

MTA Számítástechnikai és Automatizálási Kutató Intézet Budapest




COMPUTER AND AUTOMATION INSTITUTE
HUNGARIAN ACADEMY OF SCIENCES

THE AUTOMATIC GENERATION OF USER-ADAPTABLE APPLICATION-ORIENTED
LANGUAGE PROCESSORS BASED ON QUASI-PARALLEL MODULES

By

Thomas Miles Ratcliffe Ellis



A kiadásért felelős:

VAMOS TIBOR

ISBN 963 311 170 6

ISSN 0324-2951

PREFACE

During the last 30 years the computer has become an indispensable tool for scientists and technologists in all fields, and more recently has begun to play a vital role in increasingly broad areas of society. With the development of micro-electronics, computers and their applications will clearly play an ever more important role in all aspects of human activity as the twentieth century draws to a close. However, unless this vast increase in the effective power of computer systems is accompanied by a corresponding increase in their flexibility, much of the potential benefit may fail to be realised due to the (justifiable) unwillingness of their human users to adapt their behaviour to suit the requirements of the computer system when, in fact, it should be the reverse that takes place.

This dissertation describes how research over a number of years has led to an unusual approach to the construction of a particular class of computer systems which greatly increases their resulting flexibility and adaptability. The impetus for this work came from the writer's involvement with a particular subject area - the numerical control of machine tools (N.C.) - and for this reason the development of the various concepts and methodologies has taken place within the context of the same area. However the overall methodology derived is in no way restricted to this one class, but is equally applicable across a wide range of different applications, although it may fairly be claimed that the broad CAD/CAM (Computer Aided Design/Computer Aided Manufacturing) area is one of the most appropriate at the present time. For this reason the development of computer software in this area will first be examined in order to set the scene for the particular developments that will be described later. The research which, over a number of years, led to the development of a new approach to adaptable language processors is then described in some detail, while the final chapter evaluates the results of this work and indicates some possible future developments.

The conjunction of two quite disparate subjects (computer aided manufacturing and such fundamental computer science areas as compiling techniques, operating systems, etc.) makes it highly desirable to include a comprehensive bibliography, although the writer is all too aware that

some excellent works have undoubtedly been omitted due to language or other problems of access. Many of the books and papers are particularly relevant to one or more specific parts of the dissertation and will be referred to in the usual manner by the name of the author and year of publication at the appropriate point; a number of other relevant papers are, however, included in the bibliography but are not explicitly referred to, either because they are included primarily to give background information or because they are important works concerning an appropriate topic but, nevertheless, were not directly of assistance in the formulation of the writer's ideas.

Much of the early research described in this dissertation was carried out at the University of Sheffield, England. The spark that led to the final ideas discussed herein was, however, struck while the writer was visiting Budapest in 1976, and a major part of the later research was carried out while the writer was spending a year at the Computer and Automation Institute of the Hungarian Academy of Sciences (SZTAKI) as a British Council Scholar under the Anglo-Hungarian Cultural Exchange Agreement. He wishes to thank all those who made that visit possible, especially the University of Sheffield for granting him a year's leave of absence for the purpose, and the many people at SZTAKI who helped him in so many ways, both large and small. In particular, thanks are due to Tamás Forgács, Géza Gerhardt and József János for their comments and advice at several crucial stages of this research, and to József Hatvany for his guidance and support without which this dissertation might never have been completed. Finally he would like to thank his wife Maggie both for her patience during the many evenings and weekends when he was typing this dissertation, and for her encouragement which, above all, enabled the research described to be completed and presented in this form to the Hungarian Academy of Sciences.

CONTENTS

1. Background
 - 1.1 Automatic Programming and Numerical Control
 - 1.2 The Development of NC Languages
 - 1.3 The Growth of APT-like Languages
2. Application-Oriented Languages and their Processing
 - 2.1 General Principles
 - 2.2 An Outline of the NELAPT Processor
 - 2.3 An Outline of the APT IV Processor
 - 2.4 Language Modification in APT IV and NELAPT
3. The Prototype MILDAPT Processor
 - 3.1 Design Concepts for a Modular "APT-like" Processor
 - 3.2 The Input Module
 - 3.3 The (Geometric) Definition Modules
 - 3.4 The Action (Procedure) Modules
 - 3.5 The Execution Modules
 - 3.6 The CLTAPE Editor
 - 3.7 The Overall Structure of the Prototype MILDAPT Processor
 - 3.8 Conclusions Obtained from the Prototype MILDAPT Processor
4. The Dispersed Monitor Concept
 - 4.1 Parallel Processing, Semaphores and Monitors
 - 4.2 The Control of Access to Memory
 - 4.3 Synchronisation
 - 4.4 Access to the Monitor
 - 4.5 The Dispersed Monitor in Practice
5. An Adaptable Data Storage Method
 - 5.1 AOL Data Types and Their Storage
 - 5.2 A List-Based Data Storage Method
 - 5.3 Composite Surface Types
 - 5.4 Subscripted Surfaces
 - 5.5 The Method Applied to a Different Surface Representation
 - 5.6 Data Storage in a Dispersed Monitor Processor
6. The Automatic Generation of User Modules
 - 6.1 The Structure of AOL Statements
 - 6.2 Flexible Syntax Definition
 - 6.3 A List-Directed Syntax Analyser
 - 6.4 Generation of a List-Directed Syntax Analyser
 - 6.5 Generating a Module for a Dispersed Monitor Processor
7. Conclusions and Future Developments
 - 7.1 A Dispersed Processor for CAD/CAM
 - 7.2 Conclusions

References and Bibliography

1. BACKGROUND

1.1 Automatic Programming and Numerical Control

It is almost impossible for a programmer in 1983 to visualise the situation that existed a mere 30 years ago. At that time most programs were still written in machine code or in very primitive forms of assembly language and the idea of a "high-level" language as we know them today was inconceivable to all save a few. It was, in fact, in the early 1950's that the first ideas of "automatic programming" began to be voiced abroad, although these early systems tended to merely allow such "advances" as symbolic addresses, decimal numbers, floating point instructions, and improved input-output commands.

The first obvious moves away from this primitive approach came in 1954 with Laning and Zierler's algebraic system [Laning and Zierler, 1954], which was to be the world's first operational algebraic compiler, and with the start of the Fortran project at IBM [Backus, 1978b] [IBM, 1954], although Rutishauser had already made similar proposals [Rutishauser, 1952] two years earlier, and Zuse's "Plankalkül" had, unbeknown to virtually the entire world, anticipated many of these developments by almost a decade when it was designed in 1945 [Zuse, 1972]. Indeed, one of the most notable features of the very early days of programming was the lack of communication between the various groups of interested parties. Thus, although Zuse's work on his Plankalkül (program calculus) was a rather special, and purely theoretical, exercise whose results were not to be fully published until very many years later, a number of other workers independently developed their own approaches to "automatic programming" using a wide range of techniques. Several of these, such as Goldstine and von Neumann's flow diagrams [Goldstine and von Neumann, 1947] and Curry's proposals for program composition [Curry, 1949], were purely theoretical, although others, such as those of Rutishauser [Rutishauser, 1952] and Böhm [Böhm, 1954], were systems which could have been implemented on the computers of their time. However, the first real operational automatic programming system was probably Glennie's AUTOCODE [Knuth and Pardoe, 1977, pp. 447-451] which took a program written in a mixture of English words and symbols

and translated it into the machine code of a real computer - the Manchester Mark I [Lavington, 1978].

Nevertheless, the real breakthrough came with Laning and Zierler's system, although it was not, apparently, recognised as such at the time of its announcement during the 1954 symposium on automatic programming [Adams and Laning, 1954], since this was a system which, for the first time, enabled a programmer to write programs without needing to know much about the computer itself at all. At the same conference Backus and Herrick concluded their paper about the IBM Speedcoding system with a speculation about the future:

"A programmer might not be considered too unreasonable if he were willing only to produce the formulas for the numerical solution of his problem, and perhaps a plan showing how the data was to be moved from one storage hierarchy to another, and then demanded that the machine produce the results for his problem. No doubt if he were too insistent next week about this sort of thing he would be subject to psychiatric observation. However next year he might be taken more seriously." [Backus and Herrick, 1954]

How prophetic this remark was can be seen from Grace Hopper's introductory remarks at the 1956 symposium, when she stated that

"A description of Laning and Zierler's system of algebraic pseudo-coding for the Whirlwind computer led to the development of Boeing's BACAIC for the 701, FORTRAN for the 704, AT-3 for the Univac, and the Purdue system for the Datatron, and indicated the need for far more effort in the area of algebraic translators." [Hopper, 1956]

This effort was forthcoming, and the next five years were to see intense activity in this area as Fortran [Backus et al., 1957] was followed (and indeed preceded) by languages such as BACAIC [Grems and Porter, 1956], IT [Perlman, Smith and Van Zoeren, 1957], MATH-MATIC [Ash et al., 1957] and finally Algol 58 [Perlman and Samelson, 1958, 1959] [Backus, 1959] and Algol 60 [Backus et al., 1960a, 1960b], while in 1959 the United States Department of Defense agreed to sponsor the project which was to result in the

first COBOL compilers the following year [RCA, 1960] [Bromberg, 1961]. Since that time, although it did not really gain momentum until the mid-1960's, there has been a steady move towards higher level languages using (in general) an increasingly English-like form of expression, so that at the time of writing (1980) few programmers are even capable of understanding the principles of assembly code programming, far less of writing such programs themselves.

This revolution took place for a number of reasons, but possibly the most important was the fact that the spread of electronic computers from scientific research laboratories into the worlds of industry and commerce had brought with it the need for large numbers of programmers, and this in turn led to a requirement for an easier way of writing programs; it is significant that in those early days the expression used was "automatic programming" and not "high-level programming" as we would say today. Another important factor in the spread of these languages, and in particular of languages such as Fortran, Algol and Cobol, was that they enabled the user to transfer his programs from one type of computer to another without requiring him to rewrite the whole lot from scratch. The fact that the nature of these early languages was, with the benefit of hindsight, by no means ideal [Backus, 1978a] is unfortunate, but it is a remarkable tribute to their designers that they, and their basic concepts, have survived so well during a period of such explosive change in computing technology.

However, at the same time as electronic computers were beginning to move out of research laboratories another development was under way which was also to have far-reaching consequences. In the late 1940's the Parsons Corporation of Traverse City, Michigan, proposed to the United States Air Force that punched tape and servomechanism control could be applied to a milling machine in order to produce, automatically, the templates required in the production of helicopter rotor blades. Parsons received a contract from the U.S. Air Force in July 1949, and subcontracted the control work to the Servomechanisms Laboratory at the Massachusetts Institute of Technology (M.I.T.). In February 1951 the U.S. Air Force contract was switched directly to the Servomechanisms Laboratory, which was to continue this work for almost 20 years [Ward, 1968]. By early 1952 not only was the numerical control of machine tools a reality [Pease, 1952] but numerous test parts

were being made and studied for their economic impact [Stocker and Emerson, 1954].

During this period many of the control tapes were prepared by hand, but a library of computer subroutines was also developed for performing many of the necessary calculations and for producing the actual punched paper tape which was used to control the machine tool [Runyon, 1953]. As a result of this work the U.S. Air Force commissioned the production of the first automatic programming language and processing system for higher-level language preparation of machine tool control tapes. This project led to the demonstration of a prototype system in 1955 [Siegel, 1956a, 1956b] only a year after Laning and Ziegler's algebraic system had shown the way to "automatic programming" in the field of general purpose numerical programming. This prototype language used symbolic names to define and machine a two-dimensional part consisting of lines and circles, and led in 1956 to the award of a new contract from the U.S. Air Force for the development of a more complete automatic programming system.

This new contract was passed to the Computer Applications Group led by Doug Ross, whose account of the progress of the project [Ross, 1978] provides a fascinating insight into how a combination of brilliance and chance was to affect the development of the future APT language and its processor. One aspect of the APT language which was apparent to the present author when he first met APT in 1965 was that in some of its ideas and constructions it was considerably in advance of the general purpose languages already mentioned, which were at an almost identical stage of development. The two most obvious examples are the macro facility and the use of nested definitions, and Ross states that the idea for both of these concepts went back as early as 1957, although it was not until 1959 that they became a reality. However the early APT system also contained a number of less obvious techniques, such as those used for the initial translation of the input language by what was later to be generally known as a **lexical scanner**, and the use of indices to a large block of contiguous storage to allow the ARELEM routines (which calculate the cutter path) to move at will from consideration of one set of highly complex surface parameters to another - an idea that was later to be formalised in the concept of **Plex programming** [Ross, 1961] and which was essentially the same idea that was to become **bead programming** in AED [Ross et al., 1970],

records [Wirth and Hoare, 1966] or **abstract data types** [Liskov et al., 1977] in other languages, or even **controlled storage classes** in PL/I [ANSI, 1976]. Indeed, as Ross says:

"One thing that has always disappointed me is that this entire massive effort had so little direct impact on the other developments in programming language and computer science developments. For the most part the (by now) several hundred computer programming people who have worked on the system have been from industry, not academia. Paper writing, except addressed to the "in-group", has been almost non-existent. The continual evolution of more sophisticated features in systems which were in daily productive use meant that almost never was a complete, elegant, package available for concise description and documentation. Finally, the system and problem area itself is so complex and elaborate that it is difficult to isolate a single topic in meaningful fashion for presentation to those not familiar with the complete APT and numerical control culture. For the most part, APT was viewed with respect, but from a distance, by members of the computer science community, even though many of the significant developments of the '70s were first tried out in the '60s in APT." [Ross, 1978]

(In this connection the present author was intrigued and surprised to find, in the early '70s, that upon leaving industry for academia he was confronted with several "new" ideas that he had been familiar with in APT nearly ten years earlier without realising that they were ahead of their time and had yet to be discovered!)

The major features of the APT language were, in fact, specified over a single weekend in May 1957 [Ross, 1970, 1978], out of necessity rather than by choice, with the primary criterion being that the translation should be controlled by its punctuation. As we shall see, this not only simplified the construction of the original APT translator (which was the main reason for this approach) but also made possible such features as nested definitions and the concept of partial translation which is the basis of the dispersed monitor concept described later in this dissertation.

The first successful "part-programs" were processed using the initial APT II program in early 1958, and by mid-1958 field trials of the system were taking place throughout the North American aerospace industry. Thus APT, which was the first and most widely used of the special purpose application-oriented languages (or AOLs as they will henceforth be referred to), was in use at almost the same time as the first general purpose languages and was, in many ways, rather more advanced than most of these. As was to be the case with the general purpose languages, the availability of APT was to lead to a number of similar languages over the succeeding years, although such was the elegance and power of the basic APT language structure that most of these used exactly the same syntax as APT and were different only in the vocabulary used and the scope of the geometry and/or machine tool motion encompassed.

One interesting difference from the outset between AOLs, and APT in particular, and general purpose languages was that AOL programs usually contain a large amount of purely descriptive information, whereas a general purpose language has relatively little descriptive information (e.g. type or array declarations) and consists almost entirely of commands to the computer to carry out some function or action. This difference was to become more marked in later years as systems such as EXAPT [Opitz et al., 1967] progressively moved away from a deterministic approach towards a non-deterministic system which essentially says (in the case of EXAPT) "The blank is like this; make a finished part like this", and leaves the working out of the appropriate tool path, and even of which tools are to be used and in what order, to the computer system. Thus, while, as Backus was to point out later [Backus, 1978a], the von Neumann concept of computers was to lead general purpose programs down a somewhat stereotyped, and far from ideal, path, the same was not true of languages which were to be derived from the basic APT structure, which were considerably less deterministic in their approach right from the beginning. The reason for this can once more be traced back to the original design concepts:

"What kind of person will the average part programmer be? What kind of training has he had? How does he think about machined parts? These questions are important from the point of view of language design, since the final structure and format of the language must fit naturally into the framework of the part

programmer's past experience and knowledge. This does not mean, however, that the language should be tailored to fit exactly with the thought patterns and ways of doing things which the potential part programmer now employs. To a great extent these patterns of behaviour have been influenced, and, in fact, primarily determined, by the old methods for producing parts. Therefore what is actually desired is not a language form which will mirror exactly the existing techniques and thought processes but rather one which takes advantage of the improved capabilities of numerical control and automatic programming to increase the part programmer's capabilities and at the same time one which seems to be a natural extension of the already acquired skills. A properly designed language can serve to advance the over-all capabilities of the production process without appearing to be a radical and revolutionary departure calling for large expenditure of money and effort to put into practice." [Ross, 1960]

This approach to language design led Ross to the conclusion that

"... although the simplest mathematical expression for a circle is to select the proper coordinate system and then write $X^2+Y^2=R^2$, that same circle may, in fact, be more conveniently described by saying that it is tangent to two lines, with a specified radius, or that it passes through three points, etc. Thus, with the realisation that a distinction can be made between **specifying** the equation for a subpart and **describing** that subpart, a whole new area for language design is opened." [Ross, 1960]

It is the present author's belief that, despite the many plaudits heaped upon him during the last 20 years, insufficient credit has been given to Ross for these basic principles of language design, which were first expressed publicly in a lecture he presented on 29th March, 1957, entitled "Design of Special Language for Machine-Tool Programming" [Ross, 1957a] as part of a course organised by MIT on the programming of numerically controlled machine tools. This lecture was presented two months before the APT language was designed [Ross, 1957b], and thus gives a very clear indication of the underlying principles that helped to formulate that design. One of these principles concerned the fact that "a language

will usually be in a constant state of growth and revision"; the number of different languages with widely differing facilities which are based upon the original APT language is a testament both to the accuracy of that prediction and to the soundness of the underlying APT language structure.

1.2 The Development of NC Languages

The impetus for numerical control had come from the U.S. Air Force and the American Aerospace industry, and the primary use of the new machine tools was thus, initially, for the machining of complex parts for aeroplanes and, from the mid-1960s, rockets, satellites, and other equipment that came with the dawning of the "space age". Thus, once the initial concept had been developed at MIT, there was a ready acceptance of this new technology and a desire to push its performance to the ultimate limits. This, inevitably, implied some form of computer assistance with the production of the control tapes as the calculation of the cutter paths required to machine complex three-dimensional parts was virtually impossible to attempt by hand. The development of APT during 1957/1958 was a cooperative venture between the Aircraft Industries Association (AIA) - a trade association of American companies involved in aircraft manufacture or subcontracting, and thus containing most of the major manufacturing organisations in the United States - and the Servomechanisms Laboratory at MIT, and from the very beginning there was, therefore, an enthusiastic group of users who provided plenty of feedback to those who were developing the system.

This meant that the initial APT II system, which moved tools along a space curve in order to cut the part, was soon succeeded by APT III, which used the intersection of three-dimensional surfaces to control the tool path [Bates, 1962]. Both APT II and APT III were presented to the user as an "APT computer" which accepted an input language (the APT language) and which produced movement of the cutter of a machine tool as a result of obeying programs written in that language. Thus all problems concerned with input and output, calculation of values, etc. were eliminated; the part-program simply instructed the "APT computer" what it should make, and it made it! Or, to be more accurate, it produced (normally) a roll of punched paper tape which would cause the particular machine tool to make

it. The "APT computer" is, of course, simulated on a general purpose digital computer, and consists of a compiler and a large library of subroutines which are used by the compiled part-program, together with a post-processor program which tailors the output to the particular machine tool which is to be used for the actual machining of the part. The original APT II system was, of course, written in assembly code as no high-level languages existed at that time, and was produced for the IBM 704 computer (which was the model used by most of the AIA participants): however APT III was mainly written in Fortran for the IBM 7090 (which had by then largely replaced the 704 amongst the participating organisations), although certain key areas, notably all input/output operations, were written in assembly code for reasons of efficiency.

However, the increasingly rapid rate of technological development meant that during the short time since the initial public announcement of APT at a press conference on 25th February 1959 [MIT, 1959] [Am.Mach., 1959] the requirements of the general manufacturing industries had changed, and in 1961 the APT Project was reorganised as a multi-sponsored project open to all American industry. The new project, which was to be known as the APT Long Range Program (or ALRP) was not based at MIT but at the Armour Research Foundation of the Illinois Institute of Technology (subsequently to be renamed IIT Research Institute, or IITRI), which was to continue development of the system under the direction of the sponsoring organisations and to provide assistance with training, fault finding, etc. for the sponsors [Am.Mach., 1961].

As a result of this wider availability of APT within North America, and its subsequent availability to first European and then Japanese organisations, it soon became apparent that there were very substantial difficulties involved in transferring the APT III system to another type of computer. By the mid-1960s the concepts of machine independent programming were well established (although the problems were not yet fully appreciated), and the availability of Fortran IV on a wide range of computers meant that programs could be relatively easily transferred and also that there was no longer any need to write critical parts in a low-level language. IITRI therefore started to develop a "New APT" system using quite different principles in the translation, and designed to meet two major criteria - more capability and computer independence [IITRI, 1964].

A prototype system was available in 1965 on an IBM 7090 computer at IITRI [IITRI, 1965], and the first successful implementation on another computer was made in Britain by English Electric Computers on their KDF9 computer later the same year [Ellis, 1966]. During this implementation a number of alterations were made to the way in which the translator worked, both to provide a more efficient implementation and to remove some of the (inadvertently) IBM-oriented machine-dependent features [EELM, 1966a]. (These led to the secondment of the present author to IITRI to assist in the further development of the system, and, in particular, in the incorporation of changes based on the English Electric suggestions into the standard system [Ellis and Coldham, 1966] [Ellis, 1967]).

The first official (experimental) version of the New System followed in 1968, although at that stage its capability was far short of APT III which was, of course, by now widely used and still under continuous development. APT IV, as it was to be known, was finally issued as a full release in 1971 [IITRI, 1971], and has been the basis of most subsequent development of the language and computer system. Finally, in 1972 the organisation of the APT Project changed yet again as the members of the APT Long Range Program decided, for a variety of reasons both technical and legal, to incorporate themselves into a new, independent, not-for-profit organisation, to be known as called Computer Aided Manufacturing - International Inc. (CAM-I) [CAM-I, 1972], under whose umbrella all subsequent APT development has taken place.

It should not be thought, however, that APT was alone throughout this period. Just as the first general purpose programming languages had led to a large number of others, so did the appearance of APT in 1959 herald the start of a growth industry in languages for Numerical Control. By 1969 there were "at least 33 languages" [Mangold, 1970], although, since the list given by Mangold omits at least six languages which were known personally by the present author at that time, the true number was probably very much higher. The first of these languages came directly from the APT project and were AUTOPROMPT [Matsa, 1961], which was an early attempt at regional programming, and ADAPT [IBM, 1963], which was a simpler version of APT designed for two-dimensional contouring applications (2 $\frac{1}{2}$ -axis). Another early language was SPLIT [Sundstrand, 1964], which was developed by the Sundstrand Corporation and, although the writer has been unable to

verify any connection, appears to draw several of its design concepts from Siegel's original language, although the SPLIT language is not so elegant a language as was Siegel's. SPLIT is still used by Sundstrand, having been developed to full 5-axis capability, but its greatest claim to fame is that it was the direct precursor of ACTION and COMPACT II, the latter [MDSI, 1973] being claimed to be the most widely used NC programming language in the world in 1980.

It is not the intention of the writer to survey the many NC systems which were to spring up (and in many cases to die almost as quickly a few years later), but one or two others are worthy of mention. The first of these is AUTOSPOT [IBM, 1962], which was based on Westinghouse's CAMP system [Knarr, 1962] and was designed to provide a terse, rather than elegant, system for multi-axis machining centres aimed at practising process planning engineers [Nussey, 1970]. This was subsequently to become part of IBM's System/360 Numerical Control system, together with ADAPT and APT. However some of the more interesting language developments were to take place in Europe.

Although the preceding discussion has concentrated on the American scene, and upon APT in particular, it should not be thought that Europe was not also active in this new technological field. However, due to the absence of any collaboration between the two continents there were major divergences in both hardware and software. The most obvious of these was in the medium used to control the machine tool. Most European systems used magnetic tape to provide continuous control of two or more axes using time-dependent phase-analogue coded information. This meant that the controller attached to the machine tool could be quite simple, but that the input for it could only be prepared using a computer. After a short experimental period American manufacturers, on the other hand, rejected this approach and decided upon punched paper tape using controllers which contained considerably more (and expensive) hard-wired logic. There were a number of reasons for this decision, foremost amongst which were the feeling that the state-of-the-art ruled out the more desirable magnetic tape, and the fact that paper tape eliminated any dependence upon computers - a somewhat short-sighted view, but one which is very understandable, especially as the major growth area was expected to be in point-to-point machines and simple machining centres with circular interpolation and canned cycles. This

divergence in hardware led to a parallel divergence in software, and to programming systems which operated in a very different way from their American counterparts, all of which, to one degree or another, owed their basic structure and design approach to the early work carried out by Doug Ross and his team at MIT, which has been discussed above.

The best known of these systems is PROFILEDATA [Ferranti, 1964], which was developed by Ferranti Ltd. as a means of programming parts which were to be produced on machine tools controlled by the Ferranti magnetic tape control system. The output from PROFILEDATA was, in fact, a punched paper tape which was then fed into a special piece of equipment known as a "curve generator", which produced a suitable magnetic tape. Ferranti offered a very successful bureau service for their customers throughout Europe, and the system was also implemented on IBM System/360 and ICL 1900 computers. Other languages were also written which produced paper tape suitable for input to a "curve generator", such as Rolls-Royce's COCOMAT [Rolls-Royce, 1962], Hawker Siddeley's CLAM [Neave, 1964], and I.C.T.'s surface fitting program PMT2 [ICSL, 1966a]. (A more readily available description of PROFILEDATA and PMT2 can be found in [Wood, 1970]).

This proliferation of languages caused the British Ministry of Technology to set up a committee, under the auspices of the National Engineering Laboratory, to investigate the whole situation and to recommend what action should be taken, both short-term and long-term. The First Report [NEL, 1965] recommended that suitable computer programs should be developed at Government expense in order to introduce a degree of standardisation, which in turn would help in the development of the use of NC throughout British industry. In particular, it recommended that APT-compatible systems should be developed for 2½-axis and point-to-point work. The outcome of this report was the 2,CL system [NEL, 1967, 1969] which was subsequently to be renamed NELAPT. The language was essentially designed for 2½-axis work (i.e. continuous control of two axes simultaneously, with positioning control only of the third), and consisted of an "extended subset" of the APT language. The extensions were in two main areas - **patterns** and **area clearance**. The first of these merely consisted of substantially increasing the number of ways in which patterns of points could be defined (so that a sequence of operations could be carried out at each point), and introduced no new ideas. The second, **area clearance**,

feature was, however, a revival (in a different form) of Ross' regional programming ideas, which had originally been intended for APT III but had only survived in AUTOPROMPT. The concept of a closed contour was introduced, together with the facility to automatically mill a pocket defined by that contour (with optional "islands" defined by other closed contours). While limited to two dimensional contours, this feature, nevertheless, proved to be extremely useful and to provide a substantial improvement in part-programmer productivity.

Britain, of course, is only a small part of Europe, although at that time it was probably the most advanced, technologically. However, parallel developments were taking place throughout the continent, leading to further proliferation and the development of systems such as SAP-2 and APROKS [Pruuden, 1970] in the Soviet Union, PHILCON [Vliestra and Wielenga, 1970] in the Netherlands, PAGET and SURF [Olivetti, 1968] in Italy, SYMAP [Kochan, 1970] in the German Democratic Republic, and many more. However, possibly the most significant European developments during this period took place in the German Federal Republic.

In the early 1960s the machine tool manufacturers Pittler A.G., realising that some form of computer programming was essential if their first NC lathe, the Pinumat, was to achieve a sufficient throughput of small jobs, embarked on a collaborative venture with IBM to produce a suitable system. This system was known as AUTOPIT [Pittler, 1967, 1968] and was different from almost all other systems in that it included a large amount of **workshop technology** - that is the ability to automatically calculate cutter paths for roughing and finishing cuts, and to select the appropriate tools, feedrates and spindle-speeds. This program first appeared in 1964 and was so successful that it induced a number of other German lathe manufacturers to collaborate with IBM in producing a more flexible and more universal system for lathes. However, although AUTOPOL [IBM, 1968a], as it was known, was technically a very successful system, by the time it was available other developments in Germany had already overtaken it and it was never very widely used.

In 1964 several German Universities (notably those in Aachen, Berlin and Stuttgart) embarked on a project to develop a programming system which was to automatically deal with all the routine technological features such

as the selection of tools, the specification of cycles (e.g. centre drill, then drill, then tap, then chamfer), and the calculation of the optimum feedrates and spindle-speeds. The first system, EXAPT 1 [EXAPT-Verein, 1967a] [Reckziegel, 1970a], used an APT-like language which was extended to deal with the technological features and was designed for point-to-point work and simple straight-line milling. It was completed in 1966 [Engelskirchen, 1966] [Herzog, 1966], and went into use at Siemens A.G. the same year.

The logical next step was to extend these ideas to lathes, and, in collaboration with Pittler A.G., the EXAPT-Verein (as the development project had now become) turned their attention to this area in order to produce the EXAPT 2 system for turning operations, in which it was only necessary to define the shape of the blank and the desired shape of the finished part [EXAPT-Verein, 1967b] [Reckziegel, 1970b]. The third and final phase of the initial development effort was EXAPT 3 [Stute, Opitz and Spur, 1971], which was to carry the same concept into full 2½-axis milling, although this turned out to be a much more difficult task than had earlier been anticipated, and it has never really achieved the popularity of EXAPT 1 or EXAPT 2.

One effect of the diversity of effort in Europe, and the lack (in general) of significant government support on the American pattern through research or development contracts, was that the emphasis was very different on the two continents. Thus, whereas in the U.S.A. numerical control had started in the aerospace industry with the requirement for three-dimensional contouring, in Europe development had started at the other end of the spectrum. The result of this was that, with a few exceptions such as the Ferranti magnetic tape system mentioned earlier, most European machine tool and control system manufacturers started their involvement with NC with relatively simple point-to-point and straight-line milling systems. One consequence of this was that there was a rapid spread of simple fixed-format NC programming systems such as KIPPS [EELM, 1966b], AID [ICSL, 1966b] and ROMANCE [IBM, 1968b]. These used a totally different approach from APT and the APT-like systems and did not require a part-program to be written to define the part and the tool movement; instead they required the part-programmer to simply fill in values in the appropriate columns of a proforma coding sheet. These were, therefore,

what has already been defined as "package systems", as opposed to "language processors" such as APT, EXAPT, etc., and had the advantage that they were, for many users, a more readily acceptable form of computer input, as well as requiring considerably smaller computers. During the late 1960s there was much argument about the relative merits of the small fixed-format NC program package (a category which, in a sense, also included such contouring packages as PROFILEDATA [Ferranti, 1964], MILMAP [ICT, 1967] and SURF [Olivetti, 1968]) versus the larger NC programming languages such as EXAPT 1 and 2, CL (and APT, although this was not a serious contender in Europe for this type of work). However, the inherently greater flexibility of the AOL systems eventually won the day, and at the present time (1980) fixed-format packages are only rarely used and then, usually, only for certain highly specialised types of work.

Another reason for the demise of the package systems concerned the increasing number of numerically controlled machine tools and control systems. From the outset APT had been designed as two, essentially distinct, parts - the processor and the post-processor. The processor produces a file which contains all the necessary tool movements to machine the part to the specified tolerance on an idealised machine tool where the part remains stationary while the cutter moves around it. The post-processor first converts this idealised situation to movements of the various slides, etc. on the actual machine tool (with due allowance for its physical characteristics and dynamics), and then produces a control tape which contains the properly coded instructions to achieve these movements, together with any other auxiliary information; a facility is provided in the APT language for information (e.g. concerning the use of built-in machining cycles) to be passed by the processor directly to the post-processor where this is necessary. The interface between these two phases is a magnetic tape (or, nowadays, usually a disc file) known as the CLTAPE (or CLFILE) whose format is rigidly defined. This, therefore, means that in order to manufacture a part on a different machine tool it is only necessary to post-process the CLTAPE using the post-processor for the new machine tool - the original part-program does not need to be altered at all. The APT III CLTAPE format is clearly and unambiguously defined [IITRI, 1962], and is the basis for the CLTAPE formats used by most other APT-like languages. Thus NELAPT (2,CL) uses the same format with some minor extensions [Sim, 1968], while EXAPT uses a similar one, but with

significant extensions due to the need to pass some technological information to the post-processor [EXAPT-Verein, 1967c]. APT IV uses a rather different format [Rodriguez, 1965] [IITRI, 1970] in order to allow the flexibility of passing names to the post-processor instead of just code-numbers, but it can also produce an APT III format CLTAPE if required.

In general, however, non-APT-like languages and packages do not have a clearly defined interface such as the CLTAPE. Some of the systems mentioned above (e.g. KIPPS, ROMANCE, MILMAP) could cater for more than one type of machine tool or controller, but only by modifying the main program package, while others (e.g. PROFILEDATA, SURF, AUTOPIT) were firmly wedded to a particular machine tool and/or controller. Thus as organisations bought additional NC equipment the only way in which they could use the same programming system for a number of different makes of machine tool was to use APT or one of its many derivatives.

1.3 The Growth of APT-like Languages

As has already been discussed, the original APT II system was coded for the IBM 704 computer, while APT III was written, largely in Fortran II, for the IBM 7090. However, APT was always a large computer system since it had, from the outset, aimed at total generality. Once the initial APT III system, which provided full simultaneous contouring control of three axes, was available it soon became apparent that there were a large number of potential applications for which a much smaller program, with reduced capability, would be perfectly adequate. The U.S. Air Force therefore invited tenders for the development of a simpler system, using a subset of the APT language, which would run on smaller, and thus more readily available, computers and would cater for machining operations in which contouring was carried out with the tool tip on a plane (i.e. 2½-axis work). The contract was won by IBM San Jose, and resulted in the release of ADAPT [IBM, 1963] in early 1963. IBM, under the terms of the contract, released the Fortran II coding and full documentation to other computer manufacturers, and for many years since then the language has been widely used on small-to-medium-sized computers for NC programming.

Once APT and ADAPT were both available and broadly compatible [Kelley, 1964] it was clearly desirable that some consideration should be given to the desirability and feasibility of formally standardising the language(s). Thus, in the U.S.A. National Activity Report to the International Standards Organisation Technical Committee 97, Subcommittee 5 (Computers and Information Processing) for May of that year [Bromberg, 1963] we find a summary of the APT language, while in a considerably more detailed description of the APT language published shortly after [Brown et al., 1963] we learn that the American Standards Association X3.4 Committee on Common Programming Languages had also begun such a consideration. This was subsequently to lead to the formation of the APT Standards Working Group X3.4.7, but for a variety of reasons the standard definition of the APT language was not finally issued (in one of the most incomprehensible and unreadable documents it has been the writer's misfortune to encounter) until over ten years had elapsed since the original decision to attempt such a standardisation [ANSI, 1974, 1977]. The main reason for this delay, however, concerns the role of the International Standards Organisation and the development of other, non-American, APT-like languages [Mangold, 1970, 1973].

While the development of ADAPT as a 2½-axis subset of APT was a purely American development, by this time (1963) APT had become well-known (although not yet used) in pioneering circles throughout Europe. Much the same reasoning that had led to the development of ADAPT, together with the much greater emphasis on point-to-point and straight-line milling NC machine tools and a considerable amount of (in the writer's view) misguided nationalism, led to the development of three major European "APT-like" NC languages during the period 1964-1966. These were EXAPT and 2C,L (later to be called NELAPT), which were developed in Germany and Britain respectively and have been discussed in some detail above, and IFAPT [CII, 1966], which was developed in France as a series of simpler, smaller, subsets of APT. These three "national languages" were to vie with APT in the discussions in I.S.O. for a number of years as to which should have the major influence not only on the final I.S.O. standard NC language, but also on the method of definition of that language. In the event no such standard has yet (in 1980) been agreed, although, as mentioned above, the American National Standards Institute has produced an APT standard in an almost totally unintelligible format.

Meanwhile the "de facto" standard was APT, and a succession of other "APT-like" languages following essentially the same language structure and, in general, producing the same (or very similar) CLTAPEs were developed in various parts of the world. Some examples of these are CELAPT [Renault and Taboy, 1974], CKDAPT [Macurek and Vencovsky, 1973], HAPT-3D [Hyodo, 1973], LINK [Galeotti, 1973], PRAUTO [Sohlenius and Iacobaeus, 1973], PROMO [Gendre, 1974a] and SURFAPT [Gendre, 1974b]. These "second generation" APT-like languages were mainly developed for pragmatic reasons to cover an area of work which was of interest to their developer, and which either was not covered by APT or one of the "first generation" APT-like languages (ADAPT, EXAPT, IFAPT, and NELAPT) or was available in one of these but not in an acceptable form for reasons of size, complexity, cost, etc.

As was mentioned earlier, the present author was deeply involved in the development of APT IV in the mid-1960s [Ellis, 1966, 1967]; however, for reasons outside the scope of this paper, he subsequently spent several years working in a completely different area. On returning to take an active interest in this field once more in the early 1970s it was very apparent that the rapid development of APT-like languages was having a number of undesirable effects. Most notable amongst these were the fact that the consequent dilution of effort was resulting in a substantial amount of "re-inventing of wheels", and that a number of undesirable features were in danger of being perpetuated, despite the changes in both hardware and software which had led to the advent of powerful minicomputers and the development of new theories and techniques of programming.

The most obvious of these undesirable features was their size, for while APT has always, throughout its long development, been suitable only for the largest computers of the day, most of its derivatives, although by their nature less complex, still required medium-to-large computers on which to run at all efficiently. A second common feature, which possibly has some bearing on the first, is that they were, almost without exception, written in Fortran. Computer scientists often have hard words to say about Fortran and give the impression that it should never be used at all! This, in the writer's opinion, is a very considerable oversimplification and is extremely unfair to what is, without doubt, by far the most widely used scientific programming language; nevertheless it is not the ideal language in which to write a sophisticated language processor. When APT III was

being written (and indeed APT IV and the first generation derivatives ADAPT, EXAPT, IFAPT and NELAPT) the obvious high-level language to use was Fortran, both because of its almost universal availability and because it was, and still is, the language most widely used in industrial and scientific work. For the numerical part of the processing it is a good language, although the lack of flexible program constructs does tend to lead to programs which, by today's standards, are badly structured and ill-designed; however, for character handling (which is, after all, a major part of the translation stage) it is an appalling language, and requires either assembly code routines or the use of computer-dependent extensions together with a generally inflexible and inefficient methodology. (The recent extensions to the language which resulted in the new standard Fortran known as Fortran 77 [ANSI, 1978] [Ellis, 1980, 1982] largely eliminate these particular criticisms, but this language was not available until 1979/80).

However, the most serious failing of APT (and of most of its derivatives) was its monolithic structure, which meant that it could only be altered or extended by someone with an intimate knowledge of its internal structure, and then only with considerable difficulty. It was this factor which had led, in the mid-1960s, to the production of the first generation of APT-like derivative systems, rather than simply producing subsets of the APT processor for specific requirements, extended as appropriate. Furthermore, even these systems tended to exhibit the same tendency, thus leading to the still more specialised second generation derivatives of the 1970's. Clearly, this was by no means the only reason for the creation of so many similar systems, but it was certainly a major factor, as, despite their differences, some 80% of the code in a processor such as NELAPT, ADAPT or IFAPT (and only slightly less in EXAPT) is carrying out translation, analysis, calculation and output activities which have to be carried out in essentially the same way in APT and in any other APT-like processor. Not only does this mean that an enormous amount of effort was spent, and continued to be spent, on duplicating work that had already been done, but also that the lessons learned on one processor were not put to full use elsewhere.

The remainder of this document describes the work carried out by the author, initially to investigate approaches which might be taken to resolve

this problem, and subsequently to develop a methodology for designing A.O.L. processors for a general language which could easily be adapted to process a different range of input language statements or to operate on a different size of computer.

2. APPLICATION-ORIENTED LANGUAGES AND THEIR PROCESSING

2.1 General Principles

An application-oriented language (AOL) can be defined as a programming language which has been designed for use in solving a specific type of problem or class of problems, and which is not suitable, as is a general purpose language, for solving a wide range of problems in widely differing areas. These types of languages are often called "special purpose" or "problem-oriented" languages, but it is the writer's belief that the term "application-oriented" more accurately reflects the true nature of this very important class of programming languages. It is also important to emphasise the word **language**, for problems in a particular application area may almost always be solved by using one of two quite different approaches.

The first approach is to write a computer program (or set of programs) which is designed to solve the particular problem (or class of problems), and which only requires that the appropriate data is given to it in an easily digestible form. This data may be fixed format (requiring the use of special coding sheets), or free format, or interactive (which is, in a sense, only a variation on the other two); however, it will consist, essentially, of a number of discrete items provided in a given order. The alternative approach is to write a computer program (or set of programs) which is designed to read a series of **statements** which describe the problem and/or the solution which is required in some **language** which has been specially designed for this purpose. The data in this case, therefore, consists of a program which the computer program(s) must analyse and obey in order to solve the required problem. We shall call the first approach a **package system**, while the second approach is, as we have already seen, an **application-oriented language system**.

In some application areas (e.g. payroll or accounting) the package approach is normally preferable, while in others (e.g. geometrical design or numerical control) an AOL approach is normally used. However, no hard and fast rules can be laid down, as the best approach depends upon the particular combination of factors which relate to the problem(s) to be

solved. As a general rule the package approach is most suitable for problems in which either a small amount of semi-standard data leads to a complete analysis for that data (e.g. payroll calculations based on hours worked) or one of a number of standard sets of calculation is called into action to process a set of data (e.g. statistical analysis of some set of data). An AOL, however, is more appropriate where a more descriptive form of data is required (e.g. geometric design), or where the number of alternative types of calculation is too great to be accommodated by a standard format (e.g. movement of a tool across an arbitrary selection of three-dimensional surfaces).

In many cases, however, it is perfectly feasible to use either approach and examples of both can be found in many different application areas. Thus two popular computer systems for discrete event simulation are GPSS (General-Purpose Simulation System) [Gordon, 1961] [IBM, 1970], which is a package system, and SIMSCRIPT [Markowitz et al., 1963] [Kiviat et al., 1968], which is an AOL. Similarly, the statistical analysis of large amounts of data has led both to very sophisticated package systems such as SPSS (Statistical Package for the Social Sciences) [Nie et al., 1975] and to AOLs such as GENSTAT [Nelder et al., 1973]. Even in the field of three-dimensional contouring numerical control, in which the AOL now reigns supreme, earlier systems such as Profiledata [Ferranti, 1964] and PMT2 [ICSL, 1966a] were packages requiring their data to be supplied in a fixed-format manner.

As the use of computers in the solution of large and complex application areas has grown however, a pattern has begun to emerge and almost all recent systems which aim to encompass a wide range of related problems in a flexible and user-friendly manner have been AOLs. A good example of this is the GENESYS system [Genesys, 1974] for engineering design which uses extensions to the Fortran language ("Gentran") to provide a wide range of analyses which were traditionally performed by the use of simpler, stand-alone, package systems. One of the advantages of an AOL for this type of application is that it is, or should be, relatively easy to extend or alter the scope of the system by the addition of new language vocabulary while retaining the same basic syntax; a package, on the other hand, will usually require a completely new set of input data formats. Thus, in

addition to flexibility and descriptive capability, we may add extensibility as a major strength of an AOL.

The research described in the remainder of this dissertation was concerned with investigating how far this flexibility and extensibility was affected by the methods used by the AOL processor, and with the development of a radically new approach to the design of such a processor which avoids many of the problems and restrictions caused by conventional approaches. First, however, it is necessary to establish a few basic principles which will apply to the structure of all AOL processors.

The processing of an AOL program will always consist of at least four distinct phases, although the boundaries between these phases will not necessarily be clearly defined. The first phase is known as **lexical analysis** (or scanning) and consists of the conversion of the program statements in their external form (i.e. as alphanumeric characters on cards, visual display unit (VDU) or other input medium) into some appropriate internal form. The second phase is the **syntactic analysis** of the program statements - that is the interpretation and checking of the grammar of the individual statements to ensure that they are syntactically correct and to enter such values, names, etc. as are required in appropriate tables. This phase will also be responsible for detecting the majority of the errors (if any) in the program and for taking appropriate remedial action. The third phase is **semantic analysis**, during which the meaning of the program statements is established (their grammatical veracity having, of course, already been established by the previous phase). The result of this phase of processing will normally be some form of intermediate language (I.L.) which defines exactly what action is required in order to achieve the objective specified in the source language. These three phases together are collectively referred to as the **analysis** phase of the compilation process, and are followed by a **synthesis** phase which will produce the final object program [Gries, 1971]. If the AOL processor is a compiler which produces a loadable binary program then the synthesis phase will consist of **code-generation** followed, possibly, by the **loading** of the object program produced, together with procedures extracted from a standard library; however if, as is frequently the case, the intermediate language is to be interpreted then the synthesis phase merely consists of producing

the I.L. in a form suitable for interpretation, and will essentially be incorporated within the last part of the semantic analysis.

The final phase of processing in either case is **execution**, during which the compiled program is executed, or the I.L. interpreted, in order to carry out the specified actions and to produce the required results. In some cases, such as numerical control or drafting, where the AOL program is ultimately to drive a piece of equipment there may be a further **post-processing** phase which is designed to tailor the output from the main processing to some specific type of hardware.

The relationship between these phases can vary enormously, depending upon the type and purpose of the particular language processor. Thus a system which is designed to run in an off-line (or **batch**) mode on a small computer might well consist of several distinct programs (or **passes**) corresponding to lexical analysis, syntactic analysis, semantic analysis, etc., with the latter two frequently themselves being further subdivided. In this case the complete source program will be processed by the lexical analyser to produce some intermediate file of information, which will in turn be processed by the syntax analyser, and so on. On the other hand an interactive system might well process a single source statement at least as far as semantic analysis, and possibly right through to, and including, execution where appropriate, before returning to start processing the next statement. In this case there will be no clearly defined interface between the various phases other than a call to the appropriate procedure and a defined data structure within the memory of the computer. Nevertheless, regardless of the actual structure of the language processor, it will always contain lexical, syntactic and semantic analysis phases, together with an execution phase.

In many respects this situation is no different from that which exists for general purpose languages, and indeed, formally, there could be said to be no difference between such languages and AOLs. The differences only become apparent when we examine the languages with respect to their mode of use rather than their general construction. The first obvious difference is that, in general, the structure of the program, as opposed to the structure (or syntax) of individual statements, is much simpler in an AOL than in most general purpose languages. This is most apparent when we look

at the decision-making capability within an AOL, which is frequently (c.f. APT) limited to something as primitive as a Fortran arithmetic IF. However this is often compensated for by a considerably more complex structure for individual statements in an AOL. Thus, if we consider the APT statement

GOLFT/DS,PAST,2,INTOF,CS

we have a quite complicated structure built into a single statement which requires a tool to be moved to the left from its present position along the intersection of the current part surface and a second surface (DS) until it is just past (i.e. tangent to on the far side) the surface CS for the second time.

An even more sophisticated example is

GORGT/DS,PAST,CS1,L1,TO,CS2,L2

which causes the tool to move to the right along the intersection of the current part surface and the drive surface DS until **either** it is just past the surface CS1 **or** it is just touching the surface CS2. In the first case the program execution will continue from the statement labelled L1, while in the second case it will continue from the one labelled L2.

Thus we see that the AOL (APT in the above examples) can embody in a single statement the logic which would require a considerable number of separate statements in a general purpose language. The above APT examples also indicate another of the important aspects of a typical AOL, namely that it is highly **procedure-oriented**. Thus, if we adopted a Pascal-like syntax, in which reserved words appear in upper-case while procedure and variable names are in lower-case, we could write these two examples in the form:

```
golft(ds,PAST,2,INTOF,cs);  
gorgt(ds,PAST,cs1,l1,TO,cs2,l2)
```

Notice, however, that these are very sophisticated procedures since they may take a great variety of different forms of argument list. For example, the following program extract (using the same syntactic form) includes eight calls to the procedure golft, each having a different set of arguments:

```
golft(s1,feedrate1);  
golft(s2);
```

```
golft(s3,PAST,s4);  
golft(s4,ON,s5,feedrate2);  
golft(s5,PAST,s6,11,TO,s7,12);  
11) golft(s6,TO,s7,12,TO,s8,13,feedrate3);  
12) golft(s7,PAST,2,INTOF,s9);  
    jumpto(14);  
13) golft(s8,PAST,2,INTOF,s9,feedrate4);  
14) gofwd(s9,TANTO,s10)
```

A general purpose language will normally have facilities for arithmetic, program control (loops, decisions, etc.), input/output and the use of procedures. An AOL will also have facilities for all of these, but in very different proportions. In particular, input/output statements will frequently be (apparently) totally absent since the program itself will contain all the necessary data and the output will be dealt with as a by-product of processing by some of the standard procedures. Because an AOL is designed for use in a particular application area a very substantial part of the arithmetic and control (c.f. the above APT examples) will be dealt with by procedures, and calls to these form by far the greater part of a typical AOL program.

The language itself will, therefore, normally consist of a small number of control and arithmetic statements together with a large number of procedures, which may or may not deliver any values (or results). We shall refer to the procedure calls which deliver values as **definition statements** and to those which do not as **action statements**, thus reflecting their normal mode of use.

An action statement will take one of two forms, depending upon whether it requires any arguments:

```
action  
or  
action(argument1,argument2,.....)
```

A definition statement, on the other hand, will always take the form:

```
name:= type(argument1,argument2,.....)
```

It is perhaps worth noting at this point that expressing a definition statement in this way emphasises yet another "first" for APT. From its inception in 1959, APT was a **strongly-typed** language in which all variables

are declared to be of a particular type at the moment of their definition (apart from scalar variables, which are all real and are relatively little used). The full implications of such strong typing were not appreciated for some time in the world of general-purpose programming, and did not really come to fruition until some ten years later with Algol 68 [van Wijngaarden et al., 1969] and Pascal [Wirth, 1971].)

The importance of recognising the essentially procedural nature of most AOLs can be seen by examining the details of the APT language. Here we find that, for example, in 1967 the APT Dictionary listed 118 different geometric definition formats (or 294 if all major variations are included) [IITRI, 1967], while the NELAPT "extended subset" two years later had 69 geometric definitions (279 including all major variations) [NEL, 1969]. All of these definitions took the same syntactic form, i.e.

NAME = TYPE/ARGUMENT1, ARGUMENT2,

which is exactly the same as the general form of a definition statement given above.

The fact that the language is highly procedural has obvious implications for its translation and subsequent execution; however the fact that, as has already been noted, the same "procedure" may have a number of different forms means that the translation is by no means straightforward. For example, in NELAPT a circle may be defined in 20 quite different ways, such as

- a) the coordinates of its centre, and its radius
 - b) three points through which it passes
 - c) two lines to which it is tangent, and its radius
 - d) a point through which it passes, and its centre
 - e) a point through which it passes, a circle to which it is tangent, and its radius
- etc.

A very important aspect of AOLs is that it is precisely in this area of definitions (and to a lesser extent in the related area of actions) that changes are likely to be made.

The reason for this is clear when we consider, once more, the essential difference between a general purpose language and an AOL. A general purpose language provides the programmer with a set of **building blocks** such as control statements, input/output facilities and procedure calls. The language facilities can be tailored to a particular application by means of a **library** of procedures (as is done, for example, by the NAG library [NAG,1981] for numerical analysis or by the GINO library [CAD Centre,1979] for graphics), but the language itself remains unaltered and non-specific. Any extensions required for a particular application must be added using the same basic building blocks or by the addition of new library procedures, each of which must have a well-defined and fixed interface (i.e. argument list).

In the case of an AOL, however, the library of procedures and the underlying data-base are not normally directly accessible to the programmer. Thus, although a definition statement such as

```
p1:= point(l1,l2)
```

will almost certainly call a library procedure to find the point at the intersection of the two lines l1 and l2, this process is not apparent to the programmer. This means that if we wish to add a new definition to the language it will be necessary to change the language definition as well as providing one or more additional procedures. This additional complication is the price that has to be paid for using a **higher-level language** in which the programmer can express his problem in his terms (e.g. lines and points in the above example) rather than in the computer's more general and abstract terms.

The corollary to the ease of use engendered by a language which enables a programmer to express his problem in his terms is that his problems change, and therefore so must the language. For example, during the first decade of APT development one of the most active and hard-working committees of the APT Long Range Program was concerned with "New Language Definition". Furthermore, as was discussed in section 1.2, the development of APT led to a number of "APT-like" systems such as NELAPT, EXAPT, ADAPT, IFAPT, CKDAPT, etc., due to requirements for some variant, subset, or extension of the basic APT language. It is both interesting and regrettable to realise that all of these later systems involved starting again and

writing a totally new set of programs for lexical, syntactic, and semantic analyses, and a new set of library procedures for use during execution. It is instructive to look at two of these systems in some detail to see why this should be so.

2.2 An Outline of the NELAPT Processor

The first NELAPT system (or 2C,L as it was then called) was written during the period 1965-1967 [NEL, 1965, 1967]. It has subsequently been extended and revised in a number of ways, but the underlying system concepts have not changed.

The processor consists of four parts which are known as Input, Decode, Geometry and Motion. The Input and Decode sections roughly correspond to the lexical and syntactic analyses discussed above, while the Geometry and Motion sections each contain both semantic analysis and execution for a subset of the language.

The Input section reads the program (known conventionally as a **part-program**) and produces a listing, together with a sequential file which is used as input to the Decode section. During this process all reserved words are recognised and replaced by integer codes, as are any arithmetic operators or parentheses. Thus, for example, the part-program definition statement

P1 = POINT/XSMALL,INTOF,L1,C1

(which defines a point (P1) as that point of intersection of the line L1 and the circle C1 which has the smaller X-coordinate) will be converted to a 20-word sequence, as shown in figure 2.1.

The Decode section carries out the main syntactic analysis on the program, and also deals with the semantics and execution of any arithmetic. It first scans a record for any "nested expressions" (i.e. expressions enclosed in parentheses) and analyses these as separate statements, starting at the lowest level of nesting.

Word	Statement	Internal format	
1		sequence no.	generated by Input
2		(blank)	for statement label
3	P1	0	class for identifier
4		P1	
5	=	1	class for operator
6		38	sub-class for =
7	POINT	10	class for definition major word
8		1	sub-class for POINT
9	/	1	class for operator
10		44	sub-class for /
11	XSMALL	200	class for XSMALL etc. group of modifiers
12		0	sub-class for XSMALL
13	INTOF	225	class for INTOF
14		-1	(no sub-class needed)
15	L1	0	class for identifier
16		L1	
17	C1	0	class for identifier
18		C1	
19		1	class for operator
20		63	sub-class for "end of statement"

Figure 2.1 Output produced by Input section

If the statement is a geometric definition then word 3 will be 0 (indicating an undefined identifier), words 5 and 6 will be 1 and 38 (indicating =), and word 7 will be 10 (indicating a geometric major word, i.e. POINT, LINE, CIRCLE, etc.). In this case the syntax is checked using a method based on valid successors to each individual item [Brown, 1965] which leads to a unique integer being derived for each valid definition. In the case being considered the first step will be to look up the names L1 and C1 in the vocabulary table. If these have already been defined (as they should have been) they will be discovered to refer to a line and a circle, respectively. The searching of the point definitions will then lead to the number 6 as the "identity" of this form of point definition. The class of P1 (as well as of L1 and C1) will then be changed to 5 (indicating the name of a defined geometric variable) and the name P1 entered in the vocabulary table as the name of a point. If the name P1 had been followed by a subscript then this would have been stored in words 5 and 6; if the name is unsubscripted (as in this case) then the value zero is stored in word 6, with the code for an integer (17) in word 5. The definition number and a number representing the particular combination of

(minor) modifiers (in this case only XSMALL) are then placed in words 7 and 8. These are followed by any variable names (e.g. L1 and C1 in this case) or arithmetic values that are referred to in the definition. Finally the modified record is output by Decode for use by the subsequent Geometry section, as shown in figure 2.2.

Word	Contents	
1	sequence no.	
2	(blank)	for statement label
3	5	class for geometric variable name
4	P1	
5	17	class for integer
6	0	subscript - zero because P1 is unsubscripted
7	6	definition number
8	0	modifier combination
9	5	
10	L1	
11	5	
12	C1	
13	1	
14	63	"end of statement"

Figure 2.2 Output produced by Geometry section

The remaining types of statement in NELAPT are relatively easy to decode, and are analysed simply by inspection. For example there are only a total of ten different formats for "motion" statements, no more than four of which can occur for any one of the ten motion commands (GOTO, GORGT, etc.). Once again a sequence of coded integers and alphanumeric names is formed for subsequent processing by (in this case) the Motion section.

Any other types of statement are essentially copied to the output file, except that any defined geometric names are given a class code of 5 and any arithmetic expressions are evaluated and their value stored in the output record.

The Geometry section comes next, and essentially only processes geometric definitions (i.e. records in which word 3 has the value 5). The definition number in word 7 is used in a computed-GOTO statement to call the appropriate subroutine to analyse the complete record and to use the

information therein to calculate the mathematical representation of the geometrical entity; this mathematical representation is the same as that used in APT, and is called the **canonical form**. Geometric definition statements therefore do not give rise to any output.

The other types of record on the file read by the Geometry section are motion statements, which will usually contain a reference to one or more geometric entities, and post-processor statements, which are for use at a later, post-processing, stage when the output from NELAPT (in the form of a **CLTAPE**) is tailored to suit a particular machine-tool/controller system. There are no other types of statement (such as, for example, IF or JUMPTO control statements) in NELAPT. The Geometry section passes any post-processor statements directly to its output file, but modifies any motion statements by replacing the references (by name) to geometrical entities by their canonical forms before sending the record to the output file. Thus each record on the output file is totally self-contained.

Finally, the Motion (or Tool Offset) section reads the file just produced by the Geometry section and uses the information it contains to calculate all the required tool movements. The output from this final stage is a CLTAPE file containing records in a standard format which detail the tool motion required, together with other (post-processor) information such as the details of the cutting-tools to be used, required feedrates and spindle-speeds, coolant properties, etc.

The NELAPT processor therefore differs slightly from the model described earlier in which the lexical, syntactic and semantic analyses give rise to an intermediate language file which is used during execution. Instead, the initial Input (lexical analysis) phase produces a coded file which is successively refined by the following stages, each of which takes at least some types of statement right through to Execution. One interesting aspect of this approach is that, because all geometric entities are fully defined before any of the motion statements are executed, it is clearly not possible to redefine any geometric entity. This is one of the fundamental rules of NELAPT (as it was of APT III, though not of APT IV), and clearly without such a restriction the whole method would collapse.

There is one other aspect of NELAPT which must be mentioned, namely the way in which the tables used by the Input and Decode sections are set up. This is achieved by a special **Data File Maintenance Program**, which reads three sets of data, as described below, and produces two Fortran Block Data subprograms which are then compiled with the Input and Decode programs, respectively, in order to initialise the vocabulary and decoding tables.

The first set of data contains a list of all the **reserved words**, together with their integer class and sub-class codes. These are used to create the initial vocabulary tables used by Input. The second and third sets of data are known as the **successor lists** and **definition formats**, and are used to create tables for use by Decode. Their use is best explained by means of an example.

NELAPT has four ways of defining a plane surface, as follows:

```
PL1 = PLANE/CANON,s,s,s,s
PL2 = PLANE/s,s,s,s
PL3 = PLANE/P1,P2,P3
PL4 = PLANE/P1,PARLEL,PLA
```

where s is a scalar value, P1, P2 and P3 are previously defined points, PLA is a previously defined plane, and CANON and PARLEL are reserved (minor) words.

The successor lists for a plane surface definition consist of lists of entities that can follow (i.e. be successors to) other entities in a syntactically valid definition. The first such list indicates what entities can immediately follow the / after the (major) reserved word PLANE. This first list is preceded by the name of this "file" - SPL (Successors to Plane) in this case. It is apparent from examining the above formats that there are three possible successors to PLANE/ leading to the following first list:

```
SPL,3,CANON,S,P
```

to indicate that the word CANON, or a scalar (S) or a point (P) may follow PLANE/. Each of these must then have its own list of successors, and so on until all cases have been dealt with. Figure 2.3 shows the complete set of successor lists for these definitions, where ES means "end of statement".

```
SPL,      3,  CANON,S,P.  
CANON,    1,  S.  
S,        2,  S,ES.  
P,        3,  P,PARLEL,ES.  
PARLEL,   1,  PL.  
PL,       1,  ES.  
STO.
```

Figure 2.3 Successor lists for PLANE definitions

The same symbols are then used again to produce the definition formats - each of which has a unique integer code. Figure 2.4 shows how these appear for the plane definitions detailed above.

```
DPL.  
CANON,S,S,S,S = 90.  
S,S,S,S = 90.  
P,P,P = 91.  
P,PARLEL,PL = 92.  
STO.
```

Figure 2.4 Definition formats for PLANE definitions

The data file maintenance program first reads the successor lists and creates a data structure to represent them. Figure 2.5 shows the coded data structure that results from the list of plane definitions shown in figure 2.3, and it can be seen that the first item is the number of entities that may follow the / (i.e. 3 in the case of a plane definition). This is then followed by that number of triplets - one for each entity. Each triplet has as its first item the code for the entity, as its second item the number of possible successors to that entity, and finally a "pointer" to the first of that number of further (consecutive) triplets. Notice, in figure 2.5, that the triplets starting at locations 11 and 17 refer back to themselves. This is because the successor lists for S and P include S and P respectively.

This representation is stored in an array in a COMMON block, and the "location" shown in figure 2.5 is thus the subscript to that array. The triplet structure can also be thought of as a representation of a graph or tree-structure, as shown in figure 2.6.

Location	Value	Meaning
1	3	three successors to PLANE/
2	220	CANON
3	1	one successor to CANON
4	5	(successor at location 5)
5	2	S (scalar)
6	2	two successors to S
7	11	(first successor at location 11)
8	20	P (point)
9	3	three successors to P
10	17	(first successor at location 17)
11	2	S (scalar)
12	2	
13	11	
14	1	End of statement
15	0	No successors
16	0	
17	20	P (point)
18	3	
19	17	
20	266	PARLEL
21	1	
22	26	
23	1	End of statement
24	0	
25	0	
26	26	PL (plane)
27	1	
28	14	

Figure 2.5 Coded successor lists for PLANE definitions

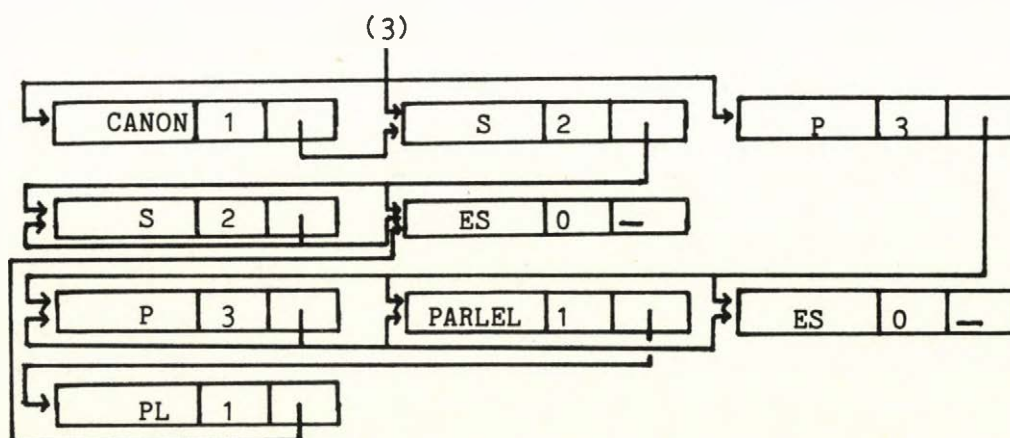


Figure 2.6 A graphical representation of figure 2.5

The list of allowable definitions (see figure 2.4) is then read and a unique number for each syntactically valid sequence is generated by means of Brown's method for encoding sequences of correlated characters [Brown, 1965]. These numbers, together with the associated definition numbers are then stored in a table using a conventional hashing technique. In the case of the plane definitions these four pairs of numbers are

(25,90), (24,90), (29,91) and (8,92)

During the decoding process a statement is first checked against the appropriate successor lists, and if it passes that test then it is encoded using the same algorithm as was used by the data file maintenance program. The resulting code number is then searched for in the definition table and if it is found then the second number of the pair is used to cause appropriate processing to take place; if the code is not found then the statement has an invalid syntax, even though it may obey the successor rules. Thus, for example, the statement

PLX = PLANE/2.5,4.7

satisfies the successor rules, but is not a valid definition.

2.3 An Outline of the APT IV Processor

Like NELAPT, APT IV was developed during the mid-1960's. However, whereas NELAPT followed a similar philosophy to APT III, with a somewhat restricted subset of the APT language, the APT IV processor was designed to provide a largely computer-independent processor for the complete APT language. Thus, although many of the subroutines for use at execution time were carried over from APT III largely unaltered, the lexical, syntactic and semantic analysis phases were totally re-written.

One of the reasons for this was a recognition by the design team that the APT language was a programming language like Fortran or Algol, albeit a special purpose language, and that the principles of compiler design which had been, and still were being, developed for general purpose languages would apply equally to APT. This was a major philosophical change from APT III and its predecessor APT II.

All the early APT literature (for example [Ross, 1960] [Bates, 1962]) refers to the APT part-program being a sequence of instructions to an "APT computer", and states that this APT computer would process the APT part-program to produce a control tape for a numerically-controlled machine-tool. Of course the APT computer did not actually exist and was simulated on a real computer such as the IBM 704 or 7090 (in the first instance), but, nevertheless, the philosophy was that this APT computer directly obeyed the APT part-program statements. In practice, the simulated APT computer processed the part-program in several stages in a similar way to that already described for the NELAPT processor.

By 1964, however, when the APT IV design was being produced [IITRI, 1964], a great deal of progress had been made in both hardware and software development, and the "APT New System" was intended to exploit the then state-of-the-art. The pilot implementation of the New System [IITRI, 1965] identified four major functions in an APT processor (Translator, Post-Translator, Subroutine Library and CLTAPE Editor) and by separating these functions endeavoured to specify the bulk of the processor in a computer-independent fashion. The first implementation (other than the development one) was made in England by English Electric Computers later in the same year by a team of three in the remarkably short time of $4\frac{1}{2}$ months [Ellis, 1966]. This implementation identified some desirable changes, especially in the link between the Translator and Post-Translator, which were simple and yet of fundamental importance [EELM, 1966a] [Ellis, 1967], and the incorporation of these (or variants of them) was the only significant design change that was made before the official release of APT IV after several years of "field trials" [IITRI, 1971].

Essentially the APT IV Translator is the complete **analysis** phase, whereas the Post-Translator is the **synthesis** phase, in the sense defined above in section 2.1. The APT IV design is such that, apart from a handful of well-specified assembly-code routines, the Translator is completely computer-independent. The separate Post-Translator allows the implementor the option of either code-generation or interpretation, using the intermediate language (I.L.) produced by the Translator. One of the results of the changes recommended by English Electric was that this phase became much simpler and, in particular, that it became possible to write an interpreter in Fortran. This phase in APT IV is now known, therefore, as Execution

Initialisation, and the implementor may either use the Fortran interpreter supplied or write his own code generator and follow a compiler approach.

The Translator contains all three analysis phases (lexical, syntactic and semantic) and is based upon the **production method** first described by Floyd [Floyd, 1961] and now a standard method for syntax-driven translators and compilers. The Translator actually uses two, independent, sets of production tables - one to process the basic syntax of the statements (including their lexical analysis), and the other to deal with the semantics of the very many forms of geometric definition statements.

The main production table consists of two parts. The first part is used by the Translator to carry out the lexical analysis of the part-program statement, and the remaining part is used to perform the syntactic (and some semantic) analysis.

The Translator reads a statement character by character using the first part of the production table to determine the next course of action. This action may be to concatenate the character with a partially formed name (or number), or to store a name in the vocabulary table (or **name table**), or to store a name, number or special symbol in a **stack**. Every time a complete entity is added to the stack the remainder of the production table is searched and compared with the stack. If the top item in the stack (the last item added) matches the first item in a production then the next item in the stack is compared with the second item in the production, and so on. If the end of the production is reached before the stack is exhausted then a positive integer value is returned by the searching routine and used in a Computed GOTO to initiate appropriate processing of the stack. If there is a difference between the items in the production and those in the stack then no match is possible and searching continues for another possible match. If no match is made with any production then a syntactic error has occurred in the input statement, since every valid combination of symbols will find a match somewhere in the production table.

The routine which carries out the comparison of the stack and production table is highly computer-dependent since it is working at the

level of individual bits of a word; however the rest of the process is computer-independent both in concept and in implementation.

This method of processing means that any nested definitions or arithmetic expressions are dealt with automatically, since the nested items will be recognised and processed before the full statement has even been read. Thus, for example, if we consider the following statement

C1 = CIRCLE/CENTER,(POINT/INTOF,L1,L2),RADIUS,1.5

we shall find that (considerably simplified) the stack will be built up as follows:

- i) C1
- ii) C1 =
- iii) C1 = CIRCLE
- iv) C1 = CIRCLE /
- v) C1 = CIRCLE / CENTER
- vi) C1 = CIRCLE / CENTER ,

At this point a production is matched which recognises that a list of **arguments** is about to follow the slash (/). Since this list is of unknown length production matching would become impossible, and so the arguments are removed from the stack and copied to a special **argument stack**:

- vii) C1 = CIRCLE /

The next item, however, is not a potential argument:

- viii) C1 = CIRCLE / (

Transfers to the argument stack are, therefore, temporarily stopped while the nested item is dealt with:

- ix) C1 = CIRCLE / (POINT
- x) C1 = CIRCLE / (POINT /
- xi) C1 = CIRCLE / (POINT / INTOF
- xii) C1 = CIRCLE / (POINT / INTOF ,

The last four items are now of exactly the same syntactic form as at step (vi), namely

"geometric surface type" / "identifier or name" ,

and so the argument is transferred to the argument stack before processing continues:

- xiii) C1 = CIRCLE / (POINT /
- xiv) C1 = CIRCLE / (POINT / L1

xv) C1 = CIRCLE / (POINT / L1 ,

and once again the argument is transferred to the argument stack:

xvi) C1 = CIRCLE / (POINT /
xvii) C1 = CIRCLE / (POINT / L2
xviii) C1 = CIRCLE / (POINT / L2)

The last five items now match with a different production - one which recognises that the end of a nested definition has been reached. A different part of the Translator is therefore called into play to process this definition, including the two items (INTOF and L1) already transferred to the stack. This is done by copying the remaining argument (L2) to the argument stack, inserting the major word (POINT) at the correct place (i.e. before INTOF), and then checking the complete definition against a second set of production tables which contain the correct syntax of all geometric definitions. If a match is found then, as with the main processing, an integer value is returned and used to control a Computed GOTO which initiates the appropriate processing. If no match is found then the syntax of the definition is invalid and a diagnostic message is produced.

During this geometric processing an I.L. record will be generated which will cause a point to be defined using a special name created by the Translator (in the absence of a user-defined one). The I.L. produced will be equivalent to that which would have been produced by the statement

\$19T\$1 = POINT/INTOF,L1,L2

(where the special name \$19T\$1 indicates that this is the first temporary name generated for a point - surface type 19). The stack is then altered so that the name of the point replaces the nested definition:

xix) C1 = CIRCLE / \$19T\$1

Processing then continues as before:

xx) C1 = CIRCLE / \$19T\$1 ,

leading (after argument transfer) to

xxi) C1 = CIRCLE /
xxii) C1 = CIRCLE / RADIUS
xxiii) C1 = CIRCLE / RADIUS ,

leading (after argument transfer) to

xxiv) C1 = CIRCLE /


```
xxv) C1 = CIRCLE / 1.5  
xxvi) C1 = CIRCLE / 1.5 ↵
```

where ↵ indicates "end of statement".

At this point geometric processing again takes over and, using the argument stack as before, the definition will be recognised and I.L. produced. Since the statement is not nested nothing is left behind and the stack is emptied before processing continues with the next statement.

Thus we see that lexical, syntactic and semantic analyses all use the same technique and, not suprisingly, it follows that they all use the same computer-dependent bit-matching routines with which to carry out the necessary searching and comparison.

The result of this analysis phase is an intermediate language **program** expressed in an all-integer form. This, in fact, was the most obvious change made during the development of APT IV from the Pilot New System, as the earlier system used an I.L. consisting of alphanumeric text in a form very suitable for use by a macro assembler (such as that on the IBM 7090 on which the Pilot New System was developed). The final form of the I.L., however, follows the English Electric approach [Ellis, 1967] and consists of an integer **opcode** (operation code), the number of operands to follow, and that number of operands. Most of the operands are in pairs with the first number defining the meaning of the second one; all names and numbers are referred to by their index in the appropriate vocabulary table.

The Translator does contain a facility for listing the I.L. produced in either, or both, of two forms. CIL (Compressed I.L.) is the form actually output, while IL shows the meaning of the various integers; in the latter case the first number of an integer operand pair is printed as one or more \$ signs instead of as an integer. Figure 2.7 shows the I.L. produced on an ICL 1906S implementation of APT IV for the statement discussed above.

```

SEQNCE 17
  17. C1 = CIRCLE/CENTER,(POINT/INTOF,L1,L2),RADIUS,1.5
    9      1      17
CALL  APT003 0 $19T$1 0 L1 0 L2
  18      7      102      0      621      0      286      0      406
RESRV C1 4 7 1
  12      4      452      4      7      1
CALL  APT049 0 C1 0 $19T$1 $ 1.5
  18      7      143      0      452      0      621      1      1293

```

Figure 2.7 APT IV Intermediate Language (CIL and IL)

It can be seen from this that the statement gives rise to three I.L. commands (plus a SEQNCE command). The first of these is a call to the execution subroutine APT003 to define the point, the second is a request to reserve storage for the canonical form of the circle C1, while the third is a call to the execution subroutine APT049 to define the circle.

The final stage of processing is execution, although this actually takes place in two stages. The first stage is the execution of the I.L. produced by the Translator to produce a CLTAPE in a standard format. This process uses the large library of special subroutines referred to earlier as the Subroutine Library. After the I.L. has been fully executed the last stage, known as the CLTAPE Editor, is initiated. This, as its name implies, carries out certain editing functions on the CLTAPE such as, for example, the repetition of parts of it (possibly with some coordinate transformation), and may also provide a listing of the contents of the tape. Although the mathematics of much of the execution phase processing is extremely sophisticated, the program structure is very straightforward and of no particular concern in the present context.

One matter which is of considerable interest, however, is the means by which the production tables used by the Translator are created. As was the case with NELAPT, these tables are created by a special program, known as the Load Complex, which creates a series of Fortran Block Data subprograms. We can illustrate the method used by means of an example.

In the above discussion, the statement was progressively reduced until the final stage involved the processing of the stack in the form

C1 = CIRCLE / 1.5 ─

after the other arguments had been transferred to the argument stack. The production that recognises this syntax is number B.33, which in a short-hand notation can be written

$$\vdash \langle v+i+ig+iG+ip+t \rangle = g / \langle v+t+n+i+i' \rangle \vdash \rightarrow \text{null } 38$$

where \vdash means left terminator (beginning of statement)
 \vdash means right terminator (end of statement)
v means variable name
i means identifier
ig means conditional geometric surface name
iG means conditional large geometric surface name
ip means conditional procedure name
t means temporary name (allocated by the Translator)
g means geometric name (major word)
n means number
i' means permanent identifier (reserved word)

This production defines all the possible classes of item that may appear on the left of the equals sign (the conditional classes are there because in APT IV most APT language words may be redefined as variable names if required), and all the possible classes which may be arguments in a geometric definition. As we have already seen, it will cause the remaining argument to be transferred to the argument stack for further analysis. If a match is found then appropriate I.L. is produced, the class of the item on the left of the = sign is changed to "variable" (if it is not already so classified), and the stack reduced to a null state. The number 38 is the value which will be returned by the initial production matching to cause the final geometric processing to take place.

This production is supplied to the Load Complex as follows:

```
RTERM
VBL,TEMP,NUMBER,IDENT,PERMID
/
GEOM
=
VBL,IDENT,CONGEO,CONBIG,CONPRO,TEMP
LTERM
38
```

The Load Complex reads all the productions and assigns two numbers to each class (i.e. VBL, TEMP, etc.) - the first is a set number and the second is a number within that set. All classes which may occur as alternatives are given the same set number. Thus if the above production were the first to be read (which it is not) the set and number assignments would be as shown in figure 2.8.

Class	Set	Number
RTERM	1	1
VBL	2	1
TEMP	2	2
NUMBER	2	3
IDENT	2	4
PERMID	2	5
/	3	1
GEOM	4	1
=	5	1
CONGEO	2	6
CONBIG	2	7
CONPRO	2	8
LTERM	6	1

Figure 2.8 An example of Set and Number assignments

As further productions are read changes may need to be made to assignments already made; however, once all the productions have been read a table will exist which includes all valid classes in sets such that no class can appear as an alternative to a class in a different set, and every class appears at least once as an alternative to at least one of the other classes in its set (unless it is the sole member of that set). The Load Complex then interrogates a computer-dependent variable to establish the number of bits in a (Fortran) word (excluding the sign bit), and attempts to allocate unique bit patterns to each class - a zone bit (corresponding to the set) and an element bit (corresponding to the number within the set). The largest set is used to determine the number of bits to be allocated for element bits, and if insufficient bits are left to allocate one for each zone then the smaller sets are combined until the number of sets equals the number of zone bits available. If this is not possible an error diagnostic is produced.

The production table is then built up by storing this internal representation of each class in successive locations. Where there are

alternatives a logical OR of the various bit patterns is performed - by definition they all have the same zone bit. Finally the number which defines the action is stored as a negative value, and since the sign bit is not used for internal representations of classes this uniquely defines the end of each production.

When the complete production table has been analysed in this way Fortran Block Data subprograms are produced to recreate these bit patterns for the Translator, together with a table giving the bit representation for each class. Thus in the writer's implementation of APT IV (version A4V1) on the ICL 1906S the part of the production table corresponding to the production B.33 discussed above is as follows:

```
DATA PRODTB(227)/8H80008000/
DATA PRODTB(228)/8H 00024$2/
DATA PRODTB(229)/8H80004000/
DATA PRODTB(230)/8H 0000@00/
DATA PRODTB(231)/8H80002000/
DATA PRODTB(232)/8H 00000'B/
DATA PRODTB(233)/8H0@00 000/
DATA PRODTB(234)/- 38/
```

Written in octal form the contents of these locations are:

1000000010000000	i.e. \neg
2000000002042402	i.e. $\langle v+t+n+i+i' \rangle$
1000000004000000	i.e. /
2000000000400000	i.e. g
1000000002000000	i.e. =
2000000000002742	i.e. $\langle v+i+ig+iG+ip+t \rangle$
0040000020000000	i.e. \vdash

Examination of these patterns enables us to see that while the second and sixth items (the two with alternatives) have the same zone bit, there are differences in the element bits - the only element bits which are common being those corresponding to the classes represented by the following:

```
2000000000002000
2000000000000400
2000000000000002
```

which, not suprisingly, are the bit patterns for "v", "i" and "t", respectively.

During translation, as each APT statement is input it is first dealt with character at a time, and then (after the initial (basic) productions have formed names, numbers and other character strings) in larger units.

The stack itself is in two columns; the first contains a pointer to the name table (which contains all APT words, all names and numbers which have appeared in the part-program so far, and the complete character set), while the second column contains the bit pattern for the class (or provisional class) of the item. After each new item has been added to the stack it is compared with the production table. A match is made between a stack item and a production table item if all the bits which are set in the stack item are also set in the production table item. Such a test is, of course, a trivial logical test at the assembly code level.

If the top of the stack matches the first item of a production then the process is repeated with the next items. If a negative production table item is reached then this signifies the end of the production and so a complete match has been made; since the negative number represents the number of the production its absolute value is returned. If one of the stack items does not match then the process is repeated with the next production. The final production is a null (or empty) production which, if reached, will therefore match anything; it will give rise to an error message indicating invalid syntax.

2.4 Language Modification in APT IV and NELAPT

Despite their very different approaches to the processing of similar languages, both APT IV and NELAPT have one important feature in common - both use a separate program to generate the tables used in the analysis of the part-program statements. At first sight, therefore, it might be considered that in order to add a new definition it should merely be necessary to alter the data for the Data File Maintenance or Load Complex program and to add an appropriate subroutine or two. In practice, however, it is not so easy.

The main reason for difficulty is that both APT and NELAPT are large and very complex suites of programs with extremely complicated underlying data structures. The insertion of even a trivial extra subroutine therefore requires a detailed knowledge of some very complex COMMON blocks, and may also require the use of several "utility" subroutines. NELAPT is

worse than APT in this respect since the layout of the data is considerably less structured, and hence is more difficult to use correctly.

In the APT IV Translator there are some 40 COMMON blocks, each containing data for only one purpose such as the Name Table, the processing Stack, the basic Production Table, etc. Many of these are very small (e.g. block 11 contains only two items - the current input statement sequence number as an integer and as a character string!), but this sub-division makes their use and documentation relatively easy. The APT IV Subroutine Library uses a single COMMON block for most of its storage, but this is defined as consisting of a number of large arrays, each of which is treated as though it were a separate COMMON block by means of EQUIVALENCE statements, thus achieving the same well-defined and logical structure.

The four NELAPT programs (Input, Decode, Geometry and Motion), on the other hand, keep most of their data in a single COMMON block (ABLANK, BBLANK, etc.) which is defined differently in almost every subroutine (so as to only refer to those items used by the particular subroutine). Although the documentation does give the detailed layout, the fact that this follows no logical pattern and is, in any case, nowhere apparent in the program itself means that it is exceedingly difficult to understand. This problem is compounded by the fact that very few of the NELAPT subroutines have any arguments but operate solely by means of information stored in a COMMON block, with the result that the linkage and usage of utility (and other) subroutines is extremely difficult to establish.

Another important point concerns the style of the programs. All APT IV subroutines are, as far as possible, self-documenting and start with a lengthy sequence of comment statements which detail the purpose of the routine, the method used (where appropriate), the arguments provided in the calling sequence, and the results produced; in addition the subroutines use (in general) fairly meaningful names for variables and arrays and are liberally annotated with comments throughout. NELAPT, on the other hand, uses meaningless variable names, and has virtually no comments at all; it is probably the least self-documenting program the writer has ever encountered, and is an excellent example of how **not** to write a large and complex program.

Nevertheless, the complications (or otherwise) involved in adding to the code of a particular system are only part of the story. It will also be necessary to add extra information for processing by the Data File Maintenance or Load Complex programs. Here again we see a difference in philosophy.

In NELAPT the fundamental syntax of the language is incorporated in the code of the Input and Decode programs. The tables produced by the Data File Maintenance program are not, therefore, concerned with the syntax of individual statements and are used only interpret statements whose basic syntax has already been checked; they are thus used to

- a) Recognise NELAPT language (reserved) words
- b) Decode geometric definition statements of the form
 name = geom/.....
 (name = geom/.....)
 (geom/.....)

In APT IV, however, the tables produced by the Load Complex do contain information about the syntax of the language and are therefore used to

- a) Recognise APT language words
- b) Recognise and interpret syntactically valid statements
- c) Decode geometric definition statements of the form
 name = geom/.....
 (name = geom/.....)
 (geom/.....)

The importance of this difference can be seen by considering the steps which are necessary to add the (non-standard) point definition

$P1 = (x,y)$ or $P2 = (x,y,z)$

to the language, including nested definitions of the same formats.

In APT this is easily achieved by adding two extra productions to the geometric production table. When either of these is recognised it returns an integer code which causes the items on the stack and argument stack to be adjusted into one of the forms produced by the standard point definitions; this is followed by a jump to the normal processing for a

statement of the form

P1 = POINT/x,y or P2 = POINT/x,y,z

In NELAPT considerably more effort is required because the appearance of a left parenthesis normally initiates the processing of a nested definition. Some rather complicated programming is therefore required, firstly to recognise that this is a point and not a nested definition, secondly to process the definition, and thirdly to reorganise the input record so that on exit from the "nested-definition analysis" the interface with the remainder of the Decode program is correct. In the writer's implementation this required 64 lines of Fortran code (plus 22 lines of comments!), and inevitably creates an overhead in the processing of all nested definitions due to the several extra checks which must be made to establish whether they are special point definitions. In addition it was also necessary to modify the Input (lexical analysis) program because in the standard version the nested expression (A-B) and the point definition (A,-B) were written to the interface file in an identical fashion. This was dealt with by making the Input program insert extra parentheses in the special point definition so that it read (A,(-B)), thus ensuring that the sign was dealt with (as part of a nested arithmetic definition) by the nesting procedures. Figure 2.9 illustrates this problem and its solution.

Word	Format in output file			
	(A-B)	original (A,-B)	modified (A,-B)	modified (A,-B) after "de-nesting"
n	((((
n+1				
n+2	A	A	A	A
n+3				
n+4	-	-	(///1
n+5				
n+6	B	B	-)
n+7				
n+8))	B	
n+9				
n+10)	
n+11				
n+12)	
n+13				
.				
.				

Figure 2.9 Input processing of parenthesised expressions

In general, however, any changes made would not involve changes to the underlying syntax but would consist of new, or modified, **action** statements of the form

action/.....

or new, or modified, **definition** statements of the form

name = def/.....

(or nested variations of this).

Neither APT IV nor NELAPT do any checking on action statements at the decoding stage but simply pass them through for subsequent analysis during the execution phase. This is because the majority of these types of statement are, in fact, post-processor commands and will never be processed by the main processor, while the remainder (the motion statements) are few in number and have a simple, easily identified, syntax. Thus any changes in the motion statements will require the writing of new, and very complex, code for direct interfacing with the execution phase. This will require a considerable understanding of how this aspect of the processor works and is not a task to be undertaken lightly. (Indeed it could be a major reason for the spread of APT-like languages since it might be easier, in many cases, to write a new set of execution phase subroutines than to modify existing ones!). In general, however, there is not likely to be any significant requirement for language changes in this area.

Definition statements, on the other hand, are a very different matter. Here there is a very wide range of formats for the list of arguments, and both APT and NELAPT, as we have seen, use tables generated from user-supplied data to analyse the argument list. In both cases, therefore, the addition of new, or modified, definitions involves the writing of one or more subroutines to process the argument list and the provision of the syntax in the appropriate form.

The differences in the case of program modification (mainly due to the layout of COMMON data storage) have already been referred to. The differences in ease of syntax specification are best illustrated by an example.

As part of an investigation into the ease of modification of APT IV and NELAPT the writer added the following three new definitions to his implementations of the two processors:

- i) POINT/circle,HEIGHT,d
defines a point at a height d above (or below) the centre of the specified circle;
- ii) LINE/point
defines a line connecting the specified point to the origin;
- iii) PATTERN/LINEAR,point,ATANGL,angle,INCR,.....
defines a linear pattern starting at the specified point and spaced along a line which makes the specified angle with the X-axis. The spacing of the points follows the word INCR and uses the normal range of options.

In APT IV this merely required the addition of three new definitions in the data for the Load Complex as follows:

- i) in the point definitions:
CIRCLE
HEIGHT
REAL
501
- ii) in the line definitions:
POINT
502
- iii) in the pattern definitions:
LINEAR
POINT
ATANGL
REAL
INCR
21

In addition, the word HEIGHT was added to the list of known vocabulary words as an identifier.

The changes necessary with NELAPT were rather more awkward because both the successor and the definition tables needed alteration. In this case the changes were as follows:

- i) Create new point successor lists as follows (where the changes are underlined):

SPT,	8,	CANON,S,,L,INTOF,XSMALL,CENTER,P,C.
CANON,	1,	S.
S,	2,	S,ES.
L,	6,	XCOORD,L,ES,C,S,T.
INTOF,	3,	L,C,T.
XSMALL,	3,	INTOF,L,T.
CENTER,	1,	C.
P,	6,	XSMALL,CLW,DELTA,THETAR,ES,T.
C,	<u>5</u> ,	ES,C,S,ATANGL, <u>HEIGHT</u> .
<u>HEIGHT</u> ,	<u>1</u> ,	<u>S.</u>
XCOORD,	1,	S.
T,	3,	P,S,T.
CLW,	1,	C.
DELTA,	1,	S.
THETAR,	1,	S.
ATANGL,	1,	S.
STO.		

Add new point definition:

C,HEIGHT,S=17.

- ii) No changes are necessary in the successor list; the new line definition is:

P=41.

- iii) Create new pattern successor lists as follows:

SPA,	6,	LINEAR,ARC,PARLEL,RANDOM,CIRCUL,MIRROR.
LINEAR,	1,	P.
P,	<u>6</u> ,	S,V,P,PAT,ES, <u>ATANGL</u> .
S,	<u>4</u> ,	ES,S,CLW, <u>INCR</u> .
V,	2,	S,INCR.
INCR,	1,	S.
C,	1,	S.
CLW,	2,	S,INCR.
L,	1,	PAT.
PAT,	4,	P,PAT,ES,V.
ARC,	1,	C.
PARLEL,	1,	PAT.
RANDOM,	2,	P,PAT.
CIRCUL,	1,	PAT.
MIRROR,	1,	L.
<u>ATANGL</u> ,	<u>1</u> ,	<u>S.</u>
STO.		

Add new pattern definition:

LINEAR,P,ATANGL,S,INCR,S=132

In addition, as with APT, the word HEIGHT was added to the list of known (and reserved) vocabulary words.

It is apparent from the above that the procedure is very similar with both systems, although slightly more awkward in NELAPT due to the need to change the successor tables. The three additional definitions were, therefore, easy to add to the internal language definition tables, although the writing and interfacing of the necessary subroutines was, as already indicated, only possible (especially with NELAPT) with the aid of a substantial amount of detailed knowledge of the coding and data layout of the Translator and Decode programs, respectively.

The above discussion has been concerned with the problems associated with the addition of new language features to APT IV and NELAPT. However, equally important is the ease (or otherwise) with which features of the language may be removed, since one of the major reasons for the spread of "APT-like" language processors was the need for smaller systems which did not accept the full range of APT geometry and motion commands. In its full form APT will deal with any surfaces that can be mathematically defined (including surfaces fitted to a mesh of data points) and can control a tool with 6 axes of motion (three linear and three rotational) across any or all of these types of surfaces. Most manufacturing organisations, on the other hand, are quite content to work with only three linear axes of motion and with a considerably simplified set of geometric surfaces (frequently only 2-dimensional).

The removal of parts of the APT IV processor would, however, be almost impossible without a considerable study of the program structure and code. For example, the Translator (version A4V1) contains 91 subroutines whose calls can be nested up to 15 levels deep; on a paged ICL 1906S computer it occupied 65536 24-bit words (c. 262K bytes). Removing any part of this program would clearly be a major task. The Execution Complex (i.e. the Subroutine Library plus Fortran Interpreter), on the other hand, contains a total of 274 subroutines, of which 211 are overlaid using fifteen different overlays; the size of the program, even with the extensive overlaying, is

60352 words (c. 241K bytes). [The overlaying is necessary even though the 1906S is a paged machine because of hardware restrictions on the amount of memory available for certain types of variables; if not overlaid the program would occupy 120223 words.] The very complex linkage between different subroutines, especially those concerned with the calculation of the tool path, and the clever, but complicated, arrangement of the global (COMMON) data would once again make any removal extremely difficult.

Suprisingly, in view of its much smaller capability, the NELAPT processor has an equally (and perhaps more) complicated structure. On the ICL 1906S the Input section contains 51 routines and occupies 13696 words (c. 55K bytes), while the Decode section contains 68 routines and occupies 25536 words (c. 102K bytes). The Geometry and Motion sections, however, are overlaid (due to the limited data space referred to above). The Geometry section contains 157 routines, of which 138 are in a complicated overlay structure which uses 25 separate overlay units, and occupies 22258 words (c. 89K bytes), while the Motion section contains 131 routines, of which 114 are in an even more complicated structure of 35 overlay units, and occupies 26432 words (c. 106K bytes). Once again the linkage between routines and the layout of global data is difficult to fully comprehend, and, as mentioned earlier, the way in which the programs have been written militates against any easy understanding of their purpose and method.

The experience obtained during this investigation, therefore, showed that the size of an AOL processor such as APT, or even NELAPT, would almost inevitably mean that it would have a large and complex **data-base** and a substantial number of **utility procedures** with which to carry out common activities, including many (though not all) of the accesses to the underlying data structure. It appeared, therefore, to be unlikely that such a processor could be designed so that it could be easily, and yet safely, modified by the non-expert unless a totally different and more modular structure was developed. The next chapter describes a prototype for such a processor, while the subsequent chapters then show how a number of new concepts were developed to enable the production of a genuinely user-adaptable AOL processor.

3. THE PROTOTYPE MILDAPT PROCESSOR

3.1 Design Concepts for a Modular "APT-like" Processor

The investigation into APT IV and NELAPT described in Chapter 2 led to the conclusion that if a processor for an APT-like AOL processor was to be user-adaptable then it must be designed in a modular fashion in such a way that individual modules might be added or removed (within reason) without any significant effect on the remaining modules. However, the effort required to write a complete NC processor with which to test out various ideas was clearly far too great for one person (with other, parallel, commitments) and so it was decided to build a prototype system based on either APT IV or NELAPT. (These two processors were both available in source language form and had both been implemented by the writer on ICL 1907 and, subsequently, 1906S computers. Other similar processors were either not available in source language form or else were only available at considerable cost).

As has already been indicated, the APT IV processor, despite its size and complexity, is better than the NELAPT processor in respect of the clarity of its coding and the structure of both its programs and its data. In addition, the production method used by the APT Translator is widely accepted as a standard method of syntax analysis, and its use for lexical, syntactic and (geometric) semantic analysis is more elegant and more economic than the use of several different methods at different stages, as is the case with NELAPT. It was therefore decided to develop an experimental system using APT IV-like methods, but one which would be modular in concept in order to allow for easier modification to the language it accepted - and hence to the processor itself. The programming of this system would be simplified by the use of as much of the code of APT IV as possible, and by restricting the system to simple two-dimensional surfaces and $2\frac{1}{2}$ -axis machining.

The major design aims of this system were identified as:

- i) The processor must consist of several modules, linked in such

a way that any module, apart from the initial (input) and final (edit) ones, must be able to be removed without any effect upon the rest of the system as long as the module is not required by a particular part-program; similarly, it must be possible to add a new or revised module without any side-effects;

- ii) All input and output must be carried out via a standard module so that the physical manner in which it takes place can easily be altered;
- iii) The processor must, as far as possible, be computer-independent, and any computer-dependent features must be clearly identified;
- iv) The final output of the processor must be a CLTAPE to international APT standards.

The key to the method adopted was the recognition (already discussed in section 2.1) that the majority of language statements will be either **action** statements of the form

action
or
action(argument1, argument2,)

or else **definition** statements of the form

name:= type(argument1, argument2,)

In particular, it was felt reasonable to assume that it was only statements of these two types (and not, for example, control statements) for which there would be any requirement for changes.

With this assumption it was possible to arrange for an initial phase of the processing to input all the part-program statements and to carry out a lexical and syntactic analysis of them, as well as a semantic analysis of statements other than action or definition types. Clearly the syntactic analysis of these two types would be limited to recognition of a valid overall structure rather than a detailed analysis of, for example, the

number and type of arguments; this detailed analysis would take place later at the same time as the semantic analysis. The result of this phase would be a form of Intermediate Language in which all action and definition statements were passed on essentially as originally input.

This Intermediate Language (I.L.) would then be processed by one or more **definition modules** (called Geometric Modules in the prototype MILDAPT system) which would analyse those definition I.L. commands that they recognised and ignore any others. In a similar way the action I.L. commands would be analysed by one or more **action modules**. Finally, the I.L. (as modified by definition and/or action modules) would be executed in a similar way to that used in APT IV.

Since a definition module would only know about a limited number of valid definitions, and would have a clearly defined interface with the rest of the system - the file of I.L. commands - it should be possible to add or remove such modules at will. In a similar manner, any action modules should also be readily replaceable.

A simple prototype system was produced, using a large amount of APT IV code, and implemented on an ICL 1906S computer during 1975/76. This system was called MILDAPT (Modular Integrated Language Driven APT) and is described in detail elsewhere [Ellis, 1977]. The prototype MILDAPT system is important only in so far as it led on to a far more general and versatile form of adaptable language processor, but in order to appreciate the background to that processor it is necessary to describe the main principles of the prototype system in some detail.

3.2 The Input Module

As mentioned above, the input module carries out the lexical and syntactic analysis (at least in part) of all input statements, and a full semantic analysis of all statements other than action or definition statements. A large part of this module was extracted from the APT IV Translator and, naturally, uses the same production method for both lexical and syntactic analysis. Thus, in a similar way to that described in section 2.3, the first part of the production table controls the lexical

analysis on a character-by-character basis, while the second (and larger) part controls the syntactic (and semantic) analysis. The major difference between the MILDAPT Input Module and the APT IV Translator concerns the treatment of action and definition statements.

In APT IV a sequence of items of the form

"geometric surface type"/"identifier or name, etc.",

or, more succinctly

$g/\langle v+t+n+i+i' \rangle,$

is recognised as a partially completed geometric statement and causes the item after the / to be transferred to an **argument stack** and the comma removed, before processing continues with the lexical input of the next item. This should lead to either

$g/\langle v+t+n+i+i' \rangle,$

if the next argument is not the last one, or to

$g/\langle v+t+n+i+i' \rangle -$

if it is the last, or to

$g/\langle v+t+n+i+i' \rangle)$

if this is a nested definition. The process is repeated as many times as necessary until the last argument has been identified, at which point processing is transferred to a separate part of the Translator which uses another set of productions to identify the particular geometric definition and then to produce appropriate I.L.

In a similar way, a sequence of items of the form

"procedure"/"identifier or name, etc.",

or more succinctly

$p/\langle v+t+n+i+i' \rangle,$

will have its arguments successively transferred to the argument stack. Unlike the geometric statement, however, a further production-based analysis is not then required.

Procedure statements in APT IV are either **motion** statements which start with one of a limited number of major words (e.g. GOLFT, GORGT, GOTO, etc.) or **post-processor** statements which are to be passed to the (machine-tool-dependent) post-processor program via the CLTAPE. The format of the arguments in motion statements is the same for most major words, and the analysis is therefore based on an initial examination of the first (major) word. If this word is not a recognised motion command then it is assumed to be a post-processor command, and is treated accordingly.

In MILDAPT the Input Module does not know about the syntax of any geometric or motion statements, but merely recognises them as being definition or action statements. Although the initial processing is the same as in APT IV, therefore, once the arguments have all been recognised and transferred to the argument stack they are simply output (together with the major word) to the file of I.L. commands for later analysis.

This partial analysis has a number of implications for the classes allocated to all recognised items, and hence for the productions themselves. In APT IV the various geometric and procedure names may be used either for that purpose or, for example, as the names of surfaces or other "variables". Initially all known APT geometric words are allocated a **conditional geometric** class, therefore, and the sequence

$\langle = + (\rangle \text{ig} /$

(where ig means "conditional geometric surface") causes the class of that particular word to be changed to **geometric** (surface). Once the statement has been reduced to the form

$\langle v + i + \text{ig} + iG + ip + t \rangle = g / \langle v + t + n + i + i' \rangle - \downarrow$

then, as described in section 2.3, the statement is fully analysed and the class of the item on the left of the equals sign is changed to **variable**.

In a similar fashion the sequence

$\downarrow ip \langle / + \downarrow + , \rangle$

causes the class of the item before the / to be changed from **conditional procedure** to **procedure**.

In MILDAPT, however, it is not possible to have a list of all possible geometric major words (with initial classes of conditional geometric) since the whole purpose of the processor is to allow the easy addition (or deletion) of definition or action statements. Thus it is perfectly acceptable to have a statement of the form

S1 = NEWSUR/"list of arguments"

which will lead to the sequence

<=+(>i/

(where i means **identifier**, or simply a name with no prior class). This is presumably acceptable and will be processed by a subsequent MILDAPT module and so MILDAPT allocates a new **implied geometric** class (gi) to the word. Subsequently the sequence

<v+vi+i+ig+iG+ip+t>=gi/<v+vi+t+n+i+i'>-|

will cause the item on the left of the equals sign to be assigned the class **implied variable** (vi).

In a similar way a sequence of the form

|i</+|-|+,>

will cause the initial item to be assigned the class **implied procedure** (pi).

The items which constitute an implied geometric definition or an implied procedure call are then incorporated in one of two new types of I.L. command - **provisional geometric** (PROGEO) and **provisional procedure** (PROPRO).

The remainder of the Input module is fairly straightforward and follows the APT IV pattern quite closely. It had been the intention to incorporate the full processing of arithmetic and control statements into the Input module, but in the initial (and, as it turned out, the only) version these were converted to APT IV-style I.L. commands and then executed by a "mini-module" which followed the Input module.

The output from the Input module is thus a file of I.L. commands of a similar nature to those produced by APT IV, together with the Name Table

containing all the names and symbols used in the part-program. This I.L. file is then passed to the first Geometric module to carry out further analysis.

3.3 The (Geometric) Definition Modules

The prototype MILDAPT system was based on the premise that each geometric (or definition) module would ignore any I.L. commands that it did not recognise and would process only those that it was designed to deal with. Thus any I.L. commands other than PROGEO commands are copied directly to the output (I.L.) file.

When a PROGEO command is encountered the subsequent items in the I.L. record are copied onto a stack. This stack is now in almost exactly the same form as the argument stack in the Input module before the output of the PROGEO I.L. command, and also in the same form as the argument stack in APT IV immediately before the secondary (geometric) production processing discussed above. It was, therefore, relatively straightforward to utilise a modified version of the corresponding part of the APT IV Translator to process this stack.

If this processing (which uses a further set of production tables which are kept within the module) fails to find a matching definition then the PROGEO command is copied to the output I.L. file for recognition (presumably) by a subsequent definition module. However, if a matching definition is found then an appropriate subroutine is called to effect the definition, and to create the **canonical form** of the surface being defined. At the same time the class of the major word is changed (if necessary) from implied geometric (gi) to geometric (g), and that of the surface name from implied variable (vi) to variable (v).

The result of processing the complete I.L. file by a single geometric module is thus an I.L. file with some PROGEO commands omitted, a modified Name Table, and a table of Canonical Forms.

After all the geometric modules have been loaded and executed there should be no PROGEO commands remaining, and all implied geometric or implied variable classes should have been changed to geometric or variable.

3.4 The Action (Procedure) Modules

The only unprocessed I.L. commands left at this stage should be PROPRO commands, and these are dealt with by one or more Procedure (or Action) modules. Like the geometric modules these first recreate an argument stack and then analyse it in an appropriate fashion. The position here, however, is slightly different from the case with the analysis of PROGEO commands because PROPRO commands may represent three completely different types of original part-program statements:

- i) Conventional motion statements
- ii) New motion or technological action (procedure) statements
- iii) Post-processor statements

The first procedure module, therefore, looks only for conventional motion statements (GOLFT, GORGT, etc.) and analyses these using code essentially the same as in the relevant part of the APT IV Translator. In particular, as mentioned above, it does not use any production tables but simply inspects the various items in turn. Any motion statements of this type give rise to one or more CALL I.L. commands for use during the subsequent execution phase. Any other forms of PROPRO (or other) I.L. commands are copied to the output file.

Any subsequent procedure modules use a similar technique (or a production-based technique if preferred) to deal with PROPRO commands arising from the relevant part-program statements.

The final procedure module (or more accurately **post-procedure module**) analyses anything that is left. Any remaining PROPRO commands are assumed to have been created as a result of post-processor statements in the original part-program and I.L. commands will be generated to reflect this, while the class of the major word will be changed, if necessary, from implied procedure (pi) to procedure (p). One exception to this process

occurs if the class of the major word has already been set to some type other than p or pi, for example if the word has already been used in some other context; in this event an appropriate diagnostic message will be produced.

The post-procedure module also checks for any PROGEO commands. Since they should all have been dealt with by one of the geometric (definition) modules, the existence of one at this stage indicates that the definition was not recognised and a diagnostic message to this effect is produced.

Finally, the post-procedure module checks that there are no "implied" classes remaining in the Name Table. Such classes are initially created by the Input module and should have been converted to a definite class by one of the geometric or procedure modules; if any remain then an error has occurred and a further diagnostic message is produced.

Diagnostics are thus produced by the post-procedure module for any invalid statements which satisfy the basic syntactic requirements of a definition or action statement. Any statements which are syntactically illegal will not, of course, be accepted by the Input module and will give rise to errors at that stage. (In the prototype MILDAPT system, therefore, two separate sets of diagnostics could be produced - one by the Input module and one by the post-procedure module; it would have been relatively simple for the Input module to pass its error information to the post-procedure module via a special I.L. command so that a single, ordered set of error messages was produced, but this was never done). If any errors have occurred at either stage then processing stops at this point; if the program is error-free (as far as can be ascertained at this stage) then processing continues with the main execution module.

3.5 The Execution Modules

The I.L. file at this stage consists essentially of a sequence of calls to routines in what APT refers to as the ARELEM complex (ARithmetic ELEMENT), interspersed with post-processor commands and, possibly, control or editing commands such as COPY or TRACUT. The ARELEM complex in APT IV is a large library of subroutines which handle all the complicated

calculations which are involved in determining the path of a tool which is to machine a defined surface. In APT IV the ARELEM consists, in fact, of two overlapping ARELEMs - one to handle simple 2-dimensional motion which can be calculated mathematically, and the other to handle more complex 2-dimensional or 3-dimensional motion for which an incremental "trial and error" approach is required.

In the prototype MILDAPT system there was only a single execution module consisting, essentially, of the APT IV 2-dimensional ARELEM. In principle different modules could have contained more sophisticated ARELEMs if required, for example to handle 3-dimensional work with only three linear axes of tool motion, full 5 or 6-axis operation, etc. For the purpose for which the prototype system was produced, however, a very simple ARELEM was sufficient.

The first execution module thus takes an I.L. file as its input, together with the table of canonical forms, and uses it to create an initial CLTAPE. This is in APT IV format, as this has the flexibility to allow extra (unprocessed) records to be inserted for any unrecognised I.L. commands so that a subsequent module may process them.

The APT III (and also NELAPT, EXAPT, etc.) CLTAPE format is entirely numeric; the APT IV format, however, may contain both numbers and character strings. In particular, a post-processor record begins with the actual post-processor major word, which is followed by the necessary additional details in character or numeric form as appropriate. The initial execution module therefore creates a PSEUDO post-processor command whenever it encounters an I.L. command that it does not recognise, and then copies the I.L. command to the CLTAPE as the rest of the PSEUDO record. Subsequent execution modules, if any, then take the CLTAPE as their input and look only for PSEUDO post-processor records.

3.6 The CLTAPE Editor

After the execution modules have finished with the CLTAPE it should contain only post-processor commands, together with any editing commands

such as COPY or TRACUT. The CLTAPE Editor, which is essentially the APT IV CLTAPE Editor, takes this file and produces a final CLTAPE from it.

Any editing commands will be acted upon to create altered or additional CLTAPE records, while any post-processor records will be copied to the final CLTAPE. Any PSEUDO records will give rise to diagnostic information, as will any error records put on the CLTAPE by subroutines in the ARELEM complex or other execution modules. The final CLTAPE is produced in either APT IV format, APT III format, or both, depending on the user's requirements.

The CLTAPE thus produced is then post-processed in the usual way.

3.7 The Overall Structure of the Prototype MILDAPT Processor

The prototype MILDAPT processor was designed to cater for several geometric (or definition) modules, several procedure (or action) modules, and several execution modules. In fact, the system implemented on an ICL 1906S computer contained two geometric modules - one containing the common APT/NELAPT definitions and the other containing some additional 2-dimensional geometric definitions - and only one procedure module and one motion (execution) module. Figure 3.1 shows the idealised overall structure and the interconnections between the modules; those modules which did not exist in the implemented version are shown lightly cross-hatched.

In the 1906S implementation the various modules were not separate programs (as in APT and NELAPT) but were "overlays" of the same program. Thus the input/output routines are part of the "root overlay", which is permanently resident, while the potentially large storage required is minimised by use of COMMON blocks in a similar way to that used in the APT IV Execution Complex. In particular, this use of COMMON means that the Name Table and Canonical Forms are automatically available for as long as they are required, but that the space used for the Name Table is used by the Execution module for the storage of surface details during motion processing, when the Name Table is no longer needed.

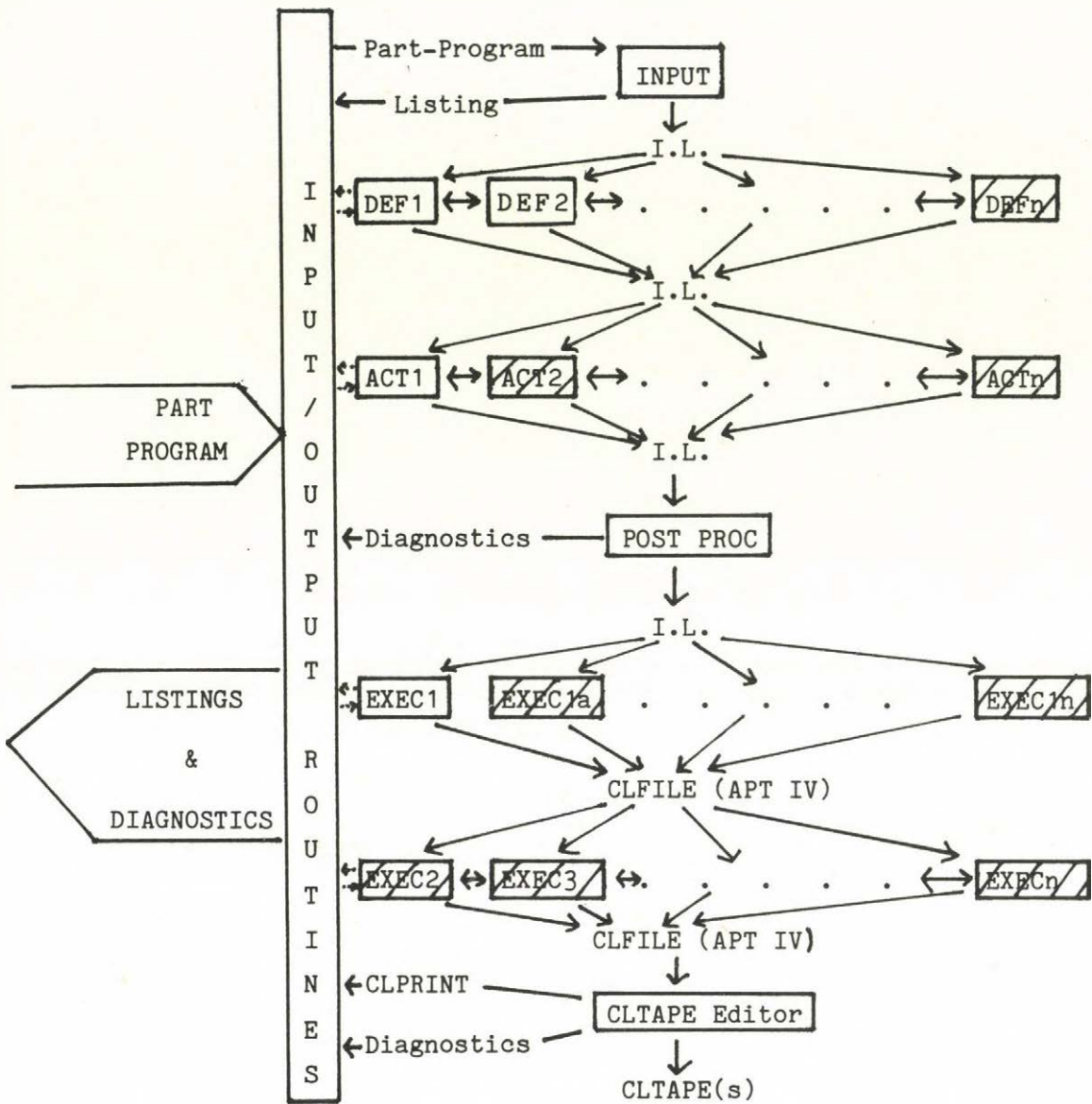


Figure 3.1 Prototype MILDAPT module structure

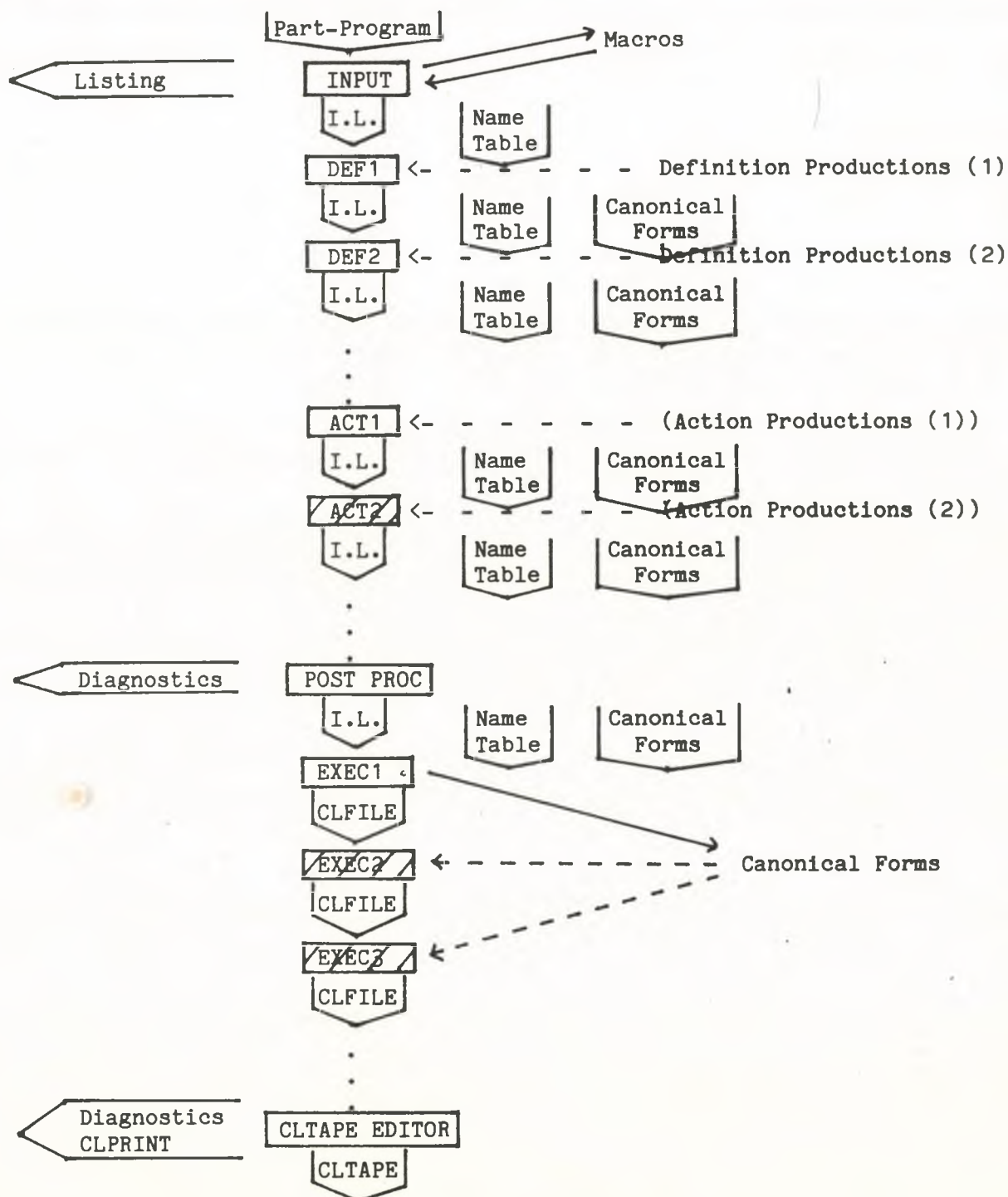


Figure 3.2 Prototype MILDAPT Information Flow

The flow of information through the prototype MILDAPT processor is shown in figure 3.2, where once again those modules not present in the 1906S implementation are shown lightly cross-hatched. Notice that, in general, the Name Table and Canonical Forms will not be required after the

first Execution module; however the canonical forms are preserved in case they are needed for some reason later on, although this should be unlikely as they are included on the CLFILE. It is also worth emphasising that, whereas the definition modules will normally require some production tables with which to analyse the input statements, this will not necessarily be the case with the action modules. Indeed, the APT IV processor (and hence also the prototype MILDAPT processor) deals with motion commands simply by inspection, as was mentioned above. Where production tables are required, however, a special loading program will be required to encode these tables in a suitable manner. In the 1906S MILDAPT system this was achieved by a modified form of the APT IV Load Complex.

3.8 Conclusions Obtained from the Prototype MILDAPT Processor

The 1906S prototype MILDAT system was, as has already been described, an experimental system based largely on code extracted from APT IV. As such it was a far from ideal system. Nevertheless it served to clarify many ideas and to identify a number of problem areas in such a modular system.

In particular, the sequential nature of the processing meant that it was possible for a geometric definition to remain unrecognised because one, or more, of the items in it would not be fully defined until a later module had processed the I.L. For example, in the prototype MILDAPT system the special (test) definition of a line joining a point to the origin was in a different module from the majority of geometric definitions. Thus the statements

```
P1 = POINT/10,20
L1 = LINE/P1
L2 = LINE/10,0,0,20
P2 = POINT/INTOF,L1,L2
```

would cause an error because the first module would process the definitions of P1 and L2, but not of P2 since (at that stage) L1 was undefined. The second module would then process the definition of L1, but P2 will remain undefined unless the first module is re-entered.

The prototype system dealt with this problem by allowing the part-programmer to specify which modules were to be loaded, and in which order (thus also avoiding the unnecessary loading of a module which finds nothing to process and simply copies the I.L. file); a simplification enabled a pre-defined sequence to be specified. However this was not a satisfactory solution on both aesthetic and practical grounds, and was a major failing of this system.

A second problem area concerned the modules themselves. Since these required (in general) some form of production table, it was necessary to provide a special program (or rather several programs) to create the necessary tables. These programs were based on the APT IV Load Complex but, nevertheless, their use required a certain amount of detailed knowledge about the processing methods used. In addition, the storage methods used, which were based on COMMON areas shared between different modules, also required the writer of a new module to have a reasonably detailed understanding of the underlying data structure for the whole system.

However this experimental system did show that it was feasible to split up a processor into several parts, each dealing with only a part of the language, although in the case of an N.C. system it was also apparent that it was only in the definition area that it was realistic to anticipate that parts of the system would be added ~~or~~ removed at will.

The prototype MILDAPT system, however, was clearly not suitable as the basis for further research, partly for the reasons outlined above and partly because it was APT-based. The longer-term intention was to develop a concept which could be applied to application-oriented languages (AOLs) in different fields. The APT IV system, and hence the prototype MILDAPT system, was so strongly N.C.-oriented that it was difficult to see how it could possibly be used as the basis of a general approach - even if it had been desirable for other reasons. It was therefore decided to scrap the prototype system, and to investigate the possibility of an approach based on parallel processing of the input statements instead of the normal sequential processing.

4. THE DISPERSED MONITOR CONCEPT

4.1 Parallel Processing, Semaphores and Monitors

The introduction of the semaphore concept by Dijkstra [Dijkstra, 1968] led to a new understanding of the inherent problems of cooperating parallel processes, and was the basis for much of the fundamental work that was subsequently carried out in this field. However, a possibly even more important step was the introduction of the concept of a **monitor** by Brinch Hansen [Brinch Hansen, 1972] and Hoare [Hoare, 1974] as a tool for the development of well-structured operating systems. A monitor can be informally defined as a collection of procedures and data which **together** control the manipulation of some resource, with the important constraint that the resource may only be accessed by means of the monitor, and that execution of the monitor procedures by various processes must be mutually exclusive in time.

The main uses of monitors in practice have been in the outer layers of operating systems to control such functions as multiple access to files [Hoare, 1974], paging [Hoare, 1973], single resource scheduling [Brinch Hansen, 1973], etc., although Lister and Sayer have described how the concept can be extended in a hierarchical fashion right down to the system nucleus [Lister and Sayer, 1977]. These examples, however, are all concerned with the same fundamental area - the control of (low level) activities **within** the operating system; as far as the author is aware, at the time when these ideas were being developed (1976/77) there had been no attempts to use the underlying monitor concepts in the very different field of application software and, in particular, in the control of parallel **user programs**, although languages such as Concurrent Pascal [Brinch Hansen, 1975, 1977] and Modula [Wirth, 1977] do provide many of the necessary tools.

The monitor concept was originally introduced as an elegant and effective tool for the control of scarce resources in environments (notably Operating Systems) in which several independent processes might be competing for those resources. In these situations the monitors are

accessible either solely by parts of the Operating System itself, or indirectly by user programs through the medium of a supervisor call or an interrupt (as, for example, in the handling of peripheral devices). The basic concept, however, appeared to be so elegant that it could well be adapted to a rather different form of use.

Let us now consider a computer system in which a number of independent user programs are operating on a common set of data with some form of loose synchronisation between them. Such a system might be a conventional data processing application (for example payroll, sales accounting, or order processing) where the synchronisation merely consists of ensuring that the various programs are run in strict sequence; or it might be an application-oriented language processor where, again, several programs will be run in strict sequence (e.g. Translate, Execute, Post-Process); or it might even be a multi-pass compiler whose separate passes are sufficiently distinct for them to be considered to be separate programs. In all these examples it is clear that the various individual programs are normally run in sequence; however there is frequently no good reason why this should necessarily be so, other than the lack of suitable tools with which to do otherwise. In the data processing situation, for example, once the input data has been sorted into an appropriate order (if necessary) the remaining passes could proceed in parallel, subject only to such obvious constraints as not allowing the output program to "get ahead" of the analysis program which is producing the results which are to be output! In an application-oriented language (AOL) processor, or a multi-pass compiler, the different programs (or passes) may well have an even greater freedom to proceed at their own pace. For example, the definition statements are frequently unrelated to most, or all, of the other definition statements, while action statements will usually only be dependent on one or two definition statements, although the order of action statements will frequently be quite rigidly defined.

One obvious problem that will arise from parallel operation of different parts of the same overall system is that of access to data (i.e. memory). In a (normal) sequential mode of processing, a program is entitled to act as though its memory area is totally sacrosanct. (In a paged system, of course, this will not actually be the case, but the Operating System will ensure that the user program is unaware of any

changes; the actual values of the program's data will never be altered other than by the program itself). However if several programs are running in parallel, and all require access to the same data (e.g. the Name Table or Canonical Forms in an APT-like system) then some form of close control must be exercised over access to this data.

A closely related, but conceptually distinct, problem concerns the synchronisation between the various user programs (or **processes**) which will, together, constitute the complete system.

4.2 The Control of Access to Memory

In the previous section we defined a monitor as a collection of procedures and data which together control the manipulation of some resource. We should therefore be able to define a monitor to control access to the common data in the memory; the problem is where this monitor should reside, and how the user programs should access it.

Let us consider a (small) program which has allocated to it a certain amount of memory additional to that required for its own purposes, and let us call this program **X**. Let us further consider a second program, which we shall call program **A**, which will read data and produce results in a conventional fashion. Finally, let us require program **A** to access and to modify some of the data stored in the extra memory allocated to program **X**. This may clearly be achieved by some simple communication system, such as that shown diagrammatically in Figure 4.1; the exact mechanism used is immaterial so long as it ensures that at some time after **A** has sent data to be stored **X** accepts it and stores it in the memory allocated to it, and that at some time after **A** has requested some data **X** extracts it from the memory and sends it to **A**, having first processed any outstanding storage requests.

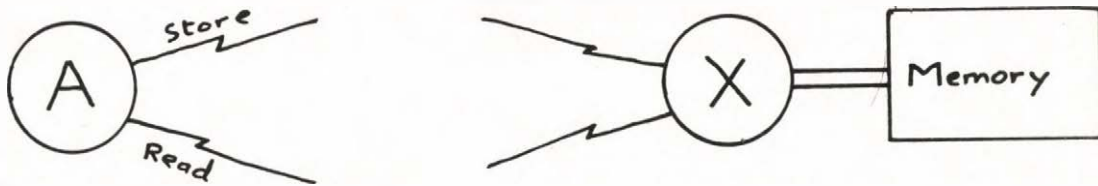


Figure 4.1 Dual access to memory

An obvious, and well-known, method is to use two message buffers - each of which is controlled by a semaphore, or similar device, which ensures that after sending a request for data A "waits" for a reply. Figure 4.2 illustrates this approach; the small program dA in the diagram represents the procedures etc. which are required to control the sending and receiving of data. It is worth pointing out that the receive buffer is shown as smaller than the send buffer because the former may only contain one message at a time, while the latter may contain several data storage requests in addition to at most one data request.

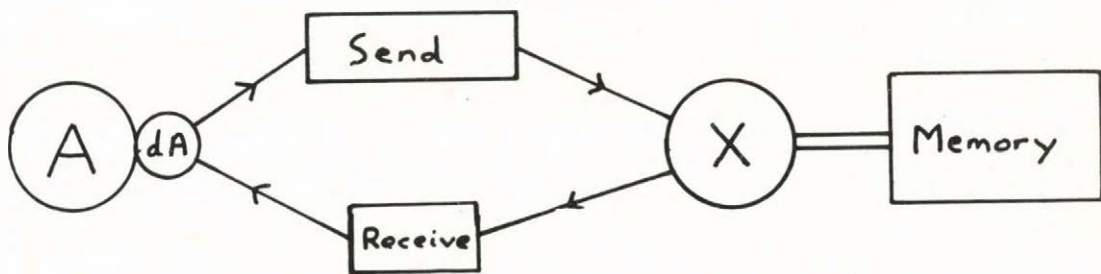


Figure 4.2 Dual memory access using buffers

Let us now postulate the existence of several additional programs, similar to program A, which we shall call program B, C, D, ... Clearly we may set up a similar mechanism for each of these programs, as shown in figure 4.3. We shall, however, ignore any inter-program synchronisation that may be required if, for example, program A may modify data which is accessed by program B, and will concentrate on the data handling; we shall then examine the synchronisation problem in the next section.

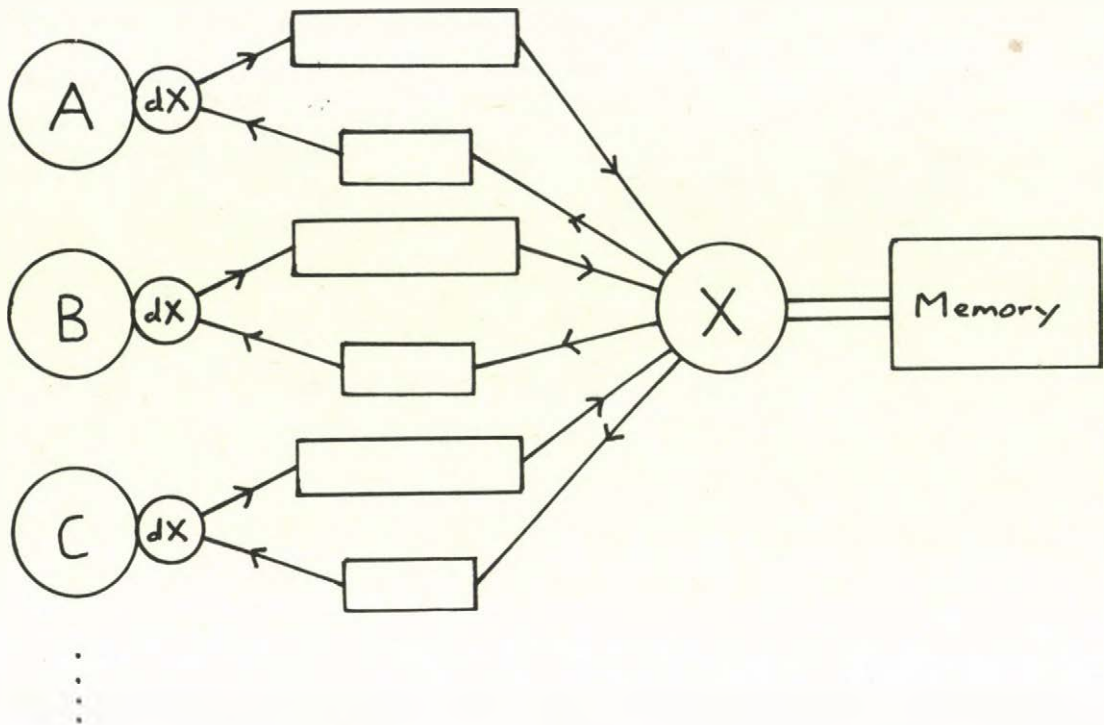


Figure 4.3 Multiple memory access using buffers

A few moments thought will show that it is not necessary to have two message buffers (as shown) for each of the programs A, B, C, ... , and that a single send buffer will suffice, as long as each message which requires a reply (e.g. a request for data) indicates from whence it was sent. As well as simplifying matters, this creates a more orthogonal structure, as every program (including program X) can proceed until it wishes to read from its input buffer and finds it empty; it must then wait until the buffer has been filled.

We therefore have a basis for synchronising the programs, and we shall examine this in more detail in the next section. Before doing so, however, we note that the "extra" program elements dA, dB, dC, etc. will contain **identical** procedures, since their sole purpose is to handle communication with program X. We may therefore rename them all as dX, resulting in the structure shown in figure 4.4.

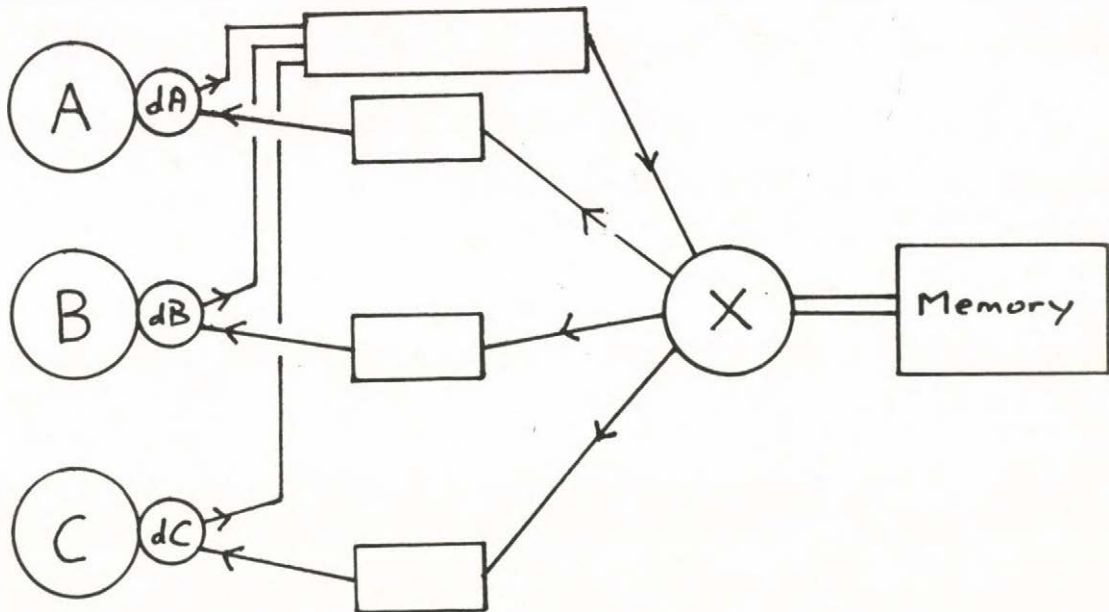


Figure 4.4 Improved multiple memory access

4.3 Synchronisation

The approach described above will enable a number of programs to access and modify a common set of data; however, unless it is possible for appropriate synchronisation to be achieved the programs may access the data in the wrong order thus, for example, causing payslips to be produced before the hours worked have been recorded!

We have seen that the basic data-handling method described above requires that programs must wait, when they expect a message, until that message is available; it follows, therefore, that program X may readily suspend any of the other programs simply by refusing to respond to a request for data. We shall use this principle as the basis for a complete synchronisation method for the whole system.

First, however, let us examine why we should wish to suspend a program, and for how long. We have already seen that in certain circumstances some programs must not "get ahead" of others; how do we define "getting ahead"? Such an expression implies some pre-determined sequence of operations, and indeed the data for most applications is deterministic - that is, it is intended to be processed in a particular order. In practice it is frequently only partially deterministic, and, for example, in a situation in which a number of geometric surfaces are to be defined the order in which the definitions are processed is irrelevant, except when one definition refers to another surface which should already have been defined. Thus we see that in many of the cases in which it is feasible to separate the processing into two or more parallel elements there will be a serial data stream of some kind which can be used to define the "order" in which the programs should be running at any time.

In the case of language processors, there is frequently another synchronising situation, which occurs at a **jump** or other interruption of the normal order of processing. A backward jump would clearly upset the simple ordering strategy already proposed as well as causing problems with the updating of data, while a forward jump has the additional problem that the destination of the jump may be unknown and must be found without allowing any of the programs to proceed beyond the jump. An obvious approach is to require programs to wait until they have all reached the jump before proceeding although, as we shall see, a less primitive solution is also possible. Finally, it may be necessary (or desirable) for one program to be able to instruct all the others to abort, or otherwise terminate processing unexpectedly due to an error.

We may deal with all these situations, as well as with others which may easily be imagined, by one small extension to the system already developed. In that system program **X** did nothing except update the common data, or extract data from it, at the request of one of the other programs. We shall now give program **X** the ability to keep a table showing the "stage" reached by every other program. This can easily be achieved by requiring that whenever one of the programs **A**, **B**, **C**, etc. wishes to read a new record from the fundamental data stream it must first ask program **X** for the record number. Program **X** can therefore suspend any program until others have reached the same point (by not replying), and can then either allow it to

continue or can alter the normal sequence (i.e. to jump) by returning an out-of-sequence record number. It is easy to see how this principle can be extended to allow one program to instruct program X to cause all other programs to abort, to reset for a new run, or to carry out some other exceptional action. The description of the MILDAPT 2 processor in section 4.5 will show how easily such synchronisation can be achieved.

4.4 Access to the Monitor

Program X, the various dX elements, and the communication buffers comprise "a collection of procedures and data which together control the manipulation of some resource" - namely, the common data. This was the first part of the informal definition of a monitor given in section 4.1 where, however, two further constraints were laid down - that the resource could only be accessed via the monitor, and that execution of the monitor procedures is mutually exclusive in time. The first of these constraints is clearly satisfied, since the common data is only directly accessible by program X; however it is clearly possible for several dX elements to be active at the same time, and we must examine this aspect in more detail.

Let us consider what happens when program A wishes to communicate with program X. First it will call the appropriate procedure within its dX element; this will place a message in the message buffer; finally, if no reply is expected it will return control to program A, otherwise it will wait for a reply.

Now let us consider program X. When it receives the message from program A it must either store some data and send no reply, or it must take some action and send a reply, or it may wish to suspend program A by not sending a reply when one is expected. In the first two cases it cannot move on to the next message in its buffer until it has completed processing the message from program A; we can therefore consider the requirement for mutual exclusion to have been met so far as program X is concerned. However in the third case the processing of the message from program A is not completed as a reply has yet to be sent; nevertheless program X must proceed to the next message.

At some subsequent point in time the conditions necessary for re-activation of program A will be satisfied. However, if a reply to the original message from program A were then to be sent two problems would arise. The first of these is that some undesirable interaction might occur between the two programs which were simultaneously "active" with respect to program X, while the second is that all the details of the original message would need to be preserved. We can avoid both of these problems by returning a special reply to program A requesting it to re-transmit the original message. This clearly avoids the need for program X to store the message, and it also eliminates the possibility of interaction since the re-transmitted message will be processed in the usual manner when it reaches the head of the queue. We may therefore consider that during its period of suspension program A is not active with respect to program X, because the original message has, in effect, been rejected and must be transmitted. The requirement for mutual exclusion is therefore satisfied for all situations.

We now have a method whereby both access to data and synchronisation can be controlled in an orderly fashion; however we must take one further step before we have a sufficiently sound system for general use, since the "monitor" will only operate correctly if a number of conventions are observed - notably those discussed above in connection with suspension and re-activation. In order to ensure that these conventions are observed we define a set of higher-level procedures operating on user-identifiable items such as, for example, PUT LINE, GET RECORD, STORE DATA, etc. These procedures will use the low-level procedures already discussed to carry out communication with program X and will be the **only** "monitor procedures" available to the user programs (A,B,C, etc.). Thus **these programs need not know of the existence of program X and the associated communication procedures**, but will merely call appropriate procedures to carry out specified activities such as input, output, and access to common data. The complete system structure may now be represented as shown in figure 4.5, where the circular form has been used to indicate that none of the user programs need have any particular significance

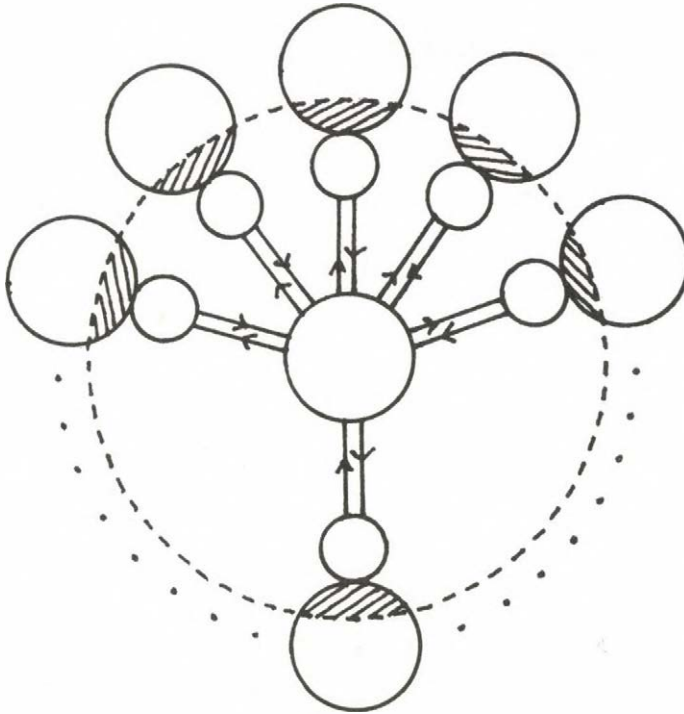


Figure 4.5 A dispersed monitor

The shaded area within each of the user programs represents the high-level monitor procedures which, alone, are accessible to the user program. The area enclosed by the dotted circle will henceforth be referred to as the dispersed monitor.

The really important feature of this concept is that because each program communicates with the rest of the system (including common data areas) only through simple, easy-to-understand, procedures it provides a very powerful tool for the provision of easily adaptable software - which was one of the primary objectives of this research. This is because program X, or the Executive module as we shall henceforth refer to it, need know little or nothing about the activities of the other programs (or modules), and they in turn need know nothing about each other.

The Executive module must, of course, know of the existence of the other modules; however, as long as there is no over-riding priority between the user modules it need know nothing about their purpose or how they achieve it. It will, of course, have a full knowledge of the common

data, as well as of a number of other matters of global interest; similarly the high-level monitor procedures will be designed to reflect the particular application both in terms of the data and of any specific synchronising philosophy. However, in neither case does this reflect a knowledge of an individual module, but merely of the overall system structure and purpose.

It is therefore possible to add or remove modules from the system at will, subject only to the requirement that the Executive module should be informed (for example, by means of initialisation parameters) of the names or other identifying characteristics of the modules to be used for a given computer run. The practical effect of this is obvious and far-reaching in its implications; in particular, it allows the testing of new facilities without in any way affecting those parts of the system which are unchanged, and it enables the system to be restructured in a dynamic fashion to suit the particular set of input data to be processed.

A further benefit which ensues is that automatic generation of user modules to process new facilities becomes relatively straight-forward, since the interface between the new module and the remainder of the system is well defined and can be built into a generator program without much difficulty. Chapter 6 will investigate this aspect of the research.

4.5 The Dispersed Monitor in Practice

The above concepts describe an abstract system that should provide the necessary characteristics for an adaptable AOL processor. It was clearly necessary to implement a simple system based on these principles in order to verify it in practice, and a skeleton processor was therefore written to run on an ICL 1906S computer, running under the GEORGE 4 Operating System [ICL, 1975].

We have already seen that statements in an AOL (and specifically in an APT-like language for NC/CAD/CAM) can be grouped into three classes:

- a) Definition statements
- b) Action statements
- c) Other statements - mainly **control** statements

The prototype MILDAPT system had already shown that it would be relatively simple to arrange for an Input module to simply pass definition or action statements to other modules without any syntax checking, and to fully analyse only those (few) remaining statements which were largely concerned with control of program flow. This Input module could therefore create an I.L. file containing a mixture of genuine I.L. and pseudo-I.L. which would be processed as appropriate by the other modules. These other modules could proceed at their own pace, subject to certain synchronising constraints, namely

- 1) No module may get ahead of the Input module (i.e. they cannot read an I.L. record before it has been produced!).
- 2) Definition and Action modules must never get ahead of the **Control** module. This is because this module, which processes statements in class (c) above, handles jumps and other transfers of control (amongst other things), and it is clearly essential that no other module should pass such a statement until it has been fully processed by the Control module.
- 3) Any module which encounters an undefined variable when it is attempting to process an otherwise acceptable statement must wait until the variable becomes defined, or until all the other modules have caught up with it. If it is still undefined then the variable must have been used before it was defined in the source part-program, and an error must be recorded.

The first two constraints are invariant, and define the broad basis for synchronisation, while the third is a dynamic factor which essentially requires a module to suspend itself according to the state of other modules of whose existence it is unaware! Furthermore, the modules themselves need not be directly aware of the constraints, since all communication with common data and the Executive is handled by the low-level monitor procedures in the modules, which are themselves only accessed via the user-oriented high-level monitor procedures referred to earlier. Nevertheless, the synchronisation of the various modules becomes a remarkably straightforward affair because the necessary tools are already part of the system.

The first two rules, above, state that the Input module must never fall behind any other modules, and that the Control module must never fall

behind any of the Definition or Action modules. There is no need for any system of semaphores (or any other synchronising device) to achieve this because the perfect mechanism already exists - namely, the stream of intermediate language commands. All that is necessary is for the Input module to record whenever it writes an I.L. record, and for the other modules to record when they read one, and all the relevant information is available. This is the central principle which governs the Executive module.

We can define four main types of communication between the modules and the Executive, namely those in which a module

- a) requires access to the common data base
- b) requires access to the intermediate language file
- c) wishes to send information to the output file(s)
- d) needs to initiate some specific synchronisation

In the initial implementation the data base is subdivided into two parts - the Name Table, which contains all the symbols used and their lexical and/or syntactic classes, and the Surface Data Table, which replaces the APT-like Canonical Forms area as the source of data concerning the various geometric surfaces. Other types of data could be added if required, but this class of communication is distinguished by the fact that, as long as the basic synchronisation rules are being obeyed, the order in which the modules happen to be running is of no consequence.

Access to the Intermediate Language file, on the other hand, is the primary synchronisation method. When the Input module writes an I.L. record it informs the Executive, which updates a table showing the **stage** (or I.L. record) reached by each module. When a module wishes to read an I.L. record it asks the Executive for the sequence number of the next record. If rules 1 and 2 are satisfied then the module is given the required sequence number and can read this record directly. This technique also deals with jumps or other interruption of the normal sequence, as the appropriate sequence number can be communicated to other modules as and when necessary. If the reading of an I.L. record would infringe the synchronisation rules then the module must be suspended until the situation has cleared. In order to fully synchronise the system, therefore, the Executive module must know about the special position of the Input and

Control modules, and also the number of other modules, so that appropriate tables can be kept. The latter information can be provided at run-time (for example, by a macro which runs the system), while the former is knowledge about the permanent parts of the system; the requirement for freely added or removed modules is not, therefore, infringed.

Access to the output files, in particular the printer, is a further problem since the information to be printed (or otherwise output) will be produced in a random order but must be output in a sequential order related to the original source program. Once again, the knowledge of the stage reached by each module can be used to decide whether such output should be output directly or stored in a buffer for output later in the correct order. Experiments have led to the adoption of a five-line buffer for each module; if a module's buffer is full then it must be suspended until some of the buffer contents have been output.

The final type of communication relates to requests from a module for it to be suspended until one or more other modules catch up with it. This may be due to an undefined variable, as discussed above, or to a jump, or to some exception condition such as the end of the program, or to an error. All that is required, however, is for the Executive to suspend the module until some condition (normally that the other modules reach the same stage) is satisfied.

The skeleton system developed to verify these principles was written in Algol 68 [van Wijngaarden et al, 1969, 1975] because this is a powerful language providing all the necessary tools for system programming [Holdsworth, 1977], and was available on the 1906S [Woodward and Bond, 1974].

The GEORGE 4 Operating System is an extremely powerful system which contains one little known feature which very greatly simplified the writing of the low-level monitor procedures. This is a concept known as a **Communication File** [ICL, 1975] which provides a means for information transfer between modules and a means for suspension and re-activation of user modules.

A communication file is a standard GEORGE character file which can be open for reading by any number of programs simultaneously and also for **appending** to (i.e. writing **after** all existing records) by any number of programs, not necessarily the same ones as are reading the file. However, if a **reader** attempts to read a record after the last one that has so far been written to the file then that program is suspended by the operating system until either a further record is appended to the file or there are no more programs remaining in the computer which are allowed to write to the file. In the first case the program is re-activated and will continue (by reading the next record); in the second case it will fail.

The low-level synchronisation and message passing is thus dealt with by the operating system, although it would not, of course, be difficult to write a suitable set of communication procedures oneself if they were not already available. The communication file has two further advantages, however.

The first of these is that, when used as described above, the file is preserved and can be listed when the processing is complete. This provides a detailed record of the way in which the various modules proceeded and interacted with each other. In a multiprocessing environment (or a simulation of one) this is a very useful diagnostic tool and was, for example, very valuable in examining the effect of different sizes of output buffers for printed output.

The second useful feature is that a program does not necessarily have to read a record from the file - it may also read it directly from the file buffer before it is written to the file (if it is quick enough). In a great many cases, therefore, a record is transferred from module to Executive, or vice-versa, without first being written to a file and then read from there.

A further feature in a production environment is that of a **destructive read**. In this mode, the reading of a record by all of those programs which are allowed to read it will cause the record to be deleted. Thus only those records which still have to be read by one or more modules will exist at any time, and at the completion of processing the whole file will have been destroyed. When considered together with the possibility of the

within-memory transfers it can be seen that in many situations the record(s) may never actually get written to the file.

Essentially, therefore, the GEORGE communication file acts as a message queuing system with the feature that any program which attempts to read beyond the end of the queue is automatically suspended (in an **idle** wait, not a **busy** one) until the queue is lengthened. One such queue (**SENDER**) is used for all messages to the Executive, while a second (**RETURNER**) is used for all the replies.

The messages in both files follow broadly the same format and consist of two parts. The first part of each message (in either direction) consists of an integer couplet in which the first integer defines the module which sent (or is to receive) the message, while the second integer defines the type of message. Let us begin by examining messages to the Executive.

There are five types of message that may be sent to the Executive module, as follows:

- 1) insertion of an item in the Name Table, or adjustment of its class code
- 2) insertion of, or request about, an item in the Surface Data tables
- 3) notification of, or a request for, an I.L.record
- 4) a message for printing
- 5) special synchronisation data

In each case the next item is an integer item (**OP**) which indicates which of, possibly, several variants of the basic forms is to follow.

Thus for type 1 messages the format is as follows:

MOD = number of sending module

TYPE = 1

OP = 0 if Name Table index is not known (i.e. this is an **insertion**),
or Name Table index if it is known.

NAME = name to be inserted in the Name Table if **OP**=0,
otherwise not relevant

CODE = class code to be inserted, or -1 if no code to be inserted

The reply will take the form:

```
MOD  = number of module which is to receive the reply (i.e. the one
       which sent the request
TYPE = 1
OP   = Name Table index
NAME = Name (i.e. the entry in Name Table)
CODE = Class Code
```

If a module wishes to insert an item in the Name Table then it uses the procedure PUT NT RECORD, which takes the following form:

```
proc put nt record = (ref int index, long bytes item, int ncode) void:
begin
  op := index; name := item; code := ncode;
  signal(nt); wait;
  index := op
end
```

where SIGNAL and WAIT are the two primitive procedures which actually deal with the inter-module communication.

Similarly, a Name Table record may be examined by use of the procedure GET NT RECORD:

```
proc get nt record = (ref int index, ref long bytes item, ref int ncode)
void:
begin
  op := index; name := item; code := -1;
  signal(nt); wait;
  index := op; item := name; ncode := code
end
```

The procedures PUT NT RECORD and GET NT RECORD are available to a user module. The procedures SIGNAL and WAIT, the global variables OP, NAME and CODE, and the constant NT (=1) are, however, invisible to the user, and are made inaccessible to him by virtue of the multi-level Algol 68-R album structure [RRE, 1976]. The module thus inserts an item in the Name Table by use of a procedure whose specification is available to him, and obtains details about any particular entry by use of another procedure. The mechanism used, and the synchronisation (if any) involved, is **totally** hidden from him.

In a similar fashion the message format for type 2 (surface data) is:

MOD = number of sending module
TYPE = 2
OP = Name Table index of the name of the surface, negated if
 the surface is being inserted in the database
SUB = subscript (if any), or zero
CLASS = class of surface, or zero if data about the surface is
 being requested
NSD = number of values in the surface definition, or zero if
 data about the surface is being requested
SDEF[1] }
: } = surface definition values, if NSD>0
SDEF[NSD] }

The reply in this case is:

MOD = module number
TYPE = 2
OP = 0 if the surface data is OK, -1 if no surface data exists
 for this surface (i.e. it is undefined)
SUB = subscript, or zero
CLASS = class of surface
NSD = number of defining values
SDEF[1] }
: } = surface definition data
SDEF[NSD] }

Once again the operation of the system is hidden from the module, which simply uses the procedures PUT SURF DATA and GET SURF DATA.

Type 3 messages are used to control the reading of the I.L. records and are, as we have already seen, the primary synchronisation method. They take the form:

MOD = sending module number
TYPE = 3
OP = 1 when the sending module is the Input module (module 1), to
 indicate that another I.L. record has been written
 = -1 from any other module as a request for permission to read an
 I.L. record

The I.L. records are written to a **direct access** file by the Input module in such a way that record *n* in the file is the *n*'th I.L. record. On receipt of the appropriate type 3 message the Executive module updates its details of the record reached by each module and checks to see if any modules were suspended awaiting an I.L. record. Any suspended modules are **freed** by being sent a type 5 (synchronisation) reply, as discussed below.

A type 3 message with a negative OP indicates that the module is ready to read a new I.L. record. The Executive first updates its record of the stage reached by that module and **frees** any modules waiting for this one to catch up; it then checks to see if this module is allowed to read another I.L. record. If it is (i.e. reading an I.L. record will not infringe any of the rules detailed earlier) then the number of the record to be read is returned:

MOD = module number

TYPE = 3

OP = number of next I.L. record, or zero if MOD=1 (i.e. Input)

The record is then read by the module using a direct access read.

If the module cannot yet be allowed to read another I.L. record then no reply is sent and, as discussed earlier, the module is suspended due to the way in which the primitive communication procedure WAIT operates. (In fact, because all replies are in a single communication file in the 1906S implementation, the module will awake briefly to read each record that is returned; any records which have a different module number are ignored however, and the module is suspended again.) Eventually the module will be allowed to proceed, and is sent a reply by the Executive. In order to avoid the necessity of the Executive keeping a detailed record of what was the last request sent by every suspended module, a **dummy** type 5 message is sent to **free** a suspended module. Because the type does not match the type of the original message the procedure GET IL RECORD resubmits its request and, this time, gets the required reply. Although adding slightly to the inter-module communication traffic, this approach dramatically simplifies the operation of the Executive, which needs only react to messages as they arrive and does not need to keep any historical details.

Type 4 messages are used to send lines of text to the Executive for printing, and take the form:

```
MOD  = sending module number
TYPE = 4
OP   = n to indicate that n line shifts should precede the printing of
       the line, or -1 to indicate a page throw before printing
LINE = the text to be printed
```

The reply is always the same unless the output buffer is full, in which case the module is suspended in the same way as described for message type 3. If the buffer is not full then the reply is:

```
MOD  = module number
TYPE = 4
OP   = 0
```

The preparation of a line of text is carried out by a set of monitor procedures (CPRINT, IPRINT, RPRINT) which allow the user program to specify where, and in what format, a character string, integer, or real number is to be printed, together with a further procedure (PUT LINE) which actually sends the message. The two numeric formatting procedures use the Algol 68 transput facilities to build up the line, as can be seen from the following:

```
proc iprint = (int pos,i,n) void:
begin
  int p:= abs pos;
  if pos<0 then clear line fi;
  outf(lp,$n(p)kn(n-1)v-$,i)
end
```

which inserts the number i in the output line starting at character position pos and occupying n characters (right justified).

Thus the user module need neither be aware of the complex inter-relationship between the modules nor even of the powerful, but complicated, Algol 68 formatting facilities, but once again only uses the **high-level** monitor procedures.

The final type of message (type 5) is concerned with special synchronisation requirements, and takes the form:

MOD = sending module number

TYPE = 5

OP = -n the module must wait until all modules have caught up, and must then go to record n. This is only sent by the Control module, and is used for jumps or other interruptions of the normal sequential processing

= 0 indicates that this module must wait until all the other modules have caught up. This would normally be because an undefined surface has been encountered, or the module wishes to write to the CLFILE

= 1 is only produced by the Input module, and indicates that the end of this part-program has been reached and that another one follows. Input must therefore wait until all other modules have caught up (i.e. finished processing this part-program) and then restart for the next part-program

= 2 is also only produced by the Input module, and indicates that the end of the last part-program has been reached. Each module will be stopped when it reaches this stage

There are two forms of reply message:

MOD = module number

TYPE = 5

OP = 0 if the module is to continue

= -1 to free a module after it has been suspended, as described for type 3 and 4 messages

There is also a special type of reply message in which the message type is negative; this simply tells a module to reset for a new part-program (-1) or to terminate itself (-2).

The skeleton system used to develop and implement these concepts consisted of five modules - Executive, Input, Control, and two (dummy) User modules which simply listed any definition or action I.L. commands, respectively [Ellis, 1979a]. As far as the 1906S computer is concerned, the Executive module is the MILDAPT processor. This module is loaded by a special macro, which also passes it the details of the other modules required. The Executive begins by initialising various data areas and communication files, and then returns control to the macro.

The macro then loads all the other modules that are required as separate, independent, programs (indeed, as separate jobs) before returning control once more to the Executive module. As each module starts to

execute it sends a special message to the Executive which consists only of its module number and a (dummy) message type; it then suspends itself until it gets a reply in the usual way.

The Executive module, on the other hand, suspends itself as soon as it is re-entered from the macro. It will be briefly awakened as each module sends its initial message, but the Executive ignores these (and suspends itself again) until all the modules which are being used have sent a message. It then replies to them all with a similar (dummy) message, thus re-activating them all. The modules can then proceed at their own pace in the manner already described.

Messages from the modules to the Executive all use the primitive procedure SIGNAL, which was seen earlier being used by the high-level procedures PUT NT RECORD and GET NT RECORD. This procedure writes the initial couplet (MOD, TYPE) to the communication file, and then uses the value of TYPE to control a **case** statement to select the other items to be written:

```
proc signal = (int st) void:
begin
  stype:= st; put(s,smess);
  case stype
  in
    put(s,(op,space,name,code)),
    put(s,(op,space,sub,class,nsd));
    if nsd>0 then put(s,sdef[1:nsd]) fi,
    put(s,op),
    put(s,(op,space,line)),
    put(s,op)
  out
    fault("invalid message type")
  esac;
  newline(s)
end
```

It should be noted that STYPE is the second element of the two-element integer array SMESS, and that OP, NAME, etc. are global variables (within the protected monitor environment) whose values have been set up as required for use by the high-level procedures.

A similar approach is used by the procedure WAIT to read records and ignore them unless the module number is correct.

It is not, of course, a coincidence that the names of these two fundamental procedures are the same as those used by Dijkstra and others as the basis for a semaphore system of control, since their synchronising role is identical; in the 1906S implementation the realisation of that function is, however, largely taken over by the Operating System through its handling of communication files. On another computer only these primitive procedures would need to be altered to handle the alternate suspension and awakening by more direct means.

We have seen, therefore, that the overall dispersed monitor concept works as intended, and that the two User modules can be added or withdrawn at will. However one aspect of the processor which we have not discussed is the storage of data. Because all the modules will require some access to the common database, notably to the Name Table and the Surface Data, this data must be stored by the Executive module. The use of a very powerful language (Algol 68), and the unusual nature of the processor (especially its adaptability) meant that the method used to store the data need not, and should not, consist of a fixed or static arrangement of data, such as is common in most AOL processors (which are frequently written in Fortran). The next chapter, therefore, discusses the approach developed to provide a flexible and adaptable data structure [Ellis, 1979b], before chapter 6 describes how this framework can be used to create a highly adaptable language processor.

5. AN ADAPTABLE DATA STORAGE METHOD

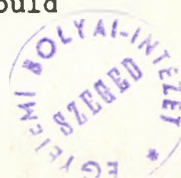
5.1 AOL Data Types and Their Storage

One of the problems with any AOL processor is that the data types it uses will usually be complex structures rather than simple numbers. Furthermore, the variety of these types and their uses means that it is highly undesirable for any arbitrary limits to be placed on the quantity of data of any one type, and yet it is clearly undesirable to reserve large amounts of unused data space. This implies some form of list-based data allocation which can expand (and contract) as required.

The particular form of processor being discussed imposes two further important constraints. The first is that in order to minimise communication traffic between modules the data structure chosen should be as concise as possible, while the second is that since the processor is, in effect, a single pass compiler the storage allocation must be carried out without any foreknowledge of the total requirement. (The processor can be considered to be a single pass processor because, although the various modules are running in parallel it is possible (and indeed likely in a medium to large sized program) for the early part of a part-program to be completely processed before the end of the same part-program has even been input.)

We shall examine the data structure which was developed to meet these requirements in the context of geometric surfaces; however the same technique is equally applicable in any other area, such as technological operations, etc.

Every geometric surface can be described by some standard representation, which will normally consist of the name of the surface (including, possibly, a subscript) and a number of real values which define its form. For example, a point may be defined by its three Cartesian coordinates, a circle by its centre and radius, a line by the coefficients of its equation, etc. There are two fundamental requirements for the representation chosen for the surface - it should be compact, and it should



be easy to use in calculations involving the surface. There is, nevertheless, frequently no uniquely desirable representation (for example, a point may equally well be defined by its polar coordinates), and as a result there are a number of different forms used in different systems, notably for the storage of complex surfaces. However, for the present it is sufficient to assume that **some** basic representation has been defined. For most of this chapter we shall use the (intuitive) forms mentioned above for points, circles and lines, since this is both readily understandable and a widely used convention; in section 5.5, however, we shall briefly show how the same technique could be used for a radically different form of surface representation.

In addition to these simple surfaces, we shall also require a means of storing information concerning composite surfaces such as patterns (consisting of an ordered set of points), free curves (which pass through an ordered set of points in a smooth fashion), and contours (consisting of an ordered set of surfaces, each of which intersects its immediate neighbours). Unlike the basic data types, these have no fixed size and will usually be of different sizes on each occasion that they are used.

The storage method chosen must, therefore, be able to store a variable number of items, each consisting of a name and a number of other items, the number of the latter being sometimes fixed and sometimes variable. The recognised way of dealing with this type of problem is by means of some form of linked data structure. This does, of course, carry an overhead, as Shave [Shave, 1975] and others have pointed out, due to the need to store the links; however in practice this is more than offset by the ability to store all surface types in one area and to eliminate the wasted space inherent in a number of arrays of fixed dimensions. It is somewhat surprising to realise that most numerical control packages, and most other geometric application programs, use fixed-size arrays in which to store the surface data. The major reason for this, of course, is that in most scientific and technological programming the only language used is Fortran (which does not support list processing); however more recently developed programming languages such as PL/I, Pascal, Algol 68, Simula and Ada do contain facilities for handling linked data structures (amongst other things), and since Algol 68 had been used to develop the initial dispersed

monitor concept into a working system it was merely necessary to decide upon the most efficient method.

5.2 A List-based Data Storage Method

The method chosen will be explained first in terms of the basic surfaces, and then extended to composite surfaces, and finally to surfaces whose type is unknown.

One of the advantages of a language such as Algol 68 is that data is **strongly typed**, and we shall take advantage of this to provide great flexibility in the storage of a wide range of different types of data. First, however, we shall define a **mode** for each of the three basic data types that we are considering:

```
mode point = struct(real x,y);  
mode line = struct(real a,b,c);  
mode circle = struct(point c, real r);
```

Note that, for simplicity, we are here restricting our surfaces to the x-y plane, and also that a circle is defined by its centre (as a **point**) and radius; this is, of course, equivalent to three **real** values but is a more natural form of representation.

We now define a new data type as a **union** of these three types, and we can then use it to build a list structure:

```
mode surface = union(point, line, circle);  
mode cform = struct(surface s, ref cform next);  
ref cform null surf = nil;  
ref cform canon:= null surf;
```

The mode **cform** can be seen to be the basic list element type, and consists of two parts - an item of mode **surface** (i.e. either a **point**, a **line** or a **circle**) and a **reference** to another item of mode **cform**; it will therefore be possible to link any number of **surface** variables together in one, composite, list. The last two declarations establish a dummy reference NULL SURF which can be used to terminate a list, and then assigns this to CANON to set up an initial (empty) list structure of that name. Woodward and Bond give an excellent description of the creation and

manipulation of lists in Algol 68 [Woodward and Bond, 1974] and it is not intended to elaborate any further on the techniques required.

The above structure makes no mention of the name of the surfaces since these will already be in the Name Table and it would be wasteful to duplicate them. The Name Table, or rather an additional table (or array) using the same indexing, can however be used to speed up the extraction (or modification) of surface data from the list. The purpose of the Name Table is to store the names of variables or other items, and to relate them to their other attributes (class codes, etc.), and its use to identify the data itself is simply an extension of this principle. This extra array will be declared as

```
[1:max]ref cform cf;
```

where [1:max] is the range of subscripts for the Name Table. The following program extracts show how simple it is to insert a new point and a new circle:

```
int i,j; point p; circle c;
```

```
  .  
  .  
  .
```

```
c
```

Assume that i is the Name Table index of a point whose coordinates are already stored in the variable p

```
c
```

```
cf[i]:= canon:= cform:= (p,canon);
```

```
c
```

Assume that j is the index of a circle c

```
c
```

```
cf[j]:= canon:= cform:= (c,canon);
```

```
  .  
  .
```

The expression (P,CANON) is a representation of an item of mode **cform** which consists of a **surface** variable (in this case a **point**) and a reference to an item of mode **cform** (i.e. CANON). Since CANON was originally set up to be the (empty) last item in the list the new **cform** item will be linked to this empty item. The global generator **cform** creates space for an item of mode **cform** and this new list item is stored there, and also assigned to the variable CANON - which therefore now points to this latest list item (the head of the list) instead of the empty list item. Finally a **reference** to CANON is assigned to the appropriate element of the array CF.

The next statement carries out a similar process, except that this time the **surface** is a **circle**, and the new list item is linked to the previous (**point**) item. The list now contains a circle, linked to a point, linked to the null item, which ends the list.

Extraction of data from this list is trivial. For example, if K is the Name Table index of the required surface then the following code is all that is required:

```
int k;  point p;  line l;  circle c;
      .
      .
      .
c
  Assume that k is the index of the surface
c
case (p,l,c)::= s of cf[k]
in
  begin
    c processing of a point stored in p  c . . . . .
  end,
  begin
    c processing of a line stored in l  c . . . . .
  end,
  begin
    c processing of a circle stored in c  c . . . . .
  end
esac
```

This uses the Algol 68 conformity clause to extract the **surface S** from the list item referred to by the pointer CF[K] and assign it to whichever of the variables on the left of the conformity clause is of a suitable mode (i.e. **point**, **line**, or **circle**). The **case** statement then branches to the appropriate processing clause, dependent upon which variable was selected.

To summarise this method of storage we can say that a global generator is used during the insertion process to create a new list item linked to the current head of the list, and that a reference to this item is stored in the array CF. During surface data extraction this array element is used in a conformity clause to obtain both the surface data **and** its type (or mode).

5.3 Composite Surface Types

One of the advantages of this approach is that it can be readily extended to composite surfaces of random size. We can illustrate this by an example, for which we shall use a **pattern** - that is, an ordered set of points. The method is simply to store this ordered set as a list, and then to link this list to the main surface data list CANON. One problem, however, is that a pattern may need to be accessed in either order, and so two pointers are required - one for each direction:

```
mode patpnt = struct(point pt, ref patpnt last,next);
ref patpnt nullpt = nil;
ref patpnt patpt:= nullpt;
patpt:= patpnt:= (p,patpt,nullpt);
mode pattern = struct(int npts, ref patpnt first);
pattern pat;
.
.
mode surface = union(point, line, circle, pattern);
```

Note that the number of points has been included as part of the mode **pattern**. This is not strictly necessary, as the end of the list is easily determined, but it is convenient for many purposes. For a contour, however, it would not be necessary to know the number of defining surfaces, and the mode could consist solely of a reference to a list (of references to surfaces).

Insertion and extraction of pattern data is now easily achieved in a similar manner to that used for basic surfaces, and the following code shows how a pattern is inserted, where, for simplicity, it is assumed that the procedure NEXT POINT delivers a reference to the next point in the pattern, the integer variable NPTS already contains the number of points in the pattern, and K is the index to the Name Table:

```
int npts;
.
.
for i to npts do
  patpt:= patpnt:= (next point,patpt,nullpt);
  next of last of patpt:= patpt;
  if i=1 then first of pat:= patpt fi
od;
npts of pat:= npts;
c k is Name Table index of pattern name c
cf[k]:= canon:= cform:= (pat,canon);
```

Note that, in this case, because there are pointers in both directions it is necessary to use a slightly more complicated method of insertion, and that it is also necessary to set up the **pattern** PAT to contain the number of points and a pointer to the list of points. The points are inserted in this list in a forward order (unlike the situation with basic surfaces) with the pointer to the next item being initially empty (i.e. referring to nullpt - the empty pattern point defined above). The statement

```
next of last of patpt:= patpt
```

inserts the forward reference as soon as it is known (i.e. when the next point has been inserted). The final point in the pattern will already refer to the end of the list, and so needs no further adjustment.

The same technique can also be used to deal with unknown, or user-defined, data types. The concept of a user-adaptable processor implies that the user may wish to define new surface data types, and it is therefore necessary for the surface data handling procedures to be able to deal with data types whose format is unknown to them!

Since all data may be assumed, at least at the lowest level, to consist of a set of real numbers, it is possible to store **any** surface data as a list of real numbers, which is linked to the main data list in exactly the same way as just described for patterns. If we refer to a user-defined data type as being of mode **other**, then we can define the necessary modes and variables as follows:

```
mode item = struct(real r, ref item last,next);
ref item null item = nil;
ref item item:= null item;
item:= item:= (0.0,item,null item);
mode other = struct(int n, ref item first);
other other type;

mode surface = union(point, line,....,other);
```

This is almost identical to the infrastructure defined for patterns, with **other** substituted for **pattern** and **item** substituted for **patpnt**. Exactly the same process is then used to insert an **other** surface, or to extract it from the overall data structure. This facility therefore allows a new surface type to be inserted extremely easily, and tested with the

relevant, new, processing procedures. Once these are fully tested it would be a relatively simple matter, if required, to make the changes to the main data handling procedures and data classes which would be necessary to incorporate this data type permanently as part of the processor.

5.4 Subscripted Surfaces

The above algorithms all reserve space for a particular surface on the first occurrence of that surface. However a subscripted surface variable presents a further problem in that the single entry in the Name Table must be used to access all the surfaces referred to by subscripted elements of the array whose name is in the Name Table. This is not, of course a new problem, and the normal solution (and in practice the only one) is to arrange for all the subscripted elements of an array to be stored consecutively.

It is not unreasonable to require the range of subscripts to be defined before their first use, and all languages known to the writer (with the partial exception of Basic) do require such an array declaration, or dimension statement. Once the range of subscripts is known it is possible to create an appropriate number of surface data list items, and then to use a list pointer to move through this **sub-list** in order to access the correct list item for a particular value of the subscript.

One way of doing this is to declare a procedure (called, say, INSERT SURFACE) which will both reserve space and insert a surface. Such a procedure could take the form shown below, in which the variable THIS CF is used as a list pointer to move through the sublist:

```
proc insert surface = (surface s, int index,sub) void:
begin
c
  If sub is positive then it is a subscript
  If it is zero then the surface is unsubscripted
  If it is negative then it is a request to reserve space for -sub
  surfaces, of which the first is s
c
ref cform this cf;
  if sub>0 then
    this cf:= cf[index];
    to sub-1 do this cf:= next of this cf od;
```



```

    cf of this cf:= s
  else
    cf[index]:= canon:= cform:= (s,canon);
    if sub<0 then
      to sub-1 do canon:= cform:= (s,canon) od
    fi
  fi
end

```

Thus, the first of N circles, all subscripted elements of the array of circles whose Name Table index is K would be inserted by the statement

```
insert surface(c,k,-n)
```

where the circle C contains the defining data. Subsequent insertion of the M'th circle would require

```
insert surface(c,k,m)
```

The subsequent extraction of the data for the I'th circle would use the following code:

```

ref cform this cf;
.
.
.
this cf:= cf[k];
to i-1 do this cf:= next of this cf od;
case (p,l,c,...)::= s of this cf
in
c
  conformity case structure as before
c
.
.
.
esac

```

5.5 The Method Applied to a Different Surface Representation

The above discussion uses a conventional, widely used, and easily understood representation of surface data. However the same technique can be used for quite different forms of data structures, should these be more appropriate. This can readily be demonstrated by means of another form of geometric surface representation which is radically different from that used above. This method is due to Sabin [Sabin, 1976], and treats all

continuous two-dimensional curves as being composed entirely of circular arcs and straight line segments. Sabin proposes a form of storage in which each arc/segment is stored as its two end points plus a **bulge factor** (which is zero for a straight line). It is not appropriate to describe the method in detail here, but it is merely necessary to point out that one great advantage of this method is that the same algorithms can be used for all surface calculations, regardless of the shape of those surfaces. Each surface, or **profile**, is represented by $3n+2$ real numbers, where n is the number of circular arc or line segment **spans** of which it is constituted. A processor which used this method, together with conventional point and pattern definitions, could be readily implemented with the data being handled in the manner already described, although since Sabin's profiles are not only bounded but also have a sense of direction it will only be necessary to traverse the lists in one direction. It is, however, then necessary to insert a profile in two stages in order that it should be in the correct order, the space used for temporary storage in the list THIS SPAN in the example below being released by the assignment

```
this span:= next of this span
```

which enables the garbage collector to re-use the current head of the list THIS SPAN.

We shall illustrate the approach required by assuming the existence of three procedures NEXT X, NEXT Y, and NEXT B which return the x , y and b (bulge factor) values for the next span, and a boolean procedure MORE SPANS which returns **true** as long as there remain some spans for the procedures NEXT X, etc. to process. (Of course, in practice these values would come out of some other part of the processing, but these fictitious procedures will serve for example.)

```
mode element = struct(real x,y,b); element elem;
mode span = struct(element e, ref span next);
mode profile = ref span;
profile null span = nil;
profile pspan,this span;
mode surface = union(point, pattern, profile);
.
.
.
c
  Insertion of a profile with Name Table index k
c
  this span:= null span;
```

```

while more spans do
  elem:= (next x,next y,next b);
  this span:= span:= (elem,this span)
od;
c
  Reverse the order
c
pspan:= null span;
while this span isnt null span do
  pspan:= span:= (e of this span,pspan);
  this span:= next of this span
od;
cf[k]:= canon:= cform:= (pspan,canon);
.
.
.
c
  Extraction of a profile with Name Table index k
c
case (p,pat,this span)::= s of cf[k]
in
c point c
  (.....),
c pattern c
  (.....),
c profile c
  while this span isnt null span do
    elem:= e of this span;
    c
      Process this span of the profile
    c
      .
      .
      .
    this span:= next of this span
  od
esac

```

5.6 Data Storage in a Dispersed Monitor Processor

As we have already seen in chapter 4, the skeleton MILDAPT system developed to verify the dispersed monitor concept required that the Executive module should look after all the surface data storage, and communicate the relevant details to other modules as and when required. The Executive module therefore has a number of modes defined for the various surface types (**real**, **point**, **pattern**, **line**, **circle**, **vector**, **plane**, and **other**) in exactly the form already described, and has a mode **surface** defined as the union of all these.

As we saw in chapter 4, a type 2 message is used to request surface data or to send details of a newly defined surface. In the case of a request for data the procedure used follows exactly the same approach as already described, with a conformity clause identifying the mode of the surface and causing a **case** statement to select the appropriate action. The surface defining values are transferred to the SDEF array, the values of NSD (the number of items in SDEF) and CLASS (the class, or type, of the surface) are set up, and then the message is returned to the requesting module. Thus, for example, the parts of the **case** statement devoted to points and circles are as follows:

```
int ....., point=4, ....., circle=11, .....
.
.
mode point = struct(real x,y,z); point p;
.
.
mode circle = struct(point c, real r); circle c;
.
.
c
  Point data extraction
c
  sdef[1]:= x of p; sdef[2]:= y of p; sdef[3]:= z of p;
  nsd:= 3; class:= point,
  .
  .
c
  Circle data extraction
c
  sdef[1]:= x of c of c; sdef[2]:= y of c of c; sdef[3]:= z of c of c;
  sdef[4]:= r of c;
  nsd:= 4; class:= circle,
  .
  .
```

Note, in particular, that although a circle is defined by its centre and radius, the centre (a **point**) must be reduced to its individual coordinates for transmission. Thus, the expression

x of c of c

first obtains **c of c**, which is a **point**, and then obtains its x-coordinate, which is the **real** value required.

The insertion of data, and the reservation of array space, is carried out by means of a procedure (INSERT SURFACE) which operates broadly along

the lines described in section 5.4, except that the Name Table index and the subscript are already available as the global variables OP and SUB (see section 4.5). The global variable CLASS is also assigned a value as a result of a type 2 message which provides surface data, and is used to determine the action to be taken:

```
if class=real then
  r:= sdef[1]; insert surface(r)
elsif class=point then
  p:= (sdef[1],sdef[2],sdef[3]); insert surface(p)
elsif ....
  .
  .
  .
elsif class=circle then
  p:= (sdef[1],sdef[2],sdef[3]);
  c:= (p,sdef[4]); insert surface(c)
elsif ....
  .
  .
  .
fi
```

This approach to data storage therefore enables a User module to access the common database without the need to be concerned with its detailed storage mechanism, or with its use by other modules. In fact the Input and Control modules, which are part of the standard system, use the GET SURF DATA and PUT SURF DATA procedures directly, but the other modules use a still higher level of monitor procedure, as discussed in the next chapter.

6. THE AUTOMATIC GENERATION OF USER MODULES

6.1 The Structure of AOL Statements

We have already examined the fundamental types of statements in an AOL, and have shown that they can be classified as

- i) Action (or procedure call) statements
- ii) Definition (or assignment) statements
- iii) Other (mainly control) statements

We have also stated that it is reasonable to assume that user modifications to an adaptable AOL processor would almost invariably concern the format of action or definition statements, or the modification of data types which will be used in such statements. The control and other, miscellaneous, statements can be assumed to be fixed.

This is the basis for the design of the dispersed monitor as a means whereby **user modules** may be added to, or removed from, a modular processor at will. The Executive, Input and Control modules will, of course, always be present and will define the overall **shape** of the language whose details are defined within the user-adaptable part of the processor.

One of the factors that distinguishes an AOL from a general purpose language is that, because it is **application** oriented, an AOL contains a large number of descriptive words and variations on a common theme. Both action and definition statements are essentially procedure calls, which return a value for subsequent assignment in the case of definition statements. These procedure calls will typically have a number of parameters, some of which may be optional. For example, a point may be defined by its coordinates (either two or three), by a circle of which it is the centre, by two lines at whose intersection it lies, by a triangle whose centroid it is, as the current position of a plotter pen or machine-tool tip, etc., and it is this fact which distinguishes this type of statement from those used in conventional general purpose programming languages.

To illustrate this further, let us consider an imaginary language which is to be used to control an automatic bakery; because it is a hypothetical language we can use it both to illustrate many of the possible advantages of AOLs, and also some of the difficulties caused by their very flexibility. In our bakery language we could define the particular mix to be used by a statement of the form

```
mix(flour,  $\alpha$ , lard,  $\beta$ , salt,  $\gamma$ , yeast,  $\delta$ , water,  $\epsilon$ )
```

for a standard white loaf, or

```
mix(raisins,  $\alpha$ , currants,  $\beta$ , flour,  $\gamma$ , .....)
```

for a fruit loaf, or even

```
mix(standard, NW)
```

to select the pre-defined mix used for standard loaves in the North West region of the country. Similarly the baking of the loaves could be initiated by a statement such as

```
bake(x, mins, at, y)
```

or simply

```
bake(x, y)
```

by omitting the keywords MINS and AT which serve no purpose other than to improve the readability of the program.

In the above example the procedures MIX and BAKE did not deliver any result, and the assumption appears to be that only one mix can be processed at a time. In general, however, at least some of the procedures will deliver results, thus enabling a greater flexibility in the use of the language. For example, the statement

```
std:= mix(flour,  $\alpha$ , .....)
```

enables a name to be attached to the type of mix so that it can subsequently be used in a statement such as

```
bake(std, x, mins, at, y, in, n)
```

which might cause the mix defined as STD to be baked for X minutes at Y° in oven number N.

This hypothetical AOL has been expressed in an Algol-like fashion;

however this is purely for illustration. Most existing AOLs do not use this type of written representation, but use a variety of different forms and, for example, an APT-like version would have statements of the form

```
STD = MIX/FLOUR,  $\alpha$ , LARD,  $\beta$ , .....  
BAKE/STD, X, MINS, AT, Y, IN, N
```

This is only a lexical difference, and is dealt with by the Input module; both forms would produce identical I.L. commands for processing by the rest of the system.

6.2 Flexible Syntax Definition

The examples given above of a baking AOL illustrated most of the areas where flexibility is required in defining an AOL. For example, in the MIX procedure only some subset of a large set of possible operand pairs would normally be used, but almost any conceivable combination would, and should, be permissible. Similarly, in the BAKE procedure it was possible to omit the keywords and rely on the order of the parameters for their interpretation. While not explicitly stated, common sense implies that in the MIX procedure call the pairs of parameters should be allowed to appear in any order, since each pair contains a keyword identifying the meaning of the second item.

The most common way of specifying the syntax of a language is by means of BNF productions [Backus et al, 1960a,b], or some similar system which defines a legal statement as a string of symbols. This does not easily allow for the flexibility inherent in an AOL, however, and some other system must be used which succinctly and unambiguously defines the various legal alternatives. (An excellent example of the problem is shown in the Standard APT definition [ANSI, 1977], which is almost totally unreadable!)

Graphs are a natural method of specifying an abstract syntactic structure, and as we shall see, they can be used very effectively to guide the syntactic analysis of a program statement. However, the definition of that syntax must also be provided in a form which can easily be presented to the computer, as well as being understood by the user. A sensible starting point would seem to be the way in which such language statements

are presented to the user in, for example, the programming manuals that accompany an AOL system. For example APT (and subsequently such APT-like languages as NELAPT, EXAPT, IFAPT, etc.) devised a means of defining the valid syntax of a statement which consisted essentially of a written form of the statement with keywords in upper case and variables/constants in lower case or underlined; alternatives were normally indicated by being bracketed together in a column, while optional items were enclosed in square brackets. Thus the following definitions all specify one of the ways of defining a circle:

- i) $\underline{C8} = \text{CIRCLE}/\left\{\begin{array}{c} \text{CENTRE} \\ \text{CENTER} \end{array}\right\}, \underline{\text{PTA}}, \left\{\begin{array}{c} \text{LARGE} \\ \text{SMALL} \end{array}\right\}, \text{TANTO}, \underline{\text{CIRA}}$
- ii) $\text{CIRCLE}/\text{CENTER}, \underline{\text{point}}, \left\{\begin{array}{c} \text{LARGE} \\ \text{SMALL} \end{array}\right\}, \text{TANTO}, \underline{\text{circle}}$
- iii) $\text{C3} = \text{CIRCLE}/\text{CENTER}, \text{P1}, \left\{\begin{array}{c} \text{SMALL} \\ \text{LARGE} \end{array}\right\}, \text{TANTO}, \text{C2}$
- iv) $\text{CIRCLE}/\text{CENTER}, \underline{\text{symbol for a point at circle center}}, \$$
 $\left\{\begin{array}{c} \text{LARGE} \\ \text{SMALL} \end{array}\right\}, \text{TANTO}, \underline{\text{symbol for a tangent circle}}$

They are all easily understandable, and come from the 4-50/4-70 APT Reference Manual [Ellis, 1968], the IFAPT Reference Manual [ADEPA, 1970], the 2CL Part-programming Reference Manual [NEL, 1969], and the APT Dictionary [IITRI, 1967].

One way of utilising this ease of understanding on the part of the user, therefore, would be to specify the syntax to the computer in essentially the same fashion, for example

$\text{CIRCLE}/\{\text{CENTRE}, \text{CENTER}\}, \text{point}, \{\text{LARGE}, \text{SMALL}\}, \text{TANTO}, \text{circle}$

and to use various types of bracketing to indicate that items were to be alternatives {one,two,three} , or optional [opt]. One problem with this approach is that there are several ways in which we may wish to qualify the information presented in the definition and there are not enough different types of brackets! Indeed, many computers will not even accept all those already used, such as {}. Furthermore, the use of brackets is less natural to a computer system than the use of **operators**. János [János, 1977] used prefix operators in this way, and gives an example of a pattern definition which may contain an arbitrary number of points given either by name or by their coordinates, and in which the optional modifier RANDOM may be

inserted either before or after the list of points:

```
PATTERN/$(?RANDOM=-164,*!(POINT,(S,S,S))):440,650
```

The four prefix operators used here are

- ? the following item is optional
- * the following parameter must appear one or more times
- ! exactly one of the following parameters must appear
- \$ the following parameters may appear in any order

Ignoring the code numbers in the above definition which make the whole statement rather difficult to read, the definition states that the keyword PATTERN is followed by / and then by two items in either order. The first of these is the optional keyword RANDOM. The second is one or more occurrences of either a point or three scalar values.

As can be seen from the above example, parentheses are used as necessary, and operators may be nested.

An extension of this concept can be used to provide both an easy to understand form of definition (without the need for any confusing code numbers) and a basis for syntax analysis. Five operators are used, as follows:

- ? the following item is optional
- & the following item may appear any number of times (but at least once)
- # exactly one of the following set of items must appear
- ! the following set of items may appear in any order
- @ the following item represents a variable of the type specified

Thus the BAKE procedure referred to in the last section could be defined as follows:

```
bake,@mixture,@real,?mins,?at,@real,?in,@real
```

although, once again, it should be emphasised that this defines only the **syntax** of the statement; its lexical representation is quite separate and will have been dealt with during the **lexical analysis** phase (by the Input module in the case under consideration). Thus any of the following statements could satisfy the above specification:

```
bake(std,50,mins,at,425,in,5)
bake[std: 50: 425: 5]
BAKE/STD,50,MINS,425,IN,5
BAKE    STD    50      MINS    AT      425.0    IN      5
BAKE    STD    50              425.0      5
etc.
```

Similarly the `mix` procedure could be defined in a number of ways, of which the following is, perhaps, the most comprehensive:

```
mix,#((standard,#(LON,SE,MID,NW,SCOT)),
!(?(flour,@real),?(salt,@real), ....., (yeast,@real)))
```

This defines the statement as consisting of the word `MIX` followed by one of (a) the word `STANDARD` followed by one of the keywords `LON`, `SE`, `MID`, `NW` or `SCOT`, corresponding to the regional standard mix, or (b) any number of the following pairs, in any order: `FLOUR` and a number, `SALT` and a number,, `YEAST` and a number, with the proviso that some, such as the yeast operand pair, must be always present, while others are optional.

This form of syntax definition, therefore, allows the syntax of an AOL statement to be presented in an unambiguous and yet easy-to-understand manner. It is now necessary to examine how such a definition may be used to analyse an incoming language statement.

6.3 A List-Directed Syntax Analyser

Methods of syntax analysis have been a fruitful area for research for many years and will, no doubt, continue to be so; however in the restricted class of languages with which we are concerned, and with the added requirement that we shall require the syntax analyser to be generated directly from a specification of the language, the position is relatively straightforward. In the general case, methods such as Floyd productions [Floyd, 1961], operator precedence [Floyd, 1963], and the multitude of other methods which succeeded them (both bottom-up and top-down) have many advantages, although a number do suffer from the disadvantage, in this case, of the code reflecting to some extent the underlying language specification. For the recognition of simple procedure statements in the

intermediate form already discussed, a relatively simple top-down approach seems most feasible, so that the input statement is processed by the syntax analyser to produce a single output - namely the identity of the valid format which it matched, or an indication that it did not match any valid format.

In order to provide for the greatest generality, and remembering that the objective is to generate the complete syntax analyser from the language definition, it would clearly be desirable for the analyser to be completely table-driven, in the sense that the code is completely independent of the language, apart from the assumption that it will be represented with the output from the lexical analysis phase in the form of a linear string. A method which completely satisfies these requirements is one based upon the well established list-directed parser, such as that described by Foster [Foster, 1970], and we shall see that this method lends itself very well to the special operators already introduced.

First, however, consider the trivial case of a statement whose format is defined, using the terminology introduced above, as a simple ordered list of items:

a, b, c, d,, n

We may represent this by a simple list, as shown in figure 6.1



Figure 6.1 An ordered list

A language composed entirely of statements of this type can then be parsed in an almost trivial fashion by an algorithm such as the following which, it hardly needs adding, is presented in Algol 68:


```
proc check = (ref list list) int:
begin
  int n:= 0;  bool more:= true;
  while more do
    if terminal(hd(list)) then
      if hd(list)=item then
        more:= (item#rterm);  n plus 1;
        item:= input
      else
        more:= false;  n:= -1
      fi
    fi;
    list:= tl(list);  more:= more and (list isnt empty)
  od
end
```

The algorithm assumes the existence of a suitable data type **list**, and procedures **HD** and **TL** which return the head and tail, respectively, of a list element, as well as a procedure **TERMINAL** which returns **true** if its argument is terminal (i.e. not a reference to another list element). The procedure **INPUT** is assumed to provide the next item in the input string, while the global variable **ITEM** is assumed to already contain the first item on entry to the procedure. The variable **RTERM** corresponds to the end of statement marker, and always appears as the head of the last element in the **syntax.tree**. The procedure **CHECK** therefore returns the number of items which were matched, or -1 if no match was made, or if the input ran out before the end of the list, or vice-versa.

By a relatively minor enhancement we can now cater for a syntax tree which contains alternatives such as

a,#(b1,b2,b3),c,,n

Figure 6.2 shows the revised list structure, and it can be seen that the three alternatives have been drawn as three branches whose common root is linked to the cell originally occupied by the single item **B**.

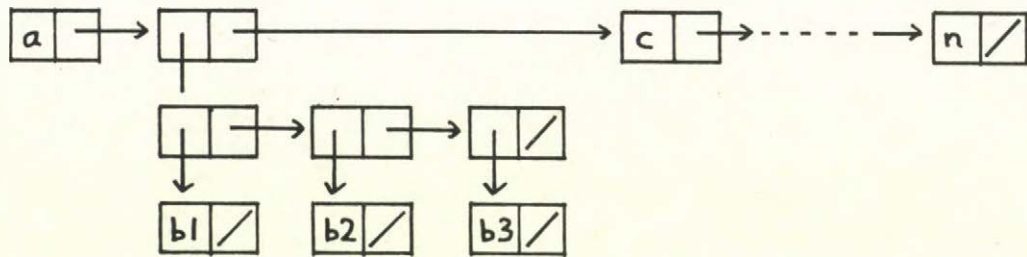


Figure 6.2 An ordered list with alternatives

The alteration to our algorithm necessary to cater for this consists merely of **dropping down** a level in the tree, and then attempting to match the input item with each alternative in turn. If we assume that the alternatives will always be single items, as in figure 6.2, then this may be achieved quite simply by adding the following **else** clause before the second **fi**:

```
else
  ref list l1,l2; bool match:= false;
  l1:= hd(list);
  while l1 isnt empty do
    l2:= hd(l1);
    if terminal(hd(l2)) then
      if hd(l2)=item then
        match:= true; l1:= empty
      else
        l1:= tl(l1)
      fi
    else
      l1:= tl(l1)
    fi
  od;
  if match then
    n plus 1; item:= input
  else
    n:= -1; more:= false
  fi
```

In practice, however, we would not wish to restrict ourselves in this way, but would call the original procedure recursively to examine the

alternatives. This requires slightly more thought concerning the philosophy to be adopted, as it is clearly necessary to examine all the alternatives and not just blindly accept the first which matches, since it is theoretically possible for it to be a subset of another list which will also match. We shall come back to this problem later.

Once we accept the recursive nature of the processing then it is clear that the alternatives **b1**, **b2** and **b3** need not be single items, nor even simple sequences of items, but may be more complex sub-trees, as shown in figure 6.3, where any of the following input sequences will be acceptable:

a, b, c, e, l,, n
a, b, d, e, l,, n
a, f, g, l,, n
a, h, k, l,, n
a, i, j, k, l,, n

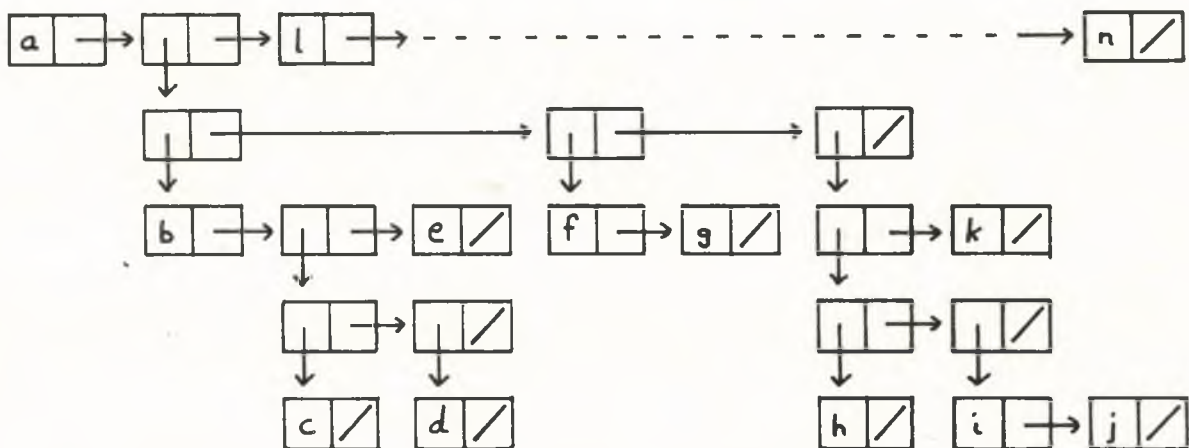


Figure 6.3 a, #((b, #((c, d), e), (f, g), (#(h, (i, j)), k)), l,, n

The next step is to extend the algorithm so that a special null item (shown as ? in the list diagrams) may be provided as an alternative. In this case, if no match is made between the input item and one of the alternatives then the algorithm does not register a failure to match, but continues as though one had been made, except that the input stream pointer is left unchanged. This, therefore, allows for an optional item, and the sequence

a, b, ?c, d,, n

would lead to the tree structure shown in figure 6.4. Once again, the recursive nature of the processing means that the item **c** can be replaced by a sub-tree of any required complexity.

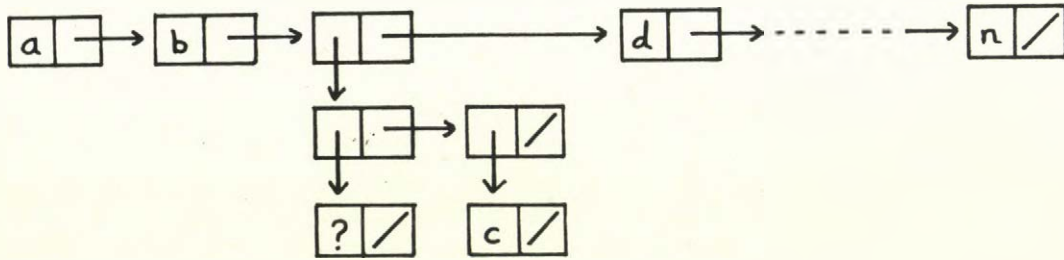


Figure 6.4 An ordered list with an optional item

A corollary to this last enhancement is to add a special **recursive** list element (shown as &) which indicates that processing is to be as for the null case, except that if a match is made then the input pointer is moved forward, but the list pointer is not, thus allowing the alternative to & to be repeated any number of times. The sequence

a, b, &c, d,, n

is thus represented by the tree structure shown in figure 6.5.

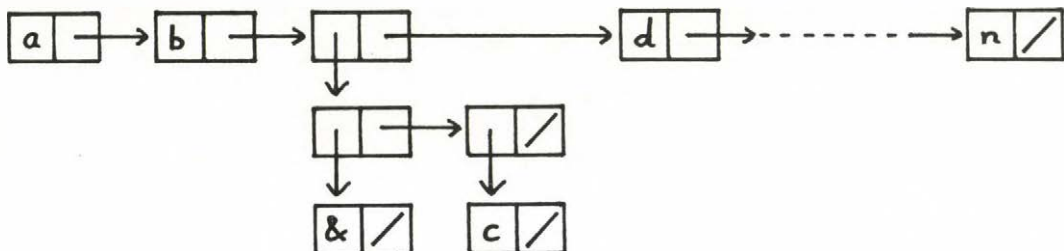


Figure 6.5 An ordered list with a recursive item

At this point we must give further consideration to the problem of choosing which of two alternatives to accept when the situation arises. In a perfect world, of course, the language definition, that is the syntax tree, would not contain any ambiguities, and if the language definition were being drawn up by the writer of the syntax analyser (or **parser**) he would attempt to ensure that there were none. However we are considering a

system which will allow the language definition to be specified by the end user (who may be assumed to have a certain amount of common sense, but who is not, and should not need to be, familiar with the inner workings of the processor) and it is quite possible for a degree of ambiguity to creep in; in some cases it can be very difficult to avoid. For example, consider the following definition

$$a, \#((b, c), (b, c, d)), \#((d, e), d), f$$

This is clearly ambiguous, and will apparently allow any of the following forms of statement:

$$\begin{aligned} a, b, c, d, e, f & \quad (1) \\ a, b, c, d, d, e, f & \quad (2) \\ a, b, c, d, f & \quad (3) \\ a, b, c, d, d, f & \quad (4) \end{aligned}$$

This is not a new problem, and considerable research has been carried out into ways (such as left factoring and the removal of left recursion) in which the syntax of a language may be expressed or transformed in order to eliminate such problems [Aho and Ullman, 1977] [Gries, 1971]. For example, the above definition could be written as

a, b, c, ?d, d, ?e, f

which, although equally ambiguous, is far more obviously so! On the other hand, the equivalent syntax

a, b, c, d, ?d, ?e, f

is no longer ambiguous, and will lead to the perfectly acceptable syntax tree shown in figure 6.7.

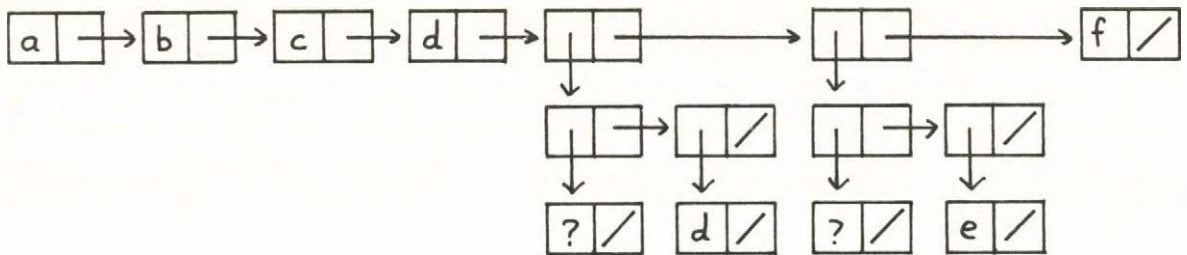


Figure 6.7 Ambiguity eliminated

If the language specification does contain ambiguities then it is still possible to avoid any backtracking by arranging to always look at the next item (or the next several items) before accepting a particular syntactic fragment. However a parser written in this way must know how many items it must look ahead (i.e. if the language is an LR(k) language then the value of k must be known). In our case, however, we are relying on the syntax definition being produced by a non-specialist, and the whole system has been deliberately designed to allow him to express himself freely. There are therefore only two realistic options.

The first of these is to assume that the language will contain no ambiguities, in which case we can use the classic **recursive descent** technique to parse every statement.

Alternatively, we must assume that the language does (or could) contain ambiguities, but that the parsing algorithms cannot possibly know how to resolve them. In this case the only solution is to backtrack and examine all the possible variations until one is found which matches the complete input sequence. For example, if the sequence

a, b, c, d, d, f

were presented to the parser with the (original) syntax tree shown in figure 6.6 and the philosophy of **first match** were being used, then the subsequence

a, b, c

would match the first alternative, and processing would continue. At the next stage the single item

d

would match with the second alternative. However this leaves the final substring

d, f

which will not match with the remainder of the tree, which is expecting the single item f. It would therefore be necessary to backtrack all the way to the first branch in the tree (in this case) and try again. This time the sequence

a, b, c, d

will be matched, followed by

d

and finally

f

and all is well. However the overhead imposed by backtracking is very large, and in this situation is totally unacceptable.

In connection with a similar problem János remarked that "we are deeply convinced that the correct form of man-machine cooperation requires not only machine but human intelligence too" [János, 1977]. The present author would fully agree with this sentiment, and proposes that in this context the problem of ambiguity should be left to the human to resolve. As long as the person creating the syntax definition is aware of the problem then he can take the necessary steps to deal with it either by simplifying and/or re-defining his statement structure, as illustrated above, or by adding an extra keyword so that, for example, the earlier definition becomes

a, #((K, b, c), (b, c, d)), #((d, e), d), f

which is completely unambiguous. One of the main reasons for keywords (or **modifiers**) in AOLs is to resolve such ambiguities, and the nature of an AOL therefore lends itself to this treatment. Nevertheless, in practice this is not a major problem due to the fact that the classes of statements that are being defined are essentially procedure calls.

We have now seen how to represent three of the four special operators introduced earlier. However the remaining one (which allows a set of items to appear in any order) requires a slightly different treatment, as the nature of the operation is directly opposed to the implicit ordering of the list structure. We can, nevertheless, represent it in the syntax tree by a variation on the method used for the other operators, using a special item in the syntax tree (shown as !), as shown in figure 6.8, which represents the sequence

a, !(b1, b2,, bN), c

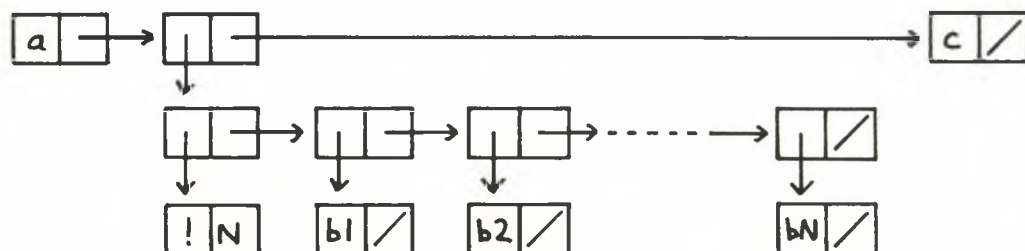


Figure 6.8 A list with unordered items

The most obvious difference between this tree and the earlier examples is that the tail of the special list element does not contain an empty reference, as in the other cases, but contains instead the number of items in the randomly ordered set. The processing is similar to that used in the other case, but with one major difference. When a sub-tree is entered which begins with the special **random** list element the whole sub-tree is copied to form a new tree which is then compared with the input sequence in the usual way. If a match is made, however, this new list is modified to remove the matching section and the process is repeated, the original count of items in the definition being used to control this loop. The input items may, therefore, be in any order, as this steadily diminishing tree always offers all those items which have not yet appeared in the input string as possible alternatives.

A further, very important, feature of this algorithm is that a record is kept of the order in which the items were matched; this can then be used to re-order the input sequence to that specified in the syntax tree, thus considerably simplifying the subsequent processing, which can assume that items will always be available in a pre-determined order, regardless of the order in which the user actually typed them.

For example, in APT (and all APT-like NC languages) one form of definition of a circle is by its centre and radius. This could be expressed as

```
CIRCLE/CENTER, @POINT, RADIUS, @REAL
```

We could very easily extend this basic definition to allow both the American and English spellings of **centre**, the reverse order for the variable data (i.e. radius before centre), and the optional omission of the keywords (since the use of defined modes for the various surface types means that the processor can tell what type of surface a variable represents). This would simply require the definition

```
CIRCLE/!((?#(CENTRE, CENTER), @POINT), (?RADIUS, @REAL))
```

which would result in the syntax tree shown in figure 6.9, where the symbol **—|** represents the end of the statement (or **right terminator**) and the areas enclosed in dotted lines represent the nested elements as follows:

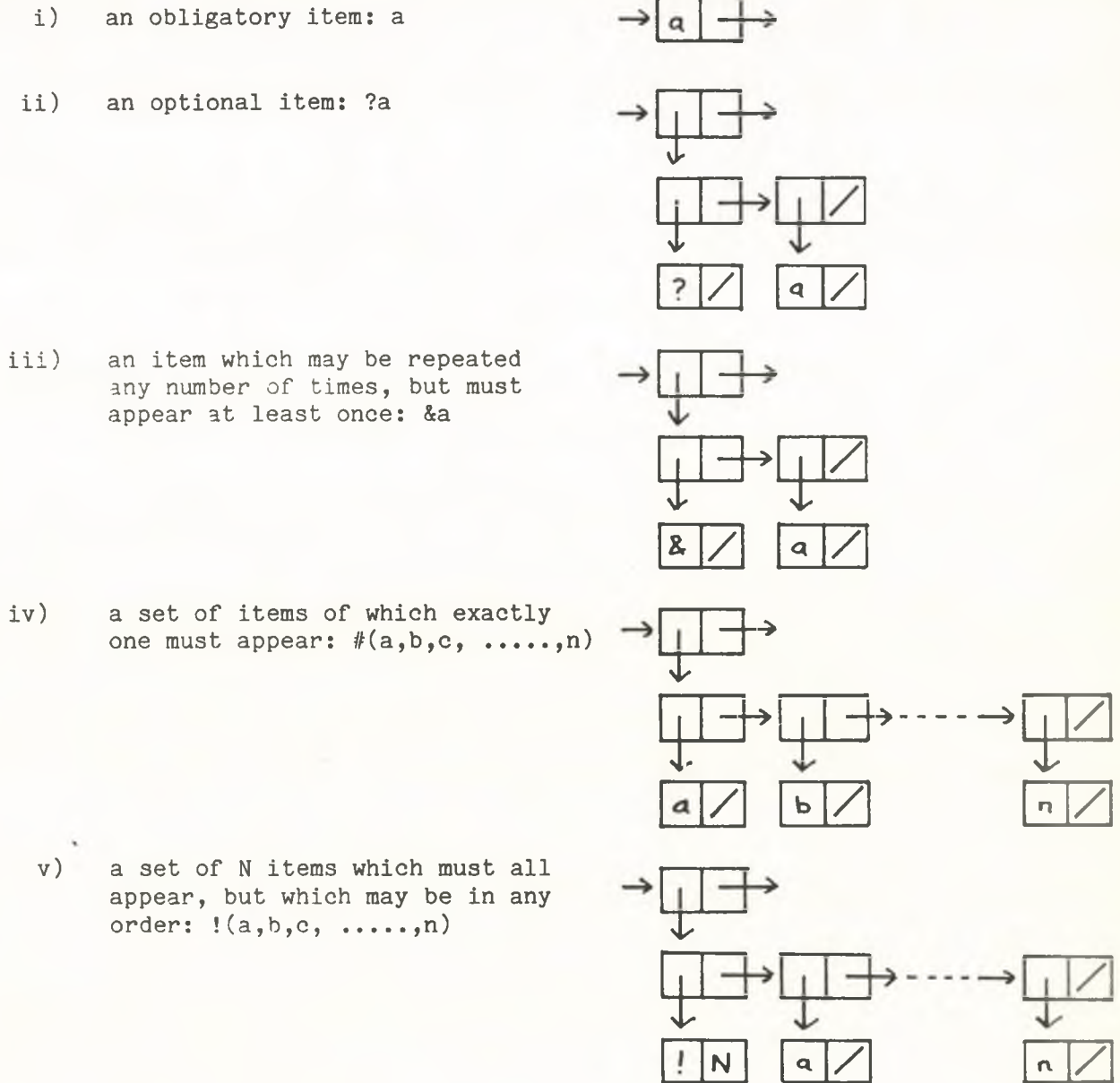
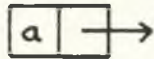


Figure 6.10 The basic tree-building blocks

6.4 Generation of a List-Directed Syntax Analyser

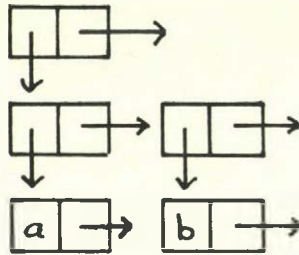
The syntax analysis of a programming language whose syntax is defined in the way described above is a particularly simple and elegant example of recursive descent parsing. The syntax tree itself will consist of some combination of the five **building blocks** shown in figure 6.10, and the comparison of an input sequence of items against such a tree requires only three basic procedures.

The first of these procedures (CHECK) examines a list element of the form



and either compares the value of *a* with the next input item, or calls a further procedure (INSPECT) if *a* is a pointer.

The procedure INSPECT therefore examines a list structure of the form



and then uses either CHECK or the third procedure (RANDOM) to process the lower levels (i.e. *a*, *b*, etc.), or alternatively uses itself recursively to go down to another level of the structure.

The procedure RANDOM is used to deal with trees of type (v) in figure 6.10, which allows items to appear in any order. Essentially, this procedure copies that part of the tree, and then uses CHECK or INSPECT to process the alternatives. As a match is made the tree is **pruned** so that fewer alternatives remain. If a complete match is made then the order in which the items appeared in the original tree is used to re-order the input items.

Since these procedures are all completely independent of the input language, the operation of the parser itself is totally determined by the syntax tree. The problem of generating the syntax analyser therefore reduces to the problem of generating the syntax tree. The standard syntax analysis procedures can then parse the input language in conjunction with the tree in an automaton-like manner so as to produce a simple **yes** or **no** for any input statement.

We have already seen that, for the class of languages with which we are concerned, the input, and hence the syntax tree, can be considered as a string of items with all punctuation and formatting removed. It follows, therefore, that the specification of the language will take the same form,

and the only problem is distinguishing between a keyword (i.e. a character string) and a reference to a particular type of variable. In section 6.2 we introduced the idea that a variable type should be preceded by an @ operator for precisely this reason, and we can therefore use it in format definitions in order to create appropriate syntax trees, such as that shown earlier in figure 6.9, or the tree in figure 6.11 which represents the pattern definition specified by

PATERN/?RANDOM,&#(@POINT,@PATTERN,(@REAL,@REAL,@REAL))

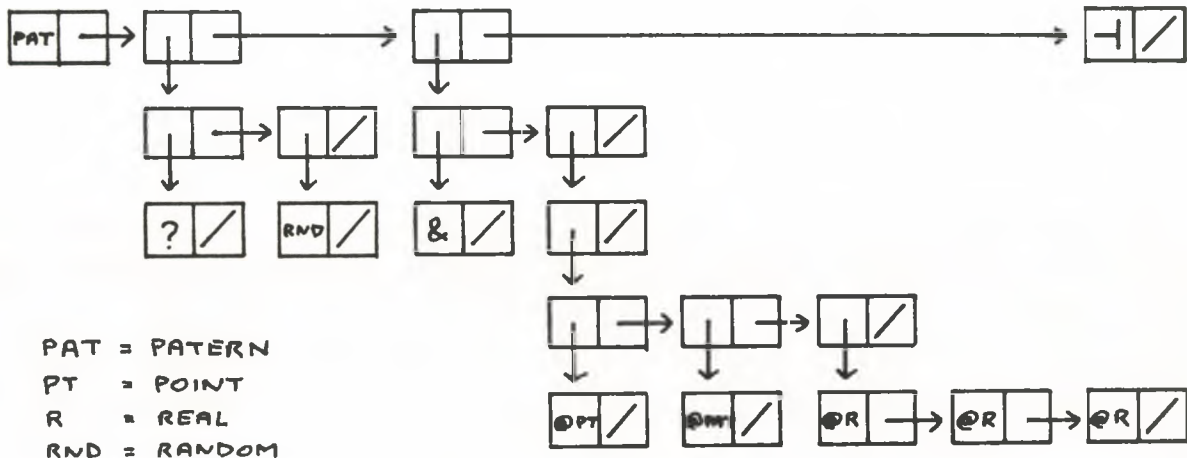


Figure 6.11 Syntax tree for a PATTERN definition

The final element in this tree is a special element whose head (|/) matches the special **right terminator** which the lexical analyser places at the end of every input statement; its purpose is to ensure that the whole input string is matched against the complete syntax tree.

The creation of such a tree from the specification is extremely simple, since each operator in the specification gives rise to a unique tree-building block. One aspect that does require further examination, however, is the representation of the various items in the tree. We can see that there are three distinct types of **heads** for list elements - pointers to other list elements, special operators and basic syntax items - while the **tails** are, with the one exception already noted, always pointers to other list elements (or **empty**, which amounts to the same thing). Since

the operators will normally be represented by integers, and the pointers (references) will require the same physical storage space, it seems logical to ensure that the syntax items should require no more space than these. There are two possibilities for keywords - namely an integer code or a pointer (reference) to the Name Table - while consideration of how to deal with variable **types** implies the use of an appropriate coding system.

In fact, a suitable coding system is already available in the main processor in the form of the various class codes and we may, therefore, use these in the generator as well. The symbol tables of both the generator and the translator are therefore loaded with the identical set of keywords, and their classes, together with the classes of the different variable types, and these are used in both the syntax tree and the input string produced by the lexical analyser for the syntax analyser. A quite minor extension of this principle allows the generator to accept keywords and variable types which have not been pre-loaded and pass this information to the translator.

The only remaining aspect of the generation process is the creation of a link between the syntax analyser and the execution phase, since the syntax tree merely enables the syntax analyser to accept or reject an input statement. The complete language specification may be thought of as either an array of trees (a forest?), each of which corresponds to one type of language statement, or as a single tree, each of whose main branches does likewise. In either case the result of parsing must be some form of computed branch to the appropriate code in the execution phase. One possibility is to provide this code to the generator in the form of a procedure which follows the definition of the corresponding statement format. A similar, though more flexible, approach is preferred by the author whereby the syntactic definition is prefaced by the name of a procedure to which control will be passed if the definition is matched with the input stream. This approach allows the procedures to be written in different languages (if applicable) or to call each other in a way which would be difficult to implement in any other approach. For convenience, within the generator definition program such procedure names are preceded by a \$ to distinguish them from keywords.

We may now use these principles to define an extended form of the bakery control language introduced earlier:

```
$type = recipe,!((#(flour,wheatmeal),@real),(salt,@real),
    (yeast,@real),?(lard,@real),?(currants,@real),
    ?(raisins,@real),?(dates,@real),(water,@real));
$std = recipe,standard,#(LON,SE,MID,NW,SCOT);
$mix = mix,@real,?lbs,@breadtype;
$knead = knead,@mixture,@real,?mins,?(wait,@real,?mins);
$divide = divide,@mixture,#(@real,&(@real,at,@real));
$prove = prove,@loaves,@real,?mins;
$cook = bake,@loaves,?#(glaze,(garnish,#(poppy,caraway))),
    @real,?mins,?at,@real,?in,@real;
finish;
```

We note that this definition uses four variable types (**breadtype**, **mixture**, **loaves** and **real**) and will require the provision of seven special procedures. It is quite straightforward apart from two points. The first of these is that the first procedure definition (\$TYPE) sets up a recipe of items which may appear in any order. The quantity of one type of flour must be specified, as must the quantity of salt, yeast and water; the other possible ingredients are all optional. Because there are such a wide range of possibilities the appropriate keywords are, of course, essential. The second point concerns the last definition (\$TYPE), where the **loaves** variable may, optionally, be followed by either the keyword GLAZE or by the keyword GARNISH, which is itself followed by either POPPY or CARAWAY. It is also worth noting that where the order cannot be altered it is usually possible to omit the keywords.

The above syntax definition program would produce a set of syntax trees which would enable a suitable language processor to correctly analyse and execute the following program:

```
BREAD = RECIPE/FLOUR,48,LARD,1.5,SALT,1,YEAST,1,WATER,30
DOUGH = MIX/30,LBS,BREAD
KNEAD/DOUGH,10,MINS
SUNPRIDE = DIVIDE/DOUGH,20,AT,1,20,AT,0.5
PROVE/SUNPRIDE,45,MINS
BAKE/SUNPRIDE,GARNISH,POPPY,40,MINS,AT,425,IN,5
```

Any, or all, of the underlined words could be omitted if required. The syntax analyser will recognise these statements and will call the appropriate procedures to execute the operations necessary to produce 20 one pound loaves and 20 half pound loaves - all garnished with poppy seed.

One point that should be emphasised is that there is no difference as far as the syntax tree is concerned between a definition statement and an action statement. In the above example program there are three definition statements and three action statements, and it follows that the three procedures TYPE, MIX and DIVIDE (and also STD which was not used in this program) will deliver a result which can then be assigned to an appropriate variable; the remaining procedures will not deliver any result. The form of the input statement will determine whether the Input module (i.e. the lexical analyser in the general case) will provide a DEFINE intermediate language command or an EXEC one, and the analysis procedure (e.g. COOK) should know what to expect and whether it should deliver a result. Once again, we are assuming a degree of human intelligence!

6.5 Generating a Module for a Dispersed Monitor Processor

The form of the syntax tree discussed above, and the procedures for generation and analysis were developed while the author was working at the Computer and Automation Institute of the Hungarian Academy of Sciences in Budapest. Although the algorithms discussed have been presented in Algol 68 this language was not available at the time, and all the development work was carried out in the systems programming language GESAL [Gerhardt, 1977] on a TPA-70 minicomputer.

GESAL contains most of those features of Algol 68 which were used by the skeleton MILDAPT processor described earlier, and it was a relatively simple task to produce a variant of the Input module which produced an I.L. file in (approximately) the same form as the 1906S version. The TPA-70 does not support multiprogramming and so it was not possible to reproduce the full system; however a form of sequential processing was all that was necessary for the development of the module generator.

The last section showed how a syntax tree could be created for use by a standard set of parsing procedures. The module generator (written in GESAL) takes the language definition as already described, together with the keywords, classes, etc. which are to be loaded into its symbol table, and creates the syntax tree(s), details of any additional keywords and/or variable types which were not pre-loaded, and the linking details for the

user-written procedures. The final **user module** is created by combining these elements with a **basic module** which contains all the parsing procedures, declarations, etc.

The TPA-70 system, therefore, consisted of a generator program and a partial processor consisting of an Input module and several generated User modules which operated in sequence. There was no Control module.

Translation from GESAL to Algol 68 is, however, a simple task, and the subsequent production of a full modular system caused no problems. It is this system that we shall now describe.

The generator program accepts a **module definition program** consisting of three parts - a preamble, a set of definitions, and a terminator. The preamble simply consists of the word **MODULE**, followed by the name which is to be given to the module, a **/**, and then either **DEFINE** or **EXEC** to indicate that this module contains only definition formats or only action formats. The restriction of a module to either definition or action statements is not, theoretically, necessary but simplifies the generation of the resulting module; the nature of the particular application also means that such a division is unlikely to cause any problems. The preamble is followed by the format definitions, which all take the form

```
$name = format;
```

where **format** is a format specification in the form already described, and **name** is the name of the user-supplied procedure to which control will be passed if the format matches an input statement. The input to the generator is terminated by a **FINISH** statement.

The generator program will load its symbol table from the same initial loading file as is used by the AOL processor itself. It then uses the algorithms already described in order to create appropriate syntax trees for the various format definitions. The final output is an Algol 68-R program which essentially consists of the declarations necessary to set up an array of trees (one for each format definition), an array of (dummy) procedures, and the main program loop which calls various standard procedures to set up the tables, parse any I.L. commands of the correct type, and take appropriate action when a match is made. Figure 6.12 shows

a typical (test) input to the generator program, which creates a module capable of processing six different types of geometric definition.

MILDAPT GENERATOR ON 15/03/79 AT 17.41.12

```
MODULE TEST01/DEFINE;
$PNT01 = POINT/@REAL,@REAL,?@REAL;
$CIR01 = CIRCLE/!((?#(CENTRE,CENTER),@POINT),(?RADIUS,@REAL));
$LIN01 = LINE/@POINT,#(LEFT,RIGHT),?TANTO,@CIRCLE;
$PAT01 = PATTERN/?RANDOM,&#(@POINT,@PATTERN);
$PNT02 = POINT/?INTOF,@LINE,@LINE;
$PNT03 = POINT/?#(CENTRE,CENTER),@CIRCLE;
$LIN02 = LINE/!(@POINT,(#(PERPTO,PARLEL),@LINE));
FINISH;
```

Figure 6.12 A module definition program

The following extract from the program generated by the definition program in figure 6.12 shows the basic structure of the generated user modules:

MILDAPT TEST01 MODULE **with** MILDAPT USER PROGRAMS **from** MA-ALBUM

begin

c This module only processes DEFINE intermediate language records c

.
.
.

```
[1:7] proc void action:= (PNT01,
                          CIR01,
                          LIN01,
                          PAT01,
                          PNT02,
                          PNT03,
                          LIN02);
```

new part:

initialise;

while more **do**

get il record(ilbuf);

if reply>0 **then**

if opcode=qseqnce **then** accept; seqno:= ilbuf[4]

elsif opcode=qtype **then**

another:= **true**;

for i **to** 7 **while** another **do**

s:= syntax[i];

if parse(s) **then**

accept; set class(ilbuf[6],surf);

action[i]; another:= **false**

fi

od

```
        elsif opcode=qfini then
            accept; more:= false
        fi
    fi;
    more:= more and reply>0
od;
end of part:
while reply>0 do get il record(ilbuf) od;
if reply=0 then
    reset; goto new part
else
    disconnect
fi
end
finish
```

In order to create the user module the generated program must be compiled together with the relevant user-written procedures (e.g. PNT01, CIR01, etc. in the above example) and the library of standard **monitor procedures**.

In principle, these user procedures may be written in any language as long as the overall system will allow them to be incorporated with the standard and generated (Algol 68) code. Unfortunately, Algol 68-R does not allow procedures to be written in any other language and so, with the 1906S implementation, the action procedures must use Algol 68. This is an aspect of the particular implementation, however, and does not reflect any such restriction in the basic concept.

The user-written action procedures will, of course, need to access the various parts of the I.L. record, as well as the data to which it refers and any other relevant data; it may also need to carry out some input or output. These activities are carried out by use of a special set of high-level monitor procedures such as

exists(s)	which returns the value true if the string S appears as a parameter in the I.L. record, otherwise false .
next param(n)	which fetches the surface data details for the variable referred to by the n'th parameter. If this is undefined then the module will be suspended, as described earlier. If n is zero then the next parameter will be fetched.
store(I,X)	sends the surface data stored in the array X for storage as the surface whose Name Table index is I.

set class(I,K) changes the class of the item whose Name Table index is I to class K. The current class is obtained from the Executive and checked to see if the change is allowed. If it is then the new details are sent to the Executive, otherwise an error message is produced.

error(n) causes error number n to be printed on the output file,

fail(n) causes error number n to be printed on the output file, terminates processing of this module, and instructs Executive to terminate all other modules!

These procedures use the ordinary monitor procedures discussed in chapter 4 such as GET SURF DATA, PUT NT RECORD, etc. to actually organise communication with the Executive module, but the use of such **very high level procedures** shields the user from the need to know anything about such communication. All the user needs to know is the structure of the data he wishes to use (but not how it is stored) and the actions that he wishes to initiate. When an appropriate input statement (or rather the corresponding I.L.) is recognised then the specified user procedure will be called into play to perform the necessary action. As long as the procedure handles all access to common data, files, input/output, etc. by use of these high level procedures then there is no limit on the complexity, or otherwise, of the actions taken by the procedure. In particular, it may use other user-supplied procedures, and thus several similar statements may have action procedures which call further (common) procedures to carry out common tasks.

7. CONCLUSIONS AND FUTURE DEVELOPMENTS

7.1 A Dispersed Processor for CAD/CAM

The work described in chapters 3 to 6 was primarily concerned with the development of techniques which would enable NC (and subsequently CAD/CAM) processors to be produced in a way which would enable them to be easily modified to suit particular requirements. The dispersed monitor concept, together with a user-oriented module generation system, achieves this objective.

The implementation of these concepts on an ICL 1906S computer has provided a practical realisation of these concepts [Ellis and Janos, 1979], and has enabled the production of several simple processors accepting different variants of APT-like languages.

The dispersed monitor concept assumes that any required number of modules can be run in parallel with one another. Unless the computer system has a large number of CPUs, utilising some form of parallel architecture then the modules are not truly parallel, of course; however, a multiprogramming system will give a good approximation to parallel execution of several programs, and this feature can be utilised (as on the 1906S) to give **pseudo-parallel** operation.

The principles utilised in the 1906S implementation (which is known as MILDAPT 2) do not depend upon the use of such a large computer (other than in the use of the GEORGE communication files to avoid the need to explicitly synchronise different processes), and any computer capable of running several programs simultaneously could simulate parallel operation, just as is done in MILDAPT 2. Furthermore, the use of Algol 68 was a result of its availability and the fact that it is (in the author's opinion) perhaps the best programming language yet produced. However the processor could be written in other languages, as was indicated by the use of GESAL for the development of the module generator and the parsing algorithms. In particular, most modern Pascal compilers contain extensions to the strict definition of the language which would make the use of Pascal

perfectly possible, and a Pascal-based processor for a minicomputer would be a more useful system than the Algol 68 version for a main-frame computer, especially if it allowed the user-written action procedures to be written in any language of the user's choice.

It is important to be clear about the objectives of the dispersed monitor concept, as applied to a CAD/CAM language processor. It is to allow the user to adapt the standard system to suit his own needs, or to produce several, related, processors. It is not designed to produce systems with very large numbers of modules.

A standard system would therefore consist of Executive, Input and Control modules, plus module definition programs for various "standard" user modules, the associated action procedures, the various libraries of monitor procedures, and the Generator program. There would also be utility programs to create the initial Name Table loading information and any other system initialisation details. If the user wished to create a sub-system (perhaps with only a limited range of surface types, or with only 2 axis motion, for example) then he would simply edit the appropriate module definition programs to delete the unwanted formats, and also would omit the corresponding procedures.

A simple alteration to the system might need little or no alteration to anything other than the module definition. For example, if the standard system contained the definition

```
$CIR07 = CIRCLE/CENTER,@POINT,RADIUS,@REAL
```

then the extension to allow English spelling would merely require the definition to be changed to

```
$CIR07 = CIRCLE/$(CENTRE,CENTER),@POINT,RADIUS,@REAL
```

A further extension will allow the (minor) modifier words to be omitted:

```
$CIR07 = CIRCLE/?$(CENTRE,CENTER),@POINT,?RADIUS.@REAL
```

This will only require a change to the definition module as long as the action procedure has been written in such a way as to ignore the minor words, which serve only as a form of punctuation and/or clarification in

this case. This is achieved by use of the procedure call

```
next param(0)
```

which will obtain the **next** item in the I.L. record, which can then be ignored (in this example) if it is not a variable. (If the actual parameter corresponding to @POINT or @REAL was not defined when the statement was being analysed by Input it will have been assigned an **implied variable** class; the parsing algorithm will then have caused the module to be suspended until the actual class can be determined).

Finally, if the action procedure has been properly written, it will even be possible to allow the order of parameters to be varied without altering the action procedure, as the parsing algorithm will always re-order them into the order given in the definition, e.g.

```
$CIR07 = CIRCLE/!((?#(CENTRE,CENTER),@POINT),(?RADIUS,@REAL))
```

In the latter two cases the action procedure will need to be altered only if it accesses the required parameters in the original statement directly, by means of procedure calls of the form

```
next param(4)
```

which is usually bad practice, or if it checks the modifiers, which is totally unnecessary in this case, as the parsing algorithm has already checked them, and they provide no additional information.

On the other hand, if a user wished to define completely new surface definitions or types of action to be taken, then he must clearly provide not only extra module definition statements, but also new procedures to implement these new facilities. In most cases these procedures will be complete in themselves, but in some circumstances they could utilise other (standard) procedures. For example, a new circle definition might be

```
$XCIR1 = CIRCLE/?#(CENTRE,CENTER),@REAL,@REAL,@REAL,RADIUS,@REAL
```

The action procedure XCIR1 could then, by suitable manipulation of the I.L. record, use PNT01 (see figure 6.12) to define a (temporary) point, and then use CIR07 to define the circle.

This particular example also illustrates very clearly the potential problems of ambiguity, which were discussed in section 6.3. If we were now to make the RADIUS keyword optional, as well as CENTRE (or CENTER), by means of the definition

```
$XCIR1 = CIRCLE/?#(CENTRE,CENTER),@REAL,@REAL,?@REAL,?RADIUS,@REAL
```

then a statement such as

```
C1 = CIRCLE/1.0,2.5,1.2
```

would not be recognised! The left-factoring solution which was discussed in section 6.3 cannot be used because of the possible presence of the keyword RADIUS. In this case the only solution is to have **two** definitions; the first of these has already been presented and demands the presence of the keyword RADIUS, therefore causing no ambiguity, while the second omits the keyword RADIUS and uses left factoring to avoid ambiguity:

```
$XCIR2 = CIRCLE/?#(CENTRE,CENTER),@REAL,@REAL,@REAL,?@REAL
```

The procedure XCIR2 can then take the appropriate steps in a very similar way to XCIR1.

Once the action procedures (if any) have been written they are combined with the generated code and parse tree to produce a new module. A particular combination of such modules, together with the Executive, Input and Control modules, constitutes a particular **processor**. In the MILDAPT 2 system, the **macro** which runs the system specifies which modules are to be used; this specification could either be in terms of the module numbers (as in MILDAPT 2) or by use of keywords which define a particular set of modules. Thus the exact form of the processor may be varied from run to run to reflect the nature of the part, or even to test a new algorithm for an existing language statement.

It has been suggested [János,1978] that another approach would be for the Executive to initially start only itself and the Input and Control modules, and for the Input module to have information regarding which language constructions require which modules. A trial modification was made to the MILDAPT 2 system to test the validity of this approach [Ellis and János, 1979], which can be used to produce a **dynamic processor** which adjusts itself automatically to suit the requirements of the part-program being processed. However this approach has a number of disadvantages.

The most obvious disadvantage is that, because the Input module must be able to initiate the loading of particular user modules it must know which modules actually exist - thus destroying one of the fundamental principles of the design, namely that modules should be able to be added or removed at will with no effect on the rest of the system.

A related problem is that some means must be devised for the Input module to be able to use the information in a statement (whose syntax is **not** being examined) in order to decide which module to load. In practice, the only realistic solution is to examine the major word (i.e. the procedure name in the idealised form of the language used in the theoretical discussion of the concept) and then load **all** modules which contain any language definition including that word. Even this produces a significant overhead for the Input module.

A corollary to this is that it becomes impossible to process the same language statement in two different modules, only one of which will be required in a particular run (e.g. in order to test a new algorithm).

A dynamic loading system is therefore likely to consist of a large number of small modules, with a consequent heavy communication load. This compares with the static loading system which contains only a small number of (quite large) modules, with a correspondingly higher degree of efficiency. There may be some situations in which a dynamic loading system is useful, but in general it will be preferable to use the dispersed monitor concept in the manner for which it was designed - to create the means whereby a user could produce several related processors to suit his own particular needs.

7.2 Conclusions

The research described in this dissertation has concentrated on three main areas - namely the design of a language processor which can easily be "tailored" to suit a user's particular requirements, the development of techniques to enable such a user to easily and automatically create additional "modules" for such a processor, and the creation of a form of data-storage suitable for use with such a flexible system.

With regard to the first of these areas it is interesting to note that the Ferranti Cetec CAM-X system provides a facility for user extensions in a manner strongly reminiscent of the monitor procedures of MILDAPT by means of a set of procedures known as GLUE (Graphical Language User Extension), which are the only directly user-accessible parts of the system [Hope, 1983a]. The structure of the CAM-X system is also similar to that of MILDAPT in its general concept as it consists of a **Supervisor** (which is all the user normally sees, as with the MILDAPT Executive) together with a number of separate systems (e.g. 2-D design and drafting, 3-D solid modelling, finite element mesh generation, graphical NC, etc.) which are used under the control of the Supervisor, which also controls the flow of data around the system. Unlike MILDAPT, however, the CAM-X system is only parallel conceptually, and not operationally. Nevertheless the broad concept of CAM-X confirms the desirability of providing facilities in a CAD/CAM processor for the user to adapt it to his own particular needs.

A much closer relationship is apparent with the MINDS system (Minicomputer NC Programming and Design System) which was developed at the Computer and Automation Institute of the Hungarian Academy of Sciences [Lukács, 1981]. This incorporates a problem-oriented language analyser and control system (PLACSY) which analyses language definitions expressed in a very similar way to those in MILDAPT in order to generate a language processor based on a number of **functional modules** [János and Lukács, 1981] [Andor et al, 1982]. This similarity is not suprising since it was this common aim which led to the author first visiting Budapest in 1976, following the Prolamat'76 conference at which both János [János, 1977] and the author [Ellis, 1977] presented papers describing their early work in this area.

The PLACSY language definitions are specified in a manner very similar to that described in Chapter 6, and are used to generate **system tables** which specify the syntax of the language (c.f. the generated syntax trees of MILDAPT). These system tables are then used by the language analyser of the PLACSY system to analyse an input program and produce an internal representation of the program; this is then processed by the PLACSY **Control Unit** in conjunction with a number of **functional modules**. The whole system is written in GESAL [Gerhardt, 1977], as are the procedures which constitute the functional modules.

The language-driven processor produced by PLACSY is a sequential system which involves several passes (or levels of processing) to activate all the required functional modules, and thus operates in a very similar fashion to the experimental (and incomplete) version of MILDAPT used by the author in Budapest in 1977/78 (see section 6.5) - which was also written in GESAL. PLACSY, however, is designed for use on small computers in a way which MILDAPT was not.

The existence of several small MILDAPT processors at the University of Sheffield [Ellis, 1981], and the use of the PLACSY and MINDS systems on a much larger scale in Budapest [Andor et al, 1982] [Kovács and Turai, 1983] is proof, if proof were needed, that the concept of user-defined languages and language-driven processors is both viable and useful. The adoption of a similar facility within the purely commercial CAM-X system (and its popularity with its users [Hope, 1983b]) is proof that such systems are also commercially desirable.

The dispersed monitor is a more difficult concept to assess, however. It was originally conceived in the context of a large main-frame computer and has been shown to provide considerable flexibility in the design of the language processor(s), and to enable an existing processor to be easily modified, extended, or otherwise "customised". However, it is a technique less well suited to a small minicomputer of the kind now frequently used for CAD/CAM applications as it requires a number of programs to run in parallel (or pseudo-parallel by use of the computer's normal multi-programming facilities) and imposes a not insignificant overhead. Nevertheless, the principle of parallelism is a very important one, especially with the growth of computers based on multi-processor architecture and the advent of array processors containing several hundred (parallel) CPUs. New languages such as Modula [Wirth, 1977], Ada [DoD, 1980] [Barnes, 1982] and Edison [Brinch Hansen, 1981] have been designed with the concepts of parallelism firmly in mind, and it seems inevitable that future systems will endeavour to identify possible parallel operations whenever possible. The experience gained with the dispersed monitor approach to language processing will undoubtedly be valuable in this new, future, era - not least because it has demonstrated some of the benefits to be obtained even with pseudo-parallel operation. Furthermore, the recognition of the fact that programs need not be obeyed in sequence, even

if it is more convenient to write them that way, could have important implications in future language design.

In this context it is important to emphasise that "language" is merely a convenient word for identifying the means whereby a human designer communicates with a computer. A graphical CAD system can be considered as a language in this sense, and it is in areas such as this that parallel analysis and processing of the designer's various specifications and requirements becomes more realistic (and more challenging). This is a major area for future research.

The third aspect of the research mentioned above is concerned with the development of suitable data-structures for flexible user-generated language-processors. The approach described in chapter 5 is notable for its use of lists and trees to create a single, unified, form of storage. This is by no means the only possible approach, but it does have a number of advantages, as already described. One of its major advantages, however, is that it is a natural form of data structure to a program designer using a modern language such as Algol 68, Gesal, Pascal, etc., especially when the underlying program concepts are also based on lists and trees. In his very interesting tutorial paper Hunt [Hunt, 1982] shows how the programming languages used by "systems engineers" not only determine the ultimate forms of solutions to problems but may also shape the way in which they think about those problems. Thus the use of a powerful and logically consistent language leads naturally to a powerful and logically consistent form of data structure - and one which also provides considerable flexibility, as was illustrated in chapter 5.

The research described in this dissertation has therefore been carried out in several different areas of computer science in the context of the user's view of a (CAD/CAM) computer system, and with the intention of improving his ease of use of that computer system. In this the author has been following a well-trodden path, as is evidenced by the following quotations from the handout provided by Ross for his lecture on "Design of Special Language for Machine Tool Programming" [Ross, 1957a], which was referred to in the opening section of this dissertation:

"The primary purpose of a language is to provide a means for a human to express himself in a given problem area.

"The language also serves another very important function which is not as easily recognised. It helps the human to organize his thinking about the problem area.

"A language will usually be in a constant state of growth and revision.

"Since the language is likely to be in a continual state of flux and improvement, **it should be relatively easy to make the corresponding changes in the translation programs.**"

Hopefully, this research has made some small contribution to this 25-year old ambition.

REFERENCES AND BIBLIOGRAPHY

- Adams, C.W. and Laning, J.H. **The M.I.T. system of automatic coding: Comprehensive, Summer Session, and Algebraic.** In [ONR, 1954], pp. 40-68, 1954.
- ADEPA. **IFAPT Reference Manual.** Association pour le Developpement du système unifié de Programmation Automatique, Hainaut, France, 1970
- Aho, A.V. and Ullman, J.D. **Principles of Compiler Design.** Addison-Wesley, Reading, Massachusetts, 1977.
- Am.Mach. **APT - The language that commands machine tools.** American Machinist, 9 March 1959 (pp. 106-107).
- Am.Mach. **APT program offered to all industry.** American Machinist/Metalworking Manufacturing, 25 December 1961 (page 66).
- Andor, L., Gaál, B., Lukács, G., Nádor, L., Turai, I., and Várady, T. General Purpose Software Components of Minicomputer Based CAD-Systems. IFIP W.G. 5.2 Conference on CAD Frameworks - preprints, Røros, Norway, 1982.
- ANSI. **American National Standard Programming Language APT** (ANSI X3.37-1974). American National Standards Institute, New York, 1974.
- ANSI. **American National Standard Programming Language PL/I** (ANSI X3.53-1976). American National Standards Institute, New York, 1976.
- ANSI. **American National Standard Programming Language APT** (ANSI X3.37-1977). American National Standards Institute, New York, 1977.
- ANSI. **American National Standard Programming Language FORTRAN** (ANSI X3.9-1978). American National Standards Institute, New York, 1978.
- Ash, R., Broadwin, E., Della Ville, V., Katz, C., Greene, M., Jenny, A., and Yu, L. **Preliminary Manual for MATH-MATIC and ARITH-MATIC Systems (for Algebraic Translation and Compilation for UNIVAC I and II).** Remington Rand UNIVAC, Philadelphia, Pa., 1957.
- Backus, John. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In **Proc. International Conference on Information Processing**, pp. 125-132, UNESCO, Paris, 1959.
- Backus, John. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Problems. *Comm. ACM* 21(8), pp. 613-641, August 1978(a).

- Backus, John. The History of FORTRAN I, II, and III. **ACM Sigplan History of Programming Languages Conference - Preprints**, ACM Sigplan Notices 13(8), pp. 165-180, August 1978(b).
- Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P. (ed.), Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., and Woodger, M. Report on the algorithmic language ALGOL 60. Num.Math. 2, pp. 106-136, 1960(a).
- Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P. (ed.), Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., and Woodger, M. Report on the algorithmic language ALGOL 60. Comm. ACM 3(5), pp. 299-314, May 1960(b).
- Backus, J.W., Beeber, R.J., Best, S., Goldberg, R., Haibt, L.M., Herrick, H.L., Nelson, R.A., Sayre, D., Sheridan, P.B., Stern, H., Ziller, I., Hughes, R.A., and Nutt, R. The FORTRAN automatic coding system. In **Proc. Western Joint Computer Conference**, 1957.
- Backus, J.W. and Herrick, H. IBM Speedcoding and other automatic-programming systems. In [ONR, 1954], pp. 106-113, 1954.
- Barnes, J.G.P. **Programming in Ada**. Addison-Wesley, London, 1882.
- Bates, Edgar A. Automatic Programming for Numerically Controlled Machine Tools - APT III. In **Computer Applications - 1961** (eds. Robert S. Hollitch and Benjamin Mittman), pp. 140-156, Macmillan, New York, 1962.
- Beech, David. Modularity of Computer Languages. Software - Practice and Experience 12(9), pp. 929-958, September 1982.
- Blake, P.L. (ed.) **Advanced Manufacturing Technology**. (Proc. PROLAMAT'79), North-Holland, Amsterdam, 1980.
- Böhm, C. Calculatrices digitales: Du déchiffrement de formules logico-mathématiques par le machine même dans la conception du programme. Ann. Matematica Pura Appl. 37(4), pp. 175-217, 1954.
- Brebner, Elliot J. and Wenker, Jerome. **An Overview of N/C Programming Languages Yesterday and Tomorrow**. ASTME Technical Paper MS69-716, American Society of Tool and Manufacturing Engineers, Dearborn, Michigan, 1969.
- Brinch Hansen, P. Structured multiprogramming. Comm. ACM 15(7), pp. 574-578, July 1972.
- Brinch Hansen, P. **Operating System Principles**. Prentice-Hall, Englewood Cliffs, N.J., 1973.
- Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1(2), June 1975.

- Brinch Hansen, P. **The Architecture of Concurrent Programs.** Prentice-Hall, Englewood Cliffs, N.J., 1977.
- Brinch Hansen, P. Edison - a Multiprocessor Language. *Software - Practice and Experience* 11(4), pp. 325-361, April 1981.
- Bromberg, Howard. COBOL and Compatibility. *Datamation* 7(2), pp. 30-34, February 1961.
- Bromberg, Howard. USA National Activity Report to ISO/TC97/Subcommittee 5 - Computers and Information Processing (15 May 1963). *Comm. ACM* 6(9), pp. 502-505, September 1963.
- Brown, Richard M. Use of a Conditional Base Number System for Encoding Sequences of Correlated Characters. *Comm. ACM* 8(4), pp. 229-230, April 1965.
- Brown, S.A., Drayton, C.E. and Mittman, B. A Description of the APT Language. *Comm. ACM* 6(11), pp. 649-658, November 1963.
- CAD Centre. **GINO-F User Manual** (Issue 2). Computer Aided Design Centre, Cambridge, 1979.
- CAM-I. **By-Laws of Computer Aided Manufacturing - International Inc.** Computer Aided Manufacturing - International Inc., Chicago, June 1972.
- CII. **IFAPT.** (in French) Compagnie Internationale pour l'Informatique, Louveciennes, 1966.
- Curry, H.B. **On the Composition of Programs for Automatic Computing.** Memorandum 9806, Naval Ordnance Laboratory, Silver Spring, Maryland, 1949.
- Davis, Alan M. The Design of a Family of Application-Oriented Requirements Languages. *IEEE Computer* 15(5), pp. 21-27, May 1982.
- Dijkstra, E.W. Cooperating sequential processes. In **Programming Languages** (ed. F. Genuys), Academic Press, New York, 1968.
- DoD. **Reference Manual for the Ada Programming Language.** United States Department of Defense, Washington D.C., November 1980.
- EELM. EELM's Modifications to the New System. Appendix 8 of **APT New System Task Group - A Report of the first meeting held at E.E.L.M. Kidsgrove on Friday 11th March, 1966.** English Electric-Leo-Marconi, Kidsgrove, March 1966(a).
- EELM. **KIPPS Part-Programming Reference Manual.** English Electric-Leo-Marconi, Kidsgrove, 1966(b).
- Ellis, T.M.R. The Implementation of the APT New System on the English Electric-Leo-Marconi KDF9 Computer. In **Proc. APT Technical Meeting, April 19-20, 1966,** pp. 36-46, APT Long Range Program, IIT Research Institute, Chicago, 1966.

- Ellis, T.M.R. **Visit to U.S.A.: August 21 - December 23, 1966**
Report No. K/AA y 60, English Electric-Leo-Marconi, Kidsgrove,
16th January 1967.
- Ellis, T.M.R. **4-50/4-70 APT Reference Manual**. International Computers
Limited, London, 1969.
- Ellis, T.M.R. **MILDAPT - Modular Integrated Language Driven APT**.
In [McPherson, 1977], pp. 101-116, 1977.
- Ellis, T.M.R. **Parallel Processing in an Adaptable Application Oriented
Language Processor**. Software - Practice and Experience 9(3),
pp. 183-190, March 1979(a).
- Ellis, T.M.R. **A Useful Data-Structuring Technique**. Software - Practice
and Experience 9(6), pp. 477-484, June 1979(b).
- Ellis, T.M.R. **Structured Fortran - a Fortran 77 programming course**.
Computing Services, University of Sheffield, 1980.
- Ellis, T.M.R. **Notes on generating MILDAPT modules**. Report No. R3/5/81,
Computing Services, University of Sheffield, 1981.
- Ellis, T.M.R. **A Structured Approach to Fortran 77 Programming**.
Addison-Wesley, London, 1982.
- Ellis, T.M.R. and Coldham, V. **Visit to U.S.A.: April 17-29, 1966**.
Report No. K/AA y 58, English Electric-Leo-Marconi, Kidsgrove,
May 1966.
- Ellis, T.M.R. and János, J. **The Automatic Generation of CAD/CAM
Processors**. SME Technical Paper MS79-164, Society of Manufacturing
Engineers, Dearborn, Michigan, 1979. (Also in [Blake, 1980],
pp. 331-347, 1980)
- Ellis, T.M.R. and Semenov, O.I. (eds.) **Advances in CAD/CAM**.
(Proc. PROLAMAT'82), North-Holland, Amsterdam, 1983.
- Engelskirchen, W.H. **The EXAPT System**. In **Proc. 2nd European APT Technical
Meeting, October 27-28, 1966**, pp. 61-78, APT Long Range Program,
IIT Research Institute, Chicago, 1966.
- EXAPT-Verein. **EXAPT 1 Sprachbeschreibung**. Verein zur Förderung des
EXAPT-Programmiersystems e.V., Aachen, 1967(a).
- EXAPT-Verein. **EXAPT 2 Sprachbeschreibung**. Verein zur Förderung des
EXAPT-Programmiersystems e.V., Aachen, 1967(b).
- EXAPT-Verein. **EXAPT 1/2 CL-TAPE Beschreibung**. Verein zur Förderung des
EXAPT-Programmiersystems e.V., Aachen, 1967(c).
- Ferranti. **Profile Data Handbook (FSP 212, Issue 1)**. Ferranti Ltd.,
Dalkeith, Scotland, 1964.
- Floyd, Robert W. **A Descriptive Language for Symbol Manipulation**.
Journal ACM 8, pp. 579-584, 1961.

- Floyd, R.W. Syntactic analysis and operator precedence. *Journal ACM* 10, pp. 316-333, 1963.
- Foster, J.M. **Automatic Syntactic Analysis**. McDonald-Elsevier, London, 1970. (pp. 59-61)
- Galeotti, Marco. The LINK programming system for NC applications. In [Hatvany, 1973], pp. 349-361, 1973.
- Gendre, J.C. SURFAPT - sculptured surfaces system. In [NEL, 1974], Paper No.6, 1974(a).
- Gendre, J.C. PROMO - a NC system for small computers. In [NEL, 1974], Paper No.9, 1974(b).
- Gerhardt, G. **A programming language for system development**. Internal report, Computer and Automation Institute, Hungarian Academy of Sciences, Budapest, 1977.
- Goldstine, H.H. and von Neumann, J. **Planning and Coding of Problems for an Electronic Computing Instrument: Report on the Mathematical and Logical Aspects of an Electronic Computing Instrument**. The Institute for Advanced Study, Princeton, N.J., 1947. (Reprinted in von Neumann's **Collected Works** (ed. A. H. Traub), Vol. 5, pp. 80-235, Pergamon, London, 1963)
- Gordon, G. A General Purpose Simulation Program. In **Proc. EJCC, Washington, D.C.**, pp. 87-104, Macmillan, New York, 1961.
- Grems, Mandalay and Porter, R.E. A truly automatic programming system. In **Proc. Western Joint Computer Conference**, 1956.
- Gries, David. **Compiler Construction for Digital Computers**. Wiley, New York, 1971.
- Hatvany, J. (ed.) **Computer Languages for Numerical Control**. (Proc. PROLAMAT'73), North-Holland, Amsterdam, 1973.
- Herzog, Max. German N/C Language Committee Report. In **Proc. APT Technical Meeting, September 20-22, 1966**, pp. 345-363, APT Long Range Program, IIT Research Institute, Chicago, 1966.
- Hoare, C.A.R. A structured paging system. *Computer Journal* 16(3), pp. 209-215, August 1973.
- Hoare, C.A.R. Monitors: an operating system structuring concept. *Comm. ACM* 17(10), pp. 549-557, October 1974.
- Holdsworth, D. System implementation in Algol 68-R. *Software - Practice and Experience* 7(3), pp. 331-339, May 1977.
- Hope, A.K. Linking vs. Integration in CAD/CAM Systems. 3rd UK/Hungarian Seminar on Computational Geometry for CAD/CAM - preprints, Cambridge, 1983(a).
- Hope, A.K. Private communication, 1983(b).

- Hopper, G.M. The interlude 1954-1956. In [ONR, 1956], pp. 1-2, 1956
- Hunt, James W. Programming Languages. IEEE Computer 15(4), pp. 70-88, April 1982.
- Hyodo, Yoshihiro. HAPT-3D: A programming system for numerical control. In [Hatvany, 1973], pp. 439-448, 1973.
- IBM. **Specifications for the IBM mathematical FORMula TRANslation system, Fortran.** International Business Machines Corporation, New York, 10 November 1954.
- IBM. **Autospot II Program** (1620-CN-05X). International Business Machines Corporation, Owego, N.Y., 1962.
- IBM. **ADAPT, A System for the Automatic Programming of Numerically Controlled Machine Tools on Small Computers.** Final Tech. Eng. Report (AF 33(600)-43365), International Business Machines Corporation, San Jose, 1963.
- IBM. **AUTOPOL - An NC Program for the IBM 1130 and IBM System/360.** International Business Machines Corporation, New York, 1968(a).
- IBM. **ROMANCE - An NC Language for the IBM 1130.** International Business Machines Corporation, New York, 1968(b).
- IBM. **GPSS V User's Manual.** International Business Machines Corporation, New York, 1970.
- ICL. **GEORGE 3 and 4 Operating Systems.** International Computers Limited, London, 1975.
- ICSL. **Surface Milling using PMT2** (CS 455). International Computing Services Limited, London, 1966(a).
- ICSL. **Point-to-Point Work using AID Mk.1.** International Computing Services Limited, London, 1966(b).
- ICT. **MILMAP Numerical Control Program.** International Computers and Tabulators, London, 1967.
- IITRI. **APT III CLTAPE and PROTAPE Format Specifications.** Section 10 of **APT Computer Programs**, IIT Research Institute, Chicago, 1962.
- IITRI. **Interim Report on APT System Re-organisation.** IIT Research Institute, Chicago, September 1964.
- IITRI. **New System Development.** IIT Research Institute, Chicago, 1965. (Vol.1, March 1965; Vol.2, April 1965)
- IITRI. **APT Dictionary** (Revision No.7). IIT Research Institute, Chicago, 1967.
- IITRI. **APT IV CLTAPE Format Specification.** Chapter III.C.2 of **APT IV Computer System Manual** (Vol.1), IIT Research Institute, Chicago, 1970.

- IITRI. **APT IV A4V1 System Description**. IIT Research Institute, Chicago, September 1971.
- János, J. A decoding technique for NC processors. In [McPherson, 1977], pp. 439-446, 1977.
- János, J. Private communication, 1978.
- János, J. and Lukács, G. Define-it-yourself Languages. In **CAD in Medium Sized and Small Industries** (ed. J. Mermet), North-Holland, Amsterdam, 1981.
- Kelley, R.A. The production man's guide to APT-ADAPT. American Machinist/Metalworking Production, 22 June 1964 (pp. 97-112).
- Kiviat, P.J., Villanueva, R., and Markowitz, H.M. **The SIMSCRIPT II Programming Language**. Prentice-Hall, Englewood Cliffs, N.J., 1968.
- Knarr, C.M. CAMP I system for numerically controlled programming. Tooling Production 28(1), pp. 40-43, 1962.
- Knuth, Donald E. and Pardoe, Luis Trabb. Early development of programming languages. In **Encyclopedia of Computer Science and Technology**, Vol. 7, pp. 419-493, Marcel Dekker, New York, 1977.
- Kochan, D. The SYMAP programming system. In [Leslie, 1970a], pp. 400-403, 1970.
- Kovács, G. and Turai, I. The Development of Interactive Turn-key Systems. In [Ellis and Semenov, 1983], pp. 671-681, 1983.
- Laning, J.H. and Zierler, N. **A program for translation of mathematical equations for Whirlwind I**. Engineering Memorandum E-364, Instrumentation Lab., MIT, Cambridge, Mass., January 1954.
- Lavington, S.H. The Manchester Mark I and Atlas: A Historical Perspective. Comm. ACM 21(1), pp. 4-12, January 1978.
- Leslie, W.H.P. (ed.) **Numerical Control Programming Languages**. (Proc. PROLAMAT'69), North-Holland, Amsterdam, 1970(a).
- Leslie, W.H.P. (ed.) **Numerical Control Users' Handbook**. McGraw-Hill, Maidenhead, England, 1970(b).
- Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction Mechanisms in CLU. Comm. ACM 20(8), pp. 564-576, August 1977.
- Lister, A.M. and Sayer, P.J. Hierarchical Monitors. Software - Practice and Experience 7(5), pp. 613-623, September 1977.
- Lukács, G. **The Language Generator of the MINDS**. Computer and Automation Institute, Hungarian Academy of Sciences, Budapest, 1981.
- McPherson, D. (ed.) **Advances in Computer Aided Manufacture**. (Proc. PROLAMAT'76), North-Holland, Amsterdam, 1977.

- Macurek, I. and Vencovsky, J. NC programming in C.K.D. Praha.
In [Hatvany, 1973], pp. 151-161, 1973.
- Mangold, W.E. Status of NC Language Standardization in I.S.O.
In [Leslie, 1970a], pp. 101-109, 1970.
- Mangold, Werner E. N/C language standardization in I.S.O.
In [Hatvany, 1973], pp. 243-275, 1973.
- Markowitz, H., Hausner, B. and Karr, H.W. **SIMSCRIPT - A Simulation Programming Language**. Prentice-Hall, Englewood Cliffs, N.J., 1963.
- Matsa, S.M. **AUTOPROMPT programming system. Preliminary Programming Manual**. International Business Machines Corporation, New York, 1961.
- MDSI. **COMPACT-II Programming Manual**. Manufacturing Data Systems Inc., Ann Arbor, Michigan, 1973.
- MIT. **APT Press Conference**. MIT, Cambridge, Mass., 25 February 1959.
(Reproduced in full in Appendix C of [Ward, 1960])
- Mittman, Benjamin. Development of numerical control programming languages in Europe. In **Proc. 22nd National Conference of the ACM**, pp. 479-482, Thompson Book Co., Washington, 1967.
- Myers, Ware. CAD/CAM: The Need for a Broader Focus. *IEEE Computer* 15(1), pp. 105-117, January 1982.
- NAG. **NAG Fortran Library Manual Mark 8** (in 6 vols.). Numerical Algorithms Group, Oxford, 1981.
- Neave, R.S. **Clam**. Interim Report RSN/GEN/39, Appendix I, De Havilland Aero Department, Hatfield, Herts., 1964.
- Nelder, J.A. and members of the Rothamstead Statistics Department. **GENSTAT Reference Manual**. Program Library Unit, Edinburgh Regional Computing Centre, Edinburgh, 1973.
- NEL. **Programming of Numerically Controlled Machine Tools - NEL Report No. 187**. National Engineering Laboratory, East Kilbride, Glasgow, 1965.
- NEL. **2CL Part-programming Reference Manual - NEL Report No. 299**. National Engineering Laboratory, East Kilbride, Glasgow, 1967.
- NEL. **2CL Part-programming Reference Manual First Revision - NEL Report No. 424**. National Engineering Laboratory, East Kilbride, Glasgow, 1969.
- NEL. **CAM74 Preprints**. National Engineering Laboratory, East Kilbride, Glasgow, 1974.
- Nie, Norman H., Hadlai Hull, C., Jenkins, Jean G., Steinbrenner, Karin, and Bent, Dale H. **SPSS - Statistical Package for the Social Sciences (2nd edition)**. McGraw-Hill, New York, 1975.
- Nussey, I.D. The Purpose and Future of Part-Programming. In [Leslie, 1970a], pp. 186-205, 1970.

- Olivetti. **SURF-AUCTOR Reference Manual**. Olivetti, Ivrea, Italy, 1968.
- ONR. **Symposium on Automatic Programming for Digital Computers**. Office of Naval Research, Department of the Navy, Washington, D.C., 1954.
- ONR. **Symposium on Advanced Programming Methods for Digital Computers**. Office of Naval Research, Department of the Navy, Washington, D.C., 1956.
- Opitz, H., Simon, W., Spur, G., Stute, G. The Programming of Numerically Controlled Tools. In **Frontiers in Manufacturing Technology**, Institute of Science and Technology, University of Michigan, Ann Arbor, Michigan, September 1967.
- Pease, W. An Automatic Machine Tool, *Scientific American* 187(3), pp. 101-115, September 1952.
- Perlis, A.J. and Samelson, K. (jt.eds. on behalf of the ACM-GAMM Committee). Preliminary Report - international algebraic language. *Comm. ACM* 1(12), pp. 8-22, December 1958.
- Perlis, A.J. and Samelson, K. (jt.eds. on behalf of the ACM-GAMM Committee). Report on the algorithmic language ALGOL. *Num.Math.* 1, pp. 41-60, 1959.
- Perlis, A.J., Smith, J.W., and Van Zoeren, H.R. **Internal Translator (IT): a compiler for the 650**. Carnegie Institute of Technology, Pittsburgh, March 1957.
- Pittler. **AUTOPIT 1**. (in German) Pittler A.G., Langen, 1967.
- Pittler. **AUTOPIT 2**. (in German) Pittler A.G., Langen, 1968.
- Pruuden, J. APROKS - A programming system for NC flame-cutters. In [Leslie, 1970a], pp. 350-356, 1970.
- RCA. **RCA 501 Cobol Narrator, Programmers' Reference Manual (P 501-03.032)**. RCA Electronic Data Processing Division, Cherry Hill, N.J., December 1960.
- Reckziegel, Dieter. EXAPT 1. In [Leslie, 1970b], Chapter 7 (pp. 159-196), 1970(a).
- Reckziegel, Dieter. EXAPT 2. In [Leslie, 1970b], Chapter 8 (pp. 197-235), 1970(b).
- Renault, J. and Taboy, J. CELAPT - a CAM/CAD interactive system. In [NEL, 1974], Paper No.11, 1974.
- Rodriguez, G.W. **CLFILE and CLTAPE Record Formats**. Memorandum to A. Pfeiffer, IIT Research Institute, Chicago, 27 July 1965.
- Rolls-Royce. **Cocomat I Program**. Report CPR 1175, Rolls-Royce Aero Division, Derby, 1962.

- Ross, Douglas T. Design of Special Language for Machine-Tool Programming. In **Course Outline and Workbook for the Special Course on Programming for Numerically Controlled Machine Tools**, Servomechanisms Lab., MIT, Cambridge, Mass., March 1957(a). (Reproduced in full in [Ross, 1978], pp. 72-74)
- Ross, Douglas T. **A Proposed Basic Language for the 2D APT II**. Report No. 2D APT II-2, Servomechanisms Lab., MIT, Cambridge, Mass., 14 June 1957(b). (Reproduced in full in [Ross, 1978], pp. 78-80)
- Ross, Douglas T. The Design and Use of the APT Language for Automatic Programming of Machine Tools. In **Proc. 1959 Computer Applications Symposium**, pp. 80-99, Armour Research Foundation of the Illinois Institute of Technology, Chicago, 1960.
- Ross, D.T. A Generalised Technique for Symbol Manipulation and Numerical Calculation. *Comm. ACM* 4(3), pp. 147-150, March 1961.
- Ross, D.T. Away with the bundling board! In [Leslie, 1970a], pp. 455-461, 1970.
- Ross, Douglas T. Origins of the APT Language for Automatically Programmed Tools. **ACM Sigplan History of Programming Languages Conference Preprints**, ACM Sigplan Notices 13(8), pp. 61-99, August 1978.
- Ross, D.T., Rodriguez, J.E., and Feldman, C.G. (ed.) **AED-0 Programmer's Guide**. Softech Inc., Waltham, Mass., 1970.
- R.R.E. **Algol 68-R RRE Programmer's Manual**. Royal Radar Establishment, Malvern, England, 1976.
- Runyon, J.H. **Whirlwind I Routines for Computations for the MIT Numerically Controlled Milling Machine**. Report No. 6873-ER-8, Servomechanisms Lab., MIT, Cambridge, Mass., December 1953.
- Rutishauser, Heinz. Automatische Rechenplanfertigung bei programm-gesteuerten Rechenmaschinen. In **Mitteilungen aus dem Inst. für angew. Math. an der ETH Zürich**, Nr. 3, Basle, 1952.
- Sabin, M.A. **The Use of Piecewise Forms for the Numerical Representation of Shape**. (Candidate of Technical Sciences dissertation) SzTAKI Report 60/1977, Hungarian Academy of Sciences, Budapest, 1976.
- Sammet, Jean E. **Programming Languages: History and Fundamentals**. Prentice Hall, Englewood Cliffs, N.J., 1969.
- Shave, M.J.R. **Data Structures**. McGraw-Hill, London, 1975.
- Siegel, A. **Information Processing Routine for Numerical Control**. Report No. 6873-ER-16, Servomechanisms Lab., MIT, Cambridge, Mass., March 1956(a).
- Siegel, A. Automatic Programming of Numerically Controlled Machine Tools. *Control Engineering* 3(10), pp. 65-70, October 1956(b).

- Sim, R.M. **The logical record structure of CLDATA for the NEL 2C,L program.** NEL Memo X5/152, National Engineering Laboratory, East Kilbride, Glasgow, June 1968.
- Sohlenius, G. and Iacobaeus, U. **PRAUTO - a drilling module for different geometrical processors.** In [NEL, 1974], Paper No.1, 1974.
- Stocker, William M. and Emerson, Charles D. **Numerical Control - what it means to Metalworking.** American Machinist, October 25th 1954 (pp. 133-156).
- Stute, G., Opitz, H. and Spur, G. **EXAPT 3 Sprachbeschreibung.** Verein zur Förderung des EXAPT-Programmiersystems e.V., Aachen, 1971.
- Sundstrand. **SPLIT programming manual.** Sundstrand Corporation, Rockford, Illinois, 1964.
- Vencovsky, J. and Macurek, I. **Programming the AGIECUT NC wire spark-erosion machine using the CKDAPT system.** In [NEL, 1974], Paper No.2, 1974.
- Vliestra, J. and Wielenga, D.K. **Philcon.** In [Leslie, 1970a], pp. 53-70, 1970.
- Ward, J.E. **Automatic Programming of Numerically Controlled Machine Tools. (Final Report)** Report No. 6873-FR-3, Servomechanisms Lab., MIT, Cambridge, Mass., 15 January 1960.
- Ward, J.E. **Numerical Control of Machine Tools.** In **McGraw Hill Yearbook of Science and Technology**, McGraw Hill, New York, 1968.
- van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A. **Report on the Algorithmic Language ALGOL 68.** Num.Math. 4, pp. 79-218, 1969.
- van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., and Fisher, R.G. **Revised Report on the algorithmic language ALGOL 68.** Acta Informatica 5, pp. 1-236, 1975.
- Wirth, N. **The Programming Language PASCAL.** Acta Informatica 1, pp. 35-63, 1971.
- Wirth, N. **Modula: a Language for Modular Multiprogramming.** Software - Practice and Experience 7(1), pp. 3-36, January 1977.
- Wirth, N. and Hoare, C.A.R. **A Contribution to the Development of ALGOL.** Comm. ACM 9(6), pp. 413-431, June 1966.
- Wood, B.J. **Computer production of control tapes.** In [Leslie, 1970b], Chapter 5 (pp. 98-119), 1970.
- Woodward, P.M. and Bond, S.G. **Algol 68-R Users' Guide.** H.M.S.O., London, 1974.

Zuse, K. Der Plankalkül. Berichte der Gesellschaft für Mathematik und Datenverarbeitung 63(3), Bonn, 1972. (Manuscript prepared in 1945)

A TANULMÁNYSOROZATBAN 1982-BEN MEGJELENTEK

- 130/1982 Barabás Miklós - Tőkés Szabolcs: A lézer printer képképzés hibái és optikai korrekciójuk
- 131/1982 RG-II/KNVVT "Sztisztémü upravlenija bazani dannüh i informacionnüe szisztémü" Szbornik naucsno-iszszledovatel'szkih rabot rabocsej gruppü RG-II KNVVT, Bp. 1979. T o m I.
- 132/1982 RG-II/KNVVT T o m II.
- 133/1982 RG-II KNVVT . T o m III.
- 134/1982 Knuth Előd - Rónyai Lajos: Az SDLA/SET adatbázis lekérdező nyelv alapjai /orosz nyelven/
- 135/1982 Néhány feladat a tervezés-automatizálás területéről. Örmény-magyar közös cikkgyűjtemény
- 136/1982 Somló János: Forgácsoló megmunkálások folyamatainak optimálási és irányítási problémái
- 137/1982 KGST I-15.1. Szakbizottság 1979. és 80. évi előadásai
- 138/1982 Kovács László: Számítógép-hálózati protokollok formális specifikálása és verifikálása
- 139/1982 Operációs rendszerek elmélete 7. visegrádi téli iskola

A TANULMÁNYSOROZATBAN 1983-BAN MEGJELENTEK

- 140/1983 Operation Research Software Descriptions (Vol.1.)
Szerkesztette: Prékopa András és Kéri Gerzson
- 141/1983 Ngo The Khanh: Prefix-mentes nyelvek és egyszerű
determinisztikus gépek
- 142/1983 Pikler Gyula: Dialógussal vezérelt interaktív
gépészeti CAD rendszerek elméleti és gyakorlati
megfogalmazása
- 143/1983 Márkus Zsuzsanna: Modellelméleti és univerzális
algebrai eszközök a természetes és formális nyelvek
szemantikaelméletében
- 144/1983 Publikációk '81 /Szerkesztette: Petrőczy Judit/
- 145/1983 Telcs András: Belső állapotú bolyongások
- 146/1983: Varga Gyula: Numerical Methods for Computation of
the Generalized Inverse of Rectangular Matrices
- 147/1983 Proceedings of the joint Bulgarian-Hungarian
workshop on "Mathematical Cybernetics and data
Processing" /Szerkesztette: Uhrin Béla/
- 148/1983 Sebestyén Béla: Fejezetek a részecskefizikai
elektronikus kísérleteinek adatgyűjtő, -feldolgozó
rendszerei köréből
- 149/1983 L. Keviczky, J. Hethéssy: A general approach for
deterministic adaptive regulators based on explicit
identification
- 150/1983 IFIP TC.2 WORKING CONFERENCE "System Description
Methodologies" May 22-27. 1983. Kecskemét.
/Szerkesztette: Knuth Előd/

151/1983 Márkus Zsuzsanna: On First Order Many-Sorted
LOGIC

152/1983 Operations Research Software Descriptions /Vol.2./
Edited by A. Prékopa and G. Kéri

