THE DESIGN OF A NETWORK COMMAND LANGUAGE

PHILLIP CAMPBELL JENKINS

1980

A thesis presented in partial fulfilment of the requirements for the degree of Master of Arts in Computer Science at Massey University.

ABSTRACT

Most computer series have their own distinct operating system control language. At present there is no world-wide standard control language, but the recent development of heterogeneous networks, where users may access many different computers, has highlighted the need for one.

A project to set up an experimental network, called KIWINET, between Massey and Victoria Universities, initiated research into the design of a standard control language for the network. The result of this research is a high-level, block structured language called the Network Command Language (NCL). The design of this language and its implementation are discussed.

ACKNOWLEDGEMENTS

The initial work for this thesis was done while I was employed by the Massey University Computer Centre. I would like to thank all the staff of the Computer Centre and also the staff of the Computer Science Departments of Massey and Victoria Universities, for their continued support of the project, both as technical advisers and as constructive voices during discussions. In particular I would like to thank Mr Neil James, of Massey University Computer Centre, and Mr Keith Hopper, now of Leeds University, England, who have both been, at various time, my supervisors and who have been intimately concerned with the design of NCL.

Special thanks are owed to my present supervisor, Mr Lindsay Groves, of Massey University Computer Science Department, and his colleague Mr Tom Docker, for many useful discussions and for their ever helpful criticism.

I would also like to acknowledge the invaluable constructive criticism and suggestions concerning all phases of the design of NCL, received from Professor Unger and his colleagues of Dortmund University, West Germany.

Lastly, I would like to thank Mrs Thomson, for doing the proofreading, and my fiancée, Alison, for making even the bad times good.

CONTENTS

<><><><><><><><><><><>

TABLE OF DIAGRAMS

CHAPTER 1

INTRODUCTION

"About $\$10^9$/year are lost due to command language errors."

T. B. Steel

Said during discussion at the IFIP

Working Conference on Command Languages,

Sweden, 1974.

1.1 The emergence of networks

In 1968 the Advanced Research Projects Agency (ARPA) of the U.S. Department of Defense began to create what is now called ARPANET; a network of about 60 heterogeneous computers at geographically distributed sites throughout the United States. It includes satellite links to computers in Hawaii, Norway and London.

The main idea behind ARPANET was to allow the full resources of the computers in the network to be shared among the users of the network. This includes not only the possibility of sharing software, but also special hardware capabilities such as ILLIAC IV and also unique data sources such as large global weather data bases.

ARPANET was, and still is, a success. Following this lead several other networks have been implemented and many more are being planned. There is now COST-11 linking England, France, Switzerland and Italy; CYCLADES, a French network; EPSS, developed by the British Post Office; SITA, a special purpose network for European airlines and ALOHA, a network linking together countries in the Pacific area and Australasia.

1.2 The KIWINET project

In late 1974 and during 1975 Computer Centre and Computer Science Department staff from Victoria University, Wellington and Massey University,

Palmerston North, met occasionally to discuss the feasibility of setting up an experimental computer network linking the two sites. The aim of the network, which had been given the name KIWINET, was to enable the reduction of computing costs at both sites by sharing software and hardware resources.

Both project sites had identical computers (Burroughs B6700's) and the initial proposal was that the network would consist solely of these two machines. However it was felt that any future development of the network should not be restricted to Burroughs computers and that therefore any design of the initial network should make as few assumptions as possible about the type of computers comprising it.

In October 1975 finance was acquired for the rental of a Post Office telephone line between the two sites and shortly thereafter formal meetings were held to set up the research team and provide direction to the team's efforts. At these meetings it was felt that the project could be considered on four separate, but naturally inter-related, levels and four working groups, one for each level, were accordingly formed.

(i) Hardware Level

This level, conceptually the lowest level, deals with the data communication equipment on which the network will be implemented. The desired characteristics of this equipment (e.g. speed, reliability)

depend to some extent on the way the whole system will be used. Price/performance trade-offs must be made when deciding such things as line baud rates and what to use for front-end processors.

(ii) Data Communication Level

This level deals with the method by which messages are encoded and transmitted between nodes of the network. Decisions must be made concerning message protocols, and message switching software must be developed.

(iii) The Control Language Level

This level is concerned with the language(s) a user needs to control the network, commanding it to perform any desired tasks. Consideration must be given to what the user needs to do and a language to do it with must be provided.

It is this level with which this thesis is concerned.

(iv) Project Management Level

This, the highest level, is concerned with the services provided by the network and overall management of the project. Consideration is given to things such as the user charging system, resource co-ordination and project finance.

1.3 The Control Language Problem

The problem faced by the control language group is based on a problem faced by the users of all automated machinery since Frankenstein invented his monster; once you've got your program into a computer, once you've built your monster, you must have some way to control it, some way to tell it to start, perform certain tasks and then stop.

Nowadays, with computers, a big red 'GO' button is no longer enough. Modern-day computer users want to do complex things - they may want to fiddle with files copying them from one medium to another, renaming them and so on; they may want to execute one program only if another succeeded first; they may even want to run several programs simultaneously, synchronising their actions at several points.

The complexity of the requirements of modern-day users demands that there must be some language with which a user can control the computer, specifying what he wants done with his programs and files. Of course, it is not the computer that users control but rather the Operating System (OS) and, to be accurate, it's more a case of requesting the OS to perform the desired tasks, rather than controlling it.

There is a distinction between the language (or languages) used to program a computer and that used to control the operating system. The former are called Programming Languages (PL's) and the latter is called an Operating System Control Language (OSCL) or a Command Language (CL)*. Most modern OSCL's are designed for one and only one particular OS and no OS will understand the OSCL designed for another. To make the situation even worse each distinct computer series has its own, specially designed, idiosyncratic OS.

Now comes the problem. Most users who wish to do anything with a computer must use that computer's OSCL. (Admittedly, with some systems, e.g. the HP300, users don't have to do anything, being provided with Menus

---

* Further names are Job Control Language (JCL), Job Description Language (JDL) and just Control Language. We will use OSCL or CL only.

and Help facilities. These however are special cases and the more sophisticated users of these systems still require an OSCL.) Now, having to use an OSCL might not seem so bad for the user of a single machine but when that user is faced with a network of machines, each one with a possibly different OSCL, things get a bit difficult. Imagine Frankenstein trying to control an army of monsters each one of which speaks a different language.

1.4. The SCL decision

The control language team, confronted with the afore-mentioned problem had two choices. They could either have ignored it, providing simply a mechanism within the OS for identifying to it the site at which a particular job should be run and where files were located, or they could have provided a common control language for all machines in the network. After much discussion it was agreed that to ignore the problem was to ignore the needs of the network's eventual users and that a standard OSCL should be provided.

The team corresponded with Professor Sayani, Chairman of the CODASYL OSCL task group, and Professor Unger, Chairman of the IFIP Conference on Command Languages 1974. They replied that no standard OSCL had as yet been defined although there appeared to be a preference for a block-structured language. With this in mind a search for suitable block-

structured OSCL's was initiated. The team found only two:- ICL's System Control Language (SCL) developed for their 2900 series of computers, and Burroughs B6700/7700 Work Flow Language (WFL) with which the team was already familiar.

WFL is a batch language only. CANDE, a separate unstructured language, is provided for interactive use. These two languages are not orthogonal, different constructs being required to carry out similar functions. The team felt that a standard language should be usable with a minimum of differences in both batch and interactive modes. Futhermore it was considered that it would be difficult to adapt these languages to include the general handling of files and system resources and those features necessary to enable specification of where programs are to be run and files to be held.

At first sight SCL appeared to be more promising than WFL, offering a well constructed block system and powerful control structures, yet still remaining adaptable. Deeper investigation revealed that files and other system resources (e.g. peripherals) are handled by a large number of macros and procedures. The average user would need to know approximately 30 out of over 100 supplied and this was felt unacceptable.

However SCL was felt to be the better language and it was decided to use it as a basis around which a network control language could be designed.

At this point, in mid 1976, the author, then working for the Massey University Computer Centre, joined Mr Keith Hopper and Mr Neil James of the control language team and began work on the additions and alterations to SCL.

1.5 The aim of this thesis

The aim of this thesis is to present the author's work on the design of a standard OSCL for a network of computers.

The first stage of this work consists of defining the problem. The role of an OSCL is placed in relation to the complete interface between users and a network and the qualities and features that a standard OSCL should have are defined.

Having defined the problem the next steps is to determine what other people have done about it. A survey of work on the standardisation of OSCLs is carried out and several 'standard' OSCLs are evaluated.

The third stage of the work is to develop SCL, adding any features necessary for networks and making alterations to avoid the problems and disadvantages of previous OSCLs.

Lastly, although an implementation of the language has not been attempted, the problems involved in one are considered.

CHAPTER 2

PROBLEM ANALYSIS

The problem we are faced with is that of defining an interface between users and a network of computers. This interface must allow users to store data and get programs executed on the network. The characteristics of the interface are obviously defined by what it is acting as an interface for. Thus we must first define what a user is and what a network is.

2.1 Users

First let's tackle the term 'user'. Several studies have attempted to define 'user' with varying degrees of success. A good one-sentence definition is:-

"A user of a computer system is any entity having (occasional) access to this computer system in order to cause the elaboration of some operation." [Kugler et al 79]

This definition, although correct, is not very useful. A more helpful approach is to look at the different types of users, the wide diversity of which is perhaps one of the most important factors in control language design. A categorisation of users, detailing the various ways they use computers, will not only help our understanding of the term 'user' but also aid us in later deciding what the requirements of users are.

There are many ways to categorise users none of which will ever be completely correct. However, a working categorisation is better than none at all. The following categories are based on a consideration of the complexity of users requirements. There are four main types of user, listed here in order of increasing complexity.

(i) The naive or casual user; not a programmer but rather a user of someone else's programs; someone for whom the system should appear to be transparent.

(ii) The occasional programmer; someone working in an application field with occasional need to write and run programs, generally of a moderately sophisticated nature but usually only requiring simple operating system facilities.

(iii) The applications programmer; someone whose regular work involves writing, testing and running applications of varying degrees of complexity but which generally require the use of system resources in a non-trivial manner; includes the system software engineer.

(iv) The computer operator; someone who regularly needs to perform sophisticated operations with the system resources; needs to be able to affect the global parameters of the operating system; includes the network manager.

An important point here is that most users would like to be able to treat the system as being devoted to their own needs and wishes; they would like a personalised machine which understands a particular command language suited to their own computing abilities and which allows them to create objects and manipulate them in ways related to their own problems. Further, they would like any system messages to be tailored to their own understanding of the sysytem.

Certain other factors should also be noted concerning the limitations of users in general [Cheriton 76].

Humans

- are error prone and unpredictable
- are slow
- are single channel devices
- have small, short-term memories (7 items)
- have low information throughput
- degrade with fatigue or overload
- are sensitive to response time.

The wide diversity of users, plus their all too human failings, combine to make OSCL design a demanding problem.

2.2 Networks

A network can be loosely defined as consisting of two or more, possibly dissimilar, computers each connected to the others by some form of data communication links. See Diagram 2.1.

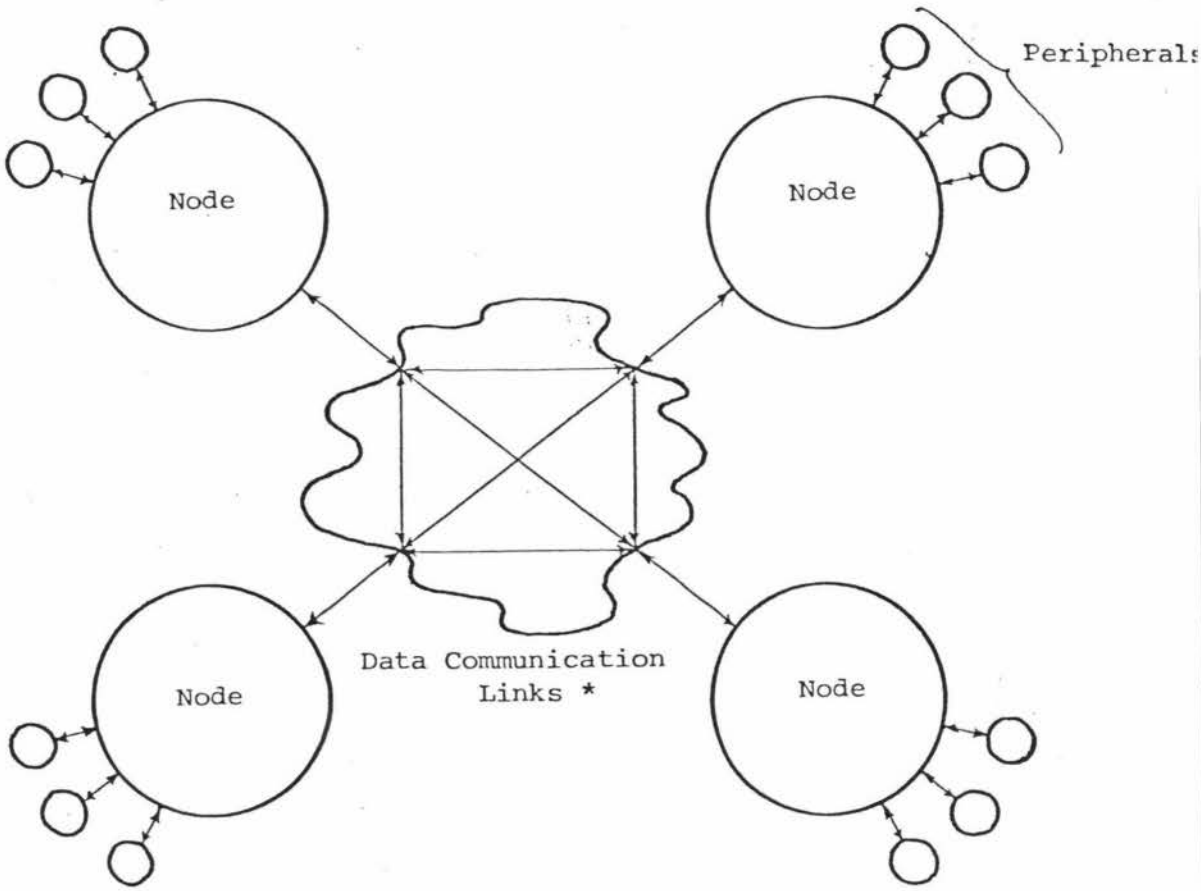


Diagram 2.1 Logical diagram of a network.

Each computer is called a *node* of the network and can act as *host* to any particular user's programs and data. A node is somtimes also called a *site*. Connected to each node is a number of I/O devices, or *peripherals*. A user can input commands at one of the peripherals, which may be a card reader or some other *batch* input device or an *interactive* terminal. Most often, for convenience, we will talk about users entering commands at terminals. When a user enters commands at a terminal the node to which the terminal is attached is called the user's *local* node. All other nodes in the network are called *remote* nodes.

---

* Some of the links shown may be virtual, i.e. they may exist only logically, there being no *direct* route between the two nodes involved.

2.3 Types of networks

A network can be classified as being either homogeneous or heterogeneous. In an homogeneous network all computers are similar (i.e. they are all of the same manufacturers series e.g. ICL 2900s) whereas the computers in an heterogeneous network may be dissimilar.

Also, there is a type of network called an hierarchical network. See diagram 2.2.

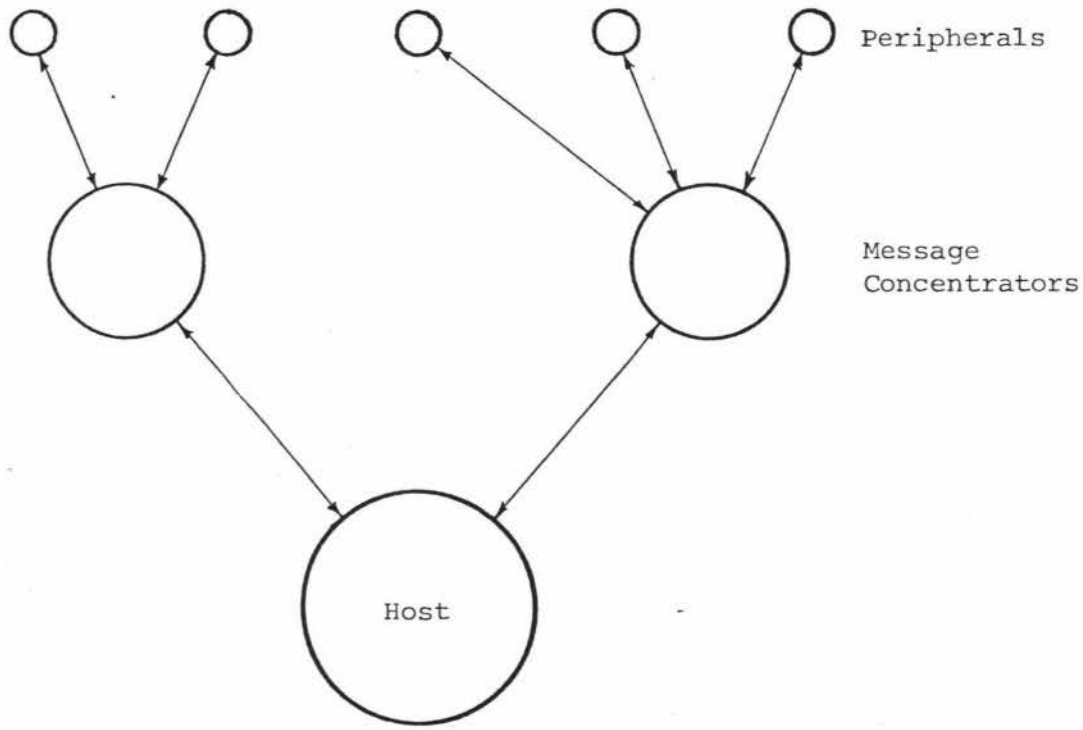


Diagram 2.2 Logical Diagram of an hierarchical network.

This is not really a true network in that, from a user's point of view, there is only one processor involved. Admittedly the concentrators also do a bit of processing but only to aid the running of the system and not for users directly.

The work in this thesis makes no assumptions about the composition or structure of networks. The results apply to all forms of networks; whether homogeneous, heterogeneous, hierarchical or otherwise.

2.4 The user-network interface.

Each node in a network has its own controlling OS and in this respect we can regard a network as being composed of OSs. Thus our interface is really between users and a network of OSs.

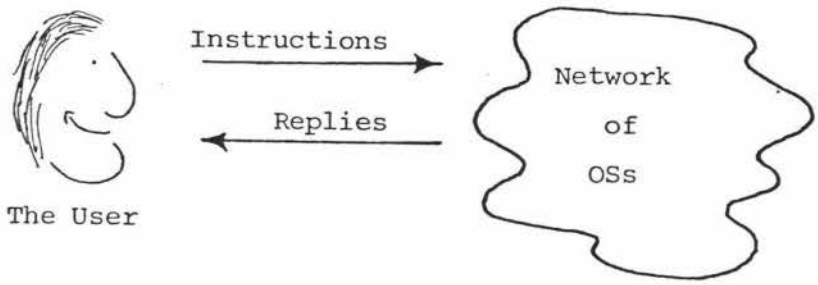


Diagram 2.3 The user-OS interface.

Any interface is a two way affair. See diagram 2.3. That is we need a user-to-OS language for specifying what the user wants done. Such a language is called an Operating System Control Language (OSCL). We also need an OS-to-user language so that the OS may make replies and responses to user commands. An OS-to-user language is called an Operating System Response Language (OSRL).

Obviously, these languages must be different as they fulfil different functions; just as a compiler's error messages are in a different language from the one it is compiling.

Within a network each OS must be able to communicate with every other OS. This requires a third language, called an Operating System Access Language (OSAL), used for sending and receiving internode commands and data.

In some respects an OSAL is a high-level message protocol acting between OSs. For example, when a user inputs statements, using an OSCL, for execution at a remote node they are embedded, as is, in an OSAL message and sent to the correct site. However, the team regard it as being just one aspect of the complete OS interface defined by the three languages. Whenever an OS communicates with anything it should do so through these three languages. See diagram 2.4.

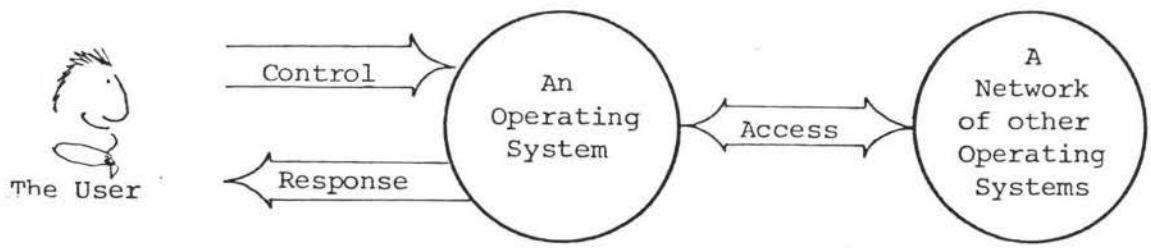


Diagram 2.4 The Operating Sytem interface languages.

2.5 The teams' objectives

In light of the previous discussions we can see that the problem we are faced with is that of designing an interface which will allow users, ranging from the naive to the highly sophisticated, to store data and run programs on a network of OSs.

The 'control language' team decided to attempt to standardise the interface by standardising all three languages. Each member of the team

tackled a separate language and each language was treated as a separate project although, naturally, there was a certain amount of interaction.

To provide every OS with a uniform interface with other OSs Mr Neil James worked on the standardisation of the OSAL. The result, a language called the Network Access Language (NAL) is described in [Hopper et al 77]. As users should not have to cater with a wide variety of system replies from different OSs, Mr Keith Hopper worked on standardising the OSRL. It was called the Network Response Language (NRL). Mr Hopper is still working on NRL in conjunction with Professor Unger and his colleagues from Dortmund University, West Germany, who are working on a similar project to KIWINET [Kugler et al 79]. The author worked on the OSCL. We called it the Network Command Language (NCL). The overall interface, defined by the languages NCL, NRL and NAL, is called the Universal Network Interface (UNIFACE).

CHAPTER 3

DESIGN CRITERIA

When the majority of the people have clear-cut criteria to go by, criticism and self-criticism can be conducted along proper lines, and these criteria can be applied to peoples' words and actions to determine whether they are fragrant flowers or poisonous weeds.

Mao Tse-Tung

Speech, Peking

We have shown that modern job control languages are akin to programming languages. Recognition of this fact ... indicates that we should criticise job control languages using the same criteria of judgement that we apply to programming languages.

D. W. Barron & I. R. Jackson

The Evolution of Job Control Languages

[Barron & Jackson 72]

Before we can begin to consider the design of NCL we must know what we want in the language, what functions it should provide and generally what 'shape' the language should have. In this chapter we are trying to compile a set of criteria for an OSCL which we will use later in evaluating other OSCLs and in deciding what features to include in NCL.

3.1 Sessions

An OS, typically, runs continually, idling until some input is presented to it. This input can come from batch or interactive terminals or from other nodes. The input is divided into sections, called *jobs* or *sessions*. A session is the major unit of work in an OSCL and is, typically, delimited by symbols such as JOB and ENDJOB.

3.1.1 User identification

For charging purposes and, more importantly, for security reasons the system must know who it is working for. Additionally, knowing who the user is will allow the system to provide a personalised interface, allowing him access to user-dependent facilities and objects.

Thus, when a user starts a session he must identify himself to the system. The process of identification at the start of a session is called *log-on*. The converse, when a user ends a session, is called *log-off*.

3.1.2 Simultaneous command execution

Some processes, for example compilers, often take a long time to execute and interactive users at a terminal can waste time waiting for them to finish. Such users often want to be able to enter several commands and have them all executed simultaneously. For example, a user might want to enter an 'execute process p' command and, while the system is still executing p, enter a 'compile file f' command and, while the system is still executing p and compiling f, enter an 'edit file d' command. This could be repeated indefinitely, allowing the user to have a large number of commands executing simultaneously. In actual practise of course, there will very likely be a limit on the number of simultaneous commands allowed.

3.1.3 Session breakin

A session will typically consist of the execution of a number of programs. An executing program is called a process. Having started a process the interactive user often needs to stop it executing because it has entered an infinite loop or perhaps the user suddenly remembers some vital adjustment that should have been made prior to execution. Also common is the need to suspend a process to correct some fault in its environment, for example a lack of input data.

Thus some means should be provided to allow a user to break into a process, suspending its execution or perhaps killing it altogether. In this respect the interactive language will obviously differ from the batch language.

3.2 Processes

3.2.1 Process initiation

The main reason for using an OSCL is to get programs executed. This includes everything from user-written programs to application packages; from compilers to standard file-handling procedures.

Obviously, the control language should allow users to initiate processes and, because it would be used so often, the language feature allowing process initiation should be as simple as possible, allowing little room for user error. The facility should also allow the user to specify where (i.e. at which node of the network) the process should be executed.

3.2.2 Process environments

A process will consume computational resources, such as CPU time and lines read. Naturally any user should not be allowed unlimited resources. The system needs to be able to restrict the amount of resources used by

(i) a user in total

(ii) any particular session.

Also, a user himself should be able to restrict the amount of each resource that can be used by his programs or find out how much has been used while they are running.

An *environment* is a set of values which specify the resources which a process may use. The environment for a particular user session should depend upon the type of user. Thus, we could, for example, use the

classification scheme described in Chapter 2 to divide users into classes and provide a separate environment for each class of user. The same applies to the total resources the user may consume. Also, if a user doesn't explicitly specify an environment for a process, a user dependent standard default environment should be used.

3.2.3 Concurrency

Sophisticated users sometimes want to execute processes concurrently, and an OSCL should provide facilities for this. Naturally a synchronisation mechanism must also be provided to allow one process to wait for another to reach a certain state and to allow the coordination of the use of shared data objects.

An example of the use of such facilities is an airline reservation system, where there is a set of distributed terminals accessing a data-base. Each terminal can initiate processes which access or update the data-base. Since many processes are allowed to access the data-base simultaneously these processes can be executed concurrently. On the other hand only one process should be allowed to update the data-base at any one time and so some form of synchronisation is neccessary.

Providing concurrency facilities and a synchronisation mechanism is especially necessary in a network situation where such facilities would allow the user to take full advantage of the multi-processor characteristics of the network.

3.3 Procedures

Users often have particular tasks or functions that need to be repeated several times within a session. To avoid the effort of rewriting the statements for a task each time it is needed, users should be able to declare procedures or macros. Further, because a procedure may operate identically on a number of objects, which may be unspecified or not even created when the procedure is declared, it should be possible to specify, when the procedure is called, which objects it should use. That is, there should be a parameter passing mechanism available.

A user may have a procedure which he needs to use in several sessions. To cater for this it should be possible to make procedures have an existence independent of the session in which they are declared (i.e. save them in the user library).

3.4 Files

The ability to store data and programs on the system is perhaps as important to users as the ability to initiate processes. All modern OSs provide filing systems for this purpose. In a network OSCL it must be possible to access files at any node in the network.

Conceptually, we can consider a file as some sort of physical container inside which there is a set of information. Connected to this is a logical description of the contained information. The container is managed by the operating system and the set of information is manipulated by the user at the programming language level. The logical description is manipulated by the user at the command language level.

A file description is a list of values; one value for each property, or attribute, of the file. Some of these values are maintained by the OS (e.g. the last access date), while others are maintained by the user (e.g. the title).

Each OS in a network will support a different set of attributes for files. A standard OSCL system will necessarily have to provide a standard set of attributes. Thus the user will have an abstracted view of files in the network; some of the finer details of files, especially their physical attributes such as cylinder address, will be hidden from view.

What then should an OSCL allow users to do with files? Obviously, it should allow users to create and manipulate files and their attributes. Furthermore, because users often want to store data on the system for long periods they should be able to create permanent files; that is files which have an existence independent of the session in which they were created.

Because it is often necessary to change the data held in a file an edit facility should be provided. It should also be possible to delete files which are no longer needed, or copy files from medium to medium or site to site.

Another important aspect of files concerns the binding of logical descriptions to physical containers. Consider a program which is designed to use a card reader file for input. The program will obviously include a logical description of the desired file. To check the program, several files of test data are created and stored on disk for convenience during testing. To get the program to use the test files the user could

modify, recompile and execute the program for each file. However, it would be more convenient if, when initiating the program, he could specify a logical file description which will override that which the program uses. An OSCL should provide facilities for doing this.

3.5 Data types

When passing values to processes, handling file attributes and testing conditions (when specifying the flow of control in a session) the user needs to be able to define objects of an adequate range of data types (e.g. integer, boolean and string) and perform operations on them.

Structures of objects are also useful for holding objects related in some way to each other. For example, an array could be used to hold transaction files, one from each shop in a chain, which are to processed, via an iterative loop, against a master file.

3.6 Blocks and scope

As already stated, the team had decided on a block structured language before the author joined it. As in programming languages, blocks are useful for controlling the scope of variables. Because it is possible to handle files, which can have an existence independent of the duration of a session, careful consideration must be given to scope rules in an OSCL.

3.7 Control structures

A session is a series of statements, sections of which may need to be repeated or even skipped if certain conditions occur. There are three distinct types of language construct for controlling such things:-

(i) Conditional

This type of construct is used to decide whether or not a particular action should be executed, or to select an action from two or more alternatives. For example, a user may wish to test if a particular program has successfully created the correct input files for a second program before executing the second program, or if the first program has aborted because of an error, the user may wish to execute one of several recovery programs depending on the type of error. The ALGOL-60 IF-THEN-ELSE statement and the PASCAL CASE statement provide conditional control.

(ii) Repetion

This type of control is required when a user wants to repeat an action until some terminating condition is met, such as a counter reaching a limiting value. For example, a user may wish to repeatedly execute a program, using as input a number of files of test data held in an array. The ALGOL-60 FOR-STEP-UNTIL statement provides repetitive control.

(iii) Unconditional transfer

This type of control is required when it is necessary to continue execution from another point in the session. For example, in the middle of a loop a condition may arise which requires a branch to the end of the loop. The ALGOL-60 GOTO statement allows unconditional

transfer control.

A standard OSCL should provide for all three types of control.

3.8 General principles

The preceeding sections outlined the features which a standard OSCL should contain. It is felt however, that if a language doesn't follow certain general design principles, users may still find it difficult or inconvenient to use. The principles which are felt to apply are:-

(i) Simplicity

A language should appear simple to users, allowing them to express their desired algorithm in a natural manner. It should be easy to learn and easy to understand. Statements which could be construed as to have more than one meaning should be avoided. Similarly, constructs should not be error prone. That is, a construct should be sufficiently distinct so that a slight misspelling doesn't transform it into a completely different construct. Allowing synonyms and abbreviations for commands also aids simplicity. Another aspect of simplicity is that of economy of concepts. The language should be designed so that a small number of easily learnt concepts provide the user with powerful facilities.

(ii) Uniformity

A language should be uniform so that there are as few special cases as possible. Basically this means that similar things should be done in similar ways. For example, procedure calls and process initiations should have similar, if not identical, syntax. The importance of this principle can't be overstressed. Language rules such as "i before e

except after c (most of the time)" should always be avoided; they will only cause confusion. The converse of this rule should also be kept in mind; that is distinct features should have distinct rules.

(iii) Self documenting

A program should be as self explanatory as possible. This implies that language symbols should, as far as possible, be in the user's own daily language (e.g. English) and that the use of arbitrary symbols (e.g. //) should be minimised. A free format input, allowing comments, would also help in this respect. System responses and error messages should be as meaningful as possible and tailored to the user's understanding of the system. A 'HELP' or 'EXPLAIN' system should be available to guide inexperienced users.

This list of design principles is not entirely complete and other language designers would emphasize other principles. However, these principles, especially simplicity and uniformity, played an important part in the design of NCL.

CHAPTER 4

A SURVEY OF OTHER WORK

A camel is a horse planned by a committee.

Vogue

July 1958

Moth: They have been at a great feast of languages and stolen the scraps.

Costard: O, they have lived long in the alms basket of words.

William Shakespeare

Loves Labour Lost.

Act V Scene 1

This chapter covers a survey of other work in the field of OSCL standardisation. The control language team had already conducted a survey but they were only searching for a suitable block structures language and had therefore considered only WFL and SCL. The aim of this survey is not to find a suitable language but rather to determine what other researchers have done to solve the control language problem and to evaluate any proposed standard OSCLs.

4.1 Manufacturer supplied OSCLs

Historically, the major development of OSCLs has always been done by computer manufacturers, rather than programming language designers. As operating systems became more and more complex, so did the command languages provided for them. An excellent paper by D. W. Barron and I. R. Jackson and another by C. B. Mason cover the development of OSCLs from genesis to the present day [Barron & Jackson 72], [Mason 77].

Unfortunately "since existing job control languages were, to all appearances, knocked together in a rough and ready way at the same time that the operating system was being developed, the facilities of the system and the way they are described in the language tend to be inextricably confused." [Barron & Jackson 72]. This has lead to the fact that most modern OSCLs do not qualify as candidates for standardisation. For example, the infamous IBM OS/360 JCL is very cryptic, requires huge manuals and is probably a major deterrent to many would-be users!

In the last paragraph the word 'most' was used advisedly as two, recently developed OSCLs, WFL and SCL are much better than their predecessors.

4.1.1 WFL

Work Flow Language (WFL) developed by Burroughs Corporation for their B7000/B6000 range of computers, is a good, high-level OSCL [Cowan 75] [Burroughs 77]. It is block structured and provides ALGOL60 like control structures; specifically IF-THEN-ELSE, DO-UNTIL, WHILE-DO and GOTO statements. It also provides an ON statement for testing when a process fails or when the complete job is restarted after a system crash, and a WAIT statement which may be used to cause the session to wait for a specified condition to become true.

WFL supports integer, real,boolean and string data types; variables of these types may be declared and expressions may be assigned to them. Files and process environments (tasks, in Burroughs terminology) may also be declared. These are treated as structured objects, and the user can assign values to their attributes. An environment, which is attached to a process at initiation, can be used to monitor and control the process.

Procedures, with parameters, may be declared. Procedures and normal processes may be initiated sequentially or concurrently. Also provided is a number of command statements, each starting with a special keyword, which allow manipulation of the users library of files and other housekeeping tasks, such as changing the user's password.

The interactive form of WFL, called CANDE (Command AND Edit), is completely different from the batch form. In CANDE there are no blocks, control structures, declarations and procedures. Many of the basic command statements have different forms. For example, in CANDE the COMPILE command is followed by the name of the source file whereas in WFL it is followed by the name of the object file!

This differentiation between WFL and CANDE is felt to be a serious disadvantage of WFL, requiring users to learn two different languages for essentially identical tasks.

4.1.2 SCL

ICLs 2900 series System Command Language (SCL) was designed to be easily divisible into subsets, provide access to all system functions and be user modifiable [Brunt & Tuffs 76], [ICL 74(1), 74(2), 75].

SCL is block structured and provides control structures based on ALGOL68; specifically IF-THEN-ELSE-FI, FOR-WHILE-REPEAT and an ON statement. A GOTO statement is provided. It allows declaration and manipulation of integer, boolean and string variables and arrays of the same.

Procedures, with parameters, may be declared and a large number of standard procedures for complex tasks, are provided. Each parameter of a procedure may be defined to have a keyword, which is used to identify the parameter when the procedure is called, and a default value. This is an excellent facility, reducing the amount of non-essential information the user is forced to enter.

Complete session may be initiated concurrently and an event mechanism is provided for synchronisation purposes.

A major fault of SCL is that most things are done with system procedures and often in unorthogonal ways (the manual describing these procedures is over 8 centimeters thick!).

4.2 OSCL study groups

In recent years there have been several studies made of OSCLs by noncommercial standardisation groups; partly as a reaction against manufacturers' efforts and partly because of growing realisation that standardisation would bring many benefits These studies, up to 1977, are well documented by P. H. Enslow and C. B. Mason [Enslow 75], [Mason 77]. The major groups involved in these studies are as follows:-

(i) American National Stndards Institute, Standards Planning and Requirements Committee OSCL Study Group. (ANSI/X3/SPARC/OSCL)

(ii) Conference on Data Systems Languages OSCL Task Group (CODASYL OSCLTG)

(iii) Dutch Job Control Language Committee (Dutch JCL group)

(iv) US Federal Information Processing Standards (FIPS)

(v) IEEE Computer Society Technical Committee on OSs (IEEE TCOS)

(vi) Association for Computing Machinery Special Interest Group on OSs (ACM SIGOPS)

(vii) International Federation for Information Processing, Technical Committee 2 Working Group 2.7 (IFIP TC2 WG 2.7)

(viii) British Computer Society Job Control Language Working Group (BCS JCL WG)

(ix) Code Inc: Standard Job Control Language (CODE: SJCL)

(x) IBM Scientific Users Group (SHARE)

(xi) ARPANET Users Interest Working Group: Common Command Language (ARPANET USING: CCL)

(xii) National Bureau of Standards (NBS)

Of these groups FIPS, IEEE TCOS, CODE: SJCL, SHARE and NBS have, for various reasons, stopped working on the OSCL problem without producing anything significant; ACM and IFIP are basically acting as coordinating bodies (holding conferences, publishing papers etc); and ANSI/X3/SPARC/OSCL, CODASYL OSCLTG, the Dutch group, USING: CCL, and BCS JCL WG have yet to produce lenguage designs (they have all "drawn back from the mammoth task of actually developing 'the' standard OSCL." [Mason 77]).

4.3 Individual Efforts

Several languages have been developed by small teams or individual workers, as opposed to large standardisation groups and committees.

4.3.1 UNIQUE

I. A. Newman of Loughborough University, England has worked on the development of UNIQUE [Newman 75,78]. It is a high-level machine independent command language and has been implemented on a variety of machines. UNIQUE is command oriented. A job is composed of a set of activities, each of which specifies a complete single task such as the execution of a FORTRAN program. Four types of activity are specified; system enquiries, program execution, file library manipulation and interactive execution. The user is forced to keep these activities separate.

Each activity is split into a number of phases, which specify the logical steps of the activity, and each phase is split into a number of command statements, called directives. Each directive is a command

name followed by a number of parameters, each of which is a keyword followed by the value to be used; defaults may be supplied. The first phases of an activity specify the computer resources which may be used.

The control structures in UNIQUE are very primitive. Each phase produces a result which can be tested by the phase following it using the ACTION command and a branch made accordingly.

Variables may not be declared; the only objects which UNIQUE manipulates are files and programs. But even this is restricted; for example there are no facilities for manipulating the titles of files and special peripherals may not be defined.

Procedures are not provided and, although concurrent execution is allowed, a synchronisation mechanism is not provided.

D. Rayner has made a detailed evaluation of UNIQUE (in comparison with ABLE and GCL) [Rayner 75], and points out that, amongst other things, UNIQUE only caters for 80-90% of user requirements. Users, via a *drop through* mode, must use the host OSCL for the remainder of their requirements. Also, UNIQUE is translated into the host OSCL which, Rayner argues, introduces unacceptable time overheads.

4.3.2 ABLE

I. T. Parsons of Bristol University has developed ABLE (A Better Language Experiment) [Parsons 75] based on EULER [Wirth & Weber 66], a programming language developed from ALGOL60.

ABLE may not be used interactively. It provides type-free variables ranging over integers, logicals, strings, lists, procedures and references. It is particularly oriented towards lists and strings.

Control structures are provided in the form of WHILE and FOR loops, plus conditional statements including IF and CASE (no GOTO is provided). It is block structured and allows nested declarations.

A procedure facility, with parameters, is provided and procedures can be saved in the users library. Processes may by executed concurrently although no synchronisation mechanism is provided.

As with UNIQUE, ABLE does not cater for all user requirements and is translated into the host OSCL [Rayner 75]. It may not be used interactively and no file edit facility is provided.

4.3.3 GCL

R. J. Dakin of UKAEAs Culham Laboratory, has produced yet another machine independent command language called a General Control Language (GCL) [Dakin 75(1), 75(2)]. It was designed for satellite systems where some small computer translates GCL into the OSCL of the selected host and has been implemented on three systems.

GCL is function oriented, the functions (value returning procedures) being built upon a set of primitive, machine independent functions. It provides type-free variables ranging over integer, string and list values. It is not block structured and provides only primitive conditional and loop constructs.

The user may define procedures with parameters, which may have keywords and defaults, and may initiate them using the RUN function. Process environments may not be defined and concurrent execution is not provided for. The file handling facilities are rudimentary and no edit facility is provided.

As with UNIQUE and ABLE, GCL does not cater for all user requirements and is translated into the host OSCL [Rayner 75].

4.3.4 JOBOL

L. Moore from Birkbeck College, London, has developed a language called JOBOL aimed at demonstrating the feasibility of transporatable OSCLs [Moore 75]. JOBOL is a simple language that allows files to be handled very easily as basic entities. Strings may also be used but only as literals. It provides a small number of simple standard procedures and allows loops, for which the control variables are files, and conditional statements. A form of ON statement is also provided. It is not block structured.

JOBOL can not be used interactively and concurrent execution is not catered for. Process environments may not be specified.

4.3.5 CCL

Common Command Language (CCL. Distinct from ARPANET USING:CCL) has been developed by J. Madsen and C. Gram of the Technical University of Denmark and has been implemented under IBMs OS/MVT [Madsen 77, 78].

CCL is a high-level command language based on SCL (see section 4.1.2) and containing most SCL facilities, the main exception being a procedure facility, but also providing several extra facilities.

In CCL users can not only make files permanent but also normal variables of type integer, boolean and string. Such variables remain in existence from session to session and may be shared by users. Files are also treated as variables and may be declared and created within a session.

Concurrent execution is not provided for and no break-in facility is defined.

4.3.6 ANON

R. T. Tomlinson, of the Texas A&M University, has developed an anonymous language, henceforth called ANON for the purposes of discussion [Tomlinson 76]. ANON is based on ALGOL60 and has been designed in a machine independent manner around a set of primitive operators.

ANON is block structured and each block may be labelled. A block may be called from another block or initiated concurrently. EXIT may be used to terminate a block and return a value from it via the label. A WAIT statement can be used to cause the program to wait until the specified, asynchronously running blocks have terminated.

DO-WHILE and IF statements are provided and also an ESCAPE statement, which may be used to terminate a DO loop.

Files may be declared and bound to a logical name as used by a program. Files may be copied and removed from the user's library. ANON does not provide for variables such as integers or strings, and process environments may not be specified. It may not be used interactively.

4.3.7 FOSIL

G. N. Baird, of the US Navy, has developed Fredettes Operating System Interface Language (FOSIL) which was designed to provide an interface between "a knowledgeable user and more than one OS" for the accomplishment of data processing functions [Baird 75].

A FOSIL program is comprised of calls on the six functions provided. Each function has a number of parameters which must be specified. The functions provided are:-

(i) introduce a session specifying the user, accounting information and the resources the session may use.
(ii) initiate a compile
(iii) initiate a process
(iv) input a file of data
(v) describe a file that is used by the other functions or by programs
(vi) terminate a session

FOSIL is intended for COBOL users and is limited in scope accordingly. It does not provide a default mechanism, variables, procedures or allow concurrent processing. There are no control structures, all function calls being executed in sequence.

CHAPTER 5

THE DEVELOPMENT OF NCL

Thou art weighed in the balances, and art found wanting.

The Bible, Daniel V. 27.

This chapter will cover the development of NCL from SCL. At the time SCL was chosen, it was realised that it had certain faults. The process of correcting these faults and generally 'improving' the language, was long and involved; many minute details being deliberated upon at length and several interim language syntaxes being produced. We will concentrate here on the major additions and alterations to SCL. Many of the finer details, for the sake of clarity, have been omitted. The complete syntax and semantics of NCL are to be found in Appendices A to D.

5.1 Sessions

In SCL the major unit of work is called a job and is delimited by calls on the standard macros JOB and ENDJOB (**** may be used instead of ENDJOB). An interactive job is called a session and is delimited by calls on LOGIN and LOGOUT.

It is felt that such differentiation between batch and interactive work is pointless. In NCL the major unit of work is called a session, whether batch or interactive, and is delimited by calls on the standard procedures HELLO and GOODBYE (see section A3).

In an interactive session statements are executed as soon as they are completly entered. This means that the staements within a structured statement, say a block, are not executed until the statement is complete. For example

```
  :
BEGIN
     X:= 5;
     IF Y = 3 THEN          } These statements are not executed until
          myprog            } the END is reached.
     FI
END;
  :
```

5.1.1 User identification

No user would like to be charged for anothers' work and thus, obviously, a users identification must be unique. A user cannot, therefore, use his own name for identification. This is solved by providing a *usercode* system where each user is issued with a unique, alphanumeric code which he must enter at log-on.

In SCL a user enters his usercode, and session title, as one parameter of the log-on procedure call. For example

```
JOB(MAS123JENKINS.FIXIT)
```

In NCL the only change is to enter the session title as a separate parameter. For example

```
HELLO(MAS123JENKINS, FIXIT)
```

With a network the question arises as to whether a user can have one usercode for the whole system or must, alternatively, have a separate usercode at every site he wants to do work. Making this decision is actually the task of the project management team and is beyond the scope of this thesis. However, recognising that the decision is arbitrary and may vary from network to network, it was felt that NCL should cater for the worst-case situation. That is, NCL should assume that users have a separate usercode for every site. This means that when a user initiates processes at another site he must specify his usercode for that site. In NCL this is done via the USING phrase of the AT clause (see section 5.2.1). For example

```
AT VICTORIA USING VIC456JENKINS
    FILEUPDATE;
```

will cause the program FILEUPDATE to be executed at the site called VICTORIA using the usercode VIC456JENKINS. For any network where users have one usercode for the whole network the implementation of NCL could

either omit the USING phrase or make it optional.

5.1.2 A personalised interface

Having identified the user the system should endeavour to provide an interface tailored to suit the users needs and understanding of the system.

SCL caters for this by providing users with *profiles*. A profile sets up the environment in which the session will run. That is, it specifies the computational resources, e.g. CPU time, that the session may use. It also specifies a set of statements, called the *outer* SCL, which is exectuted before the session starts. The outer SCL performs a variety of set-up functions including the declaration of various global variables and WHENEVER statements (see section 5.7) to handle completion codes.

Users are divided into classes depending upon the computing resources they are likely to consume and a standard profile is provided for each class of user. At log-on a user may select, from a restricted set of profiles, the one he wants to use, otherwise the standard profile is automatically used.

With NCL a study of the concept of profiles led to the more general concept of *abstract machines*. An abstract machine (AM) is an interface which allows the user to ignore the finer details of the machine it is running on. Am AM is defined by four things:-

(i) A command language which defines how the user may create and manipulate objects in the machine.

(ii) A response language which defines how the AM will make responses to the user.

(iii) A set of standard objects (e.g. files and standard procedures) which the user of the AM may access. These, in effect, specify the initial state of the AM.

(iv) A set of values which define what computing resources the AM may consume.

For example, an operating system is an abstract machine. If users want, they can ignore the fact that the machine they are really using (the real machine) understands certain binary codes and has a word oriented memory and pretend that they are using a machine which can directly understand the OSCL (and thus has an instruction set which includes all OS facilities, i.e. instructions such as COPY, COMPILE, EXECUTE and the like) and makes replies in the OSRL and allows them access to objects called files.

If we examine the roll of UNIFACE we can see that it is an AM itself; an AM which hides the finer details of the OSs on which it runs. It has access to all the files the OS maintains but presents a standardised view of the attributes of files. It may access all the OS facilities but only allows them to be used via NCL.

In NCL it was decided to provide each user with his own abstract machine, called the *user abstract machine* (UAM). This machine acts as a window on the facilities provided by UNIFACE. It specifies a restricted set of the standard procedures which the user may access. It only allows the user to access the files which he owns, or those to which their owners

have granted him access. It indicates the computing resources the user may consume overall and how much he may consume in any one session. It also indicates to the NRL handler the level of the users computing competence so that system responses may be tailored to suit the user. In UNIFACE the control language part is the same for all UAMs, i.e. NCL.

UNIFACE provides all of the facilities needed to support all the different UAMs. When a user logs-on, UNIFACE provides the correct UAM which defines the user's environment within that machine.

The image we have now is one of machines within machines, as in diagram 5.1.

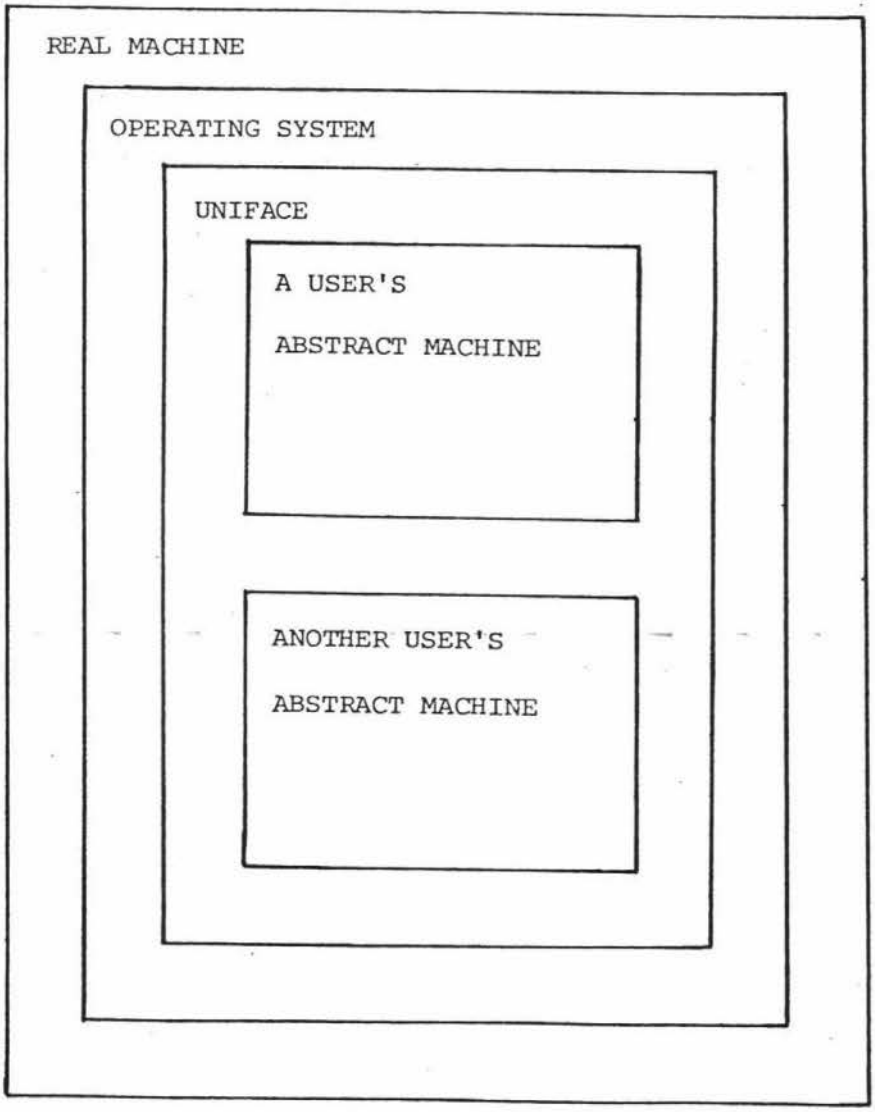


Diagram 5.1 Nested Abstract Machines

Or from a user's point a view, an hierarchy of machines as in diagram 5.2.

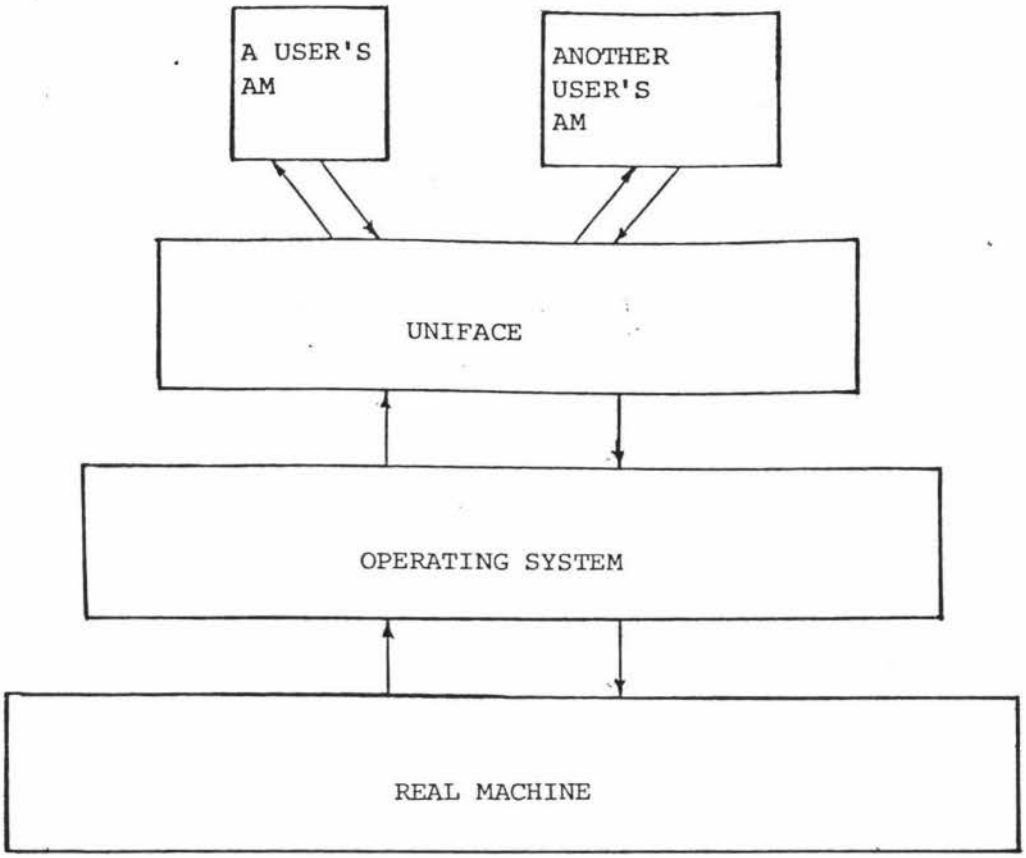


Diagram 5.2 The abstract machine hierarchy

5.1.3 Simultaneous command execution

SCL provides a RUNJOB macro which allows the user to initiate complete batch sessions which are run concurrently with the user's main session (see section 5.2.3). NCL also provides for this via the standard procedure SUBMIT (see appendix D). However, it was felt that it should be possible to have commands executed simultaneously without starting completely new sessions. For example, a user may wish to find out how a process he has initiated is getting on by executing the standard procedure REPORT within the current session.

In NCL there are two statement streams per terminal. At any particular time a terminal is connected to one of the streams and anything input at that terminal goes to that stream. If the stream is already doing something the input is queued. Entering a ? will cause the terminal to switch to the other stream. The user can then enter statements which are executed simultaneously with the other stream. For simplicity only two streams are provided and thus only two commands may be executed simultaneously.

5.1.4 Session breakin

SCL provides for session breakin via some hardware feature of the user's terminal, e.g. a breakin button. After breakin, which suspends the current process, the user may enter a limited set of SCL statements plus calls on the macros QUIT and CONTINUE.

It was felt that the system should not rely on the hardware features of terminals. Considering the problem it was realised that the dual stream system, described in the previous section, already provided for a form of breakin. All that was needed was to add three special commands (BREAK, CONTINUE and FINISH) which can be used to affect the process being executed by the other stream (see section A3.3). BREAK causes the other process to be suspended and thus entering ?BREAK is equivalent to hitting a breakin button in SCL.

5.2 Processes

5.2.1 Process initiation

SCL provides a simple mechanism for process initiation; a program is stored in a code file and may be initiated by merely naming the code file. For example

```
MYPROGRAM;
```

will cause the program stored in the file MYPROGRAM to be initiated. In NCL this remains unchanged

To allow users to specify where programs are to be executed an AT clause was added to NCL (see section A3.4) which when added to the front of a procedure call or a block of statements, specifies where it should be executed. For example

```
AT VICTORIA USING VIC456JENKINS
    PAYROLL;
```

will initiate a *subsidiary* session at the site called VICTORIA, in which the program PAYROLL is executed. This session remains in force only as long as it takes to execute PAYROLL.

5.2.2 Process environments

SCL provides environments for complete sessions via profiles (see section 5.1.2). It does not, however, provide environments for individual processes. It was felt that NCL should allow the user to do this and ENVIRONMENT was introduced as a basic variable type (see section A4.6.3). For example

```
ENVIRONMENT VAR quickie;
```

declares an environment called quickie.

An environment performs two functions; it allows the user to restrict the resources used by a process and to monitor the progress of a process while it is running. Accordingly, an environment is a structured object composed of two sets of attributes; one to specify resource limits and the other to hold status information about each resource. For example CPUTIME and CPULIMIT specify the processing time used by the process and the maximum time it may use, respectively. Thus

```
quickie.CPULIMIT:= 5;
```

sets the CPULIMIT attribute to five seconds and any process using quickie as its environment will be limited to five seconds processing time. While the process is executing the user can access quickie.CPUTIME to find out how much time the process has used. (The other attributes of environments are described in Appendix C.)

Default values are specified for all resouce limit attributes. If such an attribute is not explicitly given a value the default value is used. Defaults are user dependent and specified in the UAM.

There must be some means by which a user can connect together a process and the environment to be used to control and monitor it. Furthermore, if two processes, running concurrently (see section 5.2.3), could both use the same environment, say quickie, then accessing status attributes such as CPUTIME, poses the problem as to which process is being refered to. Thus the combination of process and environment must be unique. That is any one environment can be attached only one process at any one time.

There are two possible ways to specify the connection between a process and an environment.

(i) Attach the process to the environment. For example,

```
quickie.PROCESS:= myprog;
```

If myprog was initiated, a search of the existing environments would be made to determine which one had been specified. If no environment was specified then the default values would be used. Using this method, it is possible that one process could become attached to several environments. This could be avoided by taking note of which was the latest assigned environment and using that one. This would also mean that changing the environment used for a process would consist merely of attaching the process to the desired environment, thereby overriding the earlier attachment.

(ii) Attach the environment to the process. This itself cand be done in two ways; before initiation, for example

```
myprog.ENVIRONMENT:= quickie;
```

or at initiation, for example

```
myprog[quickie];
```

(as used in WFL). Neither of these two methods require a search of the environments. Furthermore, because only one environment may be attached to a process the system does not have to remember which is the latest assigned environment. However, with this method, it is possible that one environment could become attached to several processes. If a process tries to use an environment already being used by another process, it will not be initiated and a run-time error will occur.

It was felt that method (ii) involved fewer implementation problems and also seemed the more natural way to do things. The choice now had to be made between attachment before initiation or, the WFL way, at initiation.

Consider the following assignment statement, where P1 and P2 are processes returning values:-

```
X:= P1 + P2;
```

Using the WFL way would require something like

```
X:= P1[E1] + P2[E2];
```

or

```
X:= P1 + P2 [E1,E2];
```

where E1 and E2 are environments.
This was felt to be messy and further more the arbitrary symbols [and] don't convey the meaning of what is being done.

Considering these points it was decided to use attachment before initiation. To cater for this the attribute ENVIRONMENT was added to files. Naturally the attribute can only be used meaningfully if the file is a code file and can thus become a process (see section 5.2.1).

A user may have a particular environment which he tends to use often. To save him defining it in every session it is possible to make an environment permanent, that is make it have an existence independent of the session in which it is found. This is done via the standard procedure SAVE (see Appendix D). Environments can also be copied to other nodes and saved definitions can be deleted; again this is done via the appropriate standard procedures.

5.2.3 Concurrency

As already mentioned in section 5.1.3, SCL provides a RUNJOB standard macro which can be used to initiate an asynchronous session. The use of RUNJOB is limited; it can only be used to initiate complete sessions and

the session source must be stored in a file ready for use. To remedy this the *parallel group* construct was added (see section A6.2.4). For example,

```
PAR
     x:= 5;
     prog1;          group 1

    ,prog2;
     prog3;          group 2

RAP;
```

will cause the statments in group 1 to be executed concurrently with those in group 2 (which group starts first is undefined).

For communication between asynchronous sessions SCL provides an event and interrupt mechanism very similar to that in Burroughs B6700/B7700 Extended Algol. However to create and manipulate events required approximately fifteen standard macros. This was not felt to be very good and it was decided to investigate alternative mechanisms.

5.2.3.1 Events

Simplifying the SCL mechanism was the first thing attempted. The aim was to produce a mechanism that was easy to understand. The result was a very simple event mechanism, working as follows.

Events can be declared and initialised to either HAPPENED or NOT HAPPENED. For example,

```
EVENT VAR E:= NOT HAPPENED;
```

At any point a process can change the value of an event via an assignment. For example,

```
E:= HAPPENED;
```

A WAIT statement can be used to cause a process to wait for an event to

reach a desired state. For example

WAIT E

causes the program to wait until the event E becomes HAPPENED and

WAIT NOT E

causes the program to wait until the event E becomes NOT HAPPENED.

Events can be used to guard a critical section which accesses a shared resource. For example,

```
WAIT NOT LOCKED;              @ A @
LOCKED:= HAPPENED;            @ B @
     :                      }
     :                      }  critical section
     :                      }
LOCKED:= NOT HAPPENED;        @ C @
```

If the event LOCKED has the value HAPPENED statement A causes the process to wait until another process assigns LOCKED the value NOT HAPPENED, otherwise the process continues. A synchronisation controller ensures that only one of the processes waiting on the event will be unblocked when statement C is executed. Statement B ensures that only one process will execute the critical section at one time.

As it was soon realised, this mechanism is very weak, mainly because it is so simple. Consider the previous example. A process may execute A and, because LOCKED = HAPPENED, be forced to wait. A second process then executes C and the first becomes unblocked but before it can get to execute B a third process executes A and, finding that LOCKED = NOT HAPPENED, continues. Thus two process are allowed to execute the critical section at once. It was obvious that this mechanism should be discarded..

5.2.3.2 Semaphores

The next mechanism considered was Dijkstra's semaphores [Dijkstra 68] which work as follows.

A semaphore can be used, like an event, to guard some critical section. Variables of type SEMAPHORE may be declared and initialised to a non-negative integer value. For example

```
SEMAPHORE VAR double:= 2;
```

The value of a semaphore indicates the number of processes which can simultaneously execute the critical section. After declaration the value of a semaphore can only be changed via the use of two standard procedures (called synchronisation *primitives*) which Dijkstra named P and V.

Before executing a critical section a process, typically, must use P to *get* the semaphore guarding it. If the semaphore is non-zero then it is decremented by 1 and the process may continue. Otherwise the process is forced to wait until the semaphore becomes non-zero, via some other process leaving the critical section and using V to *free* it and thereby incrementing its value by 1. For example,

```
P(double);
    :  }
    :  }  critical section
V(double);
```

Will allow up to two processes to execute the critical section.

While investigating semaphores it was discovered that T. Agerwala had developed two primitives he has called PE and VE, which offer greater flexibility than P and V [Agerwala 77]

Consider a system where several processes are all accessing a number of semaphores. Whereas P and V operate on only one semaphore at a time, PE and VE operate on sets of semaphores and can therefore be used to force a process to wait for the occurence of a particular state of the system as defined by a particular configuration of semaphore values.

PE operates on two sets of semaphores, the second of which may be empty. The first set specifies those semaphores which the process wants and the second set specifies those which may block the process getting those in the first set. Consider the following call on PE,

```
PE([x,y,z],[p,q]);
```

where [and] are used as set delimiters as in PASCAL. A process executing the above will be blocked if any of the semaphores x,y and z have values equal to zero *or* if any of the semaphores p and q have values greater than zero.

Agerwala shows that PE and VE can be used to solve, in a simple manner, synchronisation problems that are diffuclt to solve with other primitives. The following example is a solution, described by Agerwala, of a problem proposed by P. J. Courtois et al [Courtois et al 71]. The problem is that there are two types of process, namely readers and writers, which to use a resource. If a reader is using the resource other readers may also use it, however only a single writer may use the resource at once. Furthermore, once a writer requests access to the resource it should be allowed to write as soon as possible. The solution to this problem could be used, for example, for the airline reservation system decribed in section 3.2.3. The solution, using PE and VE, is:-

```
SEMAPHORE VAR
     W:= 0,              @ the number of active writers @
     R:= 0,              @ the number of active readers @
     M:= 1;              @ for mutual exclusion of writers @

the reader:
LOOP
     PE([M],[W]);        @ when W=0 and M=1 decrement M and continue. 1@
     VE([M,R]);          @ increment M and R. 2@
     READ;               @ critical section. 3@
     PE([R],[]);         @ when R>0 decrement R and continue. 4@
POOL

the writer:
LOOP
     VE([W]);            @ increment W. 5@
     PE([M],[R]);        @ when R=0 and M=1 decrement M and continue. 6@
     WRITE;              @ critical section. 7@
     VE([M]);            @ increment M. 8@
     PE([W],[]);         @ decrement W. 9@
POOL
```

The semaphore W is used to ensure that no reader will try to read if a writer wants to write. Thus a writer uses statement 5 to indicate it wants to write. This is picked up by a reader in statement 1 where, if W is non-zero the reader is blocked from getting M. The semaphore R is used to stop writers executing if there are any active readers. Thus when a reader starts to execute the critical region it increments R via statement 2. This is picked up by a writer in statement 6; if R is non-zero it will not continue. Only when all active readers have finished, and executed statement 4, can a writer get beyond statement 6. The semaphore M is used for the mutual exclusion of writers. Thus in statement 6 a writer gets M and any other writer will be forced to wait until it frees M in statement 8. M is also used by readers to ensure that, once a reader has got past statement 1 (there being no writers wanting to write), no writer will get past statement 6 before the reader has had a chance to increment R.

5.2.3.3 Path expressions

Semaphores, although relatively easy to understand, are primitive and if used in an undisciplined manner can lead to deadlock situations. Considering this it was felt that perhaps a higher level synchronisation mechanism should be used. A mechanism devised by R. H. Campbell and A. N. Habermann , and called path expressions, was investigated [Campbell & Habermann 73].

A path expression names the procedures whose execution by processes are to be synchronised and decribes how the synchronisation is to be organised. A path expression is used by the system to create a prologue and an epilogue for each procedure it names. The prologues and epilogues use semaphores to ensure that the synchronisation defined by the path expression is implemented correctly.

A path expression indicates the synchronisation required via four *operators*:-

(i) ; is used to specify sequential action. For example the path expression p;q;r indicates that the procedures p,q and r are to be executed one after another. If a process tries to invoke q, it will be delayed until p has been executed by another process.

(ii) , is used to indicate selection. For example the path expression p,q,r indicates that one of the procedures p,q and r is to selected and the process attempting to execute it is allowed to continue whilst other processes, attempting to execute procedures not selected, are delayed until another selection is made. The selection is made from

amongst those procedures that have been invoked and is done in a random manner.

(iii) To indicate that a path expression may be repeated it is bracketed by the symbols PATH and END. For example, the path expression PATH p END indicates that p may be executed repeatedly by processes, but only one at a time.

(iv) To indicate that a path expression may be used simultaneously by several processes it is bracketed by { and }. For example, the path expression { p } indicates that p may be executed by many processes simultaneously. Once one process begins to execute p others may also do so until all executions of p are complete.

Any procedure can only be specified once in a path expression and can be used in only one path. This restriction means that it is often necessary to give one procedure two or more names by embedding it in dummy procedures.

The following example is a solution, described by Campbell and Habermann, of the readers and writers problem used in the example for semaphores.

```
PATH readattempt END                          @ensures only one read request at@
                                              @a time.                          @
PATH requestread,{requestwrite} END           @write requests may overlap.      @
PATH {openread;read},write END                @reads may overlap but writes     @
                                              @can't.                           @
```

where

```
readattempt = BEGIN requestread END
requestread = BEGIN openread END
requestwrite = BEGIN write END

reader:
LOOP
     READ
POOL

writer:
LOOP
     WRITE
POOL
```

where

```
READ = BEGIN readattempt; read END
WRITE = BEGIN requestwrite END
```

The first path ensures that only one read request may occur at a time. While one process requests reading all others have to wait until they can initiate a read attempt. This ensures that a write request is granted in the second path at the earliest possible moment. The braces in the second path ensure that all write requests are granted as long as writing is still going on. The braces in the third path allow read operations to overlap if no writing is requested at all.

Considering that path expressions were as yet untried in any language and that they seemed more difficult to comprehend than semaphores, it was felt that they should not be used in NCL. Rather the decision was to use semaphores. The primitives PE and VE are called GET and FREE respectively (see Appendix D).

5.3 Procedures

SCL provides a powerful procedure facility (although it calls them macros) with an excellent keyword and default mechanism for parameters. The facility seemed quite adequate although it was felt there were a couple of points which could be improved.

5.3.1 Saving procedures

SCL allows procedures to be stored in files; two for each procedure, a source file and a code file. Storage is accomplished via the two standard procedures INPUT and NEW_SCL. INPUT is used to create the source file and NEW_SCL is used to create the codefile from the source file. It is not possible, having declared a procedure to just save it; the procedure must be reinput using INPUT.

When considering the implementation of procedures it was felt that any unnested procedure should be automatcally stored in files when it is declared. Thus, if these files could be referenced by the user, the standard procedure SAVE could be used to save the files. To cater for this, an unnested procedure declaration was made to implicitly declare two files called CODE and SOURCE which are automatically loaded with the code and source for the procedure.

Thus a procedure declaration causes the creation of a structured object the mode of which could be defined (in PASCALese) as

```
TYPE PROCEDURE =
     RECORD
          SOURCE : FILE;
          CODE   : FILE;
     END;
```

For a procedure P, entering SAVE(P) will cause both files to be saved. Of course, if the user wants to save only the code file, say, then he could enter SAVE(P.CODE) .

5.3.2 Return value modes

In SCL a procedure can return values of all modes, i.e. INT, BOOL, STRING and SUPERSTRING values. The modes available in NCL are more extensive (see section 5.5) including modes such as REAL, SEMAPHORE and FILE. Values of all NCL modes may be returned by procedures in NCL.

5.3.3 The return statement

In SCL the return statement specifies the value to be returned by the procedure and causes the procedure to terminate. The syntax for procedures does not allow (possibly because of an oversight) a return statement to be used within any conditional statement. For example, a statment such as

```
IF x=0
THEN RETURN y;
```

is not allowed. To achieve this would require the following statement

```
IF x=0
THEN GOTO lab;
RETURN y;
lab: ...
```

Furthermore, because the return statement causes the procedure to terminate a user can't specify the value to be returned and then do some other actions before terminating the procedure.

Considering these points it was decided to provide separate statements for the two separate functions of the return statement; that is value returning and procedure exit.

SCLs GOTOs had already been replaced by EXIT statements (see section 5.7) and it was felt that EXIT statements could also be used for procedure exit. The only problem is in supplying a label. This was solved by adding a new element to the procedure structure; a label called BODY. Thus a procedure, P, may be terminated by executing

```
EXIT P.BODY;
```

It was felt that value returning should be done as an assignment to a variable of the appropriate mode. It was therefore decided to add another element to the procedure structure; a variable, of a mode defined in the procedure header, called RESULT. This can be assigned a value within the body of the procedure. For example,

```
P.RESULT:= 500;
```

will cause the value 500 to be returned by the procedure P.

Thus a procedure declaration now declares a structure composed of four elements; two files called CODE and SOURCE, a label called BODY and a return value variable called RESULT (see section A4.7).

To recap, the return statement was transformed into an EXIT statement and an assignment statement. These statements were, of course, already part of the allowable staements within a procedure and therefore no extra syntax was needed. Furthermore, they can be used unambiguously within nested procedures. For example,

```
PROC p = INT:
     BEGIN
     PROC q = VOID:
          BEGIN
            .
            .
            .
          IF x = 0
          THEN p.RESULT:= 15;
               EXIT q.BODY;
          FI
            .
            .
            .
          END;
       .
       .
       .
     END;
```

The use of RESULT also provided a simple solution to another problem. Consider the following assignment statement

```
F:= P;
```

where F is a file and P is a code file for a procedure which returns a file. The statement is ambiguous; does it mean "assign the file P to F" or does it mean "execute P and assign the reult to F" ? It was decided that if a user wanted the result of a procedure he should indicate so, explicitly, by following the procedure call with .RESULT. For example,

```
P.RESULT
```

This can be interpreted as an expression, the value of which can be found by executing the procedure P. Thus

```
F:= P;
```

will cause the file P to be assigned to F, whereas

```
F:= P.RESULT;
```

will cause P to be executed and the result to be assigned to F.

This also means that statements for compile-for-syntax and compile-and-go become distinctly different. For example,

```
ALGOL(MYPROG);
```

means compile MYPROG and discard the result, whereas

```
ALGOL(MYPROG).RESULT:
```

means compile MYPROG and execute the result.

5.3.4 Parameters

SCL allows parameters, of all data modes, to be passed either by value or by reference. It also provides two special parameter types called LITERAL and SUPERLITERAL which can be passed STRING and SUPERSTRING literals respectively, without the need for enclosing them in quotes. For example, a macro declared as follows

```
MACRO P IS (LITERAL X)BEGIN ... END;
```

could be called as follows

```
P(ABC);
```

whereas if the parameter had been declared as STRING X then the macro would have to be called as follows

```
P("ABC");
```

If the value which must be passed to a literal parameter is held in a string variable or is a string expression then the word VAL must be placed in front of it. For example, if we have a string variable declared as follows

```
STRING S:= "ABC";
```

then the macro P could be called as follows

```
P(VAL S);
```

The inconvenience of using VAL was felt to outwheigh the convenience of not having to use quotes and LITERAL and SUPERLITERAL parameters are not provided in NCL. Apart from this the SCL parameter mechanisms are retained, except for minor syntactic changes.

5.4 Files and Peripherals

5.4.1 Files

SCL does not treat files as basic objects manipulated by hte OSCL user; all file creation and manipulation is done via a large number of standard procedures. For example, the following statements

```
ASSIGN_FILE(NAME=INPUTFILE, LNAME=MASTERFILE)
NEW_FILE(NAME=TRANSFILE)
ASSIGN_FILE(NAME=TRANSFILE, LNAME=WAGESFILE)
WORKFILE(LNAME=EARNINGS)
PAYROLL
SAVEFILE(NAME=TRANSFILE)
```

cause a program called PAYROLL to be executed using as input two files called MASTERFILE and EARNINGS and producing a file of employees wages which is saved, [ICL 74(1)] page 2.33.

It is felt that files are objects, just like for example integers and strings, which the user manipulates within his environment. Therefore, just as with integers and strings, a user should be able to declare files, manipulate them, pass them as parameters to procedures and so on. For example, the preceding example would be in NCL

```
FILE VAR MASTERFILE:= (TITLE:= "INPUTFILE");
FILE VAR WAGESFILE:=  (TITLE:= "TRANSFILE",
                       NEW:= TRUE);
FILE VAR EARNINGS;
PAYROLL;
SAVE(WAGESFILE);
```

Note that in the last statement the local name of the required file is used.

In NCL a file is treated as a structured object composed of a physical container and a connected logical description. See diagram 5.3.

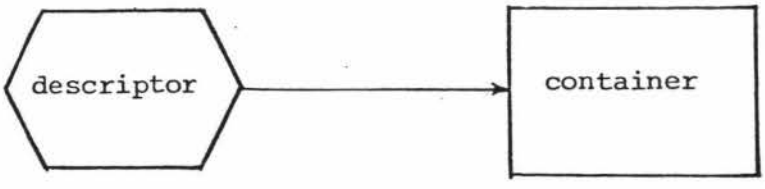


Diagram 5.3 A file.

A file declaration causes the creation of a file description only. Before the described container can be manipulated teh decription must be connected to it. This is called *opening* the file and the reverse, disconnecting them is called *closing* the file.

A file container and a file description both have separate names, called the external and the internal names of a file respectively. A description is known by its internal name within a session and may be used to describe many containers with different external names. The OS knows a container by its external name which is held in the TITLE attribute of its decriptor. For example, the following declaration

```
FILE VAR gunge:= (TITLE:= "FAMILYTREE_PRINTER_CODE");
```

declares a file whose internal name is gunge and external name is FAMILYTREE_PRINTER_CODE. See diagram 5.4.

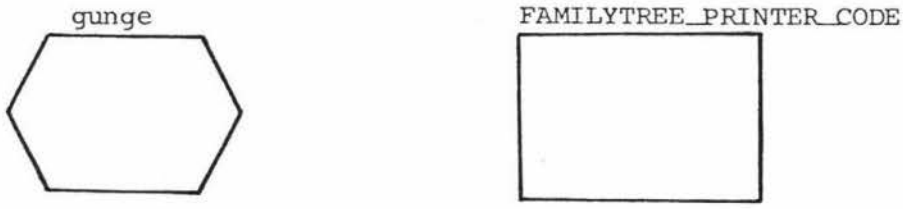


Diagram 5.4 A particular, unopen file.

As the file is not yet open the descriptor and container are not yet connected.

A file is opened by setting the attribute OPEN to TRUE, and closed by setting it to FALSE. For example, the following assignment

```
gunge.OPEN:= TRUE;
```

will cause the system to search for a container called FAMILYTREE_PRINTER_CODE and to connect gunge to it. If such a container cannot be found an error will occur and gunge will not be opened. However if gunge.NEW equals TRUE then opening will cause a new container to be created as described by the other attributes of gunge.

Although the TITLE of a file is regarded as naming it, a file is actually identified by four attributes; SITE, OWNER, TITLE and GENERATION. Every file must have a unique combination of values for these attributes.

The SITE attribute enables users to indicate where a file is located. The OWNER attribute holds the usercode of the user who owns the file. The TITLE attribute holds the external name and the GENERATION attribute is used to distinguish between files with the same SITE, OWNER and TITLE values. For example, the following declaration

```
FILE VAR data:= (SITE:= "VICTORIA",
                 OWNER:= "VIC456JENKINS",
                 TITLE:= "FAMILYTREE_DATA",
                 GENERATION:= 3);
```

declares a file which is generation number three of a file called FAMILYTREE_DATA belonging to user VIC456JENKINS at the site called VICTORIA.

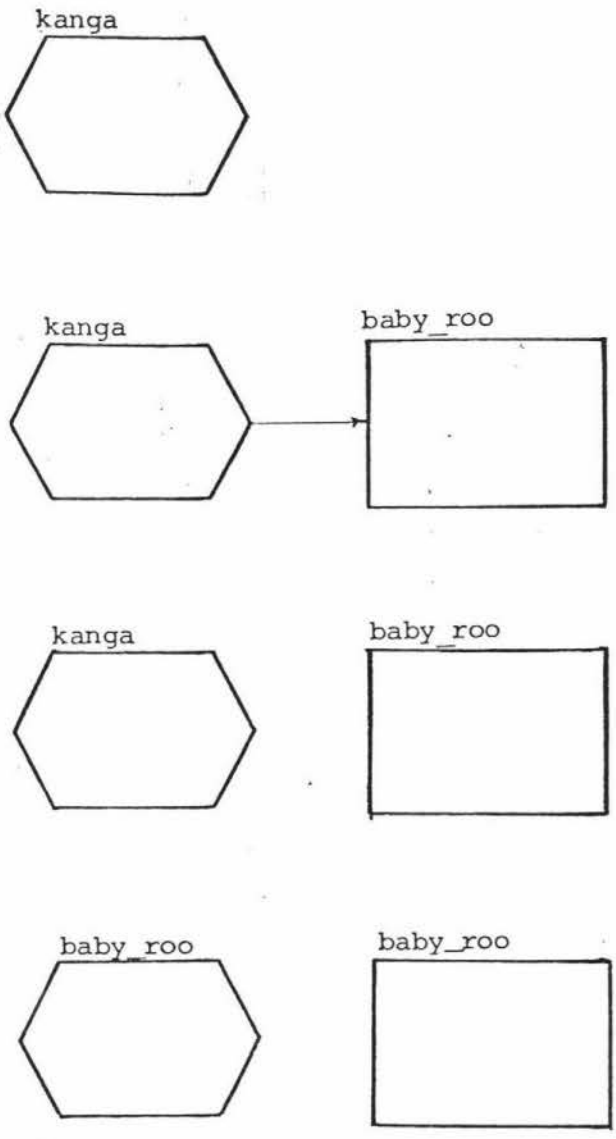
Files may be grouped together in *libraries*, each member of a library having its title start with the name of the library. For example, the files with titles

```
FAMILYTREE_DATA
FAMILYTREE_PRINTER_CODE
FAMILYTREE_PRINTER_SOURCE
```

belong to the library called FAMILYTREE. Furthermore it is possible to have libraries within libraries. Thus in the above example PRINTER is the name of a library within the FAMILYTREE library.

A file may be temporary or permanent. This is specified by the EXISTENCE attribute. For example, if gunge.EXISTENCE equals TEMPORARY then if gunge is closed the file is deleted from the user's environment. On the other hand when gunge is closed, if gunge.EXISTENCE equals PERMANENT, then a global descriptor is created which is a copy of gunge except that its internal name is the same as the file's TITLE. For example,

```
BEGIN
FILE VAR kanga:=
    (DEVICE:= DISK,
     EXISTENCE:= PERMANENT,
     NEW:= TRUE,
     TITLE:= "baby_roo");
  :
  :
kanga.OPEN:= TRUE;
  :
  :
kanga.OPEN:= FALSE;
  :
  :
END;
```



Attaching a file to a process can be done explicitly or implicitly. To do it explicitly the PROCESS attribute must be used. For example, if kanga.PROCESS is assigned the value "myprog" then when myprog is executed, and attempts to use a file called kanga, the NCL definition of kanga will override that of myprog's.

To attach a file to a process implicitly the user merely declares a file with the same name as that used by hte process. When the process attempts to open a file not explicitly provided with an override definition the system must check for implicit attachments.

Standard procedures are supplied for complex file handling tasks, such as copying and editing files; see Appendix D.

5.4.2 Peripherals

The DEVICE attribute of a file specifies which type of I/O device should be used for the file. For example,

```
f.DEVICE:= LINEPRINTER;
```

indicates that the file f should be output to a lineprinter. The actual lineprinter used would be the standard line printer for the node. The user might however want to use a special line printer (e.g. one with a lower case character set). Scl allows the user to define the characteristics of the peripheral to be used for a particular file. This is done via a number of macros which allow the user to create a peripheral definition and attach it to a file.

In NCL peripherals are treated in much the same way as files. Similarly to files, a peripheral is a structured object composed of a physical object and an associated description. Thus in NCL, peripherals may be declared and treated as variables (see section A4.6.2). For example,

```
PERIPHERAL VAR bigone:= (DEVICE:= PLOTTER,
                         HEIGHT:= 1000,
                         WIDTH:= 500);
```

declares a peripheral called bigone which is a plotter with a plotting surface measuring 1000 by 500 millimetres.

The previous considerations about the file attributes SITE, OWNER, TITLE, NEW and EXISTENCE also apply to peripherals. Peripheral definitions may be saved by the user in his directory but as standard peripheral defintions for most system peripherals are available anyhow the user doesn't often need to do so. The definition and saving of peripherals is mainly designed to be used by operators altering the system configuration.

A peripheral may be made a virtual peripheral by setting the attribute VIRTUAL to true. For example,

bigone.VIRTUAL:= TRUE;

Virtual peripherals are simulated by the spooling system of the OS and are useful in situations where several processes wish to use the same peripheral at once. For example, in most systems the site lineprinter is a virtual device; several processes may write to it at once but hte OS spools the output.

A peripheral may be attached to a file via the file attribute PERIPHERAL. For example, if

plotfile.PERIPHERAL = bigone

then whenever a process uses plotfile the peripheral defined by bigone will be used.

5.5 Data types

5.5.1 Basic types

In an OSCL various data types are needed for passing parameters to processes, handling the attributes of structured objects such as files and for specifying the flow of control within a session. The need to pass parameters to processes indicates that the standard OSCL should provide integer, real, boolean and string data types as these are common to many languages.

Integers, booleans and strings are also needed foe attribute handling. Integers are needed for attributes such as MAXRECSIZE and WIDTH; booleans are needed for attributes such as OPEN and NEW and strings are needed for attributes such as OWNER and TITLE.

To specify flow control integers are needed for counting iterations and booleans are needed for testing conditions.

To recap then, an OSCL should provide integer, real, boolean and string data types. SCL doesn't provide for reals. It was felt that NCL should and they were added to the language (see section A4.4). It was also felt desirable to be able to specify the character size of strings. This would enable a user to specify, as a bit string or in hexadecimal, the codes for characters not in the character set of his terminal. For example, 4"C1" is equivalent to "A".

As already mentioned resources (i.e. files, peripherals and environments) are considered as data types in NCL. They can be declared and treated in much the same way as the other data types. In particular they can be passed as parameters and also used in data structures.

5.5.2 Data structures

Many programming languages allow structures of variables, such as arrays or compound structures. Arrays are just as useful in OSCLs. For example, one could set up an array of values and via an iterative loop call up some process pass the values to it one at a time as parameters.

Another type of data structure not commonly found in programming languages is the queue. In OSCLs queues are useful for communication between concurrent processes (see section 5.2.3). For example, two processes could share a queue, one adding values to the tail and the other removing values from the head.

As it is commonplace for users to collect files together in libraries, some form of structure for handling libraries of files is necessary.

SCL provides *array of integer*, *array of boolean* and SUPERSTRING (which is equivalent to *flexible array of string*) modes.

In NCL *array of real* was added and SUPERSTRING was transformed into *array of string*. Also added was *array of resource* (e.g. array of file).

It was felt that queues should be provided by NCL, and therefore a mechanism to support them was investigated. The initial proposal was for a queue to be a flexible structure of objects (of the same mode) each with an allocated priority; queues would be ordered on priority and access would be allowed at the head or tail and also on the basis of priority. It was realised however, that combining priority queues with FIFO queues introduced problems (e.g. what happens if a high priority element is added explicitly to the tail of the queue?). Thus the final decision was not to use priorities and only allow access at the head or tail (see section A4.8.4).

An important characteristic of libraries of files is that all files in a library are unique; that is, they all have a unique combination of values for the attributes SITE, OWNER, TITLE and GENERATION (see section 5.4.1). To handle libraries a data structure is needed which will ensure that all values held in it are unique. In this respect arrays are not adequate as it is quite allowable for two elements of an array to hold the same value. Fortunately, there is a well defined mathematical concept which covers this situation quite adequately; that of *sets*. Mathematically a set is just a collection of distinct objects. Thus a set mechanism was introduced to NCL (see section A4.8.3).

As users often want to access libraries in alphabetical order, it was decided that sets would be ordered and to allow access on the basis of this order. For examples see section A4.8.3. This decision also means that the efficiency of a uniqueness test could be improved.

The main use for sets is for libraries of files. However, it was felt that the user may wish to use sets of other data types. For example, a user may wish to create a set of unique file titles and would therefore need a set of strings. Thus it is possible to create sets of all basic data types.

All basic data types have literals. For example "ABC" is a string literal. The same is true for data structures. For example, [1,2+6, x*y] is a literal for an array, set or queue of integers; the type of structure is determined by the context. For example, in the following declaration

```
SET OF STRING SS:= ["ONE","TWO","THREE"];
```

the literal defines a set of three strings which is assigned to the variable SS.

Literals can also be used to add elements to queues and sets. For example, where Q is a queue of integers

```
Q:= Q + [1];
```

adds the value 1 to the tail of the queue and

```
Q:= [1,2] + Q;
```

adds the values 1 and 2 to the head of the queue.

It is often necessary not only to access the value of an element of a set or queue but also to remove the element from the structure entirely. To allow this a builtin procedure, REMOVE, was introduced (see Appendix D).

REMOVE returns the value of a given element and removes it from the structure at the same time.

At this point it was noted that although sets and queues were flexible structures (i.e. the number of elements comprising them may increase or decrease) arrays were not. It was felt that to be orthogonal arrays should be changed to conform. Thus arrays are flexible, linear structures of objects all of the same mode.

5.6 Blocks and scope

SCL is a block-structured language; the scope of any variable being the block in which it is declared, plus any inner blocks. In addition, with procedures only, *external* and *residual* declarations may be made. An external declaration defines an object global to the procedure. This is called *importing* the object. A residual declaration defines an object which will remain in existence after the procedure ends. This is called *exporting* the object.

An external declaration causes a search to be made, outwards through the surrounding scopes to find the object to be imported. Thus an object can be imported through several nested blocks. On the other hand a residual declaration causes the object to be exported only one level, that is to the next outer block.

It was felt that importing objects was a good idea, increasing security by forcing users to declare what objects, global to the block, may be used within it. However no reason could be found why importing should be restricted to procedure bodies. It was suggested that NCL

support a complete import system, where all objects used in a block must either be local (i.e. declared there) or be explicitly imported. R. D. Tennent decribes such a system in [Tennent 77]. Thus in NCL the scope of any variable is hte block in which it is declared, excluding any inner blocks. To refer to a variable inside an inner block an external declaration must appear within that block before such a reference. (In NCL the word GLOBAL is used instead of EXT). For example

```
INT VAR X;
  :
  :
BEGIN
     GLOBAL INT VAR X;
     X:= 500;
       :
       :
END;
```

It was felt however that certain objects (e.g. standard procedures) are usually required throughout a session and that to be forced to import them into every block would be tiresome. Thus all permanent resource objects are automatically imported into any block; thay are global to the session and effectively available throughout it. This is similar to the effect of the *pervasive* declaration in EUCLID [Popek et al 77].

An AT clause causes a subsidiary session to be initiated at a remote site. The objects in the parent session are global to the subsidiary session and it may access them if it imports them. For example,

```
INT VAR X;
  :
  :
AT OTAGO USING WHATSHISNAME
     BEGIN
       :
       :
     GLOBAL INT VAR X;
     X:= 31;
       :
       :
     END;
```

Naturally, any permanent resource objects held at the remote site are also available to the subsidiary session.

It is necessary to be able to make newly created objects permanent. The residual declaration does not do this, only exporting the variable to the next outer block. Thus residual declarations were removed from NCL and replaced by hte standard procedure SAVE (see section A4.2) which exports resource objects to hte outermost level and makes them permanent.

5.7 Control structures

SCL provides IF-THEN-ELSE-FI and FOR-WHILE constructs. For example,

```
IF b
THEN s1
ELSE s2
FI
```

and

```
FOR i FROM x TO y BY z WHILE b DO
    s
REPEAT
```

UNLESS may be used instead of FI. It is equivalent to IF NOT... For example,

```
UNLESS b
THEN s1
FI
```

ELSF may be used instead of ELSE IF, in which case a second FI is not required. For example,

```
IF b1
THEN s1
ELSF b2
THEN s2
FI
```

is equivalent to

```
IF b1
THEN s1
ELSE IF b2
     THEN s2
     FI
FI
```

SCL also provides a GOTO statement. All statements may be labelled, the scope of a label being the block in which it is used plus any inner blocks. This ensures that jumps into an inner block may not occur. For example,

```
BEGIN
  :
  :
L1:s1
  :
  :
     BEGIN
       :
       :
   L2:s2
     GOTO L1
       :
       :
     END
  :
  :
GOTO L2          *
END
```

The GOTO marked * is invalid.

The final control structure provided by SCL is a form of the PL/I ON construct. It is called a *panoramic* conditional and is similar to an IF statement, except the word WHENEVER is used. For example,

```
WHENEVER b
THEN s
FI
```

No ELSE clause is allowed. WHENEVERs are continuously acting IF statements. That is, after every statement within the scope of a WHENEVER, if the boolean condition evaluates to TRUE then the body of the WHENEVER statement is executed. The scope of a WHENEVER is the block in which it is declared

plus any inner blocks. However, a WHENEVER in an inner block, testing the same condition as a WHENEVER in an outer block, will override the initial WHENEVER. For example,

```
BEGIN
WHENEVER x=0 THEN CHECK FI;
  :
  :
    BEGIN
    WHENEVER x=0 THEN FI;
      :
      :
    END
END
```

Within the inner block no action is taken if x becomes zero.

SCL does not provide a statement which allows a selection to be made from a number of alternatives. It was felt that NCL should and a CASE statement was introduced. It was also felt that ON should be used instead of WHENEVER, mainly because it is shorter.

At this point E. W. Dijkstra's paper on guarded commands was discovered [Dijkstra 75]. Dijkstra's concept is to guard a group of statements (commands) with a boolean expression, called the guard. The statements can only be executed if the guard evaluates to TRUE. After experimenting with the concept it was felt that it could be applied in an orthogonal manner to the IF, CASE and ON constructs.

In NCL, the body of the IF, CASE and ON constructs is a list of guarded statement groups. When executing one of the constructs the guards are tested one after another. For IF and ON constructs, all statement groups, whose guards evaluate to TRUE, are executed. However, for a CASE construct only the first statement group, whose guard evaluates to TRUE, is executed. For examples see section A6.2.2.

Because the controlling clause of a FOR statement is not a simple boolean expression, it is difficult to apply the guarded command concept to FOR statements. Thus in NCL, the SCL FOR statement remains essentially unchanged, except that the word LOOP is used instead. See section A6.2.3 for examples.

An alternative to the GOTO statement is the EXIT statement. It was felt that the GOTO is the inferior statement as the only place it is really needed is when one wants to escape from loops and blocks etc. Thus in NCL GOTOs have been replaced by EXITs. All control structures, including blocks, may be labelled, the scope of the label being the structure it labels plus any nested structures. An exit statement within a structure, using the structure's label, will cause the execution of the structure to terminate. See section A6.5 for examples.

5.8 Miscellaneous

5.8.1 The semicolon

In an interactive situation users enter statements line by line. As a statement may continue over one line the system will have problems deciding when a statement is complete. Consider the following statement,

```
SUM:= ONE + TWO
```

In an interactive environment the system has no way of knowing if the user wants to type

```
        + THREE
```

on the next line.

In a batch context this could be solved by using a keyword to indicate the start of a statement. When a keyword is encountered the present statement is obviously completed. However in an interactive situation it is not possible to look at the next line to see if it starts with a keyword and thus this method is not acceptable.

Thus there must be some way to indicate that a statement is finished. This may be done in two ways:-

(i) Assume that the end-of-line means end-of-statement, except if a special continuation symbol appears on the line.

(ii) Terminate each statement with a special symbol.

SCL uses method (i), modified slightly in that if a statement is obviously unfinished the end-of-line will not terminate it. Thus the above example could be written

```
SUM:= ONE + TWO        /+
          + THREE
```

or

```
SUM:= ONE + TWO +
      THREE
```

where /+ is the SCL continuation symbol. This may have a drawback in that if a user enters a complicated line which he thinks is complete but, because of a mistake, is actually incomplete the system would just sit and wait for more input and the user would think it was working. This happens occasionally with LISP where it is easy to make a mistake matching parentheses. This would be solved if the system prompted for input and thus the user would know if the system thought the statement was incomplete. However, this is felt to be messy and possibly confusing for the naive user. Users are forced to think things like "I've reached the end of a

line, but that's OK because the statement is obviously incomplete so I don't need a continuation symbol."

In NCL it was felt that method (ii) should be used and the semicolon was chosen. Thus the above example would be written

```
SUM:= ONE + TWO
          + THREE;
```

This method is slightly modified in that a semicolon is a statement separater rather than a statement terminater. Thus, in the following

```
BEGIN
S1;
S2
END
```

the statement S2 doesn't need a semicolon although one may be put there if desired.

5.8.2 Control structure end symbols

In NCL all control structures are bracketed by a pair of symbols. For example, IF and FI. Choosing the start symbol for a construct is relatively easy. Thus in NCL we have IF, CASE, ON, LOOP, PAR and BEGIN. Choosing the end symbol is not so easy. Basically there are two alternatives:-

(i) reverse the start symbol e.g. FI, ESAC etc.

(ii) use END followed by the start symbol e.g. END IF, END CASE etc.

The choice is quite arbitrary, however it was felt that using one symbol, as in (i), introduced fewer problems for the language translater. Thus NCL uses FI, ESAC etc. (but not NIGEB!).

CHAPTER 6

IMPLEMENTATION CONSIDERATIONS

"If to do were as easy as to know what were good to do,... "

William Shakespeare

The Merchant of Venice

Act 1 Scene 2

6.1 Introduction

This chapter will cover a design for the implementation of UNIFACE which is an interface between users and a network of OSs. UNIFACE must 'understand' NCL and attempt to do whatever the user specifies. There are certain things which the user can specify in NCL but UNIFACE can't do, such as initiating processes and manipulating files. UNIFACE must, somehow, get the underlying OS to do these things. The OS will make replies and responses to instructions from UNIFACE and UNIFACE must be able to understand these and if necessary send appropriate messages to the user in NRL. Of course, UNIFACE must be able to communicate with other nodes via NAL. See diagram 6.1.

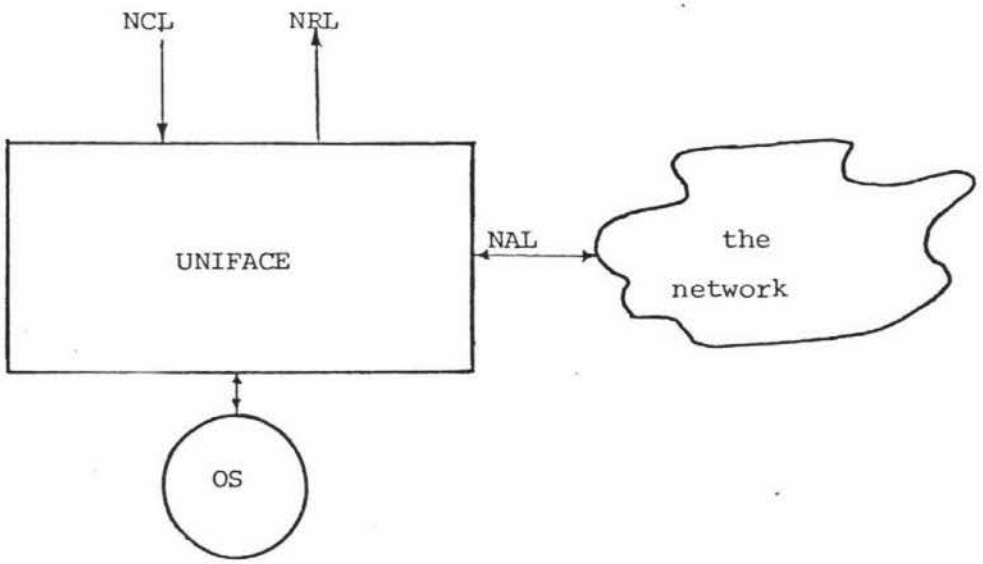


Diagram 6.1 UNIFACE overview.

6.2 General implementation methods

To implement UNIFACE we could either implement a new OS, supporting UNIFACE at each node or interface UNIFACE to each local OS. If we implement a new OS at each node we could provide a standard interface to UNIFACE and thus the implementation of UNIFACE could be standardised.

However, a new OS would have to be written for every type of machine in the network. This would involve a great deal of effort; enough in fact to make the method unfeasible. Furthermore, because an OS interfaces with executing programs, often in a unique manner, much existing software would have to be modified. This is especially true for compilers.

So we must somehow interface UNIFACE to a variety of local OSs. This itself can be done in two ways:-

(i) at the OSCL level, translating NCL into the local OSCL of each node and translating local OSRL responses into NRL.

(ii) at the supervisor call (SVC) level, translating NCL into the necessary SVCs or OS intrinsics and trapping elementary OS responses before they are turned into local OSRL responses, and using these to generate NRL messages.

See diagram 6.2.

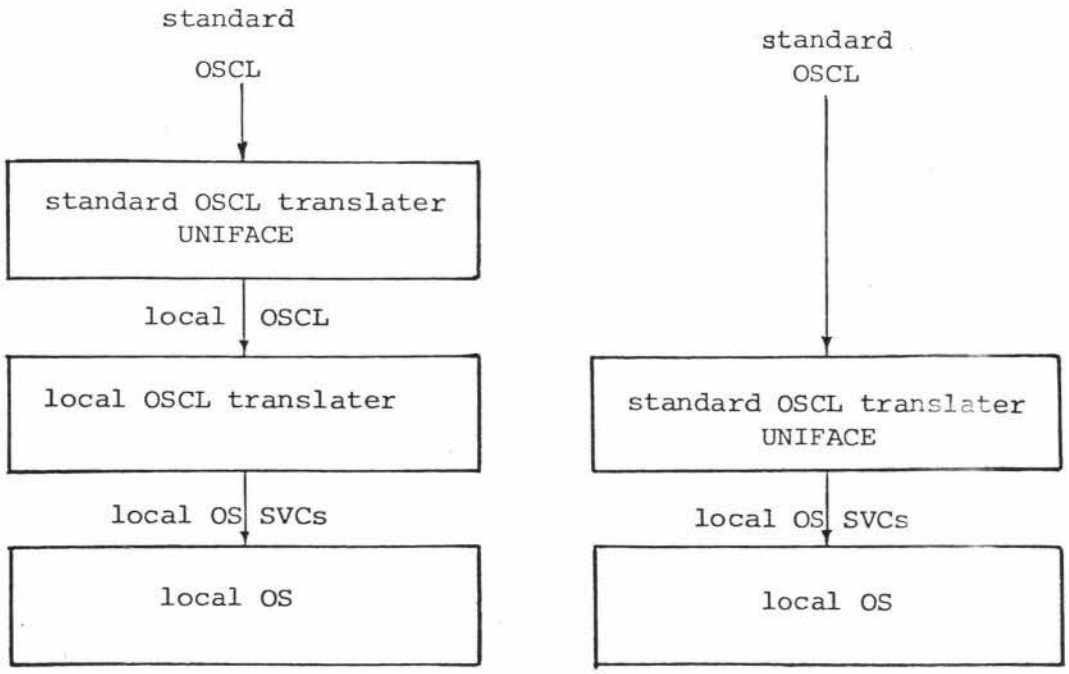


Diagram 6.2 Implementation methods

6.2.1 The local OSCL method

This method has the advantage that we don't have to directly interface UNIFACE to any OS; we can leave that up to the local OSCL handler.

There would however, be problems translating a language, such as NCL, into many present day OSCLs. The problems stem from the fact that for many NCL features (e.g. data structures, parallel groups and procedures) there is no analogous feature in many other OSCLs. Furthermore, system responses, which vary widely from machine to machine, would need to be translated into NRL. This could be done using a table of skeleton messages (unique for every model OS) against which the incoming messages would be matched. The table would supply information about the meaning of each message which, combined with names and values extracted from the message, could be used to create the appropriate NRL message.

Problems arise here if the local message is not specific enough; for example three NRL messages may be covered by one local message.

A further problem with this method is that it is likely to lead to longer response times due to the fact that two translations are made of the original input.

6.2.2 The SVC method

With this method UNIFACE would have to directly interface with a variety of OSs. Each OS will provide a different set of SVCs and will make responses in a different way. Typically, the SVCs provided will allow UNIFACE to request process initiation, file creation and manipulation

and not much else. Most other NCL features would need to be handled by UNIFACE itself. The necessary SVCs are obviously there because the local OSCL uses them. The main question here is whether enough OSs provide them in such a way so that UNIFACE can access them. After much discussion it was felt by the team that, although many older OSs don't, most newer OSs do. In fact, there seems to be trend, amongst OS designs, to provide more and more suitable SVCs.

Choosing between these two methods is difficult, in fact the discussion still continues. However, considering the pros and cons of the two methods it was felt that method (ii) was better, although of course, experience is the only way to be sure.

6.3 Interpreter vs compiler

Considering, that with method (ii), UNIFACE must handle the majority of NCL within itself, the next question was whether it should interpret NCL or compile it. In either case the system would include machine dependent sections; with a compiler-based system these would be the code emitters and with an interpretive system they would be a set of OS interface routines.

In a batch situation an interpretive system would introduce longer response times as compared to a compiler-based system. In an interactive situation, where most input is one line statements, there would probably be little difference. To reduce the response time an interpretive system could be split into two parts; a translator, which would parse the

NCL source and translate it into an intermediate code, and an interpreter for the intermediate code. This would allow all standard procedures to be translated into intermediate code and stored as such, ready to run.

An important advantage of an interpretive system is its ability to produce good run-time error recovery and meaningful error messages. Compiler-based systems are typically inferior in this area.

Considering this it was felt that the initial implementation of NCL should be interpretive; compilers could possibly be written at a later date when the system had proven to be effective.

6.4 System architecture

We are now in a position to be able to consider the overall architecture of UNIFACE. See diagram 6.3

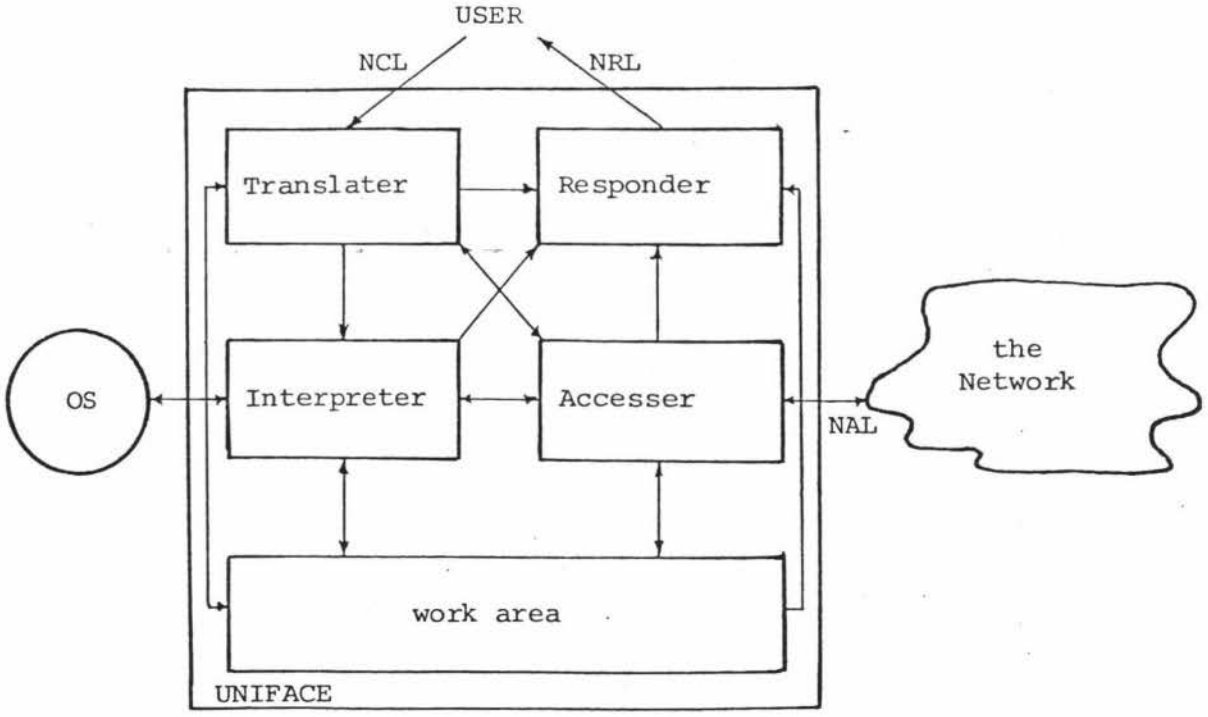


Diagram 6.3 UNIFACE architecture.

The Responder and the Accesser handle NRL and NAL respectively. As this thesis is only concerned with NCL this chapter will not cover the implementation of these sections of UNIFACE. The Translater and the Interpreter and the way they communicate with the remainder of the interface, are discussed in the following sections.

6.5 The Translater

The task of the Translater is to take the NCL input from the user, analyze it for correct syntax and translate it into the intermediate code language (see section 6.7) for the Interpreter.

Syntax analysis is a relatively well understood task and constructing a parser, using a method such as recursive descent, would be fairly straightforward; see [Aho & Ullman 77]. There are only a couple of points which need to be mentioned here.

6.5.1 The symbol table

As we shall see later (see section 6.8.1) the Interpreter needs to be able to access the symbol table at run-time. Normally, the symbol table grows and shrinks in accordance with the block structuring of the session. That is, at the end of a block the symbol is collapsed, all entries, put into the symbol table during the block, being removed. Because of this there would be nothing in the symbol table when the Interpreter accesses it at run-time. There are two basic solutions to this problem:-

(i) Don't collapse the symbol table at the end of each block. Thus, for every depth of nesting we need a chain of symbol tables for each block used at that depth. See diagram 6.4.

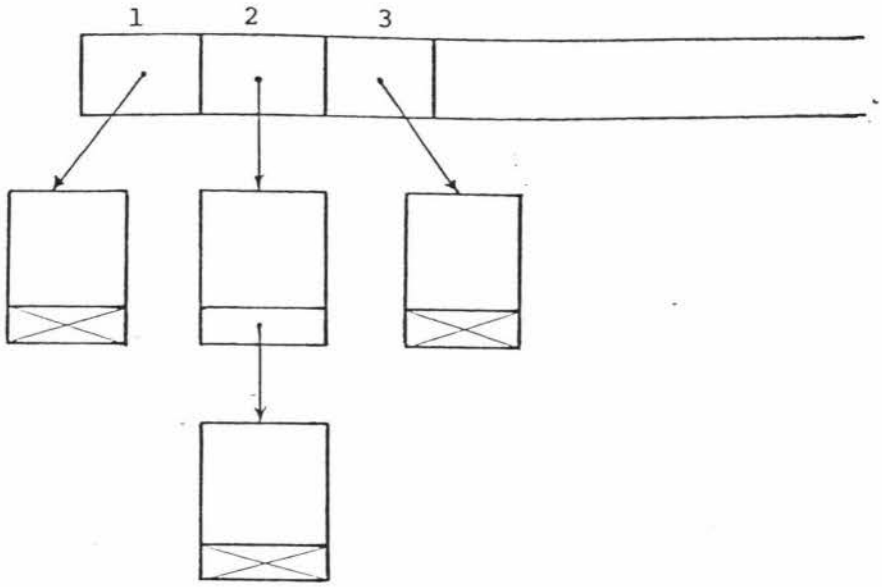


Diagram 6.4 Uncollapsed symbol table

At runtime as each block is finished it can be linked out of the chain. A disadvantage of this method is that the symbol table for a large batch session could grow to an unacceptable size.

(ii) Reconstruct the symbol table at run-time. This requires that the intermediate code include an instruction which tells the interpreter to place an entry in the symbol table. The instruction would be

DECLARATION	mode of object	name of object

This is like the normal stack allocation instruction but also causes an entry to be made in the symbol table. As identifiers may be hierarchical the name of an object would be stored in the following form

# of parts	# of chars in 1st part	1st part	etc

For example, A_SHORT_NAME would be stored as

3	1	A	5	SHORT	4	NAME

Another point to consider here is that a subsidiary session must know the names and modes of the objects in the parent session which it may access. This means that when a subsidiary session is initiated the parent session must, effectively, send it the present state of the symbol table. To do this it must scan the symbol table and, for each entry, send an instruction, such as that used in method (ii), to the subsidiary session.

Considering the general simplicity of method (ii), and that it must be used for initiating subsidiary sessions, it was felt that it was the obvious choice.

6.5.2 Error handling

If the input is syntactically incorrect error messages, in NRL, must be produced. To do this the Translater needs to send to the Responder a code indicating the nature of the error. The Responder must then create the appropriate message for the user.

If an error occurs in an interactive session the Translater may decide to abandon the statement it is currently translating. The user can then reenter the statement. Obviously, the translater must be able to discard the code that it had already produced for the statement. Thus the Translater must be able to mark the begining of statements in the output code array. As NCL is block structured and control structures are nestable, the Translater obviously needs to maintain a stack of statement start markers.

6.5.3 Real machine definition

Any real machine will have its own word-length and range of representable numbers. For example one machine may be able to represent the integer 1234567890123, whereas another machine may have too small a word-length to be able to represent it. The translater must take this into account when handling such things as numeric literals. This is done by supplying each translater with a definition of the real machine it is working on. The definition would specify such things as the word-length, the largest and smallest representable integer numbers, the largest and smallest representable real numbers and the character size.

6.5.4 The AT clause

An AT clause specifies that the statement following it, which may be a procedure call or a block, must be executed at a particular site. The Translater can deal with an AT clause in two ways:-

(i) Translate the statement before sending it to the desired remote site. This would mean that the Translater would either have to keep the real machine definitions of every site in the network and use the appropriate one when doing the translation, or only semi-translate the statement, leaving the machine dependent bits to the remote translater.

(ii) Send the statement, as is, to the remote translater which will do all the translation itself. If the statement is a procedure call, for example,

```
AT TIMBUKTU USING XYl23INCOGNITO
     MYPROG;
```

then the translater only needs to scan until it finds the next semi-colon. However, if the statement is a block, for example,

```
AT TIMBUKTU USING XYl23INCOGNITO
     BEGIN
        :
        :
           BEGIN
              :
              :
           END;
     END;
```

then the Translater must scan for the matching END, taking into account any nested blocks.

It was felt that as method (ii) presents fewer implementation problems it should be used in preference to method (i).

To save network congestion, messages should be kept as short as possible. Thus having identified the text that needs to be sent to the remote site the Translater should remove any unnecessary blanks and comments. The resulting text can then be passed to the Accesser which will send it on to the remote site where the Accesser receiving it will pass it on to the Translater. Thus, the Translater must be able to send and receive NCL text to and from the Accesser.

6.6 The intermediate code

NCL is block structured which implies that the Interpreter should use a stack mechanism for variable storage, with static and dynamic chains as is usual; see [Randell & Russel 64]. A stack could also be used for procedure linkage and expression evaluation. Considering this it is obvious that the intermediate code will be a form of reverse polish.

For reasons of portability, it was decided to base it on BCPLs OCODE [Richards 71], modifying it as necessary.

6.6.1 Procedure calls

The code for a procedure would be

MKSTK	ALLOCATE 2+P	EVAL Ps	SET RL	CALLPROC

where

MKSTK = mark stack for block entry

ALLOCATE 2+P = allocate stack entries for

- 1 return label
- 1 result label
- P parameters

EVAL Ps = evaluate the parameters

SET RL = set the return label to be the present location + 2

CALLPROC = enter the procedure

Just before the CALLPROC instruction is executed the stack would look as in diagram 6.5

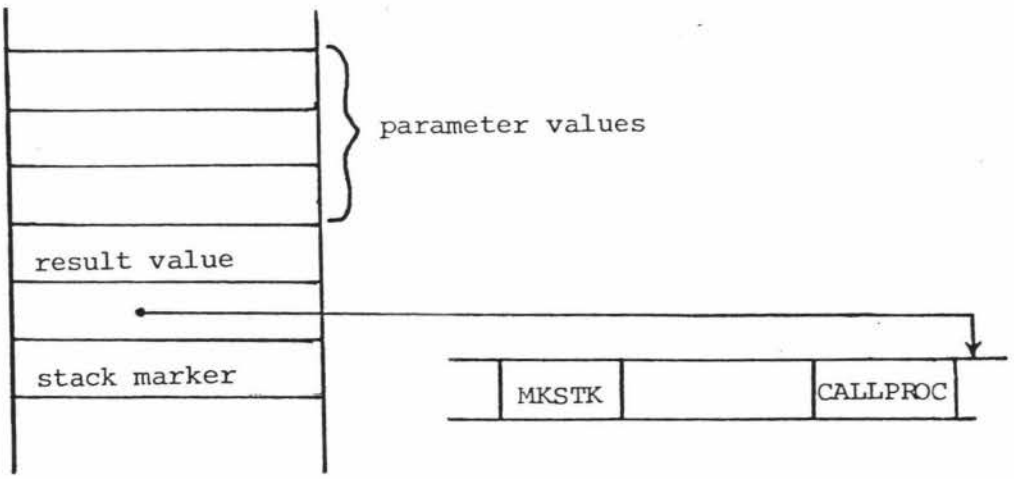


Diagram 6.5 The stack just before a procedure call.

6.6.2 Blocks and declarations

The code for a block should begin with an instruction to mark the stack (i.e. put a value on the stack which indicates a new block) and end with an instruction to cause the stack to be collapsed down to the mark.

As described in section 6.6.1 all declarations begin with a 'make symbol table entry' instruction. The remainder of the code for a declaration causes an appropriate stack entry to be made. A complex object, such as a string, data structure or resource object, could either be kept on the stack or in a separate work area with the stack entry pointing to it. As data structures and strings are flexible (i.e. may change in size) it would be easier to store them in a work area. Thus, for those objects, in addition to code for the stack entry, there must be code to cause space to be set aside in the work area.

Sets and queues are stored as linked lists. Arrays, being flexible, could also be stored as linked lists but this would, unfortunately, increase access time. The solution is to store arrays as linked lists of groups of elements, i.e.

The stack entry for an array indicates the actual length of the array and thus how much of the last group is empty.

6.6.3 Expressions and assignments

Coding expression evaluation in reverse polish is well understood and there is no need to go into details here. However, there are a few points worth mentioning.

The first concerns data structure and resource literals. If such a literal is encountered in an expression an anonymous structure is created for it in the work area. The elements or attributes comprising it are then evaluated and loaded into the structure one by one.

The second point is that the address for a resource attribute must include an offset into the structure in the work area. For example, suppose we have a file whose stack entry has a stack address of 2,3 (i.e. it is the third object declared in the second block). The address for the attribute BLOCKSIZE, which is the second file attribute, would be 2,3,2. Because of this special instructions are needed to handle resource attributes. For example, as well as the 'store' instruction which uses a normal stack address couple we also need a 'store attribute' instruction which uses an attribute address triplet.

The last point concerns the implications of subsidiary sessions, within which it is possible to access variables in the parent session. The code to get the value of such a variable can't be a simple 'load' instruction. Rather the code must be a special instruction which causes the Interpreter to ask the parent session, via the Accesser, to supply the value of the variable. A similar instruction is required when an assignment is made to a parent session variable.

6.6.4 Control structures

The NCL LOOP statement is essentially identical to many such statements in other languages and the code for such statements is well understood. However, the NCL IF, CASE, ON and PAR structures are different enough to deserve attention.

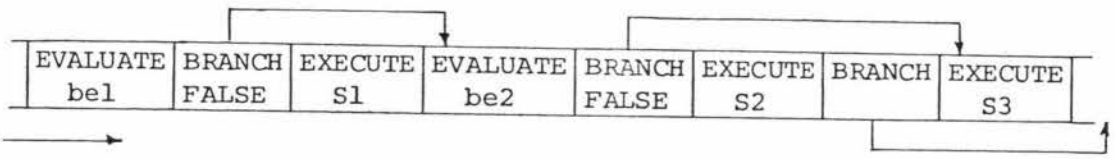
In the examples in the following sections the symbol be is used to indicate a boolean expression and the symbol S is used to indicate a statement or group of statements.

6.6.4.1 IF statements

The code for an IF statement consists of a series of guarded groups. Each group consists of a boolean expression evaluation followed by a conditional branch around the group. For example, the code for

```
IF
bel THEN S1
,be2 THEN S2
,ELSE     S3
FI
```

would be

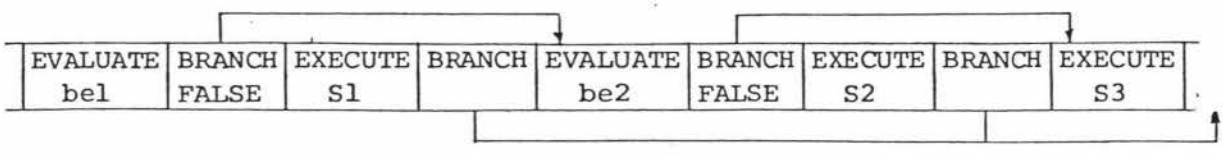


6.6.4.2 CASE statements

The code for a CASE statement is very similar to that for an IF statement except at the end of each group there is a branch to the end of the case statement. For example, the code for

```
CASE
bel: S1
,be2: S2
,ELSE S3
ESACE
```

would be



6.6.4.3 ON statements

ON statements are different from IF and CASE statements in that, firstly they are panoramic (i.e. they must be executed after every statement) and secondly they have scope and thirdly they can override earlier ON statements (see section 5.7).

Because they must be executed after every statement, the code for ON statements is kept in a separate structure fromthe main code array. After every statement in the main code array is an instruction which causes control to branch to the ON statement code structure. When the ON statements have been executed control automatically returns to the next main-code statement.

To cater for the override feature the ON statements are kept in a linked list; one guarded group per element of the list. When a new ON statement is encountered a new element for each guarded group in the statement is added to the head of the list. As an element is added a search is made, down the list, to determine if the code, to evaluate the guard of the new element, is identical to that of any existing element. If it is, the older element is linked out of the list. To enable the older element to be linked back into the list when the new element is removed at block exit, a pointer is needed, within the new element, to point to the old, overridden element. Also needed is a pointer in the old element to the previous element in the list. That is, the list is doubly linked. Thus, each element of the list would have the following general structure:-

where

FP = a forward pointer to the next element

BP = a backward pointer to the previous element

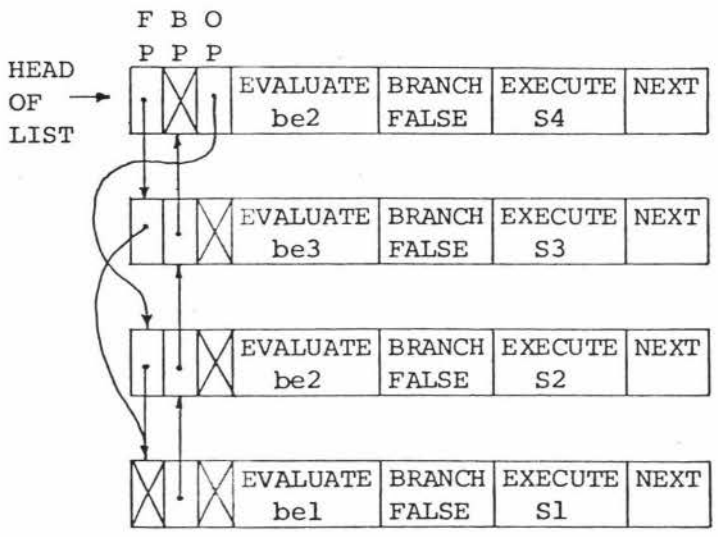
OP = a pointer to an element that has been overridden

NEXT = an instruction which causes control to pass to the element pointed to by FP. If FP is null, i.e. the end of the list has been reached, control is returned to the main code array.

Example:- the code for

```
ON
be1: S1
,be2: S2
,be3: S3
NO;
  :
  :
ON
be2: S4
NO;
```

would be



To cater for the fact that ON statements have scopes bounded by the block in which they occur we require a stack of pointers, one pointer for each level of nesting, which will indicate the block boundaries in the linked list. See diagram 6.6.

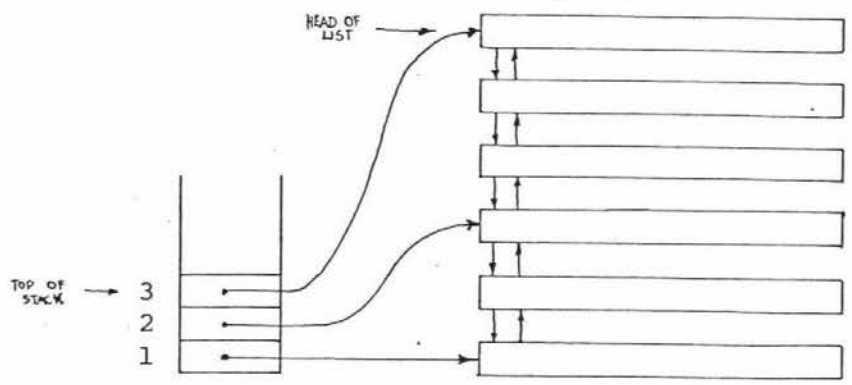


Diagram 6.6 The ON statement code structures.

When a block is exited the head-of-list pointer is reset to the pointer held in the top of the stack.

6.6.4.4 PAR statements

A par statement indicates two, or more, statement groups which must be executed in parallel. For each statement group, the main interpreter (hereafter called the father) will fireup a sub-interpreter (called a son). All the sons run concurrently.

The code for each statement group is stored in one row of a two-dimensional array. When a son is fired up, the father indicates which row of the array it should execute. The semantics of the PAR statement

indicate that the order in which the sons begin to execute the statement groups is undefined (see section A6.2.4). To ensure this a semaphore is implicitly declared, and initialised to zero, for each son. The first thing a son does is try to GET its semaphore. When the father has fired up all the sons it FREEs all the semaphores thus allowing all the sons to continue at once.

Having fired up all the sons and FREE'd the semaphores, the father must wait for them all to finish. It does this by trying to GET all the sons' semaphores. As each son finishes it FREEs its semaphore and when all the sons have finished the father can continue. For example, the code for

```
PAR
S1
,S2
RAP
```

would be

DECL SEM1,SEM2	FIREUP1	FIREUP2.	FREE(SEM1,SEM2)	GET(SEM1,SEM2)

main code stream ⟶

1	GET(SEM1)	EXECUTE S1	FREE(SEM1)	DIE
2	GET(SEM2)	EXECUTE S2	FREE(SEM2)	DIE

code for parallel statement groups

where

DECL SEM1,SEM2 = declare the semaphores SEM1 and SEM2 and initialise to zero

FIREUPn = fire up a son using row n of the statement-group code array

DIE = stop executing

6.7 The Interpreter

The task of the Interpreter is to execute the intermediate code received from the Translater. Obviously, there are some things which the Interpreter can't do itself (e.g. run a program or copy a file). The aims of this section are to determine how the Interpreter will do some of the things it has to do and to identify exactly what it will need to get the OS to do.

6.7.1 Data structures used

As mentioned in the discussion in section 6.7, the Interpreter needs a stack for variable storage. Global variables are typically held at the bottom of stacks, where they can be accessed at any point during the execution of a session. However, in NCL the number of global resource objects may change during a session as a user SAVEs new objects or DELETEs old ones. Because of this it is not possible to store global objects on the bottom of the stack. Rather, they need a separate stack of their own, hereafter called the global stack.

NCL data structures, being flexible, can't be stored on the stack and, as already mentioned in section 6.7, are kept in a special work area. Resource objects are not flexible and could be kept on the stack. However, this would require the complete object to be transfered from the stack to the global stack if it is SAVEd. To avoid this resource objects are also kept in the work area with a stack entry pointing to them.

When a process requests the OS to open a file the OS must check that the user has not specified an overriding definition of the file in NCL (see section 5.4.1). If the user has explicitly provided an overriding

definition (by using the PROCESS attribute) then this is indicated to the OS when the process is initiated. However, a user may provide an over-riding definition by merely declaring a file with the same internal name as that used by the process. In this case the OS must ask the Interpreter if it has a file definition with the same name as the file the process wants to open. To answer this the Interpreter needs to access the symbol table. Thus we need a run-time symbol table which can be accessed by the Translater and the Interpreter.

The code which the Interpreter executes is held in an array. An instruction-pointer variable indicates the location of the next instruction to be executed. The Interpreter also needs special code structures for ON and PAR statements, as discussed in sections 6.7.4.3 and 6.7.4.4.

6.7.2 The usercode system

When a user logs on he presents his usercode and password and the system must check that they are valid. To do this the system must keep a catalogue of each user's usercode and present password. Most OSs maintain a usercode system and UNIFACE could rely on this, interfacing with the OS for any usercode related operations. This would require that UNIFACE be able, not only to ask if a particular usercode is valid, but also be able to ask the OS to create new usercodes and delete old usercodes.

It was felt that UNIFACE should maintain its own usercode system, mapping all usercodes into one OS usercode. That is, as far as the OS is concerned, UNIFACE does everything for one user. Thus UNIFACE does not need to ask the OS to create new usercodes, etc and the OS interface is therefore kept to a minimum.

The usercode system could be simply maintained in a file, one record per user. Each user record would contain information such as the user's password and a reference to his AM.

6.7.3 Processes

For the Interpreter, initiating processes involves more than just specifying to the OS the file of machine code that should be executed. The interpreter must also specify the resource restrictions imposed by the process'es ENVIRONMENT. Furthermore while a process is running the Interpreter must be able to determine the values of the dynamic attributes, for example CPUTIME. The Interpreter must also be able to suspend, reactivate or kill the process in response to the special commands BREAK, CONTINUE and FINISH respectively.

Basically, the Interpreter must set up and initialise a process control block (PCB) which both the Interpreter and the OS may access. When setting up a PCB the Interpreter uses the environment specified for the process, or a default environment if none was specified, to initialise the attributes of the PCB, such as CPULIMIT.

The Interpreter must also ensure that no other process can use the same environment while the process is running. To do this it sets a special attribute in the environment to indicate that it is in use. If the user accesses dynamic attributes, such as CPUTIME, of an environment that's being used by a process, the Interpreter must use the PCB to find out the desired value.

Because two users may run the same program, say a compiler, at the same time every process currently running in the system must be given a unique identification code by the OS. Thus when the Interpreter initiates a process, indicating the PCB to be used, the OS replies with the unique identification code. To suspend, reactivate or kill a process the Interpreter must call the appropriate OS routine specifying the identification code of the process. For example, SUSPEND(1302) will cause process 1302 to be suspended.

6.7.4 Files

The Interpreter handles files via their descripters. That is, to manipulate a real file the Interpreter must build a descripter for it and then ask the OS to open it. This will cause the OS to search for the file described. If no file fits the description the OS will reply with an error code.

To open a file for which the attribute NEW is TRUE the Interpreter must first get the OS to create a file and then open it. If the user closes a temporary file then the Interpreter must ask the OS to delete the file. If the user closes a permanent file the Interpreter merely needs to add an entry to the global stack.

The set of file attributes supported by the OS may differ from the set supported by UNIFACE (see Appendix C).

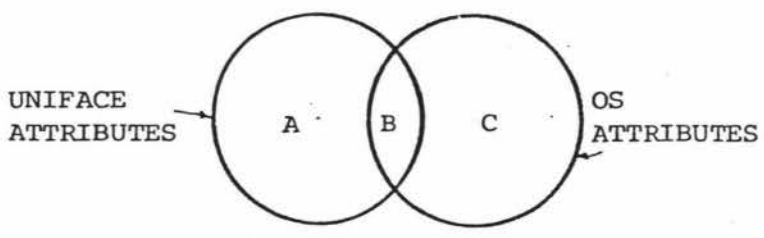


Diagram 6.7 The three types of attribute

From diagram 6.8 we can determine that there are three types of attribute; those supported solely by UNIFACE (A), those supported by both UNIFACE and the OS (B) and those supported only by the OS (C). The two sets will overlap to a greater or lesser degree depending on the OS. However, the smaller region A is the less effort UNIFACE must spend supporting files.

Some of the file attributes likely to be in region A are:-

(i) ACCESS:- Because UNIFACE supports its own usercode system this string attribute is relatively straightforward. If a user is accessing another user's file the system merely checks this attribute to ensure that he is allowed to.

(ii) EXISTENCE:- This attribute is only needed during sessions and is only used by the interpreter as a flag so there are no problems with it.

(iii) LIFETIME:- This attribute indicates the minimum time, in days, the file must be kept on the system before it may be removed. If a process attemptes to delete a file the system must check that its lifetime has run out; if it hasn't the process must not be allowed to delete it.

(iv) LASTUSEDATE:- Every time a process uses the file the system must update this attribute. To do this the system must be able to ask the OS to supply the present date.

(v) PERIPHERAL:- This attribute has, as a default value, a standard peripheral definition determined by the DEVICE attribute of the file. If it is set to indicate that some other peripheral be used then the system must ensure that the file is connected to the correct physical device.

In most cases this can be done by simply setting the 'channel' of the file appropriately.

(vi) SITE:- The value of this attribute can only be changed if the file is closed. If it is assigned a remote site name then the Interpreter must instruct the remote site concerned to set up a special 'file handling' session. This session, unlike a subsidiary session, doesn't need to know the global variables in the parent session. The file handling session runs under the usercode specified by the OWNER attribute of the file. Once the session is initiated the parent can send it instructions to declare the file, specifying the current values of all the attributes. After this, whenever the parent session reads the attribute of the file, the Interpreter must ask the file handling session for the value. Similarly, when the parent session assigns a value to an attribute the file handling session must be the one to actually do it. When a process, to which the file has been attached, uses the file the system must intercept the I/O requests sent to the OS and send them on to the file handling session.

Because the NCL file attributes typically differ from the OS attributes it is necessary for the system to store all file descripters in the users AM.

Some of the attributes in region B are maintained by the OS, for example RECORDNUM. If the user requires the value of such an attribute the Interpreter must be able to ask the OS to supply its value. Other attributes in region B may be changed by the user, for example TITLE, and the Interpreter must be able to tell the OS to change their values.

The attributes in region C are supplied with default values by the Interpreter. These default values are user dependent and specified in the UAM.

6.7.5 The OS routines

From the preceeding discussions we can see that a large portion of NCL can be handled directly by the Interpreter. There are, however, certain functions which the Interpreter must get the OS to perform. For each such function the Interpreter will support an appropriate machine dependent routine.

Firstly, there are a number of file handling routines which perform the following functions

(i) create a file

(ii) delete a file

(iii) read a file attribute

(iv) write a file attribute

(v) copy a file

(vi) input a file from a peripheral

(vii) output a file to a peripheral

(viii) open a file

(ix) close a file

Secondly, there are routines for handling processes

(i) initiate a process

(ii) suspend a process

(iii) reactivate a process

(iv) kill a process

(v) read a process attribute

(vi) write a process attribute

Thirdly, there are a couple of routines for time management

(i) read the time

(ii) read the date

These routines define how UNIFACE interfaces with an OS. They also define the SVCs that an OS should provide before it can support UNIFACE.

6.8 The User Abstract Machine

A user's abstract machine specifies what objects the user may access, including the files he owns and the standard system procedures appropriate to his class of user. See section 5.1.2.

All this information can be stored as a file of intermediate code which when executed will set up the data structures used by the Translater and the Interpreter. That is, it initialises the symbol table, the global stack, the ON statement code structure and the work area.

This implies that the standard procedure GOODBYE, after doing the normal block exit functions, must scan the data structures and produce the code necessary to recreate their present state when the user next logs on. It is already necessary for the system to be able to do this to the symbol table when it initiates a subsisiary session (see section 6.6.1).

Thus the system maintains one file of code for each user which, effectively, defines his AM. The system can keep track of these files

via the usercode system. That is, each entry for a user in the usercode system will contain a reference to his AM file.

CHAPTER 7

RESULTS AND RECOMMENDATIONS

It is a bad plan that admits of no modification

Publius Syrus

Moral Sayings 469

7.1 Results

The major result of this thesis is the definition of the syntax and semantics of NCL. NCL is a powerful, high-level language enabling users to control and monitor the execution of their programs in a straightforward manner. Concurrent execution is catered for and a synchronisation mechanism is provided. Files and peripherals may be declared and their attributes manipulated. Integer, boolean, real and string variables and structures may also be declared and manipulated. NCL also provides high-level conditional and iterative control structures and procedure facilities. NCL is designed to be used on a network of OSs, allowing users to specify where processes are to be executed and where files are located. An abstract machine mechanism is supported which allows the system to be tailored to suit the individual user and which also improves security by strictly controlling the objects the user may access.

We have also considered how NCL could be implemented on a network, outlined the problem points and suggested possible solutions. In doing this we have identified those points where the system must interface with an OS and explained, in general, how the interface could be provided.

7.2 Suggestions for further work

To conclude the thesis the author would like to present some ideas on further language refinements and other areas of interest for which, unfortunately, no time was available for investigation.

Firstly, there are several refinements which could be made to NCL.

(i) PASCAL-like scalars could be introduced. This would allow certain resource attributes to be treated in a simpler manner. For example, the file attribute EXISTENCE could have the mode

SCALAR(TEMPORARY,PERMANENT) VAR

(ii) The special commands BREAK, FINISH and CONTINUE would be better as builtin procedures. This would simplify the syntax.

(iii) It was pointed out that we had not considered monitors when we were deciding what synchronisation method to use. The semaphore mechanism we had opted for, being primitive, would allow deadlock situations to arise if the user wasn't careful. Monitors are a higher level mechanism and it is harder to generate deadlocks. At present it is felt that the choice of semaphores was the correct decision, allowing the user greater flexibility, although a more careful consideration of monitors may prove this to be wrong.

(iv) The string relation ENDS is too much like END for comfort. Perhaps TAILS could be used instead.

An interesting idea, which arose during the development of NCL, was to produce a programming language based on NCL. This could be done by adding I/O statements and REF variables and possible some other constructs. The advantage of this would be that the user of such a PL would only need to learn one language (or rather two dialects of the same language) for both programming and job control.

A further area of investigation would be the design of a good file editing facility. Although users commonly do a lot of file editing it was felt that edit facilities should not be an integral part of NCL. Rather they should be provided by a standard procedure. The design of the edit facility provided by this procedure would involve a survey of current edit languages and a derivation of user requirements in this area.

The definition of the syntax and semantics of NCL is rather informal. An interesting project would be to define the language formally, perhaps using denotational semantics. This would be a large task requiring a much more detailed analysis of how UNIFACE interfaces with various OSs.

REFERENCES

[Agerwala 77]
Some extended semaphore primitives.
T. Agerwala
Acta Informatica, Vol 8, no. 3, 1977

[Aho & Ullman 77]
Principles of compiler design.
A. V. Aho, J. D. Ullman
Addison-Wesley, 1977

[Baird 75]
Fredettes Operating System Interface Language (FOSIL)
G. N. Baird
in [Unger 75]

[Bandat 75]
Concepts for a generalised command language.
K. Bandat
in [Unger 75]

[Barron 71]
Computer Operating Systems.
D. W. Barron
Chapman and Hall, London, 1971

[Barron 77]
An Introduction To The Study of Programming Languages.
D. W. Barron
Cambridge University Press, 1977

[Barron & Jackson 72]
The Evolution of Job Control Languages.
D. W. Barron, I. R. Jackson
Software- Practise and Experience, Vol 2, pp 143-164, 1972

[BCS 77]
Journal Of Development
British Computer Society Working Party on Job Control Languages
in [IFIP 78]

[Brunt & Tuffs 76]
A User Oriented Approach to Control Languages.
R. F. Brunt, D. E. Tuffs
Software- Practice and Experience, Vol 6, pp 93-108, 1976

[Burroughs 77]
B7000/B6000 Series Work Flow Language Reference Manual.
Burroughs Corporation
Burroughs Corporation Publication 5001555, June 1977

[Campbell & Habermann 73]
The specification of process synchronisation by path expressions.
R. H. Campbell, A. N. Habermann
University of Newcastle Upon Tyne, December 1973

[Cheriton 76]
Man-machine interface design for timesharing systems.
D. R. Cheriton
Proceedings of the ACM 76 Annual Conference, pp 362-366, Oct 1976

[Chupin 75]
Command Languages and Heterogeneous Networks
J. C. Chupin
in [Unger 75]

[Courtois et al 71]
Concurrent control with readers and writers.
P. J. Courtois, F. Heymans, D. L. Parnas
Communications ACM, Vol 15, pp 667-608, 1971

[Cowan 75]
Burroughs B6700/B7700 Work Flow Labguage
R. M. Cowan
in [Unger 75]

[Dakin 75(1)]
A General Control Interface for Satellite Systems
R. J. Dakin
in [Unger 75]

[Dakin 75(2)]
A General Control Language: Language Structure and Translation.
R. J. Dakin
The Computer Journal, Vol 18, no 4, Nov 1975

[Dijkstra 68]
Cooperating Sequential Processes
E. W. Dijkstra
in Programming Languages (F. Genuys ed) New York: Academic Press, 1968

[Dijkstra 75]
Guarded Commands, non-determinancy and formal derivation of programs.
E. W. Dijkstra
Communications ACM, Vol 18, pp 453-457, 1975

[Enslow 75]
Operating System Command Languages: A brief history of their study.
P. H. Enslow
in [Unger 75]

[Gram 77]
Command Languages Incorporated in Programming Languages or Not?
C. Gram
in [IFIP 77]

[Gram & Hertweck 75]
Command Languages: Design Considerations and Basic Concepts.
C. Gram, F. Hertweck
in [Unger 75]

[Grimsdale & Kuo 75]
Computer Communication Networks
R. L. Grimsdale, F. F. Kuo (eds)
Noordhoff Leyden, 1975

[Herman et al 75]
Minimizing Command Language in a system base on a high-level language.
D. Herman, M. Raynal, L. Trilling, J. R. Verjus
in [Unger 75]

[Hopper 76]
KIWINET. An Experimental Resource Sharing Network. Report No 1
K. Hopper (ed)
Massey University Computer Unit, Report #30, August 1976

[Hopper, James & Jenkins 77]
KIWINET. System Control and Access. Preliminary Language Specifications.
K. Hopper, N. S. James, P. C. Jenkins
Massey University Computer Centre, Report #34, May 1977

[Hopper, James & Jenkins]
KIWINET. System Control and Access. Technical Progress Report.
K. Hopper, N. S. James, P. C. Jenkins
Massey University Computer Centre, Report #35, February 1978

[Huckle 76]
Creating a Uniform User Interface to a Heterogeneous Computer Network.
B. A. Huckle
SChool of Information Sciences, Hatfield Polytechnic, 1976

[Huckle 78]
An Interface for Interactive Users within a Heterogeneous Computer Network.
B. A. Huckle, G. M. Bull
School of Information Sciences, Hatfield Polytechnic, 1978

[ICL 74(1)]
System B: Writing Job Control Programs
International Computers Limited
International Computers Limited, Technical Publication 6325, 1974

[ICL 74(2)]
System B: SCL Vocabulary
International Computers Limited
International Computers Limited, Technical Publication 6400, 1974

[ICL 75]
System B: SCL Syntax
International Computers Limited
International Computers Limited, Technical Publication 6407, 1975

[IFIP 77]
TC2 Working Group 2.7 Bulletin No 1
International Federation for Information Processing
December 1977

[IFIP 78]
TC2 Working Group 2.7 Bulletin No 2.
International Federation for Information Processing
March 1978

[James 76]
KIWINET Working Paper NSJ/1: Some Aspects of Job Control.
N. S. James
Massey University Computer Unit Working Paper 76/3, 1976

[Jardine 75]
The structure of operating system control languages.
D. A. Jardine
in [Unger 75]

[Jensen & Lauesen 75]
Programming Language extensions which render JCLs superfluous.
J. Jensen, S. Lauesen
in [Unger 75]

[Kugler et al 79]
Project NICOLA. Progress report no 3.
H. J. Kugler, N. Lehmann, P. Putfarken, C. Unger
Dortmund University, 1979

[Madsen 77]
A Guide to the OS/MVT Implementation of CCL.
J. Madsen
in [IFIP 77]

[Madsen 78]
CCL - A high-level command language; Design and implementation.
J. Madsen
in [IFIP 78]

[du Masle 75]
An evaluation of the LE/1 Network CL designed for the SOC network.
J. du Masle
in [Unger 75]

[Mason 77]
Operating System Command Languages. Some Recent Developments.
C. B. Mason
in [IFIP 77]

[Moore 75]
The feasibility of a transportable job organisation language.
L. Moore
London University, Birkbeck College, Nov 1975

[Newman 75]
Machine Specific Facilities in a Machine Independent Command Language.
I. A. Newman
in [Unger 75]

[Newman 78]
Research on Machine Independent Command Languages at Loughborough University
I. A. Newman
in [IFIP 78]

[Nicholls 75]
The structure and design of programming languages
J. E. Nicholls
Addison-Wesley Publishing Company, 1975

[Parsons 75]
A High-Level Job Control Language.
I. T. Parsons
Software- Practice and Experience, Vol 5, pp 69-82, 1975

[Popek et al 77]
Notes on the design of EUCLID.
G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, R. L. London
Proceedings of the ACM conference on Language Design for Reliable Software
(Ed D. B. Wortman) SIGPLAN Notices, Vol 12, no 3, March 1977

[Randell & Russel 64]
Algol60 implementation: The translation and use of Algol60 Programs on a computer.
B. Randell, L. J. Russel
London: Academic Press, 1964

[Rayner 75]
Recent Developments in Machine-Independent Job Control Languages.
D. Rayner
Software- Practice and Experience, Vol 5, pp 375-393, 1975

[Rayner 77]
An Introduction to Network Job Control Languages.
D. Rayner
in [IFIP 77]

[Rayner 78]
Past and Present Solutions to the Problems of Network Job Control.
D. Rayner
in [IFIP 78]

[Richards 71]
The portability of the BCPL compiler.
M. Richards
Software- Practice and Experience, Vol 1, pp 135-146, 1971

[Sayani 76]
A basis for the design of Operating System Command and Response Languages.
H. Sayani
Proceedings of the ACM 76 Annual Conference, pp 373-380, Oct 1976

[Tennent 77]
Language Design methods based on semantic principles.
R. D. Tennent
Acta Informatica, Vol 8, no 2, 1977

[Tomlinson 76]
A High-Level Computer Control Language.
R. T. Tomlinson
Proceedings of the ACM 76 Annual Conference, pp 381-386, Oct 1976

[Unger 75]
Command Languages. Proceedings of the IFIP Working Conference on CLs.
C. Unger (Ed)
North-Holland, 1975

[Wirth & Weber 66]
EULER: A generalisation of ALGOL and its formal definition.
N. Wirth, H. Weber
Communications ACM, Vol 9, no 1, pp 13-25, 89-99, 1976

APPENDIX A

NCL INFORMAL LANGUAGE DESCRIPTION

CONTENTS

A1 Introduction

A complete formal description of the semantics of NCL has not been made. Rather the following sections attempt to give the reader an intuitive understanding of the use and meaning of the language.

In several cases, concepts or language constructs which have been discussed in detail in the body of the thesis are given only brief mention here.

A2 Lexical Considerations

Blanks may be used freely to make a program more readable. They have no semantic effect. Language symbols may not contain blanks but at least one blank must separate adjacent alphanumeric language symbols: for example NOBLANK does not mean the same as NO BLANK.

A line boundary is equivalent to a blank and may not appear in language symbols or literals.

Comments are enclosed within @ characters and may appear between any two language symbols. Comments have no semantic effect.

An identifier is composed of several subparts seperated by underbars. The maximum allowable number and length of subparts is implementation dependent. These limits should never be less than 10 subparts and 30 characters per subpart.

A3 Sessions (see section B3.1)

A3.1 Introduction

A session is the major unit of work performed by a user and consists of a series of declarations and statements. A session begins, at 'log on', with a call on the standard system procedure HELLO (see Appendix C) and ends at 'log-off' with a call on the standard system procedure GOODBYE.

Every session runs in its own environment and thus has limited amounts of certain computational resources e.g. CPUTIME and COST. If the user exceeds any of these limits the session will terminate. The environment of a session depends upon the user and is called the users 'profile'.

A session may be initiated from a batch input device (e.g. a card reader) or an interactive terminal.

A3.2 Session Translation and Execution

A batch session is translated in its entirety before any execution is attempted. If any syntax errors are found during translation then execution is not attempted.

In an interactive session lines are translated as they are input and error messages are relayed back to the terminal, in which case the user can then retype the line. If a semicolon is typed the statement which it ends is executed, unless that statement is nested, in which case the outer statement is not complete. For example, if the session input is

```
HELLO
  .
  .
  .
    BEGIN                    @ line A @
      .
      .
      .
    IF
      .
      .
      .
    FI;
      .
      .
      .
    END;                     @ line B @
  .
  .
  .
GOODBYE;
```

the statements between line A and line B are not executed until the semicolon on line B is encountered.

A3.3 Dual interactive sessions

The NCL system supports two interpreters per user. This enables the user to execute command language statements whilst some other

statement is being executed in the 'background'. A change-level-char,?, is used to switch input between the two interpreters. For example:-
(systems responses are underlined)

```
myprog;                          @ begin execution of myprog @
?                                @ switch interpreters @
- execution of myprog begun
DELETE(filex);.                  @ request to delete a file called filex @
?                                @ switch interpreters again
- execution of myprog stopped
+ filex deleted
```

There are three special commands which can be used to affect the operation of the 'other' interpreter. Table A1 lists these commands, their abbreviations and their effects.

Command	Abbreviation	Effect
BREAK	BRK	Any task being executed by the other interpreter is suspended.
CONTINUE	CON	Any task controlled by the other interpreter that is suspended is restarted.
FINISH	FIN	Any task controlled by the other interpreter (even if it is suspended) is killed.

TABLE A1 Special Interactive Commands

For example, (system responses are underlined)

```
myprog;                          @begin execution of myprog @
- execution of myprog begun
?BREAK                           @user remembers a necessary file equation @
- execution of myprog suspended
FILE VAR card:= (TITLE:= "sheep_mob3_ticks",
                 DEVICE:= DISK);
CONTINUE                         @ set myprog going again @
- execution of myprog continues
?                                @ back to original interpreter @
```

A3.4 Subsidiary sessions

During a session it is possible to initiate work at another site (which must be a node of the network and support the NCL system). This is done by preceding a procedure call or a block with an AT clause (see section B3.7). For example,

```
AT VICTORIA USING VIC234THOMSON
     myprog;
```

causes the procedure myprog to be executed at the site called VICTORIA under the usercode VIC234THOMSON.

This causes a *subsidiary* session to be started at the named site, which remains in force until the specified procedure call or block terminates.

Any system messages generated by the actions in a subsidiary session are relayed back to the parent session.

All objects in the parent session may be accessed by the subsidiary session. For example,

```
INT VAR X;
AT VICTORIA USING VIC234THOMSON
     BEGIN
       :
       :
     X:= 500;
     END;
```

A3.5 Printed Session Output

For every session a journal is printed containing the lines input and any output produced by the users programs. It is preceeded by a page containing the users usercode and the session title in large lettering for easy identification. The details of the contents are standardised but do not concern this document. The overall contents may be controlled by using the standard system procedure LOG.

A3.6 Interactive Prompts

During an interactive session when the translator requires input it prompts the user by issuing a character in the first column of a line. The character used depends upon which interpreter will execute the input, as follows

Interpreter	Prompt Character
1	- (minus sign)
2	+

System messages are preceeded by this prompt character to identify from which module the message originates.

A4 Objects and their declaration (see section B3.3)

A4.1 What Objects

NCL allows the user to declare and manipulate the following types of objects.

- integer, real, boolean, string (simples)
- semaphore
- file, peripheral, environment (resources)
- array, set, queue (data structures)
- procedure (special case of file)

These objects are described in greater detail in the following sections.

A4.2 The Scope of Objects

Most objects in an NCL session (apart from labels and resources) are in scope only within the block in which they are declared excluding any inner block.

For example

```
BEGIN
INT VAR X;
  :   A
  .
     BEGIN
     INT VAR Y;
       :   B
       .
     END
  :   C
  .
END
```

The scope of X is only those statements labelled A and C. If a user wished to refer to X inside the inner block then

GLOBAL INT VAR X;

must appear within that block before such a reference.

Labels are in scope throughout the block or statement they label, including any inner block. This allows multilevel exits.

Resources are special in that they can have an existence independent of any NCL session. This occurs when the object (e.g. a file) is 'saved'

(i.e. made permanent). Also, any object that is permanent at the start of a session is global to that session and available throughout it. Thus the scope of a resource object may extend beyond the block in which it was created. For example,

```
BEGIN
     FILE VAR data:= (DEVICE:= DISK);
     myprog;                @ myprog fills data with information @
     SAVE(data);
END;
data.LIFETIME:= 2;
```

The internal name of a global object (i.e. the name used within a session or program) is the same as its external name (i.e. the name used by the operating system, its TITLE). For example,

```
BEGIN
     FILE VAR data:= (DEVICE:= DISK,
                      TITLE:= "One_two_three");
     myprog;                @ myprog fills data with information @
     SAVE(data);
END;
One_two_three.LIFETIME:= 2;
```

A4.3 Declarations Generally

A declaration may appear at any point in a session, but must appear before any reference to the declared object.

An object may be declared VAR or CONST. If it is declared CONST then the object becomes a constant and its value may not be changed.

A declaration may specify an initial value for the variable. If no initial value is specified then its value is undefined and an error will occur if an attampt is made to use it. A CONST object must be given an initial value.

If, during the scope of an object with some declared identifier, another declaration of the same identifier is encountered then an error occurs.

A GLOBAL declaration 'imports' a variable into an inner block (see section 4.2)

Examples:-

```
INT VAR x;
STRING CONST aname:= 'sheep';
FILE VAR f:= (DEVICE:= DISK,
              OWNER:= 'notme');
```

A4.4 Simples

'Simples' are unstructured objects of type integer, real, boolean or string. They can be declared explicitly in a normal declaration or implicitly by appearing on the left hand side of an assignment. For example:

```
INT VAR x:= 23;
STRING CONST name:= 'Peter';
found:= FALSE;  @ found is implicitly declared Boolean @
```

Integer and real variables can be used to hold numeric values. Implementation limits on the maximum and minimum values of such variables may apply although such limits should never be more restrictive than the following limits

Integers	- maximum	32767
	- minimum	-32767
Reals	- maximum	1×10^{10}
	- minimum	-1×10^{10}

The normal arithmetic operators (+, -, x, ÷) may be applied to integer and real variables (see section A5.2 for further details).

Boolean variables can be used to hold truth values, i.e. TRUE or FALSE. The operations of logical and (AND), or (OR), exclusive or (XOR) and not (NOT) may be applied to Boolean variables. Relational operators may also be applied to objects to produce truth values. (See section A5.3 for further details).

String variables can be used to hold character values. An implementation-dependent limit on the maximum number of characters a string can hold may apply although this limit should never be less than 256 characters.

A string literal may be given a character size (see sect B3.10) which indicates how many bits should be used for each character in the string. For example 1"1100 0001" would be stored one bit per character in eight bits of memory as the value $1100\ 0001_2$. This is equivalent to 8"A" which would also be stored in eight bits as $1100\ 0001_2$.

The allowable character sizes and the corresponding character coding system used are

Character size	coding system	
8	EBCDIC	
7	ASCII	Default
6	BCD	
4	hexadecimal	
3	octal	
1	binary	

Each coding system has a particular allowable character set. If any character in a literal is not in that character set an error occurs, e.g. 1"AB" is invalid and 4"6" is invalid.

The characters in a character set are ordered into a collating sequence. Which will affect the result of string comparisons (see sect A5.3).

Logical, concatenation, subtraction, head and tail and substring-extraction operators may be applied to strings (see section A5.4 for further details).

A4.5 Semaphores

Semaphore variables can be used to synchronise the operation of concurrently executing processes. A semaphore variable is rather like an INT VAR with several restrictions as follows:-

1. A semaphore may not be implicitly declared
2. A semaphore must be declared VAR.
3. A semaphore must be given a nonnegative initial value when it is declared.
4. The value of a semaphore may only be changed via the use of the GET and FREE built in procedures i.e. a semaphore may not be used in an assignment.
5. A semaphore formal parameter must be VAR.

A4.6 Resources

Resources are complex objects. A resource variable is composed of a number of 'attributes' each of which specify some characteristic of the resource object. These attributes are described in Appendix C.

Any object is uniquely identified by the following attributes:- SITE, OWNER, TITLE and for files GENERATION.

All attributes have default values, thus it is possible to specify a resource object without having to enter unneccessary details.

An attribute of a resource object may be specified by following the object name with a period and then the attribute name. For example

```
myfile.TITLE
```

or

```
bigplotter.WIDTH
```

Resource objects can be saved in the users directory as permanent objects.

A4.6.1 Files

An NCL FILE (i.e. a logical description of a file) can be connected to a container by setting the Boolean attribute OPEN to TRUE (called 'opening' the file). Setting it to FALSE 'closes' the file, breaking the connection to the container. For example

```
myfile.OPEN:= TRUE;     @ open myfile @
myfile.OPEN:= FALSE;    @ close myfile @
```

If the attribute NEW is TRUE then a new container is created when a file is opened.

If a temporary file is closed then it is removed from the users directory. If the removed file is an output file (e.g. a line printer) then it is output on the appropriate device before removal.

On the other hand if the file is permanent then it becomes global, if not already so. (See sect A4.2).

A4.6.2 Peripherals

A peripheral is a logical description of a physical I/O device, and may be connected to a file via the PERIPHERAL attribute. For example,

```
picture.PERIPHERAL:= bigplotter;
```

attaches the peripheral bigplotter to the file picture. Whenever any I/O is done to picture the peripheral defined by bigplotter is used.

The DEVICE attribute of the file a peripheral is attached to must be compatible with the peripheral. For example, it is not possible to connect a card reader to a line printer file.

A4.6.3 Environments

An environment is composed ot two sets of objects called status and limit objects. The status objects hold the current status of certain computational resources; for example COST and CPUTIME. The limit objects specify the maximum allowable values of the status objects; for example COSTLIMIT and CPULIMIT. See Appendix C for more details.

An environment can be connected to a process by setting the ENVIRONMENT attribute of the code file of the process. For example,

```
myprog.ENVIRONMENT:= shortjob;
```

When a process is activated, the environment connected to it is checked to see if it can fit within the global environment. For example if, refering to the example above, the user entered

```
myprog;   @execute myprog in the shortjob environment.@
```

then the attributes of shortjob are tested to see whether, for example, shortjob.CPULIMIT is less than the time remaining to the user in his current session.

A4.7 Procedures (see also section A6.1)

A procedure is a structured object composed of four elements as follows:

1. A file containing the text used to declare the procedure; declared thus:

   FILE CONST SOURCE (TITLE:= <procid>+"-SOURCE")

2. A file containing the executable code of the procedure, declared thus:

   FILE CONST CODE:= (TITLE:= <procid>)

3. A variable used to return a value from the procedure, declared thus:

   <return value mode> VAR RESULT;

   RESULT can be assigned a value only within the body of the procedure.

4. A label whose name is BODY which can be used to exit the procedure. BODY is in scope throughout the procedure's body.

For example,

```
PROCEDURE example:= (INT VAR x) STRING:
     BEGIN
       :
       :
     example.RESULT:= "ABC";
       :
       :
     IF
     x = 0 THEN
          EXIT example.BODY
     FI;
       :
       :
     END;
example.SOURCE.TITLE:= "gunge";
example;                    @ execute the procedure and discard the result
example.CODE;               @ execute the procedure and discard the result
mystring:= example.RESULT;  @ execute the procedure and assign the result
                              to mystring @
```

SOURCE and CODE are not created for a procedure declared within another procedure. If the mode of a procedure is VOID then using RESULT causes an error.

The parameter of a procedure may be VAR or CONST. A CONST parameter can be used to pass a value to a procedure whereas a VAR parameter can be used to pass a reference to an object. A semaphore parameter must be VAR.

Parameters may have optional keywords and defaults. See section A6.1 for their use.

There are several standard system procedures which perform various complex functions for the user. The standard procedures are described in Appendix D.

There is also a number of built-in procedures. These procedures are performed directly by the NCL system (i.e. no code files exist for them).

The built-in procedures are also described in Appendix D. Several of these procedures perform coercion functions and as such are described in section A4.11.

A4.8 Data Structures

A4.8.1 Data Structures Generally

A data structure may be an array, set or a queue. Each may have zero or more objects which must all be of the same mode.

The number of elements in a data structure may change whenever an assignment is made to the whole structure. The number of elements in a structure is returned by the built-in procedure COUNT.

For example: - if we have an array A declared thus

```
ARRAY OF INT VAR A: = [1, 2, 6, 8, -2]
```

then

```
COUNT(A).RESULT = 5
```

After the following assignment

```
A:= A+[10]  *
```

then

```
COUNT(A).RESULT = 6
```

---

* Structures may be added together. See section A5.5 for details.

Implementation dependent limits on the maximum number of elements allowed in a structure may apply although these limits will never be less than 1024 for all structure types.

A4.8.2 Arrays

An array is a linear structure, each element being accessable by its index. Thus

arrayid [x] = the xth element.

For example, if we declare an array A thus

ARRAY OF INT VAR A:= [6, 2, 1, 8, 7]

then

A[3] = 1

A[COUNT(A)] = 7

The index of the first element is 1.

A4.8.3 Sets

The elements of a set must have unique values. If an element is assigned a value identical to any other element in the set a warning is issued and the value of the element becomes undefined.

The elements in a set are ordered according to their values. It is possible to select an element on the basis of its position in

this order.

Thus

setid|x| = the element xth in order

setid|FIRST| = the element first in order

setid|LAST| = the element last in order

For example if a set S is declared thus

```
SET OF INT VAR S: = [1, 10, 3, 12, 8]
```

then

S|2| = 3

S|FIRST| = 1

S|LAST| = 12

It is invalid to assign a value to a set element. (e.g. S|3|:= 6;) Doing so will cause an error.

A set is thus very useful for holding objects which must be unique; a prime example being the files in a users directory.

Whenever a file (or any resource object) is assigned to a set, if the file's title does not begin with the set name then the set name is added to the file title as the initial subpart. For example if we have a file and a set declared thus

```
FILE VAR myfile:= (...,TITLE:= 'scan/three');
SET OF FILE pictures;
```

and we perform the following assignment

```
pictures:= pictures + [myfile]
```

then the title of myfile becomes 'pictures_scan_three'.

If, at log-on a user's directory contains resource objects with hierachical names then appropriate sets are implicitly declared.

For example if the user owns files with the following titles

sheep_ticks_mob1

sheep_ticks_mob2

sheep_weights_mob1

sheep_weights_mob2

sheep_mobinfo

Then the following declarations are made implicitly at logon

```
SET OF FILE sheep:= [sheep_ticks_mob1,
                     sheep_ticks_mob2,
                     sheep_weights_mob1,
                     sheep_weights_mob2,
                     sheep_mobinfo];

SET OF FILE sheep_ticks:= [sheep_ticks_mob1,
                           sheep_ticks_mob2];

SET OF FILE sheep_weights:= [sheep_weights_mob1,
                             sheep_weights_mob2];
```

A4.8.4 Queues

Queues are linearly ordered structures of non-unique elements. The ordering of a queue is on a first in first out basis (i.e. the first element added to the tail is the first element to reach the head). To allow greater flexibility it is also possible to add to head and take from the tail.

It is only possible to access the first or last element in a queue. Thus

queueid|FIRST| = the element at the head of the queue

and queueid|LAST| = the element at the tail of the queue.

For example if we have a queue, myq, declared like so

```
QUEUE OF INT VAR myq:= [5, 7, 8, 2, 6]
```

then myq|FIRST| = 5

and myq|LAST| = 6

An element may be added to the tail of a queue, thus after executing myq:= myq + [10] then myq|LAST| = 10.

Similarly for the head of the queue. Thus if myq:= [7] + myq then myq|FIRST| = 7. Assigning values directly to the head or tail (e.g. myq|LAST|:= 5 or myq|FIRST|:= 18) is invalid and will cause an error.

An element may be removed from a queue by using the builtin function REMOVE. For example if

```
X:= REMOVE(myq|FIRST|).RESULT
```

then X = 5

and myq = [7, 8, 2, 6] .

A4.9 Miscellaneous Objects

A4.9.1 Labels

Labels name structured statements i.e. blocks and IF, CASE, ON and LOOP statements.

A label can only be used in an EXIT statement which when encountered causes execution of the labeled statements to be terminated.

A label can only be referenced within the statement it labels.

With a procedure <proc id>.BODY labels the body of the procedure i.e. EXIT <proc id>.BODY causes execution of the procedure body to terminate.

A4.9.2 Incremental Guard Variables

In incremental guards the control variable (i.e. the FOR variable) is implicitly declared, and can only be accessed within the loop body, where it is treated as INT CONST i.e. you can't assign to it within the loop body). For example, in the following

For example in the following

```
LOOP
FOR X FROM 1 TO 10
      :
      :
    X:= 20;  *
      :
      :
POOL
```

The line marked * is invalid and will produce an error.

A4.9.3 Standard Objects Generated at Logon

The following standard objects are always implicitly declared in the users outer block.

1. INT VAR RESULT:= 0;

   This variable is assigned a value after the execution of any procedure or user program. If an error occured during execution the value will be negative otherwise it will be zero. The value assigned will indicate the type of error that occured. Positive values may be used for user purposes.

2. STRING VAR MESSAGE

   This contains the last message sent to the session log.

3. ARRAY OF STRING VAR ADDRESS

   This contains the address to which session output will be sent. At logon it is assigned the address of the session account. Each array element contains one line of the address.

4. PERI VAR TERMINAL

   The peripheral from which the session is being run.

5. ENV VAR PROFILE

   The environment which controls the session.

A4.10 Coercions

In NCL there is but one implicit coercion as follows

Integer to Real. If an integer expression occurs where a real expression is required its value is changed to a real value implicitly.

Various explicit coercions may be performed by using the builtin functions provided for the purpose as follows:

Real to Integer.

(i) ROUND(<NUM-expression>)

corrects the expression to an integer by rounding up or down to the nearest whole number.

(ii) TRUNC(<NUM-expression>)

converts the expression to an integer by simple truncation of the fractional part.

Note:- ROUND(X) ≡ TRUNC(X+0.5)

String to Numeric

(i) CHARINT (<string-expression>

converts the character string to the integer value it prepresents. Thus CHARINT("13") = 13_{10}

and CHARINT("-210") = -210_{10}

(ii) CHARREAL (<string-expression>)

converts the character string to the real value it represents. Thus CHARREAL("6.8") = 6.8_{10}

and CHARREAL("-0.50") = -0.5_{10}

(iii) BIN(<string expression>

Converts the binary image of the string to an integer. Thus BIN("A") = 193_{10}

and BIN("") = 0_{10}

Numeric to String

(i) INTCHAR(<num expression>)

Converts the expression to a string which represents its value. Thus INTCHAR(7*2) = "14"

and INTCHAR(-1*16.3) = "-16"

(ii) REALCHAR(<num expression>)

Does the same as INTCHAR except for real expressions.

Thus REALCHAR(-1*16.3) = "-16.3"

(iii) STRING(<int expression>)

Converts the binary image of the expression into a string. Thus STRING(193) = "?????A". The length of the string depends upon the word length of the host computer.

Integer to/from Array of Bool

(i) INTBOOL(<int expression>)

Converts the binary image of the absolute value of the expression into an array of bool. The least significant bit of the integer corresponds to the array element number 1. If a bit is on/off the corresponding array element is TRUE/FALSE.

Thus INTBOOL (1) = [TRUE, FALSE, FALSE....]

and INTBOOL (3) = [TRUE,FALSE,TRUE,FALSE,FALSE....]

(ii) BOOLINT(<array bool expression>)

Does the reverse of INTBOOL.

Thus BOOLINT([TRUE]) = 1_{10}

A5 Expressions

A5.1 Expressions Generally

An expression defines a method of obtaining a value. The mode of the value depends upon the modes of the objects used in the expression and the operations performed upon them.

The following sections detail the operators which can be applied to objects. Each operator is given a 'priority'. An operator with a high priority is applied before an operator with a low priority. This rule may be overridden by using parentheses, (and), which cause the expression within them to be evaluated before any operators outside them may be applied.

For example

$$3 + 5 * 2 = 3 + 10 = 13$$

whereas $(3 + 5) * 2 = (8) * 2 = 16$

A5.2 Numeric Operators (see section B3.8)

The numeric operators all apply to int's and real's only. The normal arithmetic operators may be used as follows

multiplication	*	priority 3
division	/	
addition	+	priority 2
subtraction	-	

unary plus +
unary minus - } priority 2

If either operand is real the mode of the result is real otherwise the mode is int

Examples:-

Imagine the following declarations

```
INT VAR X, Y, Z;
REAL VAR P, Q, R;

X:= -3;     @ unary minus @
Z:= 2+X;    @ value of Z is -1 @
P:= 16/(Z*X) @ value of P is 5 @
```

A5.3 Boolean Operators (See section B3.9)

The mode of the result of all the boolean operators is bool.

The operators can be grouped as follows.

(i) Unary negation. Priority 3, mode of operand must be bool.

NOT changes TRUE to FALSE or vice versa.

(ii) Logical operators. Priority 2, mode of operands must be bool.

AND result is TRUE only if both operands are TRUE

OR result is TRUE if either operand is TRUE

XOR result is TRUE if either, but not both operands are TRUE

(iii) relations. Priority 1

The following operators give a result of TRUE if their operands satisfy the relationship defined by the operator.

Operator	relationship
= or EQ	is equal to
< or LT	is less than
> or GT	is greater than
<= or LE	is less than or equal to
>= or GE	is greater than or equal to
<> or NE	is not equal to

The operands of the above operators may be either both numeric or both strings. If the operands are strings then

1 If one string is shorter than the other then it is considered to be padded at the right with enough blanks to make their lengths equal.

2 The strings must both be of the same character size (see sect A4.4).

3 The result of a string comparison depends upon the character set used and its colating sequence. It should always be that if S1 and S2 are strings and S1 < S2 is TRUE then BIN(S1) < BIN(S2) is also true. Similarly for the other relational operators.

string 1	WITHIN string 2	result is TRUE if string 1 forms all or part of string 2
string 1	STARTS string 2	result is TRUE if the first LENGTH(string 1) characters of string 2 equals string 1

string 1	ENDS string 2	result is TRUE if the last LENGTH(string 1) characters of string 2 equals string 1

With the above three operators if the first operand or both operands are null strings the result is TRUE and if the second operand alone is a null string the result is FALSE.

There are two more relational operators as follows

IS	result is TRUE if the first operand is of the mode specified by the second operand.
IN	result is TRUE if an element of the set, specified by the second operand, is equal to the first operand.

Examples:-

If we have the following declarations

```
BOOL VAR          b:= TRUE;
                  c:= FALSE;

INT VAR           x:= 3 ;
STRING CONST      s:= "MASSEY" ;
SET OF INT VAR    si:= [1, 6, 3, 2, 5] ;
```

then the following expressions have the value specified

Expression	Value
b AND c	FALSE
x = 3	TRUE
s < "VICTORIA"	TRUE
"ASS" WITHIN s	TRUE
"MAT" STARTS s	FALSE
"SEY" ENDS s	TRUE
"" ENDS s	TRUE
x IS INT	TRUE
x IN si	TRUE

A5.4 String Operators (see section B3.10)

String operators take string operands only and produce string results.

The operators can be grouped by priority as follows.

(i) Trimmers. Unary operator. priority 3

A trimmer is a substring operator; that is the result of a trimmer is a substring of the operand.

The form of a trimmer is

[start : finish]

where start and finish are int expressions.

Start and finish must both be positive.

If finish is less than start the result is a null string.

Start specifies the index of the first character of the substring required and finish specifies the index of the last character.

For example: -

"ABCDE" [2 : 4] = "BCD"

and '"HELLO" [3 : 3] = "L"

(ii) Concatenation etc. priority 2

Concatenation

S1 + S2 the operands are concatenated into one string

Subtraction

S1 - S2 the first occurrence of S2 with S1 is removed from S1. If S2 is null or S2 is not in S1 the result is S1.

Extract the head

S1 BEFORE S2 result is the string consisting of all the characters in S1 which come before the first character of the first occurrence of S2 within S1.

Extract the tail

S1 AFTER S2 result is the string consisting of all the characters in S1 after the last character of the first occurrence of S2 within S1.

With BEFORE and AFTER if the second operand doesn't occur in the first operand then the result is a null string.

(iii) logical operations priority 1

The three logical operators AND, OR and XOR may be applied to strings.

The operation in performed bit by bit on the binary images of the two operands.

If one string is shorter than the other then it is padded to the right with binary 0's.

Examples

Imagine the following declarations

```
STRING VAR  S1:=  "HELLO",
            S2:=  "THERE",
            S3:=  "HELLO THERE!";
```

Expression.	Value
S1 + " " + S2 + "!"	"HELLO THERE!"
S3 - "E"	"HLLO THERE!"
S1 BEFORE "LO"	"HEL"
S2 AFTER "ERE"	""
1"11110101" AND 1"11110011"	1"11110001"

A5.5 Structure Operators (See section B3.12)

Structure operators take structure operands of the same mode and produce a structure of that mode as a result.

Arrays

concatenation op1 + op2. result is an array composed of the elements in the first operand followed by the elements in the second operand.

queues

Concatenation op1 + op2. result is a queue composed of the elements in the first operand followed by the elements in the second operand. Thus

result|FIRST| = op1|FIRST|

and result|LAST| = op2|LAST|

sets

union op1 + op2. result is a set composed of the elements in both sets. An element common to both sets will appear in the result once only.

difference op1 + op2 result is a set composed of the elements in op1 not in op2. If op2 = op1 the result is a set with no elements. If op2 is empty the result equals op1. If op1 is empty the result is empty.

intersection op1 * op2 result is a set composed of the elements in op1 and also in op2. If either operand is empty the result is empty. If there are no common elements the result is empty.

Examples

If we have the following declarations

```
ARRAY OF INT VAR    ary:= [1, 2, 5, 7],
                    prices:= [3, 7, 9, 10];
```

```
QUEUE OF FILE VAR      x_to_y := [F1, F2],
                       y_to_x := [F3];

SET OF FILE VAR        mine:= [mine_x, mine_y],
                       yours:= [mine_y, his_z];
```

then the following expression have the given values.

Expression	Value
ary + [8]	[1, 2, 5, 7, 8]
ary + prices	[1, 2, 5, 7, 3, 7, 9, 10]
x_to_y + y_to_x	[F1, F2, F3]
[F8] + y_to_x	[F8, F3]
mine + yours	[mine_x, mine_y, his_z]
yours + [his_q]	[mine_y, his_z, his_q]
mine - yours	[mine_x, his_z]
mine * yours	[mine_y]

A6 Statements

A6.1 Procedure Calls (See section B3.5)

A6.1.1 Procedure calls generally

A procedure call consists of the naming of a procedure or a codefile, plus the specification of parameters (if the procedure has any).

A procedure call causes the named procedure, or code file, to be executed.

If the procedure is not a void procedure (i.e. it returns a value and is thus a function) then if the value returned is desired then the call must be followed by .RESULT. For example:-
if we have a procedure p declared as follows

```
PROC p = INT: BEGIN ........END
```

then p.RESULT is an object of mode INT the value of which may be found by executing the procedure p. Thus p could be used as follows (assume INT VAR X)

```
p;  @ execute p and discard the result @
X: = p.RESULT:   @ execute p and assign result to X @
X: = p;  @ is in error as p is not an INT @.
```

Further if p is a procedure which returns a code file (i.e. p is a compiler) then

```
p;
```

means execute p and discard the result whereas

```
p.RESULT;
```

means execute p and then execute the result.

A6.1.2 Parameters of procedures

In the following discussions the term 'formal' parameter refers to parameters as found in procedure declarations, and 'actual' parameter refers to parameters as found in procedure calls.

Each formal parameter may or may not have

1. a keyword
2. a default value.

If a formal parameter has a keyword then the the actual parameter for it may be specified using the keyword (i.e. <key id>:= value) and may be positioned anywhere in the parameter list.

If a formal parameter has a default then an actual parameter does not have to be supplied for it.

For example: consider

```
PROCEDURE whatsit = (REAL CONST x KEY size DEFAULT 10;
                     FILE VAR f KEY infile;
                     STRING CONST s DEFAULT "ABC";
                     INT VAR t KEY time DEFAULT 1500)
          VOID:
```

the procedure can be called specifying the parameters by two methods

1. Positional. whatsit(,card,"STOP");
   which is equivalent to whatsit(10,card,"STOP",1500)

2. Keyword. whatsit(infile:=card, time:=1400)
   which is equivalent to whatsit(10,card,"ABC",1400)

The two methods may not be combined
For example whatsit(infile:=card,"STOP") is invalid.

A6.2 Structured Statements (See section B3.6)

A6.2.1 Blocks

A block is a group of statements bracketed by BEGIN and END. Any object which is to be used within a block must be declared there

before any reference to it.

Blocks may be nested. To allow an object declared in an outer block to be used that object must be 'imported' into the inner block by the use of a GLOBAL declaration (see section A4.2)

A6.2.2 Conditional Statements

There are three types of conditional statement; the IF, CASE and ON. The bodies of each of these statements are a series of guarded groups. A 'Guarded group' is a group of statements which are 'protected' by a guard which is a boolean expression. The statements can only be executed if the guard evaluates to TRUE.

Guarded groups can only occur in conditional statements. In an IF statement the body is scanned once and all groups for which the guard is TRUE are executed. In a CASE statement the body is scanned once and only the first group for which the guard is TRUE is executed. An ON statement acts continuously and has an associated scope, which is the block in whic it appears, plus any inner blocks. An ON statement is executed after each of the statements in its scope. When executed it is like an IF statement. Thus an ON statement can be used to test for exception conditions. If an ON statement tests the same condition as an earlier ON statement then the newer one overrides the earlier one.

Examples:-

```
IF
x = 3 THEN
     b:= TRUE;
     SAVE(myfile);
,datafile.OPEN = TRUE THEN
     datafile.OPEN:= FALSE;
FI;
```

```
CASE
NOT b:
     x:= 24;
,x = 6:
     myfile.LIFETIME:= 2;
ELSE:
     SEND("ERROR", CONSOLE);
ESAC;


ON
x < 0:
     SEND("ERROR #3",HIS_TERMINAL);     @ line C @
,x = 0:
     x:= x + 1;                         @ line D @
NO;

x:= 3;
x:= x-3;                                @ line A @
x:= x-2;                                @ line B @

ON
x = 0:
     NULL;
NO;

x:= 0;                                  @ line E @
```

When line A is executed x becomes zero, the second ON condition is true and line D is executed. When line B is executed x becomes -1 and the first ON condition is true and line C is executed. When line E is executed no action is taken.

A6.2.3 Cycles

A cycle can be used to set up iterative loops. For example,

```
LOOP
FOR count FROM 1 TO 10 BY 1     @ line A @
  :  } body of loop
  :  }
POOL;
```

The body of the loop is executed 10 times.

Line A is an incremental guard. It is also possible to have 'while guards' and 'until guards'.

For example

```
LOOP
WHILE  x > 0
:
POOL;
```

and

```
LOOP
FOR  count FROM 1 BY 2
:
UNTIL  array [count] > 100
POOL;
```

'While guards' and 'Until guards' may be used with or without incremental guards but they may not be used together in the same loop.

An incremental guard specifies how many times the body of the loop will be executed. This is done by incrementing an INT variable from one value to another by a fixed increment. The guard is composed up to four phrases as follows

(i) FOR <id>. This phrase specifies the name of the INT variable that is incremented. This variable is treated as INT CONST within the loop body. (see section A4.9.2). If the phrase is omitted an unnamed default variable is implicitly declared.

(ii) FROM <int expression>. This phrase specifies the initial

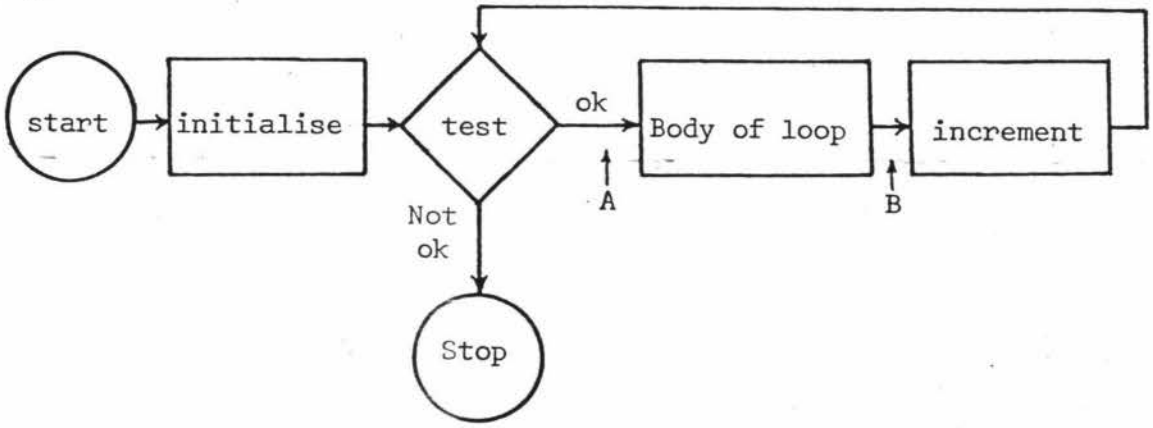
value which the control variable is to take. The default value is 1.

(iii) TO <int expression>. This phrase specifies the largest value which the control variable may be incremented to. If after incrementing the control variable its value is greater than the value specified by this clause then the loop is terminated. The default value is the maximum integer value.

(iv) BY <int expression> This phrase specifies the value by which the control variable is incremented at every iteration of the loop. The default value is 1.

If no FOR, FROM, TO or BY clauses are used none of the defaults apply and the loop is not controlled by a control variable.

The action of an incremental guard is shown in the following diagram.



A while guard is simply a boolean expression. If the expression evaluates to FALSE the loop is terminated. An Until guard is exactly the same except that a while guard is evaluated at point A in the above diagram while an until guard is evaluated at point B.

It is also possible to have no guards at all. Such a loop can only be terminated by the appropriate EXIT statement.

For example,

```
stage_right:
LOOP
  :
  :
IF
b = a THEN
     EXIT stage_right;
FI;
  :
  :
POOL;
```

A6.2.4 Parallel Statements

The body of a parallel statement consists of two or more statements seperated by semicolons. These statements are executed concurrently. When all the statements have been completed execution of the parallel statement terminates.

The order in which execution of the statements is started is undefined.

Communication between concurrent processes, as initiated by the parallel statement, may be accomplished by using semaphores and the GET and FREE builtin procedures.
(See section A4.5.).

A6.3 Assignment Statement (see section B3.7)

An assignment statement can be used to change the value of a variable (except semaphores. See section A4.5.).

The expression (which must be the same mode as the variable) is evaluated and assigned to the variable. If an error occurs during evaluation the assignment is stopped and the value of the variable becomes undefined.

It should be noted that resource variables are actually descriptions and not objects (see section A4.6.). Thus the assignment of one resource variable to another merely causes both descriptions to reference the same object. For example:- if f1 and f2 are FILES then

```
f1:= f2;
```

means that f1 now references the same file as f2. If f1 is open then it is closed before being assigned the value of f2.

If f2 is a codefile the above assignment does not mean 'execute f2 and assign the result to f1'. To attain this requires the following

```
f1:= f2.RESULT;
```

(which will work as long as f2 returns a file). (see section A6.1.1.)

A6.4 Wait Statement (see section B3.7)

A wait statement causes the process executing it to be suspended for the specified time.

The accuracy of the pause depends upon the host computer and its Operating System.

A6.5 Exit Statement (see section B3.7)

A exit statement causes execution of the statement designated by the label to terminate.

As a label is only in scope within the statement it labels it follows that the exit statement can only cause the termination of a statement it is found within.

Multilevel exits are allowed. For example

```
L1:
BEGIN
  :
  :
    L2:
    LOOP
      :
      :
    IF
    x = 0 THEN
        EXIT L1;
    FI
      :
      :
    POOL;
  :
  :
END;
```

The exit statement, if executed, causes execution of the block L1 to terminate.

A6.6 Null Statement (see Section B3.7)

The null statement consists of the word NULL or nothing at all. It has no semantic effect.

A null statement can be used in a case statement where the user wants for completeness to specify a case for which nothing is done.

e.g.

```
CASE
x = 3: proc1,
x = 4: proc2,
x = 5: NULL,
x = 6: proc3
ESAC;
```

APPENDIX B

NCL FORMAL LANGUAGE DESCRIPTION

CONTENTS

B1 Introduction

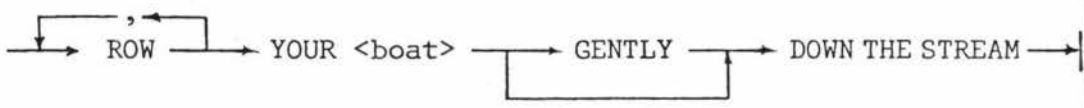
The following is a formal description of the context free syntax of NCL. The language is subject to several context sensitive constraints which are described in Appendix A.

This definition is given using a form of tramline notation with the following conventions

(i) A tramline is a diagramatic rule which defines a non-terminal. A tramline is composed of arrows, non-terminals (enclosed within brackets, <>) and terminal symbols. Trace any path around a tramline (always following the arrows) and the non-terminal and terminal symbols encountered when strung together can replace the nonterminal defined by the tramline.

For example

<song>::=



means that <song> can be replaced by

ROW YOUR <boat> DOWN THE STREAM

or ROW YOUR <boat> GENTLY DOWN THE STREAM

or ROW, ROW YOUR <boat> DOWN THE STREAM

etc

(ii) Terminal symbols are not enclosed with brackets and are symbols of the language.

(iii) ⟶Δ indicates that the tramline is continued from the corresponding Δ⟶ further on.

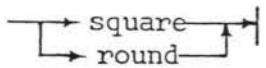
(iv) ⟶| indicates the end of the definition.

(v) Any text within curly brackets,{}, expresses rules which cannot be expressed in tramline notation.

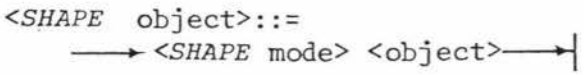
(vi) A capitalised italic word within the brackets of a non-terminal is a 'metanotion'. Metanotions are used to express several essentially similar rules in one definition.

For example:- where the metanotion SHAPE is defined thus

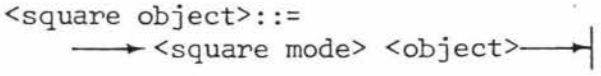
SHAPE ::=



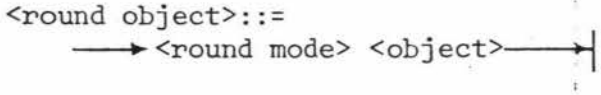
then



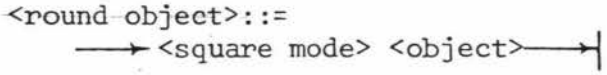
defines two rules



and



When transversing a definition the same selection for a metanotion must be made throughout. For example, in the above rule, both occurrences of SHAPE must be replaced by the same string. Thus

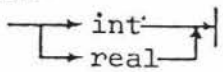


is <u>not</u> a valid rule.

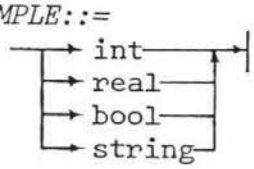
A metanotion may itself be defined in terms of other metanotions.

B2 The Metanotions

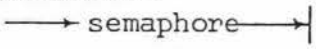
NUM::=



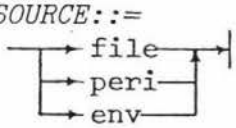
SIMPLE::=



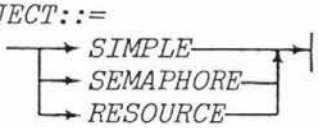
SEMAPHORE::=



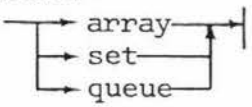
RESOURCE::=



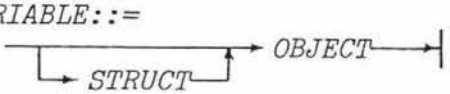
OBJECT::=



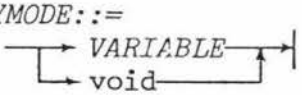
STRUCT::=



VARIABLE::=



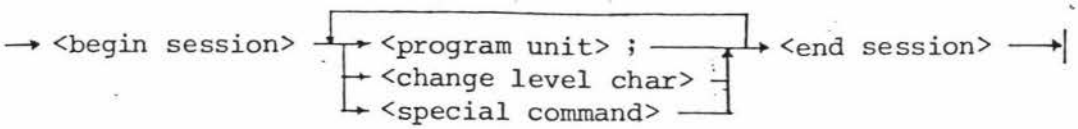
ANYMODE::=



B3 The Syntax

B3.1 Sessions

<sessions>::=



<begin session>::=

→ {HELLO proc call as defined in the standard proc specs} →|

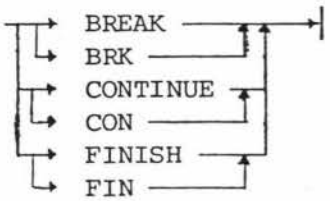
<end session>::=

→ {GOODBYE proc call as defined in the standard proc specs} →|

<change level char>::=

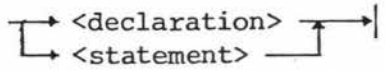
→ ? →|

<special commands>::=

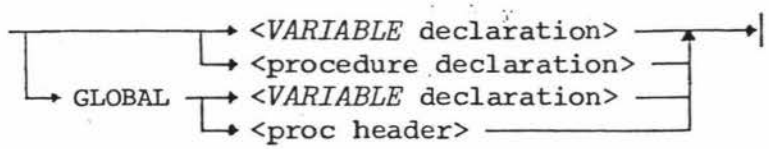


B3.2 Program Units

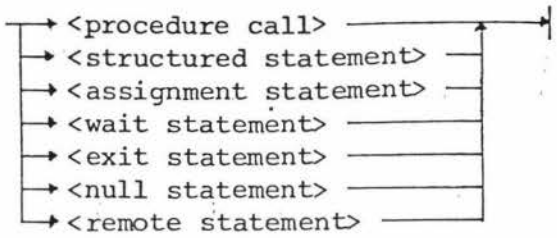
<program unit>::=



<declaration>::=

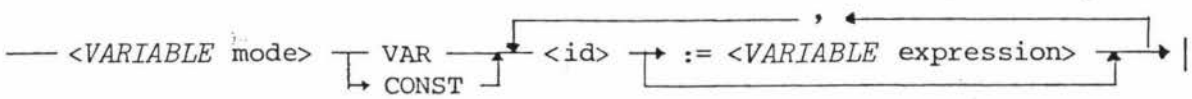


<statement> ::=



B3.3 Declarations

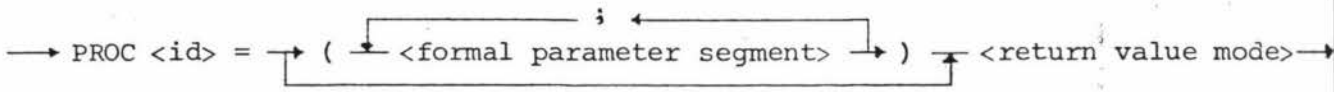
<*VARIABLE* declaration> ::=



<procedure declaration> ::=

→ <proc header> : <proc body> →

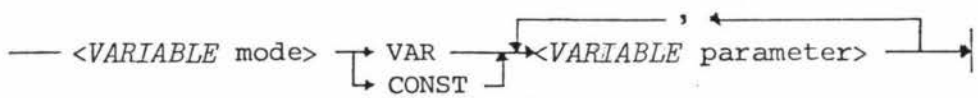
<proc header> ::=



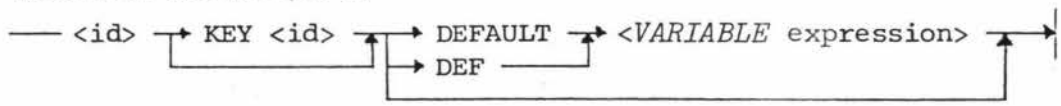
<proc body> ::=

→ <block> →

<formal parameter segment> ::=



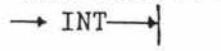
<*VARIABLE* parameter> ::=

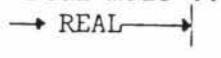


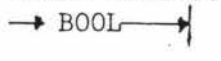
<return value mode> ::=

→ <*ANYMODE* mode> →

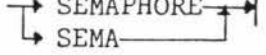
B3.4 Modes

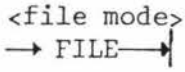
<int mode>::=


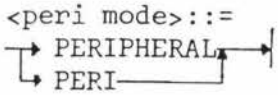
<real mode>::=


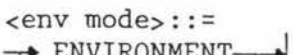
<bool mode>::=


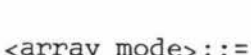
<string mode>::=

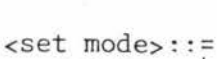

<semaphore mode>::=


<file mode>::=


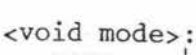
<peri mode>::=


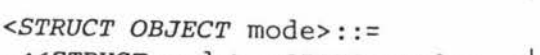
<env mode>::=


<array mode>::=


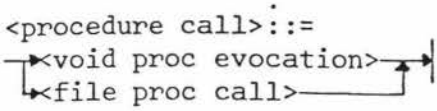
<set mode>::=


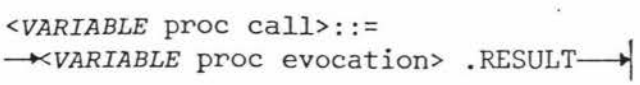
<queue mode>::=

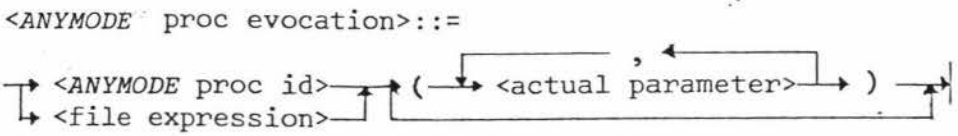

<void mode>::=


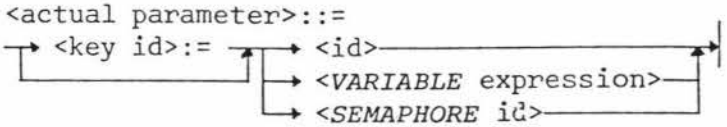
<*STRUCT OBJECT* mode>::=


B3.5 Procedure Calls

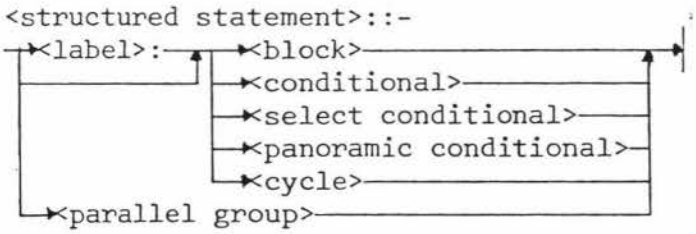


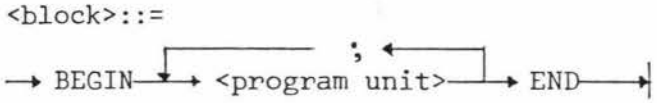


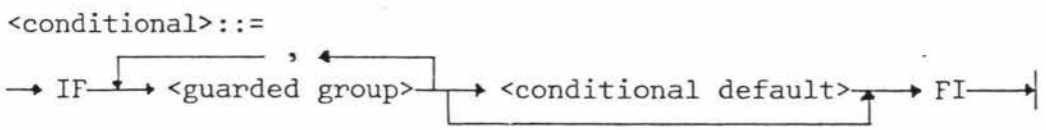


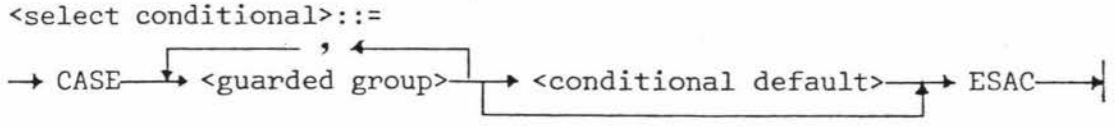


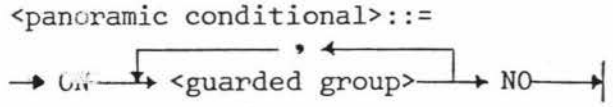
B3.6 Structured Statements

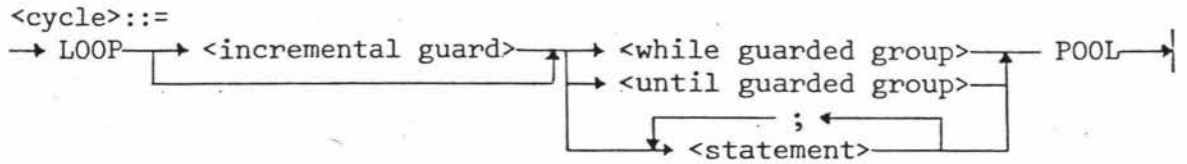


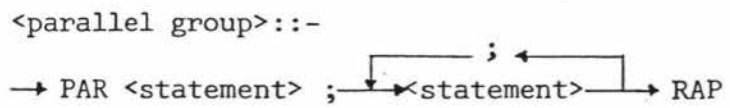




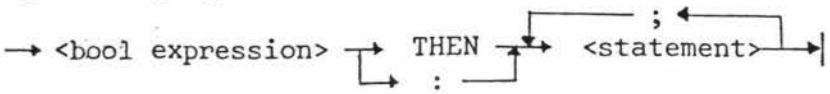




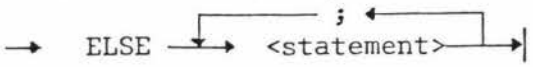




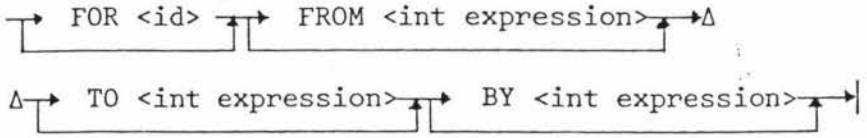
<guarded group>::=



<conditional default>::=



<incremental guard>::=

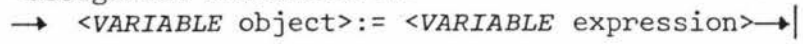


<while guarded group>::-

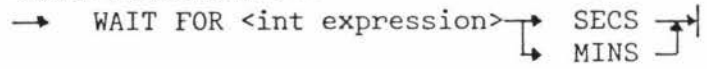
<until guarded group>::=

B3.7 Other Statements

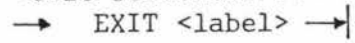
<assignment statement>::=



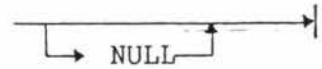
<wait statement>::=



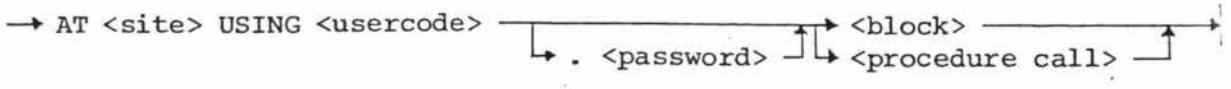
<exit statement>::=



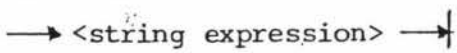
<null statement>::=



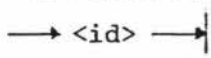
<remote statement>::=



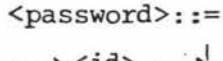
<site>::=



<usercode>::=

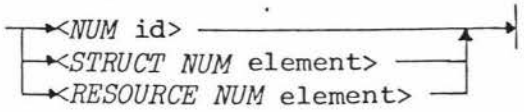


<password>::=

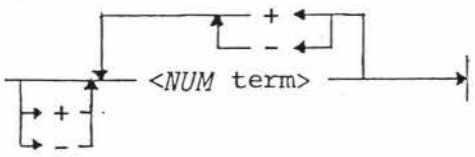


B3.8 Numeric Objects and Expression

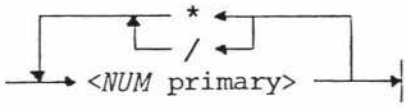
<*NUM* object>::=



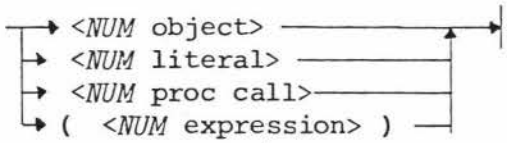
<*NUM* expression> ::=



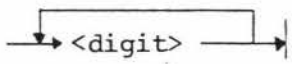
<*NUM* term>::=



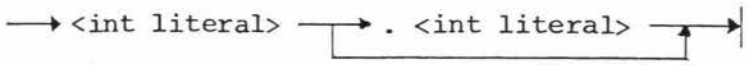
<*NUM* primary>::=



<int literal>::=

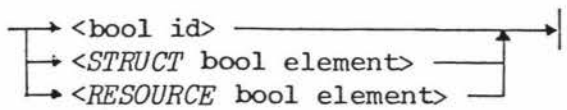


<real literal>::=

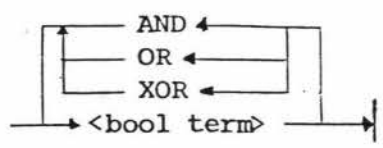


B3.9 Boolean objects and expressions

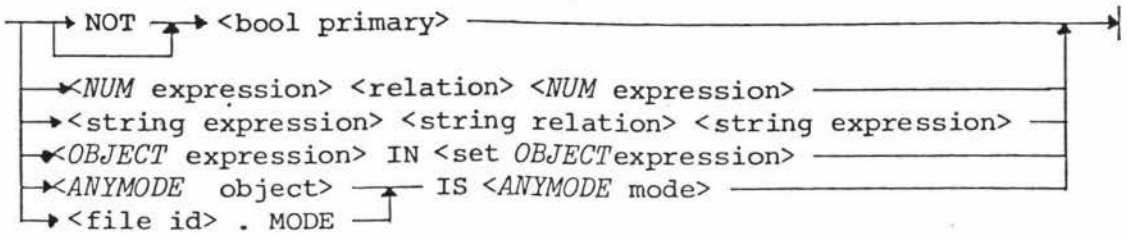
<bool object>::=



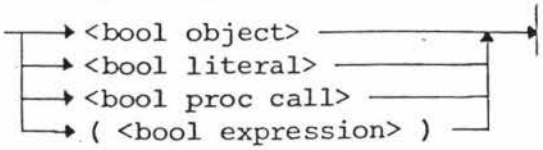
<bool expression>::=



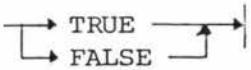
<bool term>::=



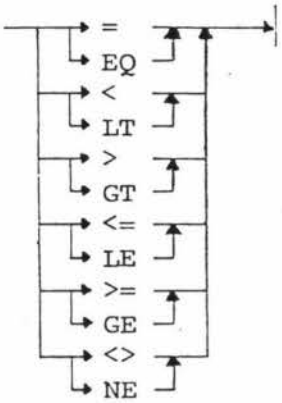
<bool primary>::=



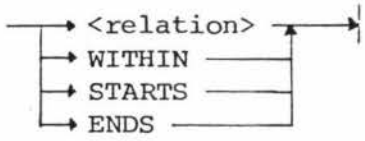
<bool literal>::=



<relation>::=

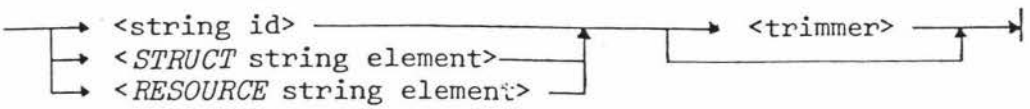


<string relation>::=

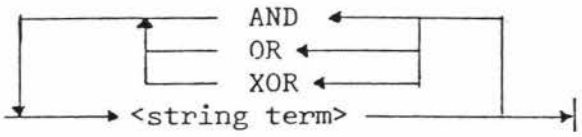


B3.10 String Objects & Expressions

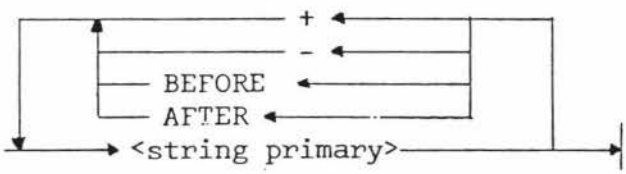
<string objects>::=



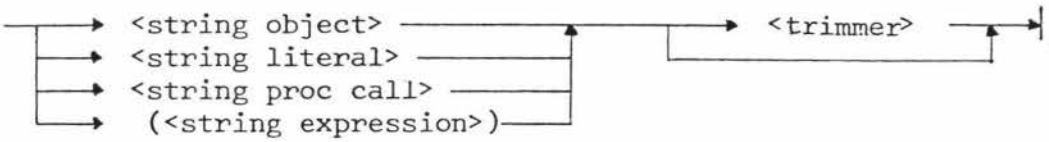
<string expression>::=



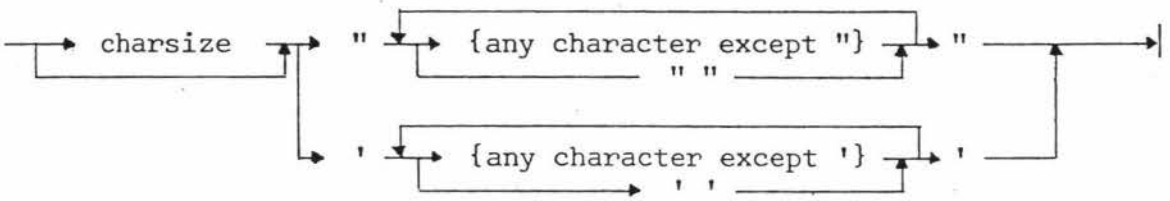
<string term>::=



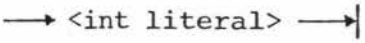
<string primary>::=



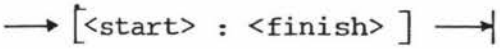
<string literal>::=



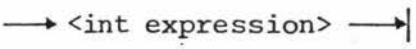
<charsize>::=



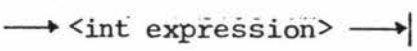
<trimmer>::=



<start>::=

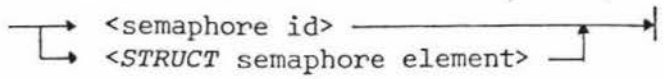


<finish>::=

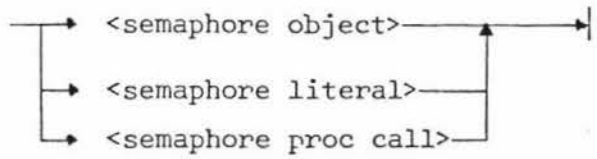


B3.11 Semaphore Objects & Expressions

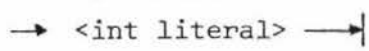
<semaphore object>::=



<semaphore expression>::=

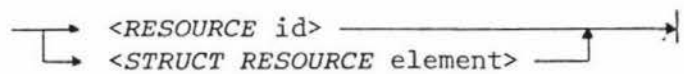


<semaphore literal>::=

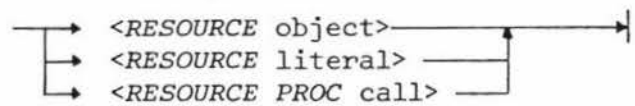


B3.12 Resource Objects & Expressions

<*RESOURCE* object>::=

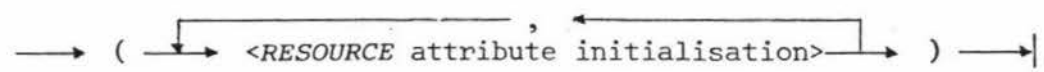


<*RESOURCE* expression>::=

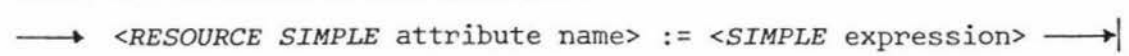


<*RESOURCE SIMPLE* element>::=

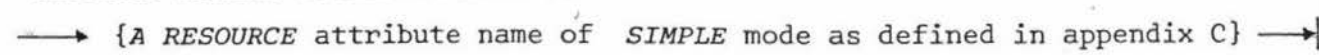
<*RESOURCE* literal>::=



<*RESOURCE*attribute initialisation>::=



<*RESOURCE SIMPLE* attribute name>::=

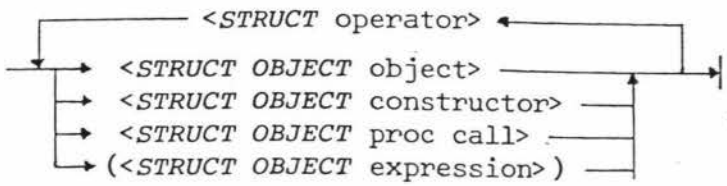


B3.13 Structure Objects & Expressions

<*STRUCT OBJECT* object>::=

→ <*STRUCT OBJECT* id> →|

<*STRUCT OBJECT* expression>::=



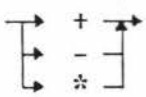
<array operator>::=

→ + →|

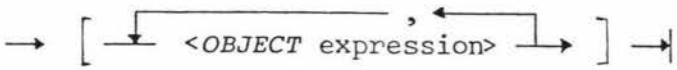
<queue operator>::=

→ + →|

<set operator>::=



<*STRUCT OBJECT* constructor>::=



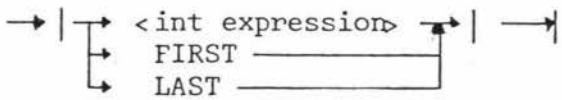
<*STRUCT OBJECT* element>::=

→ <*STRUCT OBJECT* id> <*STRUCT* element selector> →|

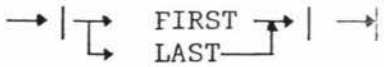
<array element selector>::=

→ [<int expression>] →|

<set element selector>::=



<queue element selector>::=



B3.14 Identifiers

<label>::=

→ <id> →|

<key id>::=

→ <id> →|

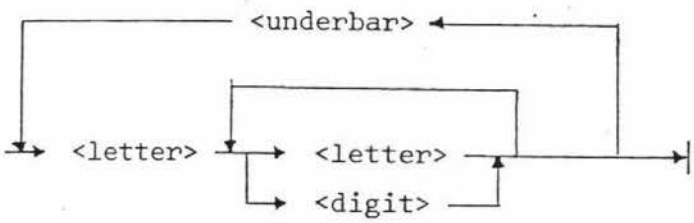
<*ANYMODE* id>::=

→ <id> →|

<*ANYMODE* proc id>::=

→ <id> →|

<id>::=



<underbar>::=

→ _ →|

<letter>::-

→ {any one of the 26 letters of the alphabet in either upper or lower case} →|

<digit>::=

→ {any one of ten numerical digits 0 to 9} →|

APPENDIX C

NCL RESOURCE ATTRIBUTES

This appendix describes the attributes of the resource objects, i.e. files, peripherals and environments.

Several attributes have values which have a meaning beyond the meaning of the value itself. For example CARRIAGECONTROL is an INT attribute and may be assigned the values, 0, 1 or 2; 0 means no carriage-control, 1 means ASA type and 2 means standard carriage-control. To aid the user a number of constants are implicitly declared at logon which may be used when assigning values to such attributes. For example:=

```
INT CONST
     NONE:= 0,
     ASA:= 1,
     STANDARD:= 2;
```

Thus a user could make the following assignment:-

```
f. CARRIAGECONTROL:= ASA;
```

If an attribute has such constants declared for it they are listed after the attribute description; their values, being arbitrary, are not listed. In several instances we will refer to a file of basic peripheral type. This means a file whose device attribute specifies a hard-copy output peripheral i.e. cardpunch, plotter, lineprinter, papertapepunch, etc.

E1 File attributes

STRING VAR ACCESS

specifies who is allowed access to the file and what sort of access they are allowed. For example:

f.ACCESS:= "OWNER (R & W), SMITH (R), BROWN (R & W)"

INT VAR BLOCKSIZE

specifies the size of physical blocks in units of UNITTYPE.

INT VAR CARRIAGECONTROL

specifies the type of format control characters appearing on each record for a file of basic peripheral type.

Constants:= NONE, ASA, STANDARD.

CONTENTS

this untyped component may only be assigned the following form of literal

```
(---
 } any data
 +++)
```

for example

```
f. CONTENTS:=
(---
RECORD ONE
RECORD TWO
RECORD THREE
+++);
```

the data is then loaded into the files container. CONTENTS may only be read via the standard procedure LIST.

INT VAR COPIES

specifies the number of backup security copies to be made and maintained by the system.

STRING CONST CREATIONDATE

is set by the system and indicates the calendar date when the file was created.

INT VAR DEVICE

specifies the type of device which should be used for the file. The system may decide to store it on a different device.

Constants:- CONSOLE, CARDREADER, MICR, OCR, OMR, TAPEREADER, CARDPUNCH, LINEPRINTER, PLOTTER, TAPEPUNCH, DRUM DISK, MAGTAPE, DISKPACK, TERMINAL.

STRING VAR ENVIRONMENT

Indicates, for a code file, the environment to be used when it is executed.

BOOL VAR DIRECTION

indicates in which direction the user wishes to read the file. If BACKWARDS is required the system should ensure that the file is stored on an appropriate medium.

Constants:- FORWARD, BACKWARD.

BOOL CONST EOF

is set by the system and indicates if the last file access reached the logical end of the file.

INT CONST ERRORTYPE

indicates the nature of any error which may have occured during the last file access.

BOOL VAR EXISTENCE

specifies whether the file is to be retained in the users directory, or removed, at the end of the current program block.

Constants:- PERMANENT, TEMPORARY.

INT VAR EXTCHARSIZE

specifies the number of bits forming a character on the physical file medium. This may only be different from INTCHARSIZE if the system provides translation facilities.

STRING VAR FORMMESSAGE

specifies a message to be sent to the operators console indicating the type of stationery to be used when printing the file.

INT VAR GENERATION

serves to identify uniquely different versions of files with the same site, owner and title.

INT VAR INFOTYPE

specifies the type of data held in the file. May only be set to CODE by a compiler.

Constants:- DIRECTORY, DATA, NCL, ALGOL, ALGOLW, ...
XFORTRAN, CODE.

INT VAR INTCHARSIZE

specifies the number of bits forming a character in the logical file.

INT VAR LABELLING

specifies the type of labelling to be given to the file.

Constants:- NONE, STANDARD, USER.

INT CONST LASTREADLENGTH

is set by the system and gives the length, in UNITTYPE units, of the last logical record read/written from/to the file.

INT CONST LASTRECORDNUM

is set by the system and gives the logical number of the last record in the file.

STRING CONST LASTUSEDATE

is set by the system and specifies the last date on which the file was opened.

INT VAR LIFETIME

specifies for a user how many days after creation the file will not be archived or removed by the system.

INT VAR MAXRECSIZE

specifies for new files the maximum length, in UNITTYPE units, of a record of the file.

INT CONST MODE

is set by the system and specifies the mode of the object returned by the file if it is a code file for an NCL procedure.

BOOL VAR NEW

if this attribute is TRUE opening the file for the first time will cause a new container to be created and the attribute set to FALSE.

BOOL VAR OPEN

specifies if the logical file description is attached to the physical file described.

INT VAR ORGANISATION

specifies the logical organisation of the records in the file.

Constants:- NONE, RANDOM, SERIAL, ISEQ, SEQ.

INT VAR KEYLENGTH, KEYPOS

specify, if the ORGANISATION is ISEQ, the length of the key in each record and the position of the key within a record. Both are specified in UNITTYPE units.

STRING VAR OWNER

specifies the user code of the owner of the file.

INT VAR PAGESIZE

specifies the size, in lines of the page for a page oriented file..

STRING VAR PASSWORD

may be set only by the owner of the file and only when the file is open. To open a file with a non-null password requires that the attribute matches the value held by the system.

STRING VAR PERIPHERAL

indicates the peripheral to be used when doing I/O operations on the file.

INT VAR RECFORMAT

specifies the format of the file's records

Constants:- FIXED, VARIABLE, UNDEFINED.

INT CONST RECORDNUM

is set by the system and indicates the logical record number of the last accessed record.

BOOL CONST RESIDENT

is set by the system and indicates if the file is physically accessible by the user without operator intervention. For example if RESIDENT is false the file may be held on an unloaded magnetic tape.

STRING VAR SITE

specifies the site where the file is located.

STRING VAR TITLE

specifies the name by which the file is identified by the OS

INT CONST TOTALBLOCKNUM

is set by the system and specifies the number of blocks in the file.

INT CONST TOTALRECNUM

is set by the system and specifies the number of records in the file.

INT VAR UNITTYPE

specifies the type of units in which attributes such as BLOCKSIZE are measured.

Constants:- BIT, CHAR, WORD.

E2 PERIPHERAL ATTRIBUTES

STRING VAR ACCESS

see the file attribute ACCESS

INT VAR CHARSET

indicates the number of different characters recognised by the device.

INT VAR CHARSIZE

indicates the number of bits in a character recognised by the device.

INT VAR DENSITY

specifies the number of bits which may be recorded per millimeter (for storage peripherals only).

INT VAR DEVICE

specifies the type of peripheral

Constants:- see file attribute DEVICE

INT VAR ERRORLIMIT

specifies the number of attempts which may be made to read/write from/to the device before an error condition is signalled.

BOOL VAR EXISTENCE

see file attribute EXISTENCE

INT VAR HEIGHT, LENGTH

specify the maximum height/length in millimeters of an I/O frame for graphic devices.

STRING VAR OWNER

see file attribute OWNER

INT VAR RESTART

specifies the part of a file to be repeated in case of a breakdown occuring in the middle of transmission

Constants:- LINE, RECORD, PAGE, ALL

STRING VAR SITE

see file attribute SITE

INT VAR SPEED

specifies the transmission rate for UNITTYPE units.

INT CONST STATE

indicates the current state of the device

Constants:- DEAD, FULL, OFF, WAITING, READING, WRITING, REWINDING.

STRING VAR TITLE

see file attribute TITLE

STRING VAR TRANSLATIONTABLE

specifies the translation to use when encoding or decoding characters to/from the device. The string is split into two halves containing the uncoded and encoded characters seperated by a *

e.g. "ABCD*XYZW"

INT VAR UNITTYPE

specifies the basic unit of data transfer to/from the device.

Constants:- BIT, CHAR, WORD.

BOOL VAR VIRTUAL

if TRUE indicates that the device may be simulated by the spooling system of the host OS.

E3 ENVIRONMENT ATTRIBUTES

STRING VAR ACCESS

see file attribute ACCESS

INT CONST COST

is set by the system and indicates the current cost of executing the process.

INT VAR COSTLIMIT

specifies the maximum allowable value for COST

INT CONST CPUTIME

is set by the system and indicates the current time, in seconds, used by the CPU in executing the process.

INT VAR CPULIMIT

specifies the maximum allowable value for CPUTIME.

INT CONST ELAPSEDTIME

is set by the system and indicates the current time, in seconds, since the process was activated.

INT VAR ELAPSEDLIMIT

specifies the maximum allowable value for ELAPSEDTIME.

BOOL VAR EXISTENCE

see file attribute EXISTENCE

STRING VAR OWNER

see file attribute OWNER

INT CONST READS

is set by the system and indicates the number of reads made by the process.

INT VAR READLIMIT

indicates the maximum allowable value for READS

STRING VAR SITE

see file attribute SITE

STRING VAR TITLE

see file attribute TITLE

INT CONST WRITES

is set by the system and indicates the number of writes made by the process.

INT VAR WRITELIMIT

indicates the maximum allowable value for WRITES.

APPENDIX D

NCL STANDARD SYSTEM PROCEDURES

The standard system-procedures are described in the following appendix. The list is not complete; future developments or suggestions from other sources may result in other procedures being added. Nevertheless the author has endeavoured to include procedures which can cater for most user needs.

Resource Handling Procedures

COPY	Copies the given resource objects from one location to another
DEFINE	Enables special attributes of resource objects to be defined, e.g. tracks per cycle for a disk file.
DELETE	Removes resource objects from the users environment. The user must have the appropriate access permission.
EDIT	Enables interactive editing of files
IN	Creates a file with a given name and reads cards into it.
LFILES	Outputs the titles of the resource objects belonging to the user.
LIST	Outputs the contents of the given file.
LPROPS	Outputs the attributes of a given resource object.
SAVE	Makes the given resource object permanent.

Session and Process Oriented Procedures

GOODBYE	Used to terminate a session
HELLO	Used to start a session for a given user with a given account.
<language>	For every language supported by the system there will be a procedure of the same name which will run the compiler with a given file, returning the code file produced.
LOG	Enables the user to alter the type and detail of information printed on the session journal.
OUTPUT	Outputs the value of a given variable
SEND	Enables messages to be displayed on the named device.
STATUS	Enables the user to enquire about the states of executing processes.
SUBMIT	causes the given file of NCL source to be started as a batch session running asynchronously and independently of the main session.

User and Account Oriented Procedures

ALTER-ACCOUNT	Enables changes to be made to a given account
DELETE-USER	Removes a given user code from the system list. Only available to operators.
MAKE-ACCOUNT	Used to create a new account.
NEW-USER	Creates a new user code on the system. Only available to operators.
PASSWORD	Enables the user to change his password.

Built in Procedures

COUNT	returns the number of elements in a given data structure.
LENGTH	returns the number of characters in a string.
TIME	returns the time in different formats depending upon the parameter.
REMOVE	returns the value of the given data structure element and removes the element from the data structure.
GET	used to procure a semaphore. Equivalent to PE as defined in [Agerwala 77]
FREE	used to release a semaphore. Equivalent to VE as defined in [Agerwala 77]