PL ISSN 0209-1593

2 1983

prace naukowo – badawcze

Instytutu Maszyn Matematycznych

rok XXV

Druk okładki IMM zam. 81/83 nakł. 350 egz.

Instytut Maszyn Matematycznych

prace naukowo-badawcze Instytutu Maszyn Matematycznych

Adres redakcyjny: Instytut Maszyn Matematycznych
Branżowy Ośrodek INTE
02-078 Warszawa, ul. Krzywickiego 34
tel. 28-37-29 lub 21-84-41 w. 244

Prace naukowo-badawcze Instytutu Maszyn Matematycznych

Basic Notions for Interactive Programming of Industrial Computer Systems

by Henryk ORŁOWSKI

Software of many industrial computers consists of table drived systems or it is produced by program generators. The interactive methods are becoming more popular for programming such systems. This paper deals with the question-answering system for programming and contains definitions of basic terms and some requirements.

1. Introduction

There is a broad class of special application programming systems which are characterized by the fact that the programmer sets up a program by answering questions supplied by the system, rather than writing lines of program instructions and directives arranged by himself. Of course, this method differs a lot from commonly known programming methods, therefore many authors hesitate to call it "programming", and prefer to speak of e.g. program generation. However, because the primary factor of this activity is to have a program, and the final aim is the program itself, we will use the term "programming".

The method can operate only if enough program pieces are prepared in advance and if they are ready for use in the system library. This approach has a number of advantages as well as disadvantages, which will be discussed later in this paper. The process of programming for the method discussed amounts in fact to:

- choosing the program pieces from the library,
- adjusting the sequence in which these programs will run,
- determining the parameters(constant data) for programs,

Author is obliged to members of Technical Committee on Special Applications Programming of European Workshop on Industrial Computer System for valuable comments and suggestions which helped to establish the present version of the paper.

- establishing circumstances at which the program or some of its parts should run,
- connecting programs to given data bases of the system (determining access rights, access paths, formats and linking).

The most commonly known example of the method is the generation of operating systems for a given hardware configuration and for given requirements of a user. For this application, a program called "the operating system generator" is used. Working in the older batch mode, the user fills a proper questionnaire and answers questions concerning the capacity of a main memory, types of external storage (tapes or disks and their types), numbers (addresses) for particular input/output channels, etc. Working in the modern mode, the user introduces similar data in an interactive way, giving the answers to questions displayed on the CRT. Independetly of the way of introducing data, the generator produces the required operating system. Therefore such a method is commonly known as program generation, and is now used not only for generation of operating systems [1].

Even earlier than program generators, the fill-in-the-blanks or fill-in-the-forms methods were used for computer programming in industrial control systems. As a rule, in this approach the control program is prepared in advance, debugged and in fact proved in many applications by the system supplier. The data from the blanks are parameters for the program and determine addresses of inputs to which signals come from a plant, times in which the computer should process the input data, values of constants for algorithms used to process the input data, etc.

Looking for a name adequate to the above mentioned methods of programming, I propose the name "Question-Answering (Q-A) Programming", as I have found that the common aspect of all these methods is the facility of extracting the information necessary for programming in a form of answers to questions written in blanks (batch mode) or displayed on CRT (interactive mode). The Q-A programming differs from other methods of programming where programmers are free in arranging programs, noting down lines in Fortran, Cobol, or even in machine code.

The aim of this paper is to propose basic terms and definitions concerning question-answering programming. More details on Q-A programming can be found in the paper by G.Musstopf, H. Orlowski and B. Tamm [2].

2. The Programming System

By Question-Answering (Q-A) Programming we mean the method of producing an object program, in which

- the Q-A language is used to supply the information for a program,
- the information included in answers is processed in one of the two following ways:
  - by program generation,
  - by processing of tables.

By Q-A language we mean the language which presents in a formalized way source information where every sentence consists of an answer to a previously posed question.

By fill-in-the-blanks language (FIB language) we mean the variant of Q-A language in which the questions together with the blanks in which answers are written by programmers are printed on paper forms.

By interactive Q-A language we mean the variant of Q-A language where the questions are stored in a computer memory and displayed successively on the terminal. After displaying a question, one expects the answer to be given by a programmer (an operator).

By program generation we mean here the production of a program performed in a way described below. In the computer library, preserved in an on-line connected storage, there are program modules previously translated, optimized, debugged and tested. Information which is contained in answers controls the choice of proper modules, connects modules to a single program and provides the parameters for that program. In some cases a program is re-translated.

By table processing we mean the execution of programs which are controlled by data placed previously in proper tables by a program called "table generator". The data from tables provide the operating system with information which program units should be processed, under what circumstances, and with what inputs and outputs of the system. The tables also contain parameters for programs.

The following three interfaces are common to all kinds of Q-A programming systems and are independent of the way of processing (Fig. 1):

i. source information

ii. edited information

iii. output data

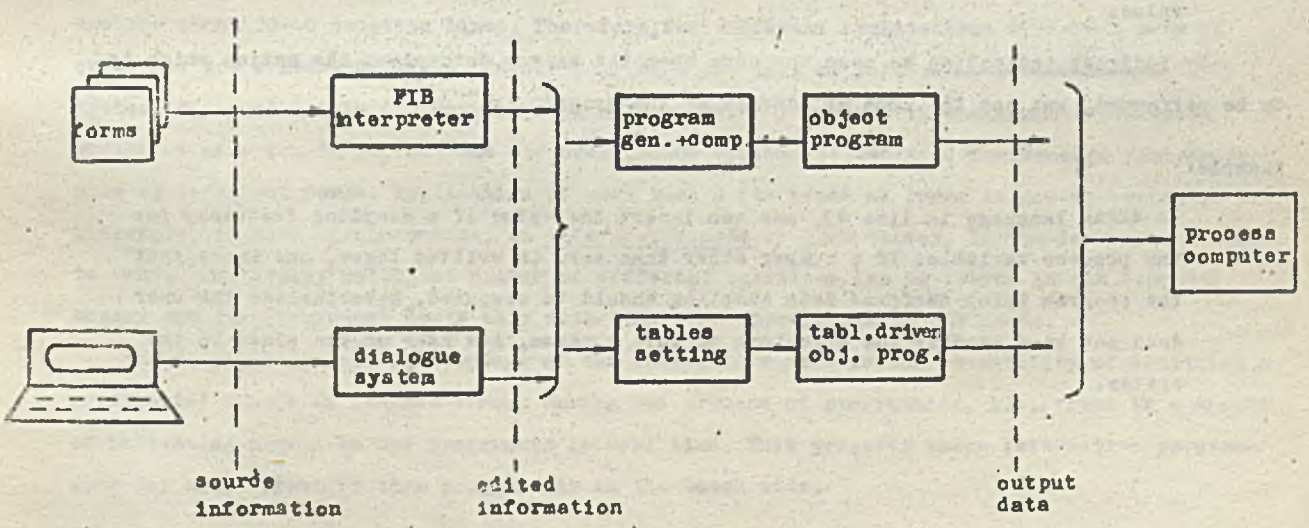


Fig.1. Software interfaces of the Question-Answering Programming System

By source information we mean data in the form of a set of answers, which are to be put into a system.

By edited information we mean data in internal code, ready to control program generation or ready to be placed into the tables of the table processing system.

By output data we mean here the output data of the target program, e.g. output data of an industrial computer system.

We belive that appropriate requirements should be specified for the three above presented interfaces. Then, the comparison of systems and their portability could be established independently of their internal structure. Only the source information part is consider below.

3. Source Information

Source information, which should be presented in Q-A language, consists of the following components:

- indication of a program segment (subroutine) which should be executed,
- determination of circumstances for executing the program segment,
- determination of addresses for communication of an industrial computer system with the external world,
- determination of parameters (constant data) for programs.

The program segment can be indicated directly or indirectly. By direct indication we mean the case when the answer determines the name or the address of the indicated subprogram.

Example:

> In SZPAK language [3,6] in line 35 one can insert the name of a program, written in Fortran, which is executed when the current process variable exceeds the upper warning value.

By indirect indication we mean the case when the answer determines the action which is to be performed, but not the name or address of the program.

Example:

> In SZPAK language in line 23, one can insert the value of a sampling frequency for the process variable. If a number other than zero is written there, one knows that the program which performs data sampling should be executed. Nevertheless the user does not know exactly the structure of this program, its name or its place in the system.

Determination of circumstances for executing the program segment consists of indication of necessary conditions for the program activation. With respect to the way of realization one can have the following cases or their combination:

1. Specifying time events in one of two ways:

   1a. Specifying an absolute time.

   1b. Specifying elapse of time after the given event.

2. Occurence of a determined external event (e.g. the interrupt signal of a given input).

3. Meeting a specified condition during executing of a program (e.g. the calculated value of a given parameter exceeds the alarm value).

Determination of addresses for communication of an industrial computer system with the external world consists of:

1. Determination of external devices, channels or process interfaces from which input data for processing should be taken.

2. Determination of external devices, channels or process interfaces to which output data should be sent.

For man machine communication the scope of activity of each operator should be exactly determined if more than one operator is working.

Determination of parameters. To perform programs, one needs constants and variable data. Variable data are taken from the environment, constants are introduced in Q-A way. These constant data are called parameters.

Source information can be introduced to the system as it was said, in a batch or in an interactive mode. For the batch mode the proper forms must be used. The information capacity of a form is limited by the geometric dimensions of one sheet of paper. The SZPAK forms, for instance, enclose about 30-40 question lines. Therefore, for different applications different sets of questions are necessary, and one needs more than one sort of forms. For instance, the SZPAK system, designed mainly for data logging and supervisory control of continuous industrial processes uses two forms, and the Japanese PROCOS system [4] designed for broader applications uses 13 different forms. Application of more than a few types of forms is now convenient. Therefore, in such circumstances, an interactive mode is much better. If the interactive mode is used, practically unlimited number of different questions can be stored in the computer memory and the programmer deals only with questions appropriate to his needs.

The second essential advantage of the interactive mode is the possibility of detecting a substantial amount of program errors during the process of programming, i.e. there is a chance of indicating errors to the programmer in real time. This property makes interactive programming far more effective than programming in the batch mode.

4. Questions

The information mentioned above is obtained from a programmer in a form of answers to questions. So the important term connected with the Q-A system is the definition of a question.

The semantic content of question consists of:

- Name of the program,
- Parameters (constant data),
- Addresses,
- Directives.

By directive we mean information necessary for the correct program production, which is introduced in the program itself. Reference data and labels for the end of a sequence of questions are examples of directives.

Reference data give reference to answers or groups of answers, but don't influence the program preparation.

Examples:

In SZPAK language, questions are grouped on forms which are referred to the process variables. Therefore, the first question of a form concerns the determination of a process variable to which the answers will be written on this form. Other examples of reference data are: name of a programmer or the data of answering a question.

From a syntactic point of view answers are very simple alpha-numeric strings, therefore it is important to avoid errors arising from mixing up answers. There are two ways of preventing such errors: to use sequencing or to add reference data to each question.

If answers from a sequence in which consecution of elements cannot be changed, we can refer answers to questions by a single reference datum for one sequence. Answers collected in an interactive system or punched on paper tape are examples of this kind. In both cases, it is sufficient to put the reference data immediately before a sequence of questions and to end the sequence by a proper label.

However, there are situations, when one cannot be sure that answers form a proper sequence on the input device, e.g. when answers are introduced by means of punched cards. In these cases, questions should be extended so that besides the basic part the answer to each question will contain information enabling the computer to place this answer in a proper order. This method is used for instance in the BICEPS language [5]. At the beginning of each answer, the name of the process variable connected with this answer is placed before the sequence number of the question within the BICEPS form.

The semantic content of questions discussed above determines the expected content of answers. However, in addition to the explicitly expressed meaning of a question, there is a certain implicit meaning. Let us call it the background content of the question. The background content has not to be revealed in a question if we do not expect an answer to it, but it is important for determining the algorithms which have to be realized by a program. Let us consider the following example.

Example:

> In a certain FIB language, the following text is written in line x: "The upper warning value". The explicit content of question connected with this line is as follows: What is the value of the parameter describing the upper warning limit of the expected and allowed changes of values of the process variable?

The background content of this question consists, among others, of the following:

i. The answer could be given in physical units accepted for the process variable in question (but not in millivolts of an input signal to an a/d converter).

ii. According to a convention determined in (i) the answer should be written as a variable of type REAL (i.e. type INTEGER is not allowed in this case).

iii. The answer can be given as percentage of the full range of the expected changes of process variable values(i.e. from 0 to the value of the upper bound).

iv. According to the meaning determined in (iii), the answer should be given as a number of type INTEGER ranging from 1 to 7, according to the following table:

INTEGER number	% of the range
1	95
2	90
3	85
4	80
5	75
6	70
7	60

Evidently the programmer and the implementor have to know the background content, but:

1. The implementor has to know the whole background content, hence we require that "the definition of the language has to include the whole background content".

2. The programer has to know only those facts from the background content which are necessary for his work. For instance, he could know, in the above example, either (i) , (ii) or (iii), (iv), This implies the following requirement: "The programmer's manual has to contain the information referring to the background content, which is necessary for a correct programming in the field which this manual deals with".

3. In the PIB version of Q-A language, information concerning the background content is not, by definition disclosed to the programmer on forms. In the interactive version of Q-A language, one can introduce a possibility of presenting the background content in the following cases:

- at request of the programmer (HELP command),
- automatically, if the system discovers an error in the answer given by the programmer (e.g. a missing parameter).

The question arises, if besides the discussed background content yet another content, not revealed by the implementor, but having influence on e.g. the effectivity of the system, can be incorporated. Let us call such content the depth content. This information provides implementation details. The following guideline is essential: one should not set up in the depth content such information, the lack of which would cause the same source programs to be executed defferently, depending on their implementations. For instance, the method of rounding arithmetic values cannot be placed in the depth content, although it is not important for the majority of programmers, because they are mostly interested in how many decimal places are necessary to obtain correct results.

The knowledge of the depth content is necessary for everybody who wants to make changes in the system, e.g. for transporting the system to other computers.

5. Documentation

The documentation of Q-A programming consists of two basic parts: external and internal.

The external Q-A documentation consists of:

- A specification of the destination and functions of the industrial computer system.
- A specification of tasks which should be performed and a description of conditions under which these tasks may or should be performed.
- A specification of both the general limits of the Q-A programming and the limits due to the given implementation.
- A description of the Q-A language.
- A description of the debugging facilities and a list of possible error messages.

The external documentation should have the following forms:

1. A definition of the Q-A language, which provides a document for implementors.
2. A description of the Q-A programming system, i.e. a document which is worked out by implementors defining the fixed system.
3. A programmer's manual of the Q-A language; this document determines in a way intelligible to users how to answer question, i.e. how to program.

The internal documentation specifies the way of performing the given implementation of a system. This can be the "know-how" of the supplier, and usually need not be revealed to users of the system.

The above presented types of documentation deal with the Q-A programming systems themselves. But it is important to notice that Q-A programming systems are not only a good tool to supply programs, but also to stimulate a good, i.e. correct, concise, and legible system drscription.

The system description of the particular application should describle all features and parameters of the system which are important for the system user from the point of view of purposes for which the computer system has been applied. Examples are: a list of measured values, sampling frequencies of individual values, alarm conditions, input/output addresses of signal lines from and to the plant.

The application software documentation should describe in a concise and complete manner the software applied for the computer system, specially for the realization of the individual user requirements. Usually the kernal part of the application software documentation consists of listings of the source program. Of course, if programs are written in languages such as IRT-Fortran, IRT-Basic etc. the listings consist not only of information essential to the user but also of information necessary for program compilation and computer operation. Therefore, the information essential to the user is spread among a large number of details necessary for the operation of the computer. In consequence, in such cases for legibility and convenience of usage, one requires two documentation sets: the synthetic tabels placed in the system description and the listing of the source programs placed in the application program documentation. Such solutions cause two inconveniences:

- additional work,

and, what is more important,

- the risk that both forms of documentation are not consistent, especially if software undergoes changes during modernization of the plant.

The above considerations show advantages of Q-A programming systems, where application software documentation gives in a natural manner homogeneous specification both for the needs of the system user and for computer operation.

The additional advantage of the application of Q-A programming is the standardisation of software documentation whithin the framework of the supplier of the software. Simply every employee must use the same forms or gets the hard copy in the same form, after interactive programming.

Conclusion

The above definitions and requirements do not cover the whole field of program generators and fill-in-the-blanks systems. Author considers the presented approach as the starting point to further work and will be very obliged for comments.

References

[1] Tamm B.: Interactive integrated system for program generators and application packages. EWICS TC 4 working paper: POL-E 5/80.

[2] Kusstopf G., Orłowski H., Tamm B.: Program Generators for Process Control Applications. IFAC/IFIP SOCOCO 1979, Prague.

[3] Orłowski H.: The fill-in-the-form language SZPAK and its supplementary language PF. Purdue Europe TC 4 on POL. 1975 Presentation, pp.49-64.

[4] JEIDA: Introduction for PROCOS. Minutes II Ann. Meeting Int. Purdue Workshop. Purdue Univ. 1974, Part II, pp. 225-226.

[5] BICEPS Language Manual. General Electric Co. Phoenix, GET-6065, 1969.

[6] Aderek A.: SZPAK-77: Fill-in-the-Blank Programming System for Industrial Process Control. EWICS TC 4 working paper: POL-E 4/80.

STRESZCZENIE

Oprogramowanie wielu komputerów zastosowanych w układach automatyki przemysłowej składa się z systemów sterowanych tablicami lub jest produkowane przez generatory programów. Metody interakcyjne stają się coraz bardziej popularne w programowaniu takich komputerów. Artykuł dotyczy systemu oprogramowania typu pytanie-odpowiedź i zawiera definicje podstawowych terminów oraz niektóre wymagania dotyczące tych systemów.

СОДЕРЖАНИЕ

Программное обеспечение многих ЭВМ применяемых в системах промышленной автоматики использует системы управляемые таблицами или производится с помощю генератора программ. Интерактивные методы становятся всё более популярными в программировании таких ЭВМ.

В настоящей статье рассматривается система программного обеспечения типа вопрос-ответ. Приводятся определения основных понятий и некоторые требования предъявляемые к таким системам.

Prace naukowo-badawcze Instytutu Maszyn Matematycznych

PHIL 5

Jacek OLSZEWSKI

This report defines a programming language for operating systems implementations. The language, Phil 5, conforms to certain design methodology of operating systems built as hierarchical structures of processes.

C o n t e n t s

1. Introduction

Process Hierarchy Implementation Language is a name chosen for the programming language described in [2] as a methodological tool for operating system design and programming. Methodology presented in [2] assumes an operating system to be certain structure of hardware processor activities called processes (not to be mixed up with what has so far been termed process in the literature). A process is understood here as a data structure and a set of instructions concerning these data. All data within the operating system can be divided into separate parts, each of which is a private (local) data of a certain process. There is no notion of data being common to several processes. Instead, there may be another process for which the data is private and it can be accessed only by instructions of the other process. Each of such processes can be active or passive. Processes may call one another, the calling one becoming passive and the called one - active. Then, the called one may rasume the calling process, itself becoming passive. Interrupts are also considered as process calls; calls not by other processes but from outside the system. On the other hand, an interrupt makes some currently active process become passive. If a process calls or resumes another one when it is active, the call or resumption is delayed until it becomes passive - an automatic exclusion of calls of the same process (short time scheduling) is assumed; However, interrupts activating the same process consecutively do not exclude each other automatically. It is possible that a process activated by an interrupt is itself interrupted and activated again. In order to avoid such interleaving of activations means are provided to impose the non-interruptable mode upon arbitrarily chosen processes.

A process that calls or resumes other processes and, in turn, itself is called or resumed can be considered as an element of certain hierarchical structure of calls and resumptions. An operating system built as such a structure is to remain constant during the system run; the number of processes and their positions within the structure do not change. This allows for better understanding of the whole system, and perhaps, for easier formal verification techniques.

Phil is a machine oriented higher level programming language that comprises the concepts of the process and the structure of processes mentioned above. It is based on Pascal including most of Pascal's data and control structures and excluding all those features that would make a dynamic allocation of resources necessary. Phil's basic assumption is not to require any run time support system nor any operating system kernel. Phil's text is to be compiled into machine code, then loaded into the store and run without any reference to the Phil compiler.

Text written in Phil starts with a number of program declarations, each of which may be understood as a process type definition. Then, there are declarations of processes, each of which creates a process or processes according to the program referred to in the declaration (analogously to the creation of variables according to their type). Moreover, the declaration shows which other processes may be called by the declared one. So, process declarations may be considered as a process structure building program. The text ends with a process call or calls - depending on the number of processors available in hardware - of those processes that are to be activated first. Another assumption concerning the language is that programs can be compiled separately of each other. Hence, a change in one program should not involve compilation of other programs.

The work on Phil implementation has been planned as a series of languages and their compilers starting from slightly extended PL/0 (cf [3]) and ending with the full programming language presented in [2]. The compilers are being programmed and tested in COPS - a meta-compiler developed at IMM and implemented on IBM/370 (cf. [1]). The first element of the language and compiler series, Phil 2.5, may be interesting only from the point of view of compiler writing methodology using COPS. Phil 5 is the series element which may also be interesting as a system programming language. However simple, it comprises all the basic concepts and ideas of the methodology presented in [2] and outlined very shortly above.

References

[1] J. Borowiec and others /ed. J. Olszewski/: Compiler Production System, IMM Warsaw, 1981, /in Polish/

[2] J. Olszewski: Design of operating system structures, WNT Warsaw, 1981, /in Polish, resume also in English/

[3] N. Wirth: Algorithms + Data Structures = Programs, Prentice Hall Inc. Englewood Cliffs, N.J., 1976

2. Syntax notation

The Phil 5 syntax is defined in a notation which is required by COPS to produce the syntax analyser automatically. Non-terminal symbols are denoted by identifiers - usually short hand notations of English words. Terminal symbols are strings enclosed in quote marks. A non-terminal symbol definition consists of one or more productions, each of which starts with the production identifier. Productions defining one non-terminal symbol have the form:

```
P1. S=T1|
P2.   T2|
...
Pn.   Tn;
```

where P's are production identifiers, S is the non-terminal symbol and T's are alternative terms of the definition. Production identifiers are necessary for the purposes of formal description of the language semantics, required by COPS. Each of terms T1, ..., Tn stands for a list of terminal and non-terminal symbols of the language, separated one from another with a comma.

There are two special symbols, meanings of which are predefined in COPS; namely, 'EPS'- denoting an empty symbol, and 'STRING'- denoting a sequence of any characters, enclosed by quote marks. If a quote mark itself is to appear within a string, it should be marked with two adjacent quote marks.

Examples of strings:

'PHIL 5' 'MARKS & SPENCER' ''''

COPS requires the syntax definition to be splitted into two parts: lexicon and syntax (proper). Each of the parts starts with word LEXICON and SYNTAX correspondingly and ends with a production at the end of which there is a dot instead of a semicolon.

3. Vocabulary (lexicon)

3.1 Letters, digits, separators and comments

Letters are defined by the following set of productions:

```
L1. LETT= 'A'|
L2.       'B'|
...
L26.      'Z';
```

Similarly, digits:

```
D1.  DIGT= '0'|
D2.        '1'|
...
D10.       '9';
```

A space character has in COPS the meaning of a separator for any language definition. Other separators for a particular language are to be listed in a COPS directive. For Phil 5 the directive is the following:

```
SEPARATOR='() :;=+- * /# >< ,. & []'
```

Two other COPS directives are used to show how comments begin and end in the language:

```
COMBEG=' (*'.
COMEND=' *)'.
```

So, comments in Phil 5 are enclosed in brackets and stars.

3.2 Identifiers

Identifiers are sequences of letters and digits, each starting with a letter:

```
I1. IDEN=LETT|
I2.      IDEN, LETT|
I3.      IDEN, DIGT;
```

Examples:

```
A   A0001   MARKS   SPENCER
```

3.3 Numbers

Numbers are unsigned sequences of digits:

```
N1. NUMB=DIGT|
N2.      NUMB,DIGT;
```

Examples:

```
1984    3   001
```

3.4 Operators

Operators are special characters or character pairs grouped in three definitions which correspond to addition, multiplication and relation kinds of operators:

```
01. AD_OP='+'|
02.       '-';
03. MLT_OP='*'|
04.        '/';
05. REL_OP='#'|
06.        '>'|
07.        '<'|
08.        '>='|
09.        '<=';
```

3.3 Other symbols

Other symbols are defined by their appearance as terminal symbols in the syntax description. They include all reserved words of the language and special symbols like brackets, the assignment symbol ':=', etc.

4. Declarations and scope rules

Every identifier must be declared unless it is a standard one. Standard identifiers may be considered as predeclared, valid throughout the whole text in Phil 3. They may be redeclared as any other identifier declared in a normal way.

Since Phil 3 is a block structured language every declaration has its scope. For types, variables and procedures the scope extends from the declaration of such an object to the end of the block within which the declaration occurs. For programs and processes, the scope extends to the end of the whole text in Phil 3. For records, identifiers of their fields are valid only in the field designators (cf. section 7.1).

5. Types

There are two basic types of data, integer and character, and three structured types, array, record and pointer. A type declaration must conform to the following definition:

```
T5. TYPE_D=IDEN |
T6.         'ARRAY', '[', NUMB_L, ']', 'OF', TYPE_D |
T7.         'RECORD', FIELD_L, VARIANT, 'END' |
T8.         '|', IDEN;
```

5.1 Basic types

Basic types are predeclared and denoted by the standard identifiers:

INTEGER , values of this type are integers from the range depending on the hardware,

CHAR , values of this type are characters of the set accepted as the external representation for the language implementation.

5.2. Arrays

An array type declaration (cf. production T7 above) specifies the type of the array components, the number of indices (dimensions) and the maximum value of each index. The number of indices and maximum index values are shown as the list of numbers:

```
L1. NUMB_L=NUMB |
L2.         NUMB_L, ',', NUMB;
```

The minimum value of each index is always 1.

Examples:

```
ARRAY [7] OF   DAY;
ARRAY [4,13] OF CARD;
```

5.3 Records

A record type definition (cf. production T8 above) specifies a number of fields and a variant part of the record. There may be 0 or more fields in so called field list:

```
F1. FIELD_L='EPS'|
F2.          FIELD|
F3.          FIELD, ';', FIELD_L;
```

The non-terminal symbol FIELD denotes, in fact, a number of the record components of the same type:

```
F4. FIELD=IDEN_L, ':', TYPE_D;
```

where the symbol IDEN_L represents a list of one or more identifiers:

```
L3. IDEN_L=IDEN |
L4.        IDEN_L, ',', IDEN;
```

The scope of the identifiers is the record itself, and they are also accessible in field designators (cf. section 7.1).

A record may have a variant part beginning with the word 'CASE' followed by so called a tag field, then the word 'OF' and a list of variants. The value of the tag field is to indicate which of the variants is chosen for a particular instance of record.

```
V1. VARIANT='EPS'|
V2.         'CASE', IDEN, ':', IDEN, 'OF', VARIANT_L;
```

The type of the tag field may only be a basic one. Hence, the symbol IDEN appears in V2 where the tag type is to be defined.

The variant list specifies one or more data structures each of which is itself like a record - consisting of a list of fields and a variant part of the structure. Each structure is labelled by one or more constants (a constant list).

```
V3. VARIANT_L=CONST_L, ':', '(', FIELD_L, VARIANT, ';'|
V4.           VARIANT_L, ':', CONST_L, ':', '(', FIELD_L, VARIANT, ')';
```

The constants should be either integers or characters according to the specified tag field type. However, a string may also be used as a label where the tag field is specified as CHAR. In such a case only the first character of the string will be recognized.

```
L5. CONST=NUMB |
L6.       'STRING';
L7. CONST_L=CONST |
L8.         CONST_L, ',', CONST;
```

Examples:

```
RECORD NAME, CHRISTIANNAME: ARRAY [10] OF CHAR;
       AGE: INTEGER;
       SEX: CHAR END;

RECORD REGISTRATION: ARRAY [7] OF CHAR;
       YEAR: INTEGER;
       CASE VEHICLE: CHAR OF
       'CAR': (NROFSEATS: INTEGER);
       'VAN', 'TRUCK': (WEIGHT: INTEGER) END;
```

In the last example only first letters of the labels will be recognized: C, V and T.

5.4 Pointers

A pointer type declaration (cf. production T8) introduces a type whose values are pointers to variables of the type specified in the declaration. A representation of pointer values is chosen to be a standard two-field record declared as follows:

```
RECORD LOC, SIZE: INTEGER END;
```

In other words, a pointer bound to any type is itself a record, the first field of which gives the location of the pointed variable, the second - its size.

5.5. Named types

Types may be declared so as to have names. Declarations of the named types form declaration lists:

```
T3. TYPE_L=IDEN, '=', TYPE_D |
T4.        TYPE_L, ';', IDEN, '=', TYPE_D;
```

Apart from the basic types the following structured types are predeclared:

```
POINTERVAL=RECORD LOC, SIZE: INTEGER END;      (cf. section 5.4 )
LINK=RECORD STATUS: ... ;  (* ITS TYPE IS HARDWARE DEPENDENT *)
           BASE, RETL: INTEGER END
```

The latter is concerned with so called links through which processes can be manipulated and resumed (cf. standard procedures HOLD and RESUME in section 10) . The STATUS field is for an internal representation of the process status (interruptable or non-interruptable, etc.) , BASE - for the process base address (depends on where in the memory the process is placed) , and RETL - for the process return address after calling another process or being interrupted.

6. Variables

Variables are defined by their declarations:

```
VB5. VAR_DF=IDEN_L, ':', TYPE_D;
```

Such a declaration specifies a number of identifiers and a type definition, according to which the identified variables are created.

Examples:

```
I, J, K:INTEGER;
C1, C2: CHAR;
P:↑ INTEGER;
A:ARRAY [4,4] OF INTEGER;
R:RECORD N:ARRAY [10] OF  CHAR;
        C: INTEGER END;
```

7. Expressions

Expressions consist of operands, arithmetic operators and parenthesis.

7.1. Operands

Operands are numbers, characters, strings and so called designators. A designator is a denotation of a variable or of its component, if the variable is an array or record or pointer.

```
VD1. VAR_D=IDEN |
VD2.       VAR_D, '[', EXPR_L, ']' |
VD3.       VAR_D, '.', IDEN |
VD4.       VAR_D, '↑';
```

An array element (production VD2) is denoted by the array designator and an expression list enclosed in array brackets, which gives values of indices of the element concerned.

```
A3. EXPR_L=EXPR |
A4.        EXPR_L, ',',EXPR;
```

It may happen that values of some of the expressions lie outside limits of indices defined in the array declaration (cf. section 5.2). Normally, run-time errors would be reported. Here, as no run-time system is to be required by the language compiler, each index value is calculated according to the following formula:

$$(e-1) \bmod m + 1$$

where e is the value of the corresponding expression and m - the maximum value of the index.

A record element (production VD3) is denoted by the record designator and a field selector.

The designator P↑ denotes the variable pointed by the pointer P. Since the pointer P may also be considered as a record of the type POINTERVAL, P.LOC and P.SIZE are also designators.

Examples (cf. section 6):

J	(INTEGER)
C1	(CHAR)
P	(↑INTEGER)
P↑	(INTEGER)
P.SIZE	(INTEGER)
A [2,2]	(INTEGER)
R.N [2]	(CHAR)

7.2 Arithmetic operators

There are two groups of arithmetic operators listed in the lexicon (cf. section 3.4): adding operators (AD_OP) and multiplying operators (MLT_OP). Their precedences are specified by the syntax of expressions.

```
E1. EXPR=EXPR, AD_OP, TERM |
E2.      AD_OP, TERM |
E3.      TERM;
TR1. TERM=TERM, MLT_OP, FACTOR |
TR2.      FACTOR;
```

First, the multiplying operators are executed, then, the adding ones. Operators of the same precedence are executed in sequence from left to right. The precedence rule may be broken by parenthesis.

```
FR1. FACTOR=CONST |
FR2.        VAR_D |
FR3.        '(',EXPR, ')';
```

First, the parenthesis is executed, then the rest of the expression.

The operators +, - and * have their usual meaning. The division operator / means the truncated quotient of its operands.

Examples (cf. section 6):

```
1981
I*J/ (J-K)
'+'
A [2,3] +A [I,J]
R.C*3
```

8. Conditions

Conditions consist of expressions, relation operators and the conjuction '|' and disjunction ' & ' symbols. Apart from the operators listed in the REL_OP symbol definition (cf. section 3.4) the 'ODD' and '=' symbols are also considered relation operators. Conditions are evaluated according to the following precedence order:

```
REL_OP, '=', 'ODD'
'&'
'|'
```

yielding a result of logical value (true or false).

```
CN1. COND=COND, '|', COND_T|
CN2.      COND_T;
CN3. COND_T= COND_T, '&', COND_F|
CN4.         COND_F;
CN5. COND_F='ODD', EXPR|
CN6.        EXPR, '=', EXPR|
CN7.        EXPR, REL_OP, EXPR;
```

Examples (cf. section 6):

```
ODD I & J=K
C1≠'A'|C2='B'
P <>3
A [K,2] <= R.C&R.N [5≠'X'
```

9. Statements

Statements denote actions. There are nine different kinds of statements; six of which are called simple statements: assignement, procedure or process call, compound statement, repetition, selection, and empty statement.

```
S1. S_STMT=VAR_D, ':=', EXPR|
S2.        IDEN, ARGS|
S3.        'BEGIN', STMT_L, 'END'|
S4.        'REPEAT', STMT_L, 'UNTIL', COND|
S5.        'CASE', EXPR, 'OF', CASE_L, 'END'|
S6.        'EPS';
```

Apart from the six simple statements there are also if-, while-, and for-statements.

```
S8. STMT=S_STMT|
S9.      'IF', COND, 'THEN', S_STMT, IF_TAIL|
S10.     'WHILE', COND, 'DO', STMT|
S11.     'FOR', IDEN, ':=', EXPR, 'TO', EXPR, 'DO', STMT;
```

9.1 Assignements

The designator to the left of the ':=' symbol denotes a variable or its component, if it is an array or record (cf. production S1 above). After the assignement the variable has a new value; the value of the expression standing to the right of the ':=' symbol. Types of the variable and the expression must be compatible. A string of the length m may be assigned to an array of n characters. If m < n, the string is extended with space characters so as to match the array size. If m > n, characters n+1, n+2, ... , m are ignored.

Examples (cf. section 6):

```
J:=0
C1:='?'
P1:=I*J/(J-K)
A[1,1]:=A[2,2] *A[3,3]
R.N:='ARTHUR'
```

9.2 Procedure and process calls

Procedure and process calls have the same form (cf. production S2). They may have arguments, i.e. a list of expressions:

```
A1. ARGS='EPS'|
A2.      '(', EXPR_L, ')';
```

9.2.1 Procedure calls

A procedure call stands for evaluation of its arguments, substitution of the procedure parameters by the evaluated arguments, and execution of the procedure body (cf. section 10). Types of arguments must be compatible with their corresponding parameters. Evaluation of an argument involves evaluation of indices or field selection, if it is a component of an array or record.

If a formal parameter is assigned to in the procedure body, the corresponding evaluated argument must be a designator.

Examples (cf. section 10):

```
PUT(CURRENT);
GET(NEXT)
```

9.2.2 Process calls

A process call stands for evaluation of its arguments, creation of so called argument record, fields of which are representations of the arguments, and certain action which stops the calling process and activates the called one. If an argument is a designator, a pointer value that points to the denoted variable is defined and placed in the corresponding field of the argument record. Otherwise, the value of the argument is placed in the field.

From the point of view of the calling process the call is a statement that lasts as long as the process resumption does not take place. Such a resumption is to be made by the called process by means of two standard procedures, HOLD and RESUME (cf. section 10).

9.3 Compound statements

Compound statements (cf. production S3) denote sequences of actions, denoted in turn by list of statements.

```
S17. STMT_L=STMT |
S18.         STMT, ';', STMT_L;
```

9.4 Repetitions

A repeat statement (cf. production S4) specifies repetition of the statement sequence. After each execution of the sequence the condition is evaluated. If it is true, the repetition stops.

Example:

```
REPEAT I:=J; J:=A [J]   UNTIL    J=0
```

9.5 Selections

A case statement (cf. production S5) specifies execution of one of the statements listed in the case list and labelled by lists of constants.

```
S14. CASE_L=CASE_D|
S15.       CASE_L, ';', CASE_D;
S16. CASE_D=CONST_L, ':' STMT;
```

First, the expression is evaluated, then its value is matched consecutively against the labels until the matching label is found, then the statement labelled by the label found is executed. The type of the expression should be either INTEGER or CHAR. Correspondingly, labels should be numbers or characters. String labels are also accepted although their first characters only are compared with the expression value. If there is no matching label found, the last case is chosen.

Example:

```
CASE C OF
'A', 'B': X:=1;
'C': BEGIN X:=3; Y:=0 END;
'D': X:=2;
'OTHERWISE': BEGIN X:=0; Y:=0 END END
```

9.6 If statements

There are two different forms of the if statement (cf. production S9): so called if-then statement and if-then-else statement.

```
S12. IF_TAIL= 'EPS' |
S13.          'ELSE', STMT;
```

An if-then statement specifies evaluation of the condition and, depending on the result - true or false - execution or skipping of the statement following the 'THEN' symbol. An if-then-else statement specifies evaluation of the condition and, depending on the result, execution of the statement following the 'THEN' or 'ELSE' symbol. Notice that the statement following the 'THEN' symbol must be a simple statement, i.e. it may be neither if - nor

while- nor for-statement.

Examples:

```
IF X≠0 THEN Y:=1/X
IF A[1]='A' THEN B:=A ELSE C:=A
```

9.7 While statements

A while statement (cf. production S10) specifies the repeated execution of the statement following the 'DO' symbol depending on the value of the condition. The condition is evaluated before the statement execution. If it is false, the while statement is finished.

Example:

```
WHILE  I>2  DO BEGIN  M[I]:=M[I-1]; I:=I-1 END
```

9.8 For statements

A for statement (cf. production S11) specifies a progression of values assigned to the so called control variable and the cyclic execution of the statement following the 'DO' symbol. Each cycle begins with the assignement to the control variable and ends with the statement execution. The initial value assigned is the result of the evaluation od the expression following the ':=' symbol. Then, the value is increased consecutively by 1 until it reaches the value of the expression following the 'TO'symbol.

Example:

```
FOR I:=1 TO J-1  DO M[J+1-I]:=M[J-I]
```

10. Procedures

Procedure declaration contains an identifier called the procedure identifier, a list of formal parameters and a block called the procedure body. The block contains declarations and a compound statement.

```
B. BLOCK=TS, VS, PS, 'BEGIN', STMT_L, 'END';
```

Declarations of types and variables have the form of declaration lists preceded by symbols 'TYPE' and 'VAR' correspondingly.

```
T1. TS='EPS'|
T2.     'TYPE', TYPE_L, ';';
VB1. VS='EPS'|
VB2.    'VAR', VAR_L, ';';
VB3. VAR_L=VAR_DP|
VB4.       VAR_L, ';', VAR_DP;
```

All types, variables and procedures declared within the procedure body are local to the procedure. Values of the local variables are undefined when the procedure body is entered. In addition to the local objects, parameters and global objects are also known within the body provided their identifiers are not redeclared. Global objects are those declared in the procedure environment and textually preceding the procedure declaration.

Since procedures may be declared as local objects, their declarations may be nested.

```
P1. PS='EPS'|
P2.     'PROCEDURE', IDEN, PAR_L, ';', BLOCK, ';', PS;
```

However, there may be no recursion of procedure calls; no procedure may be called within its own body. (Implicit recursion is excluded by the definition of global objects as no two procedures can precede each other textually).

The procedure body may include all kinds of statements except process calls.

Both restrictions, one concerning recursion, the other one - process calls, are due to the intended lack of any dynamic memory allocation routines which, otherwise, would be the necessary part of a run time support system. Recursive procedures obviously require memory allocation for their local variables whenever they are being called. In this respect, procedures that have process calls in their bodies are similar to the recursive ones. A process call always means that the calling process is stopped until it is resumed or started again from its beginning. In the latter case it may happen that the same procedure is called again. So, new memory should be allocated to its local variables since its previous call has not been finished yet.

The procedure parameter list may consist of several parameter groups, each of which is a list of parameter identifiers and a type identifier.

```
P4. PAR L='EPS'|
P5.       '(', PG_L, ')' ;
P6. PG_L=PG|
P7.      PG_L, ';', PG;
P8. PG=IDEN_L, ':', IDEN;
```

There is no explicit specification whether a parameter is of value or variable kind. Nevertheless, such a qualification is established in the compilation process; parameters which are being assigned to in the procedure body get the variable status while others have the value status. Value parameters can be considered as local variables which initial values are results of evaluation of the corresponding arguments. Variable parameters stand for those arguments of the procedure call which are variables. Parameters are local to the procedure, i.e. their scope is the procedure declaration. Types of arguments in the procedure call must be compatible with the parameter types.

Examples:

Suppose there is an array QUEUE of links, representing the first-come-first-served queue of process links, declared with N as its maximum index. FIRST is a variable in which the queue head index is kept and LAST - its tail index. Initially, when the queue is empty, both indices are set to 1. In order to operate the queue the following procedures may be used:

```
PROCEDURE PUT (L:LINK);
BEGIN QUEUE[LAST] :=L; LAST:=LAST+1-LAST/N*N END;
PROCEDURE GET(L:LINK);
BEGIN L:=QUEUE[FIRST]; FIRST:=FIRST+1-FIRST/N*N END;
```

10.1 Standard procedures

The following standard procedures are predefined:

HOLD(L:LINK) - assigns to L a process link; in the process called by another process it is the link of the calling one, in the process activated by an interrupt it is the link of the interrupted process,

RESUME(L:LINK) - stops the current process and resumes the one whose link is the value of L,

IN(D, L, S: INTEGER) - initiates transmission of S characters from the device no. D to the memory starting from the address L,

OUT(D, L, S: INTEGER)- initiates transmission of S characters to the device no. D from the memory starting from the address L,

BIN(A: ARRAYOFCHAR; I, N: INTEGER) - converts a string of characters contained in A under the index I into a numer and assigns the numer to N,

DEC(N,I:INTEGER; A:ARRAYOFCHAR) - converts the value of N into a string of characters and assigns the string to A beginning from the index I.

In both procedures, BIN and DEC, ARRAYOFCHAR denotes a one-dimensional array of characters. In other words, A stands for a variable declared as such an array. The first character of the string is pointed to by I. After the procedure is finished I points to a character which is next to the last one of the string concerned. The value 1 of the array index is considered next to its maximum value specified in the array declaration. Hence, strings are placed as though the array were a circular data structure.

The BIN procedure acts as follows:

- it ignores consecutive characters until a sign followed by a digit or a digit is encountered,
- it accepts consecutive characters until a character other than a digit is encountered or until 5 digits have been read,
- it converts the resulted string into the number.

The DEC procedure converts the given number into the string consisting of 6 characters - spaces for leading zeros, then another space or '-' depending whether the number is positive or negative, and the number's significant digits.

11. Programs

Programs can be understood as process types or patterns. According to a program several processes may be declared analogously to variables being declared according to their types. Unlike procedures, programs can not be called. They may only be referenced in process declarations (cf. section 12).

A program declaration consists of its prefix, heading and block.

```
PR. PROG=PREF, 'PROGRAM', IDEN, PPAR_L, PPAR_T, ';', BLOCK, '.';
```

The prefix is to allow declarations of certain types prior to their use in the program heading.

```
PF1. PREF='EPS'|
PF2.       'PREFIX', TS;
```

The heading contains the program identifier, its parameter list and parameter record. The parameter list consists of process names that can be referenced in process calls within the program block.

```
PP1. PPAR_L='EPS'|
PP2.        '(', IDEN_L, ')';
```

The parameter record corresponds to argument records accompanying calls of the process declared according to the program (cf. section 9.2.2).

```
PT1. PPAR_T=' EPS'|
PT2.        'PARAMETER','RECORD', FIELD_L, VARIANT, 'END';
```

The parameter record does not differ syntactically from the regular record (cf. section 5.3). Semantically, however, its field types are restricted to the following ones: CHAR, INTEGER, LINK, POINTERVAL and pointer to any type. (Only such types are concerned when corresponding argument records are defined; cf. section 5.3.) Identifiers of the parameter record fields can be referenced in the program block as they were variables. In other words, the parameter record has no identifier for itself; its field identifiers are direct designators.

The scope of identifiers introduced in the prefix, parameter list and parameter record is the program text. The scope of the program identifier is the process structure (cf. section 12).

Example:

```
PREFIX TYPE MESSAGE=ARRAY[20] OF CHAR;
PROGRAM TRANSMITTER (S) PARAMETER RECORD
        OPERATION: INTEGER; PAR: ↑MESSAGE END;
(* TRANSMITTER GETS MESSAGES FROM A PROCESS THAT CALLS OPERATION 1 AND SENDS THEM TO A PROCESS
   THAT CALLS OPERATION 2. OPERATION 3 IS TO INITIALIZE TRANSMITTER. S STAND FOR SCHEDULER
   WHICH OPERATIONS ARE THE FOLLOWING: 1 - IT DELAYS TRANSMITTER AT THE S(1,L) CALL UNITIL
   THE S(2,L ...) CALL TAKES PLACE, 2 - IT RESUMES TRANSMITTER DELAYED AT THE S (1,L)CALL AND
   THE PROCESS BY WHICH TRANSMITTER HAS BEEN CALLED. PROCESSES CALLING TRANSMITTER MAY DO IT
   IN ANY ORDER. *)
VAR M:MESSAGE;
    I:INTEGER; (* 1 - MESSAGE RECEIVED, 2 - MESSAGE SENT *)
    SENDER, RECEIVER:LINK;
BEGIN CASE OPERATION OF
1: BEGIN HOLD(SENDER); IF I=1 THEN S(1, SENDER);
   I:=1; M:=PAR↑; S(2, RECEIVER, SENDER)END;
2: BEGIN HOLD(RECEIVER); IF I=0 THEN S(1, RECEIVER);
   I:=0; PAR↑:=M; S(2,SENDER, RECEIVER) END;
3: BEGIN I:=0; HOLD(RECEIVER); SENDER:=RECEIVER; RESUME(SENDER) END END.
```

12. Processes

Processes are created, identified and incorporated into the process structure by their declarations. Each of such processes is, in fact, an instance of the program referenced in its declaration. The declaration specifies also whether the process is interruptable or not. Identifiers of processes that are instances of the same program may be listed in one declaration group.

```
PD1. PROC_DECL_G='PROCESS', SPECIFICATION, PROC_DECL_L;
PD2. SPECIFICATION='EPS'|
PD3.                'NONINTERRUPTABLE';
PD4. PROC_DECL_L=PROC_DECL|
PD5.             PROC_DECL_L, ';', PROC_DECL;
PD6. PROC_DECL=IDEN_L, ';', IDEN, PROC_L;
```

Symbols following the colon are the program name and the list of process names which replace the program parameters (cf. section 11).

```
PD7. PROC_L='EPS'|
PD8.        '(', IDEN_L, ')';
```

Identifiers of processes are not to be chosen freely; the list of identifiers allowed for process names is the following:

```
P1 |
P2 |   for processes that can be called by other processes,
...|
Pn |
I1 |
I2 |   for processes activated by interrupts,
...|
Im |
```

where n and m depend on a particular hardware organization. Moreover, each of identifiers beginning with the letter I is associated with a specific hardware interrupt.

Every process declared in such a way can be active or passive. It becomes active when it is called or resumed by another process or when the interrupt with which the process name is associated occurs. It becomes passive when it calls or resumes another process or when it is interrupted. When the process is being called or resumed while active, the call or resumption is delayed until it becomes passive. So, mutual exclusion of calls of the same process is imposed automatically. However, a process activated by an interrupt can be itself interrupted and activated again unless it is specified NONINTERRUPTABLE. Hence, the automatic exclusion of interrupts is imposed only with respect to processes so specified.

In contrast to procedures, processes retain values of their local data during their passive periods. In other words, values of all variables declared in the corresponding program block are preserved between consecutive active periods of the process. This, of course, does not concern variables which are arguments of process calls occuring in the program block. Values of such variables can be changed by the called processes.

Example (cf. section 11):
PROCESS P6, P7, P8: TRANSMITTER(P1)

13. Process structures

A process structure is created by a series of process declarations preceded by program declarations and followed by an initial process call. (In the case of multiprocessor hardware there have to be as many initial process calls as processors).

```
PS1. STRUCTURE=PROG_L, PROC_DECL_S, '.', INIT_PROC_CALL;
PS2. PROG_L=PROG|
PS3.        PROG_L, PROG;
PS4. PROC_DECL_S=PROC_DECL_G|
```

```
PS5.            PROC_DECL_S, ';' , PROC_DECL_G;
PS6. INIT_PROC_CALL=IDEN, '(', IDEN_L, ')';
```

The structure forms a hierarchy of processes that can call and resume one another. Initially, they are all passive except the one referred to in the initial process call. Arguments of the call are not expressions but names of other processes which in the corresponding argument record are represented by their links. Values of the links correspond to starting points of the processes. So, in the program according to which the initial process is declared there may be resumptions of the processes listed in the initial process call. Each of such resumptions activates the given process from its beginning.

14. Syntax summary

```
PS1. STRUCTURE=PROG_L, PROC_DECL_S, '.', INIT_PROC_CALL;                    13
PS2. PROG_L=PROG|                                                           13
PS3.        PROG_L, PROG;                                                   13
PS4. PROC_DECL_S=PROC_DECL_G|                                               13
PS5.            PROC_DECL_S, ';', PROC_DECL_G;                              13
PS6. INIT_PROC_CALL=IDEN, '(', IDEN_L, ')';                                 13
PD1. PROC_DECL_G='PROCESS', SPECIFICATION, PROC_DECL_L;                     12
PD2. SPECIFICATION='EPS'|                                                   12
PD3.               'NONINTERRUPTABLE';                                      12
PD4. PROC_DECL_L=PROC_DECL|                                                 12
PD5.            PROC_DECL_L, ';', PROC_DECL;                                12
PD6. PROC_DECL=IDEN_L, ';', IDEN, PROC_L;                                   12
PD7. PROC_L='EPS'|                                                          12
PD8.        '(', IDEN L, ')';                                               12
PM. PROG=PREF, 'PROGRAM', IDEN, PPAR_L, PPAR_T, ';', BLOCK, '.';            11
PF1. PREF='EPS'|                                                            11
PF2.      'PREFIX', TS;                                                     11
B. BLOCK=TS, VS, PS, 'BEGIN', STMT_L, 'END';                                10
PP1. PPAR_L='EPS'|                                                          11
PP2.        '(', IDEN_L, ')';                                               11
PT1. PPAR_T='EPS'|                                                          11
PT2.        'PARAMETER', 'RECORD', FIELD_L, VARIANT, 'END';                 11
T1. TS='EPS'|                                                               10
T2.    'TYPE', TYPE L, ';';                                                 10
T3. TYPE_L=IDEN, '=', TYPE_D|                                               5.5
T4.       TYPE L, ';', IDEN, '=', TYPE_D;                                   5.5
T5. TYPE_D=IDEN |                                                           5
T6.        'ARRAY', '[', NUMB_L, ']', 'OF', TYPE_D|                         5
T7.        'RECORD', FIELD_L, VARIANT, 'END'|                               5
T8.        '(' , IDEN;                                                      5
F1.FIELD_L='EPS'|                                                           5.3
F2.        FIELD |                                                          5.3
F3.        FIELD, ';', FIELD_L;                                             5.3
F4. FIELD=IDEN_L, ':', TYPE_D;                                              5.3
V1. VARIANT='EPS'|                                                          5.3
V2.         'CASE', IDEN, ':', IDEN, 'OF', VARIANT_L;                       5.3
V3. VARIANT_L=CONST_L, ':', '(', FIELD_L, VARIANT, ')'|                     5.3
V4.           VARIANT_L, ';', CONST_L, ':', '(', FIELD_L, VARIANT, ')';     5.3
```

```
L1. NUMB_L=NUMB |                                                    5.2
L2.        NUMB_L, ',', NUMB;                                        5.2
L3. IDEN_L=IDEN |                                                    5.3
L4.        IDEN_L, ',', IDEN;                                        5.3
L5. CONST=NUMB |                                                     5.3
L6.        'STRING';                                                 5.3
L7. CONST_L=CONST |                                                  5.3
L8.         CONST_L, ',', CONST                                      5.3
VB1. VS='EPS' |                                                      10
VB2.     'VAR', VAR_L, ';';                                          10
VB3. VAR_L=VAR_DF |                                                  10
VB4.        VAR_L, ';', VAR_DF;                                      10
VB5. VAR_DF=IDEN_L, ':', TYPE_D;                                     6
P1. PS='EPS' |                                                       10
P2.     'PROCEDURE', IDEN, PAR_L, ';', BLOCK, ';', PS;               10
P4. PAR_L='EPS' |                                                    10
P5.        '(', PG_L, ')';                                           10
P6. PG_L=PG |                                                        10
P7.       PG_L, ';', PG;                                             10
P8. PG=IDEN_L, ':', IDEN;                                            10
S1. S_STMT=VAR_D, ':=', EXPR |                                       9
S2.         IDEN, ARGS |                                             9
S4.         'BEGIN', STMT_L, 'END' |                                 9
S5.         'REPEAT', STMT_L, 'UNTIL', COND |                        9
S6.         'CASE', EXPR, 'OF', CASE_L, 'END' |                      9
S7.         'EPS';                                                   9
S8. STMT=S_STMT |                                                    9
S9.       'IF', COND, 'THEN', S_STMT, IF_TALL |                      9
S10.      'WHILE', COND, 'DO', STMT |                                9
S11.      'POR', IDEN ':=', EXPR, 'TO', EXPR, 'DO', STMT;            9
S12. IF_TAIL='EPS' |                                                 9.6
S13.      'ELSE', STMT;                                              9.6
S14. CASE_L=CASE_D |                                                 9.5
S15.         CASE_L, ';', CASE_D;                                    9.5
S16. CASE_D=CONST_L, ':', STMT;                                      9.5
S17. STMT_L=STMT |                                                   9.3
S18.         STMT, ';', STMT_L;                                      9.3
A1. ARGS='EPS' |                                                     9.2
A2.       '(', EXPR_L, ')';                                          9.2
A3. EXPR_L=EXPR |                                                    7.1
A4.         EXPR_L, ',', EXPR;                                       7.1
VD1. VAR_D=IDEN |                                                    7.1
VD2.         VAR_D, '[', EXPR_L, ']' |                               7.1
VD3.         VAR_D, '.', IDEN |                                      7.1
VD4.         VAR_D, '↑';                                             7.1
E1. EXPR=EXPR, AD_OP, TERM |                                         7.2
E2.       AD_OP, TERM |                                              7.2
E3.       TERM;                                                      7.2
TR1. TERM=TERM, MLT_OP, FACTOR |                                     7.2
TR2.       FACTOR;                                                   7.2
FR1. FACTOR=CONST |                                                  7.2
FR2.         VAR_D |                                                 7.2
FR3.         '(', EXPR, ')';                                         7.2
```

```
CN1. COND=COND, '|', COND_T |                                    8
CN2.      COND_T;                                                8
CN3. COND_T=COND_T, '&', COND_F |                                8
CN4.        COND_F;                                              8
CN5. COND_F='ODD', EXPR |                                        8
CN6.        EXPR, '=', EXPR |                                    8
CN7.        EXPR, REL_OP, EXPR;                                  8
```

15. List of standard identifiers

BASE	5.5
BIN	10.1
CHAR	5.1
DEC	10.1
HOLD	10.1
IN	10.1
INTEGER	5.1
LINK	5.5
LOC	5.4
OUT	10.1
POINTERVAL	5.5
RESUME	10.1
RETL	5.5
SIZE	5.4
STATUS	5.5

STRESZCZENIE

Opracowanie opisuje język programowania do implementacji systemów operacyjnych. Język PHIL 5, opracowany według metodologii systemów operacyjnych, zbudowany jest jako struktury hierarchiczne procesów.

СОДЕРЖАНИЕ

В настоящей работе рассматривается язык программирования для разработки операционных систем. Язык PHIL 5 построен с учетом методологии разработки операционных систем, как иерархических структур процессов.

OFERTY

Zestaw kreślący XYNETICS 1200 Hewlett Packard

Instytut Maszyn Matematycznych dysponuje nowoczesnym zestawem kreślącym XYNETICS 1200 Hewlett Packard, którego charakterystyka zamieszczona jest niżej.

Instytut Maszyn Matematycznych może podjąć się wykonania usług eksploatacyjno-obliczeniowych i programistycznych, przy realizacji których niezbędna jest graficzna forma wyników, w następujących dziedzinach:

- przemysł maszynowy (projektowanie części maszyn),
- przemysł lotniczy (dokumentacje warstwicowe, kontrolno-pomiarowe itp.),
- przemysł okrętowy (projektowanie rozkrojów blach),
- architektura i budownictwo (projekty konstrukcyjne),
- urbanistyka (plany przestrzennego zagospodarowania terenu),
- gospodarka komunalna (planowanie sieci komunikacyjnych),
- kartografia (mapy, plany, poziomice),
- zarządzanie (planowanie sieci działań, rysowanie zestawień statystycznych itp.),
- elektronika (projektowanie obwodów).

Instytut dysponuje także komputerami: IBM-370 i R-32

Na emc IBM i R-32 posiadamy oprócz standardowego oprogramowania - uruchomiony pakiet NARVIK (APT) służący do generowania programów dla obrabiarek sterowanych numerycznie (OSN).

Dla zestawu kreślącego posiadamy uruchomione na emc R-32 i IBM 370 pakiety funkcji kreślących umożliwiające graficzne przedstawienie wyników programów napisanych w języku FORTRAN IV oraz postprocesor do systemu numerycznego sterowania obrabiarkami umożliwiający rysowanie drogi narzędzia skrawającego.

W zakresie oprogramowania możemy ponadto podjąć się wykonania nietypowych pakietów programowych i postprocesorów w zależności od indywidualnych potrzeb użytkowników.

Umożliwiamy dogodną eksploatację opisanych urządzeń kreślących, jak również maszyn bazowych. Ponadto możemy udostępnić i zapewnić obsługę zestawu kreślącego do celów naukowo-badawczych, dydaktycznych i in.

Zainteresowanych naszymi usługami prosimy o podanie:

- klasy i zakresu zagadnienia, które interesuje użytkownika
- potrzebnego oprogramowania
- stopnia przewidywanego wykorzystywania naszych usług

Zgłoszenia prosimy kierować na adres:

Instytut Maszyn Matematycznych
Krzywickiego 34, 02-078 Warszawa
Kierownik Działu Planowania i Koordynacji Prac Naukowo-Badawczych
mgr inż. Jan NESTERUK
telefon 28-33-36
telex 813517

CHARAKTERYSTYKA TECHNICZNO-EKSPLOATACYJNA ZESTAWU KREŚLĄCEGO XYNETICS 1200 HEWLETT PACKARD

Zestaw kreślący pracuje samodzielnie w stosunku do maszyny bazowej (na której eksploatowane jest oprogramowanie bazowe), tzn. w systemie off-line. Maszynami bazowymi są komputery Jednolitego Systemu serii RIAD oraz IBM.

Dane do rysowania są przenoszone z komputera na zestaw kreślący za pomocą taśmy magnetycznej (taśma typu SL, gęstość 800 BPI).

Ploter XYNETICS 1200

Dane techniczne: stół poziomy
sterowanie ciągłe w dwu osiach
obszar rysowania 147 x 222 cm
prędkość rysowania max. 800 kroków elementarnych (ok 50 cm/s)
dokładność rysowania 0.005 cala (ok.0.127 mm)
powtarzalność rysowania 0.001 cala (ok.0.025 mm) w całym obszarze rysowania
liczba piór w głowicy - 4 (kolory)

Rysunki mogą być wykonywane na kalce technicznej, papierze milimetrowym, papierze białym, filmie warstwowym, foliach itp.

Uwaga: Zainteresowanych bliższymi szczegółami zachęcamy do przeczytania opracowania zamieszczonego w [illegible] 3-4/1982 Biuletynu Informacyjnego Nauki i Techniki Komputerowo - Berthold A., Topolski S.: Uwagi na temat zastosowania ploterów ze szczególnym uwzględnieniem zestawu kreślącego XYNETICS 1200 Hewlett Packard - zainstalowanego w Instytucie Maszyn Matematycznych.

GRAFIKA KOMPUTEROWA

Język **PSG** jest proceduralnym językiem, przeznaczonym do programowania graficznych urządzeń wejściowych komputerów. Został opracowany w Pracowni Grafiki Komputerowej Instytutu Maszyn Matematycznych w Warszawie i zaimplementowany na minikomputerze MERA-400 z systemem operacyjnym SOM-3.

PSG jest częścią składową tzw. bazowego systemu graficznego na minikomputerze MERA-400 (przedstawionego na rysunku), z dołączonymi on-line urządzeniami: rastrowym monitorem graficznym MERA-7954 i ploterem Benson-1220 oraz stacją pamięci taśmowej PT-305.

Język **PSG** tworzą następujące grupy procedur:

- procedury definiowania podstawowych elementów geometrycznych;
- procedury transformacji podstawowych elementów geometrycznych;

  (Za pomocą procedur definiujących i transformacyjnych można określać, tworzyć i przetwarzać elementy geometryczne, takie jak: punkty, odcinki, okręgi, elipsy i ich łuki, krzywe algebraiczne 2- i n-stopnia, znaki alfanumeryczne i specjalne itp.)
- procedury generacyjne zapisujące w Zbiorze Pośrednim Danych Graficznych ZPDG wykreowane elementy geometryczne;
- procedury obsługi zbioru ZPDG pozwalające na sterowanie informacją zapisaną w zbiorze ZPDG;
- procedury postprocesorów pobierające ze zbioru ZPDG odpowiednią porcję elementów geometrycznych i przesyłające ją do określonego urządzenia graficznego (monitora, plotera).

Wyświetlenie obrazu na monitorze wymaga wykonania następujących czynności:

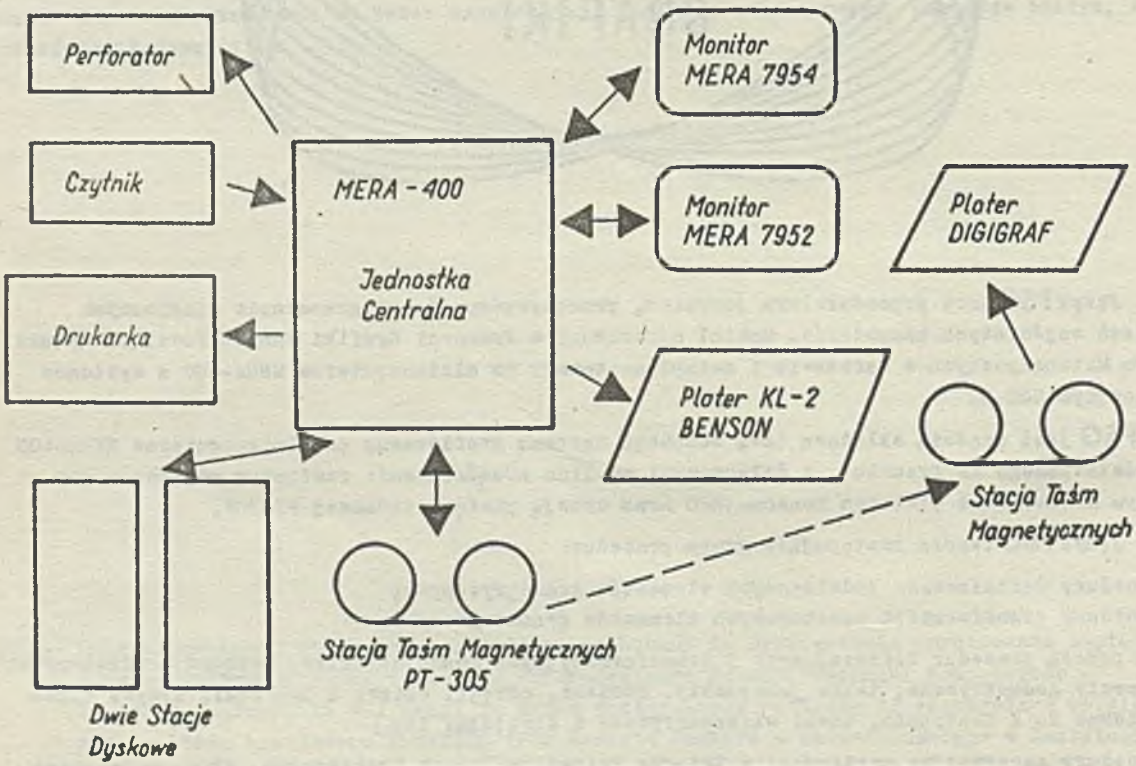
- analizowania rysunku, jego podziału na prostw elementy geometryczne i określenia tych elementów za pomocą procedur definiujących i transformacyjnych;
- zapisania elementów obrazu w Zbiorze Pośrednim Danych Graficznych (ZPDG) za pomocą procedur obsługi zbioru ZPDG oraz procedur generacyjnych;
- przesłania do monitora lub plotera danych graficznych ze zbioru ZPDG przez wywołanie odpowiedniego postprocesora.

Przedstawiony na rysunku schemat przepływu informacji w systemie graficznym jest podstawowym schematem działania użytkownika w ramach systemu. System dopuszcza kombinacje tego procesu, np. najpierw zapisanie poroji informacji w zbiorze ZPDG, a następnie wybranie odpowiednich z nich i ich wyświetlenie. Zasadnicze idea pozostaje jednak bez zmian.

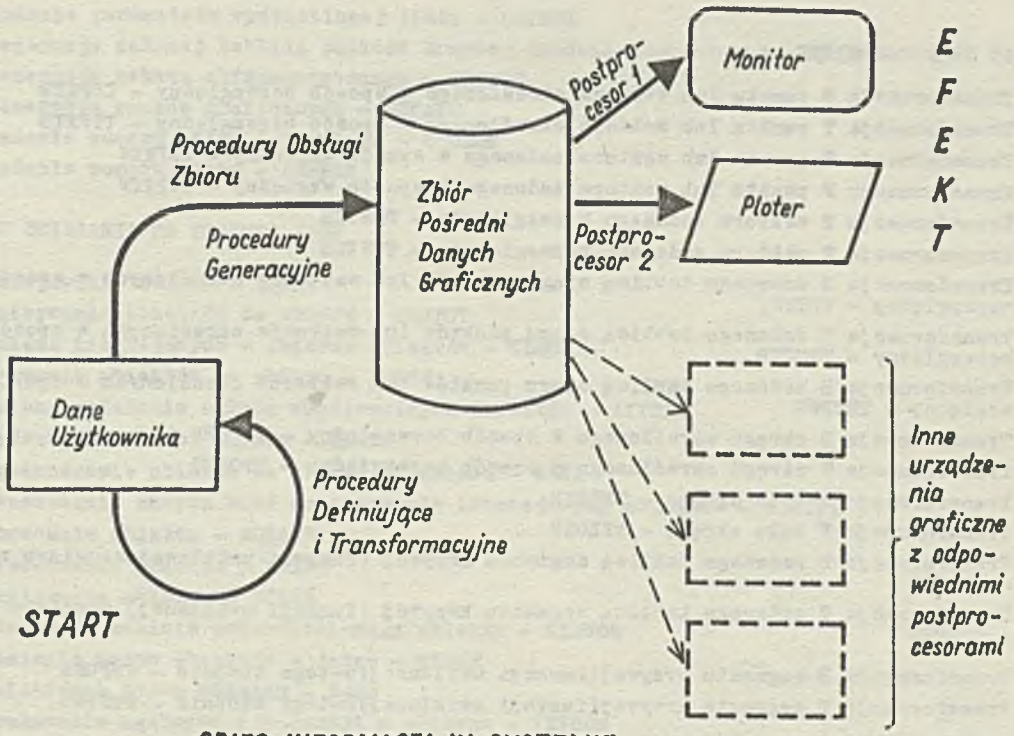
Oferujemy trzy odmiany języka PSG:

- uproszczoną U
- standardową S
- rozszerzoną R

różniące się możliwościami (liczbą i typem procedur) oraz ceną.



KONFIGURACJA BAZOWEGO SYSTEMU GRAFICZNEGO



OBIEG INFORMACJI W SYSTEMIE

PROCEDURY JĘZYKA PSG

PROCEDURY DEFINIUJĄCE

1. Definiowanie punktu za pomocą dwóch przecinających się prostych (wektorów) – DPKTDW
2. Definiowanie punktu za pomocą okręgu przecinającego się z prostą (wektorem) – DPKTOW
3. Definiowanie punktu za pomocą dwóch przecinających się okręgów – DPKTDO
4. Definiowanie punktu leżącego na okręgu, wyznaczonego przez promień tworzący kąt alfa z osią x – DPKTOK
5. Definiowanie punktu za pomocą krzywej 2-stopnia i przecinającej ją prostej (wektora) – DPKTSW
6. Definiowanie prostej (wektora) przechodzącej przez punkt i stycznej do okręgu – DWEKPO
7. Definiowanie prostej (wektora) stycznej do dwóch okręgów – DWEKDO
8. Definiowanie prostej (wektora) przechodzącej przez punkt i tworzącej kąt alfa z osią x – DWEKPX
9. Definiowanie prostej(wektora) przechodzącej przez punkt i tworzącej kąt alfa z zadaną prostą wektorem – DWEKPW
10. Definiowanie prostej (wektora) równoległej do danej prostej i odległej od niej od – DWEKRD
11. Definiowanie okręgu o danym środku, stycznego do prostej (wektora) – DOKRSW
12. Definiowanie okręgu przechodzącego przez trzy punkty – DOKRTP
13. Definiowanie okręgu o zadanym środku, stycznego do okręgu zadanego – DOKRSO
14. Definiowanie okręgu o danym promieniu, stycznego do dwóch prostych (wektorów)– DOKRDW
15. Definiowanie łuku okręgu stycznego do dwóch prostych i zawartego między nimi (narożna) – DLOKRN
16. Definiowanie krzywej (funkcji uwikłanej) drugiego stopnia przechodzącej przez pięć punktów– DFUDST
17. Definiowanie punktu leżącego na prostej (wektorze) odległego od punktu początkowego od – DPTSPD
18. Definiowanie łuku okręgu określonego punktami początkowym i końcowym oraz strzałką – DLOKRS

PROCEDURY TRANSFORMUJĄCE

1. Transformacja B punktu lub wektora określonego w sposób bezwzględny - TBPKTB
2. Transformacja T punktu lub wektora określonego w sposób bezwzględny - TTPKTB
3. Transformacja B punktu lub wektora zadanego w sposób względny - TBPKTW
4. Transformacja T punktu lub wektora zadanego w sposób względny - TTPKTW
5. Transformacja T wektora zadanego bezwzględnie - TBWEKB
6. Transformacja T wektora zadanego bezwzględnie - TTWEKB
7. Transformacja B zadanego tablicą ciągu punktów lub wektorów określonych w sposób bezwzględny - TBTPWB
8. Transformacja T zadanego tablicą ciągu punktów lub wektorów określonych w sposób bezwzględny - TTTPWB

S

9. Transformacja B zadanego tablicą ciągu punktów lub wektorów określonych w sposób względny - TBTPWW
10. Transformacja B okręgu określonego w sposób bezwzględny - TBOKRB
11. Transformacja T okręgu określonego w sposób bezwzględny - TTOKRB
12. Transformacja B łuku okręgu - TBLOKR
13. Transformacja T łuku okręgu - TTLOKR
14. Transformacja B zadanego tablicą segmentu krzywej (funkcji uwikłanej)drugiego stopnia - TBFUDS
15. Transformacja T zadanego tablicą segmentu krzywej (funkcji uwikłanej) drugiego stopnia - TTFUDS

R

16. Transformacja B segmentu krzywej(funkcji uwikłanej)n-tego stopnia - TBFUNS
17. Transformacja T segmentu krzywej(funkcji uwikłanej)n-tego stopnia - TTFUNS

PROCEDURY GENERACYJNE

U 1. Generacja punktu zadanego w sposób bezwzględny - GPNKTB

S 2. Generacja punktu określonego w sposób względny - GPNKTW

U 3. Generacja zadanego tablicą ciągu punktów określonych bezwzględnie - GTABPB

S 4. Generacja zadanego tablicą ciągu punktów określonych względnie - GTABPW

U 5. Generacja odcinka (wektora) określonego w sposób bezwzględny - GWEKTB

S

6. Generacja odcinka (wektora) określonego względnie - GWEKTW
7. Generacja zadanego tablicą ciągu odcinków (łamanej) określonych względnie - GTABŁW
8. Generacja zadanego tablicą ciągu odcinków (łamanej)określonych bezwzględnie - GTABŁB

U 9. Generacja zadanego tablicą zbioru wektorów określonych bezwzględnie - GTABWB

R 10. Generacja krzywej gładkiej przechodzącej przez zadane tablicą punkty, określone bezwzględnie - GTABKG

U

11. Generacja okręgu określonego bezwzględnie - GOKROB
12. Generacja łuku okręgu określonego bezwzględnie - GLOKRB

S

13. Generacja elipsy - GELIPS
14. Generacja łuku elipsy - GLELIP
15. Generacja segmentu krzywej (funkcji uwikłanej) drugiego stopnia - GFUDST
16. Generacja pola trójkąta - GPOLET
17. Generacja pola kołowego - GPOLEK
18. Generacja pola elipsy - GPOLEL
19. Generacja pola krzywej drugiego stopnia - GPOLKD
20. Generacja segmentu krzywej algebraicznej(funkcji uwikłanej)n-tego stopnia - GFUNST

U 21. Generacja osi układu współrzędnych - GUKWSP

R 22. Generacja logarytmicznego układu współrzędnych - GLUKWS

S 23. Generacja siatki prostokątnej - GSIATP

R 24. Zadanie parametrów wyświetlonej linii - GLINOS
R 25. Generacja zadanej tablicą punktów krzywej średnio-kwadratowej - GTABKS
U 26. Generacja tekstu alfanumerycznego - GTEKST
U 27. Generacja znaków graficznych - GZNAKG
R 28. Zadanie numeru pióra lub koloru - GCOLOR
R 29. Zadanie numeru bloku - GNRBLK

PROCEDURY DZIAŁANIA NA ZBIORZE ZPDG

U 1. Inicjowanie zbioru - OPEN
U 2. Wpisywanie obiektów do zbioru - OBJECT
U 3. Zmiana istniejących w zbiorze obiektów - CLOSE
U 4. Usuwanie obiektów ze zbioru - PURGE
U 5. Zmiana położenia układu współrzędnych obiektów - AXISES
R 6. Wydruk zawartości zbioru - LIST
S 7. Przenoszenie obiektu ze zbioru do zbioru - SEND
R 8. Generowanie nowych bądź rozszerzenie istniejących obiektów - OBJUNI
S 9. Obracanie obiektu - ROTATE
S 10. Przesuwanie obiektu - TRANS
S 11. Skalowanie obiektu - SCALE
R 12. Tworzenie odbicia zwierciadlanego obiektu - MIRROR
R 13. Scalanie dwóch obiektów w jeden - MERGE
R 14. Zmienienie nazwy obiektu - NAME
S 15. Drukowanie ogólnych informacji o zbiorze - INFROM

PROCEDURY POSTPROCESORA MONITORA MERA 7954, KREŚLAKA BENSON 122, KREŚLAKA KL-2, DIGIGRAF 1612, NE 240

U 1. Wybieranie obiektu do wyświetlania przez podanie jego nazwy - PMR1, PPB1, PPK1
U 2. Wybieranie ciągu obiektów do wyświetlania przez podanie ich numerów - PMR2, PPB2, PPK2.

Pracownia Grafiki Komputerowej IMM przyjmie do realizacji zlecenia na sprzętowe i programowe włączenie monitora MERA 7954 (lub innego urządzenia graficznego) do innych niż MERA-400 systemów komputerowych.

Szczegółowych informacji udziela:

Instytut Maszyn Matematycznych
Pracownia Grafiki Komputerowej
ul. Krzywickiego 34
00-278 Warszawa
tel. 21-84-41 w. 396, 428, 413
teleks 813517

„TECHNIKA-TEKST" 0/748/3005/A

Informacja o cenach i warunkach prenumeraty na 1984 r.
- dla czasopism Instytutu Maszyn Matematycznych

● Cena prenumeraty rocznej

Techniki Komputerowe - Biuletyn Informacyjny	1560.-	dwum.
Przegląd Dokumentacyjny - Nauki i Techniki Komputerowe	1260.-	dwum.
Informacja Ekspresowa - Nauki i Techniki Komputerowe	2400.-	mies.
Prace naukowo-badawcze Instytutu Maszyn Matematycznych	660.-	3x w roku

● Warunki prenumeraty

1/ dla osób prawnych - instytucji i zakładów pracy:
- instytucje i zakłady pracy zlokalizowane w miastach wojewódzkich i pozostałych miastach, w których znajdują się siedziby oddziałów RSW "Prasa-Książka-Ruch" zamawiają prenumeratę w tych oddziałach;
- instytucje i zakłady pracy zlokalizowane w miejscowościach, gdzie nie ma oddziałów RSW "Prasa-Książka-Ruch" i na terenach wiejskich opłacają prenumeratę w urzędach pocztowych i u doręczycieli;

2/ dla osób fizycznych - prenumeratorów indywidualnych:
- osoby fizyczne zamieszkałe na wsi i w miejscowościach, gdzie nie ma oddziałów RSW "Prasa-Książka-Ruch" opłacają prenumeratę w urzędach pocztowych i u doręczycieli;
- osoby fizyczne zamieszkałe w miastach - siedzibach oddziałów RSW "Prasa-Książka-Ruch" opłacają prenumeratę wyłącznie w urzędach pocztowych nadawczo-oddawczych właściwych dla miejsca zamieszkania prenumeratora. Wpłaty dokonują używając "blankietu wpłaty" na rachunek bankowy miejscowego oddziału RSW "Prasa-Książka-Ruch";

3/ Prenumeratę ze zleceniem wysyłki za granicę przyjmuje RSW "Prasa-Książka-Ruch", Centrala Kolportażu Prasy i Wydawnictw, ul. Towarowa 28, 00-958 Warszawa, konto NBP XV Oddział w Warszawie nr 1153-201045-139-11. Prenumerata ze zleceniem wysyłki za granicę pocztą zwykłą jest droższa od prenumeraty krajowej o 50% dla zleceniodawców indywidualnych i o 100% dla zlecających instytucji i zakładów pracy.

● Terminy przyjmowania prenumeraty na kraj i za granicę:

- do dnia 10 listopada na I kwartał, I półrocze roku następnego oraz na cały rok następny,
- do dnia 1-każdego miesiąca poprzedzającego okres prenumeraty roku bieżącego.

Zamówienia na prenumeratę "Prac naukowo-badawczych Instytutu Maszyn Matematycznych przyjmuje Dział Sprzedaży Wysyłkowej Ośrodka Rozpowszechniania Wydawnictw Naukowych PAN, Warszawa, Pałac Kultury i Nauki, tel. 20-0 tel.20-02-11 w.2516. Egzemplarze pojedyncze Prac są do nabycia w księgarni ORWN PAN, Warszawa, Pałac Kultury i Nauki, tel.20-02-11 w.2105.

Cena 220 zł