THE WRITING OF NODAL PROGRAMS

R. Cailliau

This note deals with the general problem facing every application programmer: "I know my problem, I know Nodal syntax, how do I write a reliable, maintainable program?"

The note has 3 major parts:

1. a presentation of program structure diagrams.
2. the development phases of a program.
3. worked examples.

The first part is unrelated to Nodal itself, but the diagrams are used in the examples and it is therefore useful to study it.

The second part is essentially a set of comments on everyday programming problems.

The third part gives fully worked-out examples. The examples have been chosen to include as many instances as possible of the common algorithms used in interactive application programs. To avoid fixation in a certain class of problems, the examples are unrelated to control of the accelerator. The main reasons are that

a) programs doing real control are usually bigger and require introduction to the equipment controlled, this being outside the scope of programming itself.
b) interactions in control programs are via the console hardware, which cannot be used on the development computer or in a course.

The general design philosophy of programs remains the same however, and is probably better visible in small examples.

CERN LIBRARIES, GENEVA
PROGRAM STRUCTURE DIAGRAMS

R. Cailliau

A Program Structure Diagram (PSD) is a graphical, language-independent way of representing the structure of a program. Although the flow of execution of a program can be deduced from a PSD, a PSD is not a flowchart. We could give a recursive description of a PSD, but for the human eye some conceptually equivalent items are best represented in diagramatically different ways. Since it would not be nice to mess up a recursive description from the beginning, we prefer an informal one.

1. Actions

There are three basic diagrams:

a) The compound action, represented by an oblong box containing a summary of what it does (not how it does it!). Example:

```
close
valves
```

b) The decision, represented by a box in two parts, the corners of the top part cut off. The top part contains an expression, the lower part, as many boxes as there are decisions on the value of the expression. The simplest form is that in which the expression has a Boolean result:

```
X < max
true: read the next value
false: error message
```
This is the equivalent of:

"if x<max. then read the next value, else error message."

Notice that the lower boxes are like compound action boxes, containing summaries of what they do. They also are labelled by the value of the expression for which they will be selected.

Consider:

![Diagram with button numbers and actions]

This diagram unit means that if the button number is, for example 3, then "display histogram" is the action chosen.

It is more complex than that of the first example. On the other hand, one may have only a single box as lower part:

![Diagram with index > maximal]

true: error & stop.

c) The iteration, represented by an octagon horizontally divided into two parts. The upper part contains a repetition condition, the lower part a summary of what the body of the loop does.

Example:

for all motors:

read the motor's description
The most common repetition conditions will look like:

- for first to last element
- while something
- repeat until something

2. Refinement

The summary of an action, as written in a box or in the lower part of an iteration or decision diagram, can be "expanded". This means more detail can be shown. Usually the expansion is a sequence of actions. This is represented by a sequence of diagrams, horizontally arranged and "hanging" from a common line, connected to the original box. Example:

Such a representation means that one can either read the top diagram to get a rough idea, or the sequence of bottom diagrams to get more refined information. The sequence of diagrams must occur in the order of execution, and must form a reasonably complete description. All the diagrams on the lower level of refinement (as one might call it) should remain on the same level of detail. If some need more refinement, they should then be expanded on a still lower level.

Notice that the vertical lines between diagrams do not mean any "flow of control", only structural binding. If one wants to deduce "flow", then it is to be found by reading a sequence in an expansion from left to right without going up or down. That gives all the "flow" at a certain level of detail.
3. **Beginning and end of refinement**

   a) A program is represented by a single compound action box containing its name.

   b) Refinement stops at the moment that further refinement of a diagram body would lead to a sequence of diagrams that would contain action summaries dependent on the programming language used. Such diagrams would not contribute to the documentational purpose of the PSD since the information that could be put in them can more easily and reliably be found on the program listing. Moreover, sometimes the implementation language does not allow to express even the three simple diagrams and the one clustering method we chose to build PSD's from.

   A PSD is there mainly for purposes of communication, i.e. it is intended:

   a) to help think clearly about the problem to be programmed,

   b) to help clarify thoughts between people working on the same problem,

   c) to document the program for later modifications.

   Therefore, a PSD must above all be readable and must not contain any details that are purely implementational. For such reasons, all names must be meaningful, and should not be abbreviated to artificial lengths imposed by some languages. Natural iteration specifications must be used, independent of how they are coded. The purpose is to specify what is going on (not how it is done).

4. **Example**: Computation of the mean value of a set of numbers

Suppose we choose to represent the set of numbers by a sequence of elements in a file, ending in the end-of-file mark. The sequence can contain any number of elements, including zero. We must accumulate the sum of the elements and also how many there are. When we find the end-of-file mark, we must output the accumulated sum divided by the number of elements. The first level of refinement is:

```
mean
value
program
```
The second is:

open the file, set sum to zero, set number to zero, read an element

while we have not read the end-of-file mark

accumulate the sum & number, read the next element

at least one element?

true: output mean = sum / number false: output message "empty file"

This is a sequence that must be read from left to right. Notice that the decision is the last action of the program. There is no need for an explicit stop-box. Only the action of the iteration box needs further refinement:

sum ← sum + element

increment number of elements

read an element (signal end-of-file if necessary)

The total PSD for this simple program then becomes:
6. What a PSD does not do

A PSD cannot show the method of using the program, i.e. it cannot be substituted for a user manual.

A PSD cannot show the reasons for structuring the program in the way represented, but alternative PSD's can be drawn to be able to discuss and select a suitable structure.

A PSD cannot and should not show such implementational details as overlays, although a good PSD can assist much in deciding where to cut out overlay segments.

Data structures used by the program cannot be described by PSD.

7. Coding from a PSD

Clearly, all kinds of language "features" will force one to code the actual program in a way different from that present in the PSD. This will most often be due to a lack of structuring statements in the language used.

In any case, one should use the texts of the PSD boxes literally as comments in the actual program. The illustrate the technique, we give below the coding for the mean value program in several languages:
1.10 % MEAN VALUE PROGRAM
1.20 DO 2
1.30 % WHILE WE HAVE NOT READ THE END-OF-FILE MARK
1.40 WHILE EOF. = 0; DO 3
1.50 % IF AT LEAST ONE ELEMENT THEN OUTPUT MEAN
1.60 IF N > 0; DO 4; GOTO 1.9
1.70 % FALSE: MESSAGE
1.80 TYPE ! 'EMPTY FILE!' !
1.90 SET IDEV = OPEN('R', 'TER') END

2.10 % OPEN THE FILE
2.20 SET IDEV = OPEN('R', 'INPUT:SYMB')
2.30 % SET SUM AND NUMBER TO ZERO
2.40 SET SUM. = 0; SET N = 0
2.50 % READ AN ELEMENT, THE END-OF-FILE INDICATOR IS ALSO SET UP.
2.60 SET EOF. = 0; ASK EL.
2.70 IF EL. = 0; SET EOF. = 1;
2.80 % THE END-OF-FILE IS INDICATED BY A ZERO ELEMENT.

3.10 % ACCUMULATE SUM & NUMBER, READ THE NEXT ELEMENT.
3.20 SET SUM. = SUM. + EL.; SET N = N + 1;
3.30 ASK EL.
3.40 % SET END-OF-FILE INDICATOR IF NEEDED.
3.50 IF EL. = 0; SET EOF. = 1

4.10 % OUTPUT MEAN
4.20 TYPE ! SUM./N
Pascal:
(Not Tested)

PROGRAM MEANVALUE (INPUT, OUTPUT);

VAR
   ELEMENT, SUM: REAL;
   NUMBER: INTEGER;
   INPUT: FILE OF REAL;
   OUTPUT: FILE OF CHAR;

BEGIN (* MEANVALUE *)
(* OPEN THE FILE (AUTOMATIC) *)
(* SET SUM AND NUMBER TO ZERO *)

SUM := 0;
NUMBER := 0;
GET(INPUT); ELEMENT := INPUT;

(* WHILE WE HAVE NOT READ THE END-OF-FILE MARK *)

WHILE NOT EOF(INPUT) DO
   BEGIN (* ACCUMULATE SUM AND NUMBER, READ NEXT ELEMENT *)
      SUM := SUM + ELEMENT;
      NUMBER := NUMBER + 1;
      GET(INPUT); ELEMENT := INPUT;
   END;

(* AT LEAST ONE ELEMENT? *)

IF NUMBER > 0
   THEN WRITE(OUTPUT, SUM / NUMBER)
   ELSE WRITE(OUTPUT, ' EMPTY FILE! ');

END.
PROGRAM MEANVALUE

REAL MEAN
LOGICAL EOFL
COMMON EOFL, ELEMENT, SUM, NUMBER, INPT

OPEN THE FILE
EOFL=.FALSE.
OUTPT=102
INPT=101
CALL OPEN(‘INPT:SYMB’, INPT, 1)

SET SUM AND NUMBER TO ZERO, READ AN ELEMENT.
SUM=0
NUMBER=0
READ(INPT, 1001, END=10) ELEMENT
1001 FORMAT(E20.5)
GOTO 20

EOFL=.TRUE.

WHILE WE HAVE NOT READ THE END-OF-FILE MARK

IF (EOFL) GOTO 30
CALL ACCUMU
GOTO 20

IF AT LEAST ONE ELEMENT

IF (NUMBER .EQ. 0) GOTO 40
MEAN=SUM/NUMBER
WRITE(OUTPT, 1002) MEAN
1002 FORMAT(1X,E20.7)
STOP

FALSE: ERROR MESSAGE.

WRITE(OUTPT, 1003)
1003 FORMAT(’/,’ EMPTY FILE!’)
STOP
END

SUBROUTINE ACCUMU

LOGICAL EOFL
COMMON EOFL, ELEMENT, SUM, NUMBER, INPT

ACCUMULATE SUM AND NUMBER, READ NEXT ELEMENT.
SUM=SUM+ELEMENT
NUMBER=NUMBER+1
READ(INPT, 1001, END=10) ELEMENT
1001 FORMAT(E20.5)
RETURN

EOFL=.TRUE.
RETURN
END
EOF
**THE DEVELOPMENT OF A PROGRAM**

**Phase 0**: IS THERE A PROBLEM?

In this phase we try to find out whether the problem has not already been solved elsewhere, whether it can be solved by changing the equipment or the method of operation, whether a computer program is really the best way of achieving our goals. We assume this phase has at least been recognized, it has been worked through, and its outcome was the necessity of a program.

**Phase 1**: STATE THE PROBLEM AND CONVINCE YOURSELF THAT A REASONABLE SOLUTION EXISTS

We are still not very far, but it is necessary to think up a good, common sense and simple problem statement such as:

"make a program to list all the prime numbers less than 1000"
"make a program to list all the prime numbers less than 1000000"
"make a program that plays chess as well as a grandmaster"
"make a program that wins at the Tercé".

It is equally important to be sure that at least one of the programmers knows all the mathematics, physics, engineering science or whatever is needed in order to solve the problem. The best programmer will not be able to write a program to solve second-order equations if he does not know or remember the algebra needed. Therefore, the solution invariably starts from the knowledge about the objects we are dealing with in the program. Phase 1 is completely disconnected from programming. Take your time to live through phase 1, draw diagrams, discuss, look up literature ... Invent names (good ones) for the things you are trying to think about.

**Phase 1.9**: Repeat Phase 1 because I am sure you skipped it.
Phase 2: DECIDE HOW THE PROGRAM WILL INTERACT WITH ITS ENVIRONMENT

This is the most important step in program development. Most programs in our environment are interactive, or have at least some important influence on some machinery or other programs. If you know from phase 1 how to optimise a certain beam parameter, you will now have to decide when to activate the program, who or what will control it, what to do with its results and at which instance. During phase 2, the whole method of interaction will be designed.

Notice that we do not discuss any part of the solution to the original problem, we only try to fit a nice box with the right handles and controls around the real solution.

Phase 3: DESIGN THE PROGRAM AND DATA STRUCTURES IN A LANGUAGE INDEPENDENT WAY

Here we first meet the constraints of computing: we will have to express the complete solution in a certain notation which is somewhat more restrictive than every-day English!

Use program structure diagrams, a program design language or any such tool that helps think clearly and precisely. Do spend a lot of time on getting the data structures right. If Phase 1 was OK, then there should be no major problem. Try to restrict yourself to the following:

- use different data structures for different things (do not try to save memory, especially not in this phase). If you are worried about data space, then either something went wrong in phase 1 or you are attempting something too big for your system.

- use only the procedure, the if-then-else or case, and the while. (for, repeat and loop are permitted variants). If, in phase 3, you find the need for a jump (goto) then do not try to "program around it": something else is wrong: you have not got the ideas right or you did not do phase 1 well enough. At this level, the appearance of a goto is an indication of faulty design or more
serious troubles. (Later, the goto may be necessary to realize
the control structures in the programming language you use, but
that is a different matter).

- take ten minutes to think of a good name for each object.

- make sure you have proper ways of initializing everything, and
also proper ways of terminating without leaving things lying
around.

Phase 4 : DIGESTION

Skipping this phase usually results in "frigging" : one thinks the
program is ready and starts "testing" it. This may work for trivial ten-line
programs, but it will not do any good for complicated real-life programs.
The best method to avoid wasting time in debugging and frigging is now to
let the program (as yet not typed in, let alone run!) mature in your desk,
for a week or so, like good wine. You will then have forgotten the details,
and a fresh look at it will reveal any serious bugs (especially in design
of interactions). When you take it out of your desk, try to explain it
to somebody else in its fullest detail. This is extremely useful, since
most people refuse to understand someone else's work and will put very
relevant questions. If after ten minutes of discussion, you no longer
understand it yourself, then go back to phase 1 (if you decide to persist
at all).

Phase 5 : TRANSLATION INTO A PROGRAMMING LANGUAGE (CODING)

You have made it : it is now a trivial matter to choose a language
and hand-translate your program into that language. Preferably you
choose a language that is efficient, i.e. for which little translation
effort has to be done. Most languages in wide use today however do not
even support if-then-else and therefore you will have to translate such
structures into allowed if's and goto's. The goto's you will meet now
will be the result of translation, and are virtually harmless.
Be aware that translation always causes pollution: some of the peculiarities of the programming language will infect your way of thinking and this will influence your way of designing a program in the future.

**Phase 6: Desk and Machine Checking**

After having typed the coded program, get a clean listing. Do try not to run it (that really is somewhat like eating the forbidden fruit, especially with large programs) but study it on your desk first. Check syntax, structure, spelling. If the program is really large, then you will have designed it in parts anyway, so you can look at each part individually. Parts and methods can be tested. When you have reasonable confidence, reserve 30 minutes on the machine and make sure someone has booked it right after you: you are then safe from wasting more than 30 minutes on your first tests.
Exercise: The game Kingpin

Kingpin is a simple game for one player in which the object is to surround a chess-king which moves on a chess board. Initially, only the king is on the board, and the player has a number of pawns. At every move, the player puts a pawn on the board. Pawns placed cannot move or be taken away again. The king has to move at every turn, and moves like a normal chess king: one position at a time. To make things more interesting, the player is not allowed to see the actual position of the king, only the previous one.

The game is won if the player can "pin the king", i.e. if he can place his pawns so that the king cannot move. The game is lost if the player runs out of pawns and the king can still move, or if the player puts a pawn on the king's actual position.

Development of the Kingpin program

Phase 0: We want a program anyway.
Phase 1: There are several non-trivial problems.
   a) how do we decide to represent the board, king and pawns
   b) how do we start the thing up
   c) when the king is in a position, how do we decide what will be its next position.

Much thought should be given to a) but we will assume a high speed video terminal as the communication device between player and program. The board positions can be indicated by a dot (.), the king by a letter K, pawns by letters P and the sides can be lettered and numbered, as in Fig. 1.

![Fig.1: Board]

![Fig.2]
b) and c) are closely connected. Board positions are given by a pair of coordinates. Let us put axes on the board as in Fig. 2: every position is then specified by a pair \((X,Y)\). \(X\) should be a value of the set \((A,B,C,D,E,F,G,H)\), \(Y\) of the set \((1,2,3,4,5,6,7,8)\). Since Nodal does not allow indexing by anything but integers, we will also number the \(X\)-axis by integers 1 to 8. (Notice that this is the "pollution" effect: we are throwing away the letters in the design stage because we know our language will not handle them, although some do). Usually one ends up by seeing only those solutions allowed easily, instead of trying to keep the concepts right). Initially, we must choose the king's position at random. Later, at every move, we must find all the positions around the king that are not yet occupied by pawns, and choose one of these. For the time being, we will decide to take a random choice here as well. Suppose the king is at \((X_k,Y_k)\). Then the coordinates of all positions around him are given in Fig. 3. The positions are not numbered, and it is difficult to separate the empty ones from those that are occupied. Let us number them as in Fig. 4.

![Fig. 3](image1)

![Fig. 4](image2)

For these eight positions several things must be computed:
1) the \(X\) coordinate,
2) the \(Y\) coordinate,
3) whether it is empty or not.
Once we know how many of these eight positions are empty (i.e. are valid positions to move to), we can make a choice. If no position is empty, then the king cannot move, and the player won the game.

How do we choose the move? Suppose positions 3, 5 and 7 are empty. Then we must choose among 3 possibilities. Suppose we have some generator of random numbers, which outputs numbers between 0 and 1000 in an even distribution. Taking a number from this generator, we have about exactly 1 chance in 3 that it is divisible by 3, 1 chance in 3 that the remainder of the division by 3 is 1, and 1 chance in 3 that it is 2. Hence, if we look at the remainder of division by 3, it will be 0, 1 or 2 with nearly equal probability. (Q.1: why is it only nearly fair? Q.2: can you think of other ways of using a random number to make a fair choice out of n possibilities?)

In general, if we have n positions to choose from, and m is a random number with even distribution, then (m mod n) is a random number between 0 and n-1; (m mod n) + 1 is a random number between 1 and n.

If the outcome of this computation is 1, we choose position 3; if 2 then 5, if 3 then 7. We then find the coordinates of the chosen position and move the king there.

However, we still have not solved the problem, because 3, 5, 7 are not consecutive numbers. There are several ways of putting the numbers in a row, but let us look again at what we have to know about each position.

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>X coord.</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Y coord.</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>empty</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

... | P | P |
... 3 | P | K |
... 2 | P | P |
... 3 4 5 6 ...

Fig. 5
In vertical columns of Fig. 5 are listed: the position number, its X and Y coordinate and whether it is occupied or not. (The values correspond to the little figure at the left). To choose, say the second free position, it is sufficient to scan the last row until we find the second "true" (position is empty). This happens in the fifth column. The simple linear scan is largely sufficient in this case because we must look at at most 8 positions.

What shall we do at the borders? How, for example, shall we treat the cases where the king is in A8 or in E1? There are then less than 8 positions around the king that are still within the board limits, for the others at least one of the coordinates is outside 1 to 8 (i.e. it is 0 or 9). When we compute the coordinates of the surrounding positions, we can easily recognize such "off-the-board" positions by these coordinates. We will immediately mark them "occupied" (empty = false) in the table of Fig. 5 so that they cannot be chosen for a move.

To start everything up, we clean out the board by giving all pawns to the player. Then we choose one of the 64 positions to put the king on, at random, and start playing. During the game, we display the board to the human player, then move the king (so that the player does not know where it is), and then we will invite the player to put a pawn on the board. Subsequently, we evaluate the new state of the game as follows:

1) if the player put his pawn where the king is, he loses and the game ends.

2) compute the possible moves of the king: if the king cannot move, the player wins and the game ends.

3) if the king can still move, but the player has no more pawns left, then the player loses and the game ends.

In all other cases we repeat these actions.
Notice that the position must be examined by the right order of inspections. If the player places his last pawn on the king, it is more "natural" that he loses because he put it on the king, rather than because he runs out of pawns, hence the former must be inspected first. Similarly, if the last pawn successfully pins the king, then the player should win, not lose. Running out of pawns is the weakest condition for ending the game, and must therefore be inspected last. This problem of the right order of inspecting a number of conditions that can happen simultaneously occurs quite frequently and is usually easy to solve by comparing the conditions in pairs to find the stronger and the weaker.

Phase 2: DESIGN OF USER INTERACTION

We know now how to play a game: how to start it, move the king, and how to decide correctly when it ends. We could make a simple once-only program to play a single game, repeatedly reading pawn coordinates at every move, until the game ends. But games are addictive and people want to play more than one game (especially if they lose). Moreover, this repetition is normally useful in process control, and we will therefore introduce it here. We want our program to start by playing a game, and continue playing new games until the player announces he wants to stop.

Furthermore, we want to make a fool-proof user interface: the user should not be able to type anything that crashes the program. Accepting input from the player therefore should be done with "safety belts on" (parce-qu'un petit clic vaut mieux qu'un grand choc).

Finally, we should give the player all information he needs: a nice display of the board, the number of pawns he has left, and good prompting to tell him what to input. Only absolutely valid input should be accepted. A readable error message must be written telling what went wrong, and what was expected if some invalid input is given. The program must recover and go on in all cases except the very worst. (It is one of my favourite pastimes to type garbage into any active program. It is remarkable how many bugs one can detect that way, although the program authors are not always happy with it).
In Kingpin, we will do the following:

1) we will assume that upon activation of the program, the player wants to play. (Seems trivial, but will be very visible in the first lines of code!)

2) at the end of every game, we will correctly announce the game's result, and then ask if the player wants to play another game.

3) if he wants to stop, we say goodbye (because we are polite) and stop.

Phase 3, 4, 5 : PROGRAM STRUCTURE

It is now advisable that we reread everything, to get a more complete picture. As suggested, you should let the ideas mature a little, your program will be much better.

Clearly, the interaction is the "outermost" behaviour of the program and should be structured first. (Compare this to what you first see of any piece of machinery such as a console, a computer, a car, your television etc., as opposed to what makes it really work, like the electronics, the bits, the motor, the tuner).

The broad outline is:

initialize; (we do not know what yet!)
assume the player wants to play;
while he wants to play, play one game and then ask if he wants another game;
say goodbye and stop.

This could also have been written:

initialize;
repeat : play one game;
until he wants to stop;
say goodbye and stop.
but unfortunately Nodal does not have a repeat statement, whereas it does have a while statement. (Again a case of "pollution" of your natural ways of thinking by the programming language you have at your disposal.) The program structure diagram is in Fig. 6.

![Diagram of program structure]

**Fig. 6**: first level Kingpin.  
(the numbers indicate Nodal groups)

The program is fully described by that diagram, but not very many details are visible. It is also clear that it is completely correct. In Nodal we use groups for all the actions, and the program looks like:

1.1 DO 2  
1.2 SET WTP.=1  
1.3 WHILE WTP.=1; DO 3; DO 4;  
1.4 TYPE ! "GOOD-BYE!" !  
1.5 END

The variable WTP. is used to control looping over games. Group 2 will be used for initialisation, group 3 is the whole of a game, group 4 is where one asks if the player wants to play more games. Notice that:

1) the whole program is controlled from group 1.  
2) the program ends in group 1.  
3) its overall structure is well-visible.
We now refine each of the blocks of Fig. 6 in order to write groups 2, 3 and 4. Since we do not really know yet what to do in the initialization, we leave group 2 aside. Group 4 is relatively simple, so we turn first to group 3 which contains the whole overall structure of a game. Rereading what was said on page 18, we arrive at:

initialize a game;
repeat: display the board;
move the king;
ask where the player wants to put his next pawn and put it there;
compute the possible king positions;
evaluate the current situation;
until end of this game;
type the result of this game.

The PSD is in Fig. 7.

![Diagram](image)

Fig. 7: structure of a single game
Group 3 then is:

3.1 DO 5
3.2 DO 6; DO 7; DO 8; DO 9; DO 10;
3.3 IF not end of game THEN goto 3.2
3.4 TYPE ! RS !

Problem: RS will be the result of the game, but how do we exactly write 3.3? Group 10 tells us the result: at the end of 10 we know one of the following:

1) YOU LANDED ON THE KING, YOU LOSE!
2) THE KING CANNOT MOVE, YOU WIN!
3) YOU HAVE NO MORE PAWNS LEFT, YOU LOSE!
4) not the end of the game.

The first three results end the game, they can be returned in the variable RS by 10 as strings, and then typed in line 3.4. The fourth result is not of the nature of a string and should be returned as something else: the switching "off" of a variable NOTENDOFGAME. Choose NEOG. as the name, and initialize it to 1 in group 5. (This gives us the first line of group 5) (According to your tastes, you may prefer a variable EOG. which is the inverse of NEOG.)

The Nodal statements of group 3 do not look very nice. Notice however that it is very easy to move large parts of the program around just by interchanging statements in the lines. It is also easy to add things. Line 3.3 will look like:

3.3 IF NEOG.=1; GOTO 3.2

The GOTO, we know now, is the result of the translation of the Repeat — until in more elementary If's and Goto's.
It is good practice to indicate (by a colour line or so) which parts of the PSD have so far been worked out. We continue with the difficult stuff: group 9. Reread page 16. We must set up the table of Fig. 5 to compute the possible positions for the king's move. We will keep the information in three vectors of 8 elements: X, Y, E (the rows of the table). Since this table must exist throughout the life of the program, we create the vectors in the program initialization group 2. We also need to reference the board, which will be an 8 by 8 matrix:

2.1 DIM-INT B(8,8)
2.2 DIM-INT X(8); DIM-INT Y(8); DIM-INT E(8)

To choose one out of the empty positions around the king, it would be nice to know how many there are, since we must use that number in conjunction with the random number. We count the number of empty positions in NE. We assume the king's coordinates are remembered in the variables XK and YK:

9.11 SET X(1)=XK-1; SET Y(1)=YK+1
9.12 SET X(2)=XK; SET Y(2)=YK+1

...  
9.18 SET X(8)=XK+1; SET Y(8)=YK-1
9.2 SET NE=∅
9.3 FOR I=1,8; determine E(I).

Group 9 is so simple as not to need a PSD. However, its last line is remarkable. It is clear how we fill up X and Y, we just combine figures 3 and 4. But E is more difficult. Knowing the coordinates X(I), Y(I) of the Ith position around the king, it is natural to write a FOR-loop to inspect the Ith position and determine E(I) accordingly. But how? We make this action into a group, group 11. Before going on: notice that we introduced a counting variable I. It has no other purpose than to serve as in index temporarily. To avoid problems such variables should never be relied upon to have a certain value, except in lines or groups depending on those where they are manipulated. If they are used as counters in a for-loop, no assignment should ever be done to them inside the loop.
A position next to the king's is considered "occupied" if it falls outside the board limits, i.e. if one of its coordinates is $\emptyset$ or 9. Positions inside board limits are occupied if there is a pawn on them. The PSD of groups 9 and 11 is in Fig. 8.
The really correct contents of the No-part of the bottom if block is: "set empty(I)=true, increment number of empty positions". This again goes to show how the language you use penetrates your thinking. One should constantly be aware of this, and fight against it. In Nodal we have some trouble writing group 11. Convince yourself that the following is correct:

11.1 IF X(I)>=1; IF X(I)<=8; IF Y(I)>=1; IF Y(I)<=8; GOTO 11.3
11.2 SET E(I)=∅; RETURN
11.3 IF B(X(I),Y(I))=LP; SET E(I)=∅; RETURN
11.4 SET E(I)+1; SET NE=NE+1

Again, all RETURN and GOTO statements are the result of translation of the PSD into Nodal. Notice that there is no correct way to rewrite 11 in a shorter fashion.

In 11 we examined the board for the first time. The value of LP indicates a pawn, its exact form is decided on later. Evaluation of the situation:

10.1 IF (BX1,Y1)=LX; $SET RS='YOU LANDED ON THE KING, YOU loose!'; SET NE=∅; RETURN
10.2 IF NE=∅; $SET RS='THE KING CANNOT MOVE, YOU WIN!'; SET NE=∅; RETURN
10.3 IF NP=∅; $SET RS='YOU HAVE NO MORE PAWNS LEFT, YOU LOSE!'; SET NE=∅; RETURN

The variable LX contains the value of a position with a pawn on top of the king; the variable NP is the number of pawns. This number of pawns must be decremented at each player's move, and in the beginning of each game it must be set to the number of pawns a player can use. Say we can put PW pawns on the board, then in initialize-a-game (group 5) we must write: SET NP=PW. Similarly, we must decide the value of PW in group 2, the program initialisation; for example by stating: SET PW=∅. Why not immediately set NP to 20 at every game? As an exercise, modify Kingpin so that it decides to give you more pawns on the next game if you ran out on this one, and less if you won with pawns left over!

There is only one "hard" part left: moving the king (group 7). Re-read page 17. We know: X, Y and E have been filled with the coordinates of the positions around the king and their availability. NE has the number of
empty positions. STOP! Inspect figure 7: it's not true! we compute the next positions after a move. This is all ok provided we do it once at the beginning also. This is another function of group 5: DO 9. We are safe now: we make a choice of 1 in NE, say CHC. (thus 1 <= CHC. <= NE) and then we scan E to find the CHC. th element that is true (i.e. =1). Its index is the number (according to fig. 4 and 5) of the position we ought to move the king to. The only thing left to do is to evacuate the board position XK,YK ; to update XK and YK to the new position, and to put the king in B(XK,YK).

Scanning E we do as follows: we take two variables I and J, and set them both to 0. Then we use I as index to E, incrementing it first. If E(I) is 0, we do nothing but increment I again; but if E(I) is 1, then we increment J. Every time we increment I (i.e. take the next element of E) we also look to see whether J has not yet counted up to CHC. We know this must happen, because the game would have terminated if the king could not move. CHC. is determined from the number "Random" by

\[
\text{SET CHC.}=\text{MOD(Random.NE)}+1
\]

The PSC for group 7 is in Fig. 9

---

Fig. 9: moving the king randomly.
Let LD and LK be the values for an empty position and a position with the king respectively.

The Nodal statements are:

7.1 SET CHC.=MOD(Random,NE)+1
7.2 SET J=Ø; SET I=Ø; WHILE J CHC.; SET I=I+1; IF E(I)=1; SET J=J+1
7.3 SET B(XK,YK)=LD
7.4 SET YK=X(I); SET YK=Y(I)
7.5 SET B(XK,YK)=LK

What shall Random be? The function TIME returns the number of seconds, and has milliseconds in the fraction. It will be impossible for the player to time his typing to the millisecond, so the number TIME*10 for example should be fairly random in its last few digits. Let Random be INT(TIME*10).

What is left? Initialization of the program (group 2), initialization of a game (group 5), ask player if he wants another game (group 4), display the board (group 6) and ask where he wants to put the pawn (group 8).

Parts of 2 are:

DIM-INT B(8,8)
DIM-INT X(8); DIM-INT Y(8); DIM-INT E(8)
SET PW=2Ø
SET LD=56; SET LX=113; SET LP=12Ø; SET LX=130;
TYPE ! "**** THIS IS KINGPIN ****" !!

The values of LD to LX are simply the codes for the characters '.', 'K', 'P', 'X'.

Group 5 is:

% INITIALIZE A GAME
@ =======================
5.1 SET NEOC.=1; SET NP=PW
5.2 FOR I=1,8; FOR J=1,8; SET B(I,J)=LD
5.3 SET CHC.=MOD(INT(TIME*10),64);
5.4 SET XK=INT(CHC./8)+1; SET YK=MOD(CHC.,8)+1
5.5 SET B(XK,YK)=LK
5.6 DO 9
whereby the board is first set to contain '.' in all positions. Then
the random-number generator is used to choose one position out of 64
to compute the king's initial position. Finally, the possible posi-
tions for the king's move are determined.

Displaying the board:

To generate Fig. 1, and to print the number of pawns left, the
following lines do:

```
% TYPE A FEW BLANK LINES
6.1 TYPE !!!

% TYPE 8 ROWS OF 8 SQUARES OF BOARD, WITH THEIR CONTENTS
6.2 FOR J=8,-1,1; TYPE ! &10 &1 J; FOR I=1,8; TYPE &I \B(I,J)
6.3 TYPE ! &12 "A B C D E F G H"

% FOR INFORMATION, TYPE THE NUMBER OF PAWNS LEFT.
6.4 TYPE !!! &38 &2 NP " PAWNS LEFT" !!!
```

Asking for pawn coordinates and placing a pawn:

This involves reading the player's input and analyzing it. We will
decide to accept input like A3 or D5 or C1 etc. That implies much more
than "a letter and a digit":

- one letter and one digit.
- first the letter, then the digit.
- exactly two characters, no more, no less.
- the letter must be one of 'A' to 'H', the digit one of '1' to '9'.
- all other answers are considered wrong.

The dictatorial strictness with which we filter the player's input
is the only way to protect the rest of our algorithms. The following
algorithm does all the checks:

```python
if the number of characters in the input string is not 2
then there is an error,
else if the first character is not in the set {'A' ... 'H'}
then there is an error,
else if the second character is not in the set {'1' ... '9'}
then there is an error
else the input is O.K.
```
We read the player's answer to the question "Pawn coordinates:" into the string variable ANSW.. The function SIZE(ANSW.) gives the number of characters, the function calls SUBS(1,1,ANSW.) and SUBS(2,2,ANSW.) give the first and second characters. A set of characters can be set up by initializing an array of strings, say XC for \{ 'A' ... 'H' \} and YC for \{ '1' ... '9' \}. Then FIND (XC, SUBS(1,1,ANSW.)) tells us whether the first character is in \{ 'A' ... 'H' \} and if so, even where. We have used XI and YI to remember the coordinates of the last pawn placed. Hence checking input and setting up XI and YI is done by the algorithm with PSD in Fig. 10. Two more checks are performed in that PSD: an error is signalled if the coordinate syntax is all right but there is already a pawn on the given position; and if the king is in that position then we mark the fact "pawn on top of king" by placing an 'X' there. Finally, very important, since we have put a pawn on the board, the number of pawns the player has left must be decremented.
ask pawn position and put pawn

repeat until a good pair of coordinates
  read the player's input and check it
  the king is here
    Yes: set B(XI, YI) to LX
    No: put the pawn on B(XI, YI)

decrement number of pawns

read ANSW.

length = 2 characters
  Yes: error
  No: error

first character in \{A:\#\}
  Yes: XI set, check second
  No: error

second character in \{1:3\}
  Yes: YI set, check if position empty
  No: error

a pawn on B(XI, YI)
  Yes: all O.K., a good pair of coordinates
  No: error

Fig. 10
The string arrays XC and YC are initialized in group 2:

\[
\text{DIM-STR XC; DIM-STR YC} \\
\text{FOR } I=1,8; \text{SET } XC(I)=\text{SUBS}(I,1,'ABCDEFGHIJKLMNOPQRSTUVWXYZ'); \text{SET } YC(I)=\%1 I
\]

Asking the player if he wants to play another game:

The purpose of this group is to determine whether to switch off WTP.

The treatment of player input is similar to that in group 8, but here we will allow only the answers YES or NO. PSD in Fig. 11.
4.1 $ASK "DO YOU WANT TO PLAY ANOTHER GAME?" $ANSW.
4.2 $IF ($ANSW.="YES") 4.3, 4.6, 4.3
4.3 $IF ($ANSW.="NO") 4.4, 4.5, 4.4
4.4 TYPE 1 'ANSWER WITH "YES" OR "NO" PLEASE!' !; GOTO 4.1
4.5 SET WTP.=0;
4.6 RETURN

A few concluding remarks

The careful development in levels was rather easy and straightforward.
It resulted in the creation of blocks which could be coded into groups.
The groups were absolutely independent of each other: they only cared about
their own business. To programmers used to "linear" programming, this may
seem confusing at first, because inside a group you are completely isolated
from the rest of the program. Since you know about the rest, it is sometimes
difficult to see how it will all fit together. If this story gave you the
impression of not hanging together well, it is probably because you do not
trust the method of top-down design.

A typical example of a seemingly curious block is "Move the king". It
does only what it says, and does not at all worry about whether the move is
possible or whether there are indeed NE empty positions marked in E.
These computations have been done correctly elsewhere. Structuring provides
the modularity automatically. It also produces optimized programs.

Variable list

You will have noticed it would be nice to possess a list of variables
with their meaning and nature, and the list of groups that operate them.
I have omitted making such a list because this is an example and it would
have meant repeating the list in the text after each addition to it.
Table 1 shows how one might organize such a list.

Notice that this list is not very long. Although it is possible to
"optimize" by using some variables for multiple purposes, this is very bad
practice and will invariably lead to serious bugs when the program has to
be modified.

/....
Variants

Try the following variants:

(a) adjust the number of pawns the player can use at the player's request;
(b) adjust the number of pawns to the player's strength;
(c) compute possible king moves so that the king always tries to escape to the largest unoccupied region of the board;
(d) display the board again after the final move.

Does any of these modifications imply the redesign of the program?
<table>
<thead>
<tr>
<th>NAME(S)</th>
<th>NATURE</th>
<th>MEANING</th>
<th>GROUPS THAT OPERATE THE VARIABLE(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XK, YK</td>
<td>integer</td>
<td>king's coordinates</td>
<td>initialize-a-game (5); move the king (7)</td>
</tr>
<tr>
<td>WTP.</td>
<td>flag</td>
<td>the user wants to play a game</td>
<td>initialize (2); ask if he wants another game (4)</td>
</tr>
<tr>
<td>RS</td>
<td>string</td>
<td>result of a game</td>
<td>evaluate the situation (10)</td>
</tr>
<tr>
<td>NEOG.</td>
<td>flag</td>
<td>not the end of the game yet</td>
<td>evaluate the situation (10); initialize a game (5)</td>
</tr>
<tr>
<td>B(8,8)</td>
<td>matrix of characters</td>
<td>board representation</td>
<td>initialize a game (5); put a pawn (8); move the king</td>
</tr>
<tr>
<td>X(8), Y(8)</td>
<td>vectors of coordinates</td>
<td>coordinates of positions around the king</td>
<td>compute possible next positions (9)</td>
</tr>
<tr>
<td>E(8)</td>
<td>vector of flags</td>
<td>which of X(I), Y(I) are empty</td>
<td>compute possible next positions (9, 11)</td>
</tr>
<tr>
<td>NE</td>
<td>integer</td>
<td>number of empty positions</td>
<td>compute possible next positions (9, 11); initialize (2)</td>
</tr>
<tr>
<td>LD, LK, LP, LX</td>
<td>characters</td>
<td>representation of position contents</td>
<td>initialize a game (5); put a pawn (8)</td>
</tr>
<tr>
<td>NP</td>
<td>integer</td>
<td>number of pawns the player has left</td>
<td>initialize (2)</td>
</tr>
<tr>
<td>PW</td>
<td>integer constant</td>
<td>total number of pawns (fixed)</td>
<td>initialize a game (5); move the king (7)</td>
</tr>
<tr>
<td>CHC.</td>
<td>integer</td>
<td>choice, the chosen number out of n</td>
<td>ask pawn position (8); ask if another game (4)</td>
</tr>
<tr>
<td>ANSW.</td>
<td>string</td>
<td>an input from the player</td>
<td>ask pawn position (8)</td>
</tr>
<tr>
<td>XI, YI</td>
<td>integer</td>
<td>coordinates where player wants to put a pawn</td>
<td>initialize (2)</td>
</tr>
<tr>
<td>XC, YC</td>
<td>string arrays</td>
<td>to map characters into integers</td>
<td>anywhere needed.</td>
</tr>
<tr>
<td>I, J</td>
<td>integers</td>
<td>general counters</td>
<td></td>
</tr>
</tbody>
</table>
The complete program, written according to Cookbook note Standards 24/Feb/77:

@CC (ROBERT-CAILLIAU)NOD-KINGPIN:SMB
@CC
@CC BUILDS THE NODAL PROGRAM TO PLAY THE GAME OF KINGPIN.
@CC
@CC
@NODAL

% THE GAME KINGPIN
%
% THE GAME IS PLAYED ON A CHESSBOARD. ON THE BOARD
% IS A CHESS KING, INVISIBLE TO THE PLAYER. THE KING
% MOVES LIKE A NORMAL CHESS KING (ONE POSITION AT A TIME)
% BUT IN A RANDOM MANNER. THE PLAYER KNOWS ONLY THE
% PREVIOUS POSITION, (MARKED BY A LETTER K ON THE SCREEN)
% NOT THE CURRENT ONE.
% THE PLAYER HAS A NUMBER OF PAWNS OR COUNTERS, WHICH
% HE CAN PLACE ON THE BOARD ONE PER MOVE. THE PAWNS
% CANNOT BE MOVED ONCE THEY HAVE BEEN PLACED ON THE BOARD,
% NEITHER CAN THEY BE TAKEN AWAY AGAIN.
% THE OBJECT OF THE GAME IS TO SURROUND THE KING SO
% THAT HE CAN NO LONGER MOVE (KINGPIN). THE PLAYER
% LOSES THE GAME IF HE RUNS OUT OF PAWNS WITHOUT
% BLOCKING THE KING, OR IF HE PLACES A PAWN ON THE CURRENT
% (INVISIBLE) POSITION OF THE KING. PLACING A PAWN
% ON THE VISIBLE, PREVIOUS POSITION OF THE KING IS NOT
% FORBIDDEN.
% THE PROGRAM
%
% MAIN PART.
% ==
% INITIALIZE
1.1 DO 2
% WANTS-TO-PLAY := TRUE
1.2 SET WTP.=1
% WHILE WANTS-TO-PLAY DO [ GAME; ASK-IF-MORE ];
1.3 WHILE WTP.=1; DO 3; DO 4
% SAY-GOOD-BYE;
1.4 TYPE ! "GOOD-BYE!" !
1.5 END

% INITIALIZE
% ==
2.1 DIM-INT B(8,8)
2.2 DIM-INT X(8); DIM-INT Y(8); DIM-INT E(8);
2.3 SET PW=20
% "'K'" 'X'
2.4 SET LD=[56; SET LX=[113; SET LP=[120; SET LX=[130;
2.5 DIM-STR XC; DIM-STR YC;
2.6 FOR I=1,8; $SET XC(I)=SUBS(I,I,'ABCDEFGHIJKLMNOPQRSTUVWXYZ'); $SET YC(I)=I I
2.7 TYPE ! "***** THIS IS KINGPIN *****" !!
% GAME
% =====

% INITIALIZE A GAME
3.1 DO 5

% REPEAT
% DISPLAY THE BOARD;
% MOVE THE KING
% ASK WHERE THE PLAYER WANTS TO PUT HIS NEXT PAWN AND PUT IT THERE;
% COMPUTE POSSIBLE KING POSITIONS; EVALUATE THE CURRENT SITUATION;
3.2 DO 6; DO 7; DO 8; DO 9; DO 10

% UNTIL END OF THIS GAME.
3.3 IF NEOG. = 1; GOTO 3.2

% ANNOUNCE THE WINNER OF THIS GAME.
3.4 TYPE ! RS !

% ASK-IF-MORE
% =========
4.1 $ASK "DO YOU WANT TO PLAY ANOTHER GAME?" ANSW.
4.2 $IF (ANSW. = "YES") 4.3, 4.6, 4.3
4.3 $IF (ANSW. = "NO") 4.4, 4.5, 4.4
4.4 TYPE ! 'ANSWER WITH "YES" OR "NO" PLEASE!' !; GOTO 4.1
4.5 SET WTP. = 0;
4.6 RETURN

% INITIALIZE A GAME
% =============
5.1 SET NEOG. = 1; SET NP = PW
5.2 FOR I = 1, 8; FOR J = 1, 8; SET B(I, J) = LD
5.3 SET CHC. = MOD(INT(TIME*10), 64);
5.4 SET XK = INT(CHC./8) + 1; SET YK = MOD(CHC., 8) + 1
5.5 SET B(XK, YK) = LK
5.6 DO 9

% DISPLAY THE BOARD
% ================

% TYPE A FEW BLANK LINES
6.1 TYPE !!!

% TYPE 8 ROWS OF 8 SQUARES OF BOARD, WITH THEIR CONTENTS
6.2 FOR J = 8, -1, 1; TYPE ! &10 &1 J; FOR I = 1, 8; TYPE &1 \B(I, J)
6.3 TYPE ! &12 "A B C D E F G H"

% FOR INFORMATION, TYPE THE NUMBER OF PAWNS LEFT.
6.4 TYPE !!! &38 &2 NP "PAWNS LEFT" !!!

% MOVE THE KING
% =============
7.1 SET CHC. = MOD(INT(TIME*10), NE) + 1
7.2 SET J = 0; SET I = 0; WHILE J < CHC.; SET I = I + 1; IF E(I) = 1; SET J = J + 1
7.3 SET B(XK, YK) = LD
7.4 SET XK = X(I); SET YK = Y(I)
7.5 SET B(XK, YK) = LK;
% ASK WHERE THE PLAYER WANTS TO PUT HIS NEXT PIECE AND PUT IT THERE
%==========================================================================
8.1 TYPE "PAWN COORDINATES"; $ASK ANSW.
8.21 IF SIZE(ANSW.)<>2; TYPE ! 'ILLEGAL ANSWER'; GOTO 8.1
8.22 SET XI=FIRST(X,Y,SUBS(1,1,ANSW.));
8.23 IF XI<0; TYPE ! 'ILLEGAL COORDINATE'; GOTO 8.1
8.24 SET YI=FIRST(X,Y,SUBS(2,2,ANSW.));
8.25 IF YI<0; TYPE ! 'ILLEGAL COORDINATE'; GOTO 8.1
8.3 IF B(XI,YI)=LP; TYPE ! 'A PAWN ALREADY IN THIS POSITION'; GOTO 8.1
8.4 IF B(XI,YI)=LK; SET B(XI,YI)=LX; GOTO 8.6
8.5 SET B(XI,YI)=LP;
8.6 SET NP=NP+1

% EVALUATE THE CURRENT SITUATION;
%==========================================================================
10.1 IF B(XI,YI)=LX; "$SET RS='YOU LANDED ON THE KING, YOU LOSE!'"; GOTO 10.5
10.2 IF NE=0; "$SET RS='THE KING CANNOT MOVE, YOU WIN!'"; GOTO 10.5
10.3 IF NP=0; "$SET RS='YOU HAVE NO MORE PAWNS LEFT, YOU LOSE!'"; GOTO 10.5
10.4 RETURN
10.5 SET NEOG.=0;

% COMPUTE POSSIBLE POSITIONS FOR THE KING'S MOVE
%==========================================================================
9.11 SET X(1)=XK-1; SET Y(1)=YK+1
9.12 SET X(2)=XK; SET Y(2)=YK+1
9.13 SET X(3)=XK+1; SET Y(3)=YK+1
9.14 SET X(4)=XK-1; SET Y(4)=YK
9.15 SET X(5)=XK+1; SET Y(5)=YK
9.16 SET X(6)=XK-1; SET Y(6)=YK-1
9.17 SET X(7)=XK; SET Y(7)=YK-1
9.18 SET X(8)=XK+1; SET Y(8)=YK-1
9.2 SET NE=0
9.3 FOR I=1,8; DO 11
11.1 IF X(I) > 8; IF X(I) <= 8; IF Y(I) > 8; IF Y(I) <= 8; GOTO 11.3
11.2 SET E(I)=0; RETURN
11.3 IF B(X(I),Y(I)) = LP; SET E(I)=0; RETURN
11.4 SET E(I)=1; SET NE=NE+1

SAVE KINGPIN
A simple curve-plotting program

(R. Cailliau - SPS Nodal course)

We wish to plot on the line printer curves of the form \( y = f(x) \), whereby the function \( f() \) is given by the user. For the purpose of the exercise, we will keep the program simple.

Requirements
- the user must be able to give his function
- the user must be asked for the interval \([x_0, x_1]\), \( x_0 < x_1 \), in which the function must be displayed
- we will always take 50 points in \([x_0, x_1]\) but must compute \( y_{\text{min}} \) and \( y_{\text{max}} \) in \([x_0, x_1]\) to be able to scale in the \( y \)-direction and to choose a starting point for \( y \).
- the \( y \) axis will run across the paper over 100 positions (simplicity).

![Plot of function f(x) with y_min and y_max marked.]

The plotting window is shown in Fig. 1. The problem-solving phase goes as follows:

We must approximate points on the plane to available printing positions. Always taking 50 positions in \([x_0, x_1]\) means we step through \( x \) from \( x_0 \) by \( \Delta x \) to \( x_1 \), and the increment \( \Delta x = \frac{x_1 - x_0}{49} \). Since \( y_{\text{min}} \) and \( y_{\text{max}} \) are not given, we must determine them. We can do this by looking at \( y_i = f(x_i) \) for all
\[ x_i = x_0 + \Delta x \cdot i; \quad 0 < i < 49. \] It would be a waste of storage and time to first compute all the \( y_i \) and then find the smallest and the largest. Instead, we will compare every \( y_i \) with \( y_{\text{min}} \) and \( y_{\text{max}} \). If \( y_i < y_{\text{min}} \) then we will take \( y_i \) as the new \( y_{\text{min}} \); if \( y_i > y_{\text{max}} \) we will take it as the new \( y_{\text{max}} \).

To start off on this, we must put decent values in \( y_{\text{min}} \) and \( y_{\text{max}} \). Convince yourself that \( y_{\text{min}} = y_{\text{max}} = y_0 = f(x_0) \) is the right choice, and compare this with the way your minimum-maximum thermometer works at home if you have one (if you do not have such a thermometer, inspect one next time you go to a supermarket).

For a given \( y_i \), we must choose one of the \( 100 \) printing positions. Now \( \Delta y = y_{\text{max}} - y_{\text{min}} \) and the distance \( y_i - y_{\text{min}} \) contains \( (y_i - y_{\text{min}})/\Delta y \) such intervals. This last number varies between \( 0 \) and \( 99 \). We will round it, and then add \( 1 \) to get an integer between \( 1 \) and \( 100 \), which we call \( y_p \).

The \( x \) axis is printed at \( y = 0 \) (position 1), how many blanks must we output between it and the position for \( y_i \)? If \( y_p = 1 \) we should not even try to print the position. If \( y_p = 2 \), then there should be no blanks, if \( y_p = 3 \) then one etc.: if \( y_p > 2 \), \( y_p - 2 \) blanks. Plotting is done by computing all \( y_i \) again and taking a new line right before output of a position. The first \( y_i \) (i.e. \( f(x_0) \)) is on the \( y \)-axis and must not be output.

The user interface can be made very fancy, to provide fixing of scaling, "friendly" scaling, tabulation, etc., but the only extra we will allow here is this: after having made 1 plot of a particular function, the user may want to change \([x_0, x_1]\), because he is looking for a certain feature (such as a zero or a minimum) which happens to lie outside the interval he just chose. Therefore we will repeat the plotting until he indicates that he's finished examining the curve.
Program structure development

Since the whole program is a big loop until-done-with-examining, the first approximation PSD is: (Fig. 2)

\[ \text{CURVE- PLOT} \]

- initialize and get \( f() \)
- while the user examines the function
  - make a plot according to his desires
  - leave

\[ \text{Fig. 2} \]

We will leave the initialisation block until later and concentrate on the making of a single plot. For each plot, the values of \( x_0 \) and \( x_1 \) must be got, they must be verified (\( x_1 > x_0 \)), the values \( y_{\min} \) and \( y_{\max} \) must be determined from them, then the curve has to be printed in the window \([x_0, x_1], [y_{\min}, y_{\max}]\) , and finally we must find out whether that was enough or whether he (the user) wants another go. This results in Fig. 3.

\[ \text{Plot according to desires} \]

- get \( [x_0, x_1] \)
- determine \( y_{\min}, y_{\max} \)
- plot the function in the window \([x_0, x_1], [y_{\min}, y_{\max}]\)
- does the user want more? 
  - No: he no longer examines this function

\[ \text{Fig. 3} \]
The first block of Fig. 3 is simple: Fig. 4

The second block is somewhat more complex but is well-known from our pondering-phase, and the last block is not even worth expanding. Fig. 5 represents determination of \( y_{\text{min}} \) and \( y_{\text{max}} \), this leaves the plotting itself as the most "difficult" bit.
When plotting the function it is interesting to the user to print some information such as \( f() \), \( x_0 \), \( x_1 \), \( y_{min} \), \( y_{max} \) so that he can keep his plots apart. We will print the \( x \)-axis using the character ",", the \( y \)-axis using "-", and the curve using "*". Remember that \( y_p \) is the position in 1 to 100 of a point of the curve. Then:

(a) the \( y \)-axis: \( x = x_0 \), if \( y_p(x_0) = 1 \) then print "*" and 99 times "-"; else if \( y_p(x_0) > 1 \) then print \( y_p(x_0) - 1 \) times "-", one "*", and \( (99 - y_p(x_0)) \) times a "-". Print "y".

(b) for all other points, i.e. for \( x = x_0 + \Delta x \) upto \( x_1 \), do the following:
1. take a new line, determine \( y_p(x) \),
2. if \( y_p = 1 \) then print "*",
   else if \( y_p = 2 \) then print ":", print "*",
   else if \( y_p > 2 \) then print ":", print \( (y_p - 2) \) times a space, print "*".

PSD in Fig. 6.
Fig. 6

Obtaining the function $f()$ from the user is not difficult in Nodal: we read its definition into a string, and make a Nodal line out of it. (The practice of constructing program lines as you go, i.e. during execution of a program, is very dangerous, unreliable and bad programming. It should never be done, except in such well-controlled cases as this, where only a functional relation between (two) variables is determined but it does not change the program logic.)

If you have noticed that we forgot a very important test during the reasoning on pages 2 and 5, then read on; if you have not noticed, then dig up your elementary algebra books and reread those pages! What have we done? On pages 2 and 5 we use $y_i$. But $y_i$ is determined as the number of intervals $\Delta y$ in $y_{i_{min}}$. This is computed by dividing $y_{i_{min}} - y_{i_{max}}$ by $\Delta y$. Suppose $\Delta y = 0$? Then our whole program goes haywire! The meaning of $\Delta y = 0$ actually is: $y_{min} = y_{max} = f(x)$, i.e. the function does not vary with $x$, it is a constant. Since this can happen, even with very complex functions, we should not try to plot if $y_{min} = y_{max}$. (But there is also the danger that
\( \Delta y \) gets very small, and division will become inaccurate, or results very large. Normally, you should test for \( \Delta y < \varepsilon \) where \( \varepsilon \) is a function of your computations and the precision of your machine.

The PSD of Fig. 3 is therefore wrong, and must be replaced by that of Fig. 7:

The structured development of the program guarantees modularity, and it is therefore easy to re-design the erroneous PSD(s), usually without having to modify many blocks.

Every program should be developed structurally and the paths of thought followed during development (with the errors!) should be written down as an essential part of the program's documentation.

Observe another small point for optimisation: \( \Delta y \) does not have to be computed every time in group 9. In Fig. 6, preceding the print of the y-axis, there should be a small block computing \( \Delta y \).
Try the following extensions or modifications: (use PSD's and write down all your thoughts)

1. if the x-axis (y=0) is within the window, then plot it too, using "+", (difficult),

2. allow for each plot the optional output of a table of values x, f(x) in [x₀ x₁] (easy),

3. what happens when x₁ - x₀ = Δx is so small that division by 49 causes underflow, and x effectively becomes zero? (easy)

4. suppose f(x) is also dependent on a parameter p: y = f(x,p). Make the program able to output a set of curves f(x) in [x₀ x₁], for n different values of p in [p₀ p₁]. (n, p₀, p₁ input by the user.) (not very difficult, but useful only if you have a digital plotter so that the n curves can be drawn on the same piece of paper.)

5. Make a simultaneous plot of y₁ = f₁(x), y₂ = f₂(x), .... yₙ = fₙ(x) all on the same window, using "₁", "₂", ...."ₙ" to recognize the different curves (difficult ?),

6. provide "friendly" scaling in both x and y. Example: try to adjust the window so that the axes cross at points with "round" coordinates (not 5.475 but 5.5, for example), and also so that the widths in x and y correspond to a pleasant multiple of divisions on the axes (such as 2, 5 or 10 divisions, but not 7). (difficult: at least a full day of mathematics).

(Think first, program later!)
1.01 X CURVE- PLOT PROGRAM
1.02 X
1.03 X
1.10 DO 2
1.20 WHILE EXAM. = 1 DO 3
1.30 TYPE "!! "BYE!!" "
1.40 END

2.01 X INITIALIZE AND GET F
2.02 X
2.03 X
2.10 TYPE "!! "CURVE- PLOT" !"FUNCTION "
2.20 $ASK F; IF SIZE(F)< 3; TYPE " !"TYPE Y=F(X) PLEASE. " !"GOTO 2.2
2.30 $SET NODLIN=99.1; "SET " F
2.40 SET EXAM. = 1
2.50 SET WX=50; SET WY=100

3.01 X MAKE A PLOT ACCORDING TO DESIRES
3.02 X
3.03 X
3.10 DO 4; DO 5
3.20 IF YMAX.-YMIN. > 0; DO 6; GOTO 3.4
3.30 TYPE "CONSTANT FUNCTION! NO PLOT MADE. " !!
3.40 DO 7
4.01 X GET X0. X1.
4.02 X
4.03 X
4.10 ASK "X0 " X0 "X1 " X1; IF X0>=X1; TYPE "X1 MUST BE > X0!!"; GOTO 4.1

5.01 X DETERMINE YMIN-, YMAX
5.02 X
5.03 X
5.10 SET DX=(X1-X0)/(WX-1)
5.20 SET X=X0; DO 99; SET YMIN. = Y; SET YMAX. = Y
5.30 FOR X=X0+DX,DX,X1; DO 8

6.01 X PLOT THE FUNCTION
6.02 X
6.03 X
6.10 TYPE "!! "FUNCTION ' F "!
6.20 TYPE "X0: X0 "X1: X1 " YMIN: YMIN. "YMAX: YMAX. "!!
6.30 SET DY=(YMAX.-YMIN.)/(WY-1)
6.40 SET X=X0; DO 99; DO 9
6.50 IF YP=1; DO 10; GOTO 6.7
6.60 DO 11
6.70 FOR X=X0+DX,DX,X1; DO 12

7.01 X ASK IF MORE
7.02 X
7.03 X
7.10 TYPE "!!!! DO YOU WANT ANOTHER PLOT? "
7.20 $ASK ANSW. ;
7.30 $IF (ANSW. = 'Y'); 7.4; 7.9; 7.4
7.40 $IF (ANSW. = 'N'); 7.5; 7.8; 7.5
7.50 TYPE !"ANSWER WITH 'Y' OR 'N' PLEASE! "; GOTO 7.1
7.80 SET EXAM. = 0
7.90 RETURN

8.10 DO 99
8.20 IF Y<YMIN.; SET YMIN. = Y
8.30 IF Y>YMAX.; SET YMAX. = Y

9.01 X COMPUTE YP
9.10 SET YP=INT((Y-YMIN.)/DY + 1.5)

10.10 TYPE "*"; FOR I=1; WY-1; TYPE "**
11.10 FOR I=1; YP-1; TYPE "**
11.20 TYPE "*
11.30 FOR I=1; WY-YP; TYPE "**
12.10 TYPE "!! DO 99; DO 9
12.20 IF YP=1; TYPE "*"; RETURN
12.30 IF YP=2; TYPE "**"; RETURN
12.40 TYPE "!"; FOR I=1; YP-2; TYPE "*
12.50 TYPE "*"