? Is there anything in the field of logic that matches a bit what we spoke of as 'attention to attention'?
There is! It's called--whatever it is called--but we can call it this: 'self-reference'. To refer, or give attention, back to oneself. To give attention to oneself, to give attention to attention.

The whole theme of self-reference in logic is a funny one, that more or less began with the 20th century--at least, that's when it begun for real--and it changed the careers of many people and led to many new types of questions that also changed the course of technology and led to the development of computers. And so it is perhaps not peculiar that those fond of computer programming quite often have been very noticable as fans of meditation ever since meditation and trance and all things 'hippie' caught on in the 1960s.

Now that you can do 'compu-speak'--now that you can do some programming--you can get a powerful glimpse into what that discussion was all about, and how it lead people, and keeps on leading people, to regard living human minds as something beyond the machine. This is something that can be written new books about forever, trust me,--it's that rich and complex a theme. So I just have to jump into a bit of it, and try to make that bit as clearly as possible on its own. Okay?

Put very, very simply, when a machine, or something very much like a machine, gets to refer to itself, it gets into some kind of turmoil, some kind of paradox, some kind of break-down, and it has to do with infinity. At least, that's how it looks if we push that state of affairs and imagine it to be complete and instant and all such things.

And so this was found by the young logician Kurt Goedel in the 1920s to mean that we cannot make a set of rules out of logic. He showed that when logic attempts to handle all questions by a set of rules, then these rules always lead to self-reference and this self-reference always lead to complications for these rules. Again, these are simplified words, but they can be given more precise meanings and then they are correct--even most mainstream scientists in the 20th century admit that.

If you find all this complicated language, I agree. So we can say something similar in a language closer to our own --here is an attempt: there is no program that can tell everything there is to know about programs. We cannot, in other words, make a 'master program', or an 'oracle program', or a kind of 'wonder program', that can be used to say all about how all possible programs behave. In other words, we cannot have full self-reference in a computer. So a computer cannot give attention, fully, to itself. A program cannot give attention, fully, to itself. So a computer doesn't have mind, doesn't have--therefore-- intelligence. So "Artificial Intelligence" is not possible --right? It all belongs together, at least if you take the word 'intelligence' very seriously.

To show a bit of this is possible if we gather our ideas and jump straight into one particular form of behaviour of programs. Look at this program, for instance--it uses the 'q1' that reduces the counter by one:

```
        ourtest=
        ll:1
        q1
        lo.
```

If you have read through all of the book so far this program is easy as a piece of cake for you--at least if you remember what 'q1' does.

This program doesn't ever exit--the PC that starts it perhaps must be powered off and rebooted. The program has no exit. The LL: thingy adds 1 to the counter, q1 takes away 1, and at LO it jumps up to LL: again and so it goes.

See? So that is an example of a program 'behaviour'--the behaviour is that the program doesn't exit. And there are extremely many possible such programs.

This program, on the other hand, is one of the extremely many that, by contrast, does exit:

```
        ourtest5=
        ll:10
        i1
        nn
        lo.
```

So 'ourtest5' prints numbers to the screen from 1 and up to 10 and then the program is done and it exits. The behaviour of this program--one of the behaviours--is that it does exit.

So, a very simple question about any program whatsoever is this: does it exit or not? And it may or may not take something off the stack in so doing.

Years after Goedel's work, a guy named Alan Turing tried to show that what Goedel did was wrong--but he didn't achieve it. Yet, in the process, he very cleverly thought out almost everything about computer machines, and he even made one--not very elegant compared to today's computers, but it did some useful work in the 1940s. After Turing again, some people worked on this theme: does a program exit or not? And that thinking led more people to get to at least some of the grand things that the chap Goedel was working on decades earlier.

So let's stick to this idea: does a program exit or not?

The programs above each fits in one card. We could, if we bothered (but don't bother) put them to i:1.

Then imagine that somebody claimed that they had put together a fantastic program that can tell whether any program at all does exit. Let's say it is on the L-disk. It is a 'master' program, or 'oracle' program--the idea of the Oracle is--in Greece--that of a holy place where any question can get a deeply right answer. And the amazing thing is that by a bit of thinking we can show that this whole idea is absurd, and when do show that it is absurd or meaningless, then we are doing something a bit similar to what this chap Goedel did. Not quite the same, but somewhat similar.

So it is an experiment in thought.

The word "experiment" means that we set something up so that we can look at it, give attention to it, and learn from it, right?

The oracle program, let's imagine, is called doesitexit. It is used like this: First the program is fetched from the L-disk, then we feed it with the location of any program it is supposed to look at, and it will tell 0 or 1, where 1 means 'yes, it does exit sometime', and 0 means

of course 'no, it never exits anytime'.
  So suppose we had one of the programs above on i:1. Then we would do something like this--first, load in the master program:

```
        ^l1
        cc
```

Then feed it the i:1 card and call it and then show the result:

```
        ^i1
        doesitexit
        nn
```

So in the first case, the little program with the 'q1', should give '0', because it doesn't exit. In the second case, the program should exit after it has counted up to ten, so the result ought to be '1', because it does exit.

So far, so good--in our thought experiment. In order to show that no oracle program of this sort can be made, we cleverly set up a program that makes use of the oracle program and acts so as to do the opposite of what the oracle says. Let us see--something like this would do, I think:

```
        funprogram=         ll:1
        ^i1                 q1
        doesitexit          lo.
        n?

        se

        ex
```

So imagine that we put this program to card i:1, and then load in the oracle program from disk L, and, finally, we give this elegant little thing at i:1 as input to the program:

```
        ^l1
        cc
        ^i1
        doesitexit
        nn
```

Right. We have set up the thought experiment. We are now going to see what can happen, given that 'doesitexit' produces a '0' or a '1' and nothing else, and that '1' means that the program does exit, '0' means that the program doesn't exit. Let's look slowly at i:1 then, the 'funprogram'. It starts out by feeding itself--since it is stored at i:1--to 'doesitexit'. It does this by the ^i1

followed by the word doesitexit.
  After 'doesitexit' has finished with its work, it is either 0 or 1 on the stack. Let's first take the case of '0'. Then the command 'n?' changes that 0 into a 1. (Remember the 'n?'? We talked about it in one of the earlier chapters in this book--it can be thought about as a question, 'is it no?').
  When the n? has changed the zero into a one, then the command 'se'--which 'sees' the next command and sees to it that this is only done when it gets a '1'--will let the next command be done. And the next command is 'ex'--exit!
  So that's absolutely rediculous. The oracle program says the program won't exit, and that leads the program to exit!
  Well, then, is there any better result by imagining that the oracle program gives a '1'? Let's see.
  The '1' then gets changed by 'n?' into a '0'. The zero leads the command 'se' to ignore the next command. So the 'ex' is ignored. Over, therefore, to the second column. But there we have the loop with 'q1' that, as we saw earlier in this chapter, never exits! Again absolutely a contradiction. The oracle program says that the program shall exit, and just that leads it not to exit, at all!

Musing over this interesting situation, it means that no matter what program is on the L-disk, it either isn't correct--it produces wrong results--or it is so that it doesn't work for every program, just for some programs. And that type of thing--it doesn't do it right everywhere, only somewhere, that is exactly summed up in the title of young Goedel's work from about 1930, in his single, long word--"incompleteness". It isn't complete. It may not be wrong--this master program or set of rules or whatever--but if it isn't wrong, it isn't covering everything,--it is, in other words, incomplete.

[Note to advanced readers: you may wonder whether the program at i:1 is really exactly how it should be, since 'doesitexit' has to be loaded in first. But we wanted to show the idea of the thing. If you want to be more exact in that way, you can just copy the program at the L-disk to the I-disk and, on top of that, put the card with our little program. Then you do exactly as above.]

Now, with the benefit of hindsight--with the benefit of having had a lot more time to look at all these things--

let us dwell a little bit about the whole thing. Remember that earlier in this book we talked about how it is that something funny may arise once we say "..and etcetera?" We may have perfectly good results when we count from 1, via 2, up to 3, and up to, say 1000, with a certain type of set of rules or whatever. But if we just say, 1, 2, 3, and up--1, 2, 3, ...,--1,2,3 etc--then we are into an infinity that does really strange things, new things,--things that we cannot control in the same way.

You see that 'infinity' comes in not only when we go up and up and up but also when we ask for something to, as it were, 'embrace itself' fully.

Enough of infinity for now--let's not over-dwell on it. But let's just bring with us the sense that if the word 'intelligence' is a deep one, if it means a living and creative inborn talent we have, in our minds, in our souls and spirits, to grasp something anew and freshly--and to grasp even its own flow, its own dance, then intelligence is something a machine cannot have and so the idea that a thing made in a human factory or laboratory can have some kind of 'artificial intelligence' is a misuse of words. The program may be clever, it may work really well to do a complicated task. But let's not call it 'intelligent'.

Part C, chapter 6: INTERACTIVITY WITH KEYBOARD

In this chapter, we are going to get one step further with the game--or, more precisely, with some stuff we need for the game. We are soon there, but in this first volume of the Art of Thinking--before you launch yourself into a full study of all the upcoming volumes and other books of mine--we should touch on a couple of themes, including how to make nice curves on the screen. But to wet your thirst for the next volume, there we'll handle curves really well --and we'll use there something called the "AngelPen". That is a set of simple functions that allows you, with just a few cards, to make also pretty hefty graphics. It's a little too much to get into all that in this book, but you have, with this book, mostly all you need to get on also with AngelPen and other exciting things as well, which we then cover, as said, in Vol. 2. And I'll work to keep the language fairly simple, for most of the parts of the chapters, anyway, also in the next volumes. (A few of the chapters--but they will tell in the beginning of them --are clearly for adults, in that you must have a command of the English language more typical for those who are past their teens. But that's only for a few of the chapters, okay?)

Before we get to the game-relevant bit,--the 'coding', as we can call it, can we think together about what it takes to make a well-working program? I mean, how should we lay it out on the cards, how should we name it--how should we design it,--you follow? I don't mean technically, right now. I mean--think of yourself as an artist, also when you code. Now what kind of artwork is a good program?

Is it open? Is it closed? Is it dancing? Slender? Graceful and elegant? Is it heavy?--no, it's not heavy. You want every part of a program to be light-footed, and not dense with indechiperable complicated unnecessary stuff. It is, it has to be, slender. Not too long, not too wide. It has to have some elegance, some dance about it. Right? The more cards you make, the more likely it is that the program will be interesting also for others to look into, someday. Then, the more you have managed to divide it up into meaningful small portions, with nice uplifting and meaningful names, and just the right amount of comments to clarify here and there--the more it communicates something to others. And to yourself, too, when the program takes

time to make.
  A beautiful program is interesting to work with: beauty attracts attention quite naturally. You don't want to look too much away from beauty in daily life. And so a fit, well-trained, slender, graceful program attracts attention and that means that the programmer is doing a work that calls on more of herself in the whole working flow.
  Some might object that this idea of 'slender being more beautiful' is something that has to do with culture, it is only an idea we have, and that 'fat can be just as beautiful'. But the point about the healthily skinny body is that all its features can participate in the movement--because there are nerves, muscles, bones, and so on, not just fat that packages it all into a bubble-like form.
  In the programming world, lots of comments, lots of unnecessary variables, lots of unneccessary extras in each function--that is all about 'fat programming'--and we must avoid fat programs and make slender programs, beautiful programs.
  So in G15 PMN, it's really easy to start making new programs--new functions--you just write whatever= and keep on writing it and then write a dot . when done with it. And part of the reason it is so easy to start making new functions is to encourage you to make many small meaningful slender graceful functions as you build up a program as a whole. Then it will be as a dance scene, many beauties acting together. You can get fond of functions--yes, some of them can even be as close companions!

So to the game. We want what they call an 'interactive' program--it is active, and it is being active together with your activity--inter-active--active together, active 'in between you'. So we need to look at how we do this type of thing really well. How can we for instance read something from the keyboard that the game-player is doing with the keyboard, while not stopping the program from pushing the spaceship around on the screen? This is simple to do once we know how. We just need to use what we already know a bit cleverly.
  You may remember that we have already used 'ck'--"Check the Keyboard"--in some place where we had some kind of loop where we used the mouse. This 'ck' gives a simple '1' or '0' depending on whether a key is pressed. We used it, in some chapter or another, simply to get the program to exit when we press any key.
  The clue, to use both keyboard and the mouse to steer

stuff when a loop goes on is to combine the use of ck with something that reads the particular key pressed, such as 'ki'. But the point is this: only call 'ki'--which we can think of as "Keyboard Input"--when 'ck' has signalled that a key indeed has been pressed. Then 'ki' will pick the key up (and if the game player has pressed many keys very rapidly, the 'ki' will read one-by-one in the right way--for, as we say in tech-language, the keyboard is 'buffered'--in that way, a slow program or a slow PC will be more acceptable to a rapid typer--no keyboard presses are lost in empty space).

So you call 'ck' again and again, inside the loop, a loop which also calls such as 'md' to read from the mouse. Each time 'ck' gives '1', but only then, we call 'ki'. In that way, the loop will keep on flying and the game or whatever it is will keep on making the dazzling movements on the screen, while the program will respond, will handle the keyboard input.

Let's put this idea to a very simple test. In this pair of cards, we ignore the mouse, we simply want the program to exit when the <ESC> button is pressed--which, by the by-now-famous "Ascii" list some chapters earlier on was given the number "27"--a number which is said to be associated with "completing things".

And each time the keyboard is pressed, we want something to happen on the screen. Let's say, the screen gets blank and the Ascii number of the is put to the screen. Very simple, but it illustrates the idea.

So, if you like, put these two cards to i:1 and i:2 and have nothing (what we also call 'the nilcard') as i:3, and I'll quickly run through how it works:

```
        catchher=           ce
        sh
        ki
        sx                  ix
        ix                  makenumber
        27                  100
        eq                  100
        n?                  bx.
```

And i:2:

```
        easydoesit=         f
        dance               n?
        ll:1                se
        ck
                            ex
        se
```

```
                             q1
        catchher             lo.
```

Let's check how it works before I go through them. Just do the normal 'i1' then 'cc' at the Terminal, then type the name of the function and press lineshift. The name is, in this case, 'easydoesit'.

So each time you press a key, the display shows neatly what number it has--Ascii key for letters, digits, normal visible signs. Then it also gives numbers for function keys, most are well above 250, but lineshift, <TABL> and such are under 32. And as you press <ESC>, it shows 27, and the program is done!

And how does it work?

'Easydoesit' starts by putting 'dance'--which is the start-number '1'--on the stack.

Then the loop starts. It is the loop of the LL:1 kind that has a 'q1' in it, so it keeps on and on. In each run of the loop, 'ck' is called--keyboard checking, right?

Each time the keyboard is pressed, 'catchher' is called.

The 'se' only does the next command and only when it is '1' as input to it, so when keyboard is not pressed, the 'catchher' isn't started. Okay?

Then the 'f' makes a copy of what is on the stack. To begin with, that's surely '1'. Then 'n?' changes a '1' into a '0', or a '0' into a '1'. It means that when that which is on the stack has changed into 0, the 'ex' will get done. The 'ex' exits the program.

So, clearly, the 'catchher' takes away one number from the stack and puts a '0' there when <ESC> is pressed. Let's see how it does it--first the 'sh', which takes away one from the stack.

Then 'ki' and 'sx', meaning that the key pressed--the number of it--will be stored. The 'sx' stores it in 'ix', right?

Then '27' followed by 'eq' and then 'n?'--all after the 'ix', which brings back the number of the key pressed-- s this compares with <ESC>, which is number 27, and 'n?' switches around the result. So when it is <ESC>, it will be '0' on the stack. Which is what we want.

In the right column of i:1, the 'ix' is used again. This time, 'makenumber' is called. That is a function which is talked about in the texts for the Third Foundation--what it does is very simple: it turns the number on the stack into a quote, so that the functions, such as 'bx', that wants a quote, can be used to show numbers.

So 'makenumber' means, really, 'make a number quote'.

Then the position '100' and '100' is picked to show it with 'bx', that is the way to 'eXpress it with B9font'. The screen is cleansed by the 'ce' or 'Clear Entire screen', before 'bx' is used. So that's card i:1.

When you have tried 'easydoesit' you will notice that, on the stack, there is a number still. Type

```
        nn
```

just after running the program once and it will be a 0 there. That's because the 'ex' just exited the program without the cleansing of the stack first. How do we get the program fixed so it cleanses up the stack?

That's easy.

We'll make use of a friend of 'se', called 'd2'. It is a funny friend. For when 'se' does the next command when it gets a '1', and not when it gets a '0', 'd2' is opposite. So a way to remember what 'd2' does is this: 'distract attention away from the next two commands when it gets 1.'

You follow? Distract--ie, look somewhere else, when it has got a '1'. So d2 is about the next two lines. You can use any of d2..d8, to vary the amount of commands, and there's even a 'dh' for a 'wHole card'. Be sure that all of these, like 'se', only wants simple commands, two or three-letter functions there--no funny things like LL: or numbers or quotes or the like.

Shall we try it? Make sure i:4 has nothing on it--that i:4 is a 'nilcard', in other words--and put this to i:3:

```
        easydoesit3=         f
        dance
        ll:1                 d2
        ck
                             sh
        se                   ex
                             q1
        catchher             lo.
```

And start up the terminal, type 'i1' and 'cc' and then 'easydoesit3'. It is exactly the same program, only that this cleans up really neatly, right?

So you see, right before 'd2' was used, we dropped the 'n?'. That's the big easy way to switch between 'se' and such as 'd2': add or remove a 'n?'!

And just before the 'ex' we inserted 'sh'. You see how it all works out really nicely!

Part C, chapter 7: THE LOVELINESS OF WAVES

How important are waves to you? If you think about it, there are waves--in one ways or another--absolutely everywhere where there is life.

Waves on an ocean never cease to dazzle, right? If you pay them attention, you'll see that, responding to winds and deeper ocean currents and objects in the water, they can vary just about endlessly.

What mood do they put you in? Waves on water always tend to level out. That's a bit like music that soothes or is even slightly sad; but the waves are so grand, they put everyone in a state of something at least near the meditative.

Now there are so many many more kinds of waves. Anything that more or less has a rhythm is some kind of wave to it. You have radio waves: sound waves--the waves of dance--the waves of touching--the waves of sexuality--the waves of exercise--and the wave-like formations, so quickly broken of flames and fire--and other types of waves, such as the effects of food on your palate as you eat, and your hunger and thirst are stilled, and shifted, so that the initial hunger passes away--but some other, smaller hungers and thirsts may then arise. And there are the mental waves, the brain waves, if you like--and waves of feeling: and the telepathic waves--you name it!

No wonder then, that for as long as there has been addition and multiplication and numbers, people have thought about waves, and sought to find the 'essential wave'. You may already know a little or a lot of what is called 'trigonometry'--you know how sides of triangles of various shapes are compared. How a triangle where one angle is right or 90 degrees or how we call it--90 degrees may also be talked about as 'a quarter of a full circle', which again may be talked about as 'a quarter of two pi', or 'half a pi'--such a triangle has features such as the 'squares of each of the lengths of the shorter sides adds up to be equal to the square of the length of the longer side'. Square, as you may know, is to multiply something with itself. Going the other way is 'square root'.

So all this has been part of numerical thinking for as long as there has been humanity, I think it is right to say. And in designing anything really grand, it's good to have a sense of what comes out easily as 'essences', as 'cores', as basic knowledge.

A famous type of curve is called 'the sine curve', and equally famous, about, is the 'cosine curve'. Wanna have a look? They are very simple functions when inside G15 PMN: just give them any number between 0 and 'tp' or 'two pi', which is, the way we calculate when we do it without decimal numbers, is anywhere from 0 to 62832; and they'll produce a number anywhere from -10000 to 10000 (or, as we also say in G15 PMN, from !10000 to 10000).

These curves are usually drawn up so that at increasing vertical values, we go up on the screen. To turn around a value that goes this way to our own screen values can be done by taking 767 and substracting the y value. We can also use a ready-made function called 'yp', which does that job for us.

I have added 10000 and some more to get only positive values here: and then added some more so as to get a pleasing effect of putting the curve elegantly rather in the middle of the screen. To find the exact numbers, I experimented, changed the program, did REB many times, and gradually found numbers that produced a pleasing result.

In the next cards, you just have to change one single line--where it says 'si'--the PMN two-letter command for 'SIne wave'--into 'cs'--'CoSine wave' to vary between the two.

So here's i:1--I'll go through i:2 and i:2 afterwards:

```
plotthis=           i1
|in:into,from       100
14000               rd
ad
```

```
        s2                    i2
        50                    35
        ad                    rd
        s1                    yp.
```

And i:2--note that the line that says 'si' could say 'cs', and in that way you shift between sine waves and cosine waves. (They're pretty much the same except they start at different places in their 'waving'.)

```
        elegance=             ix
        ce                    m1
        tp                    mm
        800                   f
        rd                    |Try cs or si:
        sx                    si
                              plotthis
        ll:800                lo.
```

And finally, you can add this to i:3 (be sure i:4 is fully empty--that it has only 'nilchars'), so the program starts automatically after you have done the '^i1' and the 'cc':

```
        &elegance&
        zz
```

Does it work? Pretty neat, right? And here's the explanation:

Let's begin with 'elegance', since it is the top-level program. It does a loop, as we see, and in each loop it calls on 'plotthis'--the card before it. So we tackle i:2 first then i:1.

Step by step: the 'elegance' first calls 'ce', to Clear the Entire screen. Right? We know that from earlier chapters.

Then read the next four commands together: 'tp' is two pi, or one full circle. Then '800' and 'rd' and then the result of this is stored by 'sx' (so we can later get to it by 'ix'). Now 'rd' is almost exactly the same as 'di', except that it does 'Rounded Division'. It is a small variation of 'di', which does a little bit extra calculation (with the help of 'mo', in fact), so that it tries to get 'as good result as possible' by not always rounding down, but sometimes rounding up, if you know what I mean.

So 'tp'--one full range for the sine curve, or the co-sine curve--is chopped up into 800 parts. That's a fairly meaningful number, is it not?--given that the screen size of a typical G15 PMN PC is 1024 x 768.

It follows that LL:800 makes a loop of 800. We see the LO or 'lower' part of it to the bottom of the right column.

Alright, the rest of i:2 is really simple: get the counter value of the loop--the 'i1' we've seen so many times by now--or, rather, the 'm1' which is one less than 'i1' so it starts exactly at zero, at basis--and multiply it by 'mm' with the little chopped-up-circle part as we stored in 'ix'.

We 'forge' a copy of this value by 'f', then call the very sine wave function 'si'. This leaves a result on the stack. So now we have two values on the stack--what we gave to 'si', and what came out of it--and with these two values we call 'plotthis', and the program does what we instructs it to do on card i:1.

At i:1, then, 'plotthis' tosses these two values around quite a lot before 'yp' is called to put a pixel on the screen. Just where on the screen I fumbled with a little bit before reaching just that which we have there.

Often, when we have two numbers coming into a function, you'll see something like

```
        s2
        s1
```

Or it can start at 's3' when we have three numbers coming into a function. This is because of the 'LIFO' we talked about quite early in the book: last in, first out. So the second number is the first out and we can, for simplicity, put it into 'i2'. The first number is the last out and we can put it into 'i1'.

But here I add something to the numbers just before they are stored. It's just a convenience, you can do it in so many other ways,--you can add after you have stored them, or you can store them other places, or just shift them around on the stack and not bother about storing them if you're so inclined. There are just so many ways of making every G15 PMN program.

It is, however, good that you see some variations as to how it can be done so that you get better at reading other people's code and so forth.

The exact numbers of i:1 matters little as long as x is finally made to fit within about 0 to 1023 and y is made to fit within about 0 to 767, right? So in i:1 the numbers are divided, both of them are divided, since they are quite huge. And the hugest, of course, is the first of the two numbers, stored in i1--which is the same as is given to 'si' or 'cs'--it can go up over 60000. In dividing on 100, we get it to go up to a little over 600, and that suits our purposes.

The second number can go up to 20000 or so. In having added some thousands to it, we 'lift' the curve from the bottom of the screen. And the division here is only 35.

Then 'yp' is called, which expects x and y on the screen in the up-side down way, so that the classical sine curve, and then--if you change card i:2--the classical cosine curve--gets to be shown.

This sort of curve is used all the time when more shall we say 'chaotic' curves are made, eg with the 'angelpen' functions we'll talk about in the next volume.

It goes without saying (again) that your own learning process can get speedy improvement when you make this and that little adjustments of the programs. What I give here you can use as starting-points. Think before you change the program in each case and don't sweat over it if it behaves suddenly absolutely without apparent meaning: programs are like that sometimes. And it's not always worth it to figure out exactly why a crazy behaviour is taking place: rather, put the program back in order and then do a more careful adjustment.

Part C, chapter 8: THE BEAUTY OF FIRST-HAND PROGRAMMING

We're rapdily moving towards the last pages of this first volume and in the next chapter, the very last one, we'll have our game. It will be a somewhat longer chapter but I think we can consider all of it, or at least most of it, a repetition--apart from the game itself, exactly how it is put together. Apart from a few simple-to-explain commands, we already know all we need to know to get a simple game up and flying. And then you can modify it further to make it truly your own, if you wish.

In the next volume, a series of new exciting (I hope) themes are explored. One of them is about how to speed up something that for the PC is time-consuming when done as three-or-more-letter commands--you'll see how we can make a new two-letter command that goes several times as fast--good for games with a bit of complex graphics.

Now, as a kid, if you have spent time with this book, you have already heard me say 'first-hand' several time. And the first volume in a series called "Art of Thinking" simply have got to have something more about this theme before we've done--even if it is slightly complicated. So bear over with me a discussion about it--it'll pay off later, as you come further with all the things you are presently learning more about.

And because there's more programming than usual in the next chapter, we don't have to include any programming in this chapter. So in that sense, we can have a somewhat easier time. Just relax and climb your winged horse, your pegasus, and let it fly where it wants to.

I'll talk about 'first-hand' by first talking about what is definitely not 'first-hand', by a series of examples, not just from programming. Then we'll see this theme better together.

First, imagine a child who is so in her attitudes that every time she scratches her knee or foot or hand a little bit, she runs to an adult and let the adult do whatever is done to help healing the scratch. She has no idea how to handle the scratch herself--she never wants to look at it--and she never learns about it. Imagine that when she grows up, she still has that attitude--it starts getting a bit silly, right? That's having a 'second-hand' attitude to your own body: another's hand, a 'second hand', has to come in whenever there is anything to heal.

Then let's imagine that a child is the same way when it comes to handling disappointments. Whenever there is any even slight disappointment, the child runs away to some adult and have to have a story told or something so she forgets about the disappointment. She refuses to look at her own mind, her own feelings, her own thoughts, and so she doesn't learn about her mind, how to be more flexible and more free from expectations,--and imagine that she is still this way when she's an adult. That's having a 'second-hand' attitude to your own mind. Not very practical, right?

So the child who has a first-hand relationship to own body helps in the healing and gradually does it herself; and she who has a first-hand relationship to her own disappointments, and so forth, gradually learns how to calm herself without running away and having to listen to others all the time.

Then imagine a whole society in which nobody is really doing programming--not even those who say that they are doing 'coding' or programming--except a tiny group. That tiny group tells the rest that all they have to do is to write some words, and the computer does at once amazing things that nobody understands anything of--such as making a flashy image of a rotating human being. The little group programmed on this rotating human being, the '3d' as they call it, for years and years and it is a program as long as the thickest book you can imagine and they are telling others that if they just start up this program, they are programmers themselves. See the funny business here? See how it is fishy? All of society, when they are taught programming in such a 'second-hand' way, gradually becomes stupid. For instead of working with numbers and pixels and loops, they are handed over some words so as to call something ready-made, something cut and dried, put in a box, ready to be used but not ready to be understood.

The tiny little group have probably lost overview over just how they made their fancy programs: they, too, in the process, have become 'second-hand programmers'. So one day when something doesn't work, nobody understands anything anymore. Right? And in the meantime, instead of using computers, Personal Computers, PCs, as part of education, society has begun putting their minds--or their lack of mind--"over" to the PCs, over to the computers, small or big,--and so they have become more second-hand in all sorts of way.

So a whole society can fall apart in a thousand ways when this enormous theme of first-hand programming is ignored. The people who are in love with second-hand programming may say such as "it is more efficient if just a few makes the big programs, and everyone else just uses them". But is it so?

And so, you see, the very phrase "first-hand programming" grew out of my phrase "first-hand relationship to data", which I coined when I released the first forms of the computer language that eventually became G15 PMN (it was then called Firth, among other things).

First-hand programming, then, is what you now have opened up to, if you hadn't opened up to it before--simply by working with this book. Of course it's natural and delicious to use big programs that others have made and you can still be a first-hand programmer: but as a first-hand programmer, you will be a little sceptical to absolutely gigantic programs that contain lots and lots of stuff that probably nobody really understand excepts in bits. You will like a bit smaller programs even when they give a little more work--when they are made so that you yourself can go into every it of them and know that, at least in principle, you can get a grip on them.

All right: you can take this as a starting-point for an imagining of several alternative societies, how they develop in the coming millenia, when first-hand programming is taught to all--and when it is not practised at all--and variations in between. Read science fiction and make your own science fiction and you see the magnitude of the questions! And so, by looking ahead not just to your bodily life right now, but by looking ahead to how humanity develops over the centuries, you'll find that your decisions, your own actions, have all sorts of effects. Nothing is wiped out: even if very few actions have your name-tag attached to them, they do go into the flow of events and things are affected. The phrase for this you may have heard: 'butterfly effects'--the idea is that the tiniest little changes of wind can have enormous influence on the future's weather and, by that, also on society, and that the influence is greater the longer you look into the future. This sort of idea was worked out in a branch of thinking called 'chaos theory' and computer modelling of wheater helped a lot.

I have promised that there is no programming in this chapter and it won't be. Just one little 'useful thing' and we're done!

===>USEFUL THING TO KNOW: HOW TO CHECK IF A CARD IS EMPTY
A card may look empty if we open it, and still it may have some data in it. A very good example of this is how such as the Gem image editor stores images: it uses really huge numbers, numbers that are so big that it is out of the question that they are within the range that shows as letters and digits and such. So here's how to check if you have got a really empty card, what we call a "NIL CARD", or whether it has got something on it after all. Open the

first card of the image that the B9Edit editor is showing, the smiling girl at the beach in bikini. The first card is d:10000. Go to that card by <CTR>-L and type in 'd10000' and press <ENTER>.

Now that card could be a nil-card. But if you go to the menu-mode--press <CTR>-W and you're in menu mode--and click on the various squarish signs on that card, you see that behind those signs are really grand numbers. Each number has information about several green-tones, packed together. A real NIL-card has zero (basis), '0', in all the places. So d:10000 is an example of a card that has data and that, since it's part of the G15 PMN system, should be left as it is.

Part C, chapter 9: FINALLY, YOUR OWN GAME!

Everything in the game we are making here is discussed in the various chapters in this book, and a lot more is discussed in them, of course--and yet we'll go through the cards of the game, so that we refresh our knowledge. And this card is then small enough and simple enough and well enough explained that you can work with it and change it and make new things and eventually grow your own beautiful game and show it to the world.

****
****
MANUSCRIPT IS UPDATED AS IT IS BEING WRITTEN. STAY TUNED!
****