

Two high-level tools
for the development of the software
on mini-computers

by

Louis Julien Lucien Doucet
Ingénieur Civil des Mines

University of London

January 1974

A B S T R A C T

Two high-level tools for the implementation of the software on mini-computers -- the System Programming Language and the Syntax Description Language -- are presented.

A System Programming Language is the highest possible mode of expression compatible with software writing and system programming applications. The general characteristics of such a language and its interface with the hardware are discussed. The implementation of the CTL Modular One System Programming Language is presented as an example. The problem of portability of the resulting software is analysed and solutions are proposed.

The Syntax Description Language is a method of description of the syntax of programming languages in terms of transition diagrams. The use of this method for the syntactical analysis and the compilation of various types of languages is discussed and examples are given. The Syntax Description Language may be used as a preprocessor to the System Programming Language. The production of the software of programmable terminals is given as an example of application of this integrated system.

à mes parents,

" Un travail réglé
et des victoires après des victoires,
voilà sans doute
la recette du bonheur. "

ALAIN

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	12
FOREWORD	13

INTRODUCTION

Chapter I - JUSTIFICATIONS AND METHODOLOGY	16
Chapter II - CHARACTERISATION OF THE MINI-COMPUTERS	
1. Hardware	19
2. Types of use	21
3. Software	24

PART I - THE SYSTEM PROGRAMMING LANGUAGE

Chapter III - SYSTEM PROGRAMMING LANGUAGES	
1. General philosophy of the system programming languages	
1.1. language level	29
1.2. self-documentation	31
1.3. programming structures	34
1.4. data types and structures	37
1.5. machine interface	38
1.6. debugging facilities	40

2. The myth of external portability	
2.1. external portability	40
2.2. L.S.D.: an externally portable language	
2.2.1. aims of L.S.D.	42
2.2.2. the cost of macro-processing	42
2.2.3. level of the language	43
2.2.4. the cost of machine independence	44
2.2.5. other points	44
2.3. an alternative	45
3. The internal portability	
3.1. problem	46
3.2. internal portability	47
3.3. the "onion ring" design	47
3.4. standard interfaces	50

Chapter IV - P. L. MODULAR ONE

1. Basic characteristics and program elements	
1.1. identifiers	52
1.2. program components	
1.2.1. declaratives	52
1.2.2. statements	53
1.2.3. statement labels	54
1.2.4. statement switch prefixes	54

2. Data elements	
2.1. data classes and constants	56
2.2. variables	57
2.3. organisations	
2.3.1. arrays	60
2.3.2. stacks	60
2.3.3. structures	61
2.3.4. programming example	63
3. Expressions and conditions	
3.1. expressions	64
3.2. conditions	65
4. Statements	
4.1. descriptive statements	67
4.2. data movement and computational statements ...	67
4.3. control statements	68
4.4. conditional statements	69
4.5. repetition statements	70
4.6. data allocation statements	70
4.7. machine interface statements	71
4.8. debugging statements	71
5. Program structure and storage allocation	
5.1. program structure	72
5.2. declarations and storage allocation	
5.2.1. common declarations	73
5.2.2. global declarations	73
5.2.3. local declarations	74
5.2.4. argument declarations	76
5.3. controlled allocation	77

Chapter V - THE EXECUTIVE

1. General structure

1.1. characteristics and general philosophy	78
1.2. the S.P.L. context	79
1.3. structure of the Executive	80
1.4. tasks	81
1.5. protection of the tasks and I.O. operations ..	82
1.6. scheduling mechanism	82

2. Scheduling commands

2.1. task control blocks (TCB)	83
2.2. the CREATE A TCB command	86
2.3. the KILL A TCB command	87
2.4. the ACTIVATE AND GO ON command	88
2.5. a programming example	89
2.6. the ACTIVATE AND WAIT command	90
2.7. the RETURN command	92
2.8. the STATUS TASK and SUSPEND commands	92

3. Transput commands

3.1. generalities	93
3.2. Input-Output control blocks (IOCB)	95
3.3. the CREATE AN IOCB command	97
3.4. the KILL AN IOCB command	97
3.5. the TRANSPUT AND GO ON and TRANSPUT AND WAIT commands	98
3.6. the STATUS IO command	98

PART II - THE SYNTAX DESCRIPTION LANGUAGE

Chapter VI - STRUCTURE OF A TRANSLATOR

1. Terminology and notations	100
2. The analyser	104
3. The scanner	
3.1. the feeding and preprocessing modules	
3.1.1. reserved word languages	106
3.1.2. keyword languages	106
3.2. the lexical analyser	
3.2.1. syntax	107
3.2.2. semantics	108
3.3. lexical parameters	
3.3.1. lexically free languages	109
3.3.2. other languages	109
4. The symbol module	
4.1. the symbol table	111
4.2. analytical types	112
4.3. symbol parameters	112
5. The semantical module, the epilogue and the services	
5.1. the semantical module	113
5.2. the epilogue	114
5.3. the services	114

Chapter VII - "SYDELA"

1. Transition diagrams	
1.1. the original model	116
1.2. the loophole problem	119
1.3. the back-up problem	119
2. Generalised transition diagrams	
2.1. semantical actions	121
2.2. counter transitions	123
2.3. the "EXIT" and "PULL" transitions	124
2.4. secondary entries	125
2.5. multiple output diagrams	126
3. "SYDELA"	
3.1. generalities	126
3.2. declaratives	
3.2.1. terminal declarations	127
3.2.2. name declarations	128
3.2.3. synonyms (class) declarations	128
3.3. statements	129

Chapter VIII - ANALYSER'S STRATEGIES

1. An extra-syntactic example	131
2. Processing of keyword languages	
2.1. the deterministic approach	133
2.2. a non-deterministic solution	134
3. Declaratives and type checking	
3.1. the semantical solution	136
3.2. the syntactical solution	137
4. Use of multiple output diagrams	
4.1. a FORTRAN example	137
4.2. a COBOL example	139

CONCLUSION

Chapter IX - AN APPLICATION: PROGRAMMABLE TERMINALS

1. Architecture of the programmable terminal software	
1.1. an elementary terminal	142
1.2. an "intelligent terminal"	143
1.3. an "intelligent" terminal for a computer network	145
2. Implementation techniques	
2.1. the problems of the local syntactical analysis	146
2.2. structure of the software	148
2.3. methods of implementation	150

APPENDICES AND REFERENCES

Appendix I - CHARACTERISTICS OF SOME MINI-COMPUTERS	153
Appendix II - A SYNTACTICAL DESCRIPTION OF L. S. D.	
1. Basic elements	155
2. Statements	155
3. Expressions	157
4. Declarations	158
5. Conditional expressions	159

Appendix III - A SYNTACTICAL DESCRIPTION OF P. L.

MODULAR ONE

1. Basic elements	160
2. Program	160
3. Procedure	161
4. Declaratives	161
5. Constant & cell	162
6. Statements	163
7. Composite statements	164
8. Assignment, expression, condition	164

Appendix IV - COMPARISON OF THE L. S. D. AND P. L.

MODULAR ONE STATEMENTS	166
------------------------------	-----

Appendix V - THE "SYDELA" ANALYSER AND SOME SERVICES

1. The format of the "SYDELA" generated tables	168
2. The "SYDELA" analyser	170
3. The "SEARCH" and "COPY" procedures	173
4. Some services	175

Appendix VI - A "SYDELA" DESCRIPTION OF ALGOL

REFERENCES	198
------------------	-----

ACKNOWLEDGEMENTS

The author is grateful to the administration, faculty and staff of the late Institute of Computer Science of the University of London for the use of the exceptional facilities of the Institute.

The author is especially grateful to Professor David J. HOWARTH of London University who directed and supervised this research and with whom the author has had numerous fruitful discussions. He provided a great number of arguments and has had an infinite patience.

The author wishes to thank the British Council, the Royal Society, the Institut de Recherche en Informatique et en Automatique and the French Foreign Office for their generous financial assistance.

FOREWORD

The research for this thesis was undertaken between October 1971 and December 1973 under the direction of Professor David J. HOWARTH of the University of London, Institute of Computer Science.

The work originated in the need for a high-level system programming language for the CTL Modular One computer. The P. L. Modular One language was designed during the last three months of 1971, and in February 1972, a first version of the compiler, written in PL/I on the University College IBM 360/65 computer, was available.

A second faster version, also written in PL/I, was soon begun, and the definitive documented compiler was finished in September 1972. In the meantime a wide range of Modular One utilities had been implemented on the 360: an assembler and its text editor, a decompiler, compression and conversion programs, loaders, etc...

"SYDELA", short for SYntax DEscription LAnguage, is a high level formulation of the transition diagrams first introduced by M. E. CONWAY (7) and more recently used by N. WIRTH (20) in the implementation of the PASCAL compiler. This method has been used to write the P. L. Modular One compiler and various other Modular One utilities. Its potentialities for the coding of all types of

translators appeared to be immense. The first "SYDELA" compiler, conceived as a PL/I preprocessor, was coded in October 1971 as a tool for the implementation of P. L. Modular One. A new version of the language, specially designed to cope with the requirements of a small machine, was implemented as a P. L. Modular One extension in September 1972.

Between October 1972 and June 1973, "SYDELA" has been used to write syntactical analysers for COBOL/F 360, CDC ALGOL, CDC SCOPE and CDC EXTENDED FORTRAN. The method appeared to be very practicable and the time required for the coding of a complete syntactical analyser for a high-level language was cut down to less than 1 programmer-month.

Although reference is constantly made to the CTL Modular One computer, the methods and tools described in this thesis can easily be adapted to any mini-computer.

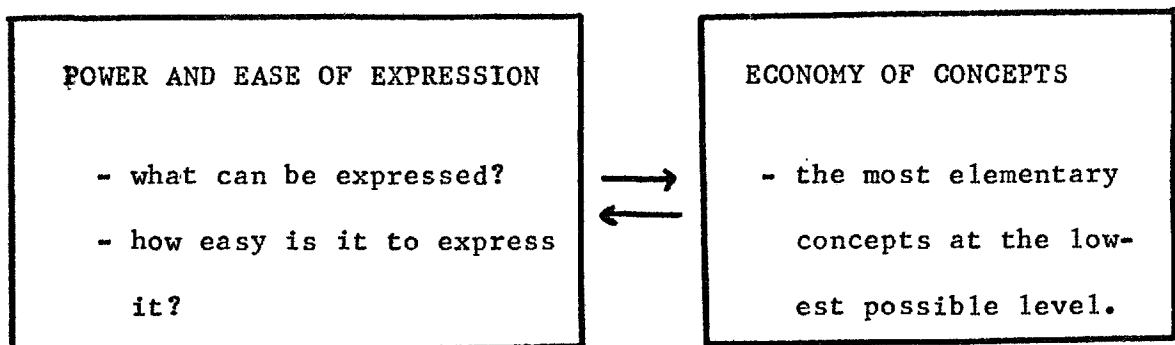
INTRODUCTION

CHAPTER I

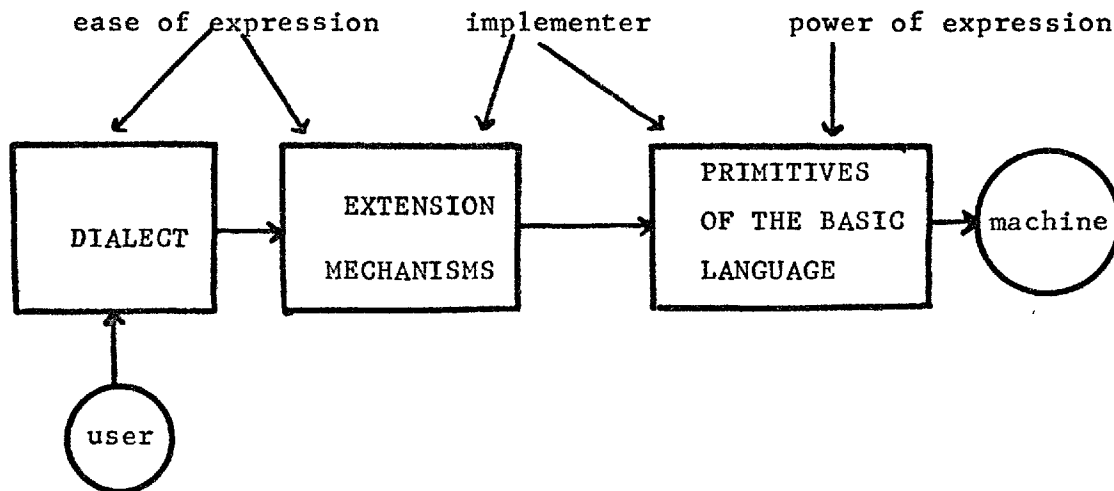
JUSTIFICATIONS AND METHODOLOGY

A major objective for the implementer of a computer system is to provide a reliable interface at the highest level of expression compatible with the intended types of application. However, in the present situation of the development of programming systems which are part of larger complex software systems, the stresses imposed on the design and on the implementation of new programming tools are numerous.

The basic dilemma which is presented to the designer can be summarized as follows:



A solution to this problem is provided by the concept of extensible language which can be analysed as follows:



Although this approach has proved to be inefficient in terms of execution speed and size of the object program, the important concept of a primitive language close to the machine has led to interesting developments in the field of system programming languages.

A system programming language is a high-level assembler language which tries to conciliate the ease of expression with the efficiency of the object code. Its purpose is to replace the low-level languages so that it can be used as a basis for the development of other programming tools. For example, the Syntax Description Language described in Part II is an extension of a system programming language: the user of this system needs not know the assembler language of the object machine; the knowledge of the system programming language is sufficient.

Four important points need to be taken into consideration in the design of a primitive language and of its associated extensions.

- a- the user confidence is a fundamental factor in the success of a new programming tool. This confidence can be gained at two levels:
 - at the system level by providing safe translators and a range of reliable utilities, debugging systems and diagnostics;

- at the expression level by designing the language in such a way that the association of a symbolic representation to a mathematical or mental concept can be done safely; what is more naturally written must be the normal representation of a given concept. At this level the analysis is more subjective, but is certainly worth being made;
- b- the ability to adapt to various contexts and types of applications is important at a time when programming systems are becoming more and more diversified;
- c- the speed of development of the complete system can be improved by concentrating on:
 - the definition of the requirements;
 - the specification of the syntactical and semantical characteristics of the language;
 - the fast implementation of the compiler;
 - the production and testing of utilities;
 - the commercialisation, documentation, teaching, etc... of the system;
- d- the development of computer networks and data transmissions systems allows the implementation of a complete programming system on a small machine without being limited by considerations of size of the compilers and translators which can be implemented on larger remote installations.

CHAPTER II

CHARACTERISATION OF THE MINI-COMPUTERS

The term "mini-computer" is a vague one, and direct comparison between these machines is difficult. The wide range of their use and the differences in their technologies make the correlation in cost and quality for a specific type of application impossible. Objective comparisons can only be made between systems providing the same services in similar environments. This chapter attempts to outline the major characteristics of the "mini-computers" from the hardware, software and types of use points of view.

1. HARDWARE.

A typical mini-computer can be identified as having the following characteristics:

- cost of the basic system between £5000 and £20000;
- memory size from 1K to 256K bytes by increments of 1, 2, 4, 8 or 16K bytes;
- memory cycle time from 0.5 to 2 microseconds;
- word length 6, 8, 12, 16, 18 or 24 bits;
- comprehensive set of logical and bit manipulation instructions;

- direct memory access facility;
- normal range of computer peripherals;
- comprehensive interrupt handling system.

Mini-computers are compact machines requiring little space. Their modularity means that upgrading and modification are just a matter of plugging in different circuits rather than changing the whole system. The environment does not need to be controlled and most mini-computers can be installed right in the place where the information to be processed originate. These machines are thus familiar tools on the shopfloor rather than mythical monsters behind the closed doors of a climatized computer centre.

The typical processor of a mini-computer uses short words and the need to keep its price competitive means that some costly hardware features are often absent. The instruction set is generally complex involving various modes of addressing and direct comparison between two mini-computers can only be made in terms of execution times for specific functions and not by a mere statement of the execution speed of the basic instruction. The user will not always be able to take advantage of the flexibility of addressing which is offered.

The length of the word is an important factor. The longer it is, the greater the efficiency and the greater the cost. A compromise between economy and efficiency is generally a 16 bits words in most applications or 24 bits on scientifically oriented machines. Machines used in process-control or data transmissions may use an 8 bits word, but the low cost is often overridden by the lack of precision and the complexity of the instruction set.

The Direct Memory Access is usually provided on mini-computers as part of the transfer system. This allows the peripherals and the central core to exchange data without interfering with the CPU's processing. The Direct Memory Access is particularly useful in applications using fast peripherals or in the multi-processor systems allowing communications between processors, core and peripherals.

The development of Virtual Memory systems on mini-computers renders a disc necessary. This may slow down the overall speed of the system considerably if the disc is not fast enough, or cost an unjustifiable amount of money if the data rate is too slow to optimize the transfer speed. As a general rule, the data transfer speed and the length of the transferred word should be matched with the device speed and length of word. This tends to be an important consideration as few manufacturers do market their own peripherals, and non-standard peripherals is a common characteristic to many mini installations.

Microprograms are also a common feature to many mini-computers. The microprograms stored in a read-only memory provide the elementary operations which are not hard-wired (floating point operators, editing, data movement, conversions...). They may also be specially designed for a given type of application, where commonly used pieces of code which make little access to the main core may be advantageously set as microprograms.

2. TYPES OF USE.

The four major areas of application of mini-computers are process-control, communications, education or science and commercial. Each of these applications requires different features from the

machine and uses it in a different hardware and software environments.

Few process-control systems require extensive calculations, and the absence of hard-wired floating point arithmetic is not necessarily preventing the use of minis in this area. The good performance of mini-computers in data transfer and in interrupt handling make them suitable for most real-time control systems. The process-control user wants a cheap, simple and reliable device with no or few peripherals and a single piece of software which runs for the machine's lifetime. All the computer will be doing is I/O transfers and conversions with little calculation; but a great number of processor will generally be required.

In communications, the work is oriented towards character handling on a random basis. Very flexible adapters and a great variety of interfaces and terminals is the main objective. A basic communications system will consist of a CPU, front-end processors, communications control units, terminal system controllers and terminals. The very "soft" interfaces of the mini-computers and their low costs together with the wide range of peripherals that they support make them perfectly suitable for that purpose. The modularity of most minis and the small core increments mean that they can be easily expanded to meet expanding demands such as the ones which arise from the addition of terminals to an existing system. However, it often appears that the minis' CPUs are not fully used and that the overheads are high; therefore, one tends to increase the number of tasks allotted to the front-end processor which can perform useful transformations and checks on the transmitted characters.

In the scientific and education areas, the major drawback of mini-computers is that they are unsuited to calculation and batchwork. Hard-wired floating point arithmetic is an exception and the short size of the word leads to a lack of precision. This can be overcome by the use of efficiently coded microprograms to implement the arithmetic operations. The shortage of core space renders the development of complex operating systems and compilers impossible. One thus tends to use the mini-computers in a single language and multi-user environment where their communications capacities are better exploited. BASIC and APL systems are now common on many minis. In this case, as in the case of process-control systems, the mini houses a single piece of software (basically a reentrant interpreter with a communication package) which lasts for the machine's lifetime.

Mini-computers are well suited to commercial applications, particularly in the areas where real-time control is involved such as in stock control or order entry. A mini-computer is as cheap as conventional Visible Record Systems but can grow more flexibly and more than one user can have simultaneously access to the system. However, the use of a mini-computer requires more expertise than the use of desk equipments, more especially as minis do not always have full-sized high level language compilers, and their peripherals are less reliable than the ones of small integrated systems. On the other hand, the wide range of special purpose peripherals makes mini-computers very adaptable to all environments. For example, minis are perfectly well suited to the access and processing of subsets of a large data-bank or can be used as stand-alone real-time systems with a number of remote

terminals.

3. SOFTWARE.

The analysis of the types of use of the mini-computers shows that there are two broad classes of users. The process-control, communications and, to a lesser extent, some scientific applications users need a single piece of software which is repeatedly executed. The commercial and most education and scientific applications users want a good deal of software covering everything from the debugging programs to compilers and complete packages.

Very few mini-computer manufacturers do market their own software for all types of applications. The user is then faced to the purchase of an unexpensive processor and long and costly development periods to produce a working system. Furthermore, most minis can only be programmed in low level language to achieve efficient performances. The user must employ skilled programmers who have a detailed knowledge of the machine and of its assembler language. These people are scarce and tend to be specialised in a given type of material and have some difficulties to adapt to different ones.

The manufacturers will invest in the development of a range of stand-alone packages (say emulators of hardware terminals) which will allow them to find immediately a competitive place on the market for these kinds of applications. The first effort is not concentrated on producing a valid general purpose operating system or executive. The result is then, that, in all the packages, the line between the executive functions and the application programs is badly defined. The major effort is put on the fast produc-

tion of small-sized workable packages. At a later stage, when a general purpose operating system is made necessary by marketing imperatives, the need to keep an upward compatibility greatly affects the design of the system which suffers from the lack of a well defined architecture in the previous packages.

Compilers for high level languages exist on a certain number of mini-computers. FORTRAN, COBOL, BASIC and CORAL-66 are the four most common high level languages available on mini-computers in the U.K. FORTRAN may be used both in scientific and process-control applications. However, the lack of bit manipulation operators and its limited number of input/output techniques make it inefficient in terms of instructions generated to execute a given task in comparison with the same task coded in assembly language. In business data processing applications, COBOL-like languages can be reasonably well compiled. For the conversational user, BASIC and APL are now available on some mini-computers. Even sophisticated real-time and commercial problems can be easily coded in these languages, but with important penalty on core storage and execution time. Universal System Programming Languages like CORAL 66 or LSD are now more common on mini-computers, however their low efficiency (a CORAL-66 program is 40% longer than the equivalent assembler program) make them unsuitable for the development of some pieces of software.

The most common utilities which can be found on minis are text editors, hexadecimal or octal dumps, debugging programs, binary copy programs, disc initialisation, dumps/restore routines, sort/merge packages and various file handling utilities. Assemblers and assembly languages vary greatly from one machine to another;

they may be one, two or three pass and generate absolute or relocatable code. The most advanced assemblers include program sectionning and linking, production of reentrant and simultaneously reusable code, macro definition and conditional assembly facilities. Very few systems allow the link-editing of mixed assembler and high level language sequences.

PART I

THE SYSTEM PROGRAMMING LANGUAGE

CHAPTER III

SYSTEM PROGRAMMING LANGUAGES

This chapter discusses the general characteristics and requirements of system programming languages for mini-computers and their implications on the portability of the resulting software. A distinction is made between two types of portability. The external portability, which aims at the use of the same programming language for a range of different machines, is condemned as being unefficient and misleading. The internal portability, which is restricted to the problem of adapting software to the variations of the peripheral equipments and of the software facilities of a given installation, is analysed and solutions are proposed.

1. GENERAL PHILOSOPHY OF SYSTEM PROGRAMMING LANGUAGES.

System programming languages (SPL) aim to provide the users with high level facilities of expression together with an efficient coding of the object programs. SPLs may be used for application programming as well as for system development. This implies a large flexibility in their definition: the language should not hide the machine from the user who wants to use its specific features, but it should not force its limitations on the user who wants to ignore the basic structure of the machine.

1.1. language level.

A SPL generally reflects both the characteristics of the machine on which the object code is to be executed and the way programs are commonly written in high level languages. Hence the programmer may specify some of the hardware registers of the machine as operands and control the allocation of core space for the data items that he defines. He may also ignore these factors and use the language exactly as he would use Algol or P.L. / 1. Even in the same program, some sections which are repeatedly executed may be coded in a low level form as some other sections may be coded more freely.

Taking as an example the coding of a compiler, one may define three major levels of programming.

- a- The lexical level contains the critical loop of the compiler which is performed for every character of the input stream. This part of the compiler should be coded with the greatest efficiency, even if it may be detrimental to a good understanding of the structure of the program. Furthermore this part of the compiler is very rarely modified, hence a clear coding is not an absolute necessity.
- b- The syntactical level consists of a loop which is executed each time a new syntactical unit (name, constant, keyword, special symbol...) is encountered in the input stream. In the coding of this part of the compiler, one should look for a fair compromise between efficiency

and clarity of programming as one can have to modify it from time to time.

- c- The semantical level consists of the set of all the code generators and the actions which are specific to a given combination of input syntactical units. This part of the compiler is the one which is the most often subject to modifications (maintenance, releases, extensions...). The important factor is then to make this level as easy to understand as possible by use of a high level form of expression.

The following P.L. Modular One examples illustrate the differences of coding between these three levels.

EXAMPLES: converting a binary positive integer X in its ISO 7 five digits external representation to be stored in BUF(0).... BUF(4).

form 1: high (semantical) level programming.

```
I = 4; A = X;
WHILE (I >= 0)
    BUF(I) = A \ 10 + 48; A = A / 10;
    I = I - 1;
END;
```

form 2: intermediate (syntactical) level programming.

In this form the control variable of the WHILE loop is forced to be the hardware register M (\$), hence saving intermediate load and store instructions for indexing and testing.

```

$ = 4; A = X;

WHILE ($ >= 0)

    BUF($ ) = A \ 10 + 48; A = A / 10;

    $ = $ - 1;

END;

```

form 3: low (lexical) level programming.

In this form, instructions are saved by forcing the variable A to be the accumulator (@) and using the fact that, when a division is made, the accumulator (@) contains the quotient and the extension (#) holds the remainder.

```

$ = 4; @ = X;

WHILE ($ >= 0)

    # = @ \ 10 + 48;

    BUF($ ) = #;

    $ = $ - 1;

END;

```

The numbers of object instructions for each of these levels are respectively 21, 16 and 10. This is obviously an extreme example, but it clearly shows the way in which a system programming language can provide a complete set of solutions to the dilemma between the ease of coding and the efficiency of the resulting object code.

1.2. self-documentation.

One of the most immediate advantages of high level languages lies in their external syntactical expression which makes programs almost self-documenting. There is an obvious progression in the legibility of the three following forms of expression.

form 1: assembly expression.

```

LDM I;
LDA X(M);
ADA Y;
STA Z;
STA W;

```

form 2: intermediate assembly-like expression.

(@ stands for the accumulator and \$ for the index)

```

$ = I;
@ = X($);
@ = @ + Y;
Z = @;
W = @;

```

form 3: high level expression.

```

W, Z = X(I) + Y;

```

However, even the most elementary compiler would generate exactly the same object code for all these three examples. The main difference between these forms lies in the expressive power of the external representations. On a small machine, the difference is even more important. For example, the simple operation:

```

Reg A, Reg B := Reg M v Reg B;

```

which can be symbolically coded:

```

@, # := $ v #;

```

must be expressed in the Modular One assembly language as follows:

```

CPYL RA,RB M1,B1,EQV,NOT1;

```

which has no expressive power at all!

A psychological study of an assembly language programmer would show the following steps in the elaboration of a program:

- a- make a high-level, mathematical and macroscopic representation of the algorithm;
- b- extract a part of the algorithm and formulate it in an intermediate, more precise and concise representation. This representation may be similar to the one of form 3 above;
- c- assign registers and names to the various items entering in the intermediate formulation. The result of this step is an expression like the one of form 2 above;
- d- transcribe this formulation in actual assembly statements, using general or personal conventions, habits, tricks...;
- e- add some comments to explain the meaning of this section, if it seems necessary;
- f- select another part of the algorithm and go to step b.

As a result, the program generally suffers of three major defects, namely:

- a- the comments are generally insufficient, as the space available on the card is limited and the work of writing comments rather tedious;
- b- comments only give a microscopic description of the algorithm;
- c- there is no assurance that the comments accurately describe the statements which have been actually coded.

One should also notice that, in most case, the work of steps c and d in the above description can be done automatically. The only few instances where special care must be taken are encountered when a very efficient coding is required and the intermediate representation does not imply this optimal coding.

A system programming language deals with this kind of requirements, by using a variable level of programming which allows a compact expression of the simple unambiguous statements and however gives to the user a permanent control on the generated code, so that he can intervene in a lower form of expression at some critical points. The tedious work of transcription is thus suppressed, and the time which is gained can be advantageously used for concentrating on some interesting points in the program and for adding some general macroscopic comments.

1.3. programming structures.

One of the most commonly used construction in programs is the Algol form:

if condition then statement-1 else statement-2

which is generally compiled in the following sequence of instructions:

```

TST  condition
JMP  L2
.
.
.
JMP  EXIT

L2: .
.
.
EXIT: .....
```

An extensive use of this type of high level construction greatly helps the programmer and reduces the number of possible sources of errors. A SPL should include this class of facilities, as long as it does not conflict with the structure of the machine. For example, it would be irrelevant to implement recursive procedures

if the hardware does not provide a local workspace pointer register.

Although high level programming structures are a great improvement in the ease of programming, they are generally detrimental to the efficiency of the object code and may hide some distinctive features of the hardware of the machine. In FORTRAN, a DO-loop expresses nothing more than the equivalent sequence of IF and GOTO statements; it can be considered as a simplification in writing. But the DO-loop restricts the normal potential power of expression of the hardware by forcing some limitations (no variable steps, etc...). More generally, a high level programming structure is balanced by a loss in the capacity of expression.

When defining the programming structures of a SPL, one should try to conciliate the ease of coding with the most accurate representation of the hardware architecture of the machine. Three major factors are used in the design:

a- respect of some form of "tradition".

It would be ridiculous to depart from some common traditions in the programming world, when the result does not fully justifies this change. Most system programmers are not Algol 68 specialists and one will gain an important amount of the time needed for teaching the language if its structures are already familiar. At the limit, the very syntactical description of the language should give most of the required information on the semantical meaning of its constructions. Furthermore the respect of some form of tradition will ease the solution to the problem of software transportation (see 3.).

b- analysis of the requirements of system programming.

System programming is very different from application programming; the types of the problems which are dealt with require different modes of expression. In general a system programmer will need a better understanding of the processes for space allocation and recovery, block structuration, passing of arguments, etc... These processes must be built in the language in a way which is transparent enough for the user who wants to understand them fully. Some constructions are also very common to system programming: multiple switches, label modifications, logical tests on bit patterns, etc... for which satisfactory modes of expression must be provided.

c- the hardware requirements.

In practice one generally builds in a SPL the structures which are both commonly used and efficiently compiled. However, there will be some cases when the structures are not adapted to the coding of a specific problem. The user must then be given the possibility of switching to a lower level of expression. This interleaving of different levels of coding implies that the programmer must always have a good knowledge of the code which is generated for a given construction to be able to interfere at the critical points. The rules defining the use of procedures together with the meaning of names, the linking of functions, communication of arguments and the process of allocation of space to variables are fundamental to a good use of a SPL. These characteristics may vary considerably from one language to

another because they are their most machine-dependent features. A standard scheme for these mechanisms must be built in the language to cover most applications, but the user should be able to over-ride this system, if he judges it necessary and control all by himself.

1.4. data types and structures.

A wide variety of data types (integers, labels, pointers, bitstrings, ...) and data structures (vectors, arrays, stacks, linked structures, ...) is used in system programming. However the kind of structure which is used to solve a specific problem is often dictated by the architecture and the addressing modes of the machine. Hence the designer of a SPL bears an important responsibility in the definition of the data types and structures built in the language. The general rule, like in the definition of programming structures, is that a data organisation must not hide the machine from the user when he wants to over-ride the standard facilities provided nor should it restrict the addressing capacities of the hardware.

examples:

data types

- real constants and variables are not commonly used in system programming, they should not be implemented on a machine which has no wired floating point arithmetic. When there are different lengths of real available, the SPL must provide some ways of specifying the length of the operands or operators (short, long, double...) together with a standard default length.

- when no special bitstring manipulation operator exists in the hardware, the length of a bitstring variable can be forced to be a multiple of the length of a word or of a smaller addressable unit.
- pointers are an identification of an address together with some other machine-dependant information (segment, global, common, local, protection, etc...).
- labels may be associated with a block, a procedure or a statement. Label variables may be used in conditional branch instructions.

data structures

- vectors are very often used, attention should be especially given to the optimisation of the vector references (allowed forms of indexing).
- arrays with more than one dimension should not be implemented if there is no hardware multiple indexing facilities (like the 360 base and index registers) as they could lead to an unefficient object code.
- stacks must be implemented only if there are built-in hardware stack instructions or some special type of addressing.
- non homogeneous aggregates may be assimilated to vectors, arrays or pointer references depending on the addressing system of the machine.

1.5. machine interface

The machine interface facilities of a SPL provide two main types of services:

- a- give access to the standard functions of the Executive which cannot be built in the language for reasons of efficiency or simplification of the syntax;
- b- allow the lowest form of programming in an assembly-like form.

The problem of the communication with the Executive is related to the question of portability and will be discussed later.

The lowest form of programming is used when the programmers feel that the functions provided by the language are not fitted to the expression of a specific problem (say coding an input/output statement). However, it must be possible to refer to variables and statements defined at a higher level. This implies a perfect knowledge of the implementation of the data types and of the various modes of addressing associated with each type of structure. A solution to this problem is to allow "pseudo-assembler" expressions where complex references to data elements can be coded in the operand part of the assembly statement.

For example, in P. L. Modular One the following statement:

```
CODE LDA X(I),
      LDM PT -> C.D.E;
```

will be allowed and will generate the following sequence of object machine instructions:

```
LDM      I
LDA X($ )
LDB      C
ADB      D
ADB      E
LDM PT(#)
```

1.6. debugging facilities.

Debugging is surely the most tedious part of a system programmer's job. This work could be greatly eased if high level debugging tools are provided as standard functions of a SPL. These facilities fall into two classes.

- static facilities
 - cross reference table
 - map of core allocation
 - actual address of the high level statements
 - "decompilation" of the source program
 - post mortem dumps
- dynamic facilities
 - snapshots and dumps of specified areas
 - conditional dumping
 - tracing
 - save and restart at specified points

2. THE MYTH OF EXTERNAL PORTABILITY.

2.1. external portability.

The degree of portability of a language measures the ease of transporting a program written in this language from one context to another one. One can make a distinction between two different kinds of portability:

- a- the external portability consists in transporting a program from one machine to another one;
- b- the internal portability is the transportation of a program from one configuration of a given machine to another confi-

guration of the same machine (with differences in peripherals, executive, operating system, core size...).

Although a lot of research work is being done on the problems of external portability, its application to a SPL seems rather mythical. It is already fairly difficult to transport automatically programs written in high level languages such as COBOL and FORTRAN, so that it seems impossible to design a language which is at the same time low-level, efficient and reasonably portable from one machine to another. The results of the OS6 research project team at Oxford University show that trying to transform a Modular One into a BCPL-machine leads to a loss in execution speed of up to a factor of 50.

The desirability of external portability for a SPL on a mini-computer may also be condemned as leading to a contradiction. On the one hand, in a low-level or system programming language, programmers are strongly influenced by the architecture of the machine, trying to make the best possible use of its specific features. On the other hand, an externally portable SPL will synthesize the characteristics of a whole range of machines, thus hiding their differences and putting forward their common factors. In this case, the programmers will be directed more by the syntactical structure of a "virtual machine" than by the characteristics of the actual existing hardware.

Furthermore, a portable SPL would contain at a high-level only the features which can be implemented on all machines, irrespective of whether it can be efficiently compiled or not. Hence, on a specific installation, a few features only will be implemented (the ones which are common to the whole range), and most of them

will be unefficiently compiled (dealing with an abstract machine far remote from the actual one).

The following critical study of an externally portable SPL will point out more precisely some practical consequences of the design of such a language.

2.2. L.S.D.: an externally portable language.

2.2.1. aims of L.S.D.

L.S.D. (Language for System Development) is a machine independent SPL designed at the U.K.A.E.A. Culham Laboratory. Its description which is self-explanatory is given in Appendix II. L.S.D. also aims to provide high-level system programming facilities on a whole range of small and medium size computers. L.S.D. has been designed to be easily compiled and macro-processed using the portable Culham SID syntax generator and STAGE-2 macro-processor.

Two implementations of L.S.D. are now available on CTL Modular One and on ICL System 4/70. These transform L.S.D. programs into equivalent assembly language programs. The L.S.D. system is designed to be used in two stages. Programs will be debugged by means of a compiler which generates an interpretative code. The final production program will be generated by means of the macro-processor which is not flexible enough for debugging purposes.

2.2.2. the cost of macro-processing.

Beyond the fact that the macro-processor is not flexible enough to allow program development, the use of this method of compilation seriously restricts the syntax of the language. It implies an extreme rigidity in the coding of the program (only one

statement per card, no possibility of spreading a statement on two cards, limitation in the use of comments, etc...), and in the syntax of some statements (forward references to label constants must be enclosed between colons i.e. GOTO :LAB:, etc...).

Furthermore it is questionable whether the development of both a compiler and an interpreter is economically sound. Two programs must be coded and debugged which achieve similar purposes, but expressed in two different ways. Generally both the compiler and the interpreter will be incomplete. The time spent on the development of the interpreter could have been more valuable if concentrated on the production of a more efficient compiler providing more facilities to the user.

2.2.3. level of the language.

In L.S.D. there is no real way of coding a statement at the lowest level:

- no interface with the Executive is provided;
- no way of referring to a high-level defined variable inside an assembly sequence of instructions.

More generally, L.S.D. seems to be at a too high level for an efficient system programming language. Expressions can be of the form:

$$A \neq (C - D)$$

or $A + ()$

which involve an allocation of auxiliary storage completely out of the control of the user. The general form of a condition is:

$$\langle \text{expression-1} \rangle \langle \text{relation} \rangle \langle \text{expression-2} \rangle$$

where relation is any of the comparison operators. On small machines

no comparison statement is generally built in the hardware. The comparison must be compiled as:

(expression-1) - (expression-2) relation 0

which leaves an unknown result in the accumulator and requires the allocation of an auxiliary core location to store the intermediate result. In procedures, one can specify a list of input (which are left unchanged by the procedure), and output (results) parameters; this is certainly too high-level for a system programming language. Furthermore, this feature has proved to be unnecessarily restrictive and difficult to implement.

2.2.4. the cost of machine independence.

As L.S.D. can be used on a wide range of small computers, its representation of the object machine is over-simplified. Hence, the "virtual LSD machine" is supposed to have a single accumulator and a restricted number of operations on the index register. The language ignores the segmentation of the core of the Modular One in three segments and, as a consequence, the notion of pointer is ill-defined.

On the other hand, some of the features which exist on small computers, but not on the Modular One, are implemented in L.S.D. The INC, CLEAR, DEC, IREG operations are irrelevant on a machine where literal operands can be specified for the operations relative to all registers. The GOTOVIA statement conflicts with the Modular One architecture which does not allow writing in the code segment.

2.2.5. other points.

Many other characteristics of L.S.D. are not adapted to the Modular One (and some other not to the ICL System4). Some inexpli-

cable limitations have also been brought to the syntax of some statements. For example, in a IF statement, both the THEN and ELSE clauses must be simple, non-composite statements. This practically restricts the power of the construction to the rather poor:

IF condition THEN GOTO L1; ELSE GOTO L2;

discussed in 1.3. In a similar manner, it is not possible to achieve the grouping of more than one statement in a single unit (like the PL/I DO or BEGIN constructions).

Appendix IV presents a comparison between the L.S.D. and P. L. Modular One sets of statements.

2.3. an alternative.

Although this study is only qualitative (measures of the efficiency of the object code should be made), it shows that a high level system programming language must be designed to represent accurately the architecture of the machine on which the object code is to be run. This is generally detrimental to the portability of the programs from one machine to another.

However, a simple analysis of SPL written programs shows that only a small amount of the coded statements are really machine dependent (in the sense that they use specific features of the hardware). The other statements are straightforward assignments or transfers of control which pose no real problem of transcription from one machine to another.

Assuming that the syntax of all SPLs do follow some general "traditional rules" (see 1.3.a), it would be easy to design a kind of "intelligent" transcription program working in conversational mode. This program would deal automatically with the transcription

of the constructions which are common to the source and the object SPLs, and ask for the help of the human operator for the ones which differ. Such a program should also include general editing facilities to allow the tracing of all occurrences of a given identifier, statement or construction, produce cross-reference tables, etc...

3. THE INTERNAL PORTABILITY.

3.1. problem.

Any COBOL or FORTRAN application programmer who has ever tried to transport one of his programs from one machine to another one, or even from one version of an operating system to a new one, knows the difficulties which are inherent to the adaptation of a program to a different context. However, most large scale computers can be considered as proper "COBOL machines" or "FORTRAN machines". The problem of transportation is not caused by the differences in the architectures of the object computers, it is principally due to the interface with the machines (executive, operating system, access to files, etc...) and its conventions. In high-level application languages, the main sources of difficulties can be located in the transposition of the Job Control Language, in the differences in the allocation and the sharing of peripherals and in the variety of file management methods. In low-level languages such as SPLs, the main problem will be at the interface with the Executive and with the operating system.

3.2. internal portability.

The degree of internal portability of a programming system measures the ease in which a program can be adapted to a configuration of a given machine different from the one which was initially intended. One can make a distinction between two kinds of internal transfers:

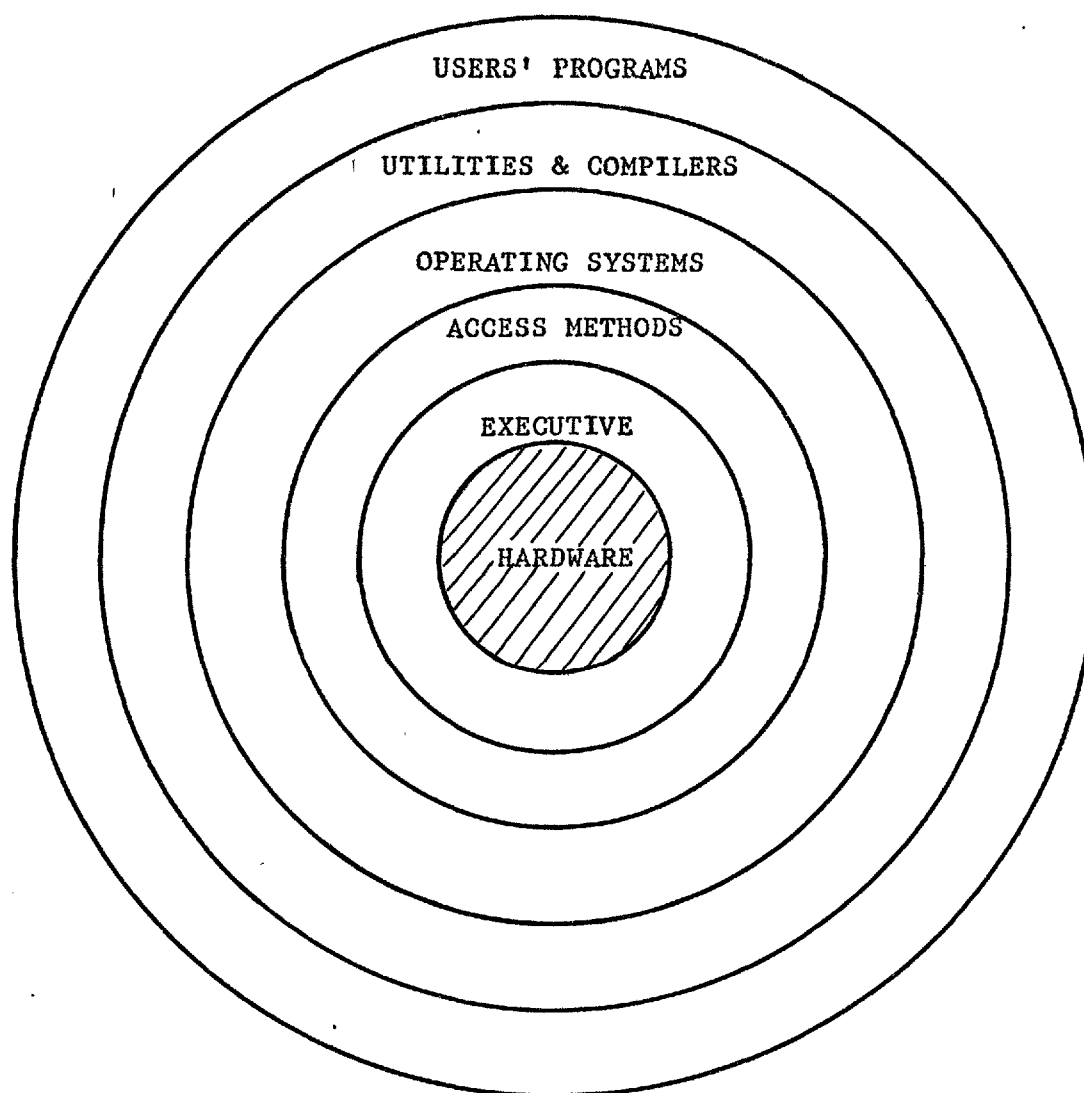
- modifications keep the Executive or Operating System and modify their context (adding or suppressing peripherals, core space, programs, utilities...);
- transplants consist in using an existing program under a different Executive or Operating System.

The "onion ring" design simplifies the transplant operations, and standard interfaces bring a solution to the problem of modifications.

3.3 the "onion ring" design.

The "onion ring" design consists in separating the software architecture in functional layers, where each layer can communicate only with the layer immediately below or immediately above. The more central the ring is, the more machine dependant it will be.

A typical "onion ring" design for a small computer will be as follows:



- the Executive provides the most elementary way of communication with the external world and with the hardware. It will generally include a simplified tasking mechanism and units to deal with the external interrupts. The Executive is obviously machine dependant and must be coded in a low level form.

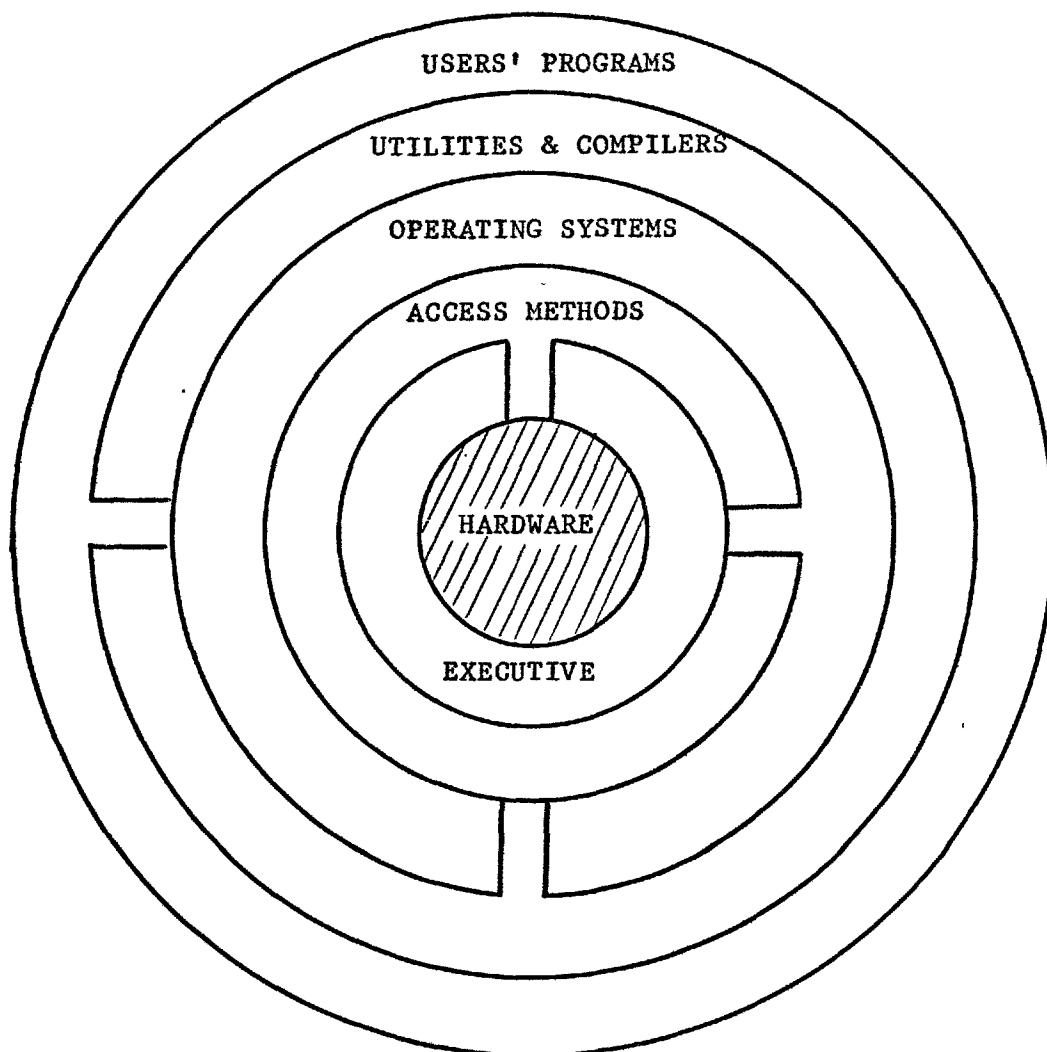
- the Access Methods are used to communicate with the external world in a machine-independant manner; the notion of 'peripheral' being replaced by 'file'. The Access Methods will deal with sequential, direct, indexed or special types of data exchange using the elementary functions provided by the Executive.

- the Operating System is the "Manager" of the architecture it uses the access methods to take control of resources which he allocates to the systems and users' tasks.

- the Utilities and the Compilers may have some special privileges and are the intermediaries between the users and the operating system.

Although some authors strongly criticize this type of design for its lack of flexibility (number of intermediate layers to access the hardware), its advantages in relation to internal portability are obvious.

If the structure of this type of design appears to be too rigid, one can always provide some channels to short-circuit a layer, thus making the communication faster.



But it must be realised that the larger those channels will be, the more detrimental it will be to the ease of transportation of the software.

3.4. standard interfaces.

This is the most obvious contributing factor to the internal portability of a program. The interfaces with the Executive and, more generally, with the inner rings of the "onion ring", must be as machine-independent as possible. This can be achieved in two independent ways.

a- the use of high level concepts.

For example, when passing an address as a parameter to the Executive, this address should be a pointer in the SPL sense. It is the Executive's job to map this relative address with the actual one.

b- the uniformisation of concepts.

In the innermost rings of the design, the SPL programmer should be able to address all peripherals in the same way, irrespective of their different structures; the notion of channel of information must replace the one of physical channel and peripherals. The association of a channel of information with a physical channel is under the Executive responsibility. Similarly, in the outermost rings, the notion of file replaces the channel of information. The access methods are responsible for the linking of a file with a channel of information.

CHAPTER IV

P. L. MODULAR ONE

The design of P. L. Modular One has been influenced by the remarks of the preceding chapter and by the type of programming which is currently made on the machine. This chapter does not give a complete and detailed description of the language which can be found elsewhere (9), it selects some characteristic features of the language and links them with the architecture of the Modular One. The reader can also refer to Appendix III for a complete syntactical description of the language.

1. BASIC CHARACTERISTICS AND PROGRAM ELEMENTS.

P. L. Modular One is a system programming language designed to cover most of the programming for the CTL Modular One computer. It may be used both for application programming and software development.

There are very few restrictions in the format of the P. L. Modular One statement. The programs may be written without consideration of column-checking. The format is entirely free. Each statement is terminated by a semi-colon and the next statement may begin in the column immediately following or after any number of intervening spaces.

1.1. identifiers.

In P. L. Modular One programs, names are assigned to data, labels, procedures, figurative constants, registers and pointers. These names must follow the syntactical rules for creating identifiers.

a- special identifiers.

The symbols @, # and \$ are used to refer to the content of the registers A, B, and M considered as integer values.

b- names.

An identifier is a string of letters and digits beginning with a letter. The string must not exceed 15 characters in length.

c- reserved words.

Some identifiers have special meanings in P. L. Modular One programs. They must not be used to name data or other programmer defined items. These reserved words, when set in the proper context, specify special actions to be taken or the nature of a data item.

1.2. program components.

A P. L. Modular One program is built from the basic constructions which are either declaratives or statements. Statements may be either simple or composite.

1.2.1. declaratives.

Declaratives give to the compiler the description of data items. There are four types of declaratives.

The COMMON statement introduces the description of the data which may be shared between two programs. The GLOBAL statement describes the items which are global to the whole program. The DECLARE statement gives the description of the items which are local to a procedure. The arguments which follow the name of a procedure describe local items which may be used for passing values to a procedure or to a subroutine.

1.2.2. statements.

There are five types of statements.

- keyword statements have a reserved word to indicate their function.

```
example:      GOTO L;

              ALLOCATE A; FREE B;
```

- replacement statements contain one of the two possible assignment symbols (= or ->=), they have no keyword.

```
example:      DELTA = A * X * X + BETA;

              PT ->= ABS;
```

- the null statement consists of only a semi-colon and indicates no action.
- a composite statement contains one or more other statements as part of its body. The final statement of a composite statement is a simple statement or one of the reserved words END or FI which are followed by a semi-colon. Composite statements may contain other composite statements as part of their bodies.

```
examples:      IF (A=0) THEN C=6; D=7 ; E=8 ;
                ELSE C=7; D=8 ; FI;
```

```
WHEN (A=0) UNLESS (B=0) UNLESS (C=0)
```

```
GOTO L;
```

```
WHILE (I 0) A = A+F(I); I = I-1; END;
```

- a group is a sequence of statements headed by the reserved word BEGIN and ended by the corresponding END statement. It may be used wherever a single statement is allowed. Execution passes sequentially into and out of a group.

example: UNLESS (I=0) BEGIN I = K/I; J=J+1; END;

1.2.3. statement labels.

Simple statements, composite statements and groups may be prefixed by one or more labels. A label identifies a statement so that it can be referenced at some other point of the program. If more than one label is specified for a single statement, they may be used interchangeably to refer to that statement.

examples: STAT1: A = 6;

LAB1: LAB2: WHEN (AOV) GOTO OVERFLOW;

OVERFLOW: ;

In the third example the label "OVERFLOW" is prefixed to a null statement.

1.2.4. statement switch prefixes.

Switch prefixes may be prefixed to some of the statements which are nested inside a CASE statement. These prefixes identify the different actions which are to be executed depending on the value of the switch variable associated to that CASE statement. Switch prefixes are unsigned integer constants connected to the statement by a right parenthesis. When different values of the

switch variable imply the same action, more than one prefix may be used for a single statement. In this case the prefixes are separated by a dot. Labels and switches may be prefixed to the same statement, but the switch prefix list must precede the label list.

```

example:      CASE N OF 6
               1) I = 3*J; L = 2;
               3.4) LAB1: LAB2: L = 4; GOTO LAB3;
               6) ;
               5) I = 3*K; GO TO LAB2;
               2) LAB3: I = 7;

               END;

```

the switch 6 is prefixed to a null statement which means that no action is to be taken when N is equal to 6.

After the execution of an action list, the control passes to the statement immediately following the END keyword, unless an unconditional jump is specified.

2. DATA ELEMENTS.

A data is the representation of a piece of information or of a value. P. L. Modular One allows the programmer to define and use a wide range of different data and data organisations. A variable is a symbolic name having a value associated to it at any time that may change during the execution of the program. A constant is the representation of some value which is left unchanged during the execution. An organisation is a complex association of homogeneous and heterogeneous variables.

2.1. data classes and constants.

The class of a data used in a P. L. Modular One program falls in one of five possible categories.

- integers are arithmetic values on which one can perform arithmetical operations;
- Booleans may have either the value TRUE or FALSE. Boolean variables and constants can be combined with the Boolean operators AND, OR, and NOT;
- bitstrings or words are the representation of a 16 bits word. The operations allowed on these words are OR, AND, NOT and SHIFT;
- labels are used to refer to a statement or to control the execution of the program;
- pointers refer to some core location. They are used to access an element in a structure.

Constants are of four different classes: integer, Boolean, bitstring or label, and may have either numerical or symbolic representations.

- numerical representations exist for integers and bitstrings. An integer constant is represented by a possibly signed decimal number; a bitstring constant is represented by an hexadecimal number prefixed by the symbol @.
- symbolic representations fall in three classes:
 - TRUE and FALSE are the standard representations of the two possible Boolean constants;
 - a label constant (see 1.2.3.) is a symbolic representation

of the address of a statement and is used to refer to that statement so that, during execution, control can be passed to that statement through a reference to its label;

- a figurative constant is a programmer defined mnemonic associated to a given value. The association is made by means of a CONSTANT declaration in some declarative statement. Any occurrence of this mnemonic in the program is equivalent to the occurrence of the associated constant.

example: BOOLEAN CONSTANT YES = TRUE,
 INTEGER CONSTANT DOZEN = 12;
 .
 .
 .
 WHEN (COUNT-DOZEN = 0) FLAG = YES;

is lexically equivalent to:

 WHEN (COUNT-12 = 0) FLAG = TRUE;

2.2. variables.

The characteristics of a variable are not immediately apparent in a name. Since these characteristics, or attributes, must be known, reserved words are used to associate a name to a given set of attributes. Variables may be either simple variables or register variables. A simple variable can be of integer, Boolean, bitstring, label or pointer class. A label variable will refer to a label constant and may be used for conditional switching. A pointer variable will contain the address of a core location, this value can be set by a pointer assignment such as:

 P ->= A;

which sets the pointer variable P to the address of A.

Some other valid representation of variables are register variables. The register A considered as an integer, written $\textcircled{a}$ for short, register B, written $\#$, and register M, written $\$$ are variables. These tell the compiler to leave the registers as they are and therefore to generate no code. Hence $\textcircled{a} + X$ is a valid expression, but $X + \textcircled{a}$ is not. Register M is an allowed subscript when no code is to be generated because the register already contains the required value. Excessive use of the $\textcircled{a}$, $\#$, $\$$ register variables should be discouraged since it makes the program more difficult to read. However a correct use of it may result in an increased efficiency of the generated object code. Hence the multiple assignment

$$A(I), B(I) = C + D - E;$$

may be better written:

$$A(I), B(\$) = C + D - E;$$

since multiple assignment are done left to right, so that $B(\$)$ means $B(I)$ because register M has already been loaded with I in the assignment to $A(I)$.

In the same manner, the sequence of statements:

$$A = I + 1; B(A) = A;$$

may be more optimally coded in:

$$\$, A = I + 1; B(\$) = \$;$$

and the sequence:

$$A = A - 1; \text{ WHEN } (A = 0) \text{ GOTO } L;$$

may be more efficient if written:

$$\textcircled{a}, A = A - 1; \text{ WHEN } (\textcircled{a} = 0) \text{ GO TO } L;$$

The programmer may like to use the registers with a class different from integer or to give symbolic names to them. He may therefore use the REGISTER attribute.

INTEGER REGISTER A = @;

means that any further occurrence of A will be equivalent to the occurrence of @. In the same way:

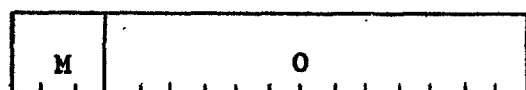
LABEL REGISTER X = #,

WORD REGISTER W = \$,

declare X and W to be the symbolic representations of the registers B and M respectively considered as label and bitstring variables.

2.3. organisations.

In P. L. Modular One, data items may be grouped together to form data aggregates called organisations. A given element in an organisation is accessed through indirect addressing by an "indirect word" which describes the organisation. On Modular One, an indirect word has the following structure:



O is a 13 bits address specifying the position of the item. The field M specifies the segment (Y or Z) to which the item belongs together with the register which is used as an index, these may be M, B or none. These three indexing modes dictate the types of organisation which must be built in the language.

2.3.1. arrays.

Data elements having the same characteristics may be grouped together to form an array. An array is a vector of elements, each of which has the same class. Any given item in the array is referred to by giving its position relative to the beginning of the array by means of an index.

The implementation of an array uses the M indexing facilities of the indirect word. An array is described by an indirect word giving in its 'O' section the address of the first element and specifying in its 'M' field an indexing by the M register. When referring to an individual item of the array, one loads register M with the specified index and addresses the element indirectly through the description word.

The limitation on the possible operations that can be performed in register M restricts the form of an index to:

$$\left\{ \begin{array}{l} \text{constant} \\ \text{variable} \\ \text{register-M} \end{array} \right\} \left[\begin{array}{l} \{+\} \\ \{-\} \end{array} \right] \left\{ \begin{array}{l} \text{constant} \\ \text{variable} \end{array} \right\} \dots$$

Hence

$$B(6), L(\$), A(\$ + 6), K(I - 1)$$

are valid references to array elements, but

$$B(\oslash), L(I + \$), A(A(\$)), K(2\&I)$$

are not.

2.3.2. stacks.

A stack is a collection of homogeneous elements, but, unlike an array, only one element can be accessed at any given time. This

element is known as the top of the stack. When referring to the top, the programmer uses the name of the stack. A stack name may be used wherever a simple variable is allowed. Every appearance of a stack name on the left hand side of an assignment causes the definition of a new top and the assignment of the right hand side expression to that new top. If SU has been declared as a word stack of 10 elements

WORD STACK SU(10),

the statement:

SU = W & 0FF0;

causes the value of the expression to be stacked on SU. The statement:

SU, SU = W;

causes the stacking of two values on SU.

When the programmer wants to refer to a value which has been previously stacked, he may use a DEQUEUE statement which decrements the stack pointer of the specified number of steps. All the values which were stacked over the new top are lost.

A stack is implemented by means of an indirect word which specifies no indexing. At each assignment, it is incremented by one. The DEQUEUE statement decrements the indirect word of the specified number of units.

2.3.3. structures.

Structures use the last indexing facility available on Modular One: B indexing. By definition, a pointer is an indirect word whose 'M' field specifies B indexing and 'O' field the address of a core location. A structure is a complex aggregate of

non homogeneous data which possess a logical relationship to one another.

Access to a structured item is made by means of forms and pointers. The general format of a reference to a structured item is:

pointer -> form-name [. form-name]

The pointer indicates the base address of the element, the form name gives the offset relative to this base which is loaded in register B for indexing. When more than one form name is specified, all offsets are added to give the position of the required element relative to the base address contained in the pointer.

A form is declared by means of a FORM attribute. Hence the declaration:

INTEGER FORM IX ORIGIN (2),

causes the reference P->IX to be a reference to the item situated two locations after the word pointed by P. This word is considered as an integer. When more than one form is used to refer to an item, the class of the item referred to is the class of the last form in the list.

The offset specified in the ORIGIN clause of the FORM declaration may be a variable. When referring to such a form, the offset is the current value of the variable at the time of the reference.

INTEGER FORM F1 ORIGIN (6),

INTEGER FORM F2 ORIGIN (J),

.
.
.

P ->= AREA; J = 6; A = P -> F1.F2;

J = -50; B = P -> F1.F2;

assigns to A the value of the item situated 12 locations after AREA and to B the content of the location situated 44 positions before AREA.

2.3.4. programming example.

The following example defines a simple linked list structure. A cell consists of two words, the first one contains an integer information, the second one contains the relative position of the next element in the list. We have the following declarative statements:

```
COMMON ARRAY AREA (1000);
GLOBAL POINTER HEADER, POINTER TRAIL,
      INTEGER FORM INFORMATION ORIGIN (0),
      INTEGER FORM LINK ORIGIN (1);
```

We may define the following procedures.

```
PROCEDURE CREATION; /* CREATES AN EMPTY LIST */
  TRAIL, HEADER ->= AREA; HEADER -> LINK = 0;
  RETURN;
END CREATION;

PROCEDURE WRITE (INTEGER INF);
/* ADDS AN ELEMENT TO THE LIST */
  TRAIL -> LINK = 2; TRAIL = TRAIL + 2;
  TRAIL -> LINK = 0; TRAIL -> INFORMATION = INF;
  RETURN;
END WRITE;
```

```

PROCEDURE DELETE (INTEGER INF);
/* DELETES ALL CELLS WITH INF */
DECLARE POINTER P, POINTER Q;
P = HEADER;
LOOP I, @ = P- LINK;
    WHEN (@=0) RETURN; Q = P+I;
    IF (Q->INFORMATION - INF = 0) THEN P->LINK = I+Q->LINK;
        ELSE P = Q;
    FI;
END;
END DELETE;

```

3. EXPRESSIONS AND CONDITIONS.

3.1. expressions.

The general philosophy of P. L. Modular One implies that the compiler does not allocate core location without explicit directions from the programmer. Hence complex expressions should not be allowed as they would lead to an allocation of auxiliary space to store the intermediate results. There are several binary operators and shifts which all have the same precedence. Therefore $A - B * C$ implies $(A-B) * C$ unlike the usual convention. There must be some other restrictions to the syntax of expressions, not to allow the programmer to use expressions such as

$$@ = I + @;$$

which cannot be executed in a straightforward manner.

The syntax of an expression is very simple once it has been realised that a primary is whatever loads a register (i.e. a cons-

tant, a variable, a procedure reference or a register itself), and that a secondary is something which can be directly accessed in the address part of an instruction (i. e. a variable or a constant).

An expression is then:

$$[\text{unary-operator}] \text{ primary } \left[\begin{array}{c} \text{binary-operator} \\ \text{shift} \end{array} \right] \dots$$

A primary is a simple variable, a register variable, a stack reference, an array element, a structured element, a constant or a procedure reference. A secondary may be a simple variable, a stack reference, an array element, a structured element or a constant. A unary operator may be +, - or / (for logical NOT). A binary operator is any of the P. L. Modular One set: +, -, *, /, \ (remainder), & (logical AND), ! (logical OR). A shift is the shift operator (?) followed by an integer (for an arithmetic shift) or a bitstring (for a logical shift) constant.

The class of an expression is derived from its operands and operators. Some combinations of operands and operators, such as multiplying or dividing a label or a pointer, are invalid and the compiler will issue a warning message.

3.2. conditions.

Conditions are used in the construction of the conditional statements. The general format of a condition is:

(test)

where a test may be a Boolean expression, an overflow test or a comparison.

- Boolean expressions may be used in conditions:

WHEN (A ! B & C) GO TO L;

IF (A) THEN GOTO L; ELSE GO TO P; FI;

WHILE (TRUE)

- an overflow test may take any of the following three forms:

$$\left\{ \begin{array}{l} \text{AOV} \\ \text{BOV} \\ \text{MOV} \end{array} \right\}$$

which are reserved words of the language which stand for the overflow of registers A, B and M respectively;

- a comparison has the general format:

<integer-expression> <relation> 0

as the Modular One hardware is only able to test for zero. <relation> stands for any of the operators:

=	for equal
/=	for not equal
>	for greater
>=	for greater or equal
<	for less
<=	for less or equal

Hence:

(A+B = 0), (A < 0), (A+1 /= 0)

are valid conditions, but

(A+B > C+D), (A=3), (0 < A)

are not.

4. STATEMENTS.

P. L. Modular One statements can be grouped in eight different classes:

- . descriptive statements;
- . data movement and computational statements;
- . control statements;
- . conditional statements;
- . repetition statements;
- . data allocation statements;
- . machine interface statements;
- . debugging statements.

This classification has been defined only for description purpose. It has no fundamental significance in the language since some statements may belong to more than one of these classes.

4.1. descriptive statements.

The descriptive statements are used for describing the characteristics of the data by specifying attributes. The four descriptive statements are COMMON, GLOBAL, DECLARE and PROCEDURE. A different type of core allocation corresponds to each of these statements. These features will be discussed in 5.2.

4.2. data movement and computational statements.

A data movement involves the assignment of a value to a variable. There are two replacement statements.

- a- the assignment statement is identified by the assignment symbol (=). It has one of the two forms:

A = B;

A = B + C;

The first one is a simple data movement, the second one involves a computation. The left hand side variable may be any simple or register variable, a stack name, an array element or a structured element. The following are valid assignments:

@ = 6; I = \$ + 3; A(\$ + 7) = #;

P -> F1.F2 = L * 3 + G;

- b- the pointer replacement is identified by the pointer replacement symbol (->=) and is used to set a pointer to the address of a data element, an array or a stack.

P ->= A;

Both the assignment and pointer replacement statements may be used to assign the same value to more than one variable at a time. The multiple replacements have one of the following forms:

A, B, C = I + 2;

P, Q ->= AREA;

The DEQUEUE statement mentioned in 2.3.2. can also be considered as a data movement statement.

4.3. control statements.

The GO TO statement is generally used as an unconditional branch. If the destination of the GO TO is specified by a label variable, it may be used as a switch by assigning different label constants to the label variable. If the label variable is an array, the switch may be controlled by varying the index.

A procedure may be invoked by a CALL statement which names the procedure. The control is returned to the point of invocation by execution of the RETURN statement. The CALL statement also provides a means for initialising some of the arguments of the invoked procedure by specifying a list of values and references. The RETURN statement with a parenthesised variable or constant in a function procedure is used to return a value to a function reference.

The EXIT statement is used in association with composite statements and groups to pass control to the statement immediately following one of the containing statements. EXIT or EXIT *i* transfers control to the statement following the statement or group which is containing the EXIT. EXIT *i*, where *i* is an unsigned positive integer, passes control to the statement immediately following the *i*th containing statement.

4.4. conditional statements.

The WHEN and UNLESS statements provide the most common form of conditional branch and are generally used with a simple comparison or Boolean expression.

WHEN ($A < 0$) $A = -A$;

WHEN ($A \neq 0$) UNLESS ($B = 0$) $C = A/B$;

The IF statement is used in association with the FI statement to delimit the alternatives of a conditional statement.

IF ($A < 0$) THEN $X = 6 * A$; $Y = A + 20$;

ELSE $X = 6 / A$; $Y = A - 20$;

FI;

The CASE statement is a multiple choice branch depending on the value of a control variable. The control is transferred to the list of statements which is prefixed by the current value of the switch variable (see 1.2.4.).

4.5. repetition statements.

The LOOP statement indicates the imperative repetition of a group of statements. Control may pass out of the group by execution of a GOTO statement or of an EXIT statement. The following LOOP statement could be used to compute $N!$

```
A = N; L = 1;
```

```
LOOP L = L * A; A = A - 1; WHEN (A = 0) EXIT; END;
```

The WHILE and UNTIL statements indicate the conditional execution of a group of statements and its repetition while or until a condition is met. Control may pass out of the group by execution of a GOTO statement, of an EXIT statement or by the occurrence of the tested condition. The preceding example could have been coded in any of the two following ways:

```
A = L; L = 1;
WHILE (A > 0)
L = L * A; A = A - 1;
END;
```

```
A = 1; L = 1;
UNTIL (A - N > 0)
L = L * A; A = A + 1;
END;
```

4.6. data allocation statements.

The ALLOCATE and FREE statements allow the programmer to specify allocation of controlled variables and the freeing of the space which has been previously allocated. The user has a full control on the allocation and the freeing of the storage.

Multiple allocations of the same variable cause the stacking of the generations of variables. The `FREE` statement deletes some allocated variables and makes the space available for further allocations. A more complete discussion of these features is made in 5.3.

4.7. machine interface.

These are the most machine dependant features of the language. The `COMMAND` statement calls standard Executive functions or services. More generally, it is the way of communication with the internal rings of the "onion ring" design (see III - 3.3.). The `CODE` statement provides a means of entering assembly-like instructions in a P. L. Modular One program. The `ENTER` statement is used to insert an hexadecimal constant in the object code. The `SAVE` statement saves the context of a procedure which may be reset by the execution of a `RESET` statement.

4.8. debugging statements.

P. L. Modular One provides a comprehensive set of debugging statements which produce a dynamic trace of the execution of the program. The `TRACE` and `STOP TRACE` statements delimit a section of the program in which the tracing may be initiated. The `WAKE` and `SILENCE` statements respectively activate and deactivate the tracing process. When the trace is active, the program produces a printed record of the result of all assignments, conditional statements and switching statements.

5. PROGRAM STRUCTURE AND STORAGE ALLOCATION.

5.1. program structure.

A P. L. Modular One program consists of one or more blocks of statements called procedures and of a common section of code called main segment. Procedures may invoke other procedures or call themselves recursively. The structure of a P. L. Modular One program is as follows:

```

PROGRAM program-name [MAIN] ;

    { global and common declarations }

PROCEDURE proc-1 [arguments] ;

    { procedure body }

END [proc-1] ;
.
.
.
END [proc-n] ;

    { main segment }

END [program-name] ;

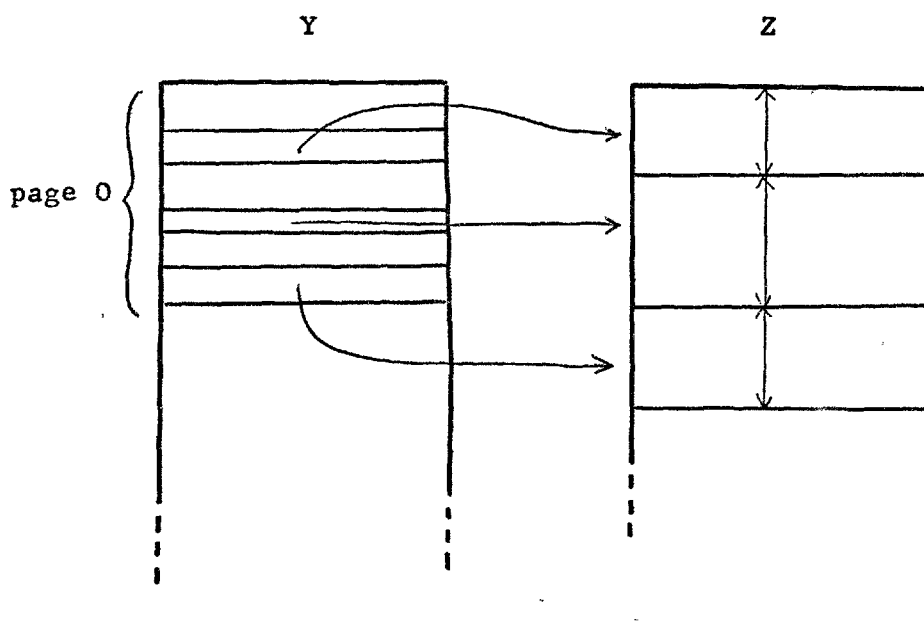
```

5.2 declarations and storage allocation.

There are four different manners of specifying the characteristics of a name. Each one of these manners implies a different scheme of allocation and different scopes for the variables which it defines.

5.2.1. common declarations.

Simple variables, arrays and stacks may be declared common to two P. L. Modular One programs. This is achieved by means of a COMMON statement at the beginning of both programs. Common data are allocated space in the Z segment of core, but the data descriptors (indirect words) are set in the first page of segment Y.



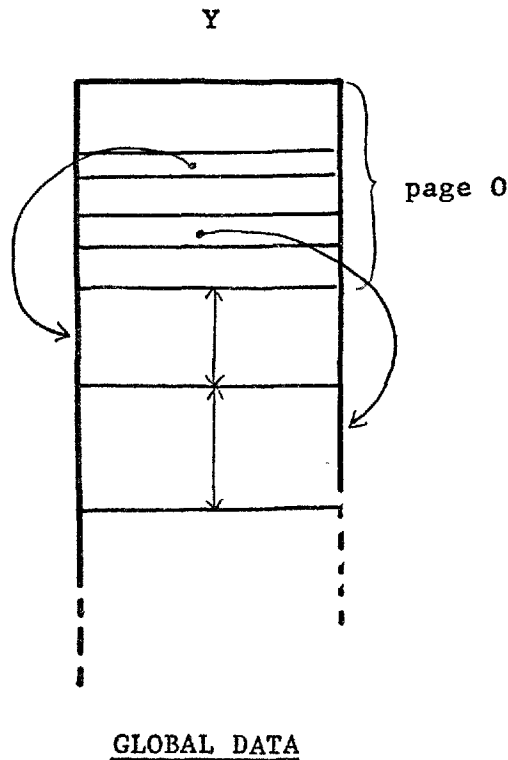
COMMON DATA

Common names remain known throughout the program and can be referenced at any point of it. The sharing of common data between two P. L. Modular One programs is made by the sharing of their Z segments.

5.2.2. global declarations.

Global declarations associate an area of global storage with a name. The data descriptors are set in the first page of

segment Y and the space is allocated sequentially in the rest of segment Y.



The GLOBAL statement must appear at the beginning of the program, it is a non executable statement, therefore global arrays and stacks cannot have adjustable sizes. Global data can be statically initialised by means of an INITIAL attribute.

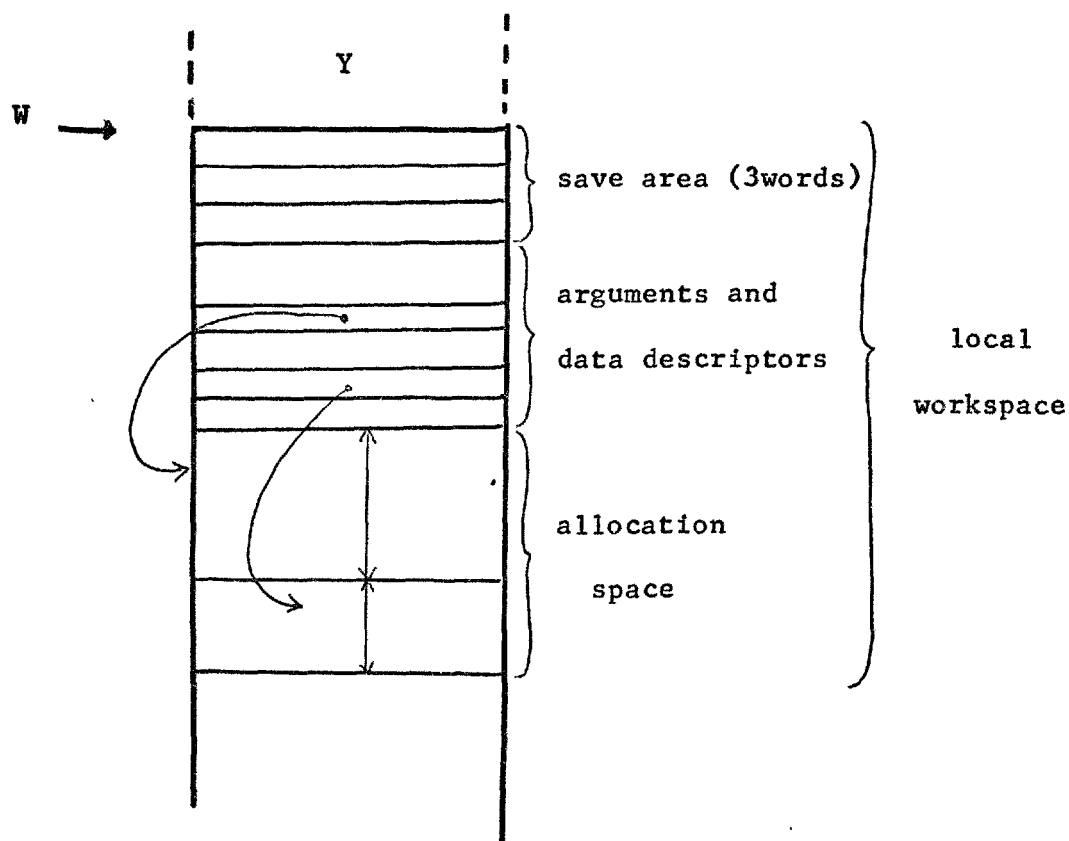
5.2.3. local data.

Declarations local to a procedure are made by means of the DECLARE statement. Any type of declaration may appear in a DECLARE statement, but the names which are declared in that statement are unknown outside the procedure.

Local data are allocated space in the local workspace of the procedure. This allocation is dynamic, i. e. instructions

are generated by the compiler which are executed each time the procedure is invoked. On execution of a RETURN statement, the whole workspace is freed and all the information contained in local data is lost.

The local workspace of a procedure is pointed by register W (the relocation register) and is set up as follows:



LOCAL DATA

The save area contains the following information:

- the return address of the procedure;
- the value of the W of the invoking procedure;
- the upper bound of the local workspace.

The arguments and data descriptors are allocated space contiguously at the bottom of the local workspace. The local data are set sequentially in the allocation space which lies in the top

part of the local workspace.

5.2.4. argument declarations.

Argument declarations define local items which may be initialised on a reference to the procedure. They also provide a means for passing parameters to a procedure. The argument declarations appear after the procedure name in the PROCEDURE statement.

example:

```
PROCEDURE SCAL(INTEGER I, REFERENCE TO INTEGER ARRAY
               A, REFERENCE TO INTEGER ARRAY B,
               REFERENCE TO INTEGER J);

  I = I - 1; J = 0;
  WHILE (I = 0) J = A(I)*B(I) + J;
  I = I - 1;

  END;

  RETURN;

  END SCAL;
```

This procedure, when invoked:

```
CALL SCAL(L, *V1, *V2, *VALUE);
```

computes the scalar product of the two vectors V1 and V2 (call by reference) of dimension L (call by value) and puts the result in the variable VALUE (call by reference), the value of L is left unchanged.

5.3. controlled allocation.

A variable that bears the CONTROLLED attribute is allocated space only upon the execution of an ALLOCATE statement specifying that variable name. The CONTROLLED attribute may appear with simple variables, stacks and arrays in GLOBAL and DECLARE statements. A controlled variable is stacked: that is, storage may be allocated even when a previous allocation for that variable exists. At any given moment, one can only access the latest allocation, however, one may keep a trace of the previous allocations by use of pointers.

The ALLOCATE statement extends the global or local workspace of the amount of space required for storing the specified controlled variable and sets the variable descriptor to the address of that space. The FREE statement decreases the local or global workspace of the specified number of locations. On completion of a RETURN statement in a procedure, all the controlled variables allocated in the procedure are lost.

CHAPTER V

THE EXECUTIVE

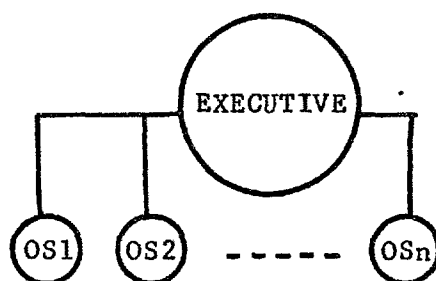
The Executive is the kernel of the "onion ring" design described in chapter III - 3.3. Its content is essentially depending on the hardware and on the interrupt and special state mechanisms. This chapter gives the general outlines of an Executive and of its interface with a System Programming Language. As an illustration, details are given on the way in which such an Executive can be implemented on Modular One to interface P. L. Modular One.

1. GENERAL STRUCTURE.

1.1. characteristics and general philosophy.

The overall structure of the Executive has been dictated by five major requirements.

- a- It should be only an Executive and should not prevent the development of any type of operating system with varieties of tasking and scheduling mechanisms and core management techniques.



- b- The Executive should be provided with an interface which allows the programming of all the rest of the software in a high-level language. That implies that all the parameters passed to the Executive must be compatible with the SPL data items.
- c- The Executive must support applications with a high rate of input/output operations such as the programmable terminal software. Therefore, as little code as possible must be written in special (non interruptible) state.
- d- A mechanism for efficiently driving the peripherals and processing the external interrupts must be built in, but some freedom must be left as to the way in which the peripherals are allocated or controlled.
- e- A simplified elementary tasking and scheduling system must be included to allow parallel processes, but this should be limited to a minimal number of basic primitives to permit any type of scheduling in the design of the supported operating systems (priority, queues,...).

1.2. The SPL context.

The P. L. language includes a statement which is compiled as a switch to some dedicated location of the Executive. Hence:

COMMAND 4;

would result in a switch to the context contained in location 4 of the Executive which corresponds to some predefined type of action or service.

Registers can be explicitly set to any P. L. constant or variable and explicitly tested on completion of the Executive call. For example:

```

DECLARE POINTER PT;
.
.
.
PT -> = A;
@ = PT; # = 6;
COMMAND INPUT;
WHEN (@ = 0) GOTO ERROR;

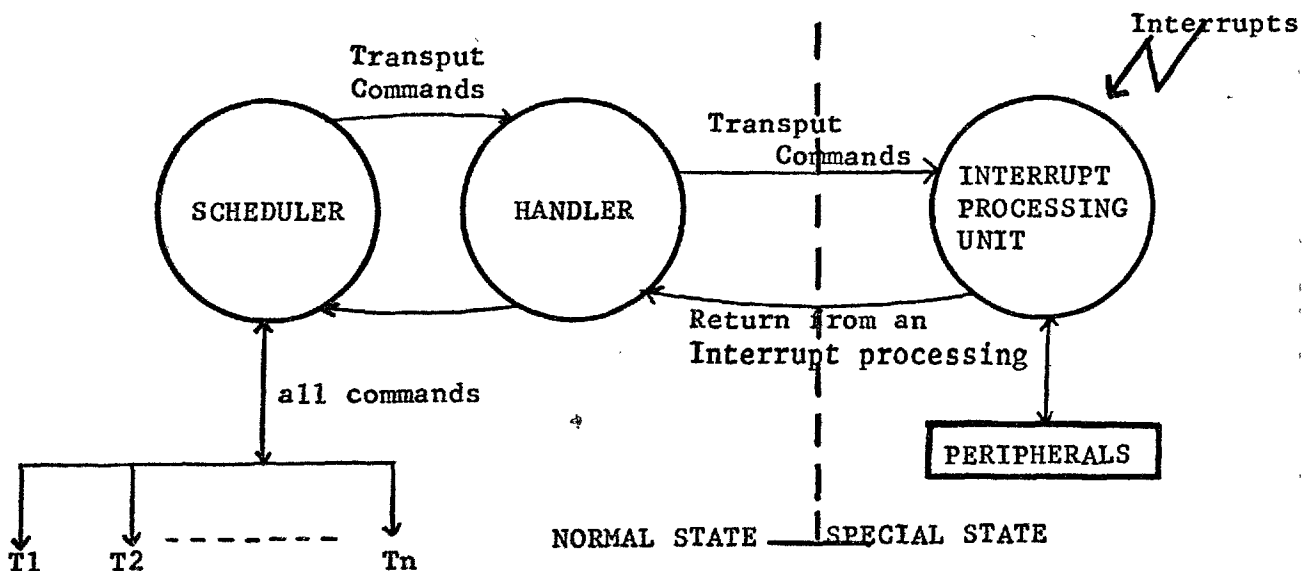
```

could be a typical sequence of P. L. instructions required for an Executive invocation.

1.3. structure of the Executive.

The Executive consists of three major modules.

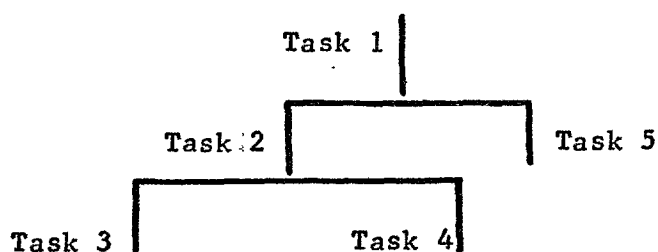
- the Scheduler controls the tasking mechanism;
- the Handler drives the peripherals;
- the Interrupt Processing Unit deals with the external interrupts.



The Scheduler is entered each time that a COMMAND statement is issued from a task. It despatches the Input/Output (Transput) commands to the Handler and processes the other commands. The Scheduler manages a list of Task Control Blocks describing the processes active in the system. The Handler is entered each time that a transput command is issued. On completion of the processing of an interrupt, control is also passed to the Handler. The Handler controls a list of Input-Output Control Blocks which describe the current transput operations. The Interrupt Processing Unit is entered at each interrupt. Its main role is to associate an interrupt with a peripheral and the corresponding Input-Output Control Block.

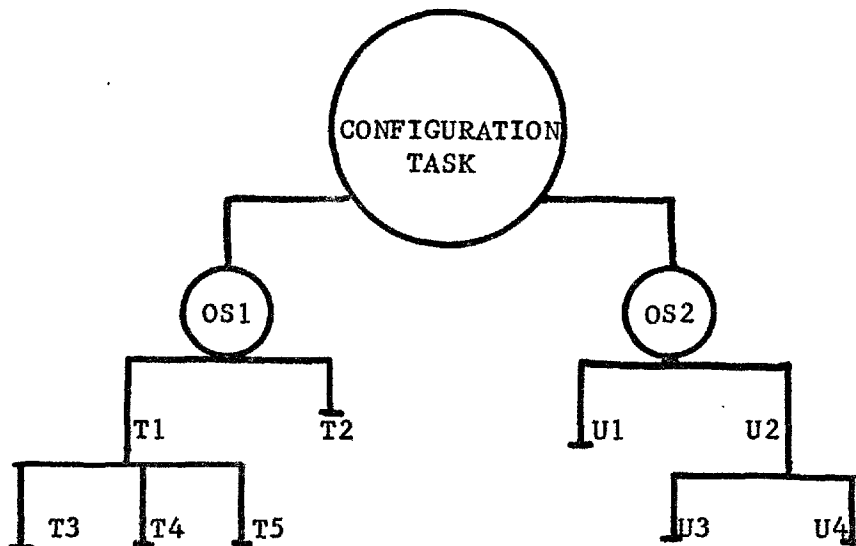
1.4. tasks.

A task is a process which may have or gain control of a certain number of resources (peripherals, core space, processor time ...). A task may initiate other tasks which are hierachical-ly dependent of the initiating task. Thus one can build a tree which describes this dependence relation.



The root task is generally an operating system which controls the allocation of the resources between all its dependant sub-tasks. One can also imagine a construction with two or more operating systems, where the root task is acting as a configu-

ration task, allocating both peripherals and core space to the various operating systems.



1.5. protection of the tasks and I-O operations.

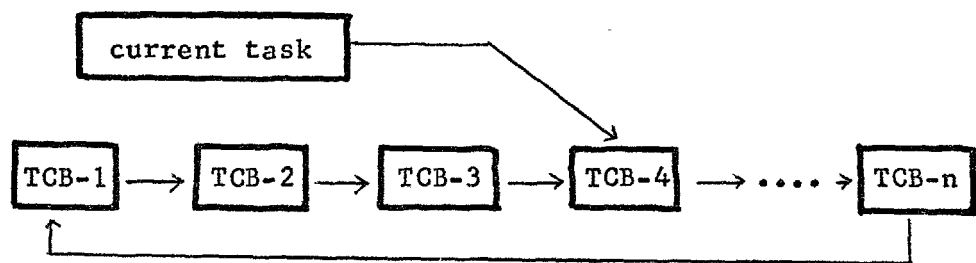
All the tasks in the system are identified by a number which is created by the scheduler at the time of the construction of the associated Control Block. This number is returned to the attaching task and must be produced at each reference to the attached task or I-O operation. The attaching task can pass this identification number to all its sub-tasks which may communicate with these I-O or tasks, and prevent the other ones from having access to this identification key.

1.6. scheduling mechanism.

In the absence of a real time clock, the scheduling of a new task can only be made when special conditions occur. These may be either interrupts or Executive calls. The re-scheduling of a task could take place at any of these occurrences. However, this mechanism involves the execution of an important number of

instructions, some of which are in special state. It would be detrimental to the performances of the I-O system if this sequence had to be executed each time that an interrupt occurs. As a consequence, the scheduling of a new task will be made only when the Executive is invoked via a COMMAND statement.

The Scheduler controls a circular list of Task Control Blocks.



When a task in control issues a call to the Executive, this call is served, then the status of the task saved in its TCB and the Scheduler tries to re-initiate the task associated to the next TCB in the list.

2. SCHEDULING COMMANDS.

2.1. Task Control Blocks (TCB).

A task control block is associated to each task and summarizes its main features at any given time. The Scheduler has an unlimited access to the list of all TCBs and may modify them on execution of a command.

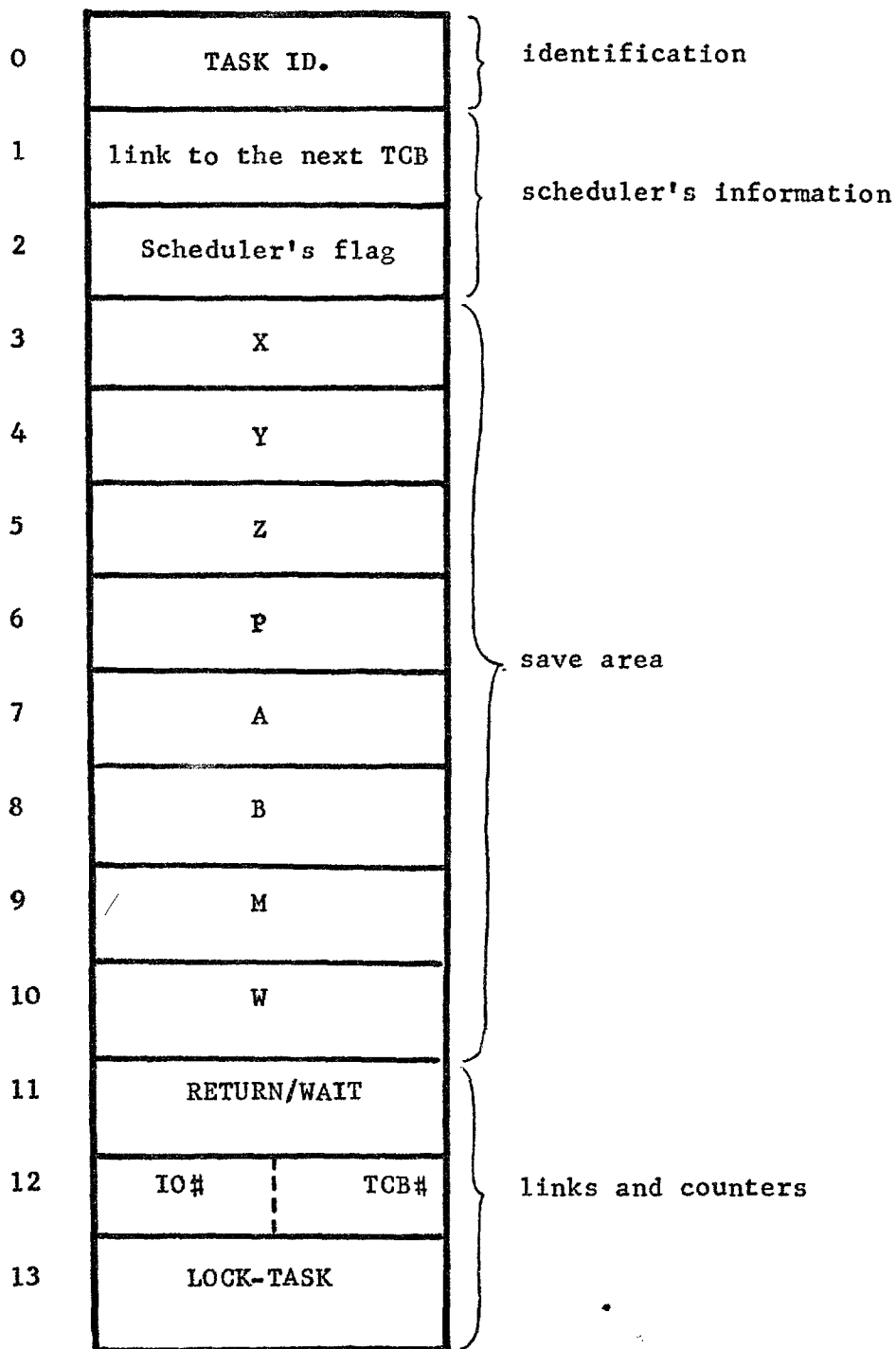
The format of a TCB is dependent on the implementation machine, however it must contain four major classes of information:

- a- the identification of the TCB;
- b- Scheduler's information;
- c- a save area;
- d- links and counters for the attached and attached

ching tasks.

On Modular One, a TCB consists of a block of 14 words with the following layout.

- word 0 - a number in the range [1 : 255] which identifies the task. This number is generated by the CREATE A TCB command and is used to refer to the TCB and its associated task.
- word 1 - link to the next TCB in the Scheduler's circular list.
- word 2 - Scheduler's flag - this identifies the current state of the task with the following code.
 - 0 - the task is active and can be scheduled;
 - 1 - the task is unactive (e.g. no yet activated);
 - 2 - the task is waiting for the completion of a transput operation;
 - 3 - the task is waiting for the completion of an attached sub-task.
- words 3-4-5 - X - Y - Z - these are the save areas for the hardware registers X, Y and Z.
- words 6 to 10 - P - A - B - M - W - save areas for the P, A, B, M and W when the execution of the task is suspended.
- word 11 - Return/Wait - this field is only used when the task is in a wait state (scheduler's flag is 2 or 3).
 - . if the task is waiting the completion of a sub-task, this field contains the TCB identification of the task of which completion is awaited;



A MODULAR ONE TASK CONTROL BLOCK

- if the task is waiting for the completion of a transput, the field contains the identification of the Input Output Control Block in question.

word 12

- IO#, TCB# -

bits 15 - 8 contain the number of transput operations currently attached to the task;
bits 7 - 0 contain the number of sub-tasks currently attached to the task.

word 13

- Lock/task - this field links backwards to the task which activated the task. It contains the actual attaching task's identification if the attaching task is waiting for its completion (ACTIVATE AND WAIT command), or the opposite of that number if the execution of the two tasks can be made in parallel (ACTIVATE AND GO ON command).

2.2. the CREATE A TCB command.

This command selects a TCB from the Scheduler's list of unused TCBs and links it to its circular list. The state of the task is set to "not yet activated" (state 1), and the identification number allocated to the TCB is returned to the calling task. This number must be presented each time that one will refer to the created TCB or its associated task. The command may be rejected if there is no more space available in the Scheduler's list. In that case an error code is returned.

example:

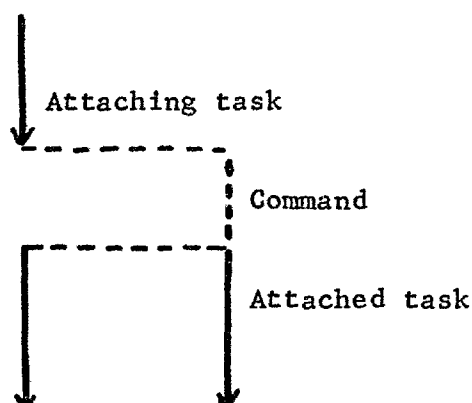
```
TRYAGAIN: COMMAND CREATEATCB;
WHEN(@= 0) GOTO TRYAGAIN; /* ERROR CODE */
TCBID = @ ; /* SAVE THE TCB IDENTIFICATION */
```

2.3. the KILL A TCB command.

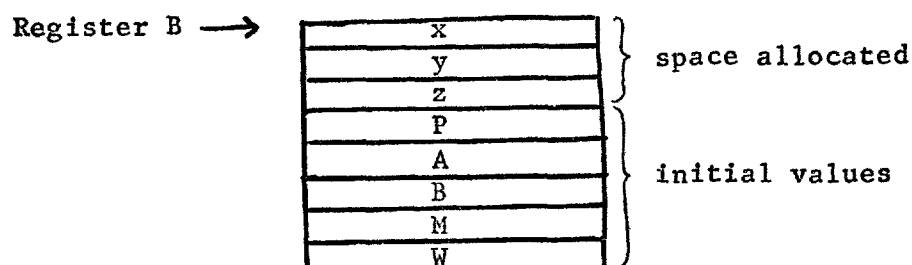
This command transfers a specified TCB from the Scheduler's list to the unused TCB list. This transfer can only take place if the specified TCB is unactive. Any other attempt will result in an error code.

2.4. the ACTIVATE AND GO ON command.

The ACTIVATE AND GO ON command attaches a sub-task to the current task. On completion of the command, the control is returned to the attaching task which may continue in parallel with the attached task.



The attaching task must produce the identification number of a TCB which has been previously created and will be associated to the attached task. The initial values of the registers and the space allocated to the sub-task must also be passed as parameters to the Executive. For example, on the Modular One, Register A will contain the TCB identifier of the attached task and Register B will point to an 8 words long area set as follows:



ATTACHING TASK'S
T.C.B.

T1	
....	
....	
....	
....	
....	
....	
....	
....	
....	
....	
....	
...	increment-
	ted by 1
....	

ATTACHED TASK'S
T.C.B.

0	T2
1	
2	set to 0
3	↑ These fields are set to values derived from the ones specified in the area pointed by Register B ↓
4	
5	
6	
7	
8	
9	
10	
11	set to 0
12	set to 0 set to 0
13	set to -(T1)

The fields x, y and z define the offsets and the extents of respectively the segments X, Y and Z of the attached task as follows:

- bits 15 - 14 indicate the segment of the attaching task which is to be allocated partially to the attached task (1 is X, 2 is Y, 3 is Z, 0 is no segment in the sub-task).
- bits 7 - 0 indicate the page offset in the designated segment of the attaching task at which the segment of the sub-task begins.
- bits 13 - 8 give the length (in page) of the segment.

2.5. a programming example.

The attaching task contains a part of a loader which loads object code in its Z segment as follows:

- XBASE, YBASE, ZBASE are respectively the first pages of the X, Y and Z segments of the loaded program;
- XL, YL, ZL denote the lengths (in page) of the three segments of the loaded program;
- ENTRY is the address of the entry point of the program.

To pass control to the loaded program, Register B will have to point to the following area.

3	XL	XBASE
3	YL	YBASE
3	ZL	ZBASE
ENTRY		
0		
0		
0		
0		

x	}	space allocated
y		
z		
P	}	initial values
A		
B		
M		
W		

This can be achieved by the following P. L. Modular One piece of program:

```

DECLARE WORD ARRAY A(8), POINTER PT;
.
.
.
/* CREATE A TASK CONTROLBLOCK */
TRYAGAIN: COMMAND CREATEATCB;
        WHEN (@ = 0) GO TO TRYAGAIN;
        TCBID = @;

/* SET X, Y & Z OF THE LOADED PROGRAM */
PT ->= A;
A(0) = XL ? 8 ! XBASE ! @C000;
A(1) = YL ? 8 ! YBASE ! @C000;
A(2) = ZL ? 8 ! ZBASE ! @C000;
WHEN (XL = 0) A(0) = 0; WHEN (YL = 0) A(1) = 0;
WHEN (ZL = 0) A(2) = 0;

/* SET THE REST OF THE CONTEXT */
A(3) = ENTRY; A(4), A($+1), A($+1), A($+1) = 0;

/* ISSUE THE COMMAND STATEMENT */
@ = TCBID; # = PT;

COMMAND ACTIVATEANDGOON;
.
.
.

```

2.6. the ACTIVATE AND WAIT command.

The ACTIVATE AND WAIT command activates a sub-task and suspends the execution of the attaching task until the completion of the attached task occurs.

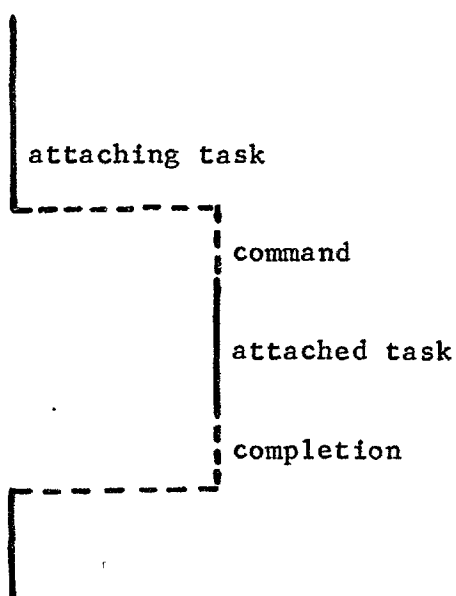
ATTACHING TASK'S
T.C.B.

T1	
....	
set to 3	
....	
....	
....	
....	
....	
....	
....	
....	
....	
set to T2	
....	inremen- ted by 1
....	

ATTACHED TASK'S
T.C.B.

0	T2
1	
2	set to 0
3	 These fields are set to values derived from the ones specified in the area pointed by Register B
4	
5	
6	
7	
8	
9	
10	
11	set to 0
12	set to 0 set to 0
13	set to T1

THE ACTIVATE AND WAIT COMMAND



The set-up required for this command is similar to the one needed for the ACTIVATE AND GO ON command.

2.7. the RETURN command.

The RETURN command terminates the activity of the current task and returns the control to the attaching task which is waiting for its completion (if any). The TCB of the task is set to "not yet activated" and can be subsequently deleted by a KILL A TCB command or reactivated by any of the two ACTIVATE commands.

The RETURN command can be executed only when the task has no sub-task and no transput operation currently attached to it (both the IO# and TCB# fields in the TCB must be zero). If this is not the case, the command is delayed until these two conditions are met.

2.8 the STATUS TASK and SUSPEND commands.

The STATUS TASK command gives information on the current status of a specified task. This status can be any of the following:

- task is active;
- task is completed or not yet activated;

- task is waiting for the completion of a transput;
- task is waiting for the completion of a sub-task.

The SUSPEND command is a dummy command which forces a return to the scheduler, thus allowing other tasks to be scheduled. This command may be used in a task which is long and makes little or no use of the executive commands to interrupt its activity and allow other parallel tasks to be activated.

3. TRANSPUT COMMANDS.

3.1. generalities.

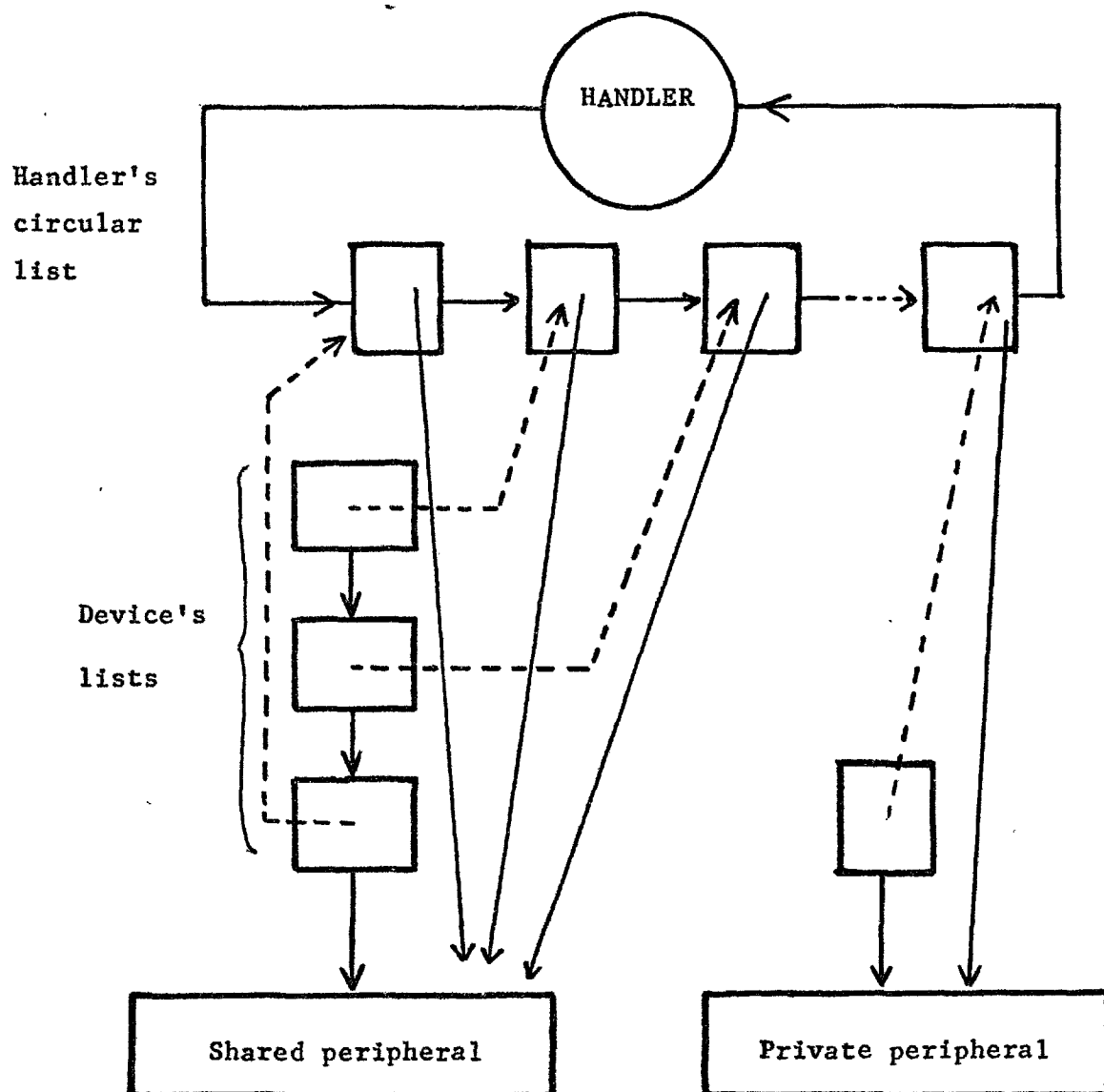
Prior to any input-output operation, a task has to create an Input-Output Control Block (IOCB) which gives access to the actual peripheral device. An IOCB has in relation to the Handler, a role similar to a TCB in relation to the Scheduler.

One must make a clear distinction between two major categories of devices.

- a shared device can be accessed by more than one task at any time. Transput demands are queued and served on a first-in-first-out basis.
- a private device can be accessed by only one task at any moment. Any transput on this device will be rejected as long as the peripheral is busy.

In terms of IOCBs, this means that, at any time, only one IOCB can be associated to a private device, as more than one can be active for a shared device. For each peripheral, the Handler creates a list of the IOCBs which are currently accessing the peripheral. A private device's list contains only one element. A shared device's list can

contain any number of element within the limits of the implementation.



The notion of transput may be more general than a simple input-output operation. For example, the interrupt key of a teletype can be considered as a private device exchanging binary information with the task which has taken control of it (i. e. which has created an IOCB specifying that device).

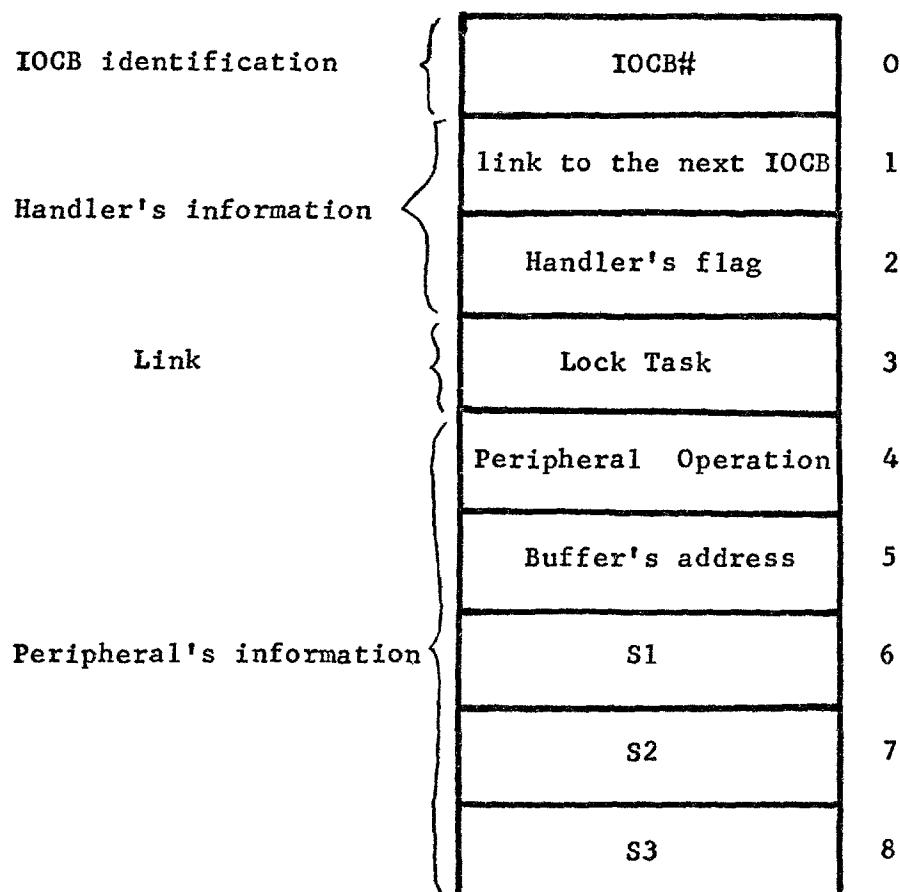
3.2. structure of an IOCB.

An IOCB must contain the following information:

- the identification of the IOCB;
- Handler's information;
- links to the attaching task;
- peripheral information.

A Modular One IOCB consists of a block of 9 words with the following layout.

- | | |
|---------------------|--|
| <u>word 0</u> | - a number in the range [1:255] which identifies the IOCB. This number is created by the CREATE AN IOCB command and is used to refer to the transput operation associated to the IOCB. |
| <u>word 1</u> | - links to the next IOCB in the Handler's circular list. All created IOCBs are linked in a way similar to the linking of the TCBs in the Scheduler's list. |
| <u>word 2</u> | - Handler's flag - this identifies the current status of the transput operation with the following code: <div style="margin-left: 40px;"> 0 - executing transput;
 1 - inactive;
 2 - waiting for peripheral availability
 (for shared peripherals only). </div> |
| <u>word 3</u> | - Lock Task - same use as the Lock Task field of a TCB. |
| <u>word 4</u> | - bits 15 to 8 identify the peripheral;
bits 7 to 0 identify the operation (input, output, input with echo, special operation). |
| <u>word 5</u> | - the buffer's address (if any). |
| <u>words 6 to 8</u> | - peripheral dependent information such as track address, codes etc... These fields must be set by the user when he initiates the operation. |



A MODULAR ONE INPUT OUTPUT CONTROL BLOCK

3.3. the CREATE AN IOCB command.

The CREATE AN IOCB command selects an IOCB from the Handler's list of unused IOCBs and links it to the circular list. The state of the transput is set to "inactive" and the identification number allocated to the IOCB returned to the invoking task. The command may be rejected in any of the three following circumstances:

- A- no more IOCB is available in the Handler's free list;
- B- the specified peripheral device is unknown or off line;
- C- the specified device is a private device and is already assigned to another task.

Programming example:

```
# = PRINTER; /* SET B TO THE DEVICE TYPE */
TRYAGAIN: COMMAND CREATEANIOCB;
WHEN (@ = 0) GO TO TRYAGAIN; /* REJECTION A */
WHEN (@ < 0) GO TO ERROR; /* REJECTIONS B & C */
IOCBID = @; /* SAVE THE IOCB IDENTIFICATION */
```

3.4. the KILL AN IOCB command.

The KILL AN IOCB command transfers an inactive IOCB from the Handler's circular list to the unused IOCB list. This transfer can only take place if the specified IOCB is "inactive" (status 1). Any other attempt will result in a rejection of the command.

3.5. the TRANSPUT AND GO ON and TRANSPUT AND WAIT commands.

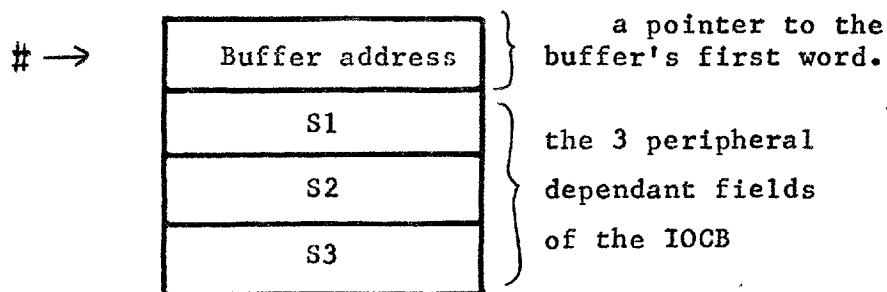
These two commands are similar to the ACTIVATE AND GO ON and ACTIVATE AND WAIT commands. They initiate a transput operation while continuing or suspending the attaching task. On Modular One, the following parameters must be passed to the Handler:

- Register A contains:

in bits 15-8 the identification of the IOCB
of the transput operation.

in bits 7-0 the type of operation.

- Register B contains a pointer to a 4 words area
set up as follows:



3.6. the STATUS IO command.

The STATUS IO command returns information on the current status of a specified transput operation. It is similar to the STATUS TASK command.

PART II

THE SYNTAX DESCRIPTION LANGUAGE

CHAPTER VI

STRUCTURE OF A TRANSLATOR

The modern methods of compilation of high-level programming languages such as ALGOL or FORTRAN are characterised by an increased formalisation of the syntactical structures of the language and an highly efficient translation process. Most compilers which have been designed using these techniques have been implemented on large scale computers, where the problems of core space limitations are not crucial. The following chapters describe a general model for compiler writing and its associated syntax description language, which is especially adapted to small size installations where the major limitation is the core size.

1. TERMINOLOGY AND NOTATIONS.

Most compilers or translators can be functionally divided into modules acting as co-routines exchanging parametric information. These modules are:

- feeding of a new input record;
- formatting and buffering of the input records;
- lexical analysis;
- searching and entering an item in the symbol table;

- syntactical analysis;
- semantical actions;
- code generation;
- etc...

The object of this chapter is to present a general model of compiler and the way of systematically coding these modules. It appears that some co-routines are rather machine-independent and thus can be generated by special purpose programs. Some other ones are strongly dependent on the language and classifications must be set between the languages. The proposed model can be used for compiling languages as different as BASIC, FORTRAN, COBOL, ALGOL or PL/I and should be valid, with very slight modifications, for the writing of any compiler.

Throughout this chapter, the following terminology will be used:

lexical types (τ): the different classes of elementary constructions which form the language (i.e. names, integers, reals, strings, special characters, etc...);

analytical type(θ): a refined lexical type, making distinctions between the different types of words and names;

reserved word language: a language in which a variable and a statement identifier cannot be the same (COBOL or ALGOL);

keyword language: a language in which a statement identifier can be used for naming variables (PL/I or FORTRAN);

lexically separated

language: a reserved word language in which the lexical types of the statement identifiers and of the

variables are different (most ALGOL implementations);

lexically free language: a keyword language where spaces have no significance and may appear inside names and identifiers (FORTRAN);

feeding module(Φ): the module of a compiler which reads a record from the source text;

preprocessing module(Π): the module of a compiler which performs simple formatting functions on a record which has just been read in;

lexical analyser($\wedge$): the part of the compiler that builds the lexical types from the input stream;

lexical parameter(λ): a parameter which is passed to the lexical analyser to control its action;

scanner(Σ): a generic name for the feeding module, the preprocessing module and the lexical analyser;

symbol module(Ψ): the module of a compiler that enters and searches items in the symbol table;

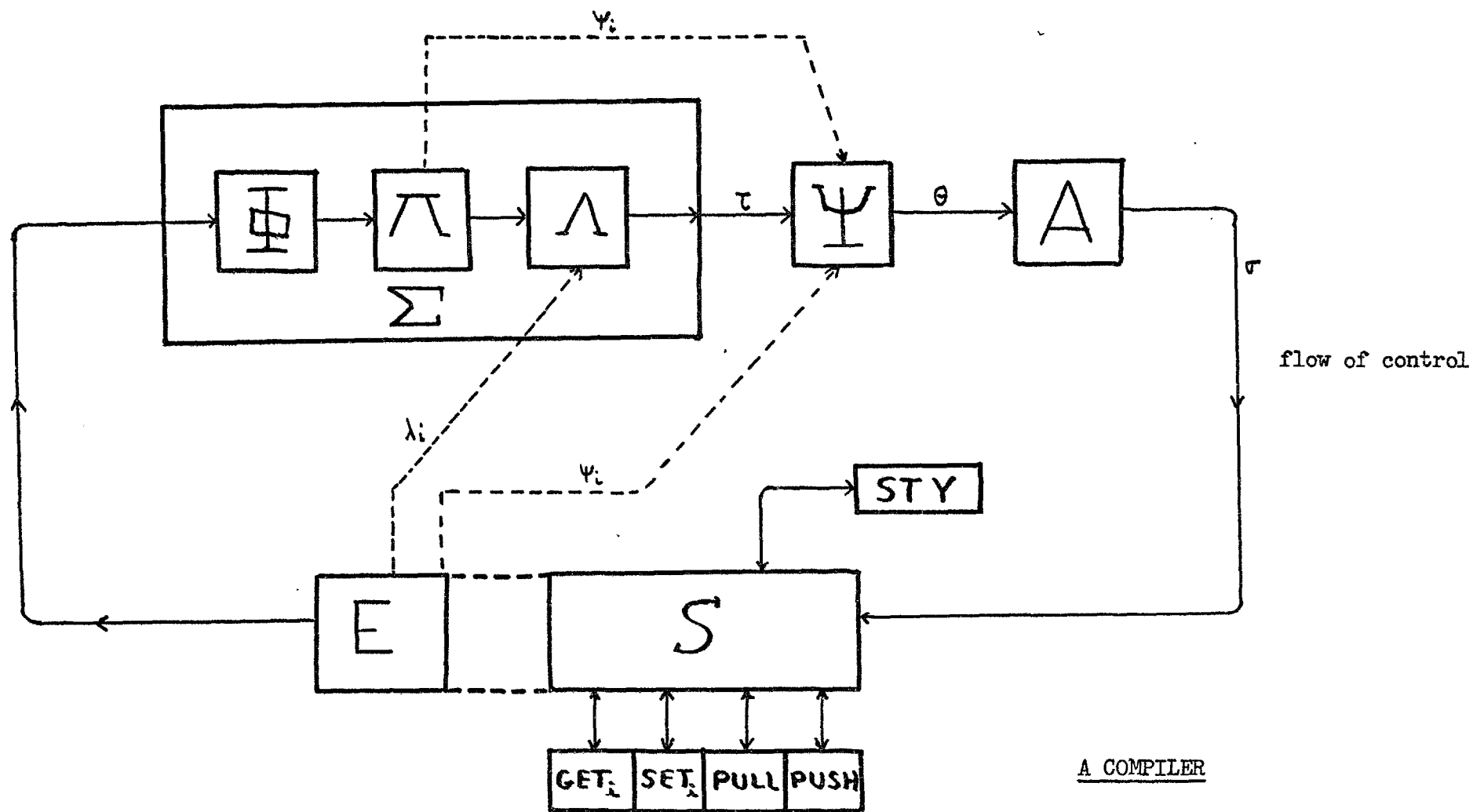
symbol parameter(ψ): a parameter which is passed to the symbol module to control its action;

analyser(A): the module of a compiler that performs the syntactical analysis of the source text;

action(τ): the address of a piece of code which is executed each time that a specific syntactical configuration is encountered; by extension: the piece of code itself;

semantical module(S): the set of all actions;

epilogue(E): a module appended to the semantical module which sets



A COMPILER

lexical and/or symbol parameters;

service: some commonly used piece of action;

reader(GETi): a service which reads a piece of information from the
symbol table;

writer(SETi): a service which enters a piece of information in the
symbol table;

typing(STY): a service which associates an analytical type with a
name;

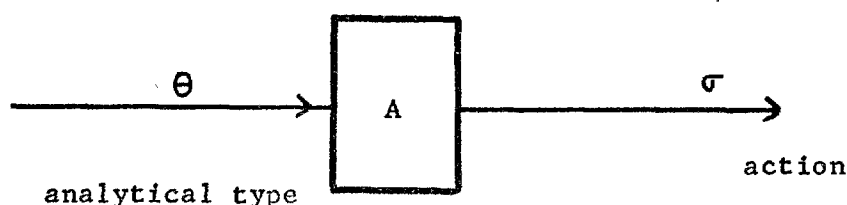
blocking-in(PUSH): the service which is invoked at the entry into
a new block of statements (in the ALGOL sense);
it defines a new lexical level in the symbol
table;

blocking-out(PULL): the service invoked at the end of a block; it
deletes the latest lexical level in the symbol
table.

2. THE ANALYSER.

The analyser is the heart of a compiler, it lies half-way
between the stream of the input characters and the sequence of
instructions which are executed depending on this input.

The analyser communicates up-stream with a set of modules
which produce a code describing the class of the symbol which is
currently processed (analytical type θ). It produces the address
of a sequence of instructions which is used down-stream by the
semantical module (action σ).

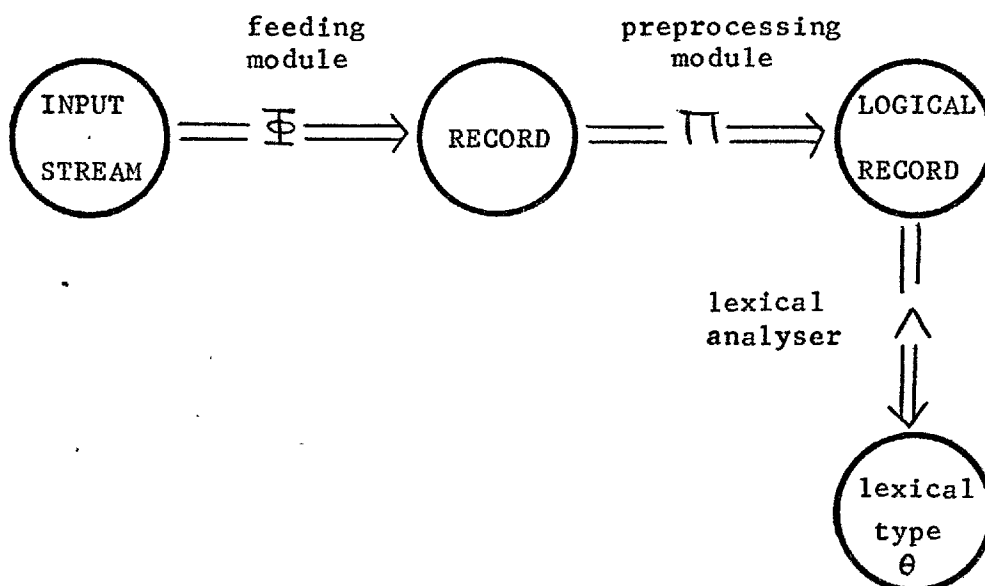


As a consequence, the analyser is entirely language-independent and a general description system can be designed which automatically generates the syntactical analyser for the described language. Chapter VII presents a system of this type and its associated language (SYntax DEscription LANGUAGE).

3. THE SCANNER.

The scanner can be functionally divided into three sub-modules which may be physically interleaved:

- the feeding module reads a record from the source text;
- the preprocessing module performs elementary operations on the source records (buffering, compression of spaces, etc...) to produce a logical record. It may also collect useful information concerning the structure of the input stream;
- the lexical analyser builds the elementary syntactical units of the language (lexical classes) and produces a lexical code.



3.1. the feeding and preprocessing modules.

The structure and function of these modules depend very much on the type of language which is analysed. Two types of strategies can be developed.

3.1.1. reserved word languages.

This is the simplest case, where the preprocessing module may be omitted or reduced to a very simple function. The logical record will be the same as the input stream record, the feeding module just reads the physical record and sets it in the logical record buffer which is submitted to the lexical analyser. The preprocessing module may consist in very simple functions such as:

- setting an end mark after the last significant item of the input record;
- compress multiple spaces in a single one.

A new record will be read in each time the lexical analyser asks for a character which is beyond the last one present in the current logical record buffer.

3.1.2. keyword languages.

The fundamental lexical ambiguity of the keyword languages is in the fact that the first word of a statement gives no indication on the nature of the statement. For example, in FORTRAN, one can write:

DO6 I=1,N

and DO6 I=1

which are both valid statements with completely different meanings. A possible strategy (see Chapter VIII, 2.) consists in making a

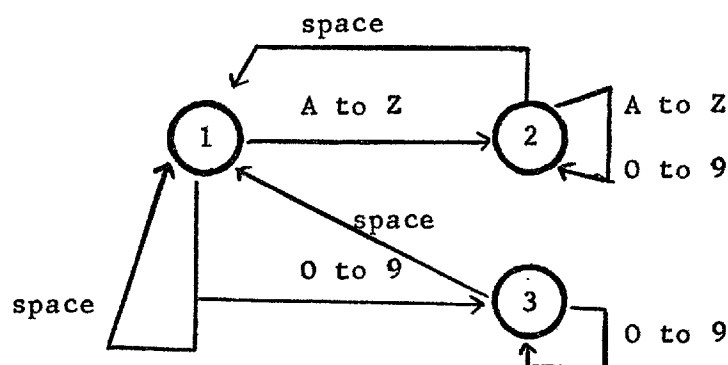
fairly simple analysis of the whole record before the syntactical checking. In this case, the logical record will have to be at least one complete statement and the preprocessing module will perform the simple scan of the statement to determine its type.

3.2. the lexical analyser.

The lexical analyser is the critical loop of a compiler; more than 75% of the processor time is spent in it. The analysis algorithm must then be as fast as possible, even if it is detrimental to its size in core.

3.2.1. syntax.

A lexical analyser can be represented by a finite state automaton which can be visualised in a diagram. For example, in a language consisting only of names (FORTRAN type) and unsigned integers, the lexical automaton will be:



This automaton can be coded into a transition matrix $T(i,j)$, where i is the current node of the diagram, j the character which is input and T the new node. In the preceding example a (3,37) matrix would be required. However, this can be reduced by taking into account the fact that some of the columns of the matrix are identical. Characters with identical entries in the matrix can be

assigned the same code by some transcoding function (generally tabulated), thus reducing the overall size of the matrix.

In the preceding example, the matrix could be compressed using the transcoding function:

$$S(X) = \begin{cases} 1 & \text{if } X \text{ is a space} \\ 2 & \text{if } X \text{ is a letter from A to Z} \\ 3 & \text{if } X \text{ is a digit from 0 to 9} \end{cases}$$

The transition matrix then becomes:

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 2 \\ 1 & \text{X}1 & 3 \end{pmatrix} \begin{matrix} \text{--- node 1} \\ \text{--- node 2} \\ \text{--- node 3} \end{matrix} \quad (\text{X} = \text{error})$$

$\downarrow$
space

$\downarrow$
A-Z

$\downarrow$
0-9

3.2.2. semantics.

The lexical analyser must also be able to produce information on the lexical classes which are analysed. These information may consist of:

- a lexical code describing the class;
- the value of constants and strings;
- information on names (hash value, length, characters..)
- etc...

These information are of semantical type; therefore it is necessary to associate to each arc an action which is executed each time that the arc is traversed. This can be done by means of a matrix $U(i,j)$, similar to T , where $U(i,j)$ gives the number of the action to be performed when the corresponding arc is traversed.

3.3. lexical parameters.

3.3.1. lexically free languages.

In FORTRAN, as in all lexically free languages, spaces may appear freely inside syntactical units, or may be omitted between two words. Hence one can write any of the following:

D 010I=1,7

DO10I = 1, 7

DO 1 0I = 1,7

etc...

When analysing this sentence, the scanner must have some information on the type of the statement which is currently processed so that it can be able to isolate the first unit.

The lexical analyser should therefore have a special mode of processing which recognizes the leading keyword of a statement. This can be achieved by testing a Boolean variable "if-it-is-an-assignment" set by the preprocessing module, and comparing the leading characters with a table of the keywords.

3.3.2. other languages.

In the other languages, there is no real ambiguity at the lexical level; a syntactical unit being separated from the next one by some kind of separator (brackets, dots, commas, spaces,...). The lexical analyser is then entirely deterministic and needs only the next character of the input stream to determine whether it is the end of a syntactical unit or not.

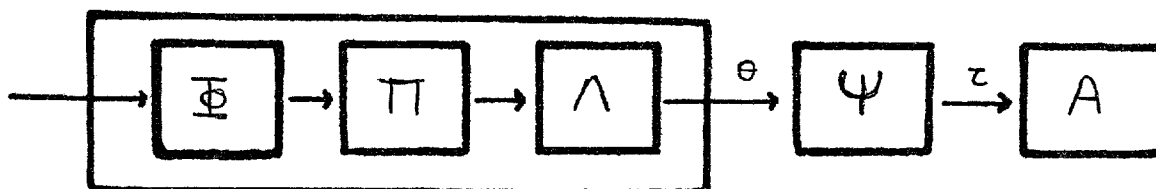
However, it may be necessary, for semantical purposes, to make a distinction between lexically identical constructions. For

example, in ALGOL (or in FORTRAN or BASIC), a label may have the same syntactical form as an unsigned integer constant. For semantical reasons one wishes to consider a label as an identifier (to enter it in the dictionary, etc...) and not as a constant. The scanner must then have an information flag to be able to distinguish between these two lexically identical forms. This is achieved by means of a lexical parameter which is set to the proper value in the appropriate context (generally in a semantical action).

A lexical parameter is a mode of interaction between the semantics and the lexical analyser. However, as it may considerably slow down the analysis, it must only be used when it is absolutely impossible to discriminate in a purely semantical manner.

4. THE SYMBOL MODULE.

The symbol module stands between the scanner and the syntactical analyser.

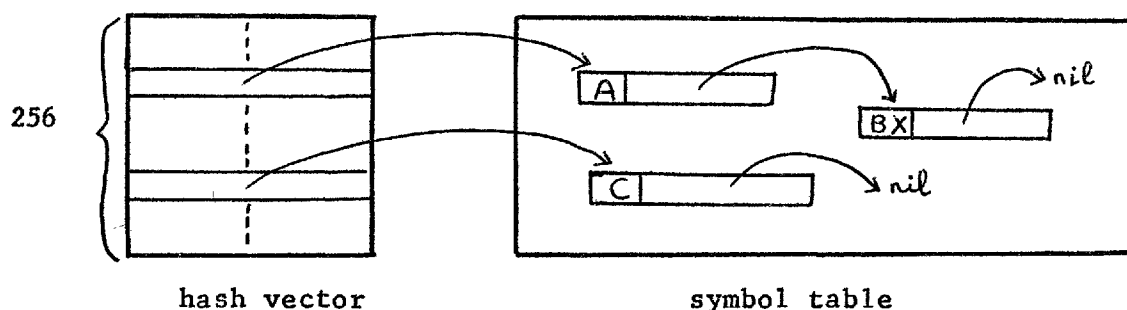


Its main purpose is to refine a lexical type into an analytical type by consultation of the symbol table. The symbol module is executed only when the θ resulting from the scanner stage is one of the codes which can be expanded. The expansion then consists in replacing the lexical type 'word' by the code of an individual analytical type such as 'MOVE keyword' or 'simple integer variable' or 'label'.

4.1. the symbol table.

The symbol table (or dictionary) of a compiler is space consuming, therefore, preference must be given to a compact storage method. This however should not penalise too much the access time to any given item in the table.

Hashed organisations seem to be the most interesting ones. A hash function is computed for each symbol (say the sum of the internal representations of each character modulo 256). A vector (the hash vector) gives the address of the first element with the given hash value. All entries corresponding to a same hash value are linked on a list. The format of the symbol table is then as follows:



Each entry has the following format:

hash chain link	TYPE	LENGTH	sym....bol	informa.....tion
-----------------	------	--------	------------	------------------

The consultation of the table and the entry of a new symbol are made by means of the SEARCH and COPY procedures, of which examples are given in Appendix V.

In the analysis and compilation of block structured languages, the symbol table is used as a stack (LIFO), special procedures are used for defining a new lexical level (blocking-in -- PUSH)

and clearing the table at the end of a block (blocking-out -- PULL). Examples of these procedures are also given in Appendix V. In most cases the keywords and reserved words of the language may be considered as variables of lexical level 0 and can be entered at the base of the symbol table.

The scope of variables in block structured languages suggests that, on a given hash chain, the last entered symbol must be the first one. This also simplifies the coding of the PULL, PUSH, SEARCH and COPY procedures.

4.2. analytical types.

An analytical type is the refinement of a lexical type. For each occurrence of a word in the source stream, the symbol module consults the symbol table and tries to map it with a known symbol. An analytical type is associated to a symbol in the dictionary by means of a typing procedure invoked in a semantical action. When the symbol does not exist in the table, it is copied in it, and a neutral analytical code is associated to it.

A proper use of the neutral and analytical codes may solve the problem of declaratives and type-checking at the syntactical level. This feature will be discussed in more details in Chapter VIII, 3.2.

4.3. symbol parameters.

A symbol parameter is a piece of information passed to the symbol module to indicate how the SEARCH and COPY procedures must be invoked. The two following examples illustrate the use of two symbol parameters.

- example 1: "if-it-is-an-assignment"

The Boolean variable "if-it-is-an-assignment" mentioned in 3.3.1. is a symbol parameter as it indicates that none of the words of the current statement are keywords. Therefore, the symbol module will not attempt to search for a name under a certain level in the symbol table.

- example 2: "level"

In block structured languages, one must specify at which lexical level the current name must be searched in the dictionary. Three possible situations can be found:

- a keyword is expected, one must search for the name at lexical level 0 in the table;
- inside a declarative statement, one must check that the name has not been already defined at the current lexical level;
- in every other case, a variable is referenced, the table must be searched at all lexical levels (excluding level 0).

This piece of information can be passed to the symbol module in the variable "level" which acts as a symbol parameter. Appendix V gives an example of the SEARCH procedure using the variable "level" as a parameter.

5. THE SEMANTICAL MODULE, THE EPILOGUE AND THE SERVICES.

5.1. the semantical module.

The semantical module is the set of all the actions associated to the syntactical configurations of the analysed language. These

actions are of various types, including the generation of the object code and the validation of some constructions which cannot be checked at the syntactical stage.

This module is highly machine-dependent; therefore it is difficult to give general rules on its coding. Examples in the next two chapters illustrate possible semantical actions and their relations with the syntax.

5.2. the epilogue.

The epilogue consists of the set of the actions (or part of actions) which interact with the other modules. These interactions are made by means of the lexical and symbolic parameters.

The epilogue is often included as part of the semantical actions, and the distinction between the epilogue and the rest of the semantics has no logical support. However the epilogue is functionally different from the rest of the actions as its consequences are generally felt outside the semantical module itself.

To clarify the programming and ease the debugging of the compiler it is strongly recommended to concentrate as much as possible of the epilogue in a few independent self-contained actions.

5.3. the services.

Although the coding of the semantical actions is a language dependent matter, it often appears that some commonly used pieces of code are entirely independent of the languages compiled. These pieces of code may be advantageously set into general procedures which are invoked when necessary.

- the TYPING procedure is used to associate an analytical type with a name, it is generally used to replace the neutral code by a more appropriate one;
- the SETi procedures are procedures which set values in the various sub-fields of the information field of a given symbol table entry;
- the GETi procedures access an information sub-field for a specified symbol table entry;
- the PUSH and PULL procedures mentioned before can also be considered as services.

Examples of programming of some services are given in Appendix V.

CHAPTER VII

"SYDELA"

"SYDELA", short for Syntax Description Language, is a machine independent system used to describe the syntax of programming languages. The grammar is expressed in terms of a certain number of transition diagrams. Each diagram acts as a finite state acceptor which recognizes derivations from certain non-terminal symbols. The "SYDELA" analyser is table driven; the tables are compact and can be easily processed by a micro-programmed routine as well as by conventional in-line code.

1. TRANSITION DIAGRAMS.

1.1. the original model.

In 1963, M. E. CONWAY described (7) a compiler where the syntax of the source language was expressed by means of a number of transition diagrams.

For example the following simple grammar:

$$\begin{aligned}\langle \text{Term} \rangle &::= \langle \text{Primary} \rangle \mid \langle \text{Term} \rangle * \langle \text{Primary} \rangle \\ \langle \text{Primary} \rangle &::= (\langle \text{Expression} \rangle) \mid \langle \text{name} \rangle \\ \langle \text{Expression} \rangle &::= \langle \text{Term} \rangle \mid \langle \text{Term} \rangle + \langle \text{Expression} \rangle\end{aligned}$$

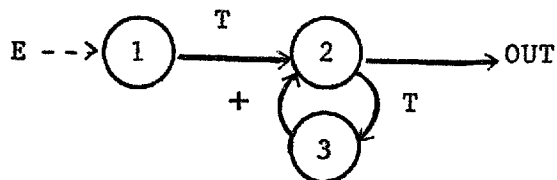
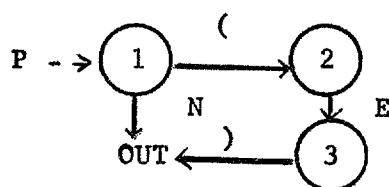
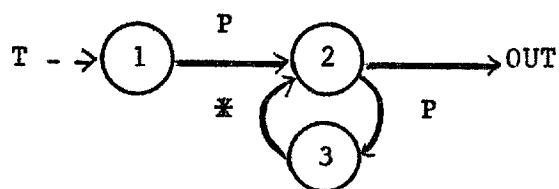
can be written in form of recursive expressions as follows:

$$T \rightarrow P \{ * P \}^*$$

$$P \rightarrow (E) \mid N$$

$$E \rightarrow T \{ + T \}^*$$

which in turn can be described by the following diagrams:



The algorithm for processing the diagrams is given below.

step a: set 'current-diagram' to the first diagram; set

'current-node' to node 1; set the input pointer
to the first character of the input stream;

step b: examine all the arcs leaving 'current-node' in

'current-diagram' which have a symbol name on them;
if there is a match between this label and the current symbol in the input stream then traverse the arc (i.e. set 'current-node' to the node pointed by the arc) and go to step f; otherwise go to step c;

step c: try to traverse the remaining labelled arcs leaving 'current-node'; each of these arcs corresponds to a transition diagram: the diagram will be successfully traversed if it is possible to go from the entrance to the exit of that diagram; this is attempted by stacking 'current-node' and 'current-diagram' and going to step b for the first node of the corresponding diagram (i.e. setting 'current-diagram' to the label, 'current-node' to node 1 of that diagram and going to step b);

step d: if there is no match in step c and there is an unlabelled arc leaving the current node then follow it and go to step g; else go to step e;

step e: the analysis is at a dead end.

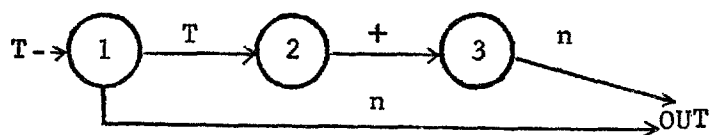
pop up the stack, reposition the input pointer to the symbol present when the current diagram was entered and try another path in step c;

step f: advance the input pointer to the next symbol in the input stream;

step g: if the current node is an 'OUT' then pop up the stack and return to the diagram and node at the top of the stack and traverse the arc corresponding to the diagram which has been tried, then go back to step g; if the new node is not an 'OUT' then go to step b for this new node.

1.2. the loophole problem.

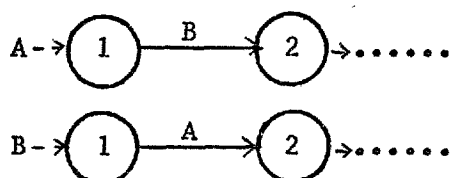
The following diagram:



which corresponds to the expression:

$$T \rightarrow T + n \mid n$$

and more generally:



which is derived from the set of equations:

$$A \rightarrow B \dots\dots$$

$$B \rightarrow A \dots\dots$$

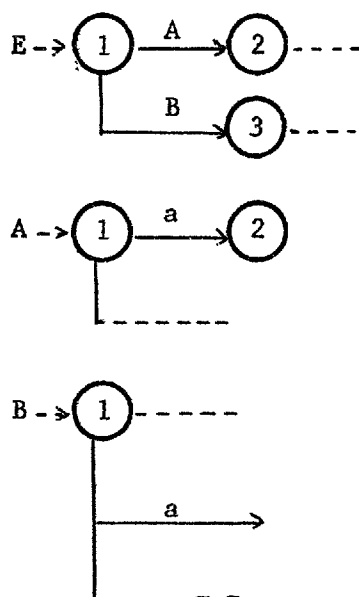
where a transition diagram makes a reference to itself (directly or indirectly) without having first read a symbol, result in an infinite loop. This situation will generally occur when the corresponding regular expression includes a left recursion (direct or indirect). V. TIXIER has demonstrated (18) the equivalence between Conway's transition diagrams and non-left-recursive functions of regular expressions.

1.3. the back-up problem.

When two arcs leading from a node are labelled by non-terminal symbols, and the two diagrams defining these syntactical entities both have arcs leading from the entrance with the same terminal symbol, step e of the algorithm will be executed for some

configuration of the input stream:

example:



Moreover the result of the analysis may depend on the order in which the arcs leading from a given node are examined.

As it is not practicable to reposition the input pointer backwards, one has to eliminate the possible causes requiring a backing up of the input stream. A set of no-back-up conditions is:

- a- there is no loophole of the kind described in 1.2.;
- b- for every node the sets of initial input symbols of all the arcs emanating from that node are distinct (the "follow sets" are distinct).

Most programming languages can be analysed using no-back-up diagrams; this may require some considerable transformation of the grammar as described in B.N.F. or some other description method. This effort of rewriting will generally be rewarded by a simplification of the analyser and a greater ease in inbedding the semantical actions in the syntax.

2. GENERALISED TRANSITION DIAGRAMS.

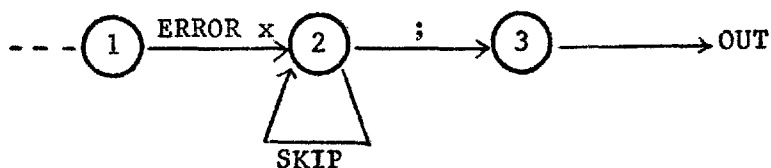
2.1. semantical actions.

The representation of the syntax of programming languages by transition diagrams is well adapted to syntax directed compilation. One can associate to each of the arcs a specific semantical action which is executed each time that the arc is traversed. The semantical actions may generate the intermediate or object code, check the semantical validity of the constructions and build the tables and vectors used in the compilation. The figure on next page illustrates the use of semantical actions to generate the polish equivalent of an unfixed arithmetic expression.

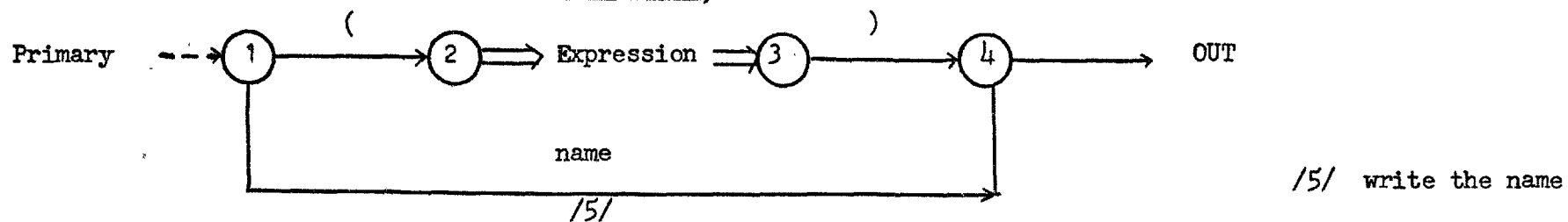
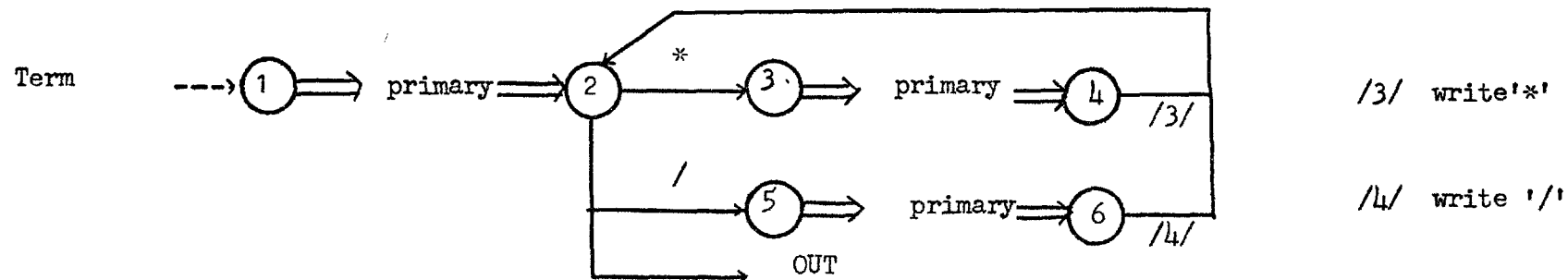
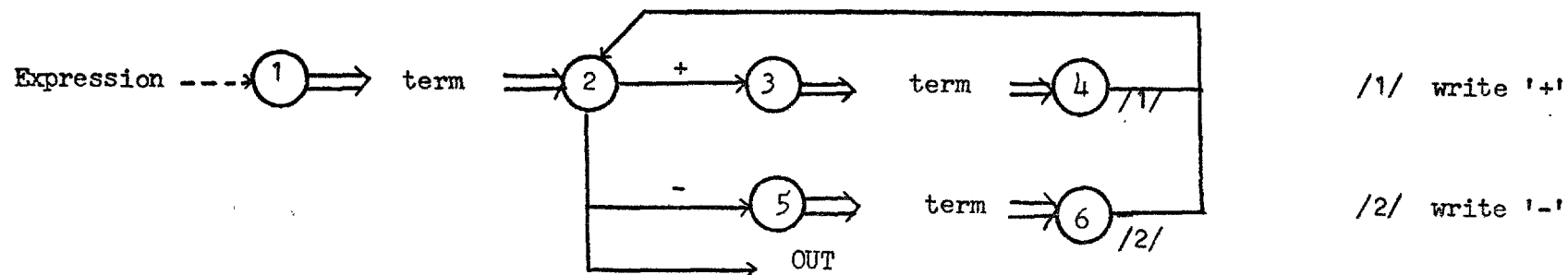
The recovery of syntactical errors in the source text, which allows the continuation of the analysis after an invalid construction has been found, and the printing of diagnostics and warnings are also part of the semantical actions. However these may be expressed in the syntax by means of two special transitions.

- the "SKIP" transition skips the current syntactical unit, irrespective of its type;
- the "ERROR" transition prints a diagnostic which is associated to the current syntactical unit;

The following diagram can be used to recover syntactical errors in an ALGOL-like language: all units are ignored until a semi colon is found.



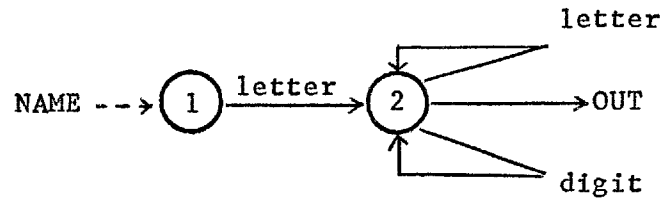
{ diagnostic x reads: "source text by-passed until
next semi-colon" }



GENERATION OF THE POLISH NOTATION CORRESPONDING TO AN INFIXED
EXPRESSION (7)

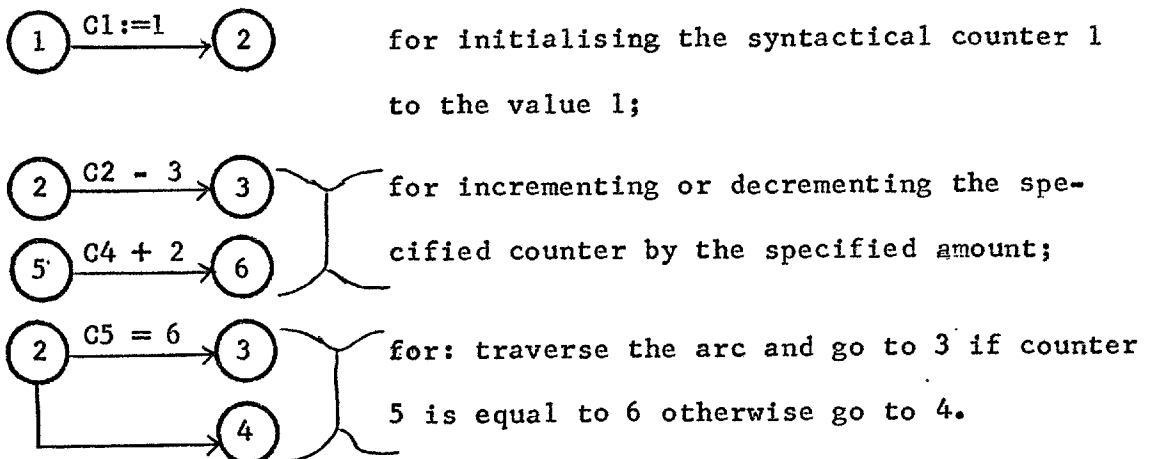
2.2. counter transitions.

The transition diagrams as defined by Conway do not allow the user to specify a maximum, minimum or exact number of occurrences of a syntactical unit.. Hence the diagram:

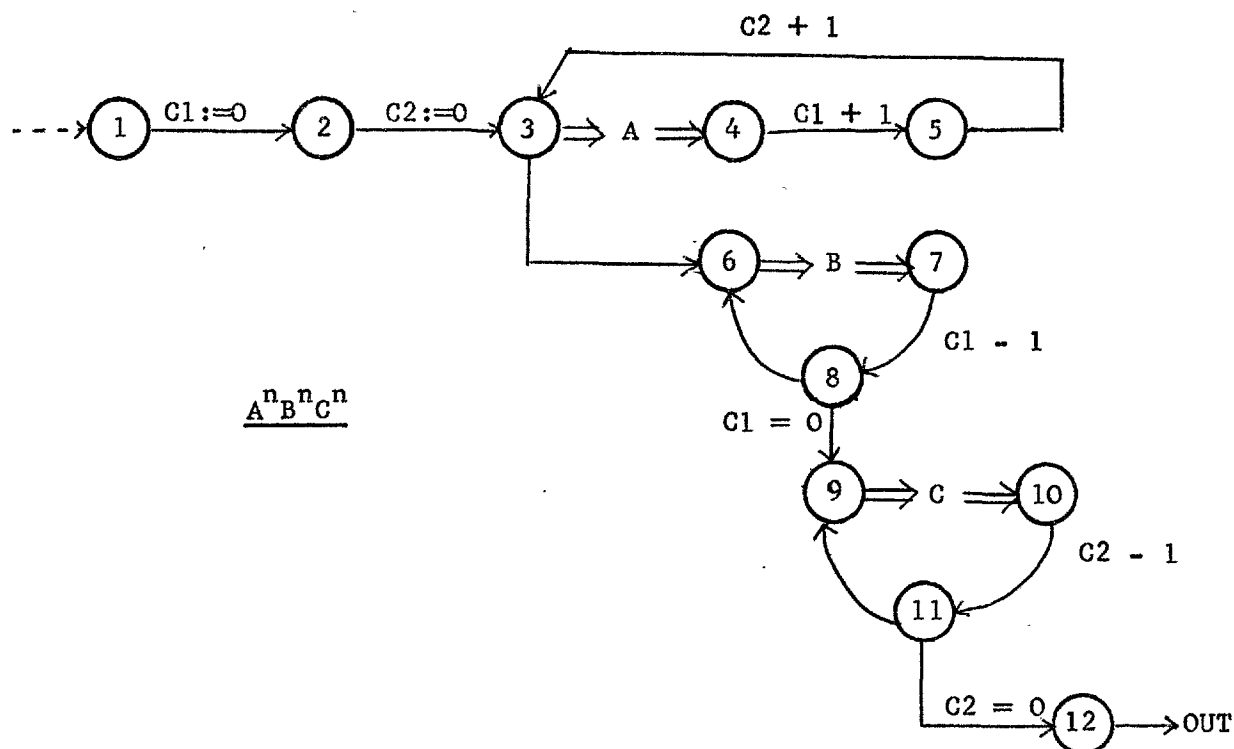


does not limit the total number number of occurrences of "letter" or "digit".

The counter transitions are special types of transition which set, test, increment or decrement a certain number of syntactical counters. One will write:

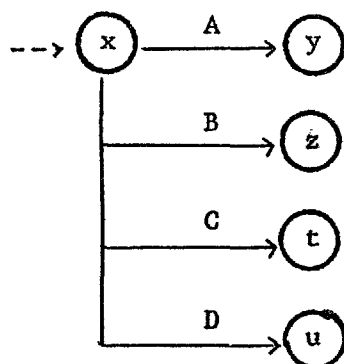


It is intuitively obvious that these transitions enlarge the set of the languages which may be analysed by transition diagrams. For example the context sensitive construction $A^n B^n C^n$ can be analysed by the following diagram.



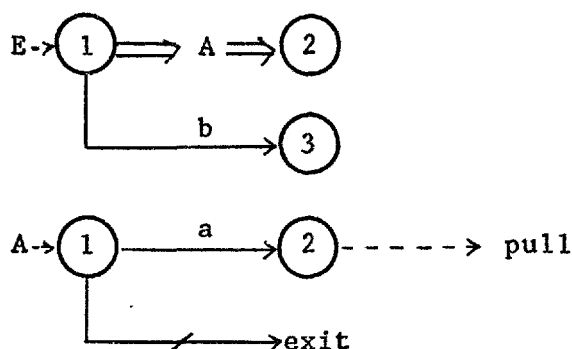
2.3. the "EXIT" and "PULL" transitions.

In the initial model, one can draw diagrams such as:

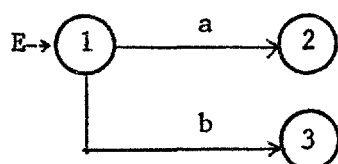


which provides a conditional decision depending on the label of the arcs. These arcs are examined successively (in an implementation defined order) until a match is found. However one or more of the arcs may be labelled by non-terminal symbols which imply the invocation of a new diagram. This diagram can be accepted (traversed) or rejected; in the later case, the next arc in the invoking diagram will be analysed.

The "EXIT" transition is the rejection output of a diagram, the "PULL" transition is the acceptance one. For example the following construction:



is equivalent to:



2.4. secondary entries.

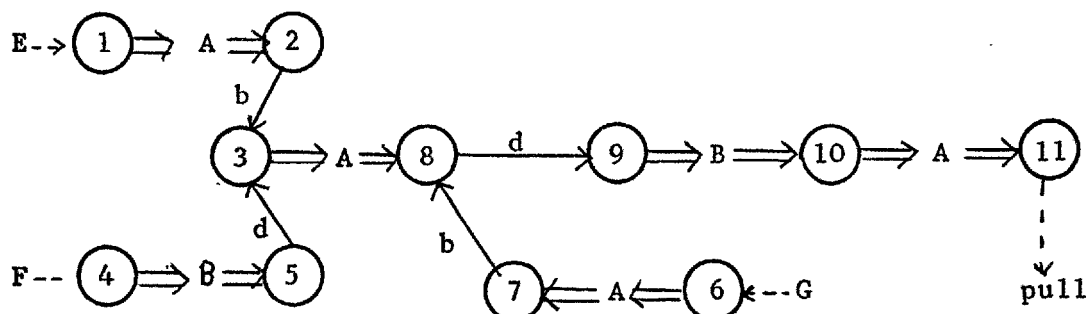
When two or more syntactical constructions differ only by some initial symbols, it is possible to draw a single diagram to describe them, using secondary entry points to refer to some of these constructions. For example the constructions:

E AbAdBA

F BdAdBA

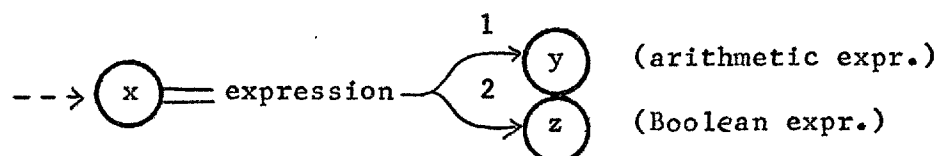
G AbdBA

can be described by the single multiple-entry diagram:

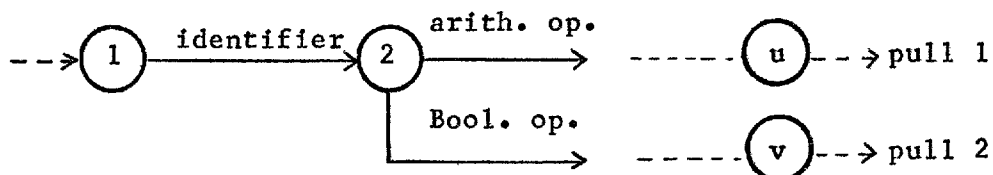


2.5. multiple output diagrams.

One of the no-back-up conditions states that two diagrams invoked from a given node should not begin with the same terminal symbols. In some cases however this may rise some difficulties. For example, in FORTRAN, both an arithmetic and a Boolean expression may begin with an identifier. One can only decide of the type of the expression when all its operators have been analysed. It would be useful to invoke a single diagram which will decide in due time of the type of the expression.



Such a diagram could be drawn as follows:



Examples of use of multiple output diagrams for the solution of potential back-up situations are given in Chapter VIII - 4.

3. "SYDELA".

3.1. generalities.

"SYDELA" is a language for the description of transition diagrams and their processing. A "SYDELA" compiler accepts as its input a linear description of the diagrams and the symbols used to label the arcs, and produces as its output various tables which can be submitted to the standard "SYDELA" analyser. "SYDELA" can be

used as a preprocessor which generates tables in a format suitable to any given language. Details on both the analyser and the tables produced can be found in Appendix V.

A "SYDELA" program consists of two parts. The declarative section defines the basic symbols of the analysed language. The statement section describes the transition diagrams.

3.2. declaratives.

"SYDELA" declaratives fall in three categories: terminal declarations, name declarations and synonyms (or class) declarations.

3.2.1. terminal declarations.

The terminal declarations define the basic lexical symbols of the language which are not keywords or reserved words. The format of a terminal declaration is:

```
* 'identifier' = ?
```

where "identifier" may be any string of characters used to identify the terminal symbol. The compiler allocates an internal (analytical) code to this identifier. In a subsequent part of the "SYDELA" program one will refer to the terminal symbol by the notation:

```
* 'identifier'
```

The following are valid terminal declarations:

```
* ';' = ?
```

```
* 'NAME' = ?
```

```
* ':= ' = ?
```

3.2.2. name declarations.

The name declarations are used to introduce the keywords or reserved words of the language. The format is:

```
** 'name-1'  [, 'name-i']  ....
```

The compiler allocates an analytical code to each of these names and enter them in the symbol table at lexical level 0.

The following declaration could be used to introduce some of the COBOL reserved words:

```
** 'PROCEDURE', 'DATA', 'ENVIRONMENT', 'DIVISION'
```

In a subsequent part of the program one will refer to these words by 'PROCEDURE', 'DATA', 'ENVIRONMENT' or 'DIVISION'.

3.2.3. synonyms (class) declarations.

A synonym declaration is used to allocate a generic name to a class of keywords or reserved words which, from an analytical point of view, can be considered as synonyms. The format of a synonym declaration is:

```
* 'generic-name' = 'synonym-1'  [, 'synonym-i']  ....
```

One can refer to the class of synonyms by the generic name or by the name of any element of the class.

In a "SYDELA" description of PL/I, one will have the following declarations:

```
* 'DECLARE' = 'DCL', 'DECLARE'
```

```
* 'PROCEDURE' = 'PROCEDURE', 'PROC'
```

```
etc...
```

3.3. statements.

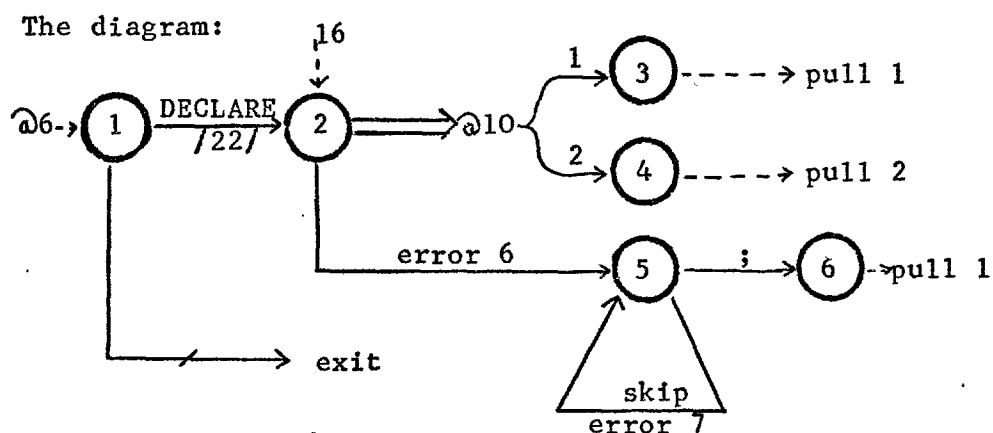
A "SYDELA" statement describes an arc of a diagram. A set of statements between semi-colons describes a diagram. The general format of a statement is:

$[\textcircled{n}] \text{ } [- p] \text{ } : \text{ label } [s] [, n_1 [\cdot n_i] \dots] [\text{ERROR } r]$

Where:

- n is the identification of a diagram or of a secondary entry point;
- p is the number of the origin node;
- label describes the arc;
- s is the number of the semantical action to be executed;
- n_i is (are) the termination node(s) of the arc;
- r is the number of the error diagnostic to be reported.

example:



will be described:

```

@6  -1  : 'DECLARE'      22, 2
      : EXIT
@16 -2  : PUSH 10        , 3. 4
      : VOID             , 5      ERRO 6
-3   : PULL 1
-4   : PULL 2
-5   : X' ; '          , 6
      : SKIP             , 5      ERRO 7
-6   : PULL 1
      ;

```

A complete "SYDELA" description of the CDC 6600 ALGOL-2 can be found in Appendix VI. Details on the implementation of the "SYDELA" analyser and the format of the generated tables are given in Appendix V.

CHAPTER VIII

ANALYSER'S STRATEGIES

1. AN EXTRA-SYNTACTIC EXAMPLE.

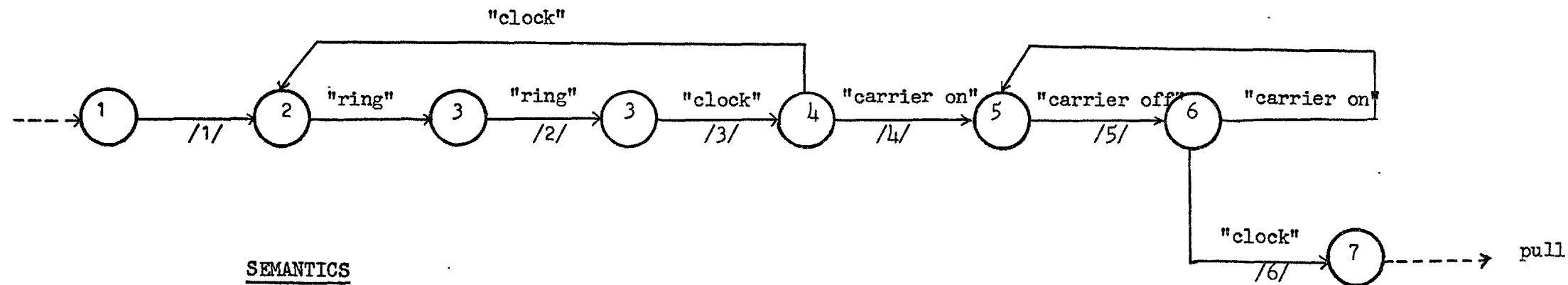
The use of "SYDELA" is not necessarily limited to the description of the syntax of programming languages. It can be used in most problems where a distinction can be made between a lexical stage, producing basic symbols, and an action (syntactic) stage, acting in accordance with patterns of basic symbols.

The following example gives a "SYDELA" description of a system which automatically processes DATEL telephone calls. The basic lexical symbols are interrupts corresponding to the occurrence of external events:

- "ring";
- "clock run out of time";
- "carrier on";
- "carrier off".

Only a small number of the many possible sequences of these interrupts are valid. The "SYDELA" diagram describes these valid combinations and the associated semantics give the meaning which must be attributed to a given combination.

SYNTAX



SEMANTICS

- /1/ line inactive
- /2/ second ring accepted; telephone off hook;
send carrier signal; initiate a 500 ms. clock cycle
- /3/ carrier signal stable; initiate a 30 s. clock cycle
- /4/ carrier received; establish the communication
- /5/ carrier gone; initiate a 500 ms. clock cycle
- /6/ terminate the session

ANSWERING DATEL TELEPHONE CALLS (12)

2. PROCESSING OF KEYWORD LANGUAGES.

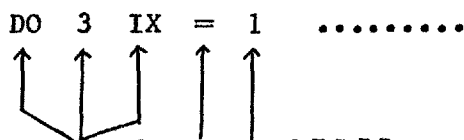
2.1. the deterministic approach.

In keyword languages, the lexical analyser is unable to make the distinction between a name and a keyword. It is the function of the symbol module to decide whether a name should be considered as a keyword in a given context. However, in most languages it is not possible to take this decision with a single pass left to right analysis. For example the FORTRAN construction:

DO3IX=1

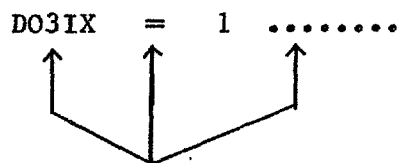
can be analysed as the beginning of a DO-statement as follows:

DO 3 IX = 1



or as an assignment

DO3IX = 1



The solution adopted in most compilers consists in performing a first pass on the statement to decide of its type (i.e. assignment or keyword statement) and then analyse it in a purely deterministic left to right fashion. Simple test can be designed to determine the type of a statement; for example, in FORTRAN, a necessary (and sufficient for the purpose of syntactical analysis) condition for a statement to be an assignment is that the two following conditions are met:

- a- there is an equal symbol (=) not included in an Hollerith constant;
- b- there is no level 0 comma following the equal symbol.

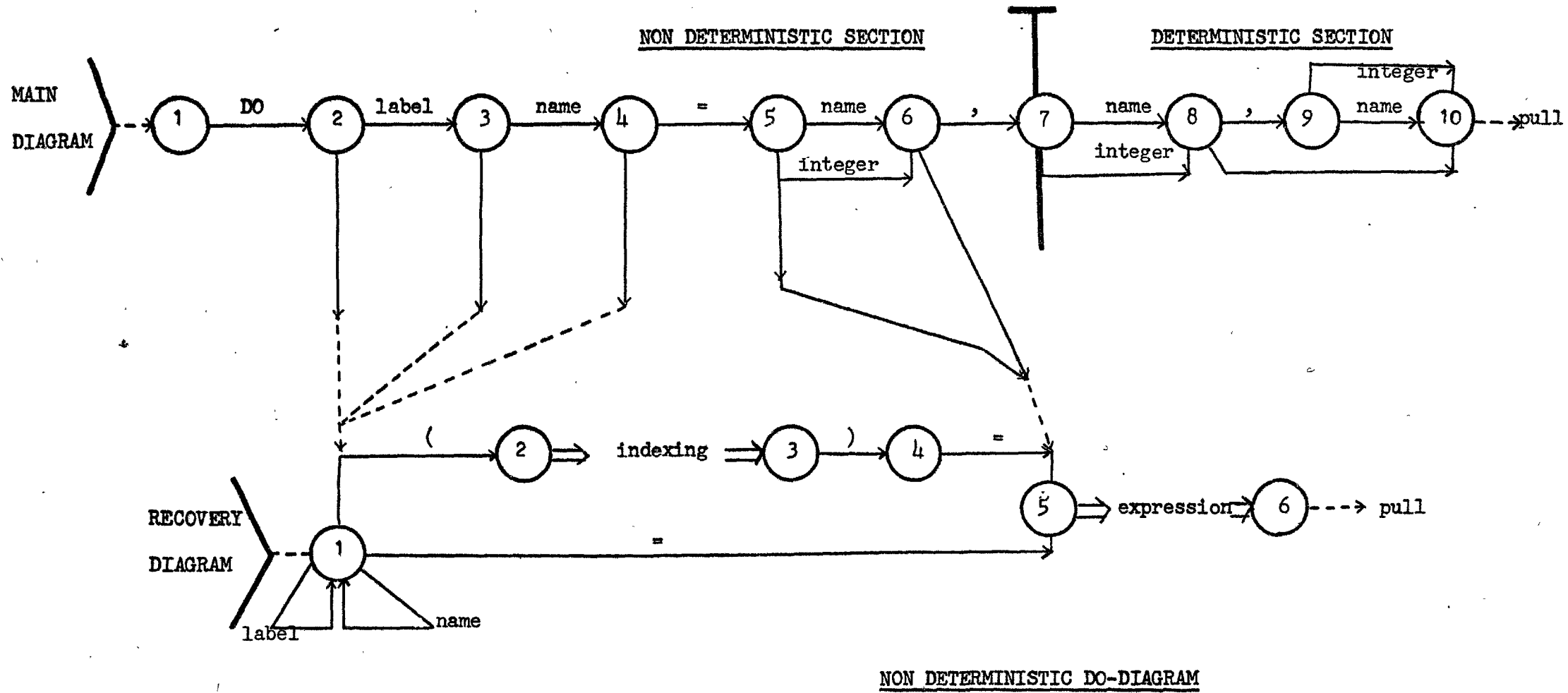
This method, which significantly simplifies the analysis, cannot be implemented on small machines without imposing drastic limitations on the maximum length of a statement (continuation cards or significant characters) due to the necessity of buffering a complete statement in core before starting its analysis.

2.2. a non-deterministic solution.

The non-deterministic approach consists in assuming that all statements beginning with one of the keywords of the language are keyword statements. If later in the analysis an error occurs, one branches to a "Recovery Diagram" which corrects the initial assumption and analyses the statement as an assignment.

The main diagram consists of a non-deterministic section in which any error results in a branch to the Recovery Diagram, and of a deterministic section when one has reached the point where no more ambiguity in the interpretation of the statement is possible. This "no return point" is situated after the first comma in a DO-statement. In a DIMENSION-statement it is situated at the very beginning of the diagram, as no assignment statement can begin with an eight letter name.

This solution is certainly more complex than the straightforward deterministic one, but it can be accommodated on a small machine, as no buffer space is needed to keep a whole statement in core.



3. DECLARATIVES AND TYPE CHECKING.

3.1. the semantical solution.

B.N.F. and most conventional syntax description languages are not suitable for the expression of conditions on the types of the variables entering in a given construction. For example the FORTRAN logical-IF-statement can be described in B.N.F. as:

$\langle \text{logical-IF-st} \rangle ::= \text{IF} (\langle \text{Boolean-expression} \rangle) \langle \text{statement} \rangle$

where $\langle \text{Boolean-expression} \rangle$ can be the simple variable 'A' if it has appeared previously in a 'LOGICAL' statement. Hence the construction:

IF (A) GOTO 38

will be syntactically correct or incorrect depending on the previous occurrence of 'A' in a 'LOGICAL' statement.

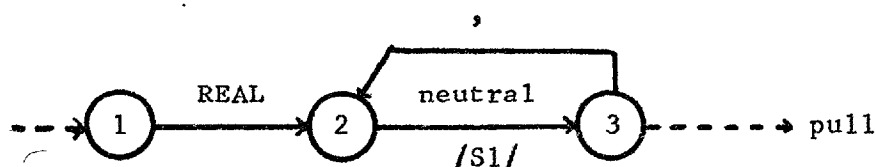
The solution which is generally adopted in compilers and analysers is to accept this construction as syntactically valid, irrespective of the type of A, and check in a semantical action whether A is Boolean or not. More generally, the homogeneity of assignments and expressions will be verified in this manner.

The separation between the syntactical stage and the semantical stage is often difficult to define. It is merely a question of convenience; all what can be reasonably (economically) done by a simple automatic analyser will be considered as syntax, all the rest as semantics. "SYDELA" is particularly well adapted to the processing of declarations and types at the syntactical stage.

3.2. the syntactical solution.

The syntactical solution to the processing of declaratives and type checking consists in considering variables as syntactically different by assigning different analytical codes to them.

For example, in FORTRAN, the diagram corresponding to the REAL-statement can be:



{ /S1/: this action consists in assigning the analytical type "simple real variable" to the current name. This can be achieved by a call to the TYPING procedure. }

Hence REAL X, I, J will assign the analytical type "simple real variable" to the names X, I and J, thus making a further declaration such as:

REAL I

or INTEGER X

syntactically incorrect. In the same manner, the statement:

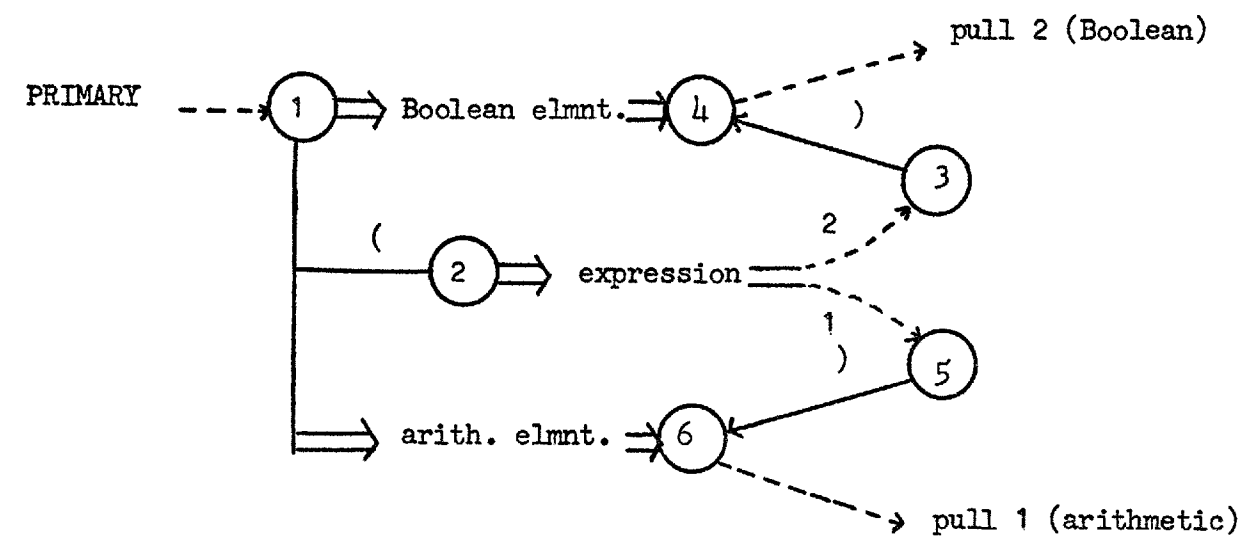
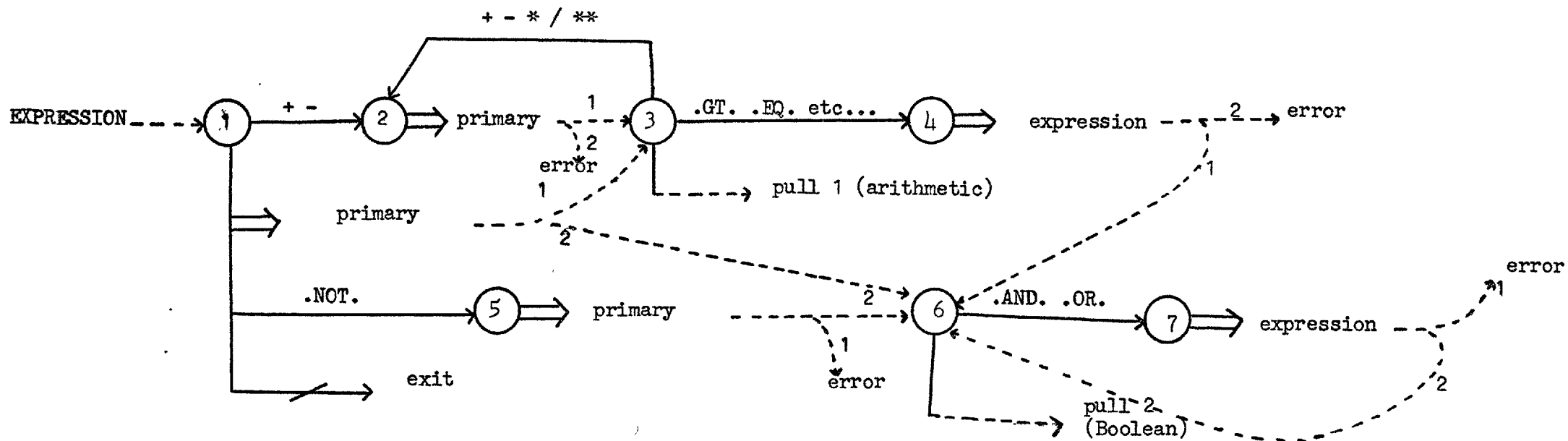
IF (X) GOTO 202

will be rejected at the syntactical stage. An example of the TYPING procedure is given in Appendix V.

4. USE OF MULTIPLE OUTPUT DIAGRAMS.

4.1. a FORTRAN example.

In FORTRAN, a conditional expression can be either a Boolean expression or a comparison or any combination of these such as:



FORTTRAN EXPRESSION

$A+2 .EQ. C .AND. B .OR..NOT. A .GT. B$

which must be read:

$((A+2) .EQ. (C)) .AND. ((B) .OR. (.NOT.((A) .GT. (B))))$

The analysis of such an expression involves a back-up because both an arithmetic and a Boolean expression may begin in the same way. The recursive use of two double output diagrams which discriminate between arithmetic (out 1) and Boolean (out 2) elements allows a no back-up analysis of the construction.

4.2. a COBOL example.

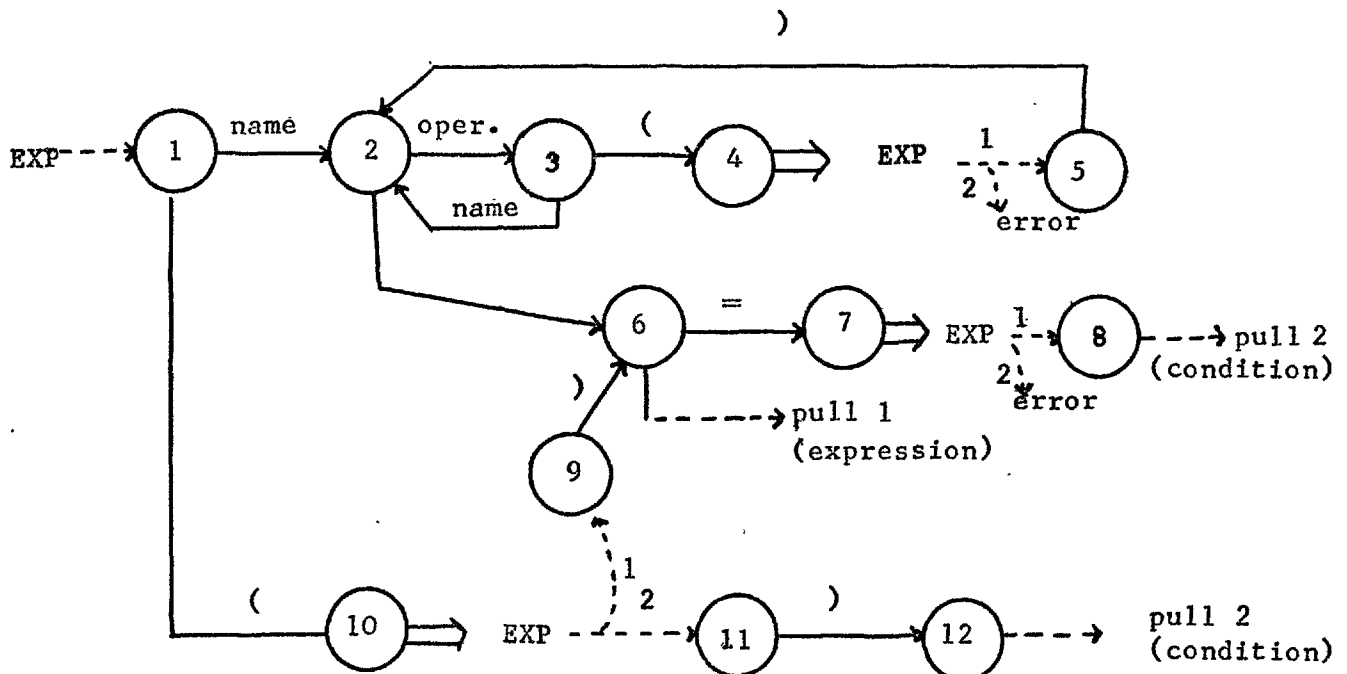
In COBOL, a condition may take one of the following forms:

$(A = B)$

or $(A + B) = C$

or $A = B$

which involves a back-up because the first parenthesis does not indicate whether it surrounds the whole condition or only the right hand side expression. This back-up may be solved by the following double output diagram.



CONCLUSION

CHAPTER IX

AN APPLICATION: PROGRAMMABLE TERMINALS

The development of large centralised computer centres of the type of the "Regional Computer Centres" implies the need for an increasing number of sophisticated terminals to access the central installations. Furthermore, in a near future, the large centres will be connected in huge national and international networks. The user of a terminal will be able to run his programs on any one of a wide range of different machines with different characteristics. Hence the tasks allotted to the terminal will be more complex than the simple transmission of strings of characters. This new type of "intelligent terminal" will decide where the programs which are submitted should be executed; the choice of the object installation being derived from external criteria or from a detailed analysis of the source program.

Even in the simpler configuration, where a mini-computer is connected to a single central installation, it is most convenient to find an economic compromise between the local and remote execution of tasks. The syntactical checking of the submitted programs is a typical example of a simple task which might be executed locally, more especially as the small micro-programmable computers, generally used as terminals, are perfectly fitted to this kind of

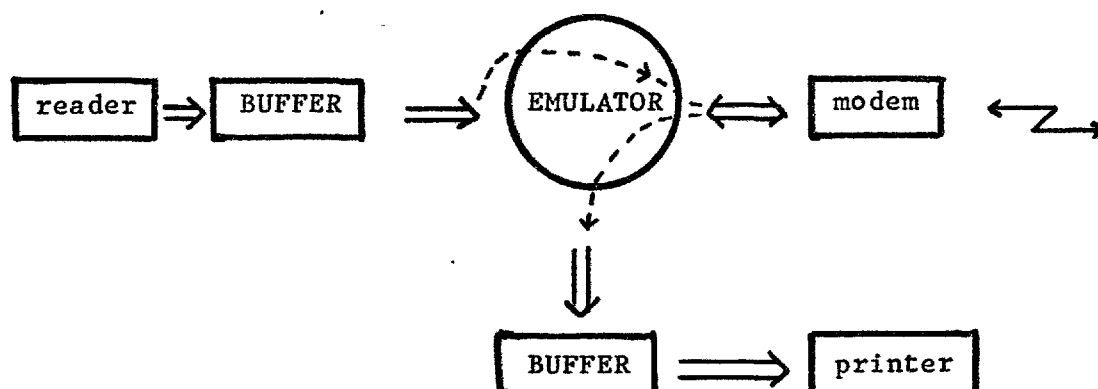
processing.

1. ARCHITECTURE OF THE PROGRAMMABLE TERMINAL SOFTWARE.

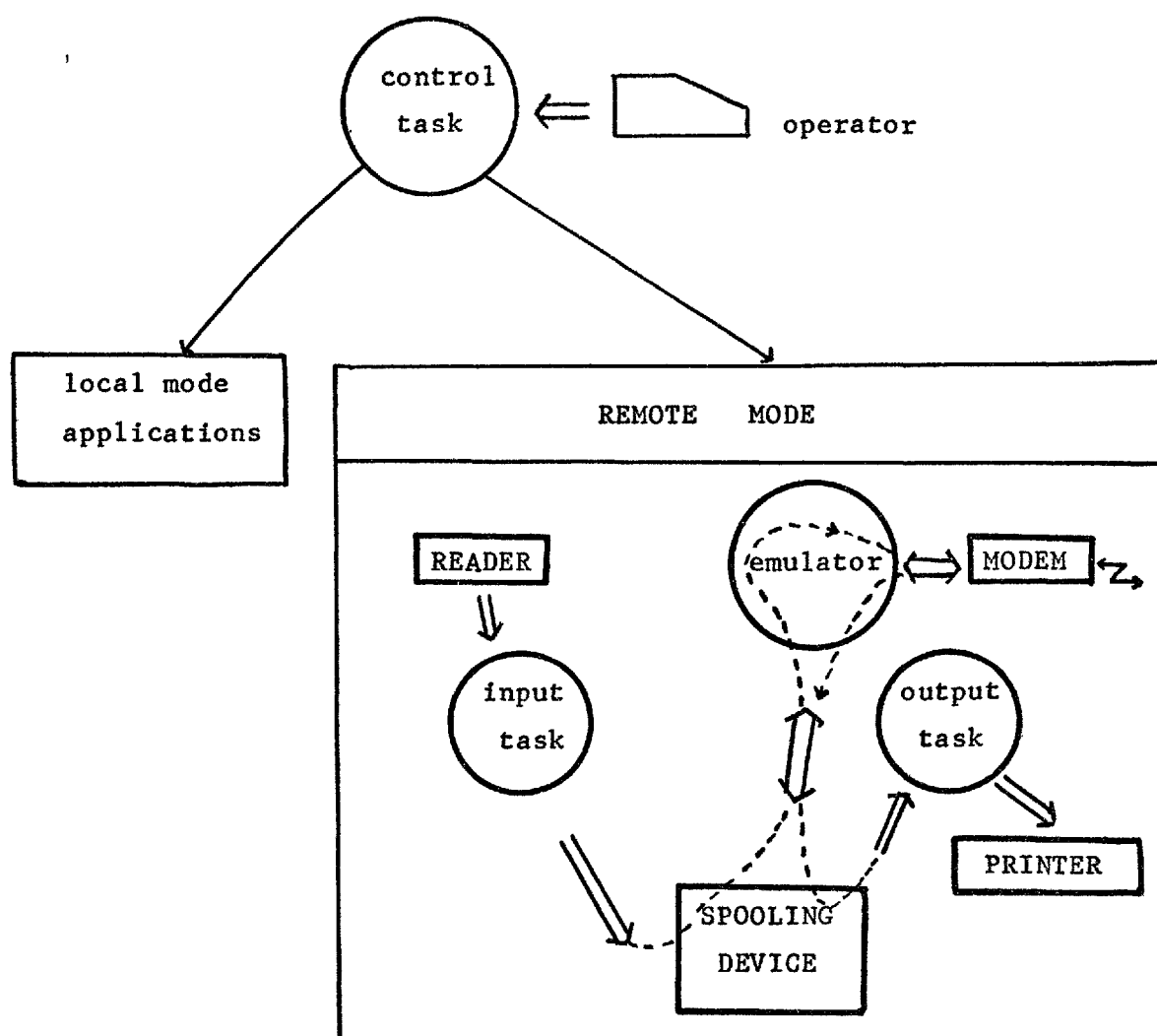
1.1. an elementary terminal.

Most manufacturers of large computer installations which support Remote Job Entry applications also supply a special hardware terminal for that purpose. Hence CDC has developed the U-200 terminal to be used in connection with its 6000-7000 ranges. However, these hardware terminals are not well adapted to other uses. This is the reason why manufacturers of mini-computers have been brought to supply comprehensive libraries of hardware terminal emulators which allow a mixed use of the machine, both in remote and local modes.

The basic functional architecture of a terminal emulator is as follows.

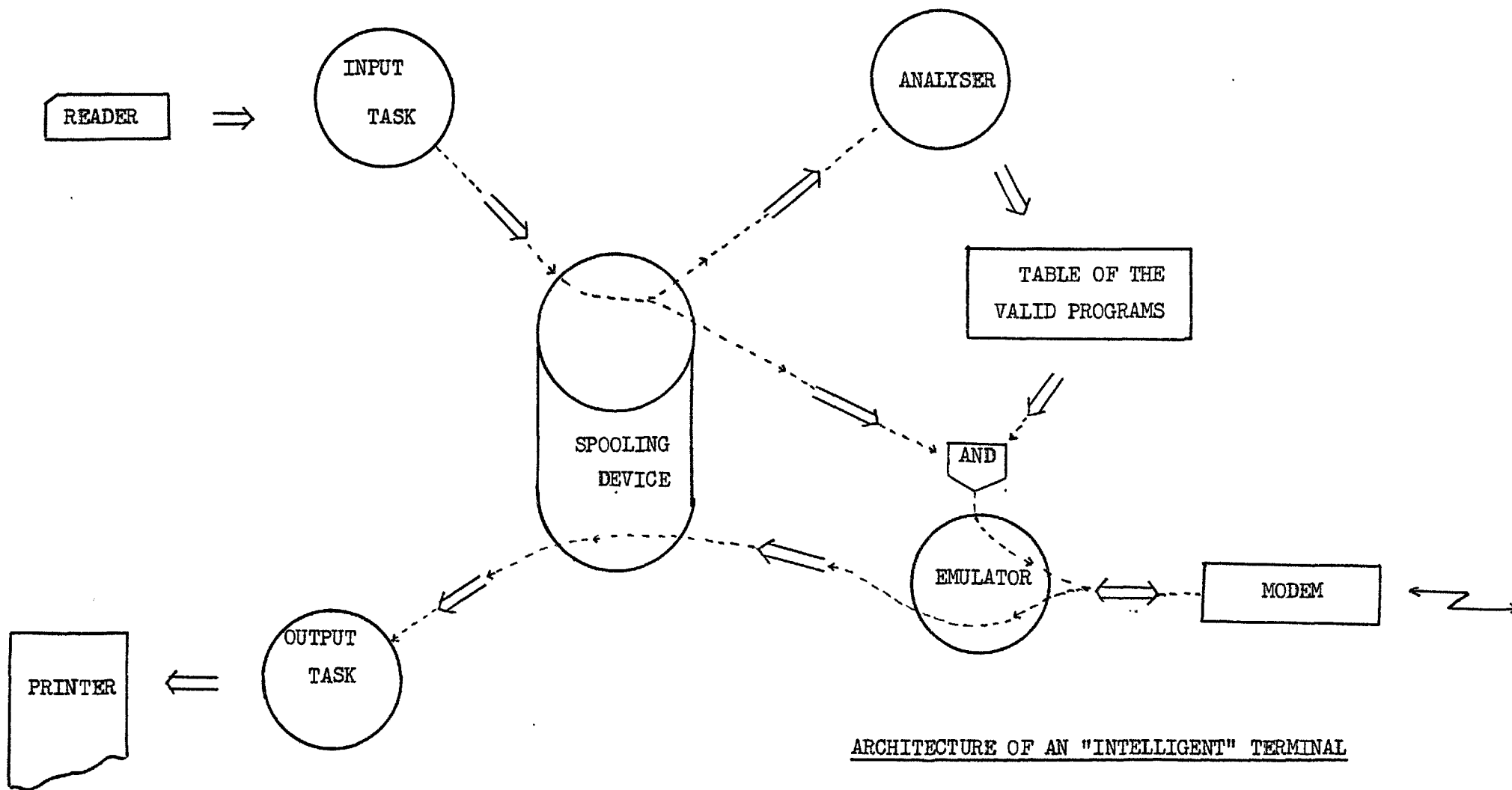


If one wants to use the machine simultaneously in local and remote modes, it is necessary to incorporate a SPOOLing system for the terminal I/O operations, not to penalize the performances of the local mode applications. The structure of the software of such a system is then:



1.2. an "intelligent terminal".

The computers which are used as programmable terminals are generally limited in core space, and their peripherals are slow. Therefore, it will be difficult to support both local and remote applications simultaneously. When the machine is used in remote mode, the peripheral overheads are very important and the time thus wasted could be used to execute simple operations on the source text. For example, one could pack the text, replacing all multiple characters by a single one. An other type of processing is the local syntactical analysis of the source programs. Only programs that have serious chances of being executed successfully will be transmitted to the remote installation; the other ones will be



rejected by the terminal and comprehensive diagnostics will be produced. The most substantial advantages of this system can be identified as follows:

- economy in the use of the remote computer's CPU time which is more expensive than the local CPU time;
- a better use of the local machine; the peripheral overhead time being used for validation purposes;
- an improvement in the turn-round time of invalid programs which are rejected as soon as they are read in;
- the unloading of the communication line which is not used to transfer incorrect programs.

1.3. an "intelligent" terminal for a computer network.

The same "intelligent" terminal can be used to communicate with more than one remote computer of different types. The modular design described in 1.2. also applies to this context: the input, output, analysis and emulation modules being multiplied in accordance with the given configuration. However some practical problems render the implementation of this multiple terminal difficult. In particular, it is necessary to design a control language which allows the user to specify the machine on which his program must be executed. One must also take into account the differences between different implementations of the same language.

The single analysis of the Job Control (or Command) Language is generally insufficient to decide which languages are used in a source deck (catalogued procedures in IBM 360 JCL, conditional

skip of a step in CDC SCOPE, etc...). If the terminal wants to be able to analyse the source text, this information must then be provided externally. This is the first step towards the design of a local Job Control Language which may:

- replace the control languages of the object machines, in which case the terminal will generate the commands which are necessary for the execution on the object computer;
- or
- be superposed on the existing command languages, in which case the user chooses the object machine and codes the required job control statements; the local language then only consists in a description of the physical layout of the submitted deck (the sequence of languages used) to allow the syntactical analysis.

The choice between these two solutions is a question of convenience which depends on the way the terminal is used.

2. IMPLEMENTATION TECHNIQUES.

2.1. the problems of the local syntactical analysis.

It is important to define the depth of syntactical analysis necessary to decide whether a program is worth being transmitted or should be rejected. A compromise must be found between the cost (core space, cost of development, execution time...) of the analyser and the frequency of some errors. This compromise can be only achieved by an empirical study of the submitted programs and a progressive adaptation of the local software to its result. One can aim at the two following objectives:

- no program which can be successfully loaded after

compilation should be rejected by the terminal;
 and - 95% (or 99%) of the programs which cause an unrecoverable abortion of the compilation must be discarded by the terminal.

An empirical study of FORTRAN programming habits (14) showed that only a little subset of the language is used and that some constructions are rarely encountered. For example, as more than 50% of the assignment statements are single replacements of the form

$$A = B$$

where no arithmetic is involved, the analyser must be able to process these statements fastly; even a slight improvement in the speed of analysis of this construction will contribute to the overall performance of the analyser. In a similar manner, the fact that the assignment, IF, GOTO, CALL, CONTINUE and WRITE statements account for more than 80% of the statements in a program can influence the design of the analyser. The same study showed that an average of 48 blank columns per card can be found in a typical FORTRAN program; this must imply that the lexical analyser should include a high-speed skip over blank columns.

The empirical study of programming habits can also be used to limit the analysis to a stage which is both economic and realistic. For example in FORTRAN the depth of DO-nesting can be analysed as follows (14):

DEPTH	1	2	3	4	5	5
PER CENT	53.5	23	15	5.5	1.5	1.5

As a consequence, one could limit the test of the correct nesting of DO-loops to level 5 without sensibly modifying the 95% (or 99%)

success performance which is aimed; but this limitation reduces the size of the test stack to 5 instead of 100 or more.

In COBOL, an element in a structure may be referenced by a qualified name such as:

C IN B OF A

Depending on the result of an empirical study of the programming habits of a class of users, one can decide to adopt one of the three following solutions.

a- easy and economic solution

accept all constructions of the type

name $\left[\begin{array}{c} \text{OF} \\ \text{IN} \end{array} \right] \text{ name} \quad \dots$

without any check;

b- intermediate solution

accept only the constructions

C IN B OF A

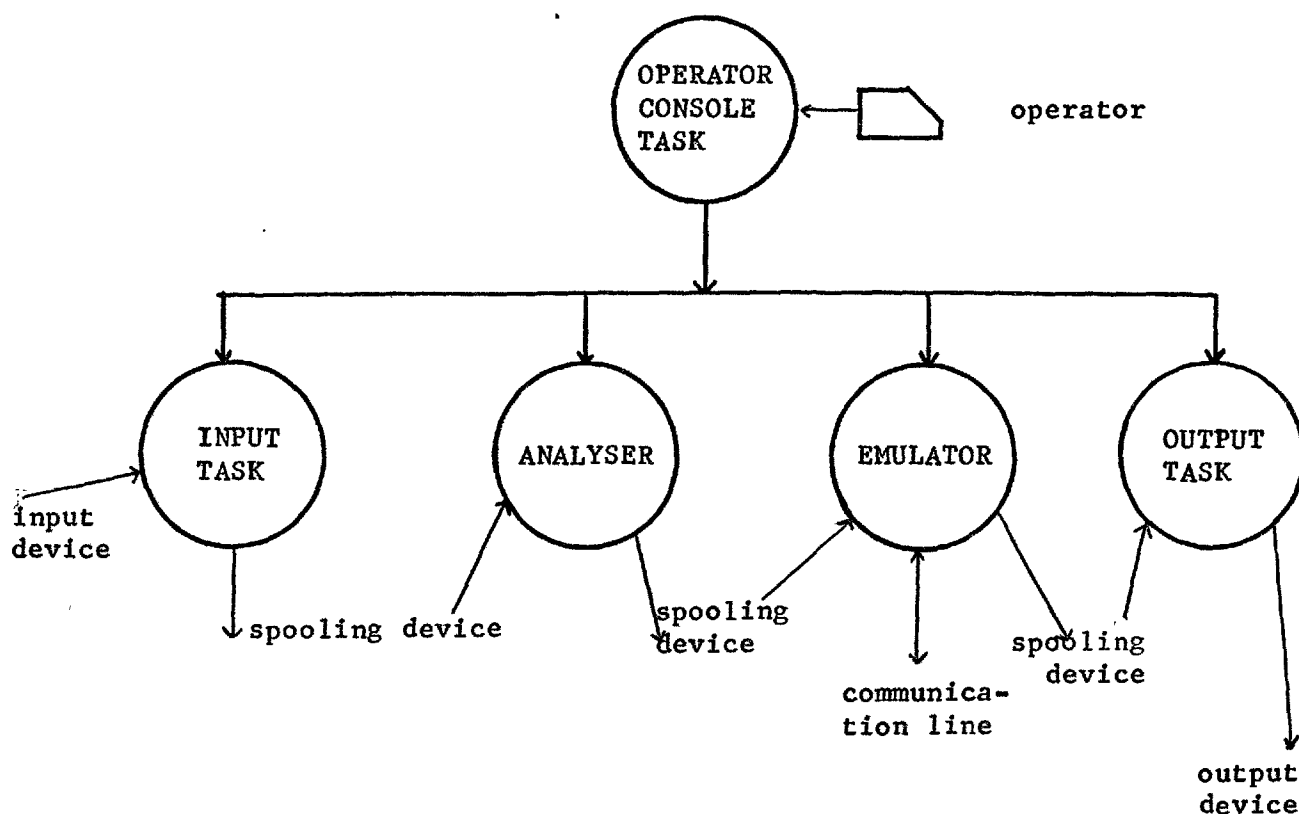
when C belongs to a structure named B which itself belongs to a structure named A;

c- complete but costly solution

verify as in b that the qualification is valid, and also that it is unambiguous (or sufficient): i.e. no other element named C belongs to a structure named B which itself belongs to a structure named A.

2.2. structure of the software.

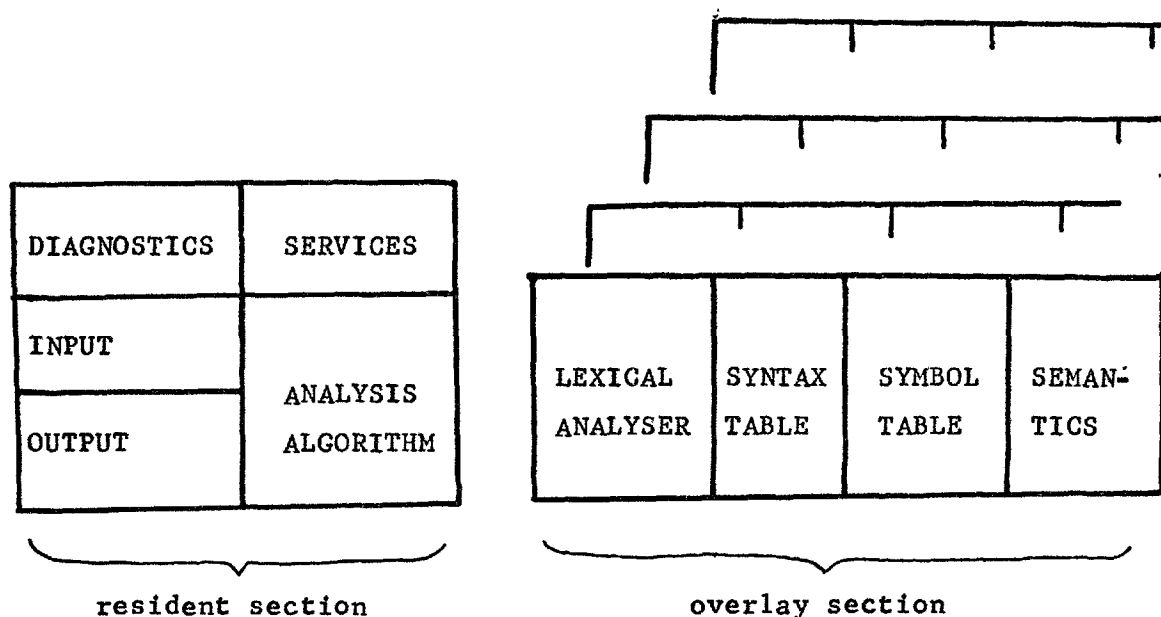
The structure of an "intelligent terminal" such as the one described in 1.2. can be designed in four concurrent tasks under the control of an operator console task.



- the operator task has the control of the operator console;
- the input task drives the input device (card reader, paper tape reader...) and has shared access to the spooling device;
- the analyser has shared access to the spooling device;
- the emulator has private control of the communication line and shared access to the spooling device;
- the output task controls the output device and can access the spooling device.

As the number of languages used on an installation may be high, all analysers for all languages cannot be resident in core all time. The analyser will then consist of a resident section containing the input, output and diagnostic modules, the syntactical

analysis algorithm and some services, and of an overlay section containing the lexical analyser, the syntax and symbol tables and the semantics of the analysed languages.



When the Command Language is imbedded in the input stream (like IBM 360 JCL), the resident section can also contain the analyser for the Command Language. In the other cases (like in CDC SCOPE or in ATLAS command language) where all commands head the stream, the Command Language can be considered as a standard language and its analyser set in the overlay section.

2.3. methods of implementation.

The adaptative approach to the software design described in 2.1. is practicable only if one has a high level form of expression for both the syntax and the semantics of the analysed languages. Furthermore, when the specifications of a language change between two releases of a compiler, one must be able to modify the local analyser accordingly. "SYDELA" is well adapted to this kind of

processing because it is a high level form of expression which can be very easily taught and can be used to build libraries of language specifications. The semantics of the analyser can be coded in a system programming language such as P. L. Modular One, which makes them self-documenting, thus simplifying the task of maintenance.

The design described in chapter VI clearly separates the functions of an analyser in self-contained modules which can be individually modified and tested. The use of an Executive such as the one described in chapter V provides a clear interface between the implementation language and the hardware, it eliminates the tedious work of coding input/output operations in a low level form and provides a sufficiently powerful multitasking mechanism.

Finally, the modular design and the high level languages used in the implementation make the system reasonably portable, both internally and externally.

APPENDICES

APPENDIX I

CHARACTERISTICS OF SOME MINI-COMPUTERS

MANUFACTURER	MODEL	CYCLE SPEED (μ s)	WORD SIZE	COST OF A TYPICAL SYSTEM(£)	PRINCIPAL USE
Allied Business System Ltd.	GRI 99	1.8	16	8000.	commercial
Arcturus Electronics	A18D	0.9	18	10000	process- control
Business Computer Ltd.	Molecular 18	1.6	18	10000	commercial
Computer Automation Ltd.	Naked Mini 16	1.6	16	5000	process- control
Control Data Corporation	System 17	0.6	18	20000	scientific and commercial
Compagnie Internation. Informatique	Mitra 15	0.8	16	20000	process- control
Computer Technology Ltd.	Modular 1	0.7	16	12000	education and communicat

Digital Equipment Corporation	PDP/8	1.2	16	12000	education, research & scientific
Digico	Micro 16	0.9	16	10000	scientific
Ferranti	Argus 700	0.5	16	20000	communicat.
General Electric Computers	2050	0.9	8	8000	process-control & communicat.
General Automation Corporation	SPC 16/80	0.8	16	20000	process-control & communicat.
Hewlett Packard	2100	1.0	16	20000	scientific and commercial
Honeywell	716	0.8	16	40000	scientific and commercial
Interdata	50	1.0	16	5000	communicat.
Micro Computer Ltd.	Minic	0.9	8	10000	process-control
Modular Computer Systems	Modcomp I	0.8	16	3000	process-control
Philips Electrológica Ltd.	P.850	3.2	16	10000	process-control & commercial
Télémécanique Electrique	T.1600	0.9	16	20000	scientific and communicat.

APPENDIX II

A SYNTACTICAL DESCRIPTION OF L.S.D.

1. BASIC ELEMENTS.

$\langle \text{null} \rangle \quad ::=$

$\langle \text{letter} \rangle \quad ::= A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L$
 $\quad \quad \quad \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X$
 $\quad \quad \quad \mid Y \mid Z$

$\langle \text{digit} \rangle \quad ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\langle \text{hexadigit} \rangle \quad ::= \langle \text{digit} \rangle \mid A \mid B \mid C \mid D \mid E \mid F$

$\langle \text{lab} \rangle \quad ::= \langle$

$\langle \text{rab} \rangle \quad ::= \rangle$

$\langle \text{vbar} \rangle \quad ::= \mid$

$\langle \text{identifier} \rangle \quad ::= \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle$
 $\quad \quad \quad \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle$

$\langle \text{integerconstant} \rangle \quad ::= \langle \text{digit} \rangle \mid \langle \text{integerconstant} \rangle \langle \text{digit} \rangle$

$\langle \text{hexadecimalconstant} \rangle \quad ::= \langle \text{hexadigit} \rangle \mid \langle \text{hexadecimalconstant} \rangle$
 $\quad \quad \quad \langle \text{hexadigit} \rangle$

2. STATEMENTS.

$\langle \text{LSDstatement} \rangle \quad ::= \quad : \langle \text{identifier} \rangle : \langle \text{unlabelledstatement} \rangle ;$
 $\quad \quad \quad \mid \langle \text{unlabelledstatement} \rangle ;$
 $\quad \quad \quad \mid \langle \text{declaration} \rangle ;$

```

<unlabelledstatement> ::= SET<exprn>
                        | SETF<exprn>
                        | <exprn>
                        | IF<conditionalexprn>
                        | IFF<conditionalexprn>
                        | GOTO<identifier>
                        | GOINDTO<operand>
                        | GOTOVIA<operand>
                        | SWITCH (<identifierlist>) <operand>
                        | LOOP (<exprn>UNTIL<exprn><step>)
                                =><operand>
                        | ENDLOOP | LOOP <operand>
                        | CALL<identifier><actualparameterlist>
                        | INC<operandlist>
                        | DEC<operandlist>
                        | CLEAR<operandlist>
                        | DUMP
                        | STOP
                        | RETURN
                        | COMMENT
                        | CODE<codeline>
                        | BEGINCODE

<actualparameterlist> ::= (<actualinput>)<actualoutput> | <null>

<actualinput>         ::= <exprlist> | <null>

<actualoutput>        ::= (<operandlist>) | <null>

<operandlist>         ::= <operand>,<operandlist> | <opernad>

<exprlist>            ::= <exprn>,<exprlist>
                        | SETF<exprn>,<exprlist> | <null>

```

$\langle \text{step} \rangle \quad ::= \text{STEP} \langle \text{exprn} \rangle \mid \langle \text{null} \rangle$
 $\langle \text{codeline} \rangle \quad ::= \langle \text{any assembly language instruction to be copied into the object program} \rangle$

3. EXPRESSION.

$\langle \text{exprn} \rangle \quad ::= \langle \text{optionalsign} \rangle \langle \text{operexp} \rangle \langle \text{restofexpression} \rangle$
 $\langle \text{restofexpression} \rangle \quad ::= \langle \text{operator} \rangle \langle \text{operexp} \rangle \langle \text{restofexpression} \rangle$
 $\quad \mid \langle \text{null} \rangle$
 $\langle \text{optionalsign} \rangle \quad ::= + \mid - \mid \langle \text{null} \rangle$
 $\langle \text{operand} \rangle \quad ::= \langle \text{constant} \rangle \mid \langle \text{identifier} \rangle$
 $\quad \mid \langle \text{subscriptedelement} \rangle \mid \langle \text{addressreference} \rangle \mid ()$
 $\langle \text{operexp} \rangle \quad ::= \langle \text{operand} \rangle \mid (\langle \text{exprn} \rangle)$
 $\langle \text{operator} \rangle \quad ::= + \mid - \mid * \mid / \mid \% \mid \& \mid \langle \text{vbar} \rangle \mid \Rightarrow \mid \text{INDEX}$
 $\quad \mid \text{ORDNL} \mid \text{SRL} \mid \text{SLL}$
 $\langle \text{constant} \rangle \quad ::= \langle \text{integerconstant} \rangle$
 $\quad \mid \langle \text{integerconstant} \rangle . \langle \text{integerconstant} \rangle \langle \text{exponent} \rangle$
 $\quad \mid @ \langle \text{hexadecimalconstant} \rangle$
 $\quad \mid \$ \langle \text{character} \rangle$
 $\quad \mid \langle \text{any string of characters not containing a prime} \rangle$
 $\quad \mid \langle \text{identifier} \rangle$
 $\langle \text{exponent} \rangle \quad ::= E \langle \text{optionalsign} \rangle \langle \text{integerconstant} \rangle \mid \langle \text{null} \rangle$
 $\langle \text{subscriptedelement} \rangle \quad ::= \langle \text{onedimensionalarray} \rangle$
 $\quad \mid \langle \text{functioncall} \rangle$
 $\quad \mid \langle \text{pointervariable} \rangle$
 $\langle \text{onedimensionalarray} \rangle \quad ::= \langle \text{identifier} \rangle (\langle \text{exprn} \rangle) \langle \text{increment} \rangle$
 $\langle \text{functioncall} \rangle \quad ::= \langle \text{identifier} \rangle \langle \text{pointers} \rangle$

$\langle \text{pointers} \rangle ::= (\langle \text{exprn} \rangle) \langle \text{pointers} \rangle \mid (\langle \text{exprn} \rangle)$
 $\langle \text{increment} \rangle ::= \text{I} \langle \text{incspec} \rangle \mid \text{D} \langle \text{incspec} \rangle \mid \langle \text{null} \rangle$
 $\langle \text{incspec} \rangle ::= \langle \text{constant} \rangle \mid \langle \text{null} \rangle$
 $\langle \text{addressreference} \rangle ::= \langle \text{lab} \rangle \langle \text{identifier} \rangle \langle \text{rab} \rangle$
 $\quad \mid \langle \text{lab} \rangle \langle \text{subscriptedelement} \rangle \langle \text{rab} \rangle$
 $\quad \mid \langle \text{lab} \rangle : \langle \text{identifier} \rangle : \langle \text{rab} \rangle$

4. DECLARATIONS.

$\langle \text{declaration} \rangle ::= \langle \text{type} \rangle \langle \text{declist} \rangle$
 $\quad \mid \text{EXTERNAL} \langle \text{arraydeclist} \rangle$
 $\quad \mid \text{MANIFEST} \langle \text{mandeclist} \rangle$
 $\quad \mid \text{DATA} \langle \text{datainitlist} \rangle$
 $\quad \mid \langle \text{dcl} \rangle \langle \text{identifierlist} \rangle$
 $\quad \mid \langle \text{seg} \rangle \langle \text{identifier} \rangle$
 $\quad \mid \text{FINISH}$
 $\quad \mid \text{ENDMODULE}$
 $\quad \mid \langle \text{procfun} \rangle \langle \text{identifier} \rangle \langle \text{formalparameterlist} \rangle$
 $\quad \mid \text{END}$
 $\quad \mid \text{PROCEDURE MAIN}$
 $\langle \text{type} \rangle ::= \text{INTEGER} \mid \text{REAL} \mid \text{CHARACTER} \mid \text{BOOLEAN}$
 $\langle \text{dcl} \rangle ::= \text{FNLIST} \mid \text{PROCLIST} \mid \text{TYPE} \mid \text{IREG}$
 $\langle \text{seg} \rangle ::= \text{PROGRAM} \mid \text{MODULE}$
 $\langle \text{procfun} \rangle ::= \text{PROCEDURE} \mid \text{FUNCTION}$
 $\langle \text{identifierlist} \rangle ::= \langle \text{identifier} \rangle, \langle \text{identifierlist} \rangle \mid \langle \text{identifier} \rangle$
 $\langle \text{formalparameterlist} \rangle ::= \langle \text{formalinput} \rangle \langle \text{formaloutput} \rangle \mid \langle \text{null} \rangle$
 $\langle \text{formalinput} \rangle ::= (\langle \text{identifierlist} \rangle) \mid ()$
 $\langle \text{formaloutput} \rangle ::= (\langle \text{identifierlist} \rangle) \mid \langle \text{null} \rangle$
 $\langle \text{declist} \rangle ::= \langle \text{dcl} \rangle, \langle \text{declist} \rangle \mid \langle \text{dcl} \rangle$

$\langle \text{decl} \rangle \quad ::= \langle \text{arraydec} \rangle$
 $\quad \quad \quad | \langle \text{identifier} \rangle$
 $\quad \quad \quad | \langle \text{optionalsign} \rangle \langle \text{constant} \rangle \Rightarrow \langle \text{identifier} \rangle$
 $\langle \text{arraydec} \rangle \quad ::= \langle \text{identifier} \rangle (\langle \text{constant} \rangle)$
 $\langle \text{arraydeclist} \rangle ::= \langle \text{arraydec} \rangle , \langle \text{arraydeclist} \rangle \mid \langle \text{arraydec} \rangle$
 $\langle \text{mandeclist} \rangle \quad ::= \langle \text{mandec} \rangle , \langle \text{mandeclist} \rangle \mid \langle \text{mandec} \rangle$
 $\langle \text{mandec} \rangle \quad \quad ::= \langle \text{optionalsign} \rangle \langle \text{constant} \rangle \Rightarrow \langle \text{identifier} \rangle$
 $\langle \text{datainitlist} \rangle ::= \langle \text{datainit} \rangle , \langle \text{datainitlist} \rangle \mid \langle \text{datainit} \rangle$
 $\langle \text{datainit} \rangle \quad ::= \langle \text{datavalues} \rangle \Rightarrow \langle \text{identifier} \rangle$
 $\quad \quad \quad | \langle \text{datavalues} \rangle \Rightarrow \langle \text{identifier} \rangle (\langle \text{constant} \rangle)$
 $\langle \text{datavalues} \rangle \quad ::= \text{BITS} \langle \text{bitlist} \rangle \langle \text{datavaluelist} \rangle$
 $\quad \quad \quad | \langle \text{dataconst} \rangle \langle \text{datavaluelist} \rangle$
 $\langle \text{dataconst} \rangle \quad ::= \langle \text{repfact} \rangle \langle \text{optionalsign} \rangle \langle \text{constant} \rangle$
 $\langle \text{repfact} \rangle \quad \quad ::= \langle \text{constant} \rangle \mid \langle \text{null} \rangle$
 $\langle \text{datavaluelist} \rangle ::= : \langle \text{datavalues} \rangle \mid \langle \text{null} \rangle$
 $\langle \text{bitlist} \rangle \quad \quad ::= \langle \text{bitnumber} \rangle , \langle \text{bitlist} \rangle \mid \langle \text{bitnumber} \rangle$
 $\langle \text{bitnumber} \rangle \quad ::= \langle \text{constant} \rangle - \langle \text{constant} \rangle \mid \langle \text{constant} \rangle$

5. CONDITIONAL EXPRESSION.

$\langle \text{conditionalexprn} \rangle ::= \langle \text{relationexprn} \rangle \langle \text{boolop} \rangle \langle \text{conditionalexprn} \rangle$
 $\quad \quad \quad \langle \text{condtail} \rangle$
 $\quad \quad \quad | \langle \text{relationexprn} \rangle \langle \text{condtail} \rangle$
 $\langle \text{boolop} \rangle \quad \quad ::= \text{AND} \mid \text{OR}$
 $\langle \text{relationexprn} \rangle ::= \langle \text{exprn} \rangle \langle \text{conditionaloperator} \rangle \langle \text{exprn} \rangle$
 $\langle \text{conditionaloperator} \rangle ::= \text{EQ} \mid \text{NE} \mid \text{LT} \mid \text{GT} \mid \text{GE} \mid \text{LE}$
 $\langle \text{condtail} \rangle \quad \quad ::= : \langle \text{unlabelledstatement} \rangle$
 $\quad \quad \quad | \text{THEN} \langle \text{unlabelledstatement} \rangle \text{ELSE}$
 $\quad \quad \quad \langle \text{unlabelledstatement} \rangle$

APPENDIX III

A SYNTACTICAL DESCRIPTION OF P.L. MODULAR ONE

1. BASIC ELEMENTS.

$\langle \text{null} \rangle \quad ::=$
 $\langle \text{letter} \rangle \quad ::= A | B | C | D | E | F | G | H | I | J | K | L | M$
 $\quad \quad \quad | N | O | P | Q | R | S | T | U | V | W | X | Y | Z$
 $\langle \text{decidigit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
 $\langle \text{hexadigit} \rangle ::= \langle \text{decidigit} \rangle | A | B | C | D | E | F$
 $\langle \text{name} \rangle \quad \quad ::= \langle \text{letter} \rangle | \langle \text{name} \rangle \langle \text{letter} \rangle | \langle \text{name} \rangle \langle \text{decidigit} \rangle$
 $\langle \text{decicst} \rangle \quad ::= \langle \text{decidigit} \rangle | \langle \text{decicst} \rangle \langle \text{decidigit} \rangle$
 $\langle \text{hexacst} \rangle \quad ::= \alpha \langle \text{hexapart} \rangle$
 $\langle \text{hexapart} \rangle ::= \langle \text{hexadigit} \rangle | \langle \text{hexapart} \rangle \langle \text{hexadigit} \rangle$

2. PROGRAM.

$\langle \text{input-deck} \rangle ::= \langle \text{plml-deck} \rangle \%$
 $\langle \text{plml-deck} \rangle ::= \langle \text{batch} \rangle | \langle \text{batch} \rangle \langle \text{plml-deck} \rangle$
 $\langle \text{batch} \rangle \quad \quad ::= \% \langle \text{options-spec} \rangle \langle \text{prog-list} \rangle$
 $\langle \text{options-spec} \rangle ::= \text{OPTIONS} | \text{OPTIONS} \langle \text{option-list} \rangle$
 $\langle \text{option-list} \rangle ::= \langle \text{elem-option} \rangle | \langle \text{elem-option} \rangle, \langle \text{option-list} \rangle$
 $\langle \text{elem-option} \rangle ::= \text{XREF} | \text{TRACE}$
 $\langle \text{prog-list} \rangle \quad ::= \langle \text{null} \rangle | \langle \text{prog-body} \rangle \langle \text{prog-list} \rangle$
 $\langle \text{prog-body} \rangle \quad ::= \langle \text{pg-header} \rangle \langle \text{general-dcl} \rangle \langle \text{proc-list} \rangle \langle \text{stmt-list} \rangle$
 $\quad \quad \quad \langle \text{pg-trailer} \rangle$

$\langle \text{pg-header} \rangle ::= \text{PROGRAM} \langle \text{name} \rangle ; \mid \text{PROGRAM} \langle \text{name} \rangle \text{MAIN} ;$
 $\langle \text{pg-trailer} \rangle ::= \text{END} ; \mid \text{END} \langle \text{name} \rangle ;$
 $\langle \text{general-dcl} \rangle ::= \langle \text{null} \rangle \mid \langle \text{declarer} \rangle \langle \text{declaratives} \rangle$
 $\qquad \qquad \qquad \mid \langle \text{general-dcl} \rangle \langle \text{declarer} \rangle \langle \text{declaratives} \rangle$
 $\langle \text{declarer} \rangle ::= \text{COMMON} \mid \text{GLOBAL}$
 $\langle \text{proc-list} \rangle ::= \langle \text{null} \rangle \mid \langle \text{proc-list} \rangle \langle \text{proc-body} \rangle$
 $\langle \text{stmt-list} \rangle ::= \langle \text{null} \rangle \mid \langle \text{statement} \rangle \langle \text{stmt-list} \rangle$

3. PROCEDURE.

$\langle \text{proc-body} \rangle ::= \text{PROCEDURE} \langle \text{name} \rangle \langle \text{arg-dcl} \rangle ; \langle \text{declare} \rangle$
 $\qquad \qquad \qquad \langle \text{stmt-list} \rangle \langle \text{proc-trailer} \rangle$
 $\langle \text{proc-trailer} \rangle ::= \text{END} ; \mid \text{END} \langle \text{name} \rangle ;$
 $\langle \text{declare} \rangle ::= \langle \text{null} \rangle \mid \text{DECLARE} \langle \text{declaratives} \rangle$
 $\langle \text{arg-dcl} \rangle ::= \langle \text{null} \rangle \mid (\langle \text{arg-list} \rangle)$
 $\langle \text{arg-list} \rangle ::= \langle \text{argument} \rangle \mid \langle \text{arg-list} \rangle , \langle \text{argument} \rangle$
 $\langle \text{argument} \rangle ::= \langle \text{class} \rangle \langle \text{name} \rangle \mid \text{POINTER} \langle \text{name} \rangle$
 $\qquad \qquad \qquad \mid \langle \text{ref-d} \rangle \langle \text{class} \rangle \langle \text{type-d} \rangle$
 $\langle \text{ref-d} \rangle ::= \text{REFERENCE} \mid \text{REFERENCE TO}$
 $\langle \text{class} \rangle ::= \langle \text{null} \rangle \mid \text{LABEL} \mid \text{INTEGER} \mid \text{BOOLEAN} \mid \text{WORD}$
 $\langle \text{type-d} \rangle ::= \langle \text{name} \rangle \mid \text{ARRAY} \langle \text{name} \rangle \mid \text{STACK} \langle \text{name} \rangle$

4. DECLARATIVES.

$\langle \text{declaratives} \rangle ::= \langle \text{dcl-list} \rangle ;$
 $\langle \text{dcl-list} \rangle ::= \langle \text{dcl-item} \rangle \mid \langle \text{dcl-item} \rangle , \langle \text{dcl-list} \rangle$
 $\langle \text{regname} \rangle ::= \langle \text{name} \rangle \mid @ \mid \# \mid \$$
 $\langle \text{storage} \rangle ::= \langle \text{null} \rangle \mid \text{CONTROLLED}$

```

<dcl-item> ::= <class><name><storage><value>
            | <class>ARRAY<name><size><storage><value>
            | <class>STACK<name><size><storage>
            | <class>FORM<name><org>
            | <class>CONSTANT<name> = <const>
            | <class>REGISTER<name> = <regname>
            | POINTER<name>
<size>      ::= ( <const> ) | ( <name> )
<org>       ::= <null>
            | ORIGIN ( <const> )
            | ORIGIN ( <name> )
<value>     ::= <null> | INITIAL ( <value-list> )
<value-list> ::= <repetition><const>
            | <repetition><const>, <value-list>
<repetition> ::= <null> | ( <decicst> )

```

5. CONSTANT & CELL.

```

<const>     ::= <name> | <hexacst> | <decicst> | +<decicst>
            | -<decicst> | TRUE | FALSE
<cell>      ::= <name> | <name>-><formlist> | <name><indexing>
<formlist>  ::= <name> | <name>.<formlist>
<indexing>  ::= ( <const> )
            | ( <name><modifier> )
            | ( $ <modifier> )
<modifier>  ::= <null> | +<const> | -<const>

```

6. STATEMENTS.

$\langle \text{statement} \rangle ::= \langle \text{unlab-st} \rangle \mid \langle \text{name} \rangle : \langle \text{statement} \rangle$

$\langle \text{unlab-st} \rangle ::= ;$

$\mid \langle \text{pter-repl} \rangle$
 $\mid \langle \text{replacement} \rangle$
 $\mid \text{GOTO} \langle \text{cell} \rangle ;$
 $\mid \text{GOTO} \langle \text{name} \rangle ;$
 $\mid \text{DEQUEUE} \langle \text{name} \rangle \langle \text{step} \rangle ;$
 $\mid \text{FREE} \langle \text{quantum} \rangle ;$
 $\mid \text{ENTER} \langle \text{hexalist} \rangle ;$
 $\mid \text{CALL} \langle \text{proc-ref} \rangle ;$
 $\mid \text{RETURN} \langle \text{result} \rangle ;$
 $\mid \text{ALLOCATE} \langle \text{alloc-list} \rangle ;$
 $\mid \text{EXIT} ;$
 $\mid \text{EXIT} \langle \text{decicst} \rangle ;$
 $\mid \text{TRACE} ;$
 $\mid \text{STOP TRACE} ;$
 $\mid \text{WAKE} ;$
 $\mid \text{SILENCE} ;$
 $\mid \text{COMMAND} \langle \text{const} \rangle ;$
 $\mid \text{CODE} \langle \text{inst-list} \rangle ;$
 $\mid \text{SAVE} \langle \text{cell} \rangle ;$
 $\mid \text{RESET} \langle \text{cell} \rangle ;$
 $\mid \langle \text{composite} \rangle$

$\langle \text{step} \rangle ::= \langle \text{null} \rangle \mid \langle \text{const} \rangle \mid \langle \text{cell} \rangle$

$\langle \text{quantum} \rangle ::= \langle \text{cell} \rangle \mid \langle \text{const} \rangle$

$\langle \text{hexalist} \rangle ::= \langle \text{hexacst} \rangle \mid \langle \text{hexacst} \rangle, \langle \text{hexalist} \rangle$

$\langle \text{result} \rangle ::= \langle \text{null} \rangle \mid (\langle \text{cell} \rangle) \mid (\langle \text{const} \rangle) \mid (\langle \text{regname} \rangle)$
 $\langle \text{alloc-list} \rangle ::= \langle \text{name} \rangle \mid \langle \text{name} \rangle, \langle \text{alloc-list} \rangle$
 $\langle \text{inst-list} \rangle ::= \langle \text{inst} \rangle \mid \langle \text{inst} \rangle, \langle \text{inst-list} \rangle$
 $\langle \text{inst} \rangle ::= \langle \text{const} \rangle \langle \text{const} \rangle \mid \langle \text{const} \rangle \langle \text{cell} \rangle$

7. COMPOSITE STATEMENTS.

$\langle \text{composite} \rangle ::= \text{WHEN} \langle \text{condition} \rangle \langle \text{statement} \rangle$
 $\quad \mid \text{UNLESS} \langle \text{condition} \rangle \langle \text{statement} \rangle$
 $\quad \mid \text{WHILE} \langle \text{condition} \rangle \langle \text{con} \rangle \langle \text{stmt-list} \rangle \text{END} ;$
 $\quad \mid \text{UNTIL} \langle \text{condition} \rangle \langle \text{con} \rangle \langle \text{stmt-list} \rangle \text{END} ;$
 $\quad \mid \text{LOOP} \langle \text{stmt-list} \rangle \text{END} ;$
 $\quad \mid \langle \text{if-st} \rangle$
 $\quad \mid \langle \text{case-st} \rangle$
 $\langle \text{con} \rangle ::= \langle \text{null} \rangle \mid \text{THEN}$
 $\langle \text{endif} \rangle ::= \text{END} \mid \text{FI}$
 $\langle \text{if-st} \rangle ::= \text{IF} \langle \text{condition} \rangle \langle \text{con} \rangle \langle \text{stmt-list} \rangle \langle \text{alternative} \rangle \langle \text{endif} \rangle ;$
 $\langle \text{alternative} \rangle ::= \langle \text{null} \rangle \mid \text{ELSE} \langle \text{stmt-list} \rangle$
 $\langle \text{case-st} \rangle ::= \text{CASE} \langle \text{variable} \rangle \text{OF} \langle \text{decicst} \rangle \langle \text{actions} \rangle \text{END} ;$
 $\langle \text{actions} \rangle ::= \langle \text{pre-st} \rangle \mid \langle \text{pre-st} \rangle \langle \text{actions} \rangle$
 $\langle \text{pre-st} \rangle ::= \langle \text{prefix-list} \rangle) \langle \text{stmt-list} \rangle$
 $\langle \text{prefix-list} \rangle ::= \langle \text{decicst} \rangle \mid \langle \text{decicst} \rangle \langle \text{prefix-list} \rangle$

8. ASSIGNMENT, EXPRESSION, CONDITION.

$\langle \text{pter-repl} \rangle ::= \langle \text{pter-list} \rangle \rightarrow \langle \text{name} \rangle ;$
 $\langle \text{pter-list} \rangle ::= \langle \text{name} \rangle \mid \langle \text{name} \rangle, \langle \text{pter-list} \rangle$
 $\langle \text{replacement} \rangle ::= \langle \text{lefthand} \rangle = \langle \text{expression} \rangle ;$
 $\langle \text{lefthand} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{variable} \rangle, \langle \text{lefthand} \rangle$
 $\langle \text{variable} \rangle ::= \langle \text{cell} \rangle \mid \langle \text{regname} \rangle$

$\langle \text{expression} \rangle ::= \langle \text{unary} \rangle \langle \text{primary} \rangle \langle \text{secondary} \rangle$
 $\langle \text{unary} \rangle ::= \langle \text{null} \rangle \mid + \mid - \mid /$
 $\langle \text{primary} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{const} \rangle \mid \langle \text{proc-ref} \rangle$
 $\langle \text{secondary} \rangle ::= \langle \text{null} \rangle \mid \langle \text{operation} \rangle \langle \text{secondary} \rangle$
 $\langle \text{operation} \rangle ::= \langle \text{operator} \rangle \langle \text{cell} \rangle \mid \langle \text{operation} \rangle \langle \text{const} \rangle \mid ? \langle \text{const} \rangle$
 $\langle \text{operator} \rangle ::= + \mid - \mid ! \mid \& \mid / \mid \backslash \mid \times$
 $\langle \text{proc-ref} \rangle ::= \langle \text{name} \rangle \langle \text{parameters} \rangle$
 $\langle \text{parameters} \rangle ::= \langle \text{null} \rangle \mid (\langle \text{parmlist} \rangle)$
 $\langle \text{parmlist} \rangle ::= \langle \text{parm} \rangle \mid \langle \text{parm} \rangle , \langle \text{parmlist} \rangle$
 $\langle \text{parm} \rangle ::= \langle \text{cell} \rangle \mid \langle \text{const} \rangle \mid @ \mid \# \mid \times \langle \text{name} \rangle$
 $\langle \text{condition} \rangle ::= (\langle \text{test} \rangle)$
 $\langle \text{test} \rangle ::= \text{AOV} \mid \text{BOV} \mid \text{MOV} \mid \langle \text{expression} \rangle$
 $\quad \mid \langle \text{expression} \rangle \langle \text{relation} \rangle 0$
 $\langle \text{relation} \rangle ::= = \mid / = \mid > \mid > = \mid < \mid < =$

APPENDIX IV

COMPARISON OF THE L.S.D AND P.L. MODULAR ONE STATEMENTS

TYPES	L.S.D.	P.L. MODULAR ONE	COMMENTS
ASSIGNMENT & STORAGE CONTROL	SET, SETF, =>	=, ->=	see note a
	--	DEQUEUE	no stack in LSD
	--	ALLOCATE, FREE	no controlled storage in LSD
	INC, DEC, CLEAR	--	these statements are irrelevant in P.L. Modular One. They can be replaced by: $\mathcal{Q} = \mathcal{Q} + 1$ $\mathcal{Q} = \mathcal{Q} - 1$, $\mathcal{Q} = 0$;
CONDITIONAL STATEMENTS	{IF IFF} cond. THEN st. ELSE st.;	IF cond. THEN st.... [ELSE st....] FI;	there is no possibility of coding a block in a LSD THEN or ELSE clause. This seriously reduces the generality of the statement.
	{IF IFF} cond. : st.	{WHEN UNLESS} cond. st.	in LSD no block in the THEN (see above)
TRANSFER OF CONTROL	GOTO label-const.	GOTO label-const.	--
	GOINDTO lab-var.	GOTO lab-var.	--
	GOTOVIA label	--	cannot be implemented on Mod-1.

ORGANISATION	--	BEGIN END;	no block structure in LSD.
STATEMENTS	--	EXIT [const] ;	--
	--	null statement	--
SWITCHING	SWITCH ;	CASE OF ...;	although syntactically different these two statements are equivalent.
LINKAGE	CALL ... ;	CALL ... ;	see note b
	RETURN;;	RETURN [value] ;	
REPETITION STATEMENTS	LOOP [(expression UNTIL expression [STEP expression]) =>]operand ; ENDLOOP ;	{ WHILE } condition { UNTIL } END; LOOP ... END ;	SEE NOTE c
IN-LINE	CODE ENDCODE;	CODE ;	--
CODING	--	ENTER ;	--
TRACING	--	TRACE; STOPTRACE; SILENCE; WAKE;	No tracing facilities in LSD.
STOP	STOP; DUMP;	--	--
MISC. STATEMENTS	--	COMMAND ... ;	no Executive interface in LSD.
	--	SAVE..; RESET..;	--

note a: - in LSD the assignment => is considered as an operator
 - SET expression; is equivalent to the P.L. Mod-1 $\mathcal{Q}$ = expression;
 - There is no floating point operation in P.L. Modular One.

note b: LSD makes a distinction between INPUT and OUTPUT parameters in a procedure. This feature which existed in the first Algol version (58) has proved to be useless and unnecessarily restrictive.

note c: In P.L. Mod-1 it is impossible to specify a control variable in a repetition statement because a register would have to be allocated to it which could not be freely used by the programmer. The user is responsible for the incrementation of the control variable.

APPENDIX V

THE "SYDELA" ANALYSER AND SOME SERVICES

1. FORMAT OF THE "SYDELA" GENERATED TABLES.

The "SYDELA" compiler processes the description of transition diagrams to produce the tables which drive the syntactical analysis.

The 3 tables generated by "SYDELA" are:

- a - the symbol table, with the format described in chapter VI.4.1, is initialised with the reserved words or keywords of the language;
- b - the automaton table gives the offset of the entry points of each diagram in the syntax table;
- c - the syntax table is the internal representation of the diagrams. Each arc spreads over two 16 bits words (or four 8 bits bytes).

The format of the syntax table depends on the type of arc. These are:

- a - labelled normal transition

CODE	SEM	ERR	NST
------	-----	-----	-----

- CODE is the code of the class labelling the arc;
- SEM is the number of the action to be executed;
- ERR is the code of the error diagnostic to be printed;
- NST is the relative position of the arc to be analysed if the match is successful.

b - VOID and SKIP transitions

x	SEM	ERR	NST
---	-----	-----	-----

- x = 0 for VOID

x = 1 for SKIP

c - PULL transition

2	SEM	ERR	ST
---	-----	-----	----

- ST is the label of the output of the diagram (1 if the output is unlabelled).

d - EXIT transition

3	SEM	ERR	0
---	-----	-----	---

e - PUSH transition

4	AUT	NST1	NST2
---	-----	------	------

- AUT is the identification of the diagram to be entered;

- NST1 and NST2 are the relative addresses of respectively the first and second output transitions of the diagram (multiple output diagrams).

f - counter transitions

x	CT	VAL	ERR	NST
---	----	-----	-----	-----

- x is 5 for >, 6 for <, 7 for =, 8 for +,
9 for -, 10 for ≠, 11 for :=.

- CT and VAL are respectively the number of the counter and the value referred to in the test or assignment.

2. THE "SYDELA" ANALYSER.

comment following is an Algol-like description of the "SYDELA" syntactical analyser. The identifiers *f1to8* work as selectors for subfields of a 16 bits word.

f1to8 extracts the first byte,
f9to16 extracts the second byte,
f9to12 extracts the first half of the second
 byte,
f13to16 extracts the second half of the second
 byte;

outerloop: feed-the-next-analytical-type;

comment extraction of the fields of the syntax table;

```

loop:   cod := f1to8  of syntax [ptr];
        sem := f9to16 of syntax [ptr];
        cnt := f9to12 of syntax [ptr];
        val := f13to16 of syntax [ptr];
        err := f1to8  of syntax [ptr + 1] ;
        nst := f9to16 of syntax [ptr + 1] ;
        if nst >> 128 then dis := 256 - nst;
                else dis := -nst;
        if cod < 11 then goto special cod ;

        comment normal labelled transition;
        if cod = type then begin if err ≠ 0 then error(err);
                                if sem ≠ 0 then semantic(sem);
                                ptr := ptr+dis; goto outerloop;
                                end;
        else begin ptr := ptr+2; goto loop; end;

```

```

        comment VOID transition;
special[0]:  if err  $\neq$  0 then error(err);
              if sem  $\neq$  0 then semantic(sem);
              ptr := ptr + dis; goto loop;

        comment SKIP transition;
special[1]:  if err  $\neq$  0 then error(err);
              if sem  $\neq$  0 then semantic(sem);
              ptr := ptr + dis; goto outerloop;

        comment PULL transition;
special[2]:  if err  $\neq$  0 then error(err);
              if sem  $\neq$  0 then semantic(sem);
              ptr := stack[top,nst];
              top := top - 1; goto loop;

        comment EXIT transition;
special[3]:  if err  $\neq$  0 then error(err);
              if sem  $\neq$  0 then semantic(sem);
              ptr := stack[top,0] + 2;
              top := top - 1; goto loop;

        comment PUSH transition;
special[4]:  top := top + 1;
              stack[top,0] := ptr; stack[top,2] := ptr + dis;
              if err > 128 then stack[top,1] := ptr - err + 256;
                      else stack[top,1] := ptr - err;
              ptr := automat[sem] + 1; goto loop;

```

```

    comment counter test transitions;

special[5]: if val < counter[cnt] then begin ptr := ptr + 2;
                                     goto loop; end;
    if err ≠ 0 then error(err); ptr := ptr + dis; goto loop;
special[6]: if val ≥ counter[cnt] then begin ptr := ptr + 2;
                                     goto loop; end;
    if err ≠ 0 then error(err); ptr := ptr + dis; goto loop;
special[7] if val ≠ counter[cnt] then begin ptr := ptr + 2;
                                     goto loop; end;
    if err ≠ 0 then error(err); ptr := ptr + dis; goto loop;
special[10]: if val = counter[cnt] then begin ptr := ptr + 2;
                                     goto loop; end;
    if err ≠ 0 then error(err); ptr := ptr + dis; goto loop;

    comment counter increment and decrement transitions;

special[8]: if err ≠ 0 then error(err);
            counter[cnt] := counter[cnt] + val;
            ptr := ptr + dis; goto loop;
special[9]: if err ≠ 0 then error(err);
            counter[cnt] := counter[cnt] - val;
            ptr := ptr + dis; goto loop;

    comment counter assignment transition;

special[11]: if err ≠ 0 then error(err);
            counter[cnt] := val;
            ptr := ptr + dis; goto loop;

```

3. THE "SEARCH" AND "COPY" PROCEDURES.

comment The "SEARCH" procedure consults the symbol table. The variable "level" specifies the lexical depth at which the research must stop. If "level" is negative, a reserved word is searched.

The "COPY" procedure enters a symbol in the symbol table at the current lexical depth.

The following variables are used in the procedures:

"symbol"	- the 16 bits word symbol table
"table"	- a 8 bits character array overlaying the symbol table
"synunit"	- an array containing the name to be searched or entered in the table. The first character is the length of the name + 1
"vecth"	- the hash vector
"hash"	- the hash value of the current name
"ptid"	- a pointer to the entry in the symbol table
"lexlev"	- a vector giving the base of the different lexical levels in the table
"topres"	- the address of the last reserved word in the table
"freemax"	- the address of the first free location in the table
"size"	- is the length of the information field of an element in the table;

comment the "SEARCH" module;

```

    lu := synunit[1]; found := false; type := 0;
    i := vecth[hash] ; ptid := 0;
loop:  if i = 0 then goto nf;
    k := 2*i + 2;
    for j := 1 step 1 until lu do
        if table[k+j] ≠ synunit[j] then goto next;
    if level < 0 & i ≤ topres then goto f;
    if level ≥ 0 & i > lexlev[level] then goto f;
next:  i := symbol[i]; goto loop;
f:     found := true; type := table[k]; ptid := i;
nf:    ;

```

comment the "COPY" module;

```

symbol[freemax] := vecth[hash];
vecth[hash] := freemax;
j := 2*freemax + 2; lu := synunit[1] ;
i := freemax + (lu/2) + 2; freemax := i + size;
table[j] := 0;
for k := 1 step 1 until lu do table[j+k] := synunit[k] ;
table[j+lu+1] := ' ';

```

4. SOME SERVICES.

comment the "PUSH" module;

currentlevel := currentlevel - 1;

lexlev[currentlevel] := freemax;

comment the "PULL" module;

currentlevel := currentlevel - 1;

i := lexlev[currentlevel];

for k := 0 step 1 until 255 do begin

L := vecth[k];

loop: if L i then begin symbol[k] := L; goto E; end;

else begin L := symbol[L] ; goto loop; end;

E: end;

freemax := i;

comment the "TYPING" procedure;

table[2 * ptid + 2] := type;

comment the "GETi" module;

j := table[2*ptid + 3];

j := ptid + (j/2) + 1;

information := symbol[j+i];

comment the "SETi" module;

j := table[2*ptid + 3]; j := ptid + (j/2) + 1;

symbol[j+i] := information;

A "SYDELA" DESCRIPTION OF ALGOL

DATE: 19/07/73
TIME: 16H46M29S
THE OPTIONS SPECIFIED ARE:
NOLIST
NOPUNCH

25 JANUARY 1973

```

&
& *****
&
& REFERENCES:
&
&     A- FOR THE BASIC SYNTAX:
&         ALGOL REFERENCE MANUAL VERSION 2 (PUB. # 60306100)
&
&     B- FOR THE ADDITIONAL ULCC CHARACTER SET:
&         ULCC BULLETIN 3.3/1
&
& *****
&
&
& *****
&
& COMMENTS:
&
&     A- FOUR SCANNING MODES FOR THE NAMES:
&         MODE = 1    LABEL DEFINITION
&         MODE = 2    LABEL REFERENCE
&         MODE = 3    OTHER DEFINITIONS
&         MODE = 4    OTHER REFERENCES
&
&     B- IN MODE 1 THE ':' FOLLOWING A LABEL IS SCANNED
&         WITH THE NAME. THE CODE 'REFLAB' OR 'DEFLAB' IS
&         RETURNED.
&
&     C- SEMANTIC 1 SKIPS UNTIL 'END', 'EOP', ';',
&         'ELSE' OR 'END' IS SCANNED.
&
&     D- SEMANTIC 2 STOPS THE ANALYSER.
&
& *****
&

```

```

&
&
& *****&
&
&      ERRORS:&
&      ALG001  INVALID CODE NUMBER.&
&      ALG002  MISSING SEMI COLON.&
&      ALG003  PREMATURE END OF SOURCE DECK.&
&      ALG004  MISSING END STATEMENT.&
&      ALG005  INVALID STATEMENT.&
&      ALG006  INVALID DESIGNATIONAL EXPRESSION.&
&      ALG007  INVALID ARITHMETIC EXPRESSION.&
&      ALG008  INVALID BOOLEAN EXPRESSION.&
&      ALG009  INVALID DECLARATION OR FORMAL PARAMETER.&
&      ALG010  MISSING ASSIGNMENT SYMBOL.&
&      ALG011  SYNTACTICAL ERROR.&
&      ALG012  MISSING LEFT OR RIGHT BRACKET.&
&      ALG013  MISSING COLON.&
&      ALG014  MISSING THEN OR ELSE.&
&      ALG015  INVALID EXPRESSION.&
&      ALG016  UNMATCHED LEFT PARENTHESIS.&
&      ALG017  DOUBLE DEFINED LABEL.&
&      ALG018  INVALID FOR CONTROL VARIABLE.&
&      ALG019  MISSING UNTIL OR DO.&
&      ALG020  INVALID CHARACTER.&
&      ALG021  INVALID COMPOSITE CHARACTER.&
&      ALG022  INVALID NUMBER.&
&      ALG023  INVALID ALGOL KEYWORD.&
&
& *****&
&

```

```

~
&
&
& *****&
&
& AUTOMATON TABLE: &
&
& @01 'ALGOL DECK' &
& @02 'UNLABELLED PROGRAM' &
& @03 'DECLARATIONS' &
& @04 'EXPRESSION' &
& @05 'SIMPLE EXPRESSION' PRIMARY ENTRY &
& @06 'SIMPLE EXPRESSION' SECONDARY ENTRY &
& @07 'SIMPLE EXPRESSION' SECONDARY ENTRY &
& @08 'SIMPLE EXPRESSION' SECONDARY ENTRY &
& @09 'SIMPLE EXPRESSION' SECONDARY ENTRY &
& @10 'SIMPLE EXPRESSION' SECONDARY ENTRY &
& @11 'SIMPLE EXPRESSION' SECONDARY ENTRY &
& @12 'SIMPLE EXPRESSION' SECONDARY ENTRY &
& @13 'BOOLEAN PRIMARY' &
& @14 'ARITHMETIC PRIMARY' &
& @15 'ACTUAL PARAMETER LIST' &
& @16 'STATEMENT' &
& @17 'UNLABELLED FOR STATEMENT' &
& @18 'UNLABELLED UNCONDITIONAL STATEMENT' &
& @19 'ARRAY INDEXING' &
& @20 'PROCEDURE DECLARATION' SECONDARY ENTRY TO @03 &
& @21 'SOURCE PROGRAM' SECONDARY ENTRY TO @02 &
&
& *****&
&

```

```

& ****&
&
&          TERMINAL SYMBOLS:
&
& ****&
CCDE = 12
*' ' = ?
CCDE = 13
*'(' = ?
CCDE = 14
*')' = ?
CCDE = 15
*'+ ' = ?      & + OR -
CCDE = 16
*': ' = ?      & : OR ..
CCDE = 17
*'; ' = ?      & ; OR .,
CCDE = 18
*':=' = ?      & := . = OR .. =
CCDE = 19
*'(/' = ?      & (/ OR LEFTSQBR
CCDE = 20
*')/' = ?      & /) OR RIGHTSQBR
CCDE = 21
*'FOR' = ?      & END OF RECORD
CCDE = 22
*'NAME' = ?      & NEUTRAL CODE
CCDE = 23
*'PARM' = ?      & PROCEDURE PARAMETER
CCDE = 24
*'STRING' = ?      & STRING OR STRING PARAMETER
CCDE = 25
*'NUMBER' = ?      & REAL OR INTEGER CONSTANT
CCDE = 26
*'SWITCHNAME' = ?
CCDE = 27
*'BOCLARNAME' = ?
CCDE = 28
*'ARITARNAME' = ?
CCDE = 29
*'BOCLNAME' = ?

```



```

&
&
& *****&
&
& OTHER RESERVED WORDS: &
&
& *****&
**
'NOT' , & ALSO TERMINAL -
'GOTO' ,
'IF' ,
'THEN' ,
'ELSE' ,
'FOR' ,
'DO' ,
'STEP' ,
'UNTIL' ,
'WHILE' ,
'COMMENT' ,
'BEGIN' ,
'END' ,
'OWN' ,
'BOOLEAN' ,
'ARRAY' ,
'SWITCH' ,
'PROCEDURE' ,
'String' ,
'LABEL' ,
'VALUE' ,
'CODE' ,
'EOP' ,

```

```

& ****&
&
& PRESETTING THE STANDARD FUNCTION NAMES: &
&
& ****&
CODE = 61
*'BOCLEFUNNAME' = ?
*'ARITFUNNAME' = 'ENTIER' ,
                'INLIST' ,
                'OUTLIST' ,
                'INPUT' ,
                'OUTPUT' ,
                'INREAL' ,
                'OUTREAL' ,
                'INARRAY' ,
                'OUTARRAY' ,
                'INCHARACTER' ,
                'OUTCHARACTER' ,
                'GETARRAY' ,
                'PUTARRAY' ,
                'CLOCK' ,
                'HLIM' ,
                'VLIM' ,
                'HEND' ,
                'VEND' ,

```

'NODATA'	,
'TABULATION'	,
'FORMAT'	,
'SYSPARAM'	,
'STRINGELEMENT'	,
'CHLENGTH'	,
'APTHOFLW'	,
'BADDATA'	,
'PARITY'	,
'EOF'	,
'REWIND'	,
'UNLOAD'	,
'SKIPF'	,
'SKIPB'	,
'ENDFILE'	,
'BACKSPACE'	,
'IOLTH'	,
'MODE'	,
'CONNECT'	,
'DUMP'	;

&
&
&

```

& ****&
&
&      AUTOMATA:
&
& ****&
&
&      AUTOMATON # 01 : 'ALGOL DECK'
&
&
@01  -1  :  'BEGIN'          3 , 2          & PUSH
        :  'CODE'           , 4
        :  *'FOR'           2 , 1          & STOP ANALYSE
        :  SKIP              , 1
        -2  :  PUSH 21        , 3
        -3  :  'EOP'         , 1
        :  *'FOR'           2 , 1          & STOP ANALYSE
        :  SKIP              , 3
        -4  :  *'NUMBER'     , 5
        :  VOID              , 6          ERRO 001
        -5  :  *';'          , 6
        :  VOID              , 6          ERRO 002
        -6  :  'PROCEDURE'   , 7
        :  *'FOR'           , 3          ERRO 003
        :  SKIP              , 6
        -7  :  PUSH 20        , 8
        -8  :  'EOP'         , 1
        :  *';'          , 3
        :  VOID              , 3          ERRO 002
        :
&
&
&      AUTOMATON # 02 : 'UNLABELLED PROGRAM'
&
&
@02  -1  :  'BEGIN'          3 , 2          & PUSH
        :  EXIT
@21  -2  :  VOID              5 , 3          & MODE = 3
        -3  :  'COMMENT'     , 4
        :  VOID              , 5
        -4  :  *';'          , 5
        :  *'FOR'           , 5          ERRO 003
        :  'EOP'           4 , 5          ERRO 004 & PULL

```

```

      : SKIP , 4
-5 : PUSH 03 , 6
      : VOID 6 , 7 & MODE = 4
-6 : *';' , 3
      : VOID , 3 ERRC 002
-7 : PUSH 16 , 8
-8 : *';' , 9
      : 'END' 1 ,12 & SKIP ALL RUBBISH
      : 'EOP' ,12 ERRO 004
      : *'EOR' ,12 ERRO 003
      : SKIP , 8 ERRO 005
-9 : 'COMMENT' ,10
      : VOID , 7
-10 : *';' , 7
      : 'EOP' ,12 ERRO 004
      : *'EOR' ,12 ERRO 003
      : SKIP ,10
-12 : PULL 4 & PULL
      ;

&
& AUTOMATON # 03 : 'DECLARATIONS'
&
@03 -1 : 'SWITCH' , 2
      : 'ARRAY' 7 , 7 & FLAG := ARIT
      : 'OWN' , 8
      : *'ARITHM' 7 ,10 & FLAG := ARIT
      : 'PROCEDURE' 7 ,11 & FLAG := ARIT
      : 'BOOLEAN' 8 ,10 & FLAG := BOOL
      : EXIT
-2 : *'NAME' 9 , 3 & TYPE == SWITCHNAME
      : VOID , 3 ERRO 009
-3 : *':=' 6 , 4 & MODE = 4
      : VOID 6 , 4 ERRO 010 & MODE = 4
-4 : PUSH 04 , 5.5
      : VOID , 6
-5 : VOID , 6 ERRO 006
-6 : PULL 5 & MODE = 3
-7 : *'NAME' 10 ,12 & TYPE == FLAG.ANAME
      : VOID ,12 ERRO 009

```

-8	:	*'ARITHM'	7 , 9		& FLAG := ARIT
	:	'BOOLEAN'	8 , 9		& FLAG := BOCL
	:	VOID	7 , 9	ERRR 011	& FLAG := ARIT
-9	:	'ARRAY'	, 7		
	:	*'NAME'	11,21		& TYPE == FLAG.NAME
	:	VOID	,21	ERRR 011	
-10	:	'PROCEDURE'	,11		
	:	VOID	, 9		
@20 -11	:	*'NAME'	12,23		& TYPE == FLAG.FUNNAME
	:	VOID	,23	ERRR 009	
-12	:	*','	, 7		
	:	*'(/'	6 ,14		& MODE = 4
	:	VOID	6 ,14	ERRR 012	& MODE = 4
-14	:	PUSH 04	,16.15		
	:	VOID	,15		
-15	:	VOID	,16	ERRR 007	
-16	:	*':'	,17		
	:	VOID	,17	ERRR 013	
-17	:	PUSH 04	,18.19		
	:	VOID	,19		
-18	:	*','	,14		
	:	*'/'	5 ,20		& MODE = 3
	:	VOID	5 ,20	ERRR 012	& MODE = 3
-19	:	VOID	,18	ERRR 007	
-20	:	*','	, 7		
	:	PULL			
-21	:	*','	,22		
	:	PULL			
-22	:	*'NAME'	11,21		& TYPE == FLAG.NAME
	:	PULL		ERRR 011	
-23	:	VOID	3 ,24		& PUSH
-24	:	*'('	,25		
	:	*';'	,29		
	:	VOID	,29	ERRR 002	
-25	:	*'NAME'	13,26		& TYPE == PARM
	:	VOID	,26	ERRR 009	
-26	:	*','	,26		
	:	*')'	,27		

	:	VOID	,27	ERRC 011	
-27	:	*'NAME'	,28		
	:	*';'	,29		
	:	VOID	,29	ERRC 002	
-28	:	*':'	,30		
	:	VOID	,30	ERRC 013	
-30	:	*'{'	,25		
	:	VOID	,29	ERRC 011	
-29	:	'COMMENT'	,50		
	:	VOID	,31		
-50	:	*';'	,31		
	:	*'EOR'	,31	ERRC 003	
	:	'EOP'	,31	ERRC 004	
	:	SKIP	,50		
-31	:	'VALUE'	6 ,32		& MODE = 4
	:	*';'	6 ,34		& MODE = 4
	:	VOID	6 ,34	ERRC 002	& MODE = 4
-32	:	*'PARM'	,33		
	:	VOID	,33	ERRC 009	
-33	:	*','	,32		
	:	*';'	,34		
	:	VOID	,34	ERRC 002	
-34	:	'COMMENT'	,35		
	:	VOID	,36		
-35	:	*';'	,36		
	:	*'EOR'	,36	ERRC 003	
	:	'EOP'	,36	ERRC 004	
	:	SKIP	,35		
-36	:	VOID	7 ,37		& FLAG := ARIT
-37	:	'STRING'	14 ,39		& T := STRING
	:	'LABEL'	15 ,39		& T := DEFLAB
	:	'SWITCH'	16 ,39		& T := SWITCH
	:	*'ARITHM'	7 ,38		& FLAG := ARIT
	:	'BOOLEAN'	8 ,38		& FLAG := BOOL
	:	'PROCEDURE'	18 ,39		& T := FLAG.FUNNAME
	:	'ARRAY'	17 ,39		& T := FLAG.ANAME
	:	VOID	,41		
-38	:	'ARRAY'	17 ,39		& T := FLAG.ANAME
	:	'PROCEDURE'	18 ,39		& T := FLAG.FUNNAME

```

      : VOID      19,39      & T := FLAG.NAME
-39 : *'PARM'     20,40      & TYPE == T
      : VOID      ,40      ERRO 009
-40 : *', '       ,39
      : *'; '      ,34
      : VOID      ,34      ERRO 002
-41 : 'CODE'      ,43
      : PUSH 16    ,42
-43 : *'NUMBER'   ,42
      : VOID      ,42      ERRO 001
-42 : PULL        4         & PULL
      ;

&
&      AUTOMATION # 04 : 'EXPRESSION'
&
@04 -1 : 'IF'      , 2
      : PUSH 05    ,10.13
      : VOID      ,14
-2 : PUSH 04      , 3.4
      : VOID      , 3
-3 : VOID        , 4      ERRO 008
-4 : 'THEN'      , 5
      : VOID      , 5      ERRO 014
-5 : PUSH 05     , 6.7
      : VOID      , 8
-6 : 'ELSE'      , 9
      : VOID      , 9      ERRO 014
-7 : 'ELSE'      ,12
      : VOID      ,12      ERRO 014
-8 : 'ELSE'      ,15
      : VOID      ,15      ERRO 014
-9 : PUSH 04     ,10.11
      : VOID      ,11
-10 : PULL, 1
-11 : VOID       ,10      ERRO 007
-12 : PUSH 04    ,14.13
      : VOID      ,14
-13 : PULL, 2
      & "ARITHMETIC EXPRESSION"
      & "BOOLEAN EXPRESSION"

```

```

-14 : VOID ,13 FRR0 008
-15 : PUSH 04 ,16.16
      : VOID ,17
-16 : VOID ,17 FRR0 006
-17 : EXIT
      ;

```

& "DESIGNATIONAL EXPRESSION"

```

&
&
&
AUTOMATON # 05 : 'SIMPLE EXPRESSION'

```

```

@05 -1 : *'+' , 2
      : 'NOT' , 7
      : *'NUMBER' , 3
      : *'ARITNAME' , 3
      : *'PARM' , 3
      : *'ARITFUNNAME' , 11
      : *'ARITARNAME' , 12
      : *'(' , 13
      : *'TRUE' , 7
      : *'BOOLNAME' , 7
      : *'BOOLFUNNAME' , 20
      : *'BOOLARNAME' , 21
      : *'SWITCHNAME' , 22
      : VOID 21,24

```

& M=MODE; MODE=2; SEARCH;

```

-2 : PUSH 14 , 3
@11 -3 : *'OPER' , 2
      : *'+' , 2
      : *'RELATION' , 4
      : PULL, 1

```

& "SIMPLE ARITHM. EXPR."

```

-4 : PUSH 05 , 6.5
      : VOID , 5
-5 : VOID , 6

```

ERPC 007

& "SIMPLE BOOL. EXPRE."

```

-6 : PULL, 2
-7 : PUSH 13 , 8
@12 -8 : *'BOOLOP' , 9
      : PULL, 2
-9 : 'NOT' , 10
      : VOID , 10
-10 : PUSH 13 , 8

```

& "SIMPLE BOOL. EXPR."

```

@C6  -11 : PUSH 15          , 3
@07  -12 : PUSH 19          , 3
      -13 : PUSH 04          ,14.15
      : VOID                ,16
      -14 : *')'            ,17
      : VOID                ,17      ERRO 016
      -15 : *')'            , 6
      : VOID                , 6      ERRO 016
      -16 : *')'            ,23
      : VOID                ,23      ERRO 016
      -17 : PULL, 1
@C8  -20 : PUSH 15          , 7
@09  -21 : PUSH 19          , 7
@10  -22 : PUSH 19          ,23
      -23 : EXIT
      -24 : *'REFLAB'        ,25
      : *'DEFLAB'           ,25
      : VOID                22,26      ERRO 015
      -26 : EXIT
      -25 : VOID            22,23
      ;

```

& "SIMPLE ARIT.EXPR."

& "SIMPLE DESI. EXPR."

& MODE = M

& "ERROR"

& MODE = M

```

&
&
&
AUTOMATON # 13 : 'PCCLEAN PRIMARY'

```

```

@13  -1 : *'('              , 2
      : *'TRUE'             , 5
      : *'BOOLFUNNAME'      , 7
      : *'BOOLNAME'         , 5
      : *'BOOLLARNAME'      , 8
      : PUSH 05              ,10.9
      : VOID                , 9
      -2 : PUSH 04           , 3.4
      : VOID                , 3
      -3 : VOID              , 4      ERRO 008
      -4 : *')'              , 5
      : VOID                , 5      ERRO 016
      -5 : PULL
      -7 : PUSH 15           , 5
      -8 : PUSH 19           , 5

```

```

-9 : VOID , 5 ERRO 008
-10 : *'RELATION' , 11
      : VOID , 5 ERRO 008
-11 : PUSH 05 , 5.12
      : VOID , 12
-12 : VOID , 5 ERRO 008
      ;

```

```

&
&
&

```

AUTOMATON # 14 : 'ARITHMETIC PRIMARY'

```

@14 -1 : *'(' , 2
      : *'NUMBER' , 5
      : *'ARITFUNNAME' , 6
      : *'ARITNAME' , 5
      : *'PARM' , 5
      : *'ARITARNAME' , 7
      : VOID , 5 ERRO 007
-2 : PUSH 04 , 4.3
      : VOID , 3
-3 : VOID , 4 ERRO 007
-4 : *')' , 5
      : VOID , 5 ERRO 016
-5 : PULL
-6 : PUSH 15 , 5
-7 : PUSH 19 , 5
      ;

```

```

&
&
&

```

AUTOMATON # 15 : 'ACTUAL PARAMETER LIST'

```

@15 -1 : *'(' , 2
      : PULL
-2 : *'SWITCHNAME' , 3
      : *'BOOLARNAME' , 4
      : *'ARITARNAME' , 5
      : *'BOOLFUNNAME' , 6
      : *'ARITFUNNAME' , 7
      : *'STRING' , 8
      : PUSH 04 , 8.8
      : VOID , 8

```

```

-3 : *,' , 2
    : *(' , 9
    : PUSH 10 , 8.8
    : VOID , 8
-4 : *,' , 2
    : *)' , 9
    : PUSH 09 , 8.8
    : VOID , 8
-5 : *,' , 2
    : *)' , 9
    : PUSH 07 , 8.8
    : VOID , 8
-6 : *,' , 2
    : *)' , 9
    : PUSH 08 , 8.8
    : VOID , 8
-7 : *,' , 2
    : *)' , 9
    : PUSH 06 , 8.8
    : VOID , 8
-8 : *,' , 2
    : *)' , 9
    : VOID , 9
-9 : *'NAME' , 10
    : PULL
-10 : *': ' , 11
    : VOID , 11
-11 : *(' , 2
    : VOID , 2
    ;

```

ERRR 016

ERRR 013

ERRR 011

&
&
&

AUTOMATON # 16 : 'STATEMENT'

```

@16 -1 : VOID 23, 2
      -2 : *'DEFLAB' , 2
          : *'REFLAB' 24, 2
          : VOID 22, 3
      -3 : 'FOR' , 4

```

ERRR 017

& M=MODE; MODE=1; SWITCH SCAN.;

& TYPE == DEFLAB

& MODE = M

```

      : 'IF' , 5
      : PUSH 18 ,11
-4 : PUSH 17 ,11
-5 : PUSH 04 ,14.6
      : VOID ,14
-14 : VOID , 6      ERRO 008
-6 : 'THEN' , 7
      : VOID , 7      ERRO 014
-7 : VOID 23, 8      & M=MODE; MODE=1; SWITCH SCAN.;
-8 : *'DEFLAB' , 8      ERRO 017
      : *'REFLAB' 24, 8      & TYPE == DEFLAB
      : VOID 22, 9      & MODE = M
-9 : 'FOR' ,10
      : PUSH 18 ,12
-10 : PUSH 17 ,11
-11 : PULL
-12 : 'ELSE' ,13
      : PULL
-13 : PUSH 16 ,11
      :

```

```

&
&      AUTOMATON # 17 : 'UNLABELLED FOR STATEMENT'
&

```

```

@17 -1 : *'PARM' , 3
      : *'ARITNAME' , 3
      : *'ARITARNAME' , 2
      : PULL      ERRO 018
-2 : PUSH 19 , 3
-3 : *':=' , 4
      : VOID , 4      ERRO 010
-4 : PUSH 04 , 6.5
      : VOID , 5
-5 : VOID , 6      ERRO 007
-6 : 'WHILE' , 7
      : 'STEP' , 9
      : VOID ,14
-7 : PUSH 04 ,14.8
      : VOID , 8

```

```

-8 : VOID ,14 FRRD 007
-9 : PUSH 04 ,11.10
   : VOID ,10
-10 : VOID ,10 ERRO 007
-11 : 'UNTIL' ,12
   : VOID ,12 ERRO 019
-12 : PUSH 04 ,14.13
   : VOID ,13
-13 : VOID ,14 FRRD 007
-14 : *', ' , 6
   : 'DD' ,15
   : VOID ,15 ERRO 019
-15 : PUSH 16 ,16
-16 : PULL
   .;

```

&
&
&

AUTOMATON # 18 : 'UNLABELLED UNCONDITIONAL STATEMENT'

```

@18 -1 : PUSH 02 , 2
   : 'GOTO' , 3
   : *'ARITFUNNAME' , 5
   : *'ARITNAME' ,10
   : *'PARM' ,10
   : *'ARITARNAME' , 7
   : *'BOOLFUNNAME' ,19
   : *'BOOLNAME' ,23
   : *'BOOLARNAME' ,20
   : PULL ERRO 005
-2 : PULL
-3 : PUSH 04 , 4.4
   : PULL
-4 : PULL ERRO 006
-5 : *':=' ,11
   : PUSH 15 , 6
-6 : PULL
-7 : PUSH 19 ,10
-10 : *':=' ,11
   : VOID ,11 ERRO 010

```

```

-11 : *'ARITNAME'      ,12
      : *'PARM'         ,12
      : *'ARITARNAME'   ,13
      : *'ARITFUNNAME'  ,17
      : PUSH 04         ,16.14
      : VOID            ,14
-14 : VOID             ,16      ERRC 007
-12 : *':='            ,11
      : PUSH 11         ,16.14
      : VOID            ,14
-13 : PUSH 19          ,12
-16 : PULL
-17 : *':='            ,11
      : PUSH 15         ,18
-18 : PUSH 11          ,16.14
      : VOID            ,14
-19 : *':='            ,24
      : PUSH 15         , 6
-20 : PUSH 19          ,23
-23 : *':='            ,24
      : VOID            ,24      ERRC 013
-24 : *'BOOLNAME'      ,25
      : *'BOOLARNAME'   ,26
      : *'BOOLFUNNAME'  ,29
      : PUSH 04         ,27.30
      : VOID            ,27
-27 : VOID             ,30      ERRC 008
-25 : *':='            ,24
      : PUSH 12         ,27.30
      : VOID            ,27
-26 : PUSH 19          ,25
-29 : *':='            ,24
      : PUSH 15         ,31
-31 : PUSH 12          ,27.30
      : VOID            ,27
-30 : PULL
      :

```

```

&
&      AUTOMATON # 19 : 'ARRAY INDEXING'
&
@19  -1  : '*' '/'      , 2
      : VOID            , 2      ERRC 012
      -2  : PUSH 04      , 4.3
      : VOID            , 3
      -3  : VOID          , 4      ERRC 007
      -4  : '*' , '      , 2
      : '*' '/' '      , 5
      : VOID            , 5      ERRC 012
      -5  : PULL
      %

```

THE TOTAL NUMBER OF TRANSITION IS: 420

REFERENCES

- 1 - D. A. BELL & B. A. WICHMANN.

An ALGOL-like Assembly Language for a small Computer.

Software - Practice and Experience: Vol. 1 - Issue 1. (1971).

- 2 - M. BLOCH.

Memento de syntaxe FORTRAN.

Revue de l'Industrie Minérale. 15 December 1967.

- 3 - C. BOKSENBAUM.

Moyens et problèmes de la compilation sur petites machines.

Revue Française d'Informatique et de Recherche Opérationnelle:

N° B.2. May 1970.

- 4 - Elimination de la récursivité gauche dans une grammaire
context-free.

Revue Française d'Informatique et de Recherche Opérationnelle:

N° B.3. December 1970.

- 5 - V. J. CALDERBANK & M. CALDERBANK.

LSD Manual.

UKAEA publication CLM-PDN 9/71. (1971).

- 6 - CONTROL DATA CORPORATION.

ALGOL Reference Manual - Version 2.

CDC publication 603061000.

- 7 - M. E. CONWAY.

Design of a Separable Transition-Diagram Compiler.

Communications of the ACM: Vol. 6 Number 7. (1963).

- 8 - DEPARTMENT OF TRADE AND INDUSTRY.
Study of Small Computers (carried out by Urwick Dynamics Ltd).
DTI - 1970.
- 9 - L. DOUCET.
P. L. MODULAR ONE: a programming language for the CTL Modular
One computer - Reference Manual.
Institute of Computer Science - February 1972.
- 10 - J. FELDMAN & D. GRIES.
Translator Writing Systems.
Communications of the ACM: Vol. 11 Number 2. (1968).
- 11 - M. GRIFFITHS & M. PELTIER.
Grammar Transformation as an aid to Compiler production.
Université de Grenoble - IMAG, note technique. February 1968.
- 12 - D. W. HARVEY.
Answering DATEL Telephone Calls by PDP-8.
Software - Practice and Experience: Vol. 1 - Issue 3. (1971).
- 13 - INTERNATIONAL BUSINESS MACHINES LTD.
PL/I Optimizing Compiler - Programmer's Guide.
IBM publication SC33-0006-1. (1972).
- 14 - D. E. KNUTH.
An empirical study of FORTRAN Programs.
Software - Practice and Experience: Vol. 1 - Issue 2. (1971).
- 15 - M. C. NEWAY, P. C. POOLE & W. M. WAITE.
Abstract Machine Modelling to Produce Portable Software.
Software - Practice and Experience: Vol. 2 - Issue 2. (1972).

16 - C. R. SPOONER.

A Software Architecture for the 70's.

Software - Practice and Experience: Vol. 1 Issue 1. (1971).

17 - J. E. STOY & G. STRACHEY.

OS6: An experimental operating system for a small computer.

The Computer Journal: Vol. 15 Number 2. (1972).

18 - V. TIXIER.

Recursive functions of regular expressions in language analysis.

Technical Report CS58 - Computer Science Departement.

Stanford University. March 1967.

19 - N. WIRTH.

PL360: a programming language for the 360 computers.

Journal of the ACM: Vol. 15. (1968).

20 - N. WIRTH.

The Design of a PASCAL Compiler.

Software - Practice and Experience: Vol. 1 Issue 4. (1972).

21 - D. WOOD.

Syntax-Generated interpretation of programming languages.

Revue Française d'Informatique et de Recherche Opérationnelle:

N° B.2. May 1970.