KERNFORSCHUNGSZENTRUM

KARLSRUHE

Mai 1973 KFK 1722

Institut für Reaktorentwicklung

Design Principles of the GRAPHIC System

G. Enderle, E.G. Schlechtendahl
U. Schumann, R. Schuster

GESELLSCHAFT
FÜR
KERNFORSCHUNG M.B.H.

KARLSRUHE

GESELLSCHAFT FÜR KERNFORSCHUNG M.B.H.
KARLSRUHE

Kernforschungszentrum Karlsruhe

KFK 1722

Institut für Reaktorentwicklung

Design Principles of the GRAPHIC-System

G. Enderle, E. G. Schlechtendahl,
U. Schumann, R. Schuster

with contributions by
K. Leinemann, W. Olbrich,
H. Schnauder

Gesellschaft für Kernforschung mbH, Karlsruhe

Abstract

GRAPHIC is a system for handling graphical information. It allows definition, management and output of graphical objects. The main components of the GRAPHIC-system are:

- A problem oriented language for specification of graphical objects and of operations to change them. The language contains subroutines, DO- and IF-statements.
- A language interpreter which builds up a data structure while analysing the statements of the language.
- A data structure for the internal representation of graphical objects and operations as nodes in a hierarchical ring structure.
- Routines for parsing structures of graphical objects, for executing operations, creating new objects and for output of graphical information.

Language and data structure allow referencing of objects by name. GRAPHIC has been implemented as a subsystem of the CAD-system ICES.

Entwurfsprinzipien des GRAPHIC-Systems

Zusammenfassung

GRAPHIC ist ein System zur Definition, Bearbeitung, Verwaltung und Ausgabe graphischer Information. Die wesentlichen Bestandteile dieses DV-Systems sind:

- Eine problemorientierte Eingabesprache zur Spezifizierung von graphischen Objekten und von Operationen zu ihrer Veränderung. Die Sprache enthält Unterprogramme, DO- und IF-Anweisungen.
- Ein Sprachinterpretierer, der während der Analyse der Eingabesprache eine Datenstruktur aufbaut.
- Eine Datenstruktur zur internen Darstellung von graphischen Objekten und Operationen in Form von Objekt- und Referenzknoten in einer hierarchischen Doppelring-Struktur.
- Routinen, die Strukturen graphischer Objekte abarbeiten, Operationen ausführen und neue Objekte oder Ausgaben graphischer Information erzeugen.

Sprache und Datenstruktur erlauben den namentlichen Zugriff auf alle graphischen Objekte. GRAPHIC wurde als ein Subsystem des CAD-Systems ICES implementiert.

Contents

Page

Page

1. Introduction

Drawings are one of the most important means of communication between scientists, engineers and technicians. For this reason many computer programs produce output of graphical information either as plots or on a display unit. In order to change details of such drawings, e. g. for editorial revision of illustrations for publication, the programs have to be modified. This may be very time-consuming or even impossible in the case that the program source is not available. So the need arose for a system able to manipulate in a convenient way graphical output produced by computer programs. Such a system should have the ability to edit (alter, delete or add) parts of a drawing and to store graphical information. Furthermore the system should supply means for specifying new drawings in a way suitable to the engineer. The graphical data processing system GRAPHIC has been developed at the Institut für Reaktorentwicklung as part of the project REGENT (Rechnergestützter Entwurf = Computer Aided Design) /1, 2/. It is a means to create graphical objects, to manipulate them, to store graphical information and to produce output, at present in the form of plots. GRAPHIC also provides the necessary interface for adaptation of other sources of graphical input.

Zum Druck eingereicht am 3.4.1973

2. The hardware and software environment of the GRAPHIC-system

GRAPHIC was realized as a subsystem of ICES (Integrated Civil Engineering System /3, 4, 5/). ICES not only offers a powerful dynamic memory management, able to handle dynamic arrays and dynamically linked load modules, but also can be used to define problem oriented languages in a flexible and convenient way. Some experiences with the use of ICES have been described in /6/. Since ICES can only be run on an IBM/360 or IBM/370 computer with the operating system OS/360, GRAPHIC also is bound to the System/360.*) The minimum hardware configuration needed is a computer IBM 360/40 with at least two disk units, type 2311 and up. These requirements are due to ICES. The minimum size of core required for GRAPHIC is 240 k Bytes. GRAPHIC was developed on a configuration IBM 370/165 and IBM 360/65 under OS/360 with ASP.

The graphical output device that is used for drawing graphical objects is a Calcomp plotter type 763, run offline. The routines to direct the plotter are taken from the Calcomp Graphics Basic and Functional Software /7/.

*) RCA, UNIVAC and Philips have announced, that they will support ICES on their computers.

3. Results of the system analysis

3.1 Types of graphical objects and their representation in internal storage

3.1.1 Basic components of a drawing

A drawing consists of points, lines, curves and symbols. We shall call these basic components of a drawing, which can be plotted or displayed immediately, graphical elements (GE). To a man looking at a drawing the lines and curves on a picture have certain relations to each other. E. g. the drawing of a house consists of lines representing the walls, the roof, the door and the windows. A Graphic Data Processing System (GDPS) therefore must be able to manage relations between GE's. A simple relation between elements is existing, if they "belong together", i. e. they can manipulated as a whole and, in a system with names for referencing objects, they can be referenced by the same name.

Different possibilities for describing the relations between objects in data structures will be discussed in the following chapters. Objects will be represented in the figures by rectangular blocks, while relations between objects are lines between blocks. Relations with undefined objects are represented as shown in fig. 1.

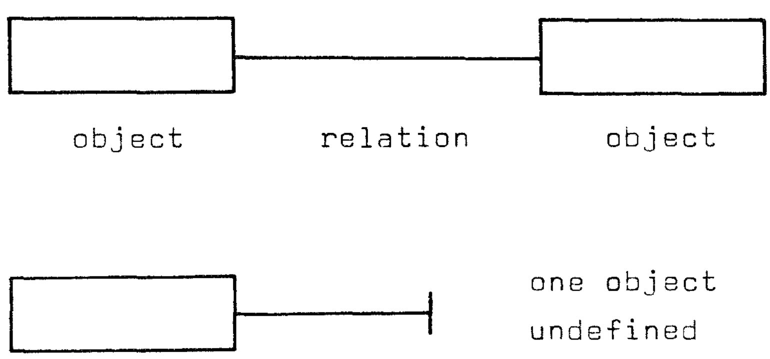


Fig. 1 Objects and relations

3.1.2 Linear lists

Linear lists are very simple structures /9/.

Fig. 2 Linear list

There are two different kinds of structures that can be represented by linear lists: <u>Set lists</u> and <u>ordered linear lists</u> /8/. A set list is a finite set of objects linked together by relations. Every object can be reached from another one by following one series of relations.

The relations just fulfill the task to show that all objects of the list are members of the same set. The order of objects in the list does not have any significance. Every object is the starting point of two relations with identical meaning.

An ordered linear list is a finite set of objects, every one possessing a left and a right relation. Every object can be reached from another one by running through a series of relations, no relation may be repeatedly passed. In an ordered list, starting at one object $O_j \in (O_1,\ldots,O_n)$ all objects O_i, $0<i<j$ can be reached only over left relations, all objects O_i, $j<i\leq n$ can be reached only by going along the right relations. The order of objects in the list cannot be changed.

The random enumeration of all members of an institute, e. g. is a set list, while an alphabetical list is an ordered list.

3.1.3 Circular list

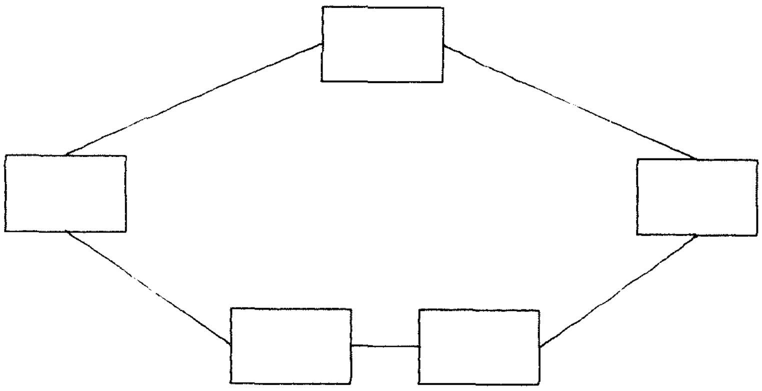


Fig. 3 Circular list

In circular lists, or rings, every object can be reached from another one by following one of two possible series of relations. Circular lists represent the membership of objects in a set. In many cases circular lists can be handled more effectively by algorithms than linear lists.

Linear lists and rings can serve only to describe "linear" or one-dimensional structures, where only two relations emerge from an object. The following chapter will take a brief look on methods for representation of more complex structures.

3.1.4 Trees

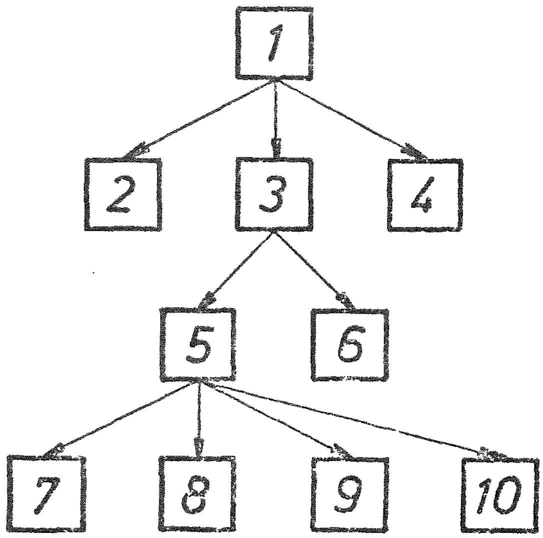


Fig. 4 Tree structure

Knuth /9/ offers a recursive definition of tree structures:

"A tree is a finite set T of one or more nodes such that

a) There is one specially designated node called the root of the tree, root (T), and

b) The remaining nodes (excluding the root) are partitioned into m>0 disjoint sets $T_1, \ldots T_m$, and each of these sets in turn is a tree. The trees $T_1, \ldots T_m$ are called subtrees of the root."

This definition describes a set tree as already discussed in /8, 10/, because the relative order of objects is of no importance. This becomes evident, if we show the tree structure above in another form:

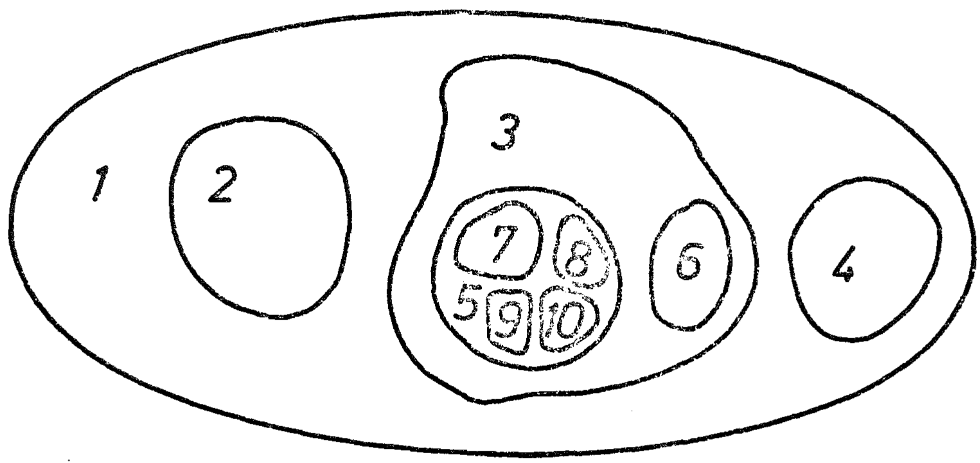


Fig. 5 Representation of a set tree

If the relative order of subtrees $T_1, \ldots T_n$ is important, the structure is called an ordered tree. A special kind of ordered trees are binary trees, where every node has one or two subtrees.

Plain trees (Knuth calls them "forests") can be converted into binary trees:

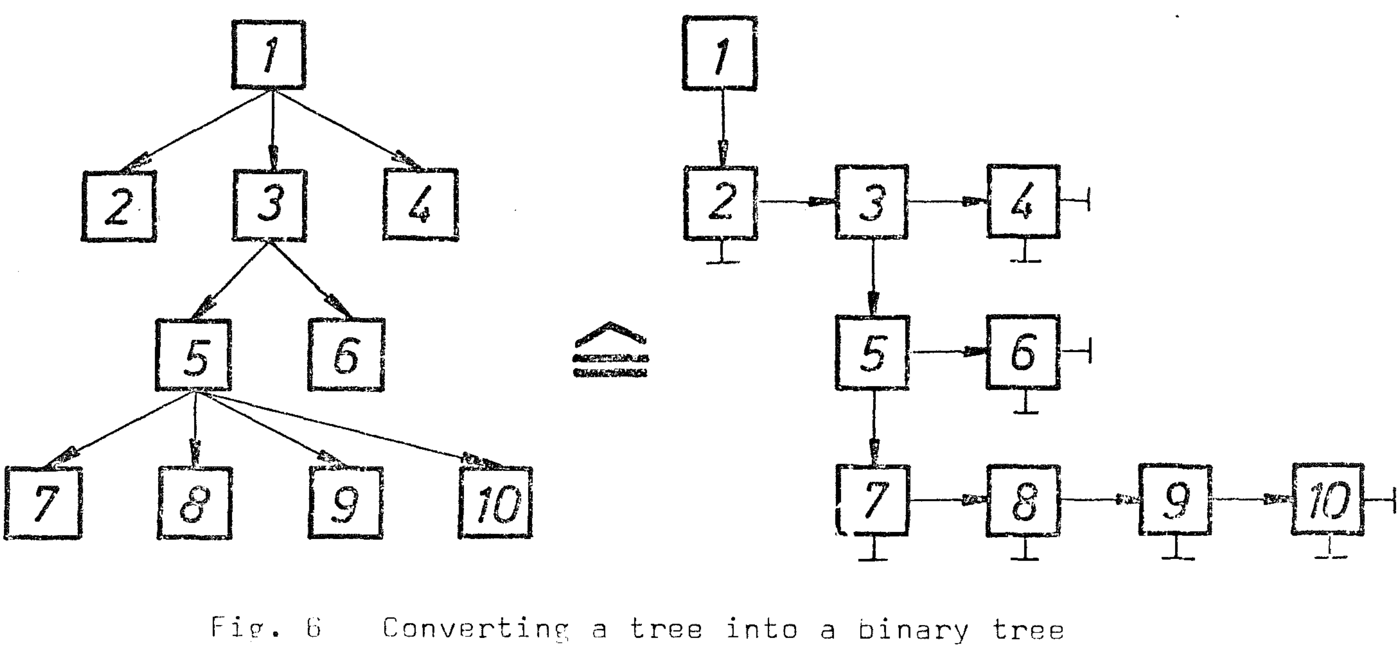


Fig. 6 Converting a tree into a binary tree

This is of some importance since binary trees often can be treated more effectively by algorithms. The terminology used for describing relations between nodes of trees is normally taken from family trees. Every root is called the father of the nodes at the head of its subtrees, these nodes are called brothers. The subtree-nodes are sons of their root. Besides the types of structures mentioned above 'multilinked structures' or "graphs" are used for representing complex relations between objects. The following chapter lines out the advantages and disadvantages of trees and multilinked structures for graphical objects in a graphical data processing system (GPDS). This leads to the data structure of GRAPHIC, which is described in more detail in chapter 5.

3.1.5 Representation of graphical objects by circular lists

Possible data structures for graphical objects are described in /9, 11, 12, 13, 14/. In SKETCHPAD /15/ and PRADIS /16, 17/ circular lists are used for graphical data structures. Since a drawing must be described by multidimensional relations, several rings are necessary, which are connected by special relations. The drawing of a triangle shall serve as an example.

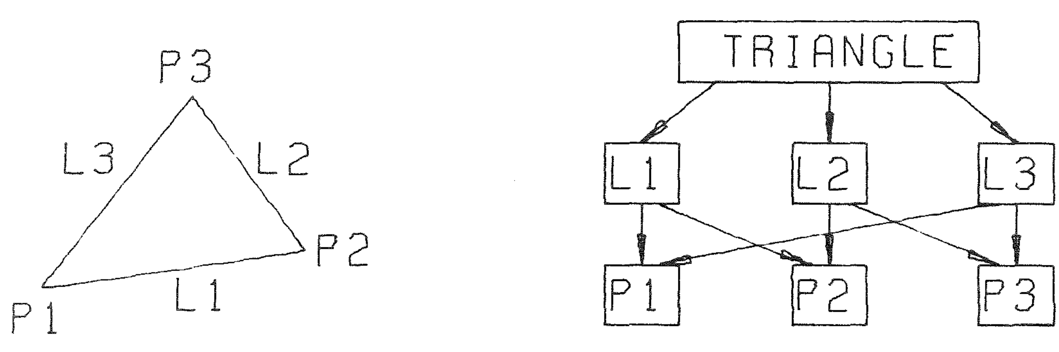


Fig. 7 Objects and relations representing a triangle

The structure "triangle" is an object, which is composed of different subobjects:

points P1, P2, P3
lines L1, L2, L3.

Fig. 7 shows the representation of the object "triangle" by three hierarchically related circular lists. Rings have the advantage that objects can easily be added to and deleted from the list.

This is true even more for doubly linked circular lists since an object can be added or deleted at any place without having to run through the whole ring. The rings in the triangle example are not ordered lists but just comprise objects of the same sort (points, lines). The order of the objects in a ring does not matter. The structure of the drawing is not represented by the rings but by pointers from an object in one ring to objects in other rings. This concept has the consequence that changing of one object (e. g. point P3) effects several other objects (lines L1 and L3 and the triangle).

3.1.6 Graphical objects as nodes in a tree structure

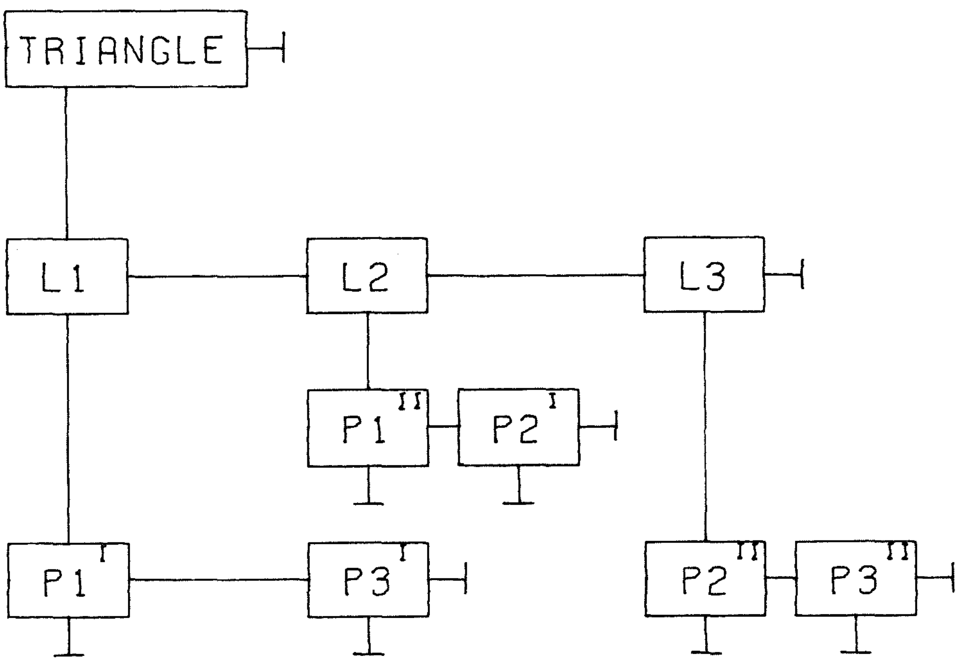


Fig. 8 Binary tree representing a triangle

Fig. 8 shows the representation of the triangle object in a binary tree. Advantages and disadvantages of trees compared to related circular lists become obvious:

Tree structures show all relations clearly and distinctly, since from one object to the other there is only one series of relations. But for this clearness we have had to duplicate objects. The object P1, e. g. has to be incorporated into the tree structure 2 times ($P1^{(I)}$ and $P1^{(II)}$). This needs not only a big amount of memory but also may require more work to be done. If the point P1 is to be shifted in a way that all objects related to it are also changed, the linear transformation for P1 must be done 2 times, for $P1^{(I)}$ and $P1^{(II)}$. On the other hand, if we wish to change only $P1^{(III)}$, this can be done very easily.

3.1.7 Directed Graphs

In order to avoid the necessity for storing several copies of one object, modified tree structures are advantageously used for representing graphical objects. The structures are directed graphs, they may be trees, but they must not. Updating of objects can be done easier than in strict tree structures. In a directed graph objects will not be copied, if they are referenced more than once (see fig. 9).

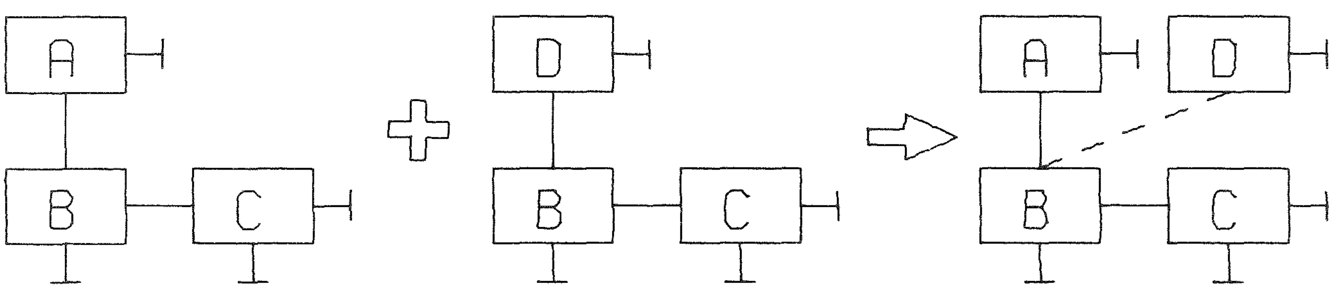
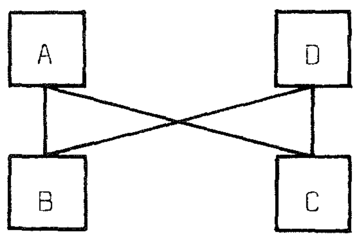


Fig. 9 Two objects referencing the same subtree

We see, that now not only a root (a father) can have more than one subtree (son), but that also a subtree (son) can possess more than one root (father). (For convenience of the terminology used here, let us allow that a son may have more than one father). The symmetry becomes obvious, if we somewhat change the presentation of the above example:



A is father of B and C
D is father of B and C
B is son of A and D
C is son of A and D

Fig. 10 Object relations represented by a graph

Let us take a look at a slightly more complicated example:

Let A be father of D and E,
let B be father of D, E and F
let C be father of E and F.

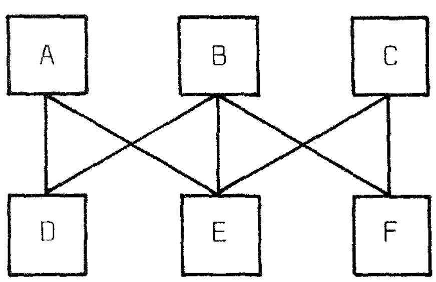
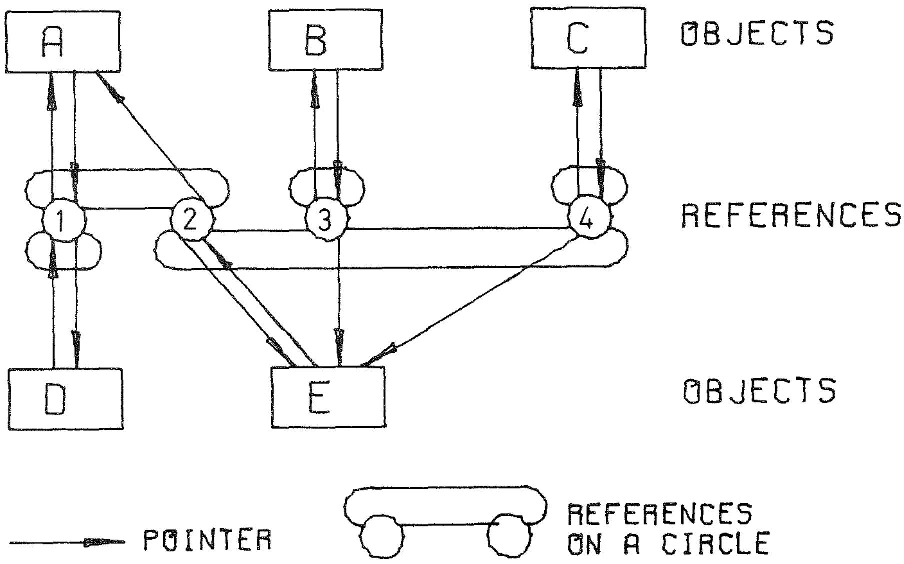


Fig. 11 A graph

We see that in such a structure every object may possess a discretionary number of relations to fathers and sons. Realization of these relations by a set of upward-pointers and a set of downward-pointers at every node would require a complicated storage-allocation-method for the space needed by the varying number of pointers. This can be avoided by leaving the task of connecting two objects to special objects, called "references". For these references the same storage allocation method can be used as for all other objects in the Graphical Data Pool (GDP).

The following example shows the method used for connecting objects by references.



A is father of D and E

B is father of E

C is father of E

D is son of A

E is son of A, B and C

Fig. 12 References connecting objects

Figure 12 demonstrates the rules applied to the use of references:

- Every reference represents one "father-of"-relation (and, corresponding to this, one "son-of"-relation).
  So, from object A to its two sons, D and E, there are two references, 1 and 2; from E to its 3 fathers, A, B and C, there are 3 references 2, 3 and 4.
- References are linked together by two rings, one comprising all references that represent <u>the relations from one father to all of his sons</u> (in fig. 12 the upper rings), the other one comprising all references representing the relations from

one son to all of his fathers (the lower rings in fig. 12).

In order to reach all subjects from one root of a structure, one has to go along the downward pointer, pass around the reference ring comprising all sons and follow the down-pointer of all references met on this ring. So we reach the next lower level of the structure. In order to reach all superobjects of an object we take the corresponding way up. Go up the upward pointer, run along the ring of references representing the relations to all fathers, from every reference met on this way you go up the upward pointer and will reach the next higher level of the structure.

The scheme introduced here for storage of graphical objects is advantageous compared to a strict tree structure only if n references take less space in memory than n-1 copies of the object including the space required for the bookkeeping which marks the various copies of one object as "belonging together". In most cases this is true since graphical elements not only consist of the data representing relations (pointers e. g.) but also of attribute data (e. g. coordinates of the points of a polygon). Furthermore most operations in a GDPS have to be accomplished for all copies of a graphical object, in our data structure they have to be done only once because only one copy of every object is stored.

3.1.8 Identifying graphical objects

A user of a graphical data processing system (GDPS) must be able to inform the system about what action he wishes to be done on what parts of the stored graphical information. There are several ways of referencing graphical objects. One way is to describe the object by specifying to the system it's relations to the other objects or to the origin of the used coordinate system. A user could say: "Shift the circle in the upper left corner of the drawing 2 centimeters to the right." A quite similar way of referencing objects is to point at them on a display unit. Another way to reference objects is to associate a name with each object, then being able to say: "Shift the circle named 'C1' 2 centimeters to the right".

We feel strongly that the first way identifying a graphical object would be of great advantage for someone wishing to change a drawing. But several problems arise if one intends to implement such a feature. Either all objects in the GDP have to be searched for the object in the upper left corner or the objects must be stored in a sequence according to their coordinates. The latter method would simplify the search for an object with a specified position on the drawing, but would not help finding, say, the circle with the greatest area.

So for the sake of efficiency and easier implementation (and because we are accustomed to use names for identifying objects in nearly all programming languages) we came to the decision of using names for identifying graphical objects in the GRAPHIC system. Nevertheless, this problem has been studied in some more detail, as described in /18/. Names may identify graphical elements (GE), e. g. points, lines, circles but also more complicated objects such as a set of several GE's. We call such a set of GE's that can be referenced as a whole a "Graphical Collection" or just collection.

3.1.9 Graphical Operations

Operations performed upon GE's are called in this paper "Graphical Operations" (GO). Examples are the creation of a line between two points or a circle through 3 points, or linear transformations such as shifting, rotating or enlarging GE's. GO's could be represented by a set of procedures, performing the task specified by the user and adding, as a result, a new GE (or a collection of GE's) to the GDP. The following example shows one shortcoming of this method (the language for instructions to a GDPS used here is selfexplaining).

```
P1 := point (x = ..., y = ...)
P2 := point (x = ..., y = ...)
P3 := point (x = ..., y = ...)
L1 := line from P1 to P3
L2 := line from P1 to P2
L3 := line from P2 to P3
TRIANGLE:= L1, L2, L3
Plot TRIANGLE
Shift P3 2 cm to the right
Plot TRIANGLE
```

In this example the two PLOT-instructions would produce identical results, since TRIANGLE was evaluated prior to the change made to point P3. If we would intend to change the triangle, we would have to write:

```
.
.
.
Plot TRIANGLE
Shift P3 2 cm to the right
L1 := line from P1 to P3
L3 := line from P2 to P3
TRIANGLE: = L1, L2, L3
Plot TRIANGLE
```

This is neither a notation to be called rather elegant nor very pleasant for the user. It would be useful if we could say:

let L1 be the line from P1 to P3, but evaluate it only when L1 is referenced.

(The notation adopted here for purposes of demonstrations is: L1 = line from P1 to P2).

A feature for reevaluating graphical operations every time their result is needed can be implemented by integrating graphical operations together with graphical elements as nodes in the graphical data structure.

The operation L1 = line from P1 to P2 thus could be represented by the structure in fig. 13.

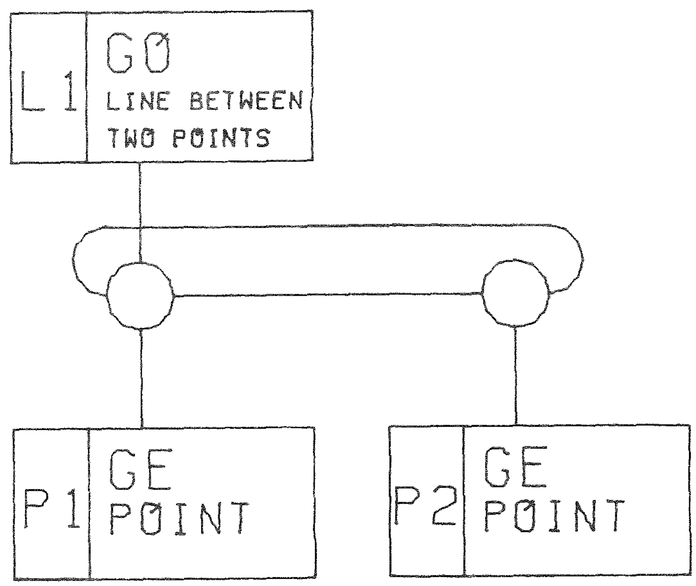


Fig. 13 The "line"-operation in the data structure

If graphical operations are included in the data structure for graphical objects we have two possibilities:

1. To evaluate an operation at once, then the result is not changed by alterations of the graphical elements upon which the operation was performed.
2. To evaluate an operation every time the result's name is referenced. In this case a change of an element will affect the result of all operations defined by referencing this element.

The execution of a task thus will take place in two steps: First, build up a data structure representing the task. Secondly, at once or later depending on the user-defined specification, parse the structure and compute a result. Besides the advantage of being able to reevaluate the result of an operation several times, this method allows to represent the task specification in the system in a way adequate to the computer, i. e. easy to parse and easy to change. This general principle can also be found in the AED-system /19/.

3.2 Possible forms of input to a GDPS

3.2.1 Interactive display terminal

When using the term graphical input most people associate with it a workplace equipped with an intelligent display unit, lightpen or joystick, alphanumeric and function keyboard /15,16/. In practice, however, only a very small number of potential users of a GDPS has an active display unit available. This is probably due to the following reasons: Active graphic display units are inadequately expensive (both by themselves and because of the computer configuration which they require as background) and there is a lack of standardized interface (especially software interface) between computers and display terminals - and this situation is unlikely to change in the next future. There are two other problems in using interactive graphic terminals: Whenever the problems which are to be treated by the GDPS become large enough to require a significant amount of computer time, the interactive mode of operation is no longer appropriate; the complete and correct documentation of the terminal session is a nontrivial job if a lightpen is used. For these reasons - at least at the present state of the art - a language which can be used both in batch and interactive mode is certainly more widely applicable than direct input at a graphic terminal.

3.2.2 Graphical languages /20, 21, 22, 23/

One way of specifying graphical elements and operations upon them is to use a language. Simple languages would be series of calls on special routines in some higher level language:

```
P1:=  point(x = 1., y = 2.)    CALL CREATE_POINT(1.,2.,'P1');
L1:=  line from P1 to P2       CALL CREATE_LINE('P1','P2','L1');
```

Typical examples of this approach are /24, 25/.

Another possiblity would be input in form of a table /26/:

NAME	ELEMENT or OPERATION	DATA	
P1	POINT	1.	2.
L1	LINE	P1	P2

It seems to us, that a graphical language closer to the natural language would be a better solution. A program in such a language can be read and understood more easily, even by someone who is not familiar with the GDPS. The input listing itself can be used as a documentation. Free formatted input allows less mistakes and more flexibility. Moreover, instead of P1:= crosspoint of L1 and L2; L3:= line from P1 to P2,
one should be able to write:
L3:= line from crosspoint of L1 and L2 to P2;

This kind of nesting of specification for graphical objects would hardly be possible in a "CALL-"language or in a tabular-input-language. Furthermore a graphical language should include some features that have proven their usefulness in many high-level languages. To these features belongs the possiblity to define procedures and pass arguments to them, do-loops and if-statements. Of course, using a language close to the natural language requires more program writting effort, the language is more redundant than tabular input, and language interpretation is more complicated and thus more expensive.

3.2.3 Input of graphical information produced by existing programs

Many programs solving scientific or technical problems produce plots. For documentation or publication often the need arises to make editorial changes on them. E. g. one wishes to add a second y-axis to a drawing representing functions of a pressure and a temperature over the time (x)-axis, or in a technical drawing changes have to be made to a design detail. This kind of modifications to program-produced plots mostly were made by hand.

It is therefore desirable to make an interface available between the graphical output of existing software and the input of a GDPS. Editing of plots could then be made easier and more conveniently. All manipulations applicable to graphical objects created by using the GDPS should also be available for objects supplied by existing programs.

3.2.4 Input from existing drawings

There is definitely a need for input of existing line drawings in many engineering applications. Hardware equipment for production of a point by point trace of line drawings is available. Output from such equipment is usually generated on magnetic or paper tape. The GDPS should provide a flexible and well defined interface for the logical adaptation of this form of input.

3.3 Output of graphical data

Possible output devices for graphical information are plotters and displays. Since the format of graphical information is different not only for plotters and displays but beyond this for the plotters and displays of different manufacturers, a clearly defined interface is necessary between the GDPS and the procedures directing the graphical output to a specific device. In the GRAPHIC system this interface will be a "plotfile" called data set on a secondary storage device that is filled by the GDPS and interpreted in a second step in order to prepare the output in a way adequate to the plotter or display unit actually used.

3.4 The basic components of the GRAPHIC system

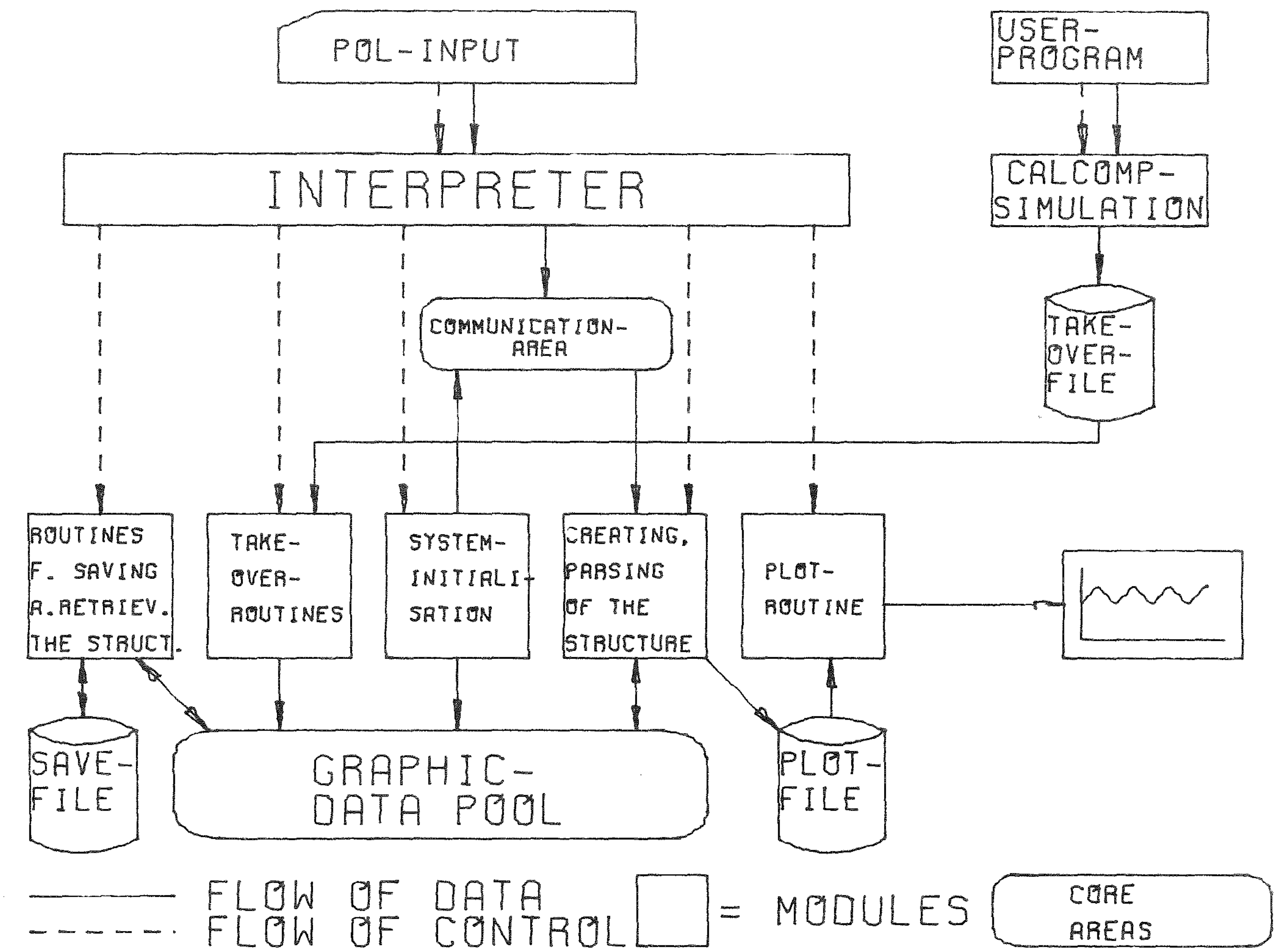


Fig. 14 Basic components of GRAPHIC

Fig. 14 shows the basic components of the GRAPHIC system. They are:

- an interpreter
- a communication area
- a module package for structuring graphical data and processing commands
- a graphical data pool
- save and interface files.

The input made up of statements in the GRAPHIC-language are analysed by an interpreter, which causes the storage of information in the data pool by calling processing routines. The interpreter also causes the manipulation of graphical data in the pool. The graphical data pool contains all objects in their internal representation. Data are passed from the language by use of the communication area. Graphical information is passed to output devices by means of a plotfile, containing all information necessary for plotting objects. This plotfile represents a clearly defined interface to the hardware and software of different plotter manufacturers. Thus, the plotter may be changed without difficulties.

Using a set of plot-simulation routines and the "takeover"-file, plot-information from other jobs may be taken over into GRAPHIC.

4. The GRAPHIC language

4.1 Basic structure of the language

The concrete syntax of the GRAPHIC language is listed in appendix A. A program written in GRAPHIC consists of a head: "GRAPHIC", a block comprising a number of GRAPHIC-statements and a tail: "END GRAPHIC". The head (in ICES called subsystem command) is used to identify the following information to the ICES system as a GRAPHIC-program. The tail starts the processing of the contents of the output-interface-file by the graphical device driving routines. Blocks consist of one or more GRAPHIC-statements and may contain a deliberate number of other blocks.

A block is characterized by a heading: BEGIN or PROCEDURE and by an ending: END.

All names that are explicitly or implicitly declared within a block are local to that block, they can be referenced in all contained blocks, but not outside the block. Explicit declaration is done by means of a declare statement (see chap. 4.5.2), implicit declaration is caused by referencing a name that has neither been used so far in the block nor in one of it's outer blocks. If a name that is local to an outer block is explicitly declared in an inner block, the name in the outer block identifies a different object than the name in the contained block. The scope of names in nested blocks is governed very much by the same rules as it is in other block oriented languages like ALGOL 60 or PL/1 /28/.

In the following chapters GRAPHIC-statements will be described in detail. Because the GRAPHIC language is implemented by use of ICES-CDL /3, 5/, rules and restrictions of this system are valid for GRAPHIC, too. Every statement has to start with a keyword, (in ICES called the command name), it can be coded free formatted on columns 1 to 80 of a card, continuation is noted by placing a "-" (hyphen) as the last character on the card to be continued. As many as five cards can form one logical statement (for an exception in GRAPHIC see the polygon specification in chapter 4.3.1.2). Comments can be placed at any card after "b$b" (blank, dollar, blank). Continued cards may have comments, the "$"-sign then has to be placed after the continuation hyphen. Blanks or commas are required as delimiters between all items of the language. The notation language used for describing the GRAPHIC language shall be shortly explained here: Capital letters represent keywords (reserved words) of the language, the underlined part of the word is required, the rest of the word may be omitted or spelled differently:

For PLOT we can write PLO
or PLOT
or PLOTTHISOBJECT.

Terms of the language that are not reserved words are printed with small letters. These items are explained after the notation of the statement, e. g.:

PLOT object
"object" must be explained here.

Brackets are used to designate parts of a statement that are optional. Described by the notation:

OPEN [PLOT] [SIZE sx sy]

The following statements are valid:

OPEN PLOT SIZE 10 20
OP SIZ 10 20
OP

If default-values are assumed in the case information is missing, the values will be given in the describing text.

If an item may be repeated a deliberate number of times or missing, this is shown by an asterisk following a closing bracket:

PROCEDURE name [(parm [,parm]*)]

If one of several possibilities can be chosen for one item in a statement, this will be notified by braces:

STANDARD [UNIT] { MM | CM | M | INCHES | FOOT | FT | YARDS | YD }

There are two versions of the GRAPHIC-language implemented: One with its words close to the German language. This version is presently used in the IRE, the statements' syntax is listed in /28/. A second version of the GRAPHIC language has the same abstract syntax and the same semantics, but a different concrete syntax, the language words being taken from the English language. The descriptions in this paper refer to this second version of the GRAPHIC language. It is used by specifying "EGRAPHIC" instead of "GRAPHIC" as the first word of a program.

4.2 SET and DEFINE-statements

As a result of using ICES-CDL for implementing the GRAPHIC-language, statements of the form used in chapter 3.2 cannot be implemented effectively. CDL-defined statements must begin with the command name. For this reason all GRAPHIC-statements have to start with a valid statement keyword. Another shortcoming of CDL is the necessity for putting all alpha-strings in quotes (this is important for names in GRAPHIC). So the statement

L := line from P1 to P2

is to be worded in GRAPHIC as follows:

SET 'L' LINE FROM 'P1' TO 'P2'

The words "FROM" and "TO" are not significant for recognizing the meaning of the statement, so they may be omitted. All keywords (called "modifier" in CDL) can be abbreviated to a specified minimum (mostly 3 letters). Thus the statement reads in its shortest form:

SET 'L' LIN 'P1' 'P2'

In this notation the symbol ":=" of our demonstration language, meaning "Evaluate the expression to the right and store the result as an object pointed to by the name standing to the left", is replaced by the keyword "SET". In chapter 3.2 the necessity for a second operator was shown, meaning "Evaluate the expression to the right every time the name to the left is referenced and then use the result of the evaluation instead of the name. We used the symbol "=" for this operator, in GRAPHIC the keyword for it is DEFINE:

DEF 'L' LINE FROM 'P1' TO 'P2'

A GRPAHIC-statement starting with the keyword DEF can be compared to "statement functions" in FORTRAN. The general form of the SET and DEF-statement is as follows:

$$\begin{Bmatrix} \text{SET} \\ \text{DEFINE} \end{Bmatrix} \quad \begin{Bmatrix} \text{name} \\ \text{OBJECT [NAMED] name} \end{Bmatrix} \quad \left[\begin{Bmatrix} \text{AS} \\ = \end{Bmatrix}\right] \text{ objectspecification}$$

At this point only the form

$$\begin{Bmatrix} \text{SET} \\ \text{DEF} \end{Bmatrix} \quad \text{name} \quad \left[\begin{Bmatrix} \text{AS} \\ = \end{Bmatrix}\right] \quad \text{objectspecification}$$

will be discussed. The other form will be treated in the context of procedures (chapter 4.5.1).

"name" is a string of one to eight characters enclosed in single quotes. All characters except the single quotation mark are allowed, e. g. 'NAME', '123', '-+*Ø', 'GOODNAME'.

If "name" is not yet declared when the statement is executed, it will be implicitly declared within the block that contains the statement.

The "objectspecification" defines the object that is to be computed and that is to be identified by "name". Possible objectspecifications are described in the following chapter.

4.3 Object specifications

4.3.1 Graphical objects

4.3.1.1 Graphical elements

Graphical elements are specified by use of the SET or DEF statements.

All graphical elements in GRAPHIC are situated in a rectangular cartesian coordinate system. All specifications of coordinate values refer to this system.

In the following syntax and semantics of the graphical elements are explained. It should be noted that the valid syntactical expressions, which describe a graphical element, may take the place of any graphical object in other expressions of the GRAPHIC language. For this reason the following syntactical description does not show complete statements of the GRAPHIC language such as

$$\left\{\begin{matrix}\text{SET}\\ \text{DEF}\end{matrix}\right\} \quad \text{name} \quad \left[\left\{\begin{matrix}\text{AS}\\ =\end{matrix}\right\}\right] \quad \underline{\text{PO}}\text{INT} \; [\text{X}] \quad [=] \quad \text{l1} \quad [\text{Y}] \quad [=] \quad \text{l2}$$

but only the expression for the graphical element.

The point

$$\underline{\text{PO}}\text{INT} \quad [\text{X}] \quad [=] \quad \text{l1} \quad [\text{Y}] \quad [=] \quad \text{l2}$$

"l1" and "l2" are the x- and y-coordinates of the point. All input specifications representing a length (e. g. coordinates, height of characters, radius of a circle) have the same form:

$$\left\{\begin{matrix}\text{real number}\\ \text{integer number}\end{matrix}\right\} \qquad \left[\left\{\begin{matrix}\text{MM}\\ \underline{\text{C}}\text{M}\\ \underline{\text{M}}\text{ETER}\\ \underline{\text{IN}}\text{CHES}\\ \underline{\text{FOOT}}\\ \text{FT}\\ \underline{\text{YARDS}}\\ \text{YD}\end{matrix}\right\}\right]$$

If the unit specification is omitted, a standard unit is used. The standard unit is "METER" unless otherwise specified by a previous "STANDARD UNIT" statement (see chap. 4.7.2). Example for the point specification:

SET 'P' POINT 2 3.5

The text

The text specification is used to define a character-string.

```
TEXT text [HEIGHT height]
[WITH]  [X]  [=]  l1  [Y]  [=]  l2  [angle]
```

"text" is a text-string not containing single quotation marks and enclosed in single quotation marks. The length of the string is limited only by the fact, that the ICES-Command-Interpreter only allows a maximum length of 390 characters for problem language commands.

Following the keyword "HEIGHT" the height of the characters may be specified. The default value is 5 mm.

"l1" and "l2" are the coordinates of the left lower corner of the first character in the string. "angle" is the angle between the text-string and the positive x-axis. In GRAPHIC, all angle-specifications have the following form:

```
{ [{real   } [DEGREES]] [{real   } [MINUTES]] [{real   } [SECONDS]] }
{ [{integer}          ] [{integer}          ] [{integer}          ] }
{                                                                   }
{ {real   } RADIANS                                                 }
{ {integer}                                                         }
```

"real" and "integer" are real constants or integer constants in FORTRAN format.

If no unit is given, "DEGREES" is assumed. If a specification for angle is missing at all, zero degrees is assumed.
Examples for text:

SET 'T1' TEXT 'THIS IS A TEXT' HEIGHT 3 MM 1 CM 2 CM 30 DEGREES
SET 'T2' TEXT 'TEXT2' 5 IN 10 IN

The axis

The axis specification is used to specify a linear or logarithmic coordinate axis.

```
AXIS  [{LINEAR     }]
      [{LOGARITHMIC}]
```

[WITH] [ORIGIN [X] [=] 11 [Y] [=] 12]
[LENGTH 13]
[ANGLE a1]
MINIMAL [COORDINATE] 14
MAXIMAL [COORDINATE] 15
[TITLE text] [{LEFT / RIGHT}]
[NORMED [SCALING]]

"LINEAR" specifies a linear scaled axis, "LOGARITHMIC" a logarithmic scaled axis, if both are omitted the default is "LINEAR". After "ORIGIN" the coordinates of the axis' origin are specified, the default values are x = 0, y = 0.

"LENGTH" is the axis' length in paper coordinates (default: 0.2 METER)

"ANGLE" is used to specify the angle of the axis (default: 0 degrees). "MINI" and "MAX" must be specified, they represent the minimal and maximal problem coordinate value, i. e. the value to be written at the beginning and the end of the axis. After the keyword "TITLE" a text can be specified that is to be written at the axis (default is no title). The maximum length of the title is 60 characters.

"LEFT" or "RIGHT" are used to inform the system whether the annotation and the title shall be placed to the right (on the clockwise side) or to the left (on the counterclockwise side) of the axis. Default is "RIGHT". If "NORMED SCALING" is specified, the values for the minimal and maximal coordinates are adjusted in a way that the numbers at the tick marks assume rounded values. This is done only for linear axes.

The operation "TRANSFORMATION" (chapter 4.3.1.2) can be used to transform any object into the space defined by two AXIS-elements. The operations "X-AXIS" and "Y-AXIS" can be used to produce automatically suitable axes for a given object and a given size of a drawing (see chap. 4.3.1.2)

Examples:

```
SET 'XAX' AXIS ORIGIN 2 2 LENGTH 20 MINI 2 MAXI 15 TITLE 'X-AXIS'
    NORMED

SET 'YAX' AXIS LOG ORI 2 2 LENGTH 30 ANGLE 90 MINI 1
    MAXI 10000 TITLE 'Y-AXIS' LEFT
```

4.3.1.2 Graphical operations

Graphical operations, like graphical elements are specified using the SET or DEF-statement. An operation is performed upon graphical elements and a result is computed, that can be referenced by the name following the SET or DEF-keyword. In case of the SET-statement the computation of the result is done immediately when the statement is executed. The result of an operation specified by a DEF-statement is computed every time the name following the DEF-keyword is referenced.

In GRAPHIC there are several object-specifications that may be element-specifications or operation-specifications. These are the specifications for

- circles
- arcs
- polygons
- spline curves
- approximation curves.

E. g. if a circle is defined by 3 points, and all coordinates of the points are given, this is the specifications of a circle-element. If, on the other hand, one or more of the three points are given by their names, an operation has to be performed in order to compute the circle.

Circles

Circles can be defined in four different ways:

- by specifying central point and radius (1)
- by specifying central point and one periphery point (2)
- as a circle through three points (3)
- as the inscribed circle of a triangle (4)

CIRCLE [BY]

```
(1)  ⎧ CENTER p1 [AND] RADIUS r                                          ⎫
(2)  ⎨ CENTER p1 [AND] POINT p2                                          ⎬
(3)  ⎪ [THE] [THREE] POINTS p1 p2 p3                                     ⎪
(4)  ⎩ INSCRIBED [CIRCLE] [OF] [TRIANGLE] [WITH] [POINTS]                ⎭
       p1 p2 p3
```

"r" is the length of the radius, e. g. 5 INCHES
"pi" are point objects - either given by their coordinates, e. g. "3.5 CM 6 CM", "X = 2 Y = 3" or given by their names or as a result of an operation, e. g. "'P1'", "INTERSECTION OF 'L1' AND 'L2'".

Examples:

```
SET 'C1' CIRCLE CENTER 'P' RADIUS 15 MM
SET 'C2' CIR CEN 10 10 AND POINT 'P2'
SET 'C3' CIRCLE GIVEN BY POINTS 'P3' X = 1 Y = 2 'P4'
SET 'C4' CIR INS 2 2 3 3 2 4
```

Arcs

Arcs can be defined in seven different ways, only four of them will be described in detail below:

- arc specified by central point, radius (or curvature) and two angles (1)
- arc specified by radius, begin-point and end-point, an additional choice between the large circle and the small circle is necessary (2)
- arc given by three points (3)
- arc given by begin-point, end-point and arc length (4).

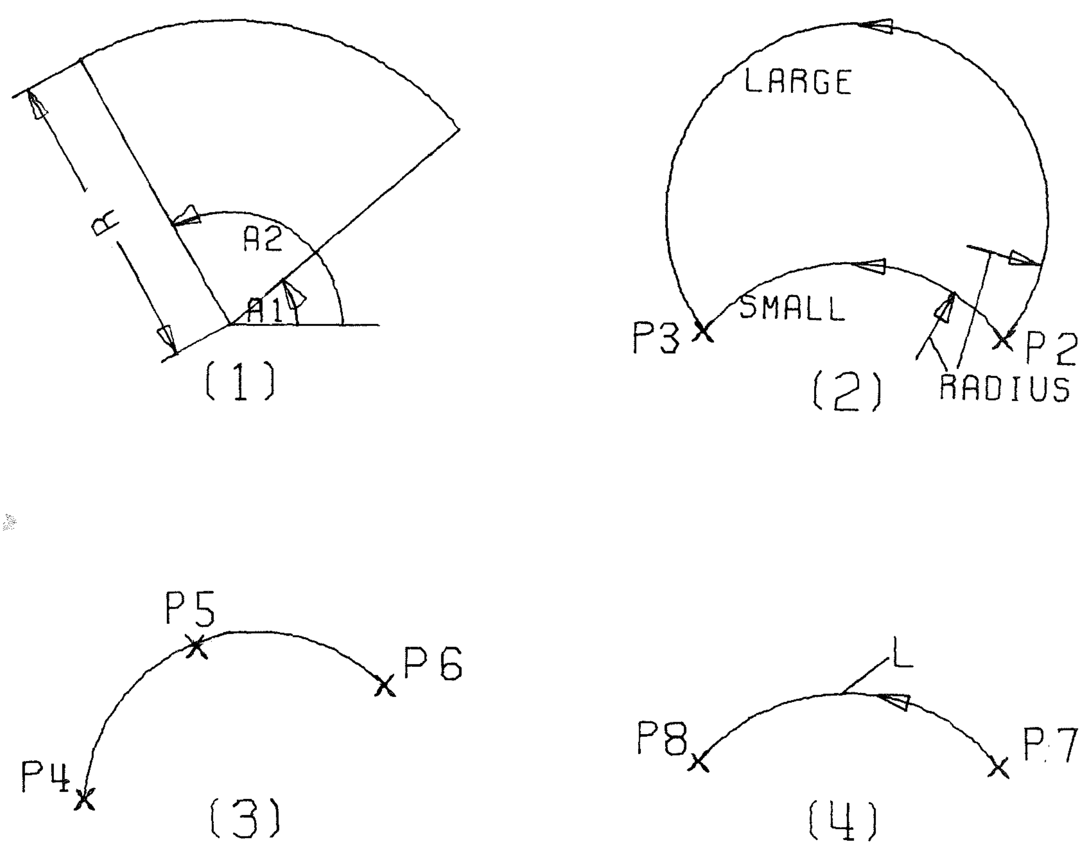


Fig. 15 Ways of specifying arcs

ARC [GIVEN] [BY]

(1) {RADIUS r / CURVATURE c} [AND] CENTER p1 [AND] ANGLES a1 AND a2

(2) {RADIUS r / CURVATURE c} [{LARGE / SMALL}] [AND] BEGINPOINT p2 [AND] ENDPOINT p3

(3) [THE] [THREE] POINTS p4 p5 p6

(4) BEGINPOINT p7 [AND] ENDPOINT p8 [AND] LENGTH l

"p1", "p2", ..."p8" are point objects, they may be specified by writing their coordinates, names of existing points or operations delivering a point as a result.
"ang1" and "ang2" are angles specified either by degrees, minutes, seconds or radians or by referring to the angle of a line, e. g.

"DIRECTION 'L'", if 'L' is the name of a line. After "LENGTH" the arc length is to be specified.

Examples:

```
SET 'A1' ARC RAD 5 INCH CENTER 'C' ANGLES 30 DEGREES AND
         DIRECTION 'LINE1'
SET 'A2' ARC CURVATURE 2. SMALL BEG 1.5 1. END 2.5 1.
SET 'A3' ARC GIVEN BY THE THREE POINTS 'P1' 'P2' 'P3'
SET 'A4' ARC BEG 0 0 END 1 1 LEN 3.5
```

Polygons, Splinefits and Approximations

Polygons, splines and approximations can be specified in three different ways:

- by listing the coordinates of the points of the curve (1),
- by specifying a list of one or more existing objects, like points or polygons, through which a new curve is to be created (2),
- by inserting a series of FORTRAN (IBM-E-level, /29/) statements defining the points of the curve (3).

```
{ POLYGON        [{CLOSED}]                                                        }
{                [{OPEN  }]                                                        }
{ SPLINE         [{CLOSED}]                                                        }
{                [{OPEN  }]                                                        }
{ APPROXIMATION  [OF]   [DEGREE n]   [INDEPENDENT [VARIABLE] {X}]                  }
{                                                        {Y}                       }

{ [THROUGH] [THE] [FOLLOWING] [POINTS] coordinates: x,y                             }
{                                    [coordinates: x,y]* END                       }
{ [THROUGH] [THE] [FOLLOWING] [COLLECTION] { object                    }            }
{                                          { (object [,object]* )      }            }
{ WITH n [VALUES] [ACCORDING] [TO]                                                  }
{        Fortran-statements, assigning                                              }
{        values to X(1)... X(n) and to                                              }
{        Y(1) ... Y(n)                                                              }
{        END                                                                        }
```

If "OPEN" is specified, an open polygon or splinefit is created; if "CLOSED" is specified, a connection is established from the

last point specified to the first point, thus creating a closed polygon orsplinefit. If neither "OPEN" nor "CLOSED" is specified, "OPEN" is assumed. Approximations may not be closed. When a splinefit element is plotted, a smooth curve is drawn through the specified points, using a modified splinefit technique /5/. Approximations are achieved by using polynomials found by the least square method. The independent variable for the approximation polynomial may be specified, default is "X". The user may also give the degree of the approximation polynomial, default is 1 (straight line). "n" must be greater or equal 0, if 0 is specified, a line representing the mean value is created.
"coordinates:x,y" is a pair of coordinate values, e. g. "2 5", "X 3 CM Y 1 INCH".
"object" is an object of the types: point, line, polygon, splinefit, approximation.

Examples:

```
SET 'P1' POL  1 1  2 2  3 3  3 2  3 1  END
```

Note: This statement is the only exception to the rule that only 4 continuation cards are allowed in the GRAPHIC language. While listing the coordinates of the points of a polygon, a spline or an approximation any number of cards may be used. A continuation hyphen is not required at the end of a card.

```
SET 'P2' POL 'SPLINFIT'
SET 'P3' POL CLOSED ('POINT1' 'LINE1' 'POLY1' ' POINT2')
SET 'P4' POL WITH 10 VALUES
       DO 1 I = 1, 10
       X(I) = I/10.
       Y(I) = EXP(X(I))
     1 CONTINUE
       END
SET 'S1' SPLINE 'P4'
SET 'S2' SPLINE CLOSED 1 1  1 2  2 3  3  2  3  1  2  0  END
DEF 'S3' SPL ('P1' POINT 5.1 5.3 'POINT1')
SET 'A1' APPROXIMATION DEGREE 3 TROUGH 'P4'
```

```
SET 'A2' APP INDEP VAR Y ( 'POLY1' 'POLY2' )
SET 'A3' APP INDEP Y DEGREE 2 WITH 10 VALUES
      DO 1 I = 1, 10
      Y(I) = I
    1 X(I) = ATAN(Y(I))
      END
```

The following graphical objects are always considered as operations.

Obtaining points and lines out of polygons

There are two GRAPHIC-operations, namely the NPOINT and the NLINE operation, which can be used to obtain the nth point or the nth line out of a polygon:

```
NPOINT  n  [{FORWARD | BACKWARD}]  [OUT]  [OF]  polygon

NLINE   n  [{FORWARD | BACKWARD}]  [OUT]  [OF]  polygon
```

The nth point or the nth line from the beginning of the polygon is created, if "FORWARD" is specified. In the case that "BACKWARD" is specified the counting of points or lines starts from the end of the polygon. "FORWARD" is the default-value. The expression "polygon" must be a polygon-specification.

Examples:

```
SET 'P1' NPOINT 5 OF 'POLY'
SET 'P2' NPO 1 BACKWARD OF 'POLY'
SET 'L1' NLINE 3 BACK OUT OF 'POLY'
DEF 'L2' NLI 10 'POLY'
```

The line

The line operation is used to create a vectorial linear connection between two points.

```
LINE [FROM] p1 [TO] p2
```

"p1" and "p2" have to be point objects, i. e. point elements, names of points or operations producing a point.

Examples:

```
DEF 'L1' LINE FROM 'P1' TO 'P2'
SET 'L2' LINE FROM 'P3' TO POINT 1. 1.5
SET 'L3' LINE FROM INTERSECTION OF 'L1' AND 'L2' TO 'P4'
```

The semicircle

The semicircle operation serves for creation of a semicircle from one point counterclockwise to a second point.

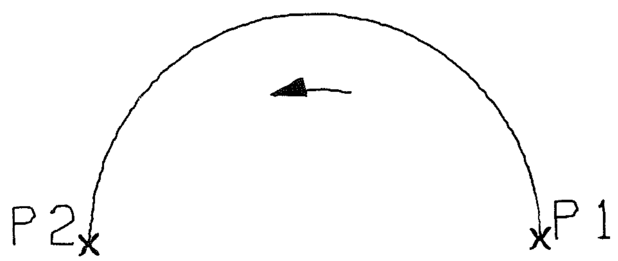


Fig. 16 The semicircle

SEMICIRCLE [FROM] p1 [TO] p2

"p1" and "p2" must be point objects.

Examples:

```
SET 'S1' SEMICIRCLE FROM POINT 10 CM 10 CM TO POINT 20 CM 10 CM
DEF 'S2' SEM 'P1' 'P2'
```

The intersection of two curves

The intersection operation is used to find the point(s) of intersection between two lines, a line and a circle or two circles. In the first case the result is a point, in the latter cases the result of the operation is a line, it's starting point and end point being the two intersection points.

INTERSECTION [{BETWEEN | OF}] ob1 [{AND | WITH}] ob2

"ob1" and "ob2" may be lines, polygons, circles or arcs. Arcs are treated as if they were a circle, i. e. they are expanded to a full circle. Polygons are treated as if they were lines, only the straight line connecting the first and the second point of the polygon is taken for computation of the intersection point(s).

We feel that it would be desirable to have a generalized intersection operation, delivering as a result all points of intersection of all kinds of curves (splines, arcs, lines, polygons, etc.) and delivering no result (the undefined object, see chap. 5. 1) if there is no intersection.

Example:

```
SET 'L1' LINE 'P1' TO 'P2'
SET 'L2' LINE 'P3' TO 'P4'
SET 'I1' INTERSECTION OF 'L1' AND 'L2'
SET 'C' CIRCLE CENTER 20 CM 20 CM RAD 10 CM
DEF 'I2' INT 'L1' 'C'
```

The extreme element

This operation is used to copy an unnamed element of a named object. To extract such an element it is identified by its relative position: "the uppermost", "the leftmost". By this way you can define a polygon by its points and extract a line, if it does have an extreme position.

```
EXTREM ELEMENT [AS]

{ {LOWER}  [{LEFT }] }
{ {UPPER}  [{RIGHT}] }   {POINT}   [OF]  object
{ {LEFT }  [{UPPER}] }   {LINE }
{ {RIGHT}  [{LOWER}] }
```

"object" may be any kind of single object or a collection, if you seek a point (POINT); "object" must be a single object or a collection containing lines (such as a polygon), if you seek a line (LINE).

The attributes upper, right, left, lower refer to the basic coordinate system.

The position of a line is given by the position of its central point.

Example (see fig. 17)

```
SET 'COL' ( POLYGON CLOSED 2 2 6 2 6 4 2 4 END CIRCLE CENTER 4 2
          RADIUS 1 )
SET 'P1' EXTR UP POINT 'COL'
SET 'P2' EXTR UP RIGHT POINT 'COL'
SET 'P3' EXTR LOWER POINT 'COL'
SET 'L' EXTR RIGHT LINE 'COL'
```

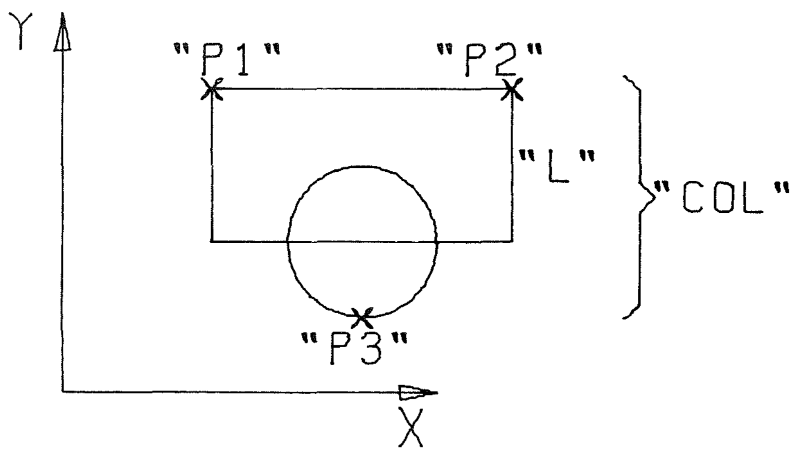


Fig. 17 Usage of the EXTREME-ELEMENT specification

Shades

The shade operation is used to specify shading of areas surrounded by polygons. A shade may be specified in one polygon or between two polygons.

```
SHADE
[DISTANCE [OF] [LINES]    dist] [ANGLE  angle]
{ [IN]  ob1                          }
{  BETWEEN ob2  [AND]  ob3           }
```

"dist" represents the distance between shading lines, default is 5 MM. By "angle" the inclination angle of the shading lines can be specified. If the ANGLE-option is omitted, 45 degrees is the default value.

"ob1", "ob2" and "ob3" have to be polygon objects. If shading IN a polygon is specified, the interior of it is shaded. If "ob1" is not a closed polygon, it is converted to a closed one by connecting it's first with it's last point.

If shading BETWEEN two polygons is specified, the first points of both and the last points of both are connected, thus creating a closed area to be shaded. Shading BETWEEN two closed polygons can be used to create windows in a shaded area.

Examples:

```
SET 'P1' POL CLOSED 0 0 4 0 4 4 0 4 END
SET 'P2' POL CLOSED 1 1 2 1 2 2 1 2 END
SET 'S1' SHADE DIST 0.1 INCH ANGLE 30 DEGREES IN 'P2'
PLOT( 'S1' 'P2' )
```

- The result is shown in fig. 18 a) -

```
SET 'S2' SHADE BETWEEN 'P1' AND 'P2'
PLOT( 'S2' 'P1' 'P2' )
```

- The result is shown in fig. 18 b) -

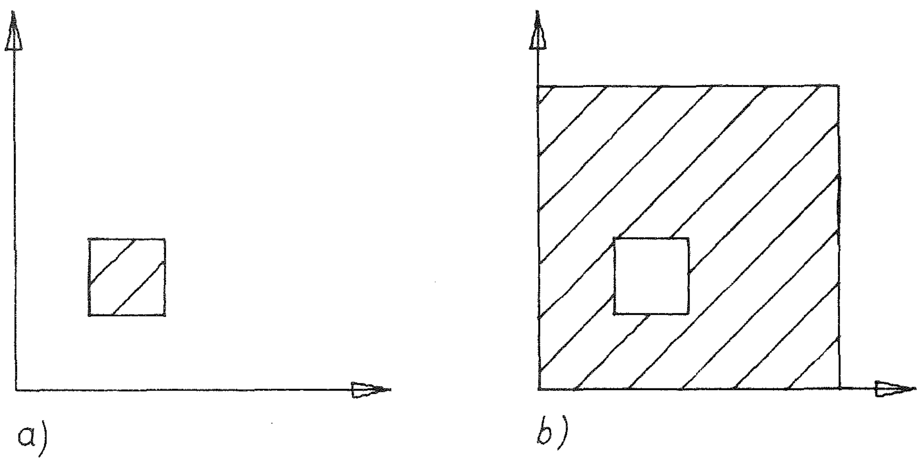


Fig. 18 Shades

The x-axis and y-axis operation

These operations are used to generate automatically suitable coordinate axes to a given object (or a collection of objects) for a specified size of the drawing.

```
{ X-AXIS }    [ { LINEAR      } ]
{ Y-AXIS }      { LOGARITHMIC }

[ ON { DIN [A] din {UPRIGHT    } } ]
     {             {BROADSHEET } }
     { DIMENSIONS  dimx  dimy    }

[ TITLE  text ]   [TO]  object
```

Either a linear scaled (LINEAR is default) or logarithmic scaled (LOGARITHMIC) axis is created, horizontal if X-AXIS is specified, vertical if Y-AXIS is specified. The situation of the axes on the drawing is shown by fig. 19.

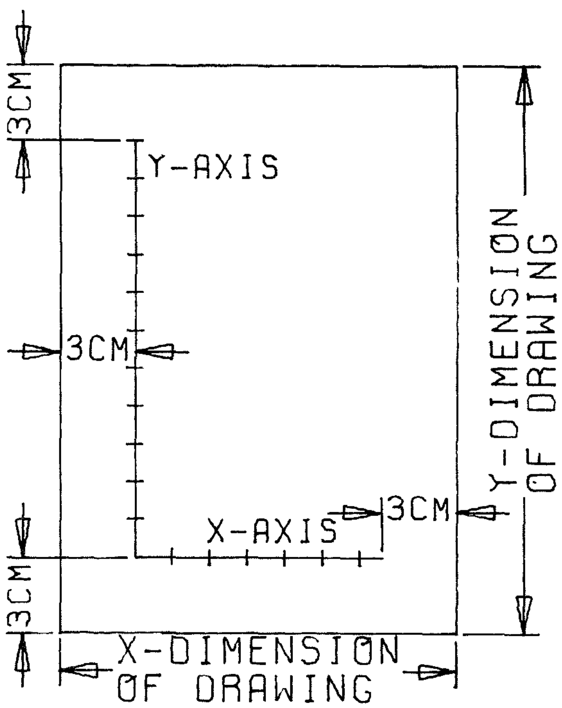


Fig. 19 Situation of axes created by the X-AXIS and Y-AXIS operation

The dimensions of the drawing are taken from the specifications after the keyword ON. "DIN A din", where "din" is an integer from 1 to 8, refers to the German standards for papersheet formats (DIN A 1 = 594 mm x 841 mm, DIN A 2 is half of DIN A 1, and so forth). After the keyword "DIMENSION" explicit dimensions of the drawing can be specified. If a dimension specification is omitted, the values are taken from the latest executed size-of-drawing statement (OPEN PLOT-statement, see chap. 4.4) or from the builtin default values, if no size-of-drawing statement has occured since the beginning.

After "TITLE" a text to be written at the axis may be specified, "text" is a string containing up to eight characters and enclosed in single quotes. If "TITLE" is omitted, "X-AXIS" is default for the x-axis, "Y-AXIS" for the y-axis. The coordinate values to be written at the tick marks of the axis are taken from "object", which may be any object or a collection of objects.

The TRANSFORMATION-operation (see later this chapter) may be used to transform an object in a way that it's size and position will correspond to the notation at the axes.

Examples:

```
DEF 'AX1' AS X-AXIS TO ( 'OB1' ' OB2' 'OB3' )
DEF 'AX2' Y-AX LOG TITLE 'PRESSURE'  TO ( 'OB1' 'OB2' 'OB3' )
SET 'AX' X-AXIS ON DIMENSIONS 10 INCH 15 INCH TO 'POLYGON'
```

The plot-specification operation

In order to specify plotting options for an object, like point symbols or line types (dashline, centerline, e. g.) the specification operation can be used.

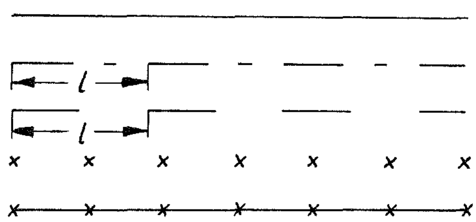
```
SPECIFICATIONS [{OF}]
               [{BY}]

               {THROUGH   }
               {CENTERLINE}
[CURVETYPE     {DASHLINE  }] [SYMBOLNUMBER number] [LENGTH l]
               {DOTTED    }
               {MARKED    }

[EVERY n] [AND] [HEIGHT height] [OF] object
```

The five linetypes are:

THROUGH
CENTERLINE
DASHLINE
DOTTED
MARKED



Default is "THROUGH"
In the case of "CENTERLINE" and "DASHLINE" the distance from the beginning of one dash to the beginning of the next one may be specified following the keyword "LENGTH". The default value for "l" is 10 mm.

In the case of "DOTTED" and "MARKED" curves, the "SYMBOLNUMBER" and the "EVERY" options can be used. By "SYMBOL" the number of a point symbol is given. Which number refers to which symbol depends on the plotter used. In our installation, the Calcomp conventions are followed /5/. By use of the "EVERY" option the system is informed that a symbol is to be plotted at every nth data point. The default value for "n" is 1.

The "HEIGHT" option is used to specify the height of the point symbols. All the options can be used for line and polygon objects. For splines and approximations only the THROUGH, DOTTED and MARKED curvetypes are possible. For texts, the HEIGHT option may be used for changing the height of characters. The "SYMBOL-NUMBER" and the "HEIGHT", options are used for specifying the kind of representation of point objects.

<u>Examples:</u>

```
SET 'POL' SPECIFICATION OF CURVETYPE DASHLINE OF POLYGON
    0 0  1 1  2 0  END

SET 'CEN' SPECIFICATION OF CURVETYPE CENTRALLINE OF LINE FROM
    'P1' TO 'P2'

SET 'SPL' SPECIFICATION OF CURVETYPE MARKED SYMBOL 3 EVERY 5
    HEIGHT 0.1 INCH OF 'SPLINE'

DEF 'P1' SPEC SYM 9 HEI 7.5 MM 'P'

SET 'BIGTEXT' SPEC HEIGHT 10 INCH 'SMALLTEX'
```

The following operations are used to specify linear transformations to be carried out with graphical objects. These operations are:

- Shifting an object in x- and in y-direction
- Enlargement or diminution
- Rotation
- Transforming an object in a way that two points of it will be placed upon two specified points on the drawing.
- Transformation of an object in a way that it will correspond to the notations of two predefined coordinate axes.

In GRAPHIC there is no statement that can be used to change directly the value of an object. Such a feature could be compared to a statement like "increment I by N;" in some languages. Instead an operation can be performed on an object and the value may be assigned again to the name of this object. The corresponding statement in a mathematical operation language would be: "I := I + N "

Shifting

This operation may be used to create an object that is a copy of another object, shifted in vertical or horizontal direction.

```
SHIFTING
{ TOWARD  p1                                         }
{ [BY]  [X]  [=]  xshift  [Y]  [=]  yshift           }    [OF]  object
```

If "TOWARD" is specified, "p1" must be a point object, and "object" is shifted by the x and y-coordinate values of this point. If TOWARD is not specified, the values by which "object" is to be shifted have to be specified explicitely. "Object" may be any object, including a collection.

Examples:

```
DEF 'OB1' SHIFTING TOWARD 'P1' OF 'OB2'
SET 'OB3' SHIFT X = 2  Y = 3 OF 'OB3'
SET 'OB4' SHIFT 5 CM 4 CM ( 'OB5' 'OB6' 'OB7'  'OB8' )
```

Enlargement and diminution

These operations are used to create an enlarged or diminished copy of an object. The center of the linear transformation, i. e. the point of the object that does not change it's position, can be specified.

```
{ ENLARGEMENT }
{ DIMINUTION  }   [BY]
[FACTOR] [X] [=] facx [Y] [=] facy
[RESPECTING [TO] p1] [OF] object
```

"facx" and "facy" are the factors by which "object" is to be enlarged or diminished. "object" may be any kind of graphical object. An enlargement by factors fx and fy delivers the same result as a diminution by factors 1/fx and 1/fy. If "RESPECTING" is specified, the center of the diminution or enlargement is the "p1", which must be a point object. If RESPECTING is not specified, the center is the point x = 0, y = 0.

Examples:

```
SET 'A' ENLARGEMENT BY FACTORS 2.5 5.0 OF 'A'
DEF 'B' DIMINUTION 10 10 RESPECTING TO 'P' OF ( 'B1' 'B2' 'B3' 'P' )
SET 'C' ENL 2 2 RES INTERSECTION OF 'L1' AND 'L2' OF ( 'L1' 'L2' )
```

The enlargement and diminution operations can be used for mirroring, too. Mirroring is achieved by specifying negative enlargement factors.

Examples:

```
SET 'A1' ENL -1  1  'A'
    (Mirroring at the y-axis)
SET 'A2' ENL 1  -1  'A'
    (Mirroring at the x-axis)
SET 'A3' ENL -1 -1  RESPECTING TO 'P' of 'A'
    (Mirroring at point 'P')
```

The rotation

The rotation operation is used to produce an object by rotating another object around a given point.

```
ROTATION
{ [BY]  [ANGLE] angle                 }  [AROUND  pointobject]
{ [BY]  DIRECTION [OF]  lineobject    }
 [OF]  object
```

The angle of the rotation can be specified in two ways. Either it is given directly in degrees, minutes, seconds or radians or it is taken from the inclination angle of a specified lineobject. In the latter case the keyword "DIRECTION" must be used. The point around which the rotation shall take place may be specified following the keyword "AROUND". If this option is omitted, the central point of the rotation will be x = 0, y = 0. "object" may be any graphical object.

The image operation

This is a special transformation operation that transforms an object in a way that two specified points of the object are placed upon two specified points of the drawing. This is achieved by shifting and enlarging the object in a suitable way. No rotation is performed.

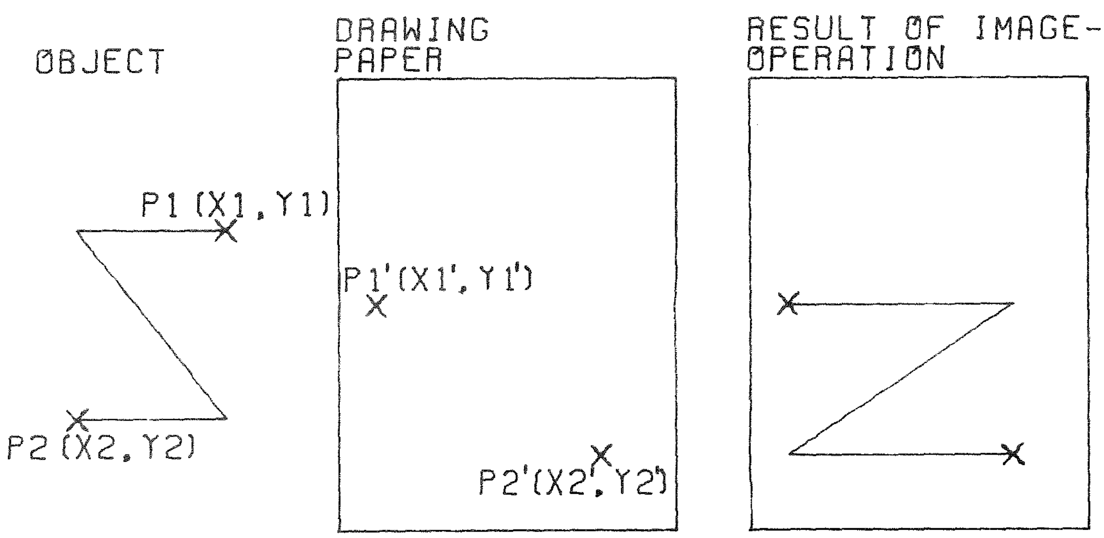


Fig. 20 The image-operation

```
  IMAGE
⎧ RECTANGULAR [SECTOR] [X] x1 [Y] y1 [AND] [X] x2 [Y] y2          ⎫
⎨ ACCORDING [TO] [X] x1 [ON] [X] x1' [AND] [Y] y1 [ON] [Y] y1'    ⎬
⎩         [AND] [X] x2 [ON] [X] x2' [AND] [Y] y2 [ON] [Y] y2'     ⎭
[OF] object
```

If the keyword RECTANGULAR is specified, the points P1 (x1, y1) and P2 (x2, y2) of the object will be placed on the left lower corner of the drawing and the upper right corner respectively. The size of the drawing must be defined by a preceding OPEN PLOT statement (see chap. 4.4).

If the keyword ACCORDING is used, the points P1 (x1, y1) and P2 (x2, y2) of the object will be placed upon the points P1' (x1', y1') and P2' (x2', y2') of the drawing. "object" may be any graphical object, including a collection.

Examples:

```
SET 'IM1' IMAGE RECTANGULAR  0 0 AND 10 10 OF ( 'OB1' 'OB2' 'OB3' )
SET 'IM2' IMA ACCORDING TO 1 ON 1 AND 1 ON 1 AND 150 ON 10 AND
    250 ON 20 OF 'DRAWING'
```

The transformation operation

This operation can be used to transform an object in a way that its situation on the drawing will correspond to the notation of two predefined coordinate axes.

```
TRANSFORMATION
[TO] [AXES] ax1  [AND]  ax2  [OF] object
```

"ax1" is to be the x-axis for the object,
"ax2" the y-axis respectively. The two axes need not begin at the same point nor need they be rectangular to each other. One or both of the axes may be of logarithmic type. They must not be parallel. Fig. 21 illustrates the effect of the transformation operation.

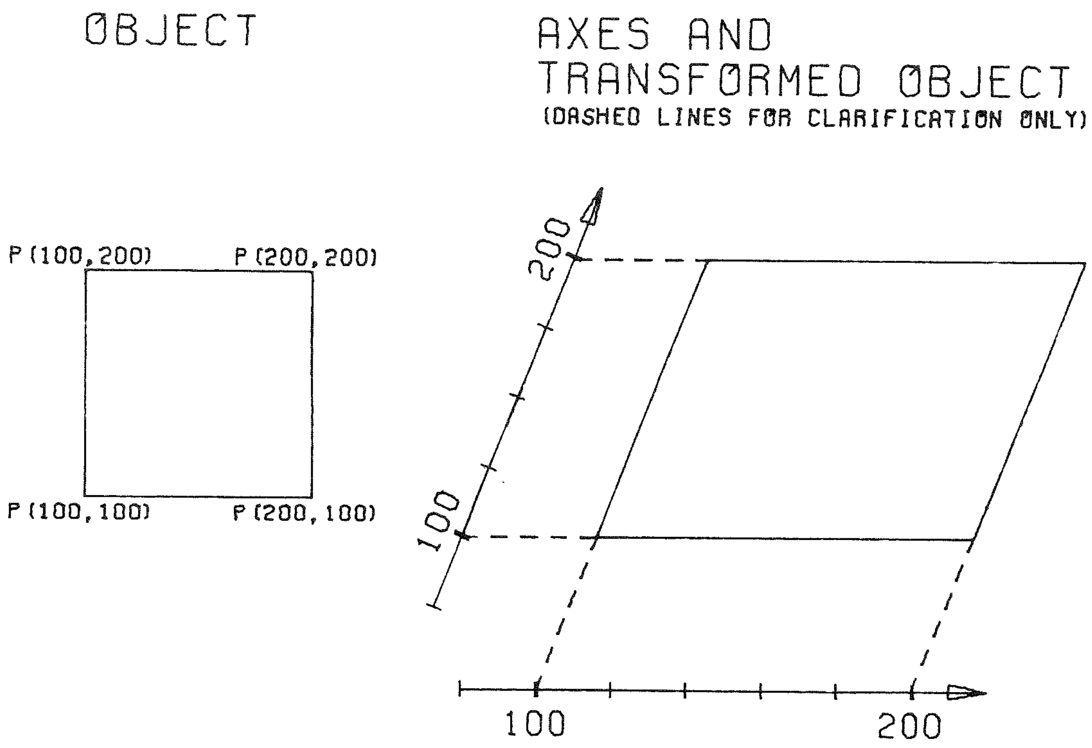


Fig. 21 Result of transformation-operation

"object" may be any graphical object, including a collection. However, texts and axes cannot be completely transformed logarithmically. The logarithmic transformation, if required, will transform in these cases only the origin and the angle of texts and origin, length and angle of axes, but will not change the shape of these objects.

The transformation operation is very often used to fit any object defined in some physical coordinates (as temperature or pressure) into the size if a sheet of paper. It is advantageously used in connection with the axis specification or the x-axis and y-axis operation.

Examples:

```
1) SET 'AX1' AXIS ORIGIN 1 1 LENGTH 20 MINIMAL COORDINATE 0
       MAXIMAL COORD 250 TITLE 'T(SEC)' NORMED SCALING

   SET 'AX2' AXIS LOG ORI 21 1 MINI 1 MAXI 1000 TITLE 'P(ATM)'

   SET 'TRANS' TRANSFORMATION TO AXES 'AX1' 'AX2' OF 'ANYOB'

   PLOT ( 'AX1' 'AX2' 'TRANS' )

2) SET 'XAX' X-AX TO ( 'OB1' 'OB2' )
   SET 'YAX' Y-AX TO ( 'OB1' 'OB2' )
   SET 'TRA' TRANS 'XAX' 'YAX' OF ( 'OB1' 'OB2' )
   PLOT ( 'XAX' 'YAX' 'TRA' )
```

4.3.2 Logical-arithmetical objects

Logical and arithmetical variables and operations have been introduced into GRAPHIC since they are needed in program control statements, like DO and IF. In the present implementation it is not possible to use an arithmetic element, i. e. the name of a variable, instead of a number in graphical statements (e. g. SET 'A' POINT 'X' CM 'Y' CM). This feature would be desirable however.

4.3.2.1 Logical-arithmetical elements

Real numbers, integer numbers (in a FORTRAN-like sense) and logical values are logical-arithmetical elements. They can be specified using the SET or the DEF statement.

The logical values are either TRUE or FALSE. The element type must not be declared, it is declared implicitely by the first use of an element name.

$$\left\{\begin{array}{l}\text{real number}\\ \text{integer number}\\ \text{TRUE}\\ \text{FALSE}\end{array}\right\}$$

Examples:

```
SET 'A' = 144              'A' will be an integer element
SET 'B' = 3.14             'B' will be a real element
SET 'C'   FALSE            'C' will be a logical element
```

4.3.2.2 Logical-arithmetical operators are listed below:

+	(plus)	prefix operator
-	(minus)	prefix operator
**	(exponentiation)	
*	(multiplication)	
/	(division)	
+	(addition)	
-	(subtraction)	

The following logical operators may be used in logical expressions:

¬	(not)	prefix operator
&	(and)	infix operators
|	(or)	

Comparison operators:

=	(equal)
¬ =	(not equal)
>	(greater than)
¬>	(not greater than)
<	(less than)
¬<	(not less than)

Logical-arithmetical elements and operations are used to form expressions, the same rules have to be applied as in higher level languages like ALGOL or FORTRAN (e. g. for the priority of the operators).

Examples:

```
SET 'A' = 'B' + 'C'
SET 'A1' = 'B1' > 'B2' | 'C1' ¬> 2 & 'D1'
SET 'A2' = ( 'A1' + 2 - 'N' ) ** 'A2'
DEF 'SUM' = 'A' + 'B' + 'C'
```

4.3.3 Nesting of object specifications and collections of objects

Some examples in chapter 4.3.1 already **indicated that object specifications** may be nested in the GRAPHIC language. In those cases where any object is allowed in the description of the statements, one of the following can be specified:

- a graphical element (POINT 2 3)
- a graphical operation (SHIFTING OF 'A')
- a name representing a graphical object ('A')
- a collection of objects ((POINT 2 3 SHIFTING OF 'A' 'A')).

A collection is a compilation from a number of existing objects. After the execution of a collection command the members of the collection can be referenced by one name.

[COLLECTION] [OF] (object [,object]*)

SET 'C' COLLECTION OF ('D' 'E' 'F')
SET 'C1' ('D1' 'E1' 'F1')

The collection object, consisting of a list of objects enclosed in a pair of parenthesis, can be used in many statements instead of a single object. These cases are specified in the statement descriptions.

In cases where only special kinds of objects may be specified (e. g. a point object), they can be given by:

- an appropriate element (POINT 3 2)
- the name of an appropriate element ('A')
- an operation delivering as a result an appropriate element (INTERSECTION OF 'C1' and 'C2')

Nestings and collections make the GRAPHIC language extremely flexible. There is a restriction of the level of nestings due to the restriction of the level of recursive calls in ICES-CDL. The maximum level allowed depends upon the operations that are nested. A level of ten will be accepted in most cases.

Example: The arrow 'AR' shall be placed in the dotted position above point 'P' rectangular to line 'L' in figure 22.

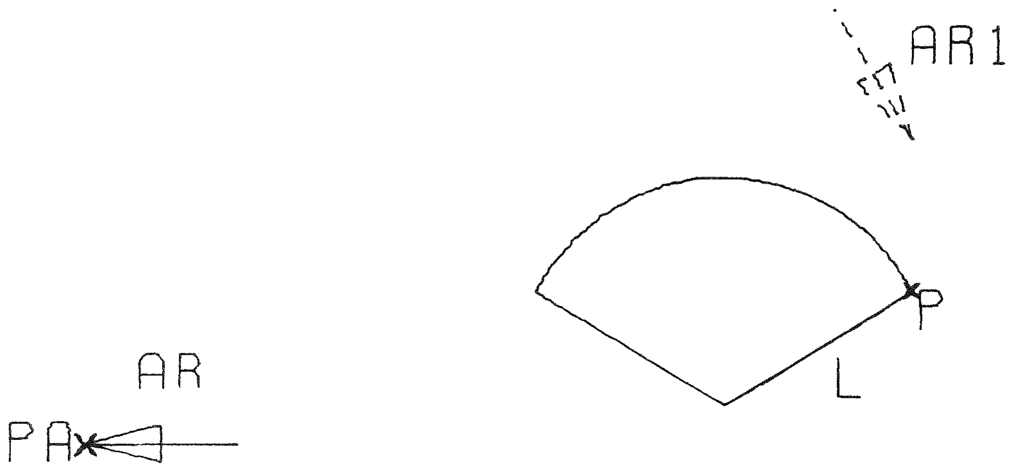


Fig. 22 Shifting and rotating an object

This may be achieved by the statement:

SET 'NEWARROW' SHIFT TOWARD SHIFT 0 2 OF 'P' OF ROTATION 90 DEGREES OF ROTATION DIRECTION 'L' AROUND 'PA' OF 'AR'

4.4 Output statements

OPEN PLOT

This statement is used to inform the system that the following graphical outputs are to be placed on a new drawing. The left lower corner of the new drawing is the zero point of the coordinate system used for the graphical objects. The size of the drawing may be specified in the OPEN PLOT statement. A drawing is a rectangular sector of the paper. The different drawings created by a GRAPHIC program are placed on the paper sheet in a way that they will not interfere with each other and use the paper (almost) as good as possible. In the first executed OPEN PLOT statement of a GRAPHIC program the paper type (white or graph paper and the pen type (ballpoint or ink) may be specified.

```
OPEN [PLOT] [ON] [FORMAT]

  [ { DIN [A] din { UPRIGHT     }                } ]
  [ {             { BROADSHEET  }                } ]
  [ {                                            } ]
  [ { SIZE [X] [=] xsize [Y] [=] ysize           } ]

  [ { WITH } ]  [ { GRAPH } ]  [PAPER]  [AND]  [ { BALLPOINT } ]
  [ { ON   } ]  [ { WHITE } ]                  [ { INK       } ]
```

The value after the keyword "DIN" refers to German paper sheet formats, after the keyword "SIZE" the size of the drawing may be specified explicitely. If neither "UPRIGHT" nor "BROADSHEET" is specified, the "UPRIGHT"-format is default. If a size specification is omitted, the size is taken from the foregoing OPEN PLOT-statement. By writing "GRAPH" or "WHITE" the paper type is selected, the "BALLPOINT" or "INK" specification serves for selecting a pen.

Examples:

```
OPEN PLOT ON DIN A 4 WITH WHITE PAPER AND INK
OPEN
OP SIZE 5 INCH 10 INCH
OP
OP DIN 5 BROAD
```

The PLOT statement

The PLOT statement is used to plot graphical objects. A scissoring option can be specified if only a rectangular cut out of the object is to be plotted.

```
PLOT [[WITH]  CUT  [X] [=]  x1  [Y] [=] y1
      [AND]        [X] [=]  x2  [Y] [=] y2  ]
      [OF]  object
```

If CUT is specified, only those parts of "object" are plotted that lie within the frame, that is defined by $P1(x_1, y_1)$ and $P2(x_2, y_2)$ (see fig. 23).

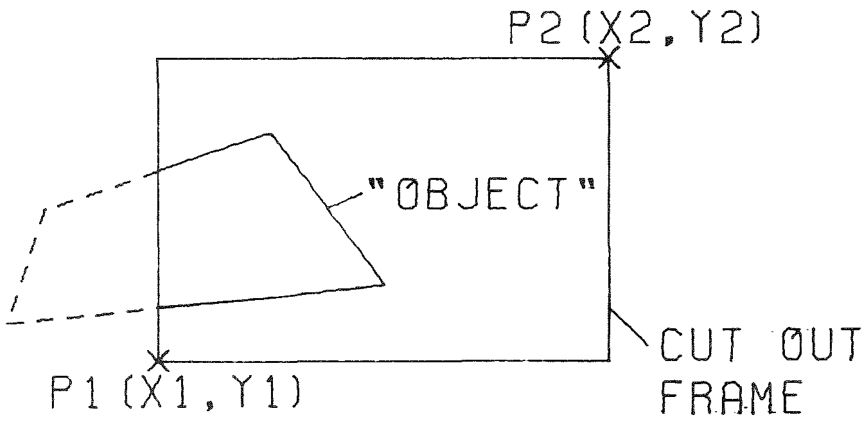


Fig. 23 The scissoring feature

"object" may be any graphical object, including a collection.

Examples:

```
PLOT 'A'
PLOT CUT 10 10  20  30  ( 'A' 'B' 'C' )
PLOT SPEC  SYMBOLTYPE 5 HEIGHT 10 MM INTERSECTION OF 'LINE1'
     AND 'LINE2'
```

The PRINT statement

This statement serves for printing logical or arithmetical values on the printfile. It may be used for program control or test purposes.

PRINT arithmetical-or-logical-expression

Examples:

```
SET 'A' = 3
PRINT 'A'                              (prints: 3)
PRINT 'A' + 3 - 'A' ** 5 + 10          (prints: -227)
PRINT 'A'  = 5                         (prints: FALSE)
```

4.5 Procedure and Declaration Statements

4.5.1 Definition and invocation of procedures

A number of GRAPHIC statements may be combined to form a procedure by placing a PROCEDURE statement in front of them and an END statement at the end. A CALL statement is used to invoke the procedure. A procedure has to be defined before it can be called (this is due to the interpretative nature of the execution of GRAPHIC statements).

Procedure definition:

PROCEDURE name [(fpar [,fpar]*)]
GRAPHIC-statements
END [RETURN object]

Procedure call:

CALL name [(apar [,apar]*)]

A procedure may be defined with (formal) parameters. Every formal parameter "fpar" is represented by a name, i. e. by one through 8 characters enclosed in single quotes. The appearance of a name in the list of formal parameters of a PROCEDURE definition implicitly declares this name as local to the procedure.

When a procedure is called, the same number of actual parameters must be specified in the list of actual parameters "apar". (The term "argument" is avoided here, because it is used in a different sense in the GRAPHIC data structure.) Every actual parameter has the form of a graphical or logical-arithmetical object. Upon execution of a call every formal parameter is implicitly DEFINED as its corresponding actual parameter. ("Call by reference" according to /30/).

Thus

```
PROC 'PLOT' ( 'A' )
PLOT ROTATION 30 DEG OF 'A'
END
CALL 'PLOT' ( POLYGON  1  1  2  2  END )
```

is equivalent to

PLOT ROTATION 30 DEG OF POLYGON 1 1 2 2 END

Results may be returned from the called procedure to the calling procedure in three ways:

- by returning one object as a result of a function procedure call
- by assigning values to the actual parameters of the particular call
- by the use of global names.

Functional procedures are characterized by the keyword RETURN (followed by an object specification) following the END word of the END statement of the procedure. Functional procedures are called by using the corresponding CALL statement as an expression (or object) in a GRAPHIC statement.

Thus

```
PROC 'ROT' ( 'WHAT' 'AROUND' )
SET 'WHAT' ROTATION 30 DEG AROUND 'AROUND' OF 'WHAT'
END RETURN ( 'WHAT', TEXT 'WHAT' )
PLOT CALL 'PLOT' ( POL  1  1  2  2  END, POINT  1  1 )
```

is equivalent to

```
PLOT ( ROTATION 30 DEG AROUND POINT  1  1  OF POL  1  1  2  2
     END, TEXT 'WHAT' )
```

One should note that the CALL always implies an evaluation of the object returned from the called procedure to the level of an element (graphical, logical or arithmetical).

Since formal parameters are considered to represent local names, assignment of a value to a formal parameter does not affect the corresponding actual parameter. E. g. in the preceeding example 'WHAT' is an object which is completely contained within is procedure block. However, the second form of the DEFINE and SET statement permits this operation.

$$\left\{\begin{matrix}\underline{\text{DEFINE}}\\ \text{SET}\end{matrix}\right\}\ \underline{\text{OBJECT}}\ [\underline{\text{NAMED}}]\ \text{name}\ \left[\left\{\begin{matrix}\text{AS}\\ =\end{matrix}\right\}\right]\ \text{objectspecification}$$

The expression

OBJECT NAMED name

describes an operation which when executed, delivers the object which (at this time) has been defined as this name. Provided that the object found again is a name, then the SET or DEF operation will use this name as the one to which a value is assigned. As an example,

```
DEFINE 'A' AS 'B'
SET OBJ 'A' = POINT 1. 1.
```

is equivalent to

```
SET 'B' = POINT 1. 1.
```

Since calling a procedure implicitly means a definition of the formal parameters, this type of statement can be used in a procedure to return objects through the parameter list.

As an example

```
PROC 'SUB' ( 'A' 'B' )
SET OBJECT 'B' ROTATION 90 DEG OF 'A'
END
SET 'A1' POINT 1. 0.
CALL 'SUB' ( 'A1' 'B1' )
```

is equivalent to

```
SET 'B1' = ROTAT 90 DEG OF 'A1'
```

Objects may be passed to and from a procedure also by use of names valid in the outer block, i. e. global objects. Global objects must be declared prior to the definition of a procedure.

Example:

```
SET 'A' POINT 1 2
DECLARE 'B'
PROC 'C'
SET 'B' SHIFT 1 INCH 2.5 INCH OF 'A'
END
CALL 'C'
```

is equivalent to

SET 'B' SHIFT 1 INCH 2.5 INCH OF 'A'

However, if 'B' had not been declared (or implicitly declared) prior to the PROC 'C' statement, then 'B' would be local to 'C' and the call 'C' statement would have no effect in this example.

4.5.2 The DECLARE statement

This statement is used to make names of objects local to the block in which the DECLARE statement is situated. If they are used in the outermost block, the names are made global. The INITIAL option serves for assigning an initial object to the name.

```
DECLARE  name  [INITIAL object]
        [name  [INITIAL object] ]*
```

Examples:

	Valid objects
DECL 'A' 'B' INIT POI 1 2 'C'	'A', 'B'point, $'C'_1$
BEGIN	
DECL 'B' INIT CIRCLE CENTER 0 0 RAD 1	'A', 'B'circle, $'C'_1$
DECL 'D'	'A', 'B'circle, $'C'_1$, 'D'
PROC 'C'	'A', 'B'circle, 'D', 'C' local
DECL 'B'	'A', 'B'new, 'D', 'C' local
END	'A', 'B'circle, 'C'procedure
END	'A', 'B'point, $'C'_1$

4.6 Program control statements

4.6.1 Do-loops

Do-loops are used if a number of statements shall be repeated several times.

```
DO  { arith. expr.  [TIMES] }
    { WHILE  log. expr.     }
```

number-of-GRAPHIC-statements

END

In the first form of the DO-statement, the arithmetic expression "arith. expr." is converted into an integer value i before the group of statements between DO and END are executed for the first time and the statements in the DO-group are executed i times.

If "WHILE" is specified, the logical expression "log. expr." is evaluated and, if the value is TRUE, the DO-group statements are executed. Then "log. expr." is evaluated again. The DO-group is executed successively until the evaluation of "log. expr." yields the value FALSE.

Example:

```
DO 10 TIMES
CALL PLOTOB ( 'A' )
SET 'A' SHIFT 1 2 OF 'A'
END
SET 'A' = 2.
SET 'C' = 20
DO WHILE 'A' > 'B' & 'C' > 0
SET 'C' = 'C' - 1
SET 'A' = 'A' + 0.1
CALL 'EVAL' ( 'OB' 'A' 'B' )
SET 'OB' ENLARGEMENT 1.1  1.1  'OB'
END
```

4.6.2 IF-THEN-ELSE-FI

The IF-statement is used to execute a piece of program depending on the result of a logical expression.

```
IF log. expr. THEN statement1
[ELSE  statement2]
FI
```

If the evaluation of the logical expression "log. expr." yields the value TRUE, the statement "statement1" is executed. Otherwise, if "ELSE" is specified, the "statement2" is executed. If "ELSE" is omitted, "statement1" is only executed in the case that the result of "log. expr." is TRUE, while no operation is executed for a "log. expr." resulting in FALSE. "statement1" and "statement2" may be single statements or a group of statements between "DO" and "END" or a block between BEGIN and END.

Examples:

```
IF 'A' > 'B' THEN DO
CALL 'AGTB'
END
ELSE CALL 'ALEB'
FI

IF 'A' & 'B' & 'C' & 'D' THEN DO
PLOT ( 'PA' 'PB' 'PC' 'PD' )
OPEN PLOT
END
FI
```

4.7 System commands

System commands are GRAPHIC statements that are executed immediately wherever they are specified, even in the definition of a procedure. System commands perform some kind of action. They do not deliver a result.

4.7.1 The TAKE OVER statement

This statement serves for taking over graphical information from an interface file previously written by a program producing a plot output. Special routines intercept all calls to the plotter software and instead write information on the interface file.

By executing the TAKE OVER command, graphical elements are created from the information on the interface file and a name is generated for every one of the elements. The name for the collection containing all the elements taken over must be specified. The generated names of the single elements can be displayed by a successive plot-statement.

TAKE [OVER] name [FROM] [FILE] [nn]

"name" is the name of the overtaken graphical collection. If "nn" is specified, this number refers to the interface file with the DD-name "FTnnF001". The default value for nn at present is 13.

The names generated for all single elements of the graphical collection start with the first three characters of "name". After the TAKE OVER command is executed the object with the name 'ZEIGNAME' contains a collection of texts, representing the generated names.

If 'ZEIGNAME' is plotted together with "name", the generated names will be written at the appropriate graphical elements.

Examples:

```
TAKE OVER 'PICTURE1'
PLOT SHIFT 3 CM 3 CM ( 'PICTURE1' 'ZEIGNAME' )
OPEN PLOT SIZE 10 10
TAKE OVER 'PICTURE2' FROM FILE 14
PLOT SHIFT 1 1 OF 'PICTURE2'
```

4.7.2 Time, Standard, Test, Trace

TIME

This command prints out date and time on the standard printfile.

Example:

```
TIME
```

Standard unit

The standard unit command is used to change the unit taken for length specifications, if no unit is specified explicitely.

```
                                   ⎧ MM    ⎫
                                   ⎪ CM    ⎪
                                   ⎪ METER ⎪
STANDARD [UNIT]                    ⎨ INCH  ⎬
                                   ⎪ FOOT  ⎪
                                   ⎪ FT    ⎪
                                   ⎪ YARDS ⎪
                                   ⎩ YD    ⎭
```

Example:

```
STANDARD UNIT CM
SET 'A' POINT  1  2
SET 'B' TEXT 'ABC' HEIGHT 6 MM  1  1
STAN INCH
SET 'C' POINT  5  6
```

The test and trace commands

These commands are used for testing and debugging purposes when new features are to be incorporated into the GRAPHIC system. "TEST" prints out the complete list of data contained in the data pool in a readable form (the printing routines used are described in /8, 10/).

The command: TRACE {ON | OFF} [TIME]

switches on or off a trace of subroutine calls. A message is printed at the beginning and before the end of every subroutine. If "TIME" is specified, at the end of a subroutine CPU-time and elapsed time since the start of the job are printed out additionally.

4.7.3 Storage and retrieval of graphical information

The users of the GRAPHIC-system have the possibility to save and retieve graphical information. In many situations it is convenient to store the content of the Graphical Data Pool (GDP) and thus the graphical task for a long period. When, during the process of testing a GRAPHIC-program, the graphical information is kept on a secondary storage device, only incorrectly specified objects have to be redefined in a new task. This way the processing time for composing a drawing can be reduced.

For these operations a sequential dataset is established. This dataset may contain several GRAPHIC-records. Each represents the content of the GDP of one task and is identified by a name included in its head. The records are written with the programs for dynamic array - I/O /8, 10/, which require a logical record length of 80 bytes in the dataset.

4.7.3.1 The RESERVE- and RELEASE-statement

In order to change libraries or any dataset in the normal jobstream in a MVT or MFT environment, it is necessary to reserve the dataset for exclusive use for this time.

{RESERVE | RELEASE} [{DSNAME = | DATASET | FILE}] 'dsname' [ON] [{DISK | VOLUME =}] 'volume'

[WITH] [DDNAME] 'ddname'

These commands enable the GRAPHIC-programmer to use the capabilities of the OS-Assembler-Macros ENQ and DEQ /31/. ENQ creates a list of combinations of dsnames and volumes, which are to be used exclusively. 'ddname' is needed to control the validity.

While processing the following commands the I/O-datasets are implicitely protected, when the corresponding declarations (FILE nn DSNAME = 'dsname' VOLUME = 'volume') are specified.

4.7.3.2 Storing graphical information

The command PUT 'name' stores the datastructure contained in the GDP and associates the structure with the name 'name'. 'name' identifies this GRAPHIC-record in the sequential dataset. The name may consist of one to eight alphanumerical characters. While saving information on secondary storage the datastructure in the GDP will not be changed.

The complete PUT-command has the following form:

```
PUT 'name'  [WITH]  [KEY]  ['key']  [ON]  [FILE]  [nn]

[ON]  [ [{DATASET  }]  'dsname'  [ON] [[{DISK    }]  'volume'] ]
         {DSNAME = }                     {VOLUME =}
```

With 'key' one can protect a GRAPHIC-record against unauthorized destruction. If it is omitted, '   ' is assumed.

The other declarations are optional and may be used to change the default values for implicite reservation of datasets as mentioned in the RESERVE- and RELEASE-statement. The specification FILE nn corresponds to a DD-name 'FTnnF001'.

All datasets must be initialized before their first use.

4.7.3.3 Reading graphical information from secondary storage

```
GET 'name'  [FROM]  [FILE]  [nn]

[ON]  [ [{DATASET}]  'dsname'  [ON] [[{DISK    }]  'volume'] ]
         {DSNAME }                       {VOLUME =}
```

GET 'name' causes the system to read information from GRAPHIC-record 'name' into the GDP. Upon execution of this command the previous structure of nodes in the GDP is deleted and the environment becomes identical to what it was, when the corresponding PUT was executed. The user can now continue to manipulate the new structure.

All other specifications of the command can be overridden in a similar way as in the PUT-statement, if one does not want to use the default options for FILE, DATASET or VOLUME.

4.7.3.4 Deletion of graphical information on secondary storage

The command is worded as follows:

```
DELETE 'name' [WITH] [KEY] 'key' [ON] [FILE] [nn]

[ON]  [ [{DATASET }] 'dsname' [ON][[{DISK    }] 'volume']]
         {DSNAME =}                  {VOLUME =}
```

DELETE 'name' 'key' deletes a GRAPHIC-record identified by 'name'from secondary storage. It will only be executed, if the user also specifies an appropriate protection key. The space in the dataset is available for new disposals. The other specifications are used in the same manner as in the PUT-or GET-command.

4.7.3.5 File-Utility-command

The FILE-command is used to handle datasets with GRAPHIC-records.

```
      {INITIALIZATION}
FILE  {INFORMATION   }  [ON] [FILE] [nn]
      {REPAIR 'name' }

[ON]  [ [{DATASET }] 'dsname' [ON][{DISK  }] 'volume' ]
         {DSNAME =}                 {VOLUME}
```

Before the first datastructure from GDP can be saved into a dataset, this dataset must be initialized with an endword using FILE INITIALIZATION.
This endword must also be restored with FILE REPAIR, if a job is terminated abnormally, while executing the PUT- or DELETE-command.
With FILE INFORMATION the user can get a table of contents with the names, keys and creation dates of all GRAPHIC-records.

4.7.4 The compile, link and go commands

These commands enable the user to compile, link and execute programs in the GRAPHIC-go-step. Hence it is possible to change or enlarge the content of libraries, while processing other GRAPHIC-jobs. This capability is very helpful for "flying" expansion of the GRAPHIC-system.

```
COMPILE [WITH] [INPUT] [FROM] [FILE] [nn]
   :  } INPUT
** EOF
```

The input after the COMPILE-statement can be made up of several ICETRAN-programs. After the last program a card containing ** EOF in column 1 through 5 must be inserted. The input is expected from file FT05F001 respectively SYSIN. For all other sources FILE nn (FTnnF001) must be specified. A corresponding DD-card is required.

```
LINK { TEMPORARY
       STANDARD
       [WITH] [OUTPUT] [ON] DATASET 'dsname' [ON] [{VOLUME
                                                   DISK}] 'volume' }

     [LOAD] [MODULE] 'name1' [WITH] ENTRY 'name1' ... 'name6'
     [AND] [PROGRAMS 'name7' ... 'name16']
```

With the LINK-command the object-modules are linked. The load-modules built up are stored into the following kinds of libraries:

temporary, **the standard library for GRAPHIC-modules**
or any other specified in the LINK-statement.

All libraries can be declared as "SHR" in the corresponding SYSLMOD-DD-card. 'name1' is used to declare the name of the load-module. 'name2' through 'name6' are alias names and 'name7' to 'name16' are other programs also to be linked into the load-module 'name1'.
The GO-statement causes the execution of a load-module 'name'.

```
GO 'name'
```

4.8 The different modes of GRAPHIC

Depending on whether a command expressed in the GRAPHIC language is executed immediately after it has been processed by the command interpreter or whether the execution takes place at a later time, the terms "execution modes" and "programming mode" are used.

4.8.1 The programming mode

Any GRAPHIC command, which is not a system command, is converted into an equivalent internal node structure. When the command was contained in a group (DO, IF) or in a block (BEGIN, PROCEDURE) it will not be executed (i. e. its interval representation will not be parsed by the parser program) until the containing block or group itself is executed. Hence, as long as there is an explicitly specified group or block open, GRAPHIC is called to be in the "programming mode".

4.8.2 The execution mode

Any GRAPHIC system command, whether it is found in the outermost block (which begins with GRAPHIC and ends with END GRAPHIC) or in a contained block or group, is executed immediately. Hence, during processing of a system command, GRAPHIC is called to be in the "execution mode".

Any GRAPHIC command which is part of the outermost block of GRAPHIC is executed immediately after its conversion into its corresponding internal representation. This shall be illustrated by the following example:

```
GRAPHIC
  :                          execution mode
BEGIN
  :                          programming mode
END
  :                          execution mode
DO WHILE 'N' < 3
  :                          programming mode
END
  :                          execution mode
END GRAPHIC
```

4.8.3 The immediate mode

It is possible to ask for immediate execution of a GRAPHIC command while being in the programming mode. To perform this, the command must be preceded by the prefix command "!".

Syntax:

```
        !  any-GRAPHIC-command
```

The command preceded by ! is called to be in the immediate mode.

Example:

```
GRAPHIC
SET 'A' = ......
DO 3 TIMES
SET 'A' SHIFTING  +3  -2 OF 'A'
! PLOT 'A'
PLOT 'A'
END
END GRAPHIC
```

In this example the first of the two PLOT commands is preceded by a ! prefix command. Hence it will be executed with the "present" definition of 'A', which is valid before the DO-loop.

Hence, the above GRAPHIC program is equivalent to

```
GRAPHIC
SET 'A' ....
PLOT 'A'
DO 3 TIMES
SET 'A' SHIFTING  +3  -2 OF 'A'
PLOT 'A'
END GRAPHIC
```

The possibility to use the immediate mode is of no great use for batch processing. However, in an interactive use of GRAPHIC, it may be helpful to modify graphical information immediately without having to leave the programming mode. The capability which is thus achieved may be considered as the equivalent of a "desk computer mode" which is provided by several interactive systems based on mathematics oriented programming languages.

Not only whole commands may be executed in the immediate mode but also any graphical object specification. The syntax is similar:

```
! any-graphical-object
```

Example:

```
SET 'A' POINT  1  1
SET 'B' POINT  2  2
PROCEDURE 'PLOT'
SET 'FIGURE' ( 'A', 'B', ! LINE FROM 'A' TO 'B' )
PLOT 'FIGURE'
END
```

Whenever procedure 'PLOT' is invoked by a CALL, the objects which are assigned to 'A' and 'B' just prior to the point of invocation, will be plotted together with a line from point 1 1 to point 2 2. The above program is equivalent to

```
DECLARE 'A'
DECLARE 'B'
SET 'LINE' = LINE FROM POINT  1  1  TO POINT  2  2
PROCEDURE 'PLOT'
SET 'FIGURE' ( 'A', 'B', 'LINE' )
PLOT 'FIGURE'
END
```

5. Data structure

5.1 Introduction

In chapter 3.1 different methods for representing graphical objects in a GDPS were described and a directed graph structure was proposed. The basic concept of this structure shall be repeated:

- Every object is represented by a node in the structure
- Every object may have an arbitrary number of sons and an arbitrary number of fathers
- The relation between a father and its sons, and vice versa, is established by special objects, called references
- All references connecting a father with all of his sons are situated on a ring
- All references connecting a son with all of his fathers are situated on a second ring.

All the objects, including references are stored in a linked list, called the "node list". A stack is maintained for all places in the list that are not occupied by a node. If a new object is to be generated, the first free list position is taken from the free place stack. If an object is destroyed, its list position is added to the stack.

When the last item is removed from the free place stack, the list is expanded automatically. The feature of dynamic arrays offered by ICES has proved to be extremely helpful for implementing the data structures described here.

In the following chapters the object nodes, the reference nodes and the attributes of object nodes will be described in detail.

5.2 Nodes

5.2.1 Object nodes

Object nodes contain a structural part for representation of the relations between objects and a part containing the description of the object itself. The latter part is called "attribute substructure" or "attribute set" or just "attributes", it has a different form for the different types of objects. For description of attributes see chap. 5.2.3.

The relation part of objects is the same for all types of objects. It consists of pointers up and down and of a pointer to the set of attributes. The downward pointer of graphical or arithmetical elements is empty. The downward pointer of operations points to the object or objects upon which the operation is to be performed. If we take a look at two operations, the semicircle from one point to another one and the shift operation, we soon recognize that there are different kinds of operation subobjects. see fig. 24.

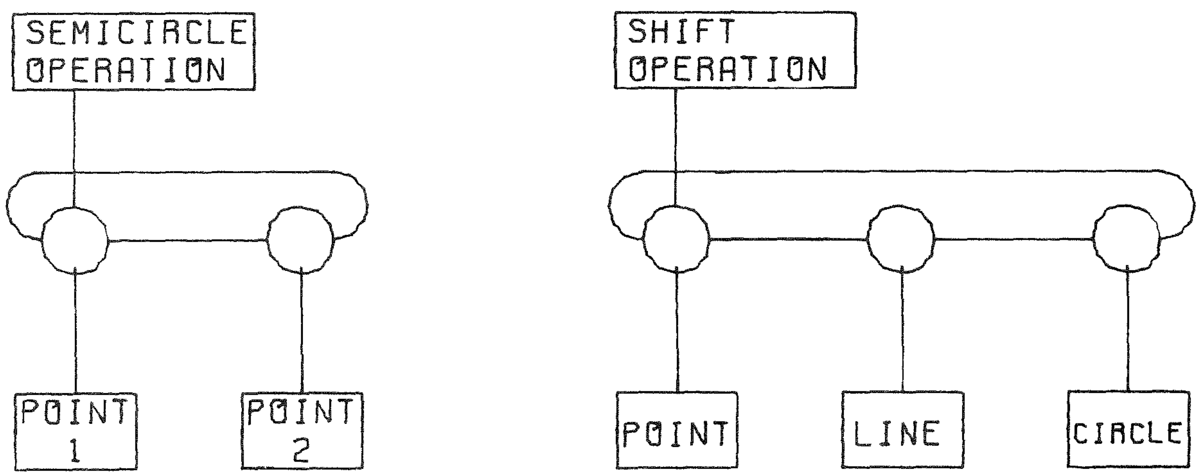


Fig. 24 Subobjects of operations

In the first case, the operation must have two and only two sons, which must be points, the order of sons is important.

On the other hand, a shift operation may have any number of sons, a change of the order of sons does not affect the result of the operation.

For this reason, an object in the GRAPHIC has <u>two</u> pointers to subobjects. One points to those subobjects that have to be in a fixed order. The ring comprising these subobjects is an ordered circular list. The sons of an object that are contained in this ordered circular list are called arguments of the object. So we say: The semicircle operation must have two arguments. The second downward pointer of an object points to a set of subobjects that may be in an arbitrary order and of any number. These sons of an object form a set ring.

We call this kind of subobjects operands of the object. The shift operation has no arguments and an arbitrary number of operands. An operation may possess both arguments and operands. An example is the rotation around a given point. The point is the argument of the operation, the objects to be rotated are the operands. Fig. 25 shows the corresponding structure. In the illustrations showing data structures the downward pointer to the operands is emerging from lower edge of the rectangle representing the object. The argument pointer is beginning at the right edge of the rectangle.

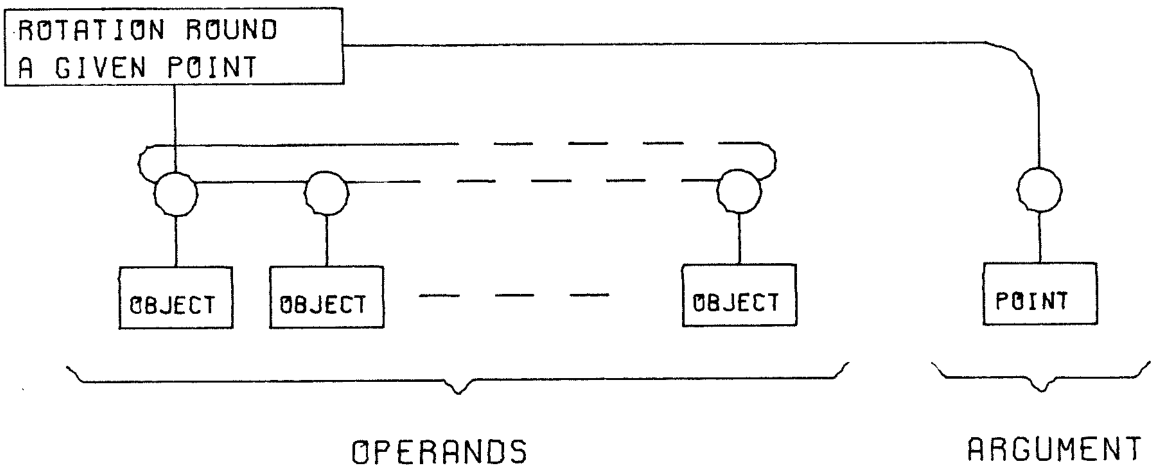


Fig. 25 Operands and argument of the rotation-operation

An operation is performed with its given arguments (if any are present) once for every one of its operands. If an operation has no operands, it is performed once with its arguments.

Every object thus possesses four pointers:
- one to its father
- one to its arguments
- one to its operands
- one to its attribute set

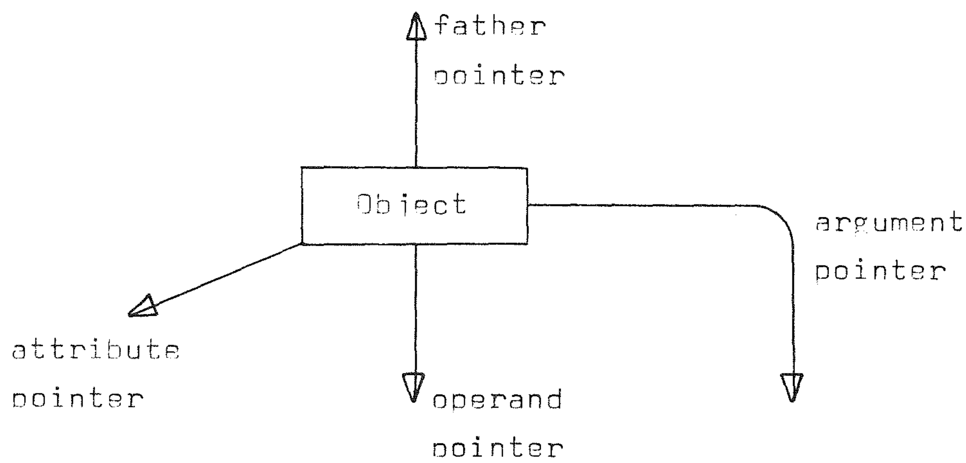


Fig. 26 Pointers of an object

Any one of the pointers may be empty, i. e. the null pointer.

5.2.2 Reference nodes

References are used to connect objects with each other. They do not possess attribute sets. Since references are no operations but only data elements representing relations between objects, they need not have a pointer to arguments and operands, but just one downward pointer. We have seen that an object has only one pointer to arguments, operands and fathers, although it may possess more than one of each. Thus the task of connecting the fathers or the sons of an object is left to the references. For this purpose references have pointers to form two circular lists: One for comprising all sons of an object (this is called the "ring with common superobject, RSUP") and another one for comprising all fathers of an object (ring of common subobject, RSUB). Figure 27 clarifies the use of both rings. In this figure only operand pointers are shown.

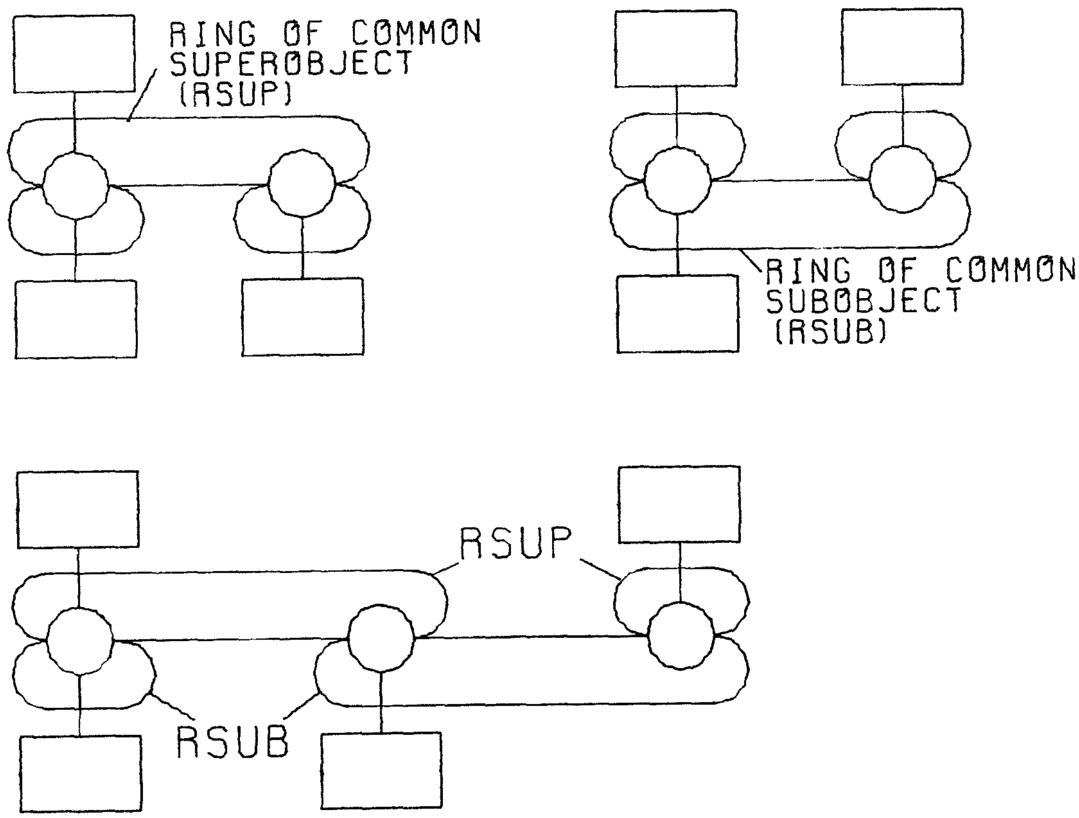


Fig. 27 Reference rings

There is one RSUP for the arguments and one for the operands of an object.

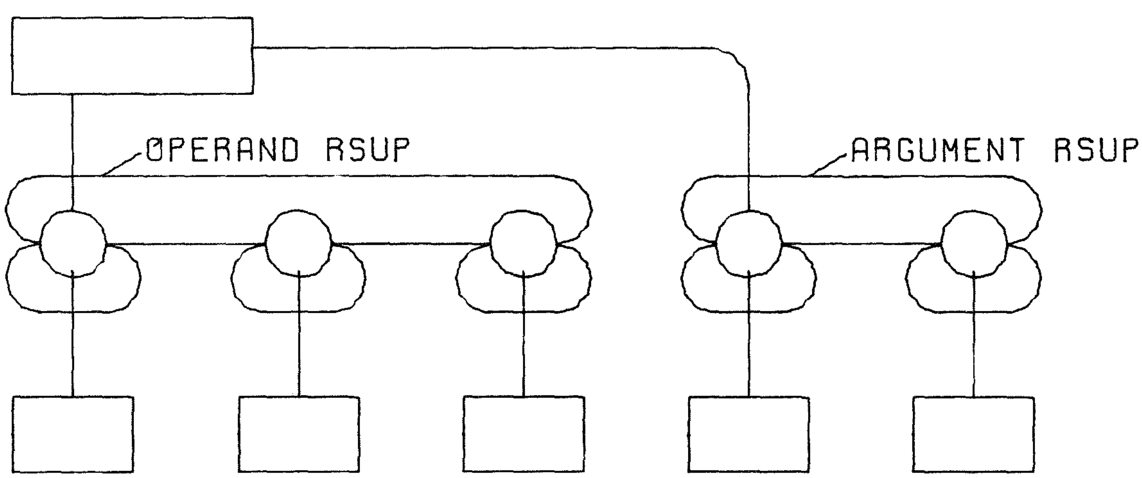


Fig. 28 Operand and argument rings of an object

The RSUP and the RSUB rings of a reference node are implemented by use of left and right pointers. Doubly linked circular lists are used because an element can easily be included into or deleted from the list without having to parse through the whole ring. Every reference possesses 6 pointers:

- one to the father
- one to the son
- the left and the right pointer of the RSUP
- the left and the right pointer of the RSUB.

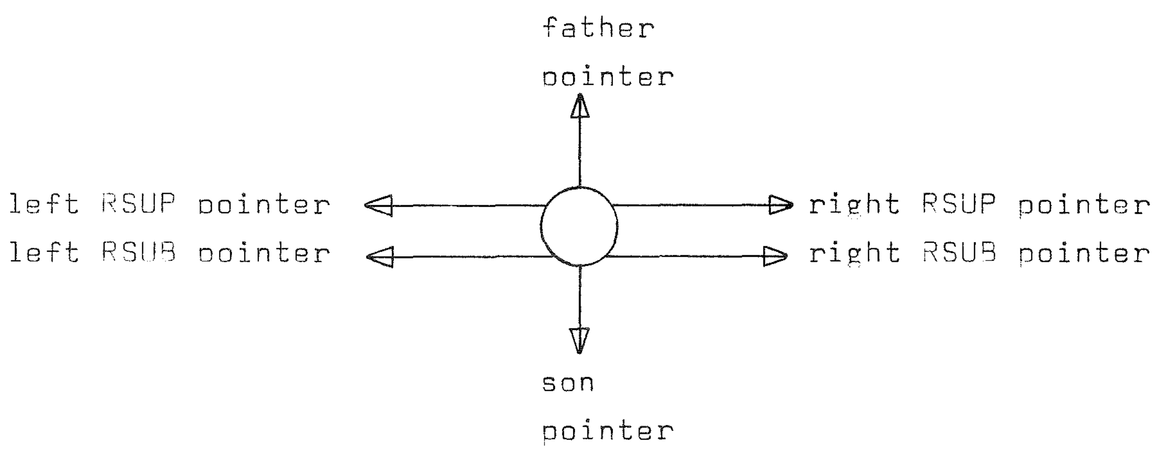


Fig. 29 Reference pointers

5.3 Types of objects and their attributes

5.3.1 Introduction

Objects in the GRAPHIC data structure differ only in their attribute set. All objects possess at least one attribute: the <u>object type</u>. According to the type of the object the remaining part of the attribute set is built up. Following object types are possible:

- graphical elements
- graphical operations
- logical-arithmetical elements
- logical-arithmetical operations
- collections

- names
- evaluate
- define
- control objects (procedure, if)
- actions
- name reference
- undefined object

The different types of objects and their attributes will be described in the following chapters.

5.3.2 Graphical elements

Graphical elements contain the basic graphical information, they do not depend on other objects, hence they have neither operands nor arguments; they can be plotted immediately. Graphical elements are: points, polygons, splinefit and approximation curves, texts, coordinate axes, arcs, circles and shades. In the attribute sets of the elements the information describing them is stored. All elements possess the attributes: object type, element type and number of dimensions. The remaining attributes are different depending on the element type and the dimensions. For a polygon-element in two dimensions e. g., the remaining attributes are number of points and the x- and y-coordinates of the points. At present all elements implemented in GRAPHIC are two-dimensional. A possible extension to three dimensions is described in /18/.

5.3.3 Graphical operations

Graphical operations are objects that, when executed, create a graphical element, a collection of graphical elements or a graphical operation according to the type, the arguments and operands of the operation. Table 1 gives a survey of graphical operations, the number and kind of arguments and operands they require and the kind of results they deliver when executed.

Operation	Number and kind of arguments	Number and kind of operands	Result
Line	2 point elements	none	line element (i. e. polygon with two points)
Semicircle	2 point elements	none	arc element
Intersection of lines or polygons	2 line elements	none	point element
Intersection of lines and circles	2 elements, at least one circle	none	line element
Nth point or line out of polygon	1 polygon element	none	point element or line element
Extreme element (leftmost, uppermost etc.)	any	none	point element or line element
Shading the interior of a polygon	1 closed polygon element	none	shade element
Shading between two poly-gons	2 polygon elements	none	shade element
x-axis y-axis to an object	any GE	none	axis element
Plot specifications (dotted lines, point sym-bol types etc.)	none	any GE	collection of elements according to number and types of operands

Table 1 Graphical operations

Operation	Number and kind of arguments	Number and kind of operands	Result
Shifting	none	any GE	
Shifting by the coordinates of a point	1 point element	any GE	
Enlargement Diminution	none	any GE	
{Enlargement / Diminution} with a specified point as the center of the transformation	1 point element	any GE	collection of elements according to member and types of operands
Rotation around the origin	none	any GE	
Rotation around a given point	1 point element	any GE	
Rotation around a given point by the declination angle of a line	1 point element and 1 line element	any GE	

Table 1 (cont.)

Operation	Number and kind of arguments	Number and kind of operands	Result
One of the linear transformation operations (shifting, enlargement, diminution, rotation) = op1	none	one of the linear transformation operations = op2	1 linear transformation operation comprising the tasks of op1 and op2
Image operation (lin. transf. so that 2 points of an object are placed on 2 points of the drawing)	none	any GE	collection of elements according to number and types of operands
Transformation of an object according to two axes	2 axis elements	any GE	
Circle through the 3 points of a triangle	3 point elements	none	1 circle element
Inscribed circle of a triangle	3 point elements	none	1 circle element
Circle given by central point and radius	1 point element	none	1 circle element

Table 1 (cont.)

Operation	Number and kind of arguments	Number and kind of operands	Result
Arc through 3 points	3 point elements	none	1 arc element
Arc given by begin and end point and arc length	2 point elements	none	1 arc element
Polygon through a number of objects	1 collection of point elements, polygon elements, splinefit elements and approximation elements	none	1 polygon element
Splinefit curve through a number of objects		none	1 splinefit element
Approximation curve through a number of objects		none	1 approximation element

Table 1 (cont.)

For execution of the operations every operation node possesses an appertaining routine which is called when the node is parsed. The attribute sets of operation objects contain at least the object type - graphical operation - and the type of operation. If further information is needed for executing the operation, it is also contained in the attribute set. E. g., the line operation needs no additional attributes, the operation "circle given by radius and central point" needs the value of the radius to be stored in the attribute set. For linear transformation the attribute set contains the transformation matrix.

5.3.4 Collections

Collection objects are nodes in the data structure possessing no arguments and a deliberate member of operands. They are used to combine objects that are to be referenced and manipulated as a whole. E. g. collections are used to comprehend the parameters of a GRAPHIC procedure. Collection objects have only one attribute, the object type.

5.3.5 Names

Names in the data structure correspond to the names used for identifying and referencing objects in the GRAPHIC language. The name itself, i. e. the character string used as an identifier is the only attribute of the name object (besides the object type). Name objects possess one argument, which is either one object or one collection of objects, they do not have any operands. A name object associates the character string in it's attribute set with the whole substructure below the argument. Because the names are integrated as nodes into the structure, no reference to a name table is required while parsing the structure. If an object is to be destroyed, e. g., all subobjects of it must be destroyed unless a named object is met in the structure. Named subobjects must not be destroyed. If names were not integrated into the structure, the name table would have to be referenced at every node, in order to ensure that no named subobject is destroyed. When the data structure is to be printed out in a readable form, it is also useful to meet the names in the structure while parsing it.

5.3.6 Evaluate and define

The evaluate-object and the define-object are used to control the parsing of the data structure. The define-object is used to name its substructure. Before the structure is parsed, the define-object is the structure's top. After the structure is parsed, the structure's top is a name-object. The substructure of the name is the result of the parsing of the original structure. The evaluate-object causes the parser-program to evaluate the substructure of the evaluate-object and to pass the result of the evaluation to the object above itself.

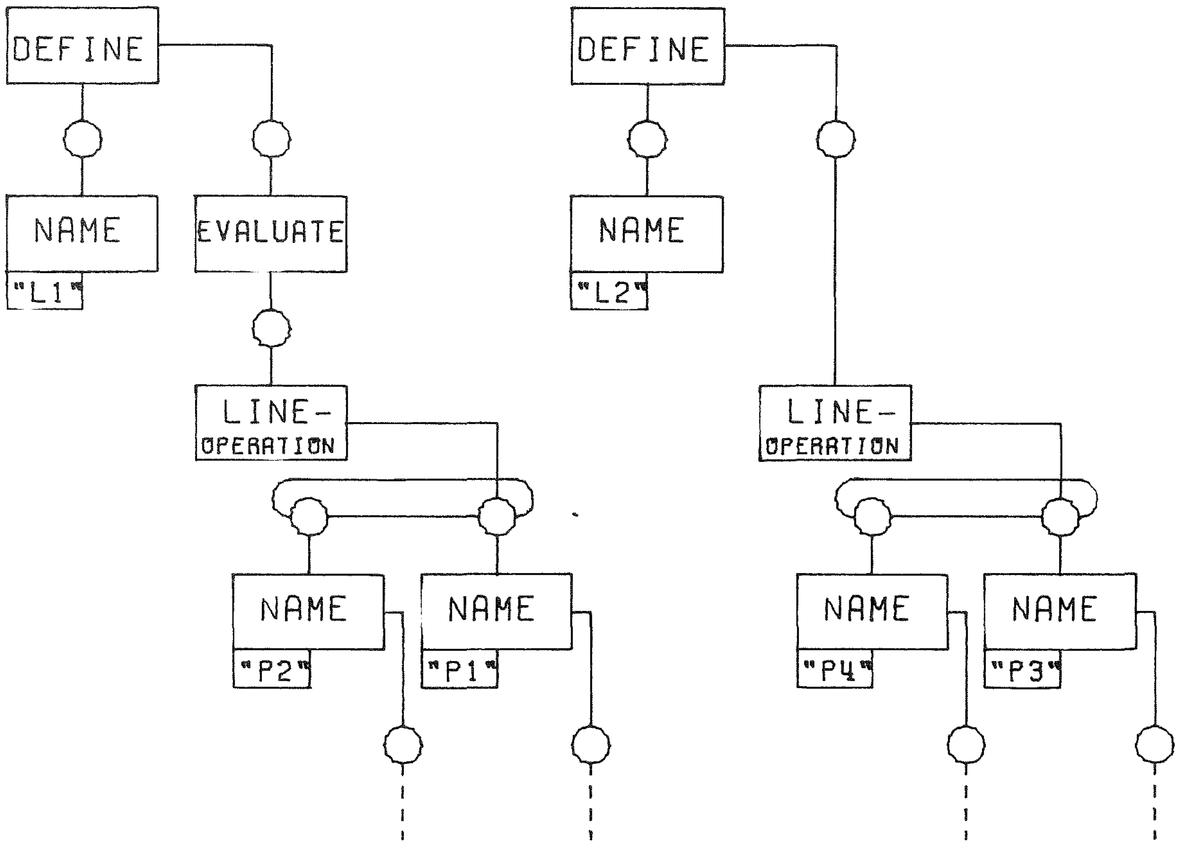
A SET-statement of the language is represented in the structure before parsing by a define-object and an evaluate-object, a DEF-statement is represented by a define-object only. Fig. 30 clarifies the use of the define-object and the evaluate-object.

GRAPHIC-statements:

```
SET 'L1' LINE 'P1' 'P2'
DEF 'L2' LINE 'P3' 'P4'
```

Define-objects have one argument, the substructure to be named, and one operand, the name itself or a name reference. Evaluate-objects possess just one operand, the substructure to be evaluated.

STRUCTURE BEFORE PARSING:



STRUCTURE AFTER PARSING:

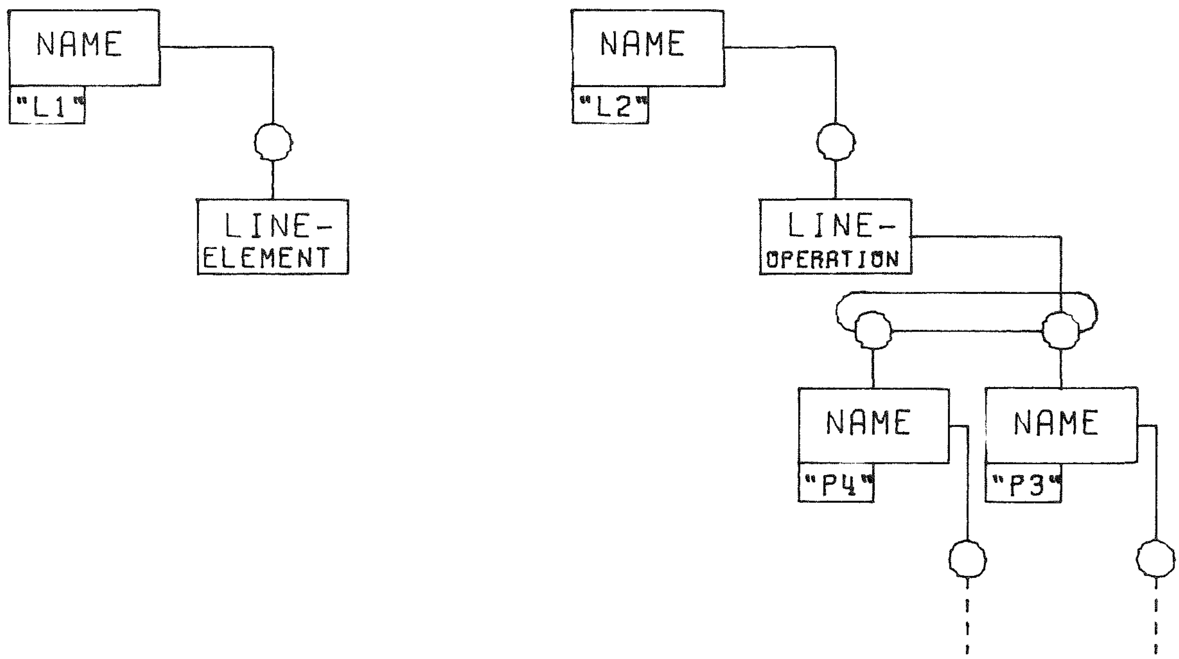


Fig. 30 The evaluate-object and the define-object

5.3.7 Actions

Action objects, when executed, do not deliver a result in form of an object, but perform some action outside of the object-node list. Actions are "open plot" and "plot". "Open plot" changes values in the communication area representing the momentary coordinate system origin used for drawing objects and the momentary size of the drawing. These values are important for successive plot-actions. The "open-plot"-action possesses neither arguments nor operands. Its attribute set contains the size of the new drawing to be opened. The "plot"-action is used to plot graphical objects. It possesses no argument, but a deliberate number of operands that must be graphical elements. When executed, the plot-object writes information on the plot-file representing its operands. The attribute set of the plot-object contains information on the rectangular cut to be scissored out, if scissoring is specified.

5.3.8 Arithmetical and logical operations and elements

Arithmetical and logical expressions are often represented as a simple binary tree /9/. Since GRAPHIC is capable of handling more complicated tree structures it has been a rather simple task to implement arithmetic and logical expressions.

5.3.8.1 Arithmetical and logical operations

GRAPHIC provides objects with an attribute set which characterizes this object as an arithmetic-logical operation corresponding to the following mathematical symbols

+, -, ¬ (prefix), +, -, *, /, **, =, ¬=, >, ¬>, <, ¬< (infix)

The operations are generic in the sense that they are able to operate on both real and integer arithmetic elements. The common arithmetic functions such as SIN, COS, EXP etc. have not been implemented so far.

5.3.8.2 Arithmetical and logical elements

The attribute set of objects representing arithmetical-logical elements is composed of the following information

- object type (as all other objects)
- element type integer, real or logical
- element value internal representation of the value

5.3.9 Control objects

One important feature of GRAPHIC is the capability to identify a number of objects as "belonging together" in a collection. The graphical collection as a set of graphical object has previously been introduced. However, this concept can easily be extended to a set of actions (such as DEFINE, OPEN PLOT, PLOT etc.), since the internal representation of actions is also in form of objects compatible with other graphical objects. There is one difference between a graphical collection of e. g. several points and lines in that a collection of actions must be an ordered sequence to represent a meaningfull program, while for points and lines the order is not important. However, the parsing algorithm of GRAPHIC has been implemented such that the order of collections (or the order of the elements contained in collections and collections of collections) is always maintained. The same principle applies also to the operands of all other object types, not only collections.

5.3.9.1 The DO group and the IF clause

Whenever a DO or an IF in the command indicates the beginning of a DO group or an IF clause which is to be completed by a corresponding END or FI an object is created with an appropriate object type. The attribute set of this object contains information as to whether it is

1) an IF clause
2) a repetitive DO (DO n TIMES)
3) a logical DO (DO WHILE logical expression).

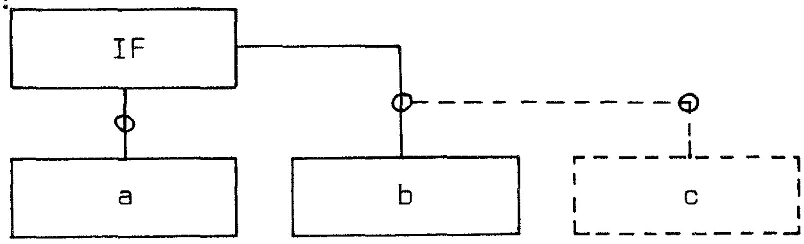
As an operand this object has a logical or arithmetical expression (which will be converted to an element prior the execution of this object); as an argument the object has one action or one collection (e. g. of actions) or in the case of an IF followed by an ELSE two actions or collections.

The function of the routines which actually perform the operations described by the DO-group-and-IF-clause object shall be described briefly with the following examples.

IF-clause

Command: IF a THEN b [ELSE c] FI

Structure:

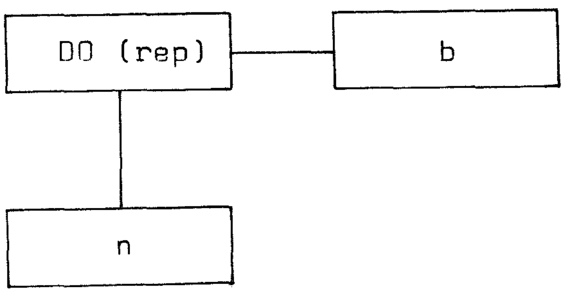


Result: If a = TRUE then the result is b otherwise no result (or c)

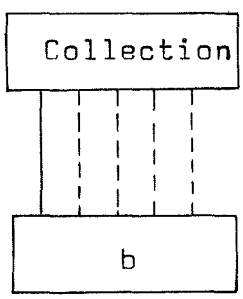
Repetitive DO

Command: DO n TIMES
b
END

Structure:



Result:

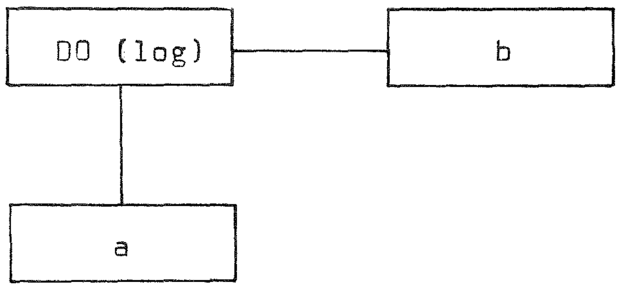


n references from the collection to b

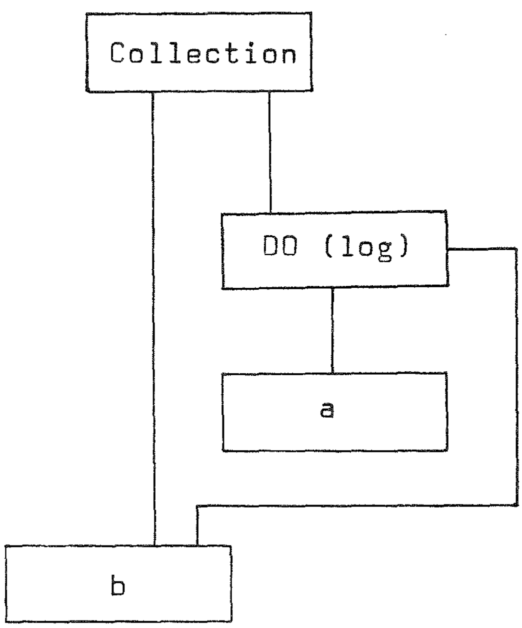
Logical DO

Command: DO WHILE a
b
END

Structure:



Result: If a is FALSE no result,
if a is TRUE a collection containing b and the logical DO group itself.



5.3.9.2 BEGIN-blocks

Whenever a new block is to be opened by the keyword BEGIN, GRAPHIC generates two objects of different types. One of these objects is the block header, the other one is the corresponding environment. The block header has no attributes besides its object type. The environment object has two attributes:

- a hash table
- a pointer index initialized to the value of the environment object node index.

The hash table will be used to associate the local names of the block with the indices of the corresponding name objects. The pointer index serves as the link between the elements of the stack of currently open environments. Since all name references within a block are converted to the appropriate name object references as long as the block is open, the hash table is no longer needed after the corresponding END of the block. The environment object is necessary as a superobject to all name objects belonging to this environment. Only in the outermost block, when the environment is the universe, the undefined object serves as a superobject to the name objects.

A reference is generated such that the environment becomes the (only) argument of the block header. The commands (or else: the objects representing the commands in the internal node structure) contained in the block will be linked to the block header as operands. Upon execution of a block header, these operands, one after the other, will be submitted in the proper sequence to the parser for further execution.

5.3.9.3 Procedures

Procedures are very similar to BEGIN-blocks. The following differences exist:

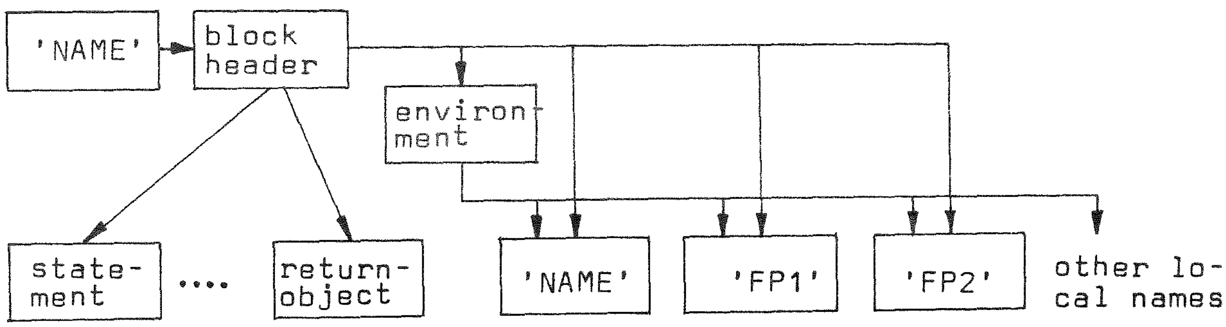
1) A procedure has a name; hence, in the block containing the procedure, there is a name object which has the block header of the procedure as its argument.
2) Within the procedure itself, the name of the procedure is implicitly declared local; hence the environment attached to the procedure block header contains a name object for the name of the procedure.
3) Procedures may have formal parameters. These are local names represented by name objects which are attached as arguments to the block header (and of course as operands to the environment). Hence a procedure block header has n+2 arguments for

a procedure with n formal parameters (1 environment + 1 local name representing the procedure name + n parameter names).

4) If the END which closes the procedure has the RETURN option (RETURN object), then the last operand of the block header will not be an action but rather the object to be returned.

```
Command:  PROC 'NAME' ( 'FP1' , 'FP2' )
          statement
          ........
          END RETURN returnobject
```

Structure:



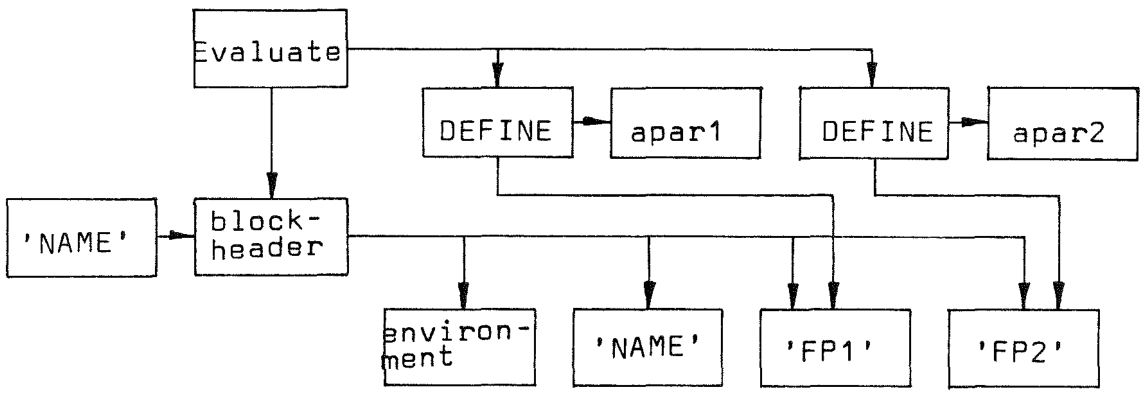
5.3.9.4 Procedure calls

When a procedure is called, an object of object type "evaluate" is generated and the name following the CALL (i. e. the name object of the called procedure) is attached to the evaluate object as an operand.

If the name of the procedure is followed by a list of actual parameters, then an assignment object (same as the one which is generated by DEFINE) is generated for each actual parameter. The actual parameter is attached to this assignment object as an argument, while the corresponding formal parameter (which is a name object in the set of arguments of the called procedure block header) is attached as an operand. The proper sequence is controlled by means of a stack which contains the actual parameter position. (A stack is required because actual parameters may themselves have the form of a function procedure call).

The evaluate object which represents the call in the internal data structure makes sure, that the actual-to-formal parameter assignments are carried out before the operand (i. e. the procedure itself) is evaluated. Since the evaluate object allows only elements (graphical or arithmetic-logical elements) or the undefined object or collections of these or nothing to be considered as a result, all actions contained in the procedure will be executed and a result (if any) will be returned in elementary form.

Command: CALL 'NAME' (apar1 , apar2)
Structure:



5.3.10 The undefined object

When an error is encountered during processing of an object specification, a special object, the "undefined object", is built into the structure instead of the erroneous object. In this way the consistency of the structure is maintained. Thus, the erraneous GRAPHIC-statement: "SET 'A' NOTHING" would cause the building up of a structure containing the name 'A' and below it the undefined object. Beyond this, the undefined object is used to represent objects in the structure that are already referenced, but not yet defined. If the GRAPHIC-statement:
"DEFINE 'L' LINE FROM 'A' TO 'B'"
is specified prior to the specification of 'A' and 'B', the undefined object would stand in the structure as the arguments of the name objects 'A' and 'B'. (Of course, in order to avoid an error, 'A' and 'B' must be specified before referencing 'L'.)
The undefined object is represented only once in the structure.

6. The interpretation of the GRAPHIC language and the building up of the corresponding data structure

6.1 Steps of the conversion of the language into the structure

During the interpretation of the words of the GRAPHIC language, a corresponding data sturcture is built up in internal storage. The integration of every object into the structure takes place in several steps.

In the first step a CDL-routine interprets the language words and stores data from the language in the communication area. In the second step a routine (called from the CDL-routine) creates an object node for the object to be integrated and introduces the node's index into the "temporary node list" (TNL). The TNL is a stack which holds the indices of all nodes not yet completely connected with their superobjects and subobjects in the structure. After creating the node and updating the TNL, the attribute set of the object is defined and filled with data. This is done according to the information passed from the language over the communication area. In the last step the object node is connected with its superobjects by structure connecting routines. The object node index is removed from the TNL, when the object is connected with all subobjects and all superobjects.

6.2 Treatment of names

6.2.1 Declaration of names, environment, name referencing

Whenever a name is encountered in a GRAPHIC command, such as 'A' and 'B' in

```
SET 'A' = LINE FROM POINT  3  2  TO  'B'
```

or in an explicit declaration or as a formal parameter, the appropriate actual environment is checked as to whether the name has been previously declared. For this purpose a hash-table is provided for each environment. The hash routines used are described in /32/. If the name is not found in a declaration statement or as a formal parameter it is considered as previously undeclared. If it is not found in another statement, then the next higher environment is checked similarly. All environments which are presently "open" are linked in a stack to this respect. This shall be explained by the following example

Program	Stack of open Environments	Remarks
GRAPHIC	Uni	One environment called "universe" is opened (Uni) and becomes actual
PROC 'A'	Uni, A	Environment of 'A' is opened and becomes actual
PROC 'B'	Uni, A, B	Environment of 'B' is opened and becomes actual
BEGIN	Uni, A, B,	Environment of this block is opened and becomes actual
END	Uni, A, B	Environment closed. Environment of 'B' becomes actual
END	Uni, A	Environment of 'B' closed. Environment of 'A' becomes actual
END	Uni	Environment of 'A' closed. Universe becomes actual
END GRAPHIC		Universe closed

The search for the name is terminated when either the name is found ("previously declared") or when it cannot be found in all the open environments including the universe.

If the name is found to be previously declared in an environment, the corresponding name object index is retrieved from the hash table and - if the command was not a declaration - this object is inserted into the TNL.

If the name is found to be previously undeclared, then it will be declared in the actual environment. This is performed by

- generating an appropriate name object
- inserting the name and the object index in the environment hash table
- attaching the name object as a subobject to the actual environment object.

Now the name is declared and can be treated as previously declared.

6.2.2 Assignment of objects to names

Objects may be assigned to names by means of a SET or DEFINE command or by the INITIAL option of a declaration. Names to which no object has been assigned, are considered to have the undefined object assigned to them by default.

Every assignment is performed by an assignment object which has - when executed - as its operand the name object to which something is to be assigned, and the object as its argument. (Note that both name and object may be results of the execution of other objects.) If the name has already an object as an argument, this object and all other object which are subobjects to this one alone are destroyed. Then the (new) object is attached as argument to the name.

Examples:

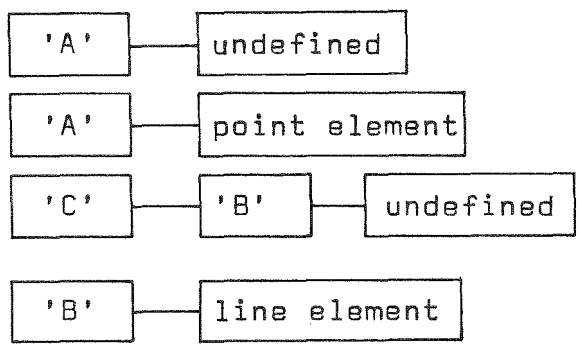
Result

```
DECLARE 'A'

DEF 'A' POINT 2 3

DEF 'C' 'B'

SET OBJ 'C' LINE FROM
    'A' TO POINT 3 2
```



6.3 An example for the conversion of a language statement into the corresponding structure

Let us consider the statement:

SET 'L' LINE 'P1' POINT 1 2

The object named 'P1' is supposed to exist already. The structure nodes corresponding to the different words of the statement are shown in fig. 31. Building up the structure is done in the same order as the words of the language are met, i. e. from the top of the structure downwards.

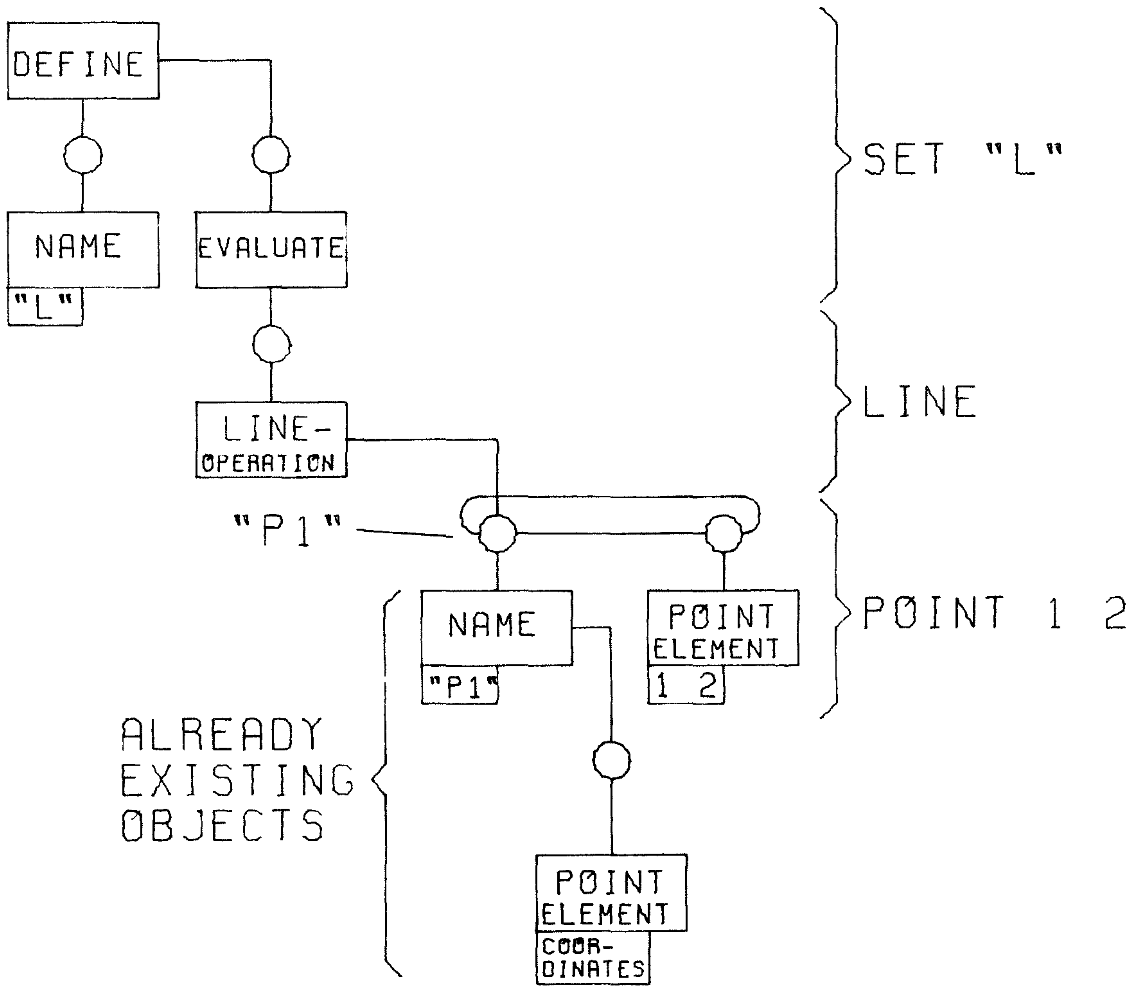


Fig. 31 Correspondence of language words and structure

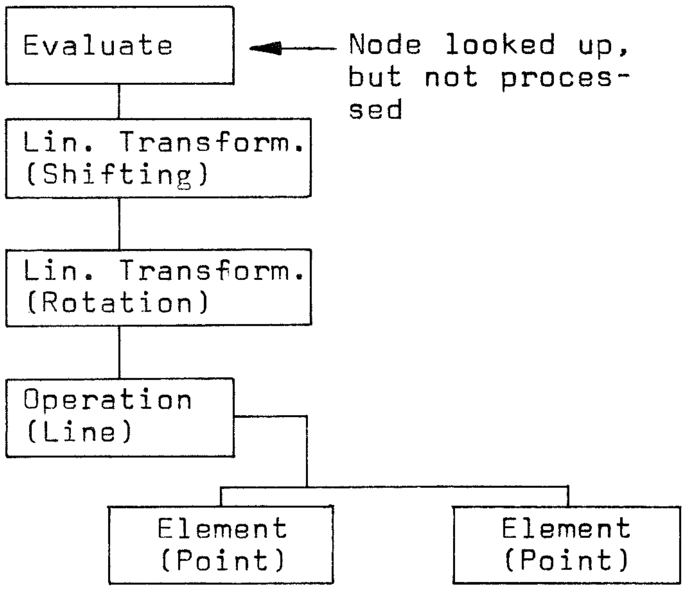
7. Parsing the structure

The graphic data structures are parsed by a program, called the parser. Parsing starts at the top of a structure. When processing an object node, the parser looks up the object type of the node being processed and the types of the arguments and operands. Depending on these object types one of two possible actions is executed:

1. The object node is executed with its arguments and every one of its operands, the result of the operation logically replaces the operation in the structure and the parser goes one level up.
2. The object cannot be executed with its arguments and operands. The parser goes down to every subobject, first to all arguments, then to all operands and transform them, until their objecttype is suitable for execution of the object.

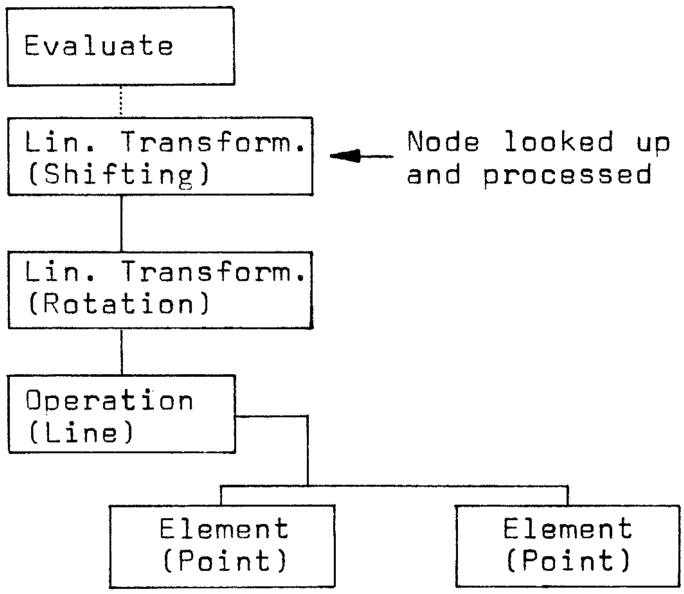
The parser looks up in a decision table which one of these actions is to be taken. If the object in process can be executed, a routine is called according to the object type. This routine is called once for every operand of the object. If there is no operand, the parser substitutes the undefined object as operand. The parameters passed to the routine by the parser are one operand object and the collection of arguments. The invoked routine may return a resulting object or a collection of objects to the parser. The routine, which is called up may detect, that one of the arguments or the operand is unsuitable for correct processing. In this case the routine will generate an error message and return either nothing or the undefined object as a result. The results delivered by all calls are integrated into a collection of results. This collection replaces the processed object in the structure. The result of the execution of an object node may be not only an element but also an operation. E. g. the result of the execution of a line operation with two point elements as arguments is a line element logically replacing the line operation and its arguments in the structure. The execution of a linear transformation operation with the operand being another linear transformation operation delivers as a result a new linear transformation operation which logically replaces the two operations in the structure. The resulting operation then has to be executed. The following example illustrates the parsing of a simple structure.

a)



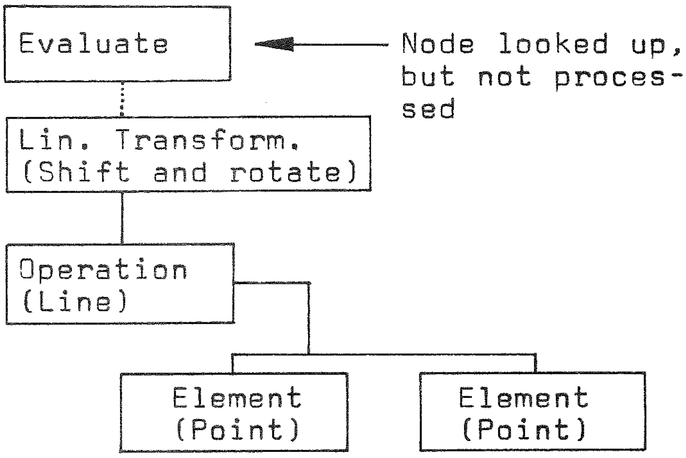
Object type: Evaluate
Operand type: Linear Transformation
Arguments: None
Action: Go down

b)



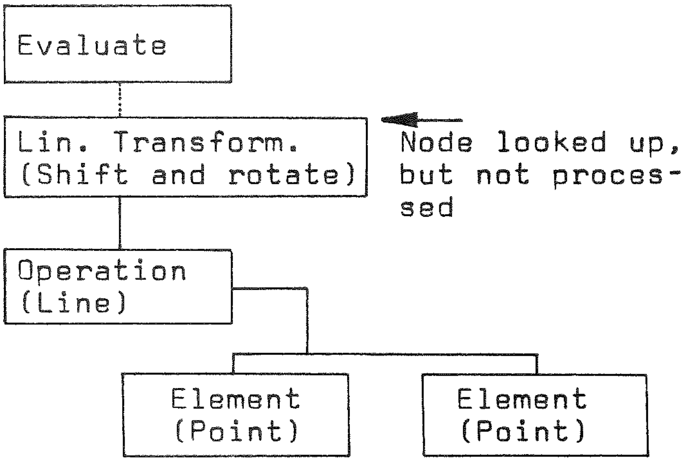
Object type: Linear Transformation
Operand type: Linear Transformation
Arguments: None
Action: Execute
Result: Linear Transformation

c)



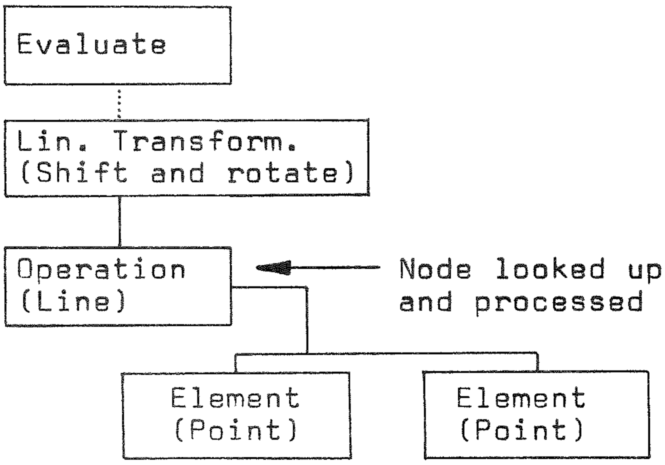
Object type: Evaluate
Operand type: Linear Transformation
Arguments: None
Action: Go down

d)



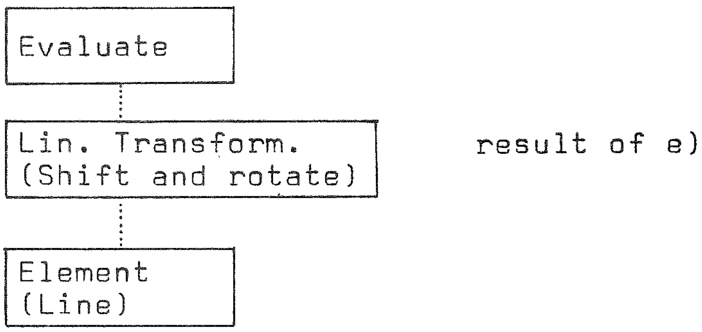
Object type: Linear Transformation
Operand type: Operation
Arguments: None
Action: Go down

e)



Object type: Operation
Operands: None
Argument types: Element
Action: Execute
Result: Element (line)

f)



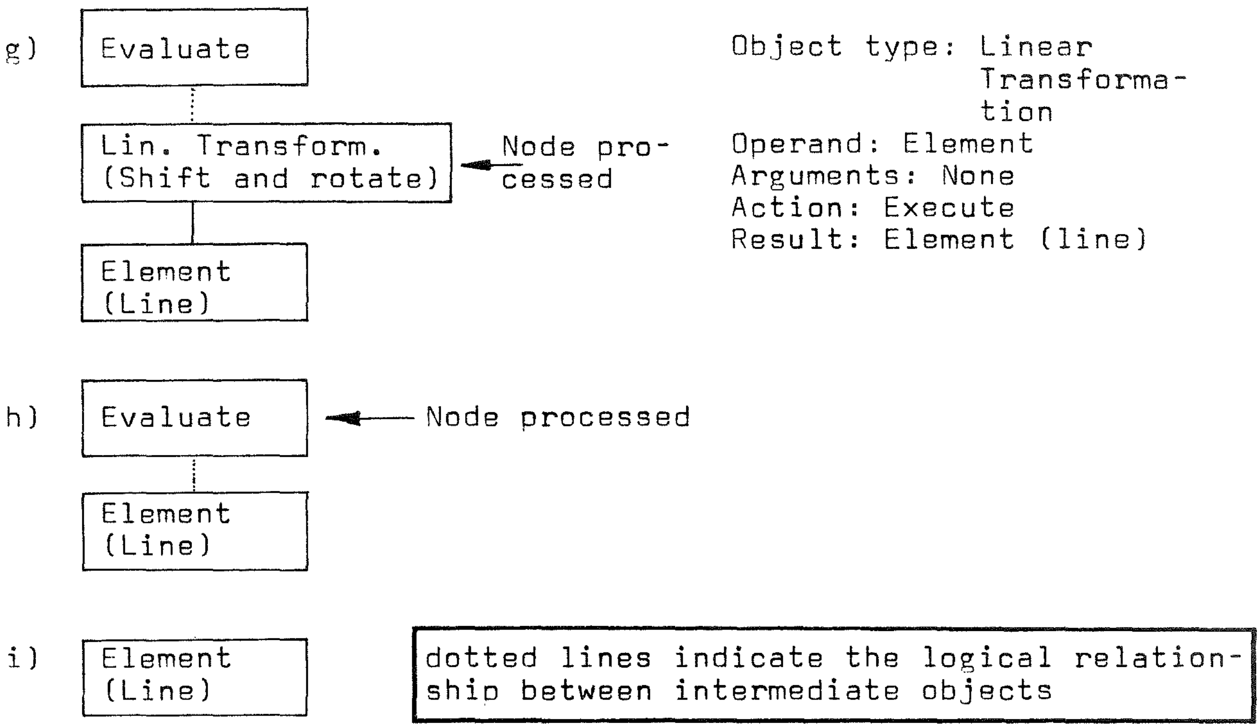


Fig. 32 Steps for parsing a structure

Since some operations, the linear transformations, can not only be executed with elements as operands, but also with linear transformations as operands, they have a different object type than the other operations. In most cases the structure must not be destroyed while parsing it. So the results of operations do not actually replace the operations in the structure. The indices of the intermediate results are kept by the parsing program in stacks and the replacement of a node by its result takes place only logically.

8. Extension of the GRAPHIC system

New species of objects can easily be integrated into the GRAPHIC system, especially new kinds of elements and operations, The following tasks have to be done in order to add new kinds of objects:

- First, define the syntax of the object specification in the GRAPHIC language.
- Then, write a CDL routine according to this syntax. The CDL routine must place necessary data from the language in the communication area. At least, the routine has to call a routine that builds up an object node for the new kind of object.
- This routine, called from the CDL, must be programmed. It has three tasks:
  Create a node for the object, define its attribute array, fill it with data and integrate the object into the structure. The first and the last task are accomplished by calling existing system routines. Only the definition of the attribute set of the new object and the taking over of the attribute data from the communication area have to be programmed completely new.
- For new operation objects the executing routine corresponding to the operation node must be set up. This routine is called when the operation node is parsed. It has a normed argument-list, containing the indices of the operation node, the operand node, the argument collection and of the resulting object to be created by the routine.
- For new element objects routines have to be prepared for the linear transformation of the element and for plotting the element, i. e. placing a set of data elements describing the element on the plotfile.
- For new object types the decision tables of the parser have to be extended.

9. Error handling in GRAPHIC

9.1 General features

So far, the correct use of GRAPHIC has been described, but GRAPHIC has been designed to run with incorrect input data too, producing the most meaningful output possible (no dumps). This is desirable especially in batch jobs in order to avoid incremental and therefore time consuming debugging. For this purpose, the GRAPHIC system contains a great number of tests to detect errors. Some of these tests are carried out on the level of the command interpreter, most of them on the level of GRAPHIC system routines (those coded in ICETRAN) and some on the level of the ICES executive system or the general machine operating system.

9.1.1 Error handling by the command interpreter

Errors detected by the command interpreter are mostly syntax errors. Whenever a syntax error is detected, a legible message is produced and the word, which caused the error, is skipped. A similar action is taken when the end of a command is found too early. This means that the command is syntactically incomplete. As an example take

SET 'A' LINE FROM POINT 3 2

(where e. g. TO POINT 7 10 is missing).

As a consequence of this type of error, the undefined object is built into the internal data structure representation of the command at places, where no correct object could be found. Thus the data structure, which is submitted to other programs, is consistent and uncontrolled breakdown is avoided.

The GRAPHIC language has been designed to work even if some statements cannot be interpreted. In many error situations, reference is made to some object which should have been defined before, but which is undefined due to errors. In these cases GRAPHIC nevertheless performs all the work requested with the correctly defined objects, producing a plot which shows at least part of the whole picture and which allows the user to check the semantics of his GRAPHIC program for this part. The concept of the "undefined object" has been very helpful to this respect.

9.1.2 Error handling by GRAPHIC system routines

Errors found by ICETRAN-GRAPHIC system routines are treated by calling a special part of the GRAPHIC-program system, the program-error-handling system or shortly "ERROR system". This ERROR system is an almost independent subsystem itself and it is connected to GRAPHIC by some clearly defined linkages only. The ERROR system may be used in other subsystems too, although it has been implemented only in GRAPHIC at this time. In case of errorsGRAPHIC calls this system and passes the following informations:

a code number (nr) which identifies the error message and
a severity-code (s) describing the importance of the error found.

The severity-code varies between 0 and 16. Messages with codes 0 to 4 are considered as warnings only, 5 to 8 are errors which can be handled by the GRAPHIC system routines to continue the execution and (very scarcely) 9 to 16 mark errors which should result in an interruption of the interpretation of the command actually processed. Moreover, the message may contain data which identify the origin of the error or the state of the program or data structures, the knowledge of which may help in understanding the reasons for the error.

All these informations are processed by the ERROR system in a way which may be controlled by other GRAPHIC system routines or by user's input statements embedded in other GRAPHIC statements (see 9.2 and 9.3). The standard actions of the ERROR system are as follows: According to the code number of the error (nr) a text, including format descriptors is read from the subsystem data set. The values submitted are formatted as described by this format. The message is completed with the error sequence number, the name of the program issuing the message and the actual (cpu) time since the start of the GRAPHIC execution. The resulting character string is printed on a print file, which is the standard print file by default. Thus the GRAPHIC programmer is supplied with a legible error message.

Besides, the ERROR system counts the number of errors of different "characteristics" (see below), so that system programs may ask whether and how many errors have occurred and the user may be informed of the total number of errors of different characteristics found while executing his job. Moreover, the ERROR system compares the actual accumulated number of errors of different kinds with predefined allowable numbers of these errors. If any limit of this type is exceeded, the ERROR system calls an exit program. By default this prints statistics, some message buffers and inhibits execution of the subsequent GRAPHIC commands, which are nevertheless checked for syntactical errors.

The "characteristic" of an error is identified by a figure (m) and is defined by default as follows:

m = 1 every error has this characteristic, so the number of errors with this characteristic is the total number of errors reported.

m = 2 an error message, which has been submitted to the ERROR system after a predefined time measured from the start of the GRAPHIC execution, has this characteristic. The ERROR system contains a routine to evaluate the maximum allowable step time for this job, so time overflow may be detected.

m = 3 to 19 the error message has the severity-code 0 to 16 accordingly.

Additionally the user may redefine the characeristic figure and define some "special" characteristics by adding ICETRAN subroutines, which are called by the ERROR system to decide whether the error message shall be qualified with this special characteristic or not. This subroutine can ask for all informations just stored in GRAPHIC to make its decision.

Besides the standard action described, the ERROR system may be controlled in a flexible way by the user. Some of its possibilities are:

- Suppress completely execution of the ERROR system, favouring effectivity against security.
- Suppress printing of messages at all.
- Suppress actual printing of messages by storing the message in a buffer of adjustable size. From there, messages may be printed later, if required, to show the history of any disastrous error situation.
- Stop execution.
- Save the actual contents of the GRAPHIC data pool on any file and then stop execution. A later job may be started to read this data and continue the task including only some correction statements. (Restart feature).
- Plot the graphical data created until the error was found and then stop execution.
- Print informations.

Most of these possibilities may be achieved by just setting some control values as described in 9.2 and 9.3. For the other ones the user has to supply an exit program by his own.

The format-texts which are read by the ERROR system according to the code number nr of the error message and may be added to the subsystem data set or changed, deleted or listed by using the subsystem TABLE-II /33/. Thus, the message text may be written in English or German or other languages, may be shortened or extended for more detailed or clearer informations without any modifications to the programs.

9.1.3 Error handling by ICES execution or operating system

Although we tried hard to program GRAPHIC so that no error should be handled at the ICES executive or operating system level, this may nevertheless happen. Reason for this may be errors which may still exist in the GRAPHIC system routines or error situations which actually occur at the operating system level as e. g. a data set, time or core storage-overflow. An error found by the ICES executive results in a message and a core dump of controllable contents and subsequently the job is terminated with user completion code 256. Errors found by the operating system usually result in a job termination with system completion code. By submitting special job control cards a core dump may be obtained in this case too.

9.2 GRAPHIC statements to control the ERROR system

The following commands of the GRAPHIC language control the ERROR system

```
GRAPHIC
ERROR
GOON
END GRAPHIC
```

GRAPHIC and END GRAPHIC have other functions too, as already described.

9.2.1 GRAPHIC and END GRAPHIC

The GRAPHIC statement causes the initialization of the ERROR system. At this time, all default values for the control parameters are assigned. END GRAPHIC causes (in connection with the ERROR system) the printing of error statistics and of the contents of the message buffer.

9.2.2 ERROR, the main control statement of the ERROR system

9.2.2.1 Control variables

The ERROR system is controlled by the following control variables which may be altered by the statements described below. The standard values are listed too. The variable m refers to the characteristic figure and varies from 1 to 19. For every characteristic the following control parameters are stored:

ik (m) the allowable number of messages. If the number of messages is greater than ik (m) an exit program is called.
standard
ik(1) (total number of messages) 1 000 000
ik(2) (time overflow) 0

ik(3) (code 0)	1 000 000
ik(4) (code 1 to 8)	1 000
ik(12 to 19) (code 9 to 19)	0

nams (m) the name of the exit program, standard nams (m) = 'EREXIT'

ids (m) the messages, with sequence numbers between ids (m) and

ide (m) ide (m) (including both) are printed immediately, other messages may be stored in the buffer or suppressed. Standard ids (m) = 0, ide (m) = 100

icont(m) if icont (m) = 1 messages with this characteristic m are not processed. Standard: icont (m) = 0

Other control values are:

lcont if lcont = 1 the ERROR system does nothing; so nearly no time is exhausted, but on the other hand the user gets no messages. Standard lcont = 0

ne print-file of the ERROR system. Standard ne = 6

ibuf if ibuf = 1 no messages are stored in the buffer. Standard ibuf = 0

lbuf number of print-lines which may be stored in the buffer. Standard lbuf = 20

mst number of standard characteristics (mst = 19)

mson number of special characteristics (mson = 0)

cpures after cpures sec. before the maximum allowable CPU-time for the actual job step all messages have the characteristic m = 2 (time overflow).
Standard: cpures = 15

cpumax after cpumax sec., measured in CPU-time since the start of GRAPHIC execution, all messages have the characteristic m = 2(time overflow). Standard cpumax = maximum allowable step time after the GRAPHIC command minus cpures

namson (ms) ms = 1,2, , mson, the names of the user delivered programs which decide whether a message shall be qualified by a special characteristic.
Standard mson = 0 and therefore no namson are defined.

9.2.2.2 Syntax of the ERROR statement

```
ERROR [:]

 ┌ ON                                                                   ┐
 │ OFF                                                                  │
 │ PRINT  [ON]  [FILE ne]  information                                  │
 │ SET  [FOR] ┌ CHARACTERISTICS m1      [TO m2]  values ┐             │
 ┤            ┤ CODE             c1      [TO c2]  values ├             ├
 │            │ SPECIALCHARACTERISTIC namson values      │             │
 │            └ PARAMETER  parameter                     ┘             │
 └ FORMAT TEST  [FROM nr1] [[TO]  nr2 ]                                 ┘

                 ┌ STATISTICS      ┐
information ::= ┤ BUFFER          ├    [ [AND]  information]
                 └ PARAMETERVALUES ┘

              ┌ [CONTROL] {ON                                      ┐
              │            OFF}                                    │
values ::=   ┤  STOP  [ AFTER ]  ik  [MESSAGES]                    ├ [ [AND]   ]
              │ [PRINT]  [FROM ids]  TO ide [MESSAGES]             │ [  values ]
              └  EXIT  [PROGRAM]  nams                              ┘

parameter ::=
 ┌  PRINT  [ON]  [FILE]  ne                                ┐
 │ [NUMBER]  [OF]  STANDARDCHARACTERISTICS  mst            │
 │ [MAXIMUM]  [CPU]  TIME   cpumax  [SEC]                  │   [ [AND]      ]
 ┤ [CPU]  RESERVETIME cpures  [SEC]                        ├   [  parameter ]
 │ [BUFFERSTORAGE] {ON                                     │
 │                  OFF}                                   │
 └ [BUFFER]  LENGTH  lbuf                                  ┘
```

m1, m2, c1, c2, ik, ids, ide, ne, mst, lbuf are integer values
cpumax and cpures are integer or real values
namson, nams are alpha strings with maximum length 6.

9.2.2.3 Semantic of the ERROR statement

ERROR ON sets lcont = 0 (see 9.2.2.1)

ERROR OFF sets lcont = 1

ERROR PRINT ON FILE ne information

On file ne (standard ne = 6) informations are printed

information = STA, a messages statistic is printed

= BUF, the contents of the message buffer is printed

= PAR, the values of the control parameters listed in 9.2.2.1 are printed

ERROR SET FOR CHARACTERISTICS m1 TO m2 values

For the characteristics m1 to m2 the "values" are assumed.

ERROR SET FOR CODE c1 TO c2 values

For the characteristics c1+3 to c2+3 the "values" are assumed.

ERROR SET FOR SPECIAL CH = namson values

The program namson decides over a special characteristic and for this characteristic the "values" are assumed.

ERROR SET PARAMETER parameter

For some of the characteristic - independent parameters listed in 9.2.2.1 the values defined in "parameter" are assumed.

```
values ::= CONTROL  ON          sets icont (m) = 0
           CONTROL  OFF         sets icont (m) = 1
           STOP AFTER ik        sets ik (m) = ik
           PRINT [FROM ids] TO ide  sets ids (m) = ids
           and ide (m) = ide; standard values for ids is 0
           EXIT nams            sets nams (m) = nams

parameters ::= BUFFERSTORAGE  ON   sets ibuf = 0
               BUFFERSTORAGE  OFF  sets ibuf = 1
               The other parameter - commands set the values
               as defined by name in the syntax and in 9.2.2.1
```

ERROR FORMAT TEST FROM nr1 TO nr2

The format texts stored for the error code numbers nr1 to nr2 on the subsystem data set are used to print test-error-messages (with some assumed values if necessary). Thus, the system programmer may test, whether the format texts are correct and clear or not. If changes are necessary, they can be done by using the TABLE-2 Subsystem.

9.2.3 The GOON statement

Syntax: GOON

Semantic: This command causes an ENABLE request of the CDL to be executed. Any preceding INHIBIT request is canceled. If an INHIBIT has been executed, no GRAPHIC system programs have been called by the command interpreter since that time.

This statement should be used just before some "conserving" statements, as e. g. ENDGRAPHIC, so that, if an error caused an INHIBIT, this conserving statement may be executed nevertheless.

9.3 ICETRAN statements to control the ERROR system

There are many programs of the ERROR system which may be called by LINK statements in GRAPHIC system programs to control the ERROR system. The full details of these calls will be described elsewhere in a following publication. Here the possibilities are listed only.

The ERROR system provides programs which may be called to

- create messages
- ask for the error codes of messages produced in subprograms of the just executed system program
- ask for statistical informations (number of messages with some characteristic since initialization or since the last question)
- get the contents of the message buffer
- sanction a previos error-message; this means that for some message characteristics the number of allowable messages is incremented by one
- test for time overflow
- change all values which control the ERROR system and which are described in 9.2.2.1
- cause printings of statistics and the message buffer.

10. Some GRAPHIC programming examples

The following figures demonstrate some capabilities of the GRAPHIC system.

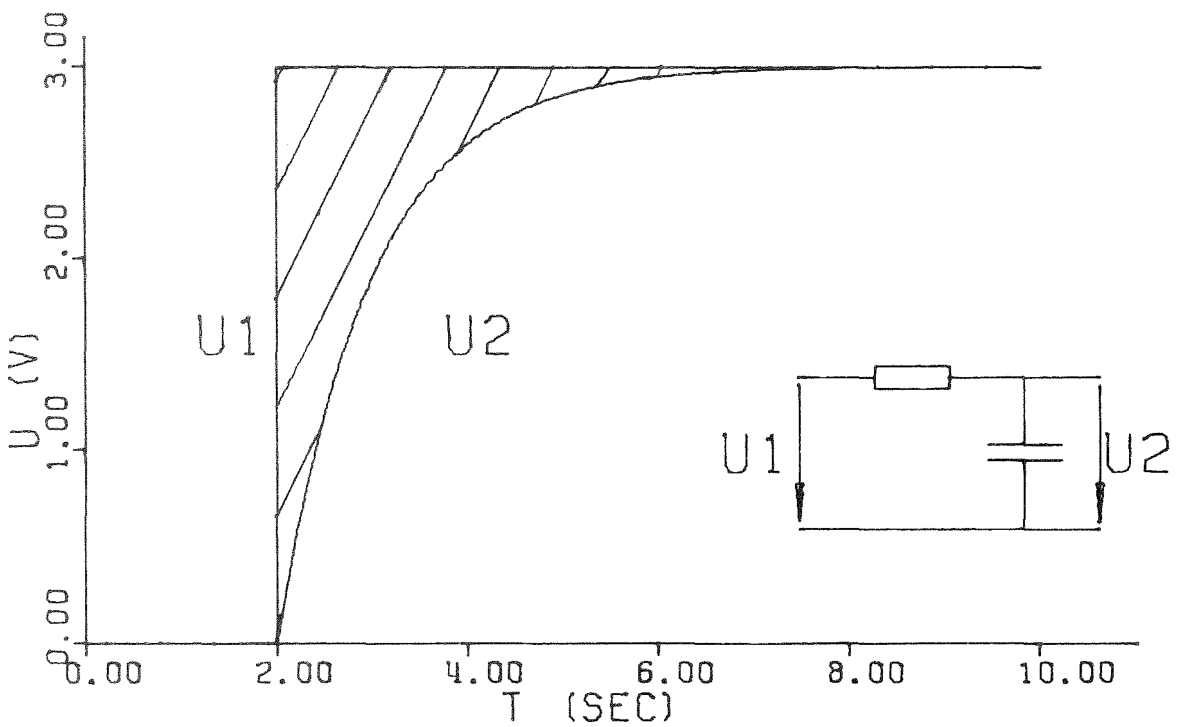


Fig. 33 GRAPHIC programming example

```
EGRAPHIC
Ø A GRAPHIC EXAMPLE
OPEN PLOT ON DIMENSIONS 20 CM 14 CM
SET 'INP' POLYGON  0  0  2  0  2  3  10  3  END
SET 'OUTP' POLYGON WITH 81 VALUES ACCORDING TO
      TAU = 1.
      DO 1 I = 1,81
      X(I) = (I+19.)/10.
    1 Y(I) = 3.*(1.-EXP(-(X(I)-2.)/TAU))
      END
SET 'U1' TEXT 'U1' 1.2 1.5
SET 'U2' TEXT 'U2' 2.8 1.5
SET 'XAX' X-AXIS TITLE 'T(SEC)' TO ( 'INP','OUTP' )
SET 'YAX' Y-AXIS TITLE 'U(V)'   TO ( 'INP','OUTP' )
SET 'TRANS' TRANSFORMATION TO 'XAX' 'YAX' OF
    ( 'INP','OUTP','U1','U2',SHADE DIST 0.4 BETWEEN 'INP' AND 'OUTP' )
STANDARD UNIT CM
SET 'ARROW' POLYGON 12.5 6.4 12.5 4.6 12.45 5.1 12.55 5.1 12.5 4.6 END
SET 'CIRCUIT' ( 'ARROW',TEX 'U1' 11.5 5.25, POL 12.5 6.5 13.5 6.5
    13.5 6.35 14.5 6.35 14.5 6.65 13.5 6.65 13.5 6.5 END, POL 15.5
    5.4 15 5.4 16 5.4 15.5 5.4 15.5 4.5 END, SHIFT 4 0 'ARROW',
    TEX 'U2' 16.6 5.35, POL 16.5 4.5 12.5 4.5 END )
PLOT ( 'XAX','YAX','TRANS','CIRCUIT' )
SAVE 'FIG.33' 'KEY1'
END GRAPHIC
```

```
STANDARD UNIT CM
OPEN PLOT DIN A 4
% FIGURE 34
SET 'P1' POINT 2 22
SET 'P2' POINT 2.5 15
SET 'P3' POINT 2 8
SET 'P4' POINT 4.2 17.7
SET 'P5' POINT 4.2 12.3
DO 20 TIMES
PLOT SPLINE CLOSED ( 'P1' 'P4' 'P5' 'P3' 'P2' )
SET 'P1' SHIFT 0.4 -0.25 'P1'
SET 'P2' SHIFT 0.375 0 'P2'
SET 'P3' SHIFT 0.4 0.25 'P3'
SET 'P4' SHIFT 0.69 -0.135 'P4'
SET 'P5' SHIFT 0.69 0.135 'P5'
END
```

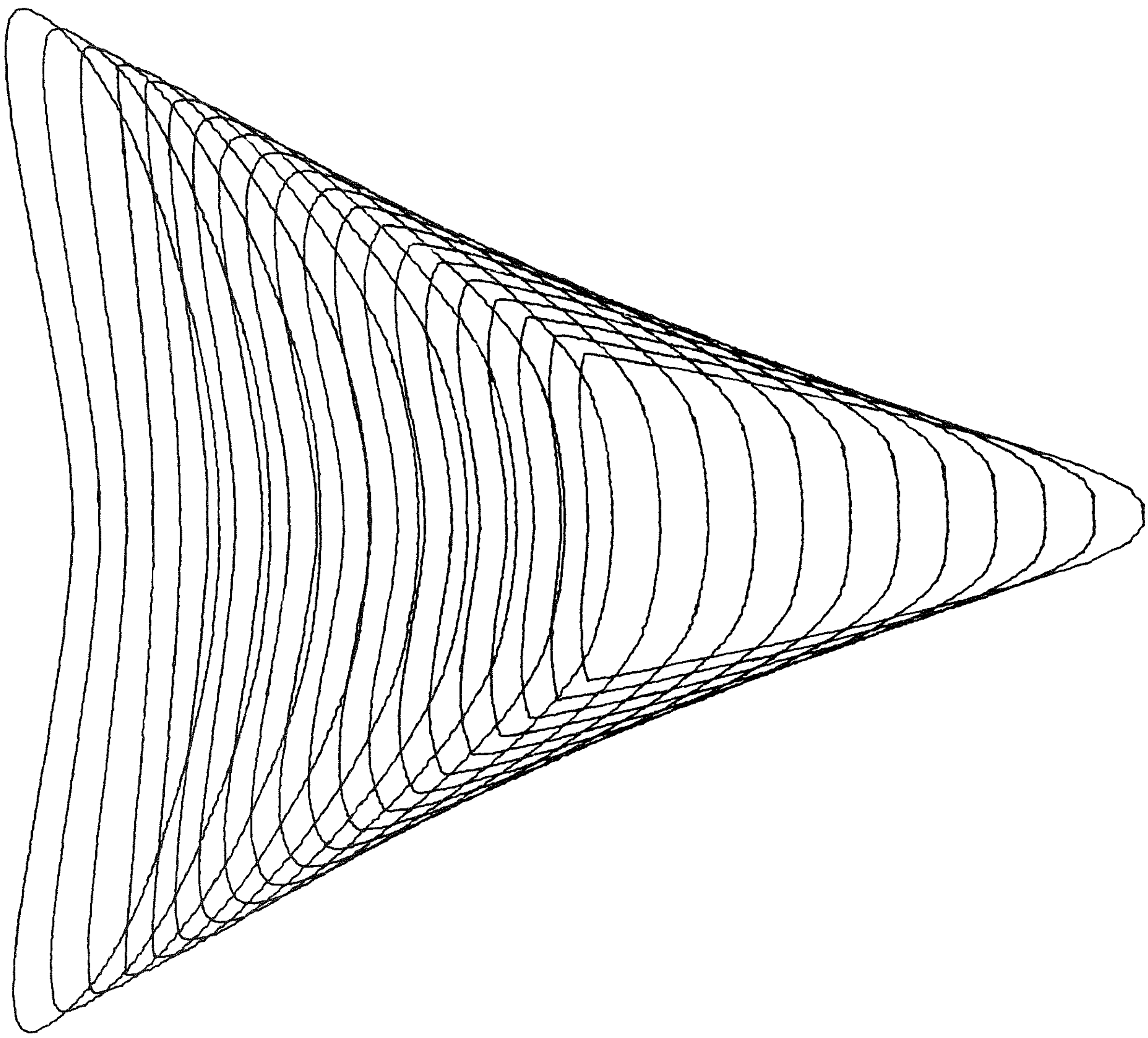
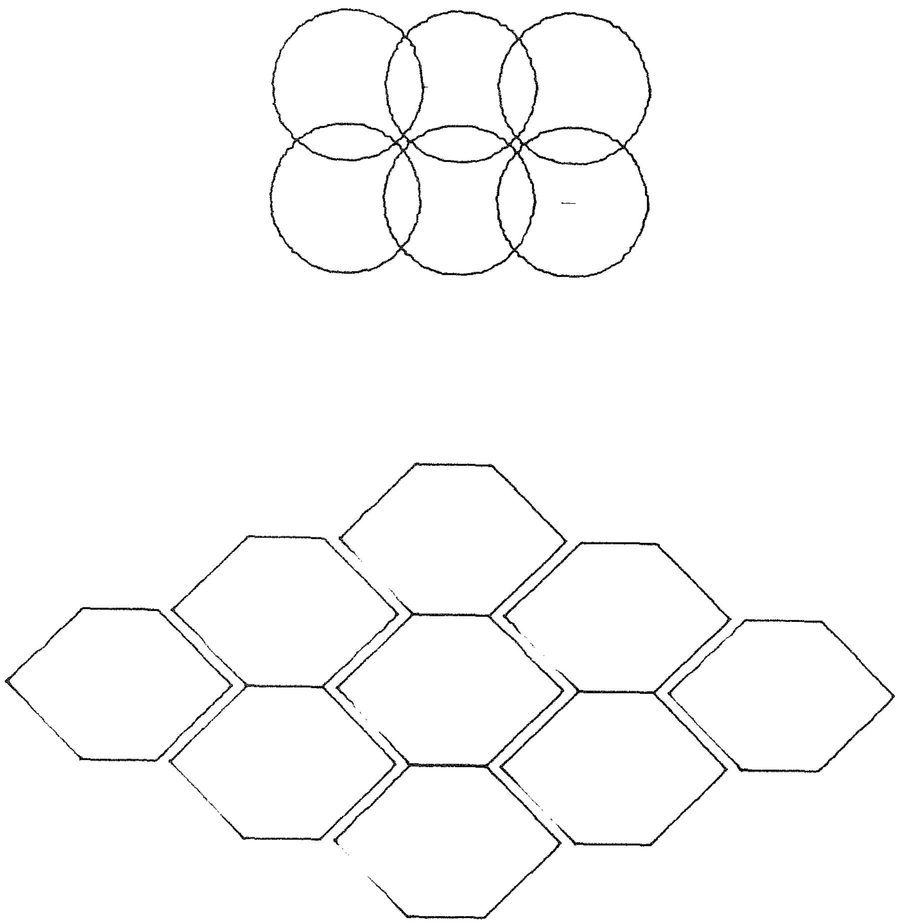


Fig. 34 Example 2



```
STANDARD UNIT CM

OPEN PLOT DIN A 4

PROCEDURE 'PATTERN' ( 'OB','N','M','P1','P2' )

SET 'OB1' = 'OB'

DO 'N' TIMES

SET 'OB2' = 'OB1'

DO 'M' TIMES

PLOT 'OB2'

SET 'OB2' SHIFT TOWARD 'P1' OF 'OB2'

END

SET 'OB1' SHIFT TOWARD 'P2' OF 'OB1'

END

END

CALL 'PATTERN' ( CIRCLE CENTER 9 25 RADIUS 1, 2, 3, POINT -
       1.5 0, POINT 0 -1.5 )

CALL 'PATTERN' ( POLYGON CLOSED 9 19 10 20 11 20 12 19 11 18 -
       10 18 END, 3, 3, POINT 2.2 -1., POINT -2.2 -1 )
```

Fig. 35   Example for using a procedure

```
SET 'CURVE' POLYGON WITH 100
      READ (12,10) (X(I),Y(I),I=1,100)
   10 FORMAT (10E14.7)
      END

OPEN PLOT DIN A 6
SET 'XAX' X-AXIS 'CURVE'
SET 'YAX' Y-AXIS 'CURVE'
SET 'C1' TRANSFORMATION TO 'XAX' 'YAX' OF 'CURVE'
PLOT ( 'C1' 'XAX' 'YAX' )
OPEN PLOT DIN A 6
PLOT ( APPROXIMATION OF DEGREE  2 'C1' 'XAX' 'YAX' )
END GRAPHIC
```

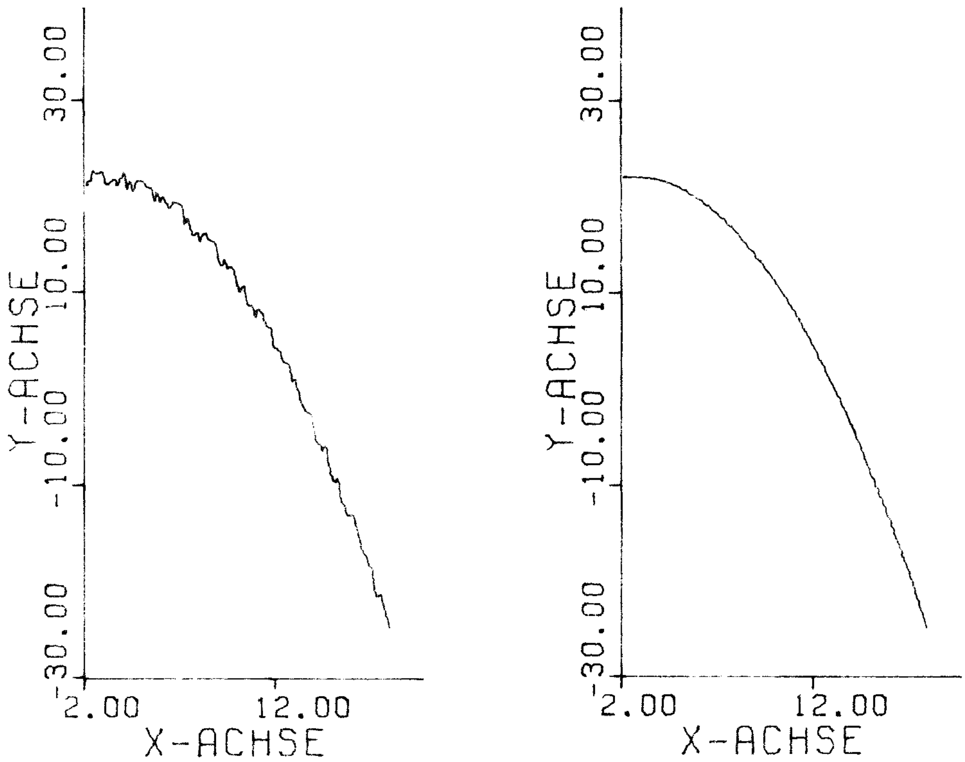
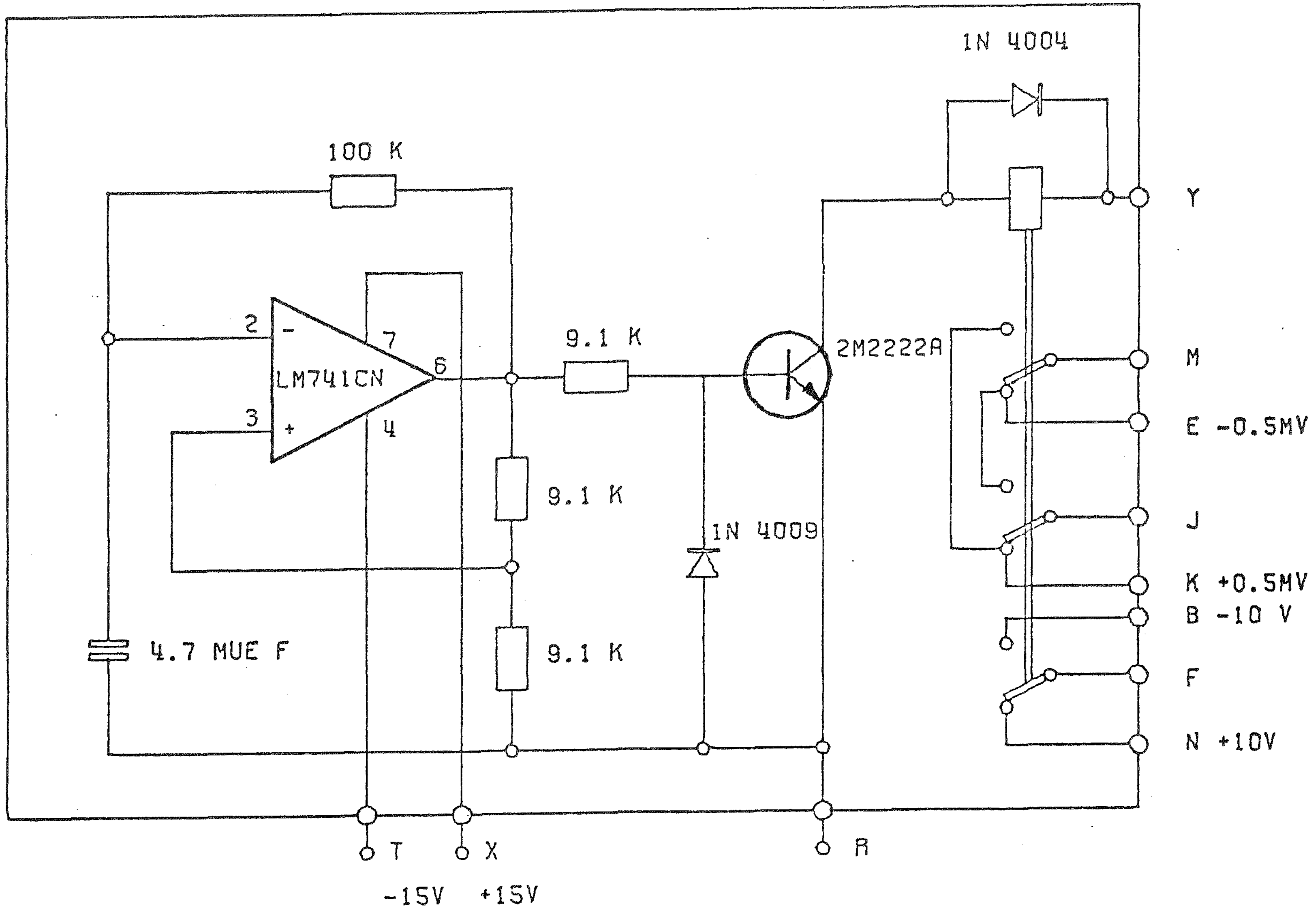


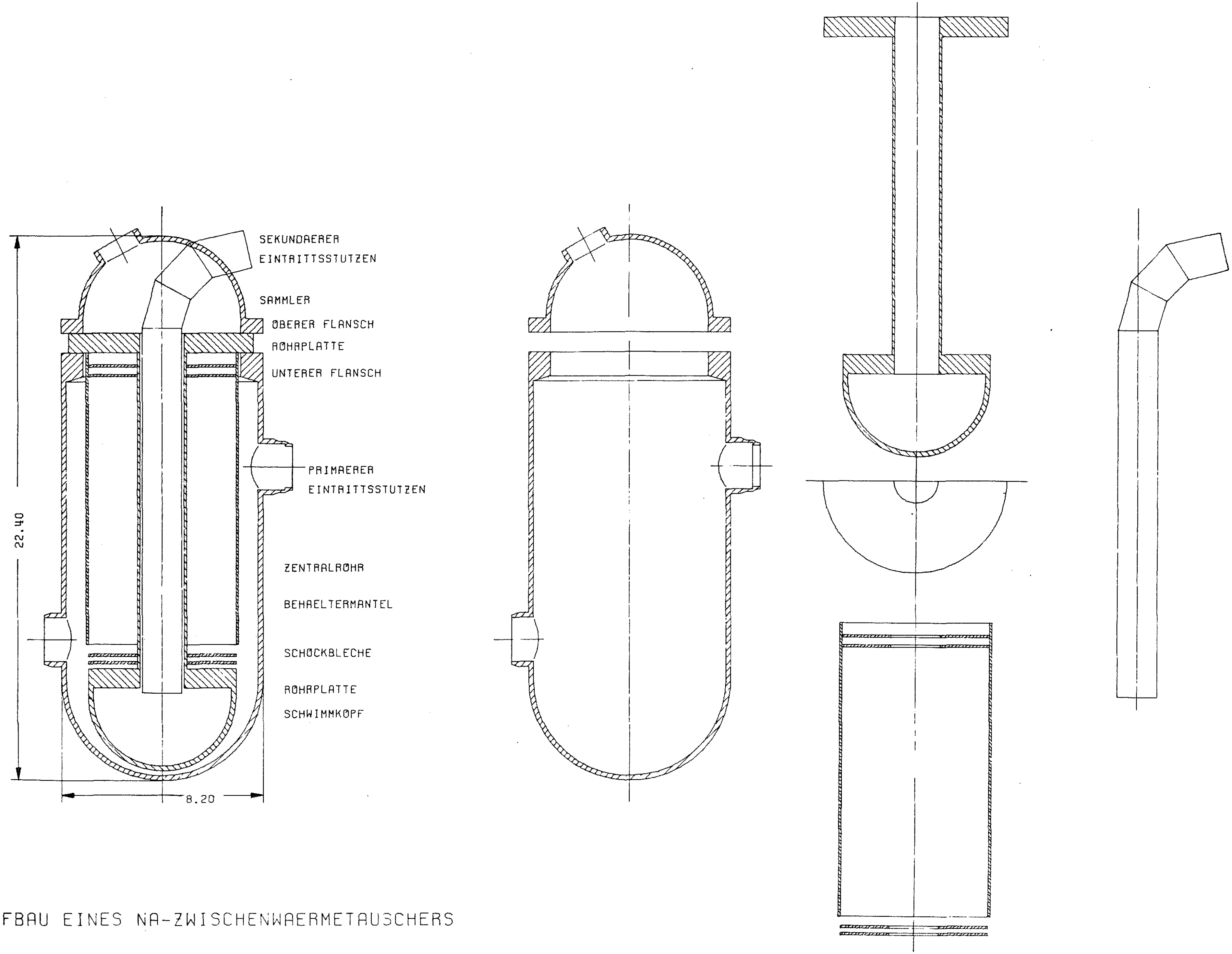
Fig. 36 Example for coordinate axes and an approximation



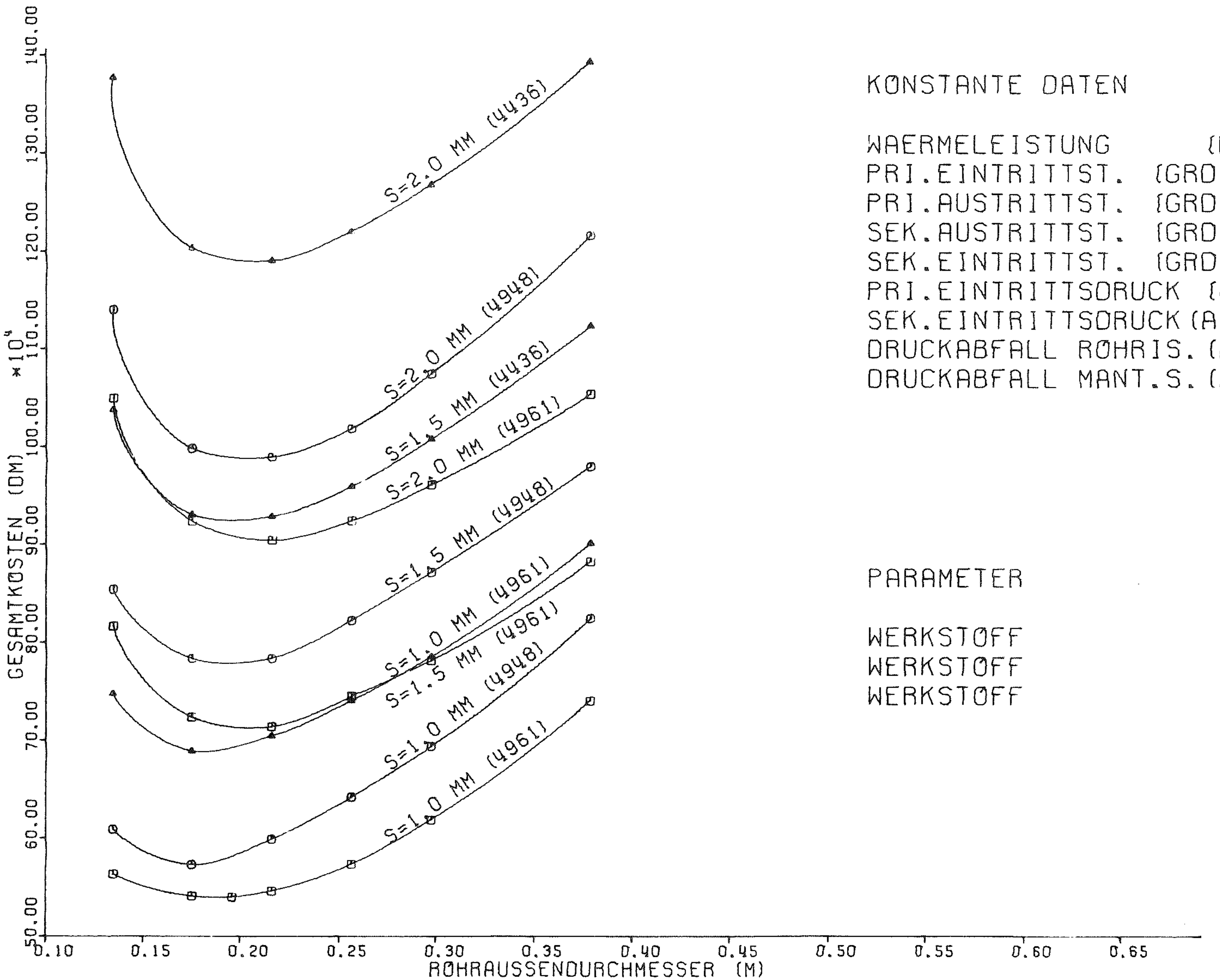
GFK/IRE	TAKTGEBER FUER MAGNETBAND E./A.-KØNTRØLLEINSCHUB	1546

Fig. 37 Electrical circuit

Fig. 38 Sodium-heat-exchanger



AUFBAU EINES NA-ZWISCHENWAERMETAUSCHERS



KONSTANTE DATEN

WAERMELEISTUNG	(MW)	0.2500E+03
PRI.EINTRITTST.	(GRD C)	0.5500E+03
PRI.AUSTRITTST.	(GRD C)	0.3800E+03
SEK.AUSTRITTST.	(GRD C)	0.5300E+03
SEK.EINTRITTST.	(GRD C)	0.3400E+03
PRI.EINTRITTSDRUCK	(ATA)	0.8000E+01
SEK.EINTRITTSDRUCK	(ATA)	0.1200E+02
DRUCKABFALL RÖHRIS.	(AT)	0.5000E-00
DRUCKABFALL MANT.S.	(AT)	0.5000E-00

PARAMETER		NR
WERKSTOFF	0.2000E+01	1
WERKSTOFF	0.7000E+01	2
WERKSTOFF	0.8000E+01	3

GESAMTKOSTEN ALS FUNKTION DES RÖHRAUSSENDURCHMESSERS
MIT DEM WERKSTOFF ALS PARAMETER

Fig. 39 Plot that has been taken over from an existing program and has been edited for publication using GRAPHIC

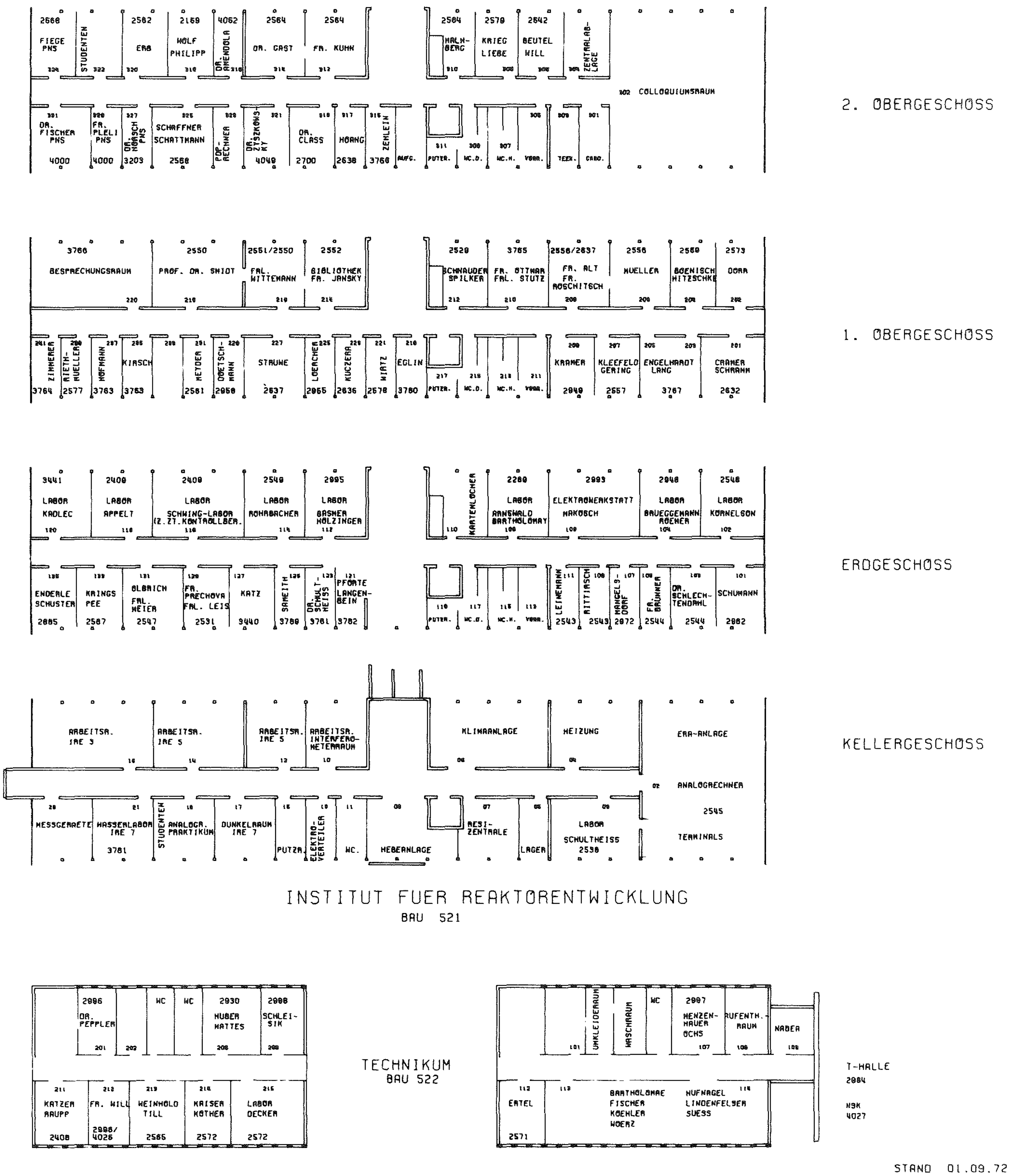


Fig. 40 Building layout of the Institut für Reaktorentwicklung

Appendix A

Concrete syntax of the GRAPHIC language

```
<program>          ::=  GRAPHIC ⊣ [<block>]* END GRAPHIC ⊣
<block>            ::=  BEGIN ⊣ <group> END ⊣ | <do> <group> END ⊣ | <group>
<group>            ::=  [[<declaration>]* [<statement>]*]*
<declaration>      ::=  <name-decl> | <proc-decl>
<proc-decl>        ::=  PROCEDURE <name> [ ( [<name>]* ) ]^1_0 ⊣ [<statement>]* END [RETURN <object>]^1_0
<name-decl>        ::=  DECLARE <name> <name-init>
<name-init>        ::=  ⊣ | INITIAL <object> ⊣
<do>               ::=  DO <a-expression> [] ⊣ | DO <a-expression> [] WHILE <la-expression> ⊣
<statement>        ::=  <graph-st> ⊣ | <la-st> ⊣ | <system-st> ⊣ | <proc-st> ⊣ | <block>
<la-st>            ::=  <st-id> [] <la-expression>
<la-expression>    ::=  <la-term> ['|' <la-term>]*
<la-term>          ::=  <la-factor> [& <la-factor>]*
<la-factor>        ::=  <a-expression> [<comparison-op> <a-expression>]^1_0
<comparison-op>    ::=  = | ¬= | > | ¬> | < | ¬<
<a-expression>     ::=  <term> [<a-op1> <term>]*
<a-op1>            ::=  + | -
<term>             ::=  <factor> [<a-op2> <factor>]*
<a-op2>            ::=  * | /
<factor>           ::=  <a-value> [** <a-value>]^1_0
<a-value>          ::=  <real> | <integer> | <l-value>
<l-value>          ::=  TRUE | FALSE | <name> | ¬<l-value> | (<la-expression>)
<system-st>        ::=  <standard> | <time> | <trace> | <test> | <take> | <reserve> | <release> | <get> |
                        <put> | <delete> | <file> | <compile> | <link> | <go>
```

```
<graph-st>          ::=  <st-id> [] <object>
<st-id>             ::=  SET [OBJECT []] <name> | DEFINE [OBJECT []] <name> | <gr-outp>
<name>              ::=  '[<char>]_1^8'
<gr-outp>           ::=  PLOT | PRINT
<la-st>             ::=  <st-id> <la-expression> | <control-st>
<control-st>        ::=  IF <la-expression> THEN <clause> FI ⊣
<clause>            ::=  <block> | <block> ELSE <block>
<object>            ::=  <name> | <specification> | <transformation> | <proc-st>
<specification>     ::=  <point> | <line> | <text> | <circle> | <axis> | <polygon> | <spline> |
                         <approximation> | <arc> | <semicircle> | <intersection> | <extrem-element> |
                         <shade> | <object-named> | <x-axis> | <y-axis> | <spec.-outp> | <npoint> |
                         <nline> | <transformation> | <collection> | <open>
<transformation>    ::=  <shift> | <enlargement> | <diminution> | <rotation> | <image> |
                         <transf. to axis>
<proc-st>           ::=  CALL <name> [([<object>]*)]_0^1
<collection>        ::=  [] ([<object>]*)
<char>              ::=  <alpha> | <sig> | <digit>
<alpha>             ::=  A | B | C | ... | Y | Z
<digit>             ::=  1 | 2 | 3 | ... | 9 | 0
<sig>               ::=  + | * | : | . | - | / | = | " | ;
```

The GRAPHIC language accepts as delimiters

ƀ , ; or ⊣.

The describing metalanguage uses the following symbols:

< > Angular brackets enclose non-terminal variables to distinguish from terminals

::= This symbol is used to define a rule for generating valid syntax by substitution of left parts by right parts.

$[\quad]_n^m$ All elements within the brackets may be repeated from n-through m-times. If n is omitted, one is assumed. If m = * the bracketed item may be repeated a deliberate number of times.

⊣ End of card

▯ This symbol is placed for ignorable words

| Separates alternative right parts of rules.

All non-terminals, which are not explained - like <point>, <line> etc. - should be taken from their description in chapter 4.

Appendix B

Abbreviations used in this report

AED	Automated Engineering Design
CAD	Computer Aided Design
CDL	Command Definition Language
CPU	Central Processing Unit
DV	Datenverarbeitung
GDP	Graphical Data Pool
GDPS	Graphical Data Processing System
GE	Graphical Element
GO	Graphical Operation
IBM	International Business Machines
ICES	Integrated Civil Engineering System
ICETRAN	ICES-FORTRAN
IRE	Institut für Reaktorentwicklung
MFT	Multiprogramming with a Fixed Number of Tasks (refers to OS/360 or OS/370)
MVT	Multiprogramming with a Variable Number of Tasks (refers to OS/360 or OS/370)
OS	Operating System
REGENT	Rechnergestützter Entwurf
RSUB	Ring with Common Subobject
RSUP	Ring with Common Superobject
TNL	Temporary Node List

References

/ 1/ Enderle, G., Katz, F., Leinemann, K., Schlechtendahl, E. G., Schnauder, H., Schumann, U., Schuster, R.: Erster REGENT-Halbjahresbericht, KFK-Externer Bericht 8/72-2, Juni 1972

/ 2/ Enderle, G., Leinemann, K., Schlechtendahl, E. G., Schnauder, H., Schumann, U., Schuster, R.: Zweiter REGENT-Halbjahresbericht, KFK-Externer Bericht 8/72-4, Oktober 1972

/ 3/ Roos, D. (ed.): ICES System: General Description, MIT, Department of Civil Engineering, R 67-49, September 1967

/ 4/ Schumacker, B. (ed.): An Introduction to ICES, MIT, Department of Civil Engineering, R 67-47, 1967

/ 5/ Jordan, J. C. (ed.): ICES-Programmers Reference Manual, MIT, Department of Civil Engineering, R 67-50, October 1967

/ 6/ Schlechtendahl, E. G., Schumann, U., (ed.): Erfahrungen mit dem Programmsystem ICES bei ingenieurtechnischen Anwendungen, KFK 1586, Mai 1972

/ 7/ Calcomp GmbH, Düsseldorf: Programme für Calcomp-Plotter der Serie 500, 600 und 700

/ 8/ Schumann, U., Schlechtendahl, E. G.: Algorithmen zur Verarbeitung von Baumstrukturen und ihre Anwendung in ICES, KFK 1536, Januar 1972

/ 9/ Knuth, D. E.: The Art of Computer Programming, Vol. 1, Reading 1963

/10/ Schumann, U., Schlechtendahl, E. G.: ICETRAN-Treestructure Routines to Save Direct Access Space and Debugging Time, Report, 8th Semiannual ICES Users Conference, San Francisco, January 1972

/11/ Encarnacao, J.:
Datenstrukturen für graphische Informationsverarbeitung, eine Übersicht, Computer Graphics Symposium, Berlin, October 1971

/12/ Williams, R.:
A Survey of Data Structures for Computer Graphics Systems, Computing Surveys 3, No. 1, March 1971

/13/ Lang, C. A., Gray, J. C.:
ASP - A Ring Implemented Associative Structure Package, CACM 11, No. 8, August 1968

/14/ Gray, J. C.:
Compound Data Structure for Computer-Aided Design - a Survey, Proceedings ACM National Meeting 1967

/15/ Sutherland, I. E.:
SKETCHPAD, A Man- Machine Graphical Communication System, Spring Joint Computer Conference, Proc. 329 - 345, 1963

/16/ Encarnacao, J. L.:
PRADIS, Ein Programmsystem für räumliche Darstellungen auf Displays, Elektronische Datenverarbeitung 7, 1970

/17/ Encarnacao, J., Hunger, J.:
Das interaktive Entwerfen und Zeichnen in drei Dimensionen auf Computer-Bildschirmgeräten, Elektron. Rechenanlagen 13, No. 1, Januar 1971

/18/ Leinemann, K., Schumann, U.:
KOSPRA - Entwurf einer Konstruktionssprache zur Beschreibung der Geometrie technischer Objekte, KFK-Externer Bericht 8/72-6 (1973)

/19/ Ross, D. T.:
The AED Approach to Generalized Computer-Aided Design, Proc. ACM 2nd Annual Conference, pp. 367 - 385, 1967

/20/ Kulsrud, H. E.:
A General Purpose Graphic Language, CACM 11, pp. 247 - 254, 1968

/21/ Comba, P. G.:
A Language for Three-dimensional Geometry, IBM Systems Journal, Vol. 3 and 4, pp. 289 - 307, 1968

/22/ Notley, M. G.:
A Graphical Picture Drawing Language, The Computer Bulletin, March 1970

/23/ Smith, D. N.:
GPL/I-APL/I Extension for Computer Graphics, Proc. Spring Joint Computer Conference, pp. 511 - 528, 1971

/24/ Bracchi, G, Ferrari, D.:
A Language for Treating Geometric Patterns in a Two-dimensional Space, CACM 14, pp. 26 - 32, 1971

/25/ Herzog, B.:
A Computer Graphics Language (DRAWL 70), National Technical Information Service, AD 715 952, August 1970

/26/ Mier, M. G.:
Versatile FORTRAN IV Computer Program for Producing Line Drawings Suitable for Publication Using the Calcomp Plotter, Air Force Avionics Laboraties, AFAL-TR-70-107, 1970

/27/ PL/1 Language Reference Manual, IBM Order No. GC 28-8201-3

/28/ Schnauder, H.:
GRAPHIC - Handbuch, to be published

/29/ FORTRAN IV (E Level Subset)
IBM Order No. C 28-6513-0

/30/ Gries, D.:
Compiler Construction for Digital Computers, John Wiley & Sons, New York, 1971

/31/ IBM System 360/OS, Supervisor and Data Management Services, Order No. C 28-6646-1

/32/ Schumann, U.:
Verwaltung von dynamischen Symboltabellen nach der HASH-Methode mit ICETRAN-Unterprogrammen, KFK-Externer Bericht 8/71-7, Oct. 1971

/33/ Logcher, R. D., Jackson, J. N.:
ICES TABLE II - Engineering User's Manual, MIT,
R 69-34, June 1969