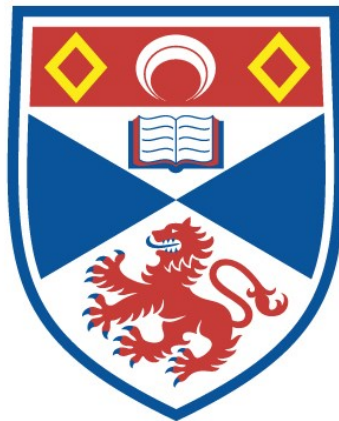


ABSTRACT MACHINE DESIGN FOR INCREASINGLY MORE POWERFUL ALGOL-LANGUAGES

Hamish Iain Elston Gunn

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1985

Full metadata for this item is available in
St Andrews Research Repository
at:

<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:

<http://hdl.handle.net/10023/13461>

This item is protected by original copyright

Th
A 239

23

ProQuest Number: 10167227

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10167227

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

**ABSTRACT MACHINE DESIGN FOR INCREASINGLY MORE POWERFUL
ALGOL-LIKE LANGUAGES**

by

Hamish Iain Elston Gunn

A thesis submitted for the degree of Doctor of Philosophy

Department of Computational Science

University of St. Andrews

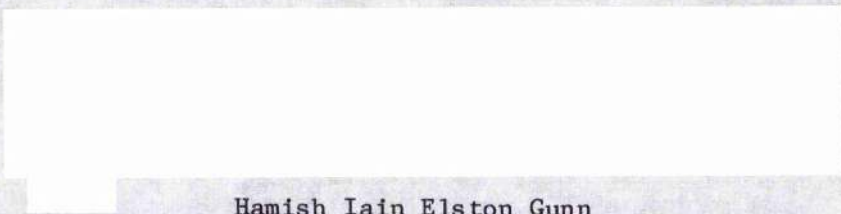
St. Andrews

September 1983




Declarations

I declare that this thesis has been composed by myself and that the work that it describes has been done by myself. This work has not been submitted in any previous application for a higher degree. The research has been performed since my admission as a research student under Ordinance General Nr. 12 on 1st October 1976 for the degree of Doctor of Philosophy.



Hamish Iain Elston Gunn

I hereby declare that the conditions of the Ordinance and Regulations for the degree of Doctor of Philosophy (Ph.D.) at the University of St. Andrews have been fulfilled by the candidate, Hamish Iain Elston Gunn.



Dr. R. Morrison

Abstract

This thesis presents the work and results of an investigation into language implementation. Some work on language design has also been undertaken. Three languages have been implemented which may be described as members of the Algol family with features and constructs typical of that family. These include block structure, nested routines, variables, and dynamic allocation of data structures such as vectors and user-defined structures.

The underlying technique behind these implementations has been that of abstract machine modelling. For each language an abstract intermediate code has been designed. Unlike other such codes we have raised the level of abstraction so that the code lies closer to the language than that of the real machine on which the language may be implemented. Each successive language is more powerful than the previous by the addition of constructs which were felt to be useful. These were routines as assignable values, dynamically initialised constant locations, types as assignable values and lists.

The three languages were,

Algol R

a "typical" Algol based on Algol W

h an Algol with routines as assignable values, enumerated types, restriction of pointers to sets of user-defined structures, and constant locations.

ns1 a polymorphic Algol with types as assignable values, routines as assignable values, lists, and type- and value-constant locations.

The intermediate code for Algol R was based on an existing abstract machine. The code level was raised and designed so that it should be used

as the input to a code generator. Such a code generator was written improving a technique called simulated evaluation. The language h was designed and a recursive descent compiler written for it which produced an intermediate code similar in level to the previous one. Again a simulated evaluation code generator was written, this time generating code for an interpreted abstract machine which implemented routines as assignable and storable values. Finally the language nsl was designed. The compiler for it produced code for an orthogonal, very high level tagged architecture abstract machine which was implemented by interpretation. This machine implemented polymorphism, assignable routine values and type- and value-constancy. Descriptions of the intermediate codes/abstract machines are given in appendices.

Contents

Chapter 1 : Abstract Machines and Programming Languages.	1
Chapter 2 : Implementation of a typical Algol.	26
Chapter 3 : Intermediate Code and Code Generator for Algol R.	42
Chapter 4 : The Programming Language h.	63
Chapter 5 : The Implementation of h.	85
Chapter 6 : The Polymorphic Programming Language ns1.	101
Chapter 7 : The Tagged Architecture Machine Implementation of ns1.	117
Chapter 8 : Summary, Conclusions and Further Work.	140
Appendix A : Algol R Intermediate Code.	154
Appendix B : h Intermediate Code.	168
Appendix C : Other ns1 Abstract Machine Instructions.	188
References :	203

CHAPTER 1

Abstract Machines and Programming Languages.

This thesis presents the results of an investigation into the implementation and design of general purpose programming languages (we use the abbreviation GPPL). The object of the work was twofold. Firstly it was to investigate a way of improving the implementation + of these languages. This was to be achieved by using the technique of abstract machine modelling. [Newe72] Secondly, the aim was to add to the power of these languages, still keeping them straightforward. Thus the implementation of additional features was investigated. In this introduction we first consider what is meant by an "abstract machine" and how they have been used in relation to programming language implementation. We then define the kind of language to which we restrict ourselves. Finally we attempt to categorise additions to such a language in order to make it more powerful. Subsequent chapters present the work and results in chronological order.

Abstract Machines.

A high level language should be designed to formulate algorithms within a specific problem area. Given such an algorithm expressed in the language, it is executed on a computer in order to give the desired solution. Ideally this computer should obey directly the commands of the

+ Before any program may be submitted for execution, it must be checked for syntactic and type correctness. These are problems which were foremost in computer science research but which were not machine related and now should occupy less of our time since earlier work has gone a long way to solving them. Thus we do not propose to dwell upon this aspect of language implementation.

language. For example, a program written in Algol 60 [Naur63] should be executed by an ideal computer whose machine language is Algol 60. We require the existence of a real machine which is the ideal machine. An attempt at such a situation is the SYMBOL [Smit71] system. A research goal was to design a computing system with substantially improved performance over conventional systems, along with a reduction in the cost of computing, by directly implementing a high level language and virtual storage, time-sharing operating system in hardware. Only one system was constructed. The SYMBOL programming language (SPL) is similar in nature to FORTRAN but typeless. The SYMBOL system translates (by hardware) SPL into a one to one reverse polish representation. The SYMBOL system is not truly a case of making the ideal machine the real machine since SPL is translated into a directly executable (high level) code.

This is a difficult problem to solve, yet the language implementor must give the programmer the impression that such ideal machines exist. With our present-day machines a program must ultimately be executed by a real machine in an equivalent form in that real machine's code. The problem before the language implementor is to perform the mapping between an ideal machine and a real machine, that is, the production of real machine programs from ideal machine programs. The closer the real machine is to the ideal machine the easier will be the implementation. In fact this principle should be a guiding light for those architects who design our real machines. The Burroughs B6500 [Cree69] was designed jointly by a hardware and software team, but unfortunately this trend seems not to have continued. A new machine should be designed to make the mapping above easy and efficient for all ideal machines supported by it. One way of easing this mapping as we have seen is to have a machine which is close to the ideal machine. In fact, this machine need not exist except on paper or as a program. Such a machine is called an abstract machine. The use of abstract machines is not only restricted to programming language

implementation but may be extended to other problems.

We may depict the situation thus,

ideal \rightarrow abstract \rightarrow real

meaning either

- a) A program in the ideal (high level) language is converted to a program for some abstract machine which in turn is converted into a program on a real machine
or
- b) The ideal machine is realised in terms of an abstract machine which is realised in terms of a real one.

The technique of abstract machine modelling for the implementation of software stems from the idea that the problem can be regarded as one of hardware design. We try to specify a machine which is ideally suited to the task being implemented. Newey [Newe72] puts it thus -

"Abstract machine modelling is based on the concept that the fundamental operations and data types required to solve a particular problem define a special purpose computer which is ideally suited to that problem. In order to obtain a running version the abstract model is realised on an existing computer by implementing its basic operations and data types."

Poole [Pool74] says

"The architecture of the abstract machine forms an environment in which the modes and operations interact to model the language. Unlike a real machine whose architecture is governed by economic considerations and technical limitations, the abstract machine has a structure which facilitates the operations required by a given programming language. The designer of the abstract machine must plan the architecture so that it can be efficiently implemented on

a real machine."

In particular we wish to examine the use of abstract machines in the implementation of high level programming languages. Like any other problem, when solving it in a single step proves difficult, a common technique is to break it down into a series of steps.

"The designer must balance the convenience and utility of these operations against the increased difficulty of implementing an abstract machine with a rich and varied instruction set." [Newe72]

With our single mapping above this involves breaking it into a sequence of mappings from the ideal machine to the real machine.

ideal \rightarrow abstract 1 \rightarrow ... \rightarrow abstract n \rightarrow real

Poole [Newe72] has called this a hierarchy of abstract machines.

"Instead of realising the initial abstract machine A1 directly on a real computer, design a second abstract machine A2. The operations of A1 are then defined in terms of A2 operations. Such a definition is independent of the realisation of A2 itself and hence A1 may be realised by realising A2. The hierarchy can be carried to any depth by defining A2 in terms of A3 etc. The base machine of the hierarchy Ak is then realised on a real computer."

An abstract machine then is like an ideal machine in that it is not realised in hardware, but is at a level closer to the real one. The levels of abstraction should naturally be chosen so that each mapping is straightforward. In terms of ease of implementation it may prove that such a technique involves less work than the single mapping. However the very number of mappings involved may contribute to confusion and/or inefficiency in the overall process.

Mappings.

A mapping from (source) machine **S** to (target) machine **T** may be considered to be the process of arranging that, given a program in **S** machine code, an equivalent program in **T** machine code is produced. "Equivalent" means that if machines **S** and **T** were realised in hardware both programs run on their respective machines would produce the same result. If one terminates, the other should with the same result. The mapping will have a "quality" associated with it related to the size and speed of the **S** and **T** programs. Size is the factor more readily measurable and is more dominant due to this tangibility.

We consider four common kinds of mapping between machines **S** and **T** where **S** may be an ideal or an abstract machine and **T** may be an abstract or a real machine.

1. Code Generation (Translation).

In this method a program in **S** machine code is directly translated into an equivalent program in **T** machine code. The translation takes place once for each **S** program which then takes no part in any subsequent mappings.

i.e. Scode program --> equivalent Tcode program

The ease with which this translation mapping takes place is closely related to the similarity between machines **S** and **T**. Another factor involved is the quality of the mapping - if the machines are grossly dissimilar then **T** programs produced by the mapping may be very inefficient with regard to the **T** machine. Thus with this kind of mapping, the implementor requires a fair degree of skill to minimise this possibility or else employ a translation technique which aids the production of good Tcode. The mapping can be considered to take place at translation time.

2. Interpretation (Simulation).

In this method, a program in Scode is fed as data to a program in Tcode. This Tcode interpreter simulates the execution of the S machine. This program is the same for every S program thus it need only be written once for a particular machine. This would appear to be a point in favour of interpretation. In addition, since the S machine is well defined no great skill is required to write it, although the programmer should be skilled in writing T programs to make an efficient interpreter.

i.e. Scode program \rightarrow Scode program + Tcode program

The simulation process involves the interpreter fetching, decoding and carrying out S instructions from the S program in the Tcode software model of the S machine. The fetching and decoding can add significantly to the time required to interpret an S program, especially if the S machine instruction format is complex, or the T machine instructions are not suited to the decoding process. The mapping can be considered to take place at execution time.

Klint [Klin81] classifies four kinds of interpretation.

Type 1

This is direct interpretation of the source text. As examples he cites macro processors and some BASIC implementations. The latter may be inefficient and have the undesirable property that syntax errors can only be detected at run time.

Type 2

This is interpretation of a high level intermediate code. He means that there is a one-one mapping between the source and the intermediate language and cites LISP [McCa62] as only being lexically preprocessed. In this case syntax errors are detected before execution. Since the intermediate language depends on the source language and not on the underlying machine Klint states,

"maximal flexibility is possible at run time and it is easy to provide diagnostics and debugging facilities at the source language level."

The SYMBOL system would be Type 2.

Type 3

This is interpretation of a low level intermediate code close to machine level, there being a one-many mapping from the source to the intermediate code. Again syntax errors are detected before execution. Flexibility is lost at run-time but these systems are more efficient than the previous types. Portability is improved since only a simple interpreter needs to be written for each machine.

Type 4

This is just direct execution of a low level code by a real machine. The interpreter is a microprogrammed instruction set. This is the most efficient of all but perhaps the most difficult for which to generate code.

3. Macro Expansion.

The mappings are related insofar as that, given a simulation mapping, a special kind of translation mapping can be produced. Each **S** instruction is treated as a macro and is replaced by the **Tcode** which simulates that instruction in the simulation mapping. Thus the new mapping produced is a translation mapping performed by macro expansion. This combines some of the advantages of both kinds of mapping namely that there is no longer the overhead of fetching and decoding of **S** instructions. The disadvantage is the increase in size of the produced **T** program compared with an equivalent one produced by better translation techniques. The macro expansion can be performed either by the compiler during the production of **Scode** or by a macrogenerator as a later phase.

4. Imbedding.

Waite [Wait73] describes one of the early and simple abstract machine modelling techniques known as imbedding. An existing host procedural language is used to support the operations of the abstract machine. He observes that in many abstract machines the flow of control constructs are usually independent of the data. Thus they are essentially invariant from one abstract machine to another. Imbedding involves the use of procedures and functions written in the host language which implement the procedural operations and data operations of the abstract machine. SLIP [Russ65] is an example of list processing features imbedded in FORTRAN.

Compilation.

In considering the use of abstract machine modelling in language implementation we must consider how it fits in with the goals of compiler design. Horning [Horn74] describes these as

- correctness (more realistically reliability)
- efficiency of runtime space and time
- efficiency of the compiler development process
- efficiency of program development using the compiler (including the efficiency of compilation)
- efficiency of target programs produced by the compiler

Hunter [Hunt81] also adds

- keeping the compiler as small as possible

Perhaps another goal would also be

- to allow the easy implementation of the language on another architecture.

Poole [Pool74] describes how most of these aims, especially the last, can be achieved by considering the compiler to be made up of two parts

- a) a language dependant translator (LDT) which depends on the characteristics of the source. This typically performs lexical analysis, syntax analysis and abstract machine code generation.

and

- b) a machine dependant translator (MDT) which depends on the target machine.

The interface between these two phases of compilation is an abstract machine. The LDT is written entirely in terms of the abstract machine. Information flow is from the LDT to the MDT although some may flow in the reverse direction. This natural and simple organisation helps the writing of the compiler. The mapping from ideal to abstract machine (or from source to abstract machine language) takes place in the LDT. An interface procedure for an abstract machine instruction is called when that instruction is needed. What these interface procedures do depends on how the mapping between the abstract machine and the real machine is to be achieved. For example, the interface procedure could simply write out an encoding of the instruction or indeed perform the abstract to real mapping by generating a sequence of real machine instructions. In the latter case the MDT would probably just be the assembler for the real machine. This organisation allows a choice of mappings - to change the mapping only the interface procedures need to be rewritten. Now, should the language be ported to another machine, all that is necessary is to rewrite the MDT to produce code for that machine.

A consideration of the four techniques.

We now consider these mappings in relation to each other in the context of implementing a GPPL. The technique of imbedding is generally

applied [Wait73] when the host is itself a high level language. For example we could imbed list-processing procedures in FORTRAN and translate a list-processing language into FORTRAN. This then necessitates a translation of the FORTRAN into real machine code. Alternatively we could write them in assembly language but this would then be almost a form of interpretation. This does not appear to be a good way of implementing a GPPL especially when compared with the other techniques. Therefore we will not consider it further.

The use of macros to implement abstract machines has been well covered [Wilk64,Brow69,Wait70,Newe72] with reasonable results. The languages implemented during the work for this thesis did not use macro mapping. It was considered but rejected in favour of examining either a translation or simulation mapping. Macro mapping seems principally a portability technique. This leaves us with perhaps the two most popular methods of implementing languages. Efficiency appears to be a prime reason for choosing a translation mapping. Interpretation is chosen when an easier implementation is needed or where portability is a major concern. Code generation according to Hunter [Hunt81] tends to produce larger, more complex, slower compilers. The two techniques may be combined in a compromise [Dak173,Daws73] where the most frequently executed program sections can be code generated and the rest interpreted.

One of the aims of this work is to use abstract machine modelling with compilers split into LDT and MDT parts. This should allow the implementor a choice between which mapping he feels is appropriate. In order to do this however we have found that particular attention must be paid to the level of the abstract machine whose code is generated by the LDT. Thus we now look at a number of intermediate languages/abstract machines and then consider intermediate language design. Klint [Klin81] feels that we should be considering type 2 interpretation, that is, a high level intermediate code. His reasons are based on the growing need for portability amongst a

wide range of (micro)processors but more relevantly to this thesis is

"the current trend towards 'very' high level languages. In such languages the primitive operations are so complex and time consuming it is irrelevant (with regard to execution time) whether they are compiled or interpreted."

Abstract Machine Examples.

We now look at fairly typical examples of abstract machines, in an attempt to see where they lie in relation to the ideal machine.

UNCOL/SLANG UNCOL [Stee61] and SLANG [Sib161] are families of abstract machines. UNCOL was proposed to solve the problem of running n languages on m machines. This would require $n * m$ translators but this could be reduced to $n + m$ if all n translators produced UNCOL code which in turn was mapped onto each of the m machines. UNCOL was never properly implemented but Steel [Stee61..] proposed how it might be. The UNCOL abstract machine was fairly low level, that is it was closer to real machines than ideal machines. It had an accumulator but no stack. There were twenty instructions consisting of an operator and an operand. The meaning of these instructions depended on the mode of the operand. Interestingly, operands were specified by a location only, there was a separate data description for them. Additional flow information was passed from the translator in order to allow optimisation. The project was abandoned but Coleman [Cole74] doubts that this was because the basic idea was unsound, just that at the time "compilation techniques were not clearly understood, and that adaptable translators were difficult to write."

SLANG arose from a project to develop a compiler-writing language. It was similar in approach to UNCOL but the translators knew something about the target machine. This was not strictly an UNCOL because, given different machine descriptions the translator would produce different

instruction sequences. Again this was low level. However it does illustrate an alternate approach to only allowing information flow from the LDT to the MDT. There is also information about the machine flowing back to the LDT. As will be seen we do not use this approach because we try to write the LDT in complete ignorance of the machines the source language will compile to, and what mapping will be used.

Janus Janus [Cole74] has been called a standard abstract machine. That is one which has been designed around a model which may be used for many similar programming languages in the UNCOL manner. Janus is implemented by macro expansion to assembly language. The authors came to the conclusion based on previous work [Newe72] that

"major problems are associated with the design of a suitable abstract machine model (and its programming language) for a given application. The task is not easy even for experienced programmers."

Janus was designed by examining existing programming languages and real machines. The intention was to find what was common to them all. The Janus machine was made up of an accumulator, an index register and a stack. The operations of the machine are all based on features of real machines which support high level languages. Also included are "pseudos" which are used to reserve data space. Thus again this is a low level abstract machine. Coleman gives the following example.

F(1,A,B+C,D[I+3]) a procedure call

```
CALL REAL PROC F()
ARGIS INT CONST C1() A 1
ARGIS, I REAL LOCAL A()
LOAD REAL LOCAL B()
ADD REAL LOCAL C()
STARG REAL TEMP T1()
LDX INT LOCAL I()
MPX INT CONST C2() E REAL
LOAD, I REAL LOCAL D(3* REAL)+
STARG ADDR ARG L1(3* ADDR)
RJMP REAL PROC F()
SPACE ADDR ARG L1(4)+
SPACE ADDR ARG (1) A C1
SPACE ADDR ARG (1) A A
SPACE ADDR ARG (1) A T1
SPACE ADDR ARG (1)
CEND REAL PROC F()
```

The code as may be seen is fairly complex. We believe that this is the case because of the attempt to be all things to all machines and languages.

O-code This is an intermediate code for a specific language and does not suffer from the complexity of Janus. O-code [Rich71] is the intermediate code for BCPL. [Rich69] This systems language has been ported to a wide range of machines. It is similar to Algol 60 [Naur63] but has only one size of data item. All items are bit patterns which may be interpreted as addresses, integers, characters or truth values by appropriate operations. The compiler is split into a LDT and MDT. Richards suggests that the interface between them should not be a macro language - "a compiler so produced will be very slow being limited by the inefficiency of the macro generator". He recommends the MDT be written by hand - "the implementor may find it easier to generate optimised code since he can optimize by algorithm rather than by a complicated set of macro definitions". O-code is designed to be compact, easy to generate by the LDT and easy to translate by the MDT which could be a simple non-optimising code generator.

The design aims of O-code hold many attractions. However it again is at a somewhat low level. The O-code machine consists of a linear memory of words and two address registers to support a stack, S addressing the top

stack element and P addressing the base of the current stack frame. Locations are addressed either absolutely or relative to P. The LDT in addition to generating abstract machine code also passes space information to the MDT. For example it is sometimes necessary for the LDT to indicate where the top of stack is relative to the current frame base. This happens when vectors are declared or at the ends of blocks. A directive is also provided which indicates where the dividing point occurs between declarations and the body of a block. The MDT then generates code to ensure that all stack items are held in their appropriate store locations as opposed to being in registers for example. Richards states "Without such a directive it would be difficult for an optimising code generator to know when stacked items could be held safely in machine registers." Another example is a directive which on entry to a routine indicates the initial stack frame size.

Richards is primarily concerned with portability but also shows how a MDT can be written to produce good code.

"O-code relies on its simplicity to be effective as a bootstrapping tool since otherwise the initial code generator for a new machine would be too difficult to write. But it is important to note that it may be compiled into efficient code using an optimising code generator."

Richards gives the following example of O-code

```
STACK 5
LSTR 11 70 40 37 78 41
      32 61 32 37 78 10
LP 2
STACK 9
LP 2 LL L3 FNAP 7
LG 76 RTAP 3
```

O-code then is low level with a heavy reliance on directives to the MDT but suitable for a translation mapping to a real or simulated machine.

P-code P-code [Nori74] is the intermediate code produced by the Pascal [Jens74] compiler. The authors suggest three implementation strategies using P-code. If the expected use was for teaching purposes or small programs only then they recommend that the simplest method is by writing an efficient assembler/interpreter. For bootstrapping the compiler then P-code should be macro expanded then the code generation routines of the compiler should be rewritten for the target machine. Finally they suggest that if storage space is the main constraint "a judicious mixture between interpretation and machine execution can be used", recommending the technique of threaded code. [Bel173] P-code is similar to O-code except that it has a larger instruction set to handle a larger set of types. An example of P-code follows.

```

procedure print_tree( head : ref );
begin
  if head <> nil then
    begin
      print_tree( head@.left )
      write( head@.val:12 )
      write( eol )
      print_tree( head@.right )
    end;
  end;
end;

```

```

ENT 4
LOD 0 4
LDCN
NEQA
FJP 94
MST 1
LOD 0 4
IND 2
CUP 1 74
LOD 0 74
IND 0
LDCI 12
CSP WRI
LDCI 0
LDCI 1
CSP WRC
MST 1
LOD 0 4
IND 1
CUP 1 74
RETP

```

P-code is medium level but designed for simulation or macro mapping. It is

not suitable for code generation. A similar code, S-code [Bail80] has been used primarily for interpretation and also for code generation. This latter mapping is by a form of macro expansion and as such suffers from increased size and little, if any, optimisation. It is faster than interpretation due to the removal of the decode loop.

I-code I-code [Robe81] is the intermediate code generated for the IMP77 [Robe77] programming language. Its designer, Robertson noted that the emphasis on abstract machine design was on enabling compilers to be quickly bootstrapped onto a new machine either by interpretation or by macro expansion. He felt that this did not allow the production of highly optimised code.

"Apparently considerations of portability and machine independence have caused problems of optimisation to be overlooked."

He was concerned with the level of abstract machines feeling that they were too low level and attempted to put machine independence and optimisation on an equal footing, regarding the intermediate code as directives to a code generator.

"Instead of the intermediate code describing the computation to be performed, it describes the operation of a code-generator which will produce a program to perform the required computation.

In essence I-code attempts to describe the results required in a way which does not constrain the method of achieving those results."

I-code is completely machine independent. Robertson's work is related to the early part of our work however his emphasis was on optimisation whereas ours was on high level abstract machine code design. The code generation method is similar to another technique described in a later chapter which

serves as a basis for our work. Thus we will limit this discussion to a brief summary.

Robertson concludes that using a high level intermediate code is a viable technique, having been used for several IMP77 compilers. Space-wise they compare favourably with other compilers but seem to be a little slow in execution time. As far as portability is concerned Robertson's method takes longer to write compilers than does O-code or P-code but once one compiler has been written it may be used as a template - "a new optimising compiler can be written in the space of a few months".

I-code is regarded by Robertson as a sequence of instructions to a stack oriented machine which produces programs for specific computers. It describes the compilation process necessary to generate an executable form of a program, not the computation defined by the program. We feel however that this is simply a way of stating that I-code is an intermediate code designed for code generation. It is not clear from his thesis what he feels the difference is between "describing the compilation process" and "the computation defined". His I-code must define the computation implied by the source.

Source language objects are defined by descriptors in a block structured manner. The stack holds copies of descriptors for the computation. The only example of generated I-code he gives is,

```

integer X
X = A(J)
X = 0 if X < 0

```

```

DEF 12 "X" INTEGER SIMPLE
  DEFAULT NONE NONE
PUSH 12
PUSH 6
PUSH 7
ACCESS
ASSVAL
PUSH 12
PUSHI 0
COMP >= 1
BGE 1
PUSH 12
PUSHI 0
ASSVAL
LOC 1

```

It is medium level and reverse polish in nature. However, more importantly, we now see a change in viewpoint of an abstract machine/intermediate language - describing a code generation process rather than something to be regarded as potentially a real machine.

Abstract Machine Intermediate Code Design.

From the above we note that historically abstract machines have tended to be at a rather low level although Robertson has raised the level somewhat. The emphasis behind abstract machine design has been portability and with Robertson's work, optimisation. It was our intention in this work to consider the implementation of very high GPPLs from the point of view of abstract machine design. Portability and efficiency certainly must be considered but also one objective was to develop a method which allowed a choice of mapping between code generation and interpretation. As will be seen, the compiler is broken into the LDT and MDT parts and we now consider the abstract machine level which forms the interface.

Hunter [Hunt81] suggests that the UNCOL/Janus type approach should be restricted.

"It would appear that, to accommodate a wide range of languages the target language has to be at too low a level to be implemented

efficiently on all machines. It seems better to design a target language for translating a particular high level language into or for implementing on a particular machine."

That is, all LDTs for languages running on a specific make of machine, should all produce the same abstract machine code. Then only one MDT need be written for that machine. The alternate approach is that one LDT be written for a particular language producing an intermediate code and that a MDT be written for every machine on which the language is to run. This is the approach taken by O-code or P-code. Such approaches however have relied on some information about the target machine being passed back to the LDT so it is not truly independent. Brown [Brow72] discusses the relative merits of using high level intermediate languages. His work is primarily concerned with macrogenerators, but using abstract machine modelling to achieve portability. He describes some attempts at implementing a high level intermediate language. One way was to macro expand it into PL/I "but the resulting implementation ... was so large and slow that it was totally unusable". A program in a high level language is more machine independent than one at a lower level - "Looking down from on high it is possible to see over a wider area than when one is close to the ground". But he warns that the differences in machines are not so great. After comparing a high and low level intermediate language which implement a macrogenerator Brown concludes that the advantages of a high level intermediate language are not very great in practice. Low level codes allow a quicker implementation.

A paper published after our work on high level intermediate code was completed had a similar intent. Kornerup [Korn80] investigated the possibility of supporting high level languages with intermediate languages which could be used both for direct interpretation and code generation.

"it would be advantageous to be able to use such intermediate

languages directly to give efficient code generation on minicomputers. This requires that the intermediate language (the hypothetical machine) is being designed not only for interpretation but also such that sufficient information for code generation is available."

Kornerup first designed a new form of P-code for Pascal. A LDT produced this and it was implemented by a microprogrammed interpreter. A MDT in the form of a code generator was also written.

The abstract machine was at a similar level to the original P-code although it differed in its architecture. For example it had three stacks, one for procedure calls, one for addresses and one for real values. Essentially it was reverse polish in nature but

"it does not contain special instructions reflecting control structures like repetition or conditionals, nor does it have instructions for accessing components of structured data."

By this they mean that such constructs are implemented by sequences of low level instructions such as jumps (as is I-code, S-code, O-code). The results of this work were that P-code was successful as a microprogrammed implementation however it was less useable. P-code apparently had some deficiencies making some generated code sequences "absurdly complicated" because some information was lost in its generation.

Their conclusions concur with ours and we summarise them here. A major problem for a code generator is in recognising the structure of the source language.

"For instance to make efficient use of the registers, one must know whether a label in the intermediate form is part of an if-statement or can be branched to by a goto-statement."

They also propose that no storage allocation should be performed by the compiler. The intermediate form should contain all the declarative information of the source program. They conclude that their attempt at a form suitable for both code generation and interpretation fails but say that if we relax the requirement that the intermediate form has to be immediately interpretable a solution is possible. This relaxation in their view is reasonable because "some processing before execution of such a form will always be necessary (e.g. assembly, linking and possibly loading)". Thus the intermediate form has to be transformed by some kind of code generation before execution. The following criteria must then be satisfied. No binding should discard information useful in generating efficient code for classical machine architecture of a wide variety. The intermediate language should be language dependant but machine independant. Storage allocation should be left to the MDT. All declarative information on the data has to be accessible and all referencing of data must have the form of a reference to the appropriate declarative information.

The subsequent chapters describe three abstract machines/intermediate codes. Further discussion on this topic is left until then.

Properties of languages under consideration.

In general we may say that the Algol family of languages satisfies broadly the criteria we use to define "a general purpose programming language". There is a wide variety of languages currently in use and we must restrict our investigations to a realistic subset. We believe the Algols to have "lasted the course", evolving through the 1960's and 1970's. They appear to be the starting point for many new languages, and if that is not the case, at least they have much in common with them (e.g. ADA, Modula, S-Algol, IMP-77, Euclid).

The Algol family.

A number of headings are introduced here under which Algol-like programming languages will be classified. These headings are chosen for their usefulness in the context of this thesis. The choice of headings is explained and those features considered to be necessary in a GPPL are outlined under those headings.

Primitive data types.

The programmer manipulates objects in some universe of discourse. We may separate what he manipulates from how he does so although the two are closely related. To an extent the choice of primitive data type defines the application area of the language. These data types are primitive in that they are simple and contain no structure. In a GPPL we would expect some representation of numbers, truthvalues, and text. The language would need operations on these.

Data structures.

The primitive objects alone are insufficient to support representations of object collections. We need data structures to allow the modelling of collections. Two common data structuring facilities are the array and the record (also called a structure). The array is a collection of homogeneous locations each identified by integer subscripts. The structure is composed of heterogeneous locations called fields each identified by name.

Control structure.

Programs are essentially algorithms thus our GPPL must support facilities for sequencing, choice and repetition. Note we exclude parallel execution. Much has been written in the literature about which control constructs are best. Those which do not easily and naturally express our ways of thinking ultimately may die out leaving a surprisingly small number of constructs. Although very few are

actually needed [Bohm66] it may be desirable to provide others which reflect in a more obvious manner, the abstractions of our problems. Our GPPL should support the one and two armed "if" construct for choice, and the "while" construct for repetition.

Abstraction mechanisms.

This is one of the most important features in programming languages. In the development of programs the desirability of specifying the "what" leaving the "how" aside, has long been realised. That is, we put aside inessential details at a particular level of abstraction and concentrate on the main theme. The practice of abstraction is realised in programming languages by the procedure or function. Our GPPL must support both with parameterisation. These entities lie at the heart of the way we should write programs and how we understand them. We will see that important as they are, they are not fully exploited by the Algol family and it is part of the purpose of this work to elevate their high status still further. This poses problems in their implementation.

Security.

Any language must have a clear policy about protecting the programmer from himself. Although it must provide him with the tools he needs for his task, it must be aware that he will make mistakes. Perhaps the most important area of security is that of type-checking. Some languages [Gunn80, Rich69] determine type by context. In these, for example if you add one value to another they are treated as integers. This is very much the assembly language approach and in fact such languages are regarded as low level. Their reasoning is that the higher level language approach to the store is too restrictive and not powerful enough for certain kinds of programming. Most of our present day machine architectures still adopt this contextual approach to data types. Arguments may exist about whether type checking should be

performed at compile time or run time, but it is agreed that nonsensical operations such as dividing a character by a boolean should be trapped. Other security areas are arithmetic exceptions, bad data structure accesses, use of uninitialised variables, nil pointers, array bound checks and so on.

Store.

We may separate the programmer's view of storage allocation into two parts, one for simple variable allocation and one for data structure allocation. The former is handled by the concept of scope and block structure. The latter embodies the concept of a heap where space is dynamically allocated without reference to the static layout of the program. Some languages allow this for structures but not arrays. We consider both desirable in our GPPL.

We have outlined the desirable features in a GPPL. Not everyone will agree with them, but we feel they are typical of many languages in current use. This then was our starting point. We attempted to find a way of implementing such a language which would be straightforward and reasonably efficient. That is, we try to find an appropriate mapping from a language to a machine.

Levels of increasing language power.

We look at three kinds of language, each more powerful than the previous by the introduction of novel, little-known or little-used features, and examine the implementation by abstract machine of these languages. The first language is a fairly typical member of the Algol family (Algol R). From there we look at a language similar to the first but which elevates procedures to the status of assignable values and which sets out to rationalise some other aspects of language features (h). Finally we look at the implementation of a language which sets out to exploit type polymorphism (nsl). The abstract machines for the first two

languages are designed with a view to code generation or interpretation, the third is so very high level that we felt code generation was inappropriate. Therefore we designed its abstract machine with interpretation in mind.

Summary.

In this introduction we have said that we wish to make life easier for the language implementor. An examination of previous work has led us in the direction of an implementation by a language dependant translator and a machine dependant translator. One interface between these programs has been the abstract machine. We have described the nature of an abstract machine and explained, on the basis of other researchers' work why this is a good technique for implementing a language. We have concentrated on two kinds of mapping a language onto a machine - translation and simulation. An abstract machine approach is feasible for both kinds. We have examined existing abstract machines and pointed out their weak points. The properties and characteristics of a high level general purpose programming language have been discussed.

In the following chapters we describe how we have implemented three increasingly more powerful languages by means of abstract machine modelling. From an initial experience-gaining implementation we attempted to develop a technique based upon the concept of abstract machines which would allow the implementor to design his own high level abstract machine language and use it as the input to a MDT. For this MDT we improved and expand a code generation method to complete our technique. We then showed that the technique could be applied to other language features and be used to output code for interpretation. Finally we used our abstract machine language design method to produce a very high level, interpretable abstract machine for a powerful polymorphic language.

CHAPTER 2

Implementation of a typical Algol.

Since the Algol family is widely used and recognised as being elegant, powerful examples of GPPLs, it was decided to implement Algol R. [Morr78] This language was based on Algol W [Site71] a language taught as a first programming language to undergraduates. It "cleaned up" a lot of the less desirable features of Algol W. An implementation for it did however exist so any subsequent one had to be an improvement. We consider this to be of average difficulty of implementation in terms of our scale of more powerful Algols. We mean by average that a reasonably competent programmer with a knowledge of language implementation techniques should be able to produce an implementation. This chapter will briefly describe the features of the language which make it a typical member of the Algol family and then describe its previous implementation. We also consider the code generation technique which was used. The next chapter is devoted to the work on its improved implementation.

The Algol R language.

This is a reasonably typical member of the Algol family, much better than Algol 60 and is a good example of a GPPL. Here, the main features with respect to implementation and the classification in the introduction are discussed.

Primitive Data Types.

Algol R supports the primitive data types integer, real, boolean character and struct. The usual operators are provided for the primitive types.

Data structures.

A struct is a pointer to a user defined data structure similar to the

record of Algol W and Pascal. The programmer writes a named template which then defines the pattern for creation of instances of that data structure. The other kind of data structure is the array. This is an object of one or more dimensions with a primitive as the base type.

Control structure.

Algol R is a sequential language, one action being performed after another. It supports choice in two main constructs - the "if" statement and the "case" statement. Repetition appears as the "while" statement and the usual sugared form, the "for" statement. An important and useful feature of Algol R is that the choice control constructs may also be used in expressions. The "if" and "case" expressions have arms which give values. The block expression is a block whose last component is not a statement but an expression which is evaluated to give the result of the whole block.

Abstraction Mechanisms.

Procedures and functions take a fixed number of formal parameters and are the abstractions of statements and expressions respectively. Passing of parameters is by value only. A procedure or a function name is allowed to be passed in on a call but these entities are not expressible values in the language.

Security.

Algol R is mainly compile time type checked. The type of a construct is manifest to the compiler. Run time checks must be imposed for some operations such as data structure accessing where array subscripts must lie within bounds, struct values must not be nil on accessing and field names must correspond to the class of structures accessed using them.

Store.

Variables of all data types are declared at the head of a block. This

reserves storage for the time control resides in the block. When control leaves the block these locations disappear. Arrays and structures, being allocated on the heap remain accessible only as long as a pointer to them resides in an accessible storage location.

These then are the high level language features which the implementor must satisfy in a reasonable fashion.

The first implementation.

The existing implementation was a compiler which generated PL360 [Wirt68] code. This was a systems language giving the user low level facilities but with high level control constructs. The prior implementation was moderately successful both in its design and maintenance and in its execution.

The compiler called code generation routines each based on an instruction for an abstract machine. Instead of these routines outputting a representation of the abstract machine instruction for interpretation, a section of PL360 code implementing the instruction was generated. It is a form of macro mapping. The effect is similar to interpretation except that the bodies of code which would have been in the interpreter main loop for each instruction, are now in line. Thus the overhead of repeated decoding is removed. This approach is well proven [Amm73] and excellent for a first or even permanent implementation. It has the disadvantage that it can generate large code. Code may be optimised by either "peephole" optimising the abstract machine instructions generated [Tane82] or on the generated machine code. [McKe65]

Approaches to code generation.

Most compilers during the syntax analysis phase also translate the source into a language intermediate in complexity between the high level language and real machine code. This is the machine code of the abstract

machine. It is done because translation in a single step

"makes generation of optimal or even relatively good code a difficult task" [Aho77]

The varieties of intermediate code all stem from an abstract syntax tree representing the source. There are some common representations such as,

Trees

An abstract syntax tree may be built explicitly and traversed during code generation as in [Baue73] or a representation of the tree may be produced for processing by a separate pass.

Reverse polish

This is a linear representation of the abstract syntax tree produced by a post order traversal.

n-tuples (typically triples and quadruples)

Here sequences of n-tuples [Grie71] are produced (we will confine this discussion to quadruples). A quadruple is a statement of the form,

$$A := B \text{ op } C$$

where A, B, C are either programmer-defined names, literals or compiler-generated temporary names. op is an operator. For example,

$(-a+b)*(c+d)$ gives

```
T1:=-a
T2:=T1+b
T3:=c+d
T4:=T2*T3
```

Cattell [Catt80] states that

"Code generators, which typically translate an intermediate notation into target machine code in one or more steps, have been relatively ad hoc as compared to the first phase of compilers, which translates a source language into the intermediate notation."

and discusses some implementations based on trees and n-tuples. We however decided to investigate an implementation based on reverse polish intermediate code. Our reasons are summarised in the following.

We wished to produce a code generator which was a separate pass because only this pass would need to be partly rewritten when porting the language to another machine. Such a split into LDT and MDT parts keeps the compiler logically and physically manageable. From a more practical point of view a compiler capable of producing such intermediate code already existed.

Building a tree would have been inappropriate and perhaps inefficient due to calls on a space allocator and the initialisation of the fields of nodes. The output of a tree to disk and its reconstitution by the code generator would have been time consuming. McKeeman [McKe74] states,

"tree transformations consume more computational resources than most other phases of translation both because the tree occupies a lot of memory and following the links takes a lot of time"

The use of n-tuples appeared to be more long winded due to the number of temporary variables which must be described. As will be seen we feel that it should not be the function of the first phase of the compiler to perform storage allocation.

The generation of reverse polish is very simple and does not require the tree to be explicitly built. Perhaps the most important reason for using a reverse polish intermediate code was the existence of a code generation technique based on this format, which appeared to be a good candidate for improvement.

The new implementation.

The next implementation of Algol R, which is the subject of the following chapter, is based on a code generation technique for expressions

and assignments described by Jensen. [Jens65] It was used in the code generator of the Gier Algol [Naur63.....] compiler. Similar techniques have been used in BCPL and IMP compilers. The paper describing it showed how code could be generated for a single accumulator machine with some dedicated registers. The technique appeared to be useful when writing a code generator for a particular machine on an ad hoc, one off basis. To implement the same language on another machine, the code generator had to be rewritten.

It was decided to investigate this little known or used technique in its application to Algol R running on an IBM/360 even though it was limited to expressions. The hope was that an improved method of code generation might arise by extending the technique to marry with the code generation of control structures.

The Method of Pseudo Evaluation as applied to Algol 60.

The input to Jensen's code generator is an intermediate form of expressions in reverse polish. This is very easy to generate and ideal for stack machine evaluation. For those machines which do not support a hardware stack, the code generator must output code whose end effect is the same as execution of stack machine code. Jensen describes the method generating code for a single accumulator machine stating,

"However, in a machine which has built in floating point operations but no special facilities for working on a stack, it will normally be faster to perform operations directly in the accumulator of the machine and to use named working locations instead of the anonymous ones which reverse polish implies."

Waite [Wait74] describes code generation as simulating the evaluation procedure in the environment (register organisation and addressing structure) of the target machine. The code generator holds a description

of the run time contents of the environment, code being emitted and the description updated when advised by the simulation. In the same manner Jensen's method models at compile time, the run time execution of the reverse polish on a stack machine, in a similar way to an interpreter for such a machine. Instead of using actual operands on the stack, it works with descriptions of the operands. In performing the simulation, code is generated, which when executed at run time effects the changes in the model. Note that there is a code generation model of a stack abstract machine and a run time model. The latter may not mimic the actual stack like the code generation model but it still produces the same result.

The algorithm.

The reverse polish is considered to be the machine language of a stack machine, which Jensen interprets as follows.

- (1) Proceed through the reverse polish form of the expression from left to right.
- (2) When an operand is encountered, place its description on top of the stack.
- (3) When an operator is encountered, perform the corresponding operation on the top element(s) of the stack. This involves the generation of code to perform the action at run time, using the information in the operand descriptions. Change the stack of descriptions to reflect the result of the operation.

The operand descriptions.

Jensen divides his operand description into two parts, the class information and the address information. The former describes what kind of operand it is. It for example may determine which machine instructions are generated. The address information tells where the operand will lie at run time. This may be implicit in the class information and thus will be

irrelevant in the corresponding address part. Some of the classes are :

- variable where value is required
- working storage where value is required
- constant
- the accumulator
- the floating point accumulator

These direct the code generator in what code to generate for operations. For example, the machine may not have an instruction to add the contents of two storage locations. Should both operands of an add operation lie in storage locations, then code must be generated to load one into the accumulator. Then an add store to accumulator instruction must be generated. This is determined by examining the classes of the operands.

The address information is not necessary for classes which are accumulators since the class is also the address. For constants, the address information contains the value, and for variables and working storage it is a block number, block offset pair.

An example

The following example is typical of the method. The following abbreviations are used.

vx - variable named x
 wl - working storage cell 1
 A - accumulator

Source : (a + b) * (c + d)

reverse polish : va vb + vc vd + *

Model :

Input	Output	Stack ->	Comment
va		va	push description
vb		va vb	" "
+	A := b A := A + a	va A A	load top of stack into A add a to accumulator
vc		A vc	push description
vd		A vc vd	" "
+	wl := A A := d A := A + c	wl vc vd wl vc A wl A	need A so store it load A with d add
*	A := A * wl	A	multiply

The inputs va and vb cause their descriptions to be pushed on the stack. When the + is met the code generator must output code which will add a and b leaving the result somewhere. Where will depend on the target machine architecture and the run time support of the language. In this example we assume an accumulator machine. The code generator model must be updated to reflect the operation. Note that the code generator models a stack thus it pops the operand descriptors and pushes the result descriptor, that is, of the accumulator. At run time b is loaded into the accumulator and then a is added to it. The descriptors for c and d are pushed. Addition is as before except the accumulator must be stored in a working location before it is free to use. The descriptor is updated to reflect this. Finally code is generated to perform the multiplication, the result being in the accumulator.

Assignment using the model.

The example above dealt with operand values only. In the case of assignment to a simple variable we care about its location not its contents. A further class is added for this - "variable where address is required". When generating code which may involve a side effect of changing a value in a variable then operands whose class is "variable .. value .." on the simulated stack must be saved in working storage. This involves a search up the stack. It must be done since we are concerned with the values in the variables. If the variable might be updated then we must preserve this value. Operands whose class is "variable .. address .." need not be saved since we are interested in the location not its contents.

Comments on the method.

Pseudo evaluation is a convenient means of keeping track of operand values and addresses. The stack model supporting this is well understood and effective for expression evaluation. The quality of the code generated for a particular machine depends to some extent on the writer of the code generator and not on the model. The value of pseudo evaluation lies in its organisational enforcement. There is little in the technique which automatically causes the generation of optimal code sequences.

However it does have something to offer because it is simple, readily understood and provides an excellent framework onto which may be built some simple optimisations. It can be improved upon by embedding it in a method for generating code for all language facilities not just expressions. The improved method has to be designed bearing pseudo evaluation in mind and must cater for all features of the language it is used to implement. We must not forget that the level of the enhanced intermediate code for Algol R must also be as high as possible and the code generation technique should not overly limit this.

Possible Improvements and Extensions to the Method.

It was decided that pseudo evaluation was a reasonable way of generating expression code either for a real machine or an interpreter. There also was scope for improving and extending the technique. If this was done, combining it with a method of code generation for other language features, perhaps a reasonable code generation technique might evolve. With this goal in mind the first task was to analyse pseudo evaluation and examine the areas which could be improved. These are discussed below.

Control and Data Structures

Jensen states that

"The basic method is a pseudo evaluation of expressions in the text." (my emphasis).

He does not mention flow of control or data structure creation. Pseudo evaluation is only used when expressions are met. The work in hand was to design a coherent technique for a complete language using pseudo evaluation as much as possible, perhaps even in code generation for control and data structures.

Types

Jensen gives examples using arithmetic operations alone. The Boolean operators in some Algols are hybrid control structures. In Algol 60 they are strict. [Naur63] Later Algols have more complex data structures and a wider range of types. The new method must be able to cope with some of these. No type information is used by Jensen's code generator, his descriptions contain class and address information only. As Hoare [Dahl72] says, a type determines the representation of a value. Surely this is a concern of the code generator as well as the compiler.

Storage Allocation

Jensen's method is very weak in this area. First of all, his values of whatever type take up the same amount of space. This makes for a very simple implementation but may be inadequate and unrealistic on machines with limited address space such as mini- and microcomputers. Secondly, his compiler does the space allocation for variables whereas the code generator does it for working storage. Each uses a different method of referring to the respective storage. As will be seen later, the code generator should do all storage allocation in a uniform way - using type information provided by the compiler.

Registers

Jensen describes code generation for a single accumulator machine. A number of architectures support several general purpose registers. [DEC81, DEC71, IBM70] While some of these may be dedicated in a particular implementation some may not. These should take part in expression evaluation to act as fast working storage where possible. This will involve the additional task of finding a register allocation algorithm.

Heap

Algol 60 does not employ a heap. The later Algols do have side effect problems involved with the non deterministic nature of garbage collection. For example, in Jensen's model a dedicated register points at elements of arrays at each level of subscription. That is, it points into the middle of the array. Now he knows by the nature of Algol 60 that the array will still be there when he accesses that element. This may not be the case with an Algol such as Algol R where arrays are assignable values. For example,

```

begin
  integer array ( 1 :: 10 ) fred ;
  integer array ( 40 :: 60 ) jim ;
  :
  fred( 10 ) := begin fred := jim ; ... 6 end ;
  :
end

```

In the above program fragment a value is assigned to the 10th element of the array currently in "fred" just before calculation of the assignment value. However the array value in "fred" is changed by the side effect. The ellipsis code could cause a garbage collection in which case the old array in "fred" might no longer be accessible. Now Jensen keeps a pointer (UAa) directly to the array element being accessed. After the assignment to "fred" the original array is "floating around" with nothing referring to it - thus it might be garbage collected. If so, no harm is done since the assignment to the tenth element will change a location in the free space of the heap. But, if the space is reallocated it will have the nasty effect of changing a value in a new data structure. Another consideration involving a heap is the identification of values which are pointers and the following of these during the marking phase of garbage collection. Thus a heap involves considerably more work in ensuring its consistency at run time.

Descriptions

The interface between Jensen's compiler and code generator is reverse polish. His description of a variable in this interface code amounts to nothing apart from its name. As we have seen, type information should be included also. We note that other entities such as procedures should be described and perhaps take part in the simulation.

Portability

Jensen describes a technique used for the implementation of Algol 60 on a particular machine. It is desirable to make available a good

high level language on a variety of machines. Thus the new code generation technique must consider portability with respect to the compiler and the code generator. In the implementation of a language there are three levels of abstraction, the source, the underlying abstract machine and the real machine. As far as portability is concerned we must investigate the mapping between the source and the abstract machine code and between the abstract and the real machine.

Optimisation

"The process of object code optimisation can be considered in two forms, often referred to as local and global optimisation. The first technique involves the optimisation of small subsections of the generated coding without being concerned with any overall features displayed in the program. The second form on the other hand, considers the whole source program in an attempt to improve the efficiency of the object code generated." [Bril72]

The technique of pseudo-evaluation lends itself to both local and global optimisation. Richards [Rich71] says

"Good global organisation is often a better way of achieving efficient code than any amount of local optimisation."

He suggests that general design decisions such as register dedication, procedure calling mechanisms, data representation, stack organisation, variable allocation and addressing are all very important. These all can be done with pseudo-evaluation and to some extent help form the simulated run time environment. The technique allows a reasonable level of local optimisation, more than peephole optimisation would allow. This may be achieved by the fact that data is described in the simulation. Instead of emitting code for an operation we need only

change the description of the result. This reflects that code production for some operations has been delayed and provides information on what must be done. This goes some way towards the situation in producing code from a tree where all subnodes are available at the same time as a node. Here, because of the postfix notation we only meet the node after the subnodes have been processed. Better code may be generated in the light of a wider context thus by building up enough information we are in a stronger position to optimise. We suggest that this is better than peephole optimisation on generated code because it is easier to save descriptions, no further pass is necessary and the context may be wider than a practicable peephole window. The technique however does not lend itself to control or data flow analysis. Similar optimisation techniques are outwith the scope of this thesis.

Summary.

Our aim was to provide a method of code generation which is readily understandable, implementable without much difficulty, able to provide reasonably good code and able to have optimisation added if necessary. One of the problems with local optimisation is the number of ad hoc rules introduced [Robe81] which tend to obscure the underlying technique. We feel that a straightforward pseudo-evaluation code generator should be written after some time considering global organisation. Some measurement should be taken of the code produced and then local optimisations layered on top.

Jensen's pseudo-evaluation technique certainly seemed to potentially fit the criteria above. Other code generation techniques were considered and rejected due mainly to their difficulty of understanding or implementation. The existence of a simple improvable technique such as Jensen's proved to be a highly attractive magnet. The next chapter

describes the implementation of Algol R using pseudo-evaluation.

CHAPTER 3

Intermediate Code and Code Generator for Algol R.

This chapter describes the implementation of Algol R. It is in two main sections, the first dealing with the design of the abstract machine intermediate code and the second dealing with the code generator.

The design of the intermediate code.

A compiler written by R. Morrison produced PL360 [Wirt68] code for an existing abstract machine. This code was used as a starting point in the design of the new intermediate code. It was chosen because of its similarity to Jensen's code, being reverse polish in nature and because it already existed. It was thought that modifying it would be a straightforward task. Although much was learned at this stage the intermediate code should really have been designed from scratch. The existing abstract machine was a stack machine using reverse polish like instructions for expressions. The abstract machine operators are described in [Morr76]. The intermediate code is described in Appendix A. The relationship between the Algol R source and the intermediate code is also shown. The major changes were to the flow of control instructions which were raised from the status of jumps and labels closer to the high level language constructs they implemented. A means of describing source entities had also to be added.

Where an abstract machine is to be interpreted, the compiler performs the code optimisation with all the knowledge of the source to hand. A code generator, in the same way, must have this knowledge also, thus the intermediate code must represent both the algorithm given by the source and the description of the data. This simplifies the compiler's task, passing on the responsibility of code optimisation to the code generator. Should the compiler make optimisation decisions and generate an intermediate code

which has lost some of the essential source information then the code generator cannot be expected to produce better code than if it had all the information available. [Brow69,Brow72,Korn80] The compiler must produce an intermediate code which is in a convenient form for subsequent processing and loses little, if any, of the source information. Most importantly it must have no implementation decisions imposed on it. It must perform the usual analysis and generate an intermediate code at a level of abstraction which does not limit the implementation. It is thus very important that the implementation be split into an LDT and MDT.

As an example of this approach, consider the original abstract machine for Algol R. This was implemented by outputting PL360 code by the code generation procedures which each "implemented" an abstract machine instruction. Variables which were to contain pointers to heap entities were allocated on a separate stack. This was to allow the collecting together of pointer values for easier handling of garbage collection. The abstract machine then had to have two stacks although there is nothing inherent in the source language which forces this. Had this aspect of the abstract machine been used in the design of an intermediate code then all code generators using that code would have been forced to implement two stacks.

Now it is conceivable that the language could have been implemented on some radical or novel architecture, for example say one with tagging. [Ilif68] All values, including pointer values could then reside on a single stack, pointers being differentiated by a unique tag. Thus the compiler must tell the code generator that space for a pointer value is to be allocated, but it must not tell it where and how much - this is the code generator's decision alone.

We now describe some of the fundamental design aspects of the intermediate code which preserve such information and also make it easy to handle in the code generator.

Intermediate code and control structures.

A trend [Rich71,Robe81,Nori74,Bail80] in intermediate code design is to throw away the essential structure of control structures. (Tree intermediate forms by their nature do not fall into this trap). The abstraction of say a "while" loop is lowered by generating jumps and labels. It may seem that in doing so a meagre amount of information is lost, since after all the real machine code will itself be in terms of jumps. Unfortunately in supposedly simplifying such a control structure, the code generator is made more difficult to write purely because the structure is lost. An analogy could be drawn with the use of goto's. [Dijk68]

"It is very important that the original constructions from the source language can be recognised. For instance, to make efficient use of registers, one must know whether a label in the intermediate form is part of an if-statement or can be branched to by a goto-statement" [Korn80]

In the design of the intermediate code the structure was preserved, in fact it appears very much like the source code. This had the advantage that debugging the intermediate code generated was very easy in this respect. The actual syntax of the intermediate code is simple and straightforward to process by the code generator. An example of the code generated follows.

<pre>while a + b < 6 do begin ... end</pre>	<pre>while stack 3 stack 4 plus.op stackconst i 6 ls.op endbool block ... endblock endwhile</pre>
--	---

Variable allocation and referencing.

Another area in which the intermediate code must not be too low level is that of referring to variables. At the highest level of abstraction,

the source language works in terms of typed values residing in locations. The objects of different types may eventually take up different amounts of storage in the real machine. In generating code for an abstract machine, a compiler (for example [Nori74]) allocates space in terms of these objects and the addressing structure of the machine. For example it may decide that a real is four bytes long, a boolean one byte long. It then calculates addresses on this basis. If this is passed on to a code generator then no optimisation is possible by that code generator. Of course, one of the two passes must decide. If it is the compiler then that compiler must be rewritten if the sizes are changed. If it is the code generator then only it needs to be changed.

Every variable in an Algol R source program has a type. It is the responsibility of the compiler to check that it will only contain values of that type throughout its lifetime. As has been mentioned the type information is required by the code generator to determine what representation a value will have at run time, and perhaps where it will reside. Variables are named, the scope rules sorting out references to them. This name is not necessary for the code generator unless say some readable form of run time error dump is required. Yet the code generator must have a means of uniquely determining variables. In an abstract machine designed for interpretation, variables are accessed by their location in the store. Several variables perhaps share the same location at different times. As we have seen the compiler must not allocate storage for variables and so cannot allocate addresses. Instead it associates a number with each variable and uses this number in all references to that variable. In the same way as names need not be unique (because of the scope rules), the numbers allocated need not be unique.

Descriptors.

Variables are not the only entities which need to be described and uniquely referred to. Procedures are typed, named entities as are structure fields. We generalise this by saying that any entity which needs unique identification and has certain attributes, must be described to the code generator. This associates a reference number with the entity and allows the code generator to build an internal description both from attributes provided by the source description and attributes determined by itself perhaps in terms of the real machine. This description is held internally by the code generator in the form of descriptors. The Algol R compiler generates partial descriptions for variables, procedures, structure classes and structure fields. Complete descriptors will be built by the code generator for these and for transient values arising from the process of pseudo evaluation.

For example, variables within a block are described and referred to as follows,

begin	block
integer x ;	declare 5 x integer
char y ;	declare 6 y char
...	enddecl
... x ... y stack 5 ... stack 6 ...
end	free 5 6
	endblock

The Code Generator.

Code is generated according to the intermediate form of the source program and some run time support. The latter reflects some of the higher aspects of the abstract machine such as the implementation of a heap and input/output facilities. Usually these interface with the generated code through calls of assembler routines. The run time support will be assumed in the following.

The Algol R code generator is itself written in Algol R. It outputs assembly code for the IBM /360 model 44. As far as possible it will be described without reference to this machine because it was designed with a view to portability and general technique. The operation and structure of the code generator depends on the underlying abstract machine and not on the target machine. Such a design means that a large part of the code generator is machine independent and need not be rewritten for a different target machine.

The structure of the code generator is dictated by two main considerations. One is the procedural nature of Algol R, the other is the stack architecture of the simulation. The source main program is compiled into a segment of intermediate code as is each procedure and function body. It is convenient to consider the main program as a procedure of the operating system because it no longer is a special case. Thus the code generator basically consists of a call of a routine called "segment" to generate code for an intermediate code segment. It generates entry, body and exit code. Within the intermediate code will be nested segments (one for each procedure) produced by the compiler so this routine is recursive.

The overall structure of the code generator is reminiscent of that of a recursive descent compiler. There are routines for each syntactic entity of the intermediate code whose function is to generate code for that entity. (In a recursive descent compiler we also parse the code.) The intermediate code instructions fall into two groups, those with structure, such as "if" and "while", and those which are reverse polish in nature. The latter have some effect on the run time stack either to change it or to push a single value on it. They are treated by handling them one after the other using a pseudo evaluation algorithm similar to Jensens', until directed to stop by a delimiter. A routine called "generate" produces code and handles the simulation of a sequence of reverse polish intermediate code instructions. It takes as a parameter the delimiter which marks the

end of the sequence. Sequences of reverse polish instructions and structured instructions form the subcomponents of structured instructions. "Generate" consists of a loop similar to an interpreter loop which repeatedly reads an instruction and simulates its evaluation possibly generating code for it. Those routines handling structured instructions will recursively call "generate" for their subsequences. We do not propose to describe code sequences generated but intend to consider what happens in the code generator for various features of Algol R. This is to demonstrate the validity of the intermediate code design in the context of implementing a typical Algol. In the following the intermediate code description in Appendix A may be referred to.

Descriptor Creation.

A "declare" instruction causes the creation of a descriptor by the code generator. All such descriptors are modelled by structure instances and pointers to them are held in an array. The descriptor number is used to index this array to access the descriptor. This array is called the descriptor definition stack. It is a stack because of the LIFO nature of the allocation and deallocation of descriptor numbers as the code generator progresses through the intermediate code.

Descriptors also reside on the simulated evaluation stack. This again is an array of pointers. Here, they describe transient entities whereas the descriptor definition stack contains the descriptions of those declared entities currently accessible at that point in the intermediate code.

A descriptor of a value has three components.

Declared

This boolean field determines whether the value lies in a declared variable or a temporary storage location (this is only relevant if the value is resident on the run time stack since both variables and temporary results reside there). It is false when stack values are

temporary, being the result of a subexpression. When no longer required the space on the stack used for temporary values can be made available for reuse immediately. Variable space can only be released on block exit.

Type

A value is classed as primitive, array, procedure, structure or field depending on its Algol R type. Each kind has its own descriptor of the type. These are,

Primitive.

The types integer, real, boolean, character and struct (pointer to structure) are each described by a primitive descriptor. This has three components, the type name (not actually used but could be for debugging purposes), the byte boundary on which values of the type must lie (this would only be relevant for certain architectures such as the /360) and the size in bytes of the values. These are used in space allocation.

Array. Array type descriptors have two components, the base type and the number of dimensions.

Procedure.

This has four components, the procedure name, its result type, a list of its parameter types and an indication of whether it is present, interface or external (see later).

Structure.

This has three components. One is the number of the descriptor of the first field of the structure and the other two relate to housekeeping information used at run time. One of these is a unique trademark for the structure class. This identifies all structures of that class at run time. The other is garbage collector flag settings.

Field. This has three components. One is the offset of the field within its structure. Another is a link to a descriptor of its following field. The third is the trademark of its containing structure class. Descriptors of fields of the same structure class are linked together via the type descriptors.

Location

This component of a descriptor is perhaps the most machine dependant, it being a description of where the value lies at run time. A value is not restricted to permanently residing in one place and may not even exist as a bit pattern at run time but as a representation (e.g. a condition code). Even though the location is heavily machine dependant we may reduce this by abstracting over common architectures in the design phase of code generator construction. This means looking at typical characteristics of real machines in relation to the intermediate code and Algol R objects. From these observations developed the location descriptions for Algol R values.

Manifest Location

We first note that some values may be manifest at compile time. There is no need to generate code to evaluate these values since we may generate them directly in store, possibly as immediate operands of instructions. Also Jensen points out that in pseudo evaluation the code generator may perform expression evaluation where the operands are manifest values. This is known as constant folding. Therefore one class of location is **manifest**. Note this is machine independant. Types which may have manifest locations in the Algol R code generator are boolean, character, integer, real, one dimensional arrays of manifest characters and procedures. The IBM /360 does not allow immediate operands except for small values eight bits or less (move immediate instruction [IBM70]). Larger values are held in an area of store called the literal pool. The

code generator ensures that only one copy of a large manifest value resides in the literal pool.

Register Location

In common with other makes of machine the IBM /360 has general purpose registers. Instructions using these are shorter and faster than those employing store. Another class of location is therefore **register**. Registers may be dedicated, that is, they may perform a specific function throughout the life of the program. For example, in the /360 implementation of Algol R, a register is dedicated to always point at the base of the topmost frame on the stack. Its contents may change but not its function. Dedicated registers may not in general take part in expression evaluation. Those which do must themselves be identified to the code generator by register descriptors. These contain components such as the register number, its type (on the /360 there are two kinds, floating point and fixed point), and others to aid the simulation (see register allocation).

Stack Location

Variables and temporary values reside on the stack. In the implementation of Algols these are usually identified by a procedure (or block) level number and an offset. [Rand64] The level number is converted at run time to a frame pointer by means of a display or static chain mechanism. There is one class, **stack**, describing values on the stack. In fact two stacks are supported, one for pointer values, however for reasons given previously the location descriptions involve only one stack class. This has components such as which stack (pointer or main), procedure level and offset within frame. As with register descriptors, there are also components used in the modelling of location allocation.

Heap Location

Locations may also be in arrays or structures. As has been seen it is not sufficient to simply point at a location on the heap (we may lose the array or structure on garbage collection). We must refer to it by a base, offset pair where the base is a pointer to the start of the array or structure. There is a class for such **heap** locations with a component for the base and the offset. The base component will be a register or (pointer) stack descriptor, describing where the base pointer is held at run time. Offsets may be manifest.

Boolean Location

There are three final classes of location all to do with boolean values. These are not descriptions of physical locations at run time but representations of boolean values.

Condition Code Representation

After test instructions, machines usually set some flag in the processor status word which reflects the result - say zero or non-zero. This condition code may be used as a representation of a boolean value, but only up to generation of some instruction which changes the flag! The flag has two states, each being able to represent true or false. The location description then is a pair

< flag state, boolean value represented >.

For example, < on, true > represents true if the condition code flag is on and false if it is off. This representation is needed for the generation of short code sequences as will be seen later, however it does need vigilance on the code generator writer's part to generate code to convert it to a more permanent representation before he generates code which could change the flag.

Label Representation

The boolean values false and true are commonly represented by zero and non-zero values respectively. Most machines have instructions to test a value and subsequently jump to a label or not depending on the result of whether it was zero or not. Thus a value may be converted to a representation as a label with an associated boolean tag. For example the code sequence,

```
test R
jump if
non zero
to label T
```

is a representation of a boolean value. True is represented by landing at the label T yet to be planted, false is represented by carrying on after the jump. Let us define this representation to be a "true label" representation and depict it as

---t--->T

or "boollabel(true, T)". Note this implies "carry on in line if false". We could have generated the sequence

```
test R
jump if
zero
to label F
```

in which case we have a "false label" representation of the same value depicted as

---f--->F

Given such a representation we can reverse the process and get a storable value. This is done by

```
1) boollabel( true, T ) -> Register( R )
```

```
---t--->    ->    R
```

```
R <- false
goto X
T: R <- true
X:
```

```
2) boollabel( false, F ) -> register( R )
```

```
    ---f--->F    ->    R
```

```
R <- true
goto X
F: R <- false
X:
```

Obviously in simple assignments there is no point in converting a value to a label representation and vice versa. They are best used in the binary logical operations **and**, and **or** which are "lazy" in that only the minimum number of operands are evaluated to determine the result.

The Simulated Evaluation stack.

Each call of the generate routine has its own simulated evaluation stack. The nature of reverse polish and the intermediate code is such that a code sequence processed by the generate routine does not affect objects lower down the stack. Thus a private stack may be used. This of course starts off empty, grows and contracts as code is processed up to a delimiter. On meeting the delimiter, the stack will be empty or contain a single descriptor of the result of the code sequence. All sequences correspond to the run time effect of producing a value or causing a change of memory location contents or both. The generate routine returns this description to its caller, which may stack it on its private stack. Each procedure which generates code for an intermediate instruction, on entry will use the stack for descriptions of that instruction's operands. On exit, it will adjust the stack to reflect the run time results of the generated code. The stack is not used in between these times. For example, a "while" instruction (see Appendix A) contains two subsequences, one for the boolean condition and one for the body. The generate routine is called twice recursively with appropriate delimiters for each. The first call returns a descriptor of the boolean value. This

descriptor is not pushed on the evaluation stack but thrown away after appropriate code is generated for the test jump. The second call does not return a descriptor since the loop body does not return a value.

Frame Space Allocation.

The implementation, for the same reasons as the previous abstract machine, uses two stacks, one being for pointer values. Space at run time for variables and temporary values is allocated on either the main stack or the pointer stack. A simulation of these stacks is held for each segment during code generation since a routine may only allocate space within its own frame. At run time the address of the base of the topmost frame is held in a dedicated register and addressing of variables and temporary results is performed using this base register and a byte offset. Values of the different types are of different sizes on the /360 and start on different byte boundaries as given by the following table.

Type	Stack	Size	Boundary
integer	main	4	4
real	main	4	4
character	main	1	1
boolean	main	1	1
struct	pointer	4	4
array	pointer	4	4
procedure	main	8	8

When the code generator needs the address of space for a new location an allocate routine is called with the type as a parameter. It examines the space simulation on the appropriate stack and returns the offset of an as yet unused location for the type. It may be desired to use as little space as possible requiring values to be optimally packed in a frame. When such space is to be made available for reuse say on block exit or when a temporary result has been used, a corresponding deallocation routine adjusts the space simulation.

The space simulation can be simple or complex depending on the implementation criteria. What is more important is that the rest of the code generator should not depend on how or where space is allocated within a frame. It provides a type and expects the offset of an area of space of the required size, on the correct byte boundary and on the appropriate stack. The current simulation involves a first fit algorithm, searching the frame model from the bottom until an appropriate "hole" is found. This is marked as allocated in the model and its address returned. Deallocation simply involves marking the area in the model as unused.

Register Allocation.

A similar model is that of register allocation. Registers are faster than store so it is desirable to keep as many operands in them as possible. In fact, depending on the architecture, operands may need to be in registers for particular instructions. The Algol R code generator employs a reasonably simple but powerful model for non dedicated register allocation. (there are 6 dedicated 8 non dedicated registers) On the /360 there are two sets of registers, one for floating point, one for general purpose. We will consider the latter since a similar situation exists for allocation of floating point registers. The code generator maintains two lists, one of registers in use and one of free registers. On entry to a routine all non dedicated registers are available for use, thus the segment routine creates an empty use list and a full free list. Each register is described by a structure which has a field which may contain a pointer to a descriptor on the simulated stack. Every register on the use list has this field pointing at operand descriptors on the simulated stack. These operands lie in the corresponding registers, that is the location fields point to the register descriptors.

When a register is required, the topmost on the free list is removed. If there is not one the bottommost on the use list is moved to the top of the use list, it now being the most recently used. A temporary storage

location is allocated and code generated to store the register contents into it. The descriptor is updated to reflect the new location of the operand value. Instead of searching the simulated stack for the corresponding operand descriptor, this is obtained immediately from the register descriptor and updated to reflect the store - that is its new location field is "stack". Thus storage locations are only used when a register is needed and all are in use. When a particular register is required which is on the use list a similar procedure is followed except that its contents are saved in a register from the free list if that is not empty, or in storage otherwise.

Control Structures.

Control Structures have a major effect on the simulation. As we proceed through a reverse polish intermediate code sequence in the procedure "generate", the simulation changes its private evaluation stack. Most importantly the locations of operand values change throughout, controlled by the decisions taken by the simulation and the instructions. When a choice instruction is met we must somehow freeze the simulation stack. Let us first consider a multi-armed choice construct such as "case". For example,

```

case stack 6 of
<arm1 code>
endswitch 3 1 9 8
<arm2 code>
endswitch 1 5
<arm3 code>
endswitch 0
switchop
endcase

```

The code generator first meets the selecting value code then the arms in turn. The code generator must save the run time state simulation after the selecting value code has been processed. It must reproduce this state before processing each of the arms in turn. The arms must also be made to produce identical states after code has been generated for them. This is

because a common piece of code is executed after the multiway construct whatever branch has been taken. Since the code generator meets arm 1 first it may choose to save the state produced by it and generate code at the end of the other arms to conform to this. Its operation would then be

```

generate code for test
save state ( call it state 1 )
generate code for arm 1
save state ( call it state 2 )
restore state 1
generate code for arm 2
generate code to adjust to state 2
:
:
restore state 1
generate code for last arm
generate code to adjust to state 2

```

Now the saving and restoring involves copying complete data structures in the simulation, not just pointers to them. Furthermore, the state of the simulation locations after arm 1 may be nothing like those after the other arms. Arbitrarily complex code may need to be generated to restore the arm 1 state. Note states will not differ in the operands on the simulated stack but only in their locations.

This situation was unsatisfactory and it seemed that a marriage between control structures and pseudo evaluation might involve a great deal of overhead in the code generator. Further analysis did however come up with a solution. Two questions were asked. How do we avoid possibly complex adjustment code and how do we avoid copying when saving states? The answer to the first helped solve the second.

All that could change in the code generation of an arm were the registers. These might be reallocated, their contents being dumped during arm simulation and would need to be reloaded with the dumped values to readjust the state. Also an arm of a control construct which was an expression would produce a new value in some appropriate location. The solution was quite simple - dump all the registers in use immediately after the test. Now, there was no need to save the state after the test because

the simulation of the arms could not change it, nothing being in a register. The code adjustment at the end of each arm is only necessary if the arms produce values. Even then, all that is required is to ensure that the result of arm 1 is in a "reasonable" location, say a register, or a label value for booleans, and to generate trivial adjustment code at the ends of the other arms to conform to this.

Repetition becomes even simpler with this method. The problem in this case is to ensure that the state after the loop body is the same as the state before the test. This state must also be the one immediately after the construct. By dumping the registers before the test, nothing can be changed by the simulation of the body. This time there is no new value to worry about. It is believed that the overhead of dumping registers will not be significant and the ease of implementation warrants its use.

Data Structures.

Algol R supports structures and arrays. These are both allocated on the heap. Arrays are built in array declarations by the "iliffe.op" instruction (see Appendix A). This is a complex operation and rather than generate in line code, a call is generated to an interface procedure to allocate space on the heap. Structures are built using a similar call but with in line code for initialising the fields. The intermediate code specifies the class of structure to be built by a stacking of the structure descriptor and a structured instruction "formvec". This latter instruction has subsequences for the initialising values of the fields.

The routine generating code for "formvec" uses the structure descriptor to determine the size of space needed on the heap and generates a call of an interface procedure to allocate it. Then it calls "generate" for each of the fields producing code to store the resulting value in the run time structure. (Note that values are packed in structures to minimise space. On meeting the "declare" for a structure and its fields,

their offsets within the structure are calculated and form part of their description.) Finally a descriptor of the struct value resulting is pushed on the simulated stack.

Accessing of an array element or structure field is done by the "sub.op" instruction. On top of the simulated stack are the descriptors of the index and array or the field and the pointer to the structure. The code generator determines which by examining the type of the second top of the evaluation stack which will be array or struct. With the former, the top will be a descriptor of an integer offset and for the latter, a field descriptor. Code is generated to perform the necessary checks and access the element or field, all the information necessary being obtained by means of the descriptors.

Thus extending pseudo evaluation to include data structures poses no problems, the descriptors being a natural and convenient way of holding type and location information concisely.

Procedure Calling and Runtime Support.

A procedure call has four instructions (see Appendix A). One stacks its description, another prepares a new frame, a third evaluates the parameters if any and the last calls it. A call is a simple control structure to the code generator. Since a new frame is to be built on top of the existing one and it is not known which register's contents will change, the code generator must arrange for all values not already in store to be dumped to store. It is at this point that it knows from where the new frame can start and allocates space there for the run time housekeeping information such as the return address and frame links. The allocation model is adjusted so that any space allocated will be above this housekeeping space, that is, in the yet to be completed frame. Code is generated to store the parameter in an allocated location, and the descriptor popped.

Generating code for the call also involves a restoration of the storage allocation model to the point before the housekeeping space allocation. Having popped all the descriptions to do with the call, the simulated stack will be as it was before the call. Function calls involve pushing a descriptor of the returned value. This is returned in a fixed register, code having been generated at the end of the function to ensure this.

The run time support is a number of procedures and functions to carry out such duties as allocating space on the heap, garbage collection, and array building. These had already been written by R. Morrison for the previous implementation. Some were written in Algol R calling machine code routines. The Algol R routines went through the code generator. This meant having a few ad hoc sections of the code generator to deal with such external (i.e. machine code) and interface (i.e. Algol R) run time support. Even so, it greatly reduced the amount of low level programming which was needed. The technique coped with it admirably, information as to whether a procedure was interface, external or not being held in a descriptor.

Summary.

We think that the technique of pseudo evaluation is suitable for languages like Algol R but it must be married to a means of code generation for features of the language other than just expressions. We believe we have shown, by the design of a suitable intermediate code that this can be done, allowing a simple structured code generator. The level of this intermediate code should be higher than that of previous codes in order to reflect the structure of the source language. This allows the design of a code generator which is structured according to the intermediate code in the same manner as a recursive descent code generator is structured according to the source it parses. By having reverse polish like sequences

in the intermediate code we may use a stack simulation as proposed by Jensen. This proved simple to do and understand.

The entities held in this simulation are described by descriptors which are easy to build and manipulate. They effectively model the source and run time attributes of objects. On top of this it is straightforward to superimpose models for allocating space within procedure frames and for allocating non-dedicated registers. Drawbacks involved with the method are associated with the difficulty of retaining information across several intermediate code instructions. This is desirable if a high level of local optimisation is to be achieved. A balance must be found where the quality of code generated is acceptable and the code generator is not too difficult to handle because of complex information representations and optimisation dependancies across instructions.

CHAPTER 4

The Programming Language h.

The technique of pseudo evaluation proved to be a reasonable way of implementing what might be called an "ordinary" Algol. The Algol family however has been enriched with more sophisticated kin, such as Pascal and Algol 68. [Wijn75] Other advances have appeared in the functional languages such as SASL. [Turn79] Furthering our aim to investigate languages and their implementation, it was decided to design an Algol-like language called h which incorporated novel and state-of-the-art features found in some current languages. There were two objectives, one a minor exercise in language design, the other to investigate implementation techniques for such powerful languages. We stress that the language design aspect of this work is secondary to implementation. It has been embarked on to provide a vehicle for the main topic, however we feel that we may have added to the features which may be included in programming languages. This chapter describes aspects of the language, the next covers its implementation.

Already existing languages were considered but they were not powerful enough or did not embody all of the features whose implementation needed to be investigated. Some did however, but without implementing them completely (for example procedures as values in Algol 68. [Wijn75] Davie [Davi79] explains and gives an example of how Algol 68 restricts the use of first class procedures).

Language Design

The language design aspect involved producing a member of the Algol family, following well known principles attributed to Tennent, [Tenn77] Landin [Land66] and Strachey. [Stra67] Morrison [Morr80] identified these and brought them together in the design of a simplified Algol. The three principles are,

The principle of correspondence

This principle states that the way names are introduced and used in a language should be the same everywhere in a program. The way names are introduced by declarations should be in a one to one correspondence with the way names are introduced as parameters. They need not share a common syntax but for each kind of declaration there should be an equivalent kind of parameter declaration. h applies this principle except in the area of structure declarations.

The principle of abstraction

Abstraction means ignoring unimportant detail and concentrating on the essential structure and nature of a problem. As far as language design is concerned it means recognising the semantically meaningful syntactic categories in a language and allowing abstraction over them. For example, abstracting over expressions gives functions, abstracting over statements gives procedures.

The principle of data type completeness

All data types must have the same civil rights in a language. The rules for using data types must be complete with no exceptions. For example if we are allowed arrays of a specific type then we should be allowed arrays of all types. If we may have sets of a specific type then we should allow sets of all types. If one type is allowed to be a parameter or result of a function then all types should be allowed.

The h Programming Language.

A number of languages were used as a foundation for h. These include Algol 60, [Naur63] CPL, [Barr63] Pascal, [Jens74] Algol W, [Site71] and Algol S. [Turn76] It attempts to embody the better features of these languages. The language is described in full in its reference manual [Gunn78] but its syntax may be seen in Appendix B where it is related to the abstract machine code generated for it. We give here a

brief overview of the language and concentrate on three main areas which may be considered to be important from either a language design or implementation point of view. We omit discussion of any topic covered in previous chapters where it arises in the h language or its implementation. The main topics of interest are user defined structures, routines as values and constancy of locations. We begin by describing the types manipulated by the language. h is statically type checked, all type errors being detected at compile time.

Void Type This is included to simplify the syntax at a small expense in the type rules. Void type is the type of what is known as a statement in other languages, that is, a construct in the language which does not produce a value and which may affect the flow of control in a program, such as a loop. A procedure type which has no result is said to return void type.

Primitive Types.

The basic types are integers, characters and logicals. Reals could also have been included but they were implemented in Algol R and would not have contributed to the work.

Enumerated types.

In h, an enumerated type is an ordered series of values defined by an enumerated type definition which lists names in order. This implicitly binds the names to the values.

e.g. `type weather = (rain, snow, sun)`

These are ordered types and they may be used as subscript values for vector accessing. The relational operators, "succ" (successor) and "pred" (predecessor) operators may be used with enumerated types. The operator "ord" (ordinal number) gives the position in the definition list for an

enumerated type value.

Files.

h supports a simple input-output interface which is adequate for a wide variety of needs. Files are ordered sequences of characters existing independantly of the program. A value of type file is a connection to a physical file.

Vectors.

These are the simpler of the two data structuring facilities which h provides. A vector value is a reference to a compound entity composed of an ordered sequence of locations all of which hold values of a particular type called the element type. Element types of vectors may be any h type including vector types. Thus two dimensional arrays may be represented by vectors of vectors. Subscript types may be integer or a user defined enumerated type.

Vector creation involves specifying the bounds and initialising values for each element. (All locations in h must be initialised when they come into existence). Vectors may also be made up of constant elements. (Constancy is described later). There are two ways of creating a vector.

a) Enumeration

e.g. at x make ['a', ch1, ch2.]

This creates a vector of three character locations initialised in turn by the results of evaluating the initialising code. The lower bound is x and the upper bound is x + 2.

e.g. at 2 make [at 1 make ["by", "be", "to"],
at 1 make ["the", "for"]]

This creates a vector of vectors. Note that the bounds are not part of the

type of a vector thus we may have elements of a vector which are vectors of differing lengths.

b) Repeated Evaluation.

e.g. vector m :: n val f(x.)

This creates a vector with lower bound m and upper bound n. Each element in turn is initialised by re-evaluating the initialising code thus allowing the prospect of a different initialising value for each element.

e.g. let i <- 0 ; q := vector 1 :: 10 val
 begin i := i + 1 ; i end

Here the vector elements are initialised with values from 1 to 10.

Strings.

A string is a possibly empty sequence of characters treated as a collection. It is composed of values not locations, thus unlike a vector the components of a string cannot be updated. A new string value must be created from already existing ones. This contrasts with older ways of considering strings as single dimensioned arrays of character locations with statically (Algol W) or dynamically (Algol R) known lengths and with lower bounds of zero or one. h, by virtue of orthogonality has vectors of characters in addition to strings. These of course only have the properties of vectors. Compared to these, strings are much more powerful entities with a richer set of operations such as length, concatenation and substring. With hindsight, it is over complex to have character vectors and strings. Characters should be eliminated from the language retaining strings.

Structures and Pointers. +

This is the first of the major parts in the design of h. We spend some time here on a discussion of it. The following sections show how we consider structures and pointers to them should be treated in programming languages. We demonstrate how some approaches are special cases of the general approach taken by h. We show how structure classes and field accessing are usually treated and describe an h construct which reduces the run time class checking of structure instances against field names, by performing most of the checks at compile time. [Gunn82]

Structure Classes and Pointer Types.

There are three approaches to the types of variables or constants allowed to contain a pointer value.

- A1) They may contain pointers to any class instance. This approach is adopted by S-Algol. [Morr79] The type is simply "pointer to structure".
- A2) They may contain pointers to instances of a specified single class as in Pascal. The type is "pointer to class x" where "x" is a class name.
- A3) They may contain pointers to instances of a specified set of classes. The Algol W reference is restricted like this. The type is "pointer to x, y, z etc." where "x", "y", "z" etc. are class names.

Approach 3, and the Pascal variant record allow the programmer the flexibility of referring to different "shapes" of structure under protection of the type rules. Approach 1 gives him complete freedom, which is probably not what he wants all the time. There will be compile or run time checks that only pointers to instances of the allowable classes are

+ The material on structures and pointers is published in [Gunn82]

stored in appropriately typed locations.

On accessing fields however, only approach 2 allows a compile time check that the field name belongs to the same class as the instance being accessed. For example, let "p" be a pointer to an instance and "p{ height }" be a proposed access to a field "height" in an instance of class "box". Consider the three approaches in turn with respect to this field access.

- A1) The compiler can only check that "p" is of type pointer. The check that it contains a value pointing to a "box" instance must be done at run time.
- A2) The compiler checks that "p" is of type pointer to "box" instances. No run time check is needed, since the compiler guarantees that no pointer to an instance of class other than "box" will reside in a location of type "pointer to box".
- A3) The compiler checks that "p" is a pointer with a class set and further that "box" is a member of this set. At run time however, a check must still be made that "p" contains a pointer to a "box" instance.

These run time checks will involve a "trademark" generated by the compiler for each class and carried round as part of each instance.

In h, all approaches are allowed. Values of type "ptr" may be pointers to any class instance. Values of type "ptr{ x }" may only point to instances of class "x". Values of type "ptr{ x, y, z }" may only point to "x" or "y" or "z" instances. In fact, it may be seen that approaches A1 and A2 are cases of A3 with an infinite class set and a class set of one member respectively. Let us define this scheme to be pointer-type restriction.

Structure Creation.

To create an instance of a structure in *h*, the programmer must specify the class name and supply expressions which will be evaluated to initialise the fields. Note that uninitialised structures are not allowed. The class name is manifest to the compiler, that is, class names are not values but denotations in the same way as procedure names in most languages. The type of the value returned by a structure creation is "ptr{ x }" where "x" is the class name. This is the only way to create pointer values; we will see later how values acquire a restricted class set of more than one member.

Type Checking on Stores.

In a completely compile-time type checked language where pointer types also include a set of classes, the compiler must check that a stored pointer value matches the type of the location. This means that the class set of the value must equal or be a subset of the location class set. For example, if "b" is of type "ptr{ x, y }" and "a" is of type "ptr{ x, y, z }" then the assignment "a := b" is allowed but "b := a" is not. This is because "a" might contain a pointer to a "z" instance and would violate the restriction "{ x, y }".

Creation of pointer types restricted to more than one class.

A structure creation results in a pointer type restricted to the single specified class. Multiple class pointer types are obtained implicitly by expressions involving choice, or explicitly by declaration of initialised variables or constants such as procedure formal parameters or structure field declarations.

Pointer types resulting from choice constructs.

Although *h* also has a "case" construct, let us consider an example of an "if" expression resulting in a pointer type.

```

if ... then box{ 3, 3 }
else if ... then triangle{ 3, 3, 3 }
else circle{ 3 }

```

There are two nested "if" expressions each with two arms. Let us consider the second "if" expression. Its first arm produces a value of type "ptr{ triangle }". Its second produces one of type "ptr{ circle }". The type of the whole expression is "ptr{ triangle, circle }". Where a construct involving choice gives a pointer value on its arms, the resulting pointer type of that construct is the union of the class sets of the arms. Thus the first "if" expression above returns a value of type "ptr{ box, triangle, circle }".

Accessing of structure fields.

In some languages whether they support class restriction or class freedom, a run time check takes place on the access to check for a nil pointer and also that the value points to an instance of the class defining the field name used. For those languages which restrict pointer types to a single class, no such class check need be made since the compiler guarantees that the value will only point to instances of a particular class.

The h compiler only allows a field access where the pointer type is restricted to a single class and the field name is defined by that class. For example, "p{ height } := 6". If "height" is a "box" field, this is only allowed if "p" is of type "ptr{ box }". No run time checks are therefore necessary other than that "p" is not nil.

Since h also allows pointer restriction to more than one class it must have some mechanism for "filtering" out unwanted classes to guarantee a restriction to a single desired class. If "v" is of type "ptr{ box, circle, triangle }" and we wish to access the "height" field of "v", we must ensure "v" points at a "box" instance and satisfy the compiler by

restricting its type.

Restriction refinement.

A typical "case" statement has an evaluated expression, several labelled arms and a possible default arm. A similar construct in h performs the refinement of pointer class sets. It is related more specifically to the "case" and union modes of Algol 68. [Wijn75] However this construct is proposed as a limited form for use in languages with pointers and structures but not unions involving other types. (A similar discussion of these may be found in Berry and Schwartz. [Berr79]) The syntax in h of the construct is given in the metalanguage proposed by Wirth. [Wirt77]

```
testclause = "test" clause "is"
             is-arm ";" { is-arm ";" }
             "isnt" restrict .

is-arm = name { "," name } restrict .

restrict = ( "use" name "in" | ":" ) clause .

e.g.
test p{ left } is
  box use pbox in begin ... pbox{ height } ... end ;
  circle, triangle use pct in get.height( pct ) ;
  line : output( "line has no height" ) ;
  isnt : output( "bad structure class" )
```

The clause after "test" must be of a pointer type. The clauses on the arms must all be of the same type. (As with other choice constructs, the arm clauses, if all pointer types, need not have the same class sets.) The arm labels are structure class names. The arm itself has two formats. The "use" format performs refinement, the other does not. With the latter, the effect is the same as a normal "case" except the selection is made on the class of a pointer value not the value itself. The arm containing the name of the selecting value class is executed. Each class name may appear once only in the labels of the "test" clause. No class name which is not in the set of the selecting value may appear.

In the "use" format, the name after "use" is a new constant of type pointer which is restricted to the classes in the label list of the arm. Its scope is the arm. It is initialised with the selecting value. Let us call this an arm constant. In the above example, "pbox" is of type "ptr{ box }" and "pct" is of type "ptr{ circle, triangle }". Those arm constants restricted to a single class may then be used to access fields of instances of that class - without a run time class check. Should none of the arms be executed then the last, the "isnt" arm, is executed as a default. If the default arm has an arm constant then this has a pointer type with a class set which is the selecting value class set minus the arm classes. The compiler does not allow an empty class set.

Comparison with another approach.

Algol W also allows pointer type restriction but not complete freedom. To access a field, at compile time the Algol W compiler checks that the field name belongs to a member of the class set of the pointer value. At run time a check is made that the class of the instance is the same as the class of the field. The user may also explicitly check that a pointer value refers to an instance of a specific class by means of the "is" operator. This takes a pointer value and a class name. It returns true or false depending on whether or not the pointer refers to an instance of that class. A fuller example is now given comparing the h approach with that of Algol W.

```

! h !
let structure valu be { vi : integer };
let structure unary be
{
  unrator : char ;
  unrand : ptr{ unary, binary, valu }
} ;
let structure binary be
{
  birator : char ;
  birandl, birandr : ptr{ unary, binary, valu }
} ;
...
test tree is
  unary use tu in
    evalunary( tu{ unrator },
               eval( tu{ unrand } ) ) ;
  binary use tb in
    evalbinary( tb{ birator },
                eval( tb{ birandl } ),
                eval( tb{ birandr } ) ) ;
  valu use tv in tv{ vi }
isnt : error( "bad structure" )

comment Algol W ;
record valu ( integer vi ) ;
record unary
(
  string( 1 ) unrator ;
  reference( unary, binary, valu ) unrand
) ;
record binary
(
  string( 1 ) birator ;
  reference( unary, binary, valu ) birandl, birandr
) ;
...
if tree is unary then
  evalunary( unrator( tree ),
             eval( unrand( tree ) ) )
else
  if tree is binary then
    evalbinary( birator( tree ),
                eval( birandl( tree ) ),
                eval( birandr( tree ) ) )
  else
    if tree is valu then vi( tree )
    else
      error( "bad record" )

```

In the examples, "tree" is assumed to be an appropriately typed pointer variable. The "eval..." are integer returning functions, as is "error".

No run time class checks are performed on the `h` field accesses. The only run time work involved is switching on the class of the instance pointed at by `"tree"`. In the Algol W example, every field access involves a run time check that the field and instance classes match. There is also the run time overhead of the `"is"` operations. The `"test"` clause is a more readable construct in the same way as `"case"` constructs are an improvement on nested `"if"`s.

Summary of structures and pointers.

The advantages of pointer types restricted to a single structure class is that no run time check on field accessing is needed to ensure that the instance class and the field class are the same. However these single class pointer types can be restrictive. It is desirable to allow a pointer to refer to instances of more than one class. The compiler checks that pointer variables or constants are limited to containing values of the specified classes when a value is assigned to them. Languages having unrestricted pointers or multiple classes impose upon their users run time class checks on field accesses. This may be avoided by the introduction of a construct allowing refinement of pointer class sets to ones with single members, in the manner of the Algol 68 union modes. [Wijn75] It has the advantage of reducing the amount of run time overhead and making the structure of the program more readable.

Procedure Values.

We consider the introduction of procedures as values to an Algol our second major feature of the language. One of the major differences between `h` and other Algols is the status of procedures. We first consider procedures in Algol-like languages and define some terms. Strachey [Stra67] states

"A procedure, ..., may only appear in another procedure call either as the operator ... or as one of the actual parameters. There are

no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in Algol are second class citizens - they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter) ... nor can we write a type procedure (Algol's nearest approach to a function) with a result which itself is a procedure."

He advocates the raising of the status of procedures and says

"... I found, both from personal experience and from talking to others, that it is remarkably difficult to stop looking on functions as second class objects. This is particularly unfortunate as many of the more interesting developments of programming and programming languages come from the unrestricted use of functions, and in particular of functions which have functions as a result. As usual with new or unfamiliar ways of looking at things, it is harder for the teachers to change their habits than it is for their pupils to follow them. The difficulty is considerably greater in the case of practical programmers for whom an abstract concept such as a function has little reality until they can clothe it with a representation and so understand what it is they are dealing with."

A number of languages have adopted the idea of first class procedures. We may divide them into three groups for the purposes of this thesis, namely the Algols, the functional programming languages and the experimental languages. The prime use of first class functions is in the functional languages which eliminate assignment and the store from the language. In our classification we include languages with assignment and store since their use is primarily in a truly functional manner. Languages

falling into this category are LISP, [McCa62] PAL, [Evan68] and SASL. [Turn79] Experimental languages are Gedanken [Reyn70] and ELI. [Wegb74] We regard Algol 68, [Wijn75] Euler, [Wirt66] Oregano [Berr71] and CPL [Barr63] as Algols as having first class procedures in some form.

In languages where procedures are not first class citizens, a procedure is named in a declaration. Procedures may be nested within each other; the scope rules determining which variables and constants are accessible from a procedure. The following example may be of help.

<pre> proc x { int xvar proc y ! declaring proc for "z" { int yvar proc z(proc par) { int zvar : par ! call proc param ! } z(y) } } y : } </pre>	<pre> non-local environment for "z" </pre>
--	--

Let us define the innermost procedure (or main program) containing a declaration of a procedure P to be the declaring or creating procedure for P. Furthermore, let us define the variables and constants accessible to P in outer procedures (or the main program), to be P's non local environment. With ordinary procedures, this environment is built up dynamically through the calls of its outer procedures (that is, those within which it is nested), culminating in a call of the declaring procedure. In fact several calls of the same routine may be pending at any one time, each with its own different environment conforming to the static layout of the declarations in the program text. [Rand64] When a procedure is passed as a parameter, its non local environment is still in existence. (This is ensured by the scope rules.) This environment breaks up and disappears on the return from the declaring and outer procedures.

A lengthier explanation of procedures as values is given in. [Weiz68]. We attempt a briefer description here. In h, the above applies to procedure values except that they are anonymous assignable entities. That is, they are expressible values which can be stored, passed as parameters and returned as results just like any other type as demanded by the principle of type completeness. The value of a procedure may be thought of as its body of code and its non local environment as described above. h employs static binding - the meaning of an identifier is determined from the text of the program surrounding its use according to the scope rules. A procedure value comes into existence at the point in the program where its code is found. Note that this occurs each time control reaches this point so that it is possible to have several procedure values in existence each having the same body part but with distinct environments conforming to the same static template. For example,

```

let constant make.adder <-
    procedure ( constant v : integer
                -> proc( integer -> integer ) )
        procedure ( t : integer -> integer )
            t + v

let add1 <- make.adder( 1 ) ; let x <- add1( 5 )
let sub1 <- make.adder( -1 ) ; x := sub1( x )

```

Here we have a procedure value which takes an integer parameter "v". (Calling of procedures in h is by value). It is actually a function and initialises the constant location "make.adder". It returns, when called, another procedure value created on the call. On the first call of the value in "make.adder" the procedure value returned has, as part of its environment, "v" initialised to 1. The result of calling "make.adder" the first time is assigned to location "add1". This value in "add1" is called with parameter "t" initialised to 5. Its result is "t + v" which evaluates to 6. Now "make.adder" is called again but this time the environment of the resulting procedure differs in that "v" is not the same one as was

created on the previous call (although the outer environment is the same). This "v" contains -1 so when the value in "sub1" is called, "t" contains 6 and v contains -1. Thus we see that even though the outermost procedure value had returned, unlike ordinary procedures its contribution to the environment is retained and does not disappear. This is because "v" is needed on the call of the innermost procedure value.

Because of the anonymity of procedure values, the concept of recursion can no longer be treated as a static concept. An example of such dynamic recursion is as follows,

```

let p1 <- procedure (->) begin ... end
let p2 <- procedure (->) if ok do p1()
! a call of p2 would not be recursive !
p1 := p2 ; ok := true
! a call of p2 would now recurse ad infinitum !

```

With ordinary procedures we may see by examining the static text, which procedures refer to each other. The scope rules of h, say that a name is not known until after its initialising clause. (This eliminates let x <- x problems). This means however that we cannot write,

```

let fac <- procedure( i : integer -> integer )
    if i > 1 then fac( i - 1 ) * i else 1

```

since the name "fac" is not known until after the initialising procedure value. Not only that, but its type is unknown until that time as well. Our solution to this is to introduce a "forward" clause which introduces the name and the type of any yet to be initialised procedure value. It must precede the initialisation in the same sequence. For the factorial function above it would be,

```

let forward fac be proc( integer -> integer )

```

Should "fac" be called before its initialisation then a run time error would occur.

Sequences, Declarations and Clauses.

A sequence is a series of declarations and clauses. It is made into a clause by enclosing it in "begin", "end" or "(", ")". A program is a sequence. Clauses are the main means of carrying out the algorithmic processes in an h program. They provide the means whereby actions can be repeatedly or selectively executed and values produced. The clauses making up a sequence may not return a value, that is they must be of type void, except for the clause terminating the sequence which may be of any type including void. The type of the sequence is the type of the clause which terminates the sequence. Declarations always return void so if a declaration terminates the sequence then the sequence is of type void. A sequence is the unit of scope in an h program.

Flow of Control.

In addition to supporting normal sequential execution as embodied by sequences, h supports choice, repetition and procedure calling.

Variable and Constant Locations. +

Our third major feature in h is the ability to dynamically initialise a location on its creation and to disallow updating of that location. A location may be created and named in a declaration. The declaration implicitly specifies what type of value the location may contain by means of an initialising value. There are two forms of such a declaration, an in-line declaration appearing in a sequence, and a formal parameter declaration appearing in a procedure value heading.

e.g. let count \leftarrow next(sum)

The initialising value may be any non-void clause, its type being known to the compiler because of the complete compile time type checking. By this

+ The material on constancy has been published in [Gunn79]

means the language excludes "uninitialised variable" errors.

For a procedure value formal parameter the initialising value is supplied on each call thus the type of the value must be present in the heading.

e.g. let add <- procedure (i, j : integer -> integer)
 i + j

It had been noted [Gunn79] that sometimes locations purposely retain the same initial value throughout their lifetime. These are known as constants in h and are declared like all ordinary variable locations except the declaration is qualified by the reserved word "constant". A constant location is created and initialised like a variable, but cannot be subsequently updated. Strachey [Stra67] originated this idea stating

"Constancy is an attribute of the L-value, and is moreover an invariant property. Thus when we create a new L-value, ..., we must decide whether it is variable or constant."

This contrasts with what are termed constants in other languages. These typically are values manifest to the compiler which may have a name bound to them. They are static in that this binding takes place at compile time. The constants in h are dynamic because the initialising value is evaluated at run time and may be different on different incarnations of the constant. For example in Pascal we may say,

const length = size - 52 /* size must also be const */

"length" is always bound to the same value. In h, we do not have this restriction. For example,

let i <- 1
while i < 10 do
begin ...

```
let constant c <- i * i
```

```
...
```

```
end
```

Each time round the loop "c" is initialised by a different value. It may not subsequently be assigned to within its scope. This eliminates the possibility which would exist if "c" was a variable, of erroneously assigning to it. Constancy may be used wherever variable locations are used, such as procedure formal parameters, structure fields and vectors.

Constancy may also be applied to vector elements and structure fields. Vectors of constant locations, being assignable values cannot have the constancy checked at compile time, therefore some overhead of checking is needed on assignment at run time to vector elements. For structures the constancy can be checked at compile time because field names (and thus their attributes) are manifest to the compiler.

An Example.

To finish this brief description of h we give a short example of how abstract data types may be supported by the language. An abstract data type may be regarded [Lisk74] as a collection of operations made available to the user, while the object operated on remains protected. This is possible in h by having a function which represents the abstract data type. When called this function returns the operations. The representation of the abstract object is made part of the environment of these operations but is not passed out but protected so that only the operations may access it. We give an example of a stack.

```
let structure STACK be
  { constant POP : proc( -> integer ) ;
    constant PUSH : proc( integer -> )
  } ;

let constant stack.instance <-
  procedure( constant limit : integer -> ptr{ STACK } )
  begin
    let constant stackint <- vector 1 :: limit val 0 ;
```

```

let stackptr <- 0 ;

STACK{ procedure( -> integer )
  begin
    stackptr := stackptr - 1 ;
    if stackptr < 0 do
      begin
        stackptr := 0 ;
        error( "STACK Underflow.'n" )
      end ;
    stackint[ stackptr + 1 ]
  end ;

  procedure( constant item : integer -> )
    begin
      stackptr := stackptr + 1 ;
      if stackptr > limit do
        begin
          stackptr := limit ;
          error( "STACK Overflow.'n" )
        end ;
      stackint[ stackptr ] := item
    end
  }
end ;

let constant opstack <- stack.instance( 10 ) &
  constant popop <- opstack{ POP } &
  constant pushop <- opstack{ PUSH } ;

pushop( 1 ) ;
pushop( 76 ) ;

```

There are five outer level declarations. The first describes a structure class "STACK" which, when created contains two constant locations. One is for a procedure value which takes no parameters and returns an integer. The other takes an integer parameter and does not return a result. The second declaration initialises "stack.instance" with a procedure which takes an integer "limit" and returns a pointer to a "STACK" structure. When called, the body of this procedure creates a vector of the required size, which is the stack, initialises a top of stack index and returns a pointer to a structure. This structure contains the two stack maintenance procedures.

The third declaration, that of "opstack" is initialised with a pointer to such a "STACK" structure as described above. The remaining two declarations initialise constant procedure locations with the fields of the

structure.

The points worthy of note in this example are that a different stack is created for each call of "stack.instance", and that the only way to access such a stack is by means of the procedures returned in the "STACK" structure. The stack itself and its top of stack index are not available for use except by these procedures.

CHAPTER 5

The Implementation of h.

We describe three aspects of the implementation of h on the PDP 11/40 running under the UNIX operating system. These are the compiler, the intermediate code and the implementation of procedure values. Other aspects of the implementation have either been covered in previous chapters or are unimportant in the context of this thesis.

The h compiler.

The compiler was written with a view to portability. It produces an intermediate code suitable for a code generator as described in chapter 3. We briefly outline its structure and operation since the compiler plays an important part in the implementation of a programming language.

The compiler is single pass, using the technique of recursive descent [Davi81] for parsing the source. It consists of the phases of lexical analysis, syntax analysis, context sensitive checking and code generation. These were layered on top of each other in the manner described for Pascal. [Jens74] It also employs a simple and reasonably efficient error reporting and recovery technique. [Turn77] Initially the compiler was written in Algol R; however on its completion it was rewritten in h as a large working example of the language. The language proved to be perfectly adequate for the task.

The Intermediate Code.

This was designed using the lessons learned in the implementation of Algol R but was free of the restriction of having to convert an already existing low-level abstract machine code. It still had to convey the information content of the source but differed somewhat from the Algol R intermediate code due to the language differences and the lack of this

constraint.

The intermediate code and code generator are not described here. We concentrate on those aspects which differ from the Algol R implementation. The intermediate code is given in full in Appendix B. Also included in that appendix is the relationship between the source and the intermediate code generated.

The compiler produces three files which describe the source information necessary for the code generator. These are,

- (1) The size file. This contains three numbers, the number of variables and constants, the number of structure definitions and the number of procedure segments. Each of these entities (structures, procedures etc.) is uniquely identified in the other two files by a number in the defined range.
- (2) The data file. This contains descriptions of all variables, constants and structures declared in the program. The information is structured according to the nesting of procedure values in the source. This is to retain environment information. Each of these entities is implicitly numbered according to its relative occurrence in the file. At the end of the data file are descriptions of all the literals used in the source and the procedure types.
- (3) The code file. This contains the intermediate form of the code. This is very simple and could be described as a kind of Reverse Polish for expressions together with an extremely simple and regular syntax for control structures. Code is output in segments, one for each procedure value (the last for the main program). Segments are not nested. Each segment has an implicit number.

This separation of code and data makes the layout of the code generator slightly better than previously with Algol R, but has the disadvantage that all the data descriptions are held throughout the code

generation. In the Algol R implementation, the space required to hold data descriptors depended solely on the static nesting depth of the source. On reflection we think perhaps the Algol R method of combining data and code, although perhaps less aesthetically pleasing, is more efficient from a code generator point of view.

The PDP11 implementation.

Having designed h and written a compiler producing an intermediate code, the next step was to implement the language on an actual machine. In particular certain features of the language not in Algol R had to be investigated, especially procedure values. Initial investigations showed that it would not be feasible to generate in-line code for a number of h's constructs on the machine available. This was so because of its 64K addressability. It was especially true of first class procedures and so it was decided to generate code for an abstract machine supported by an interpreter. This had the advantage that effort could be concentrated on areas not already covered by the Algol R implementation (thus avoiding duplication of work). In fact, the intermediate code and interpreted abstract machine code were very similar so that the code generator was simple and straightforward to write.

For a great many of the intermediate code instructions there is a one to one relationship between them and the abstract machine instructions. We will not describe the interpreted abstract machine in full. It is stack based for the evaluation of expressions and employs a heap storage allocator and garbage collector. We concentrate entirely on the implementation of procedure values and begin by briefly describing the storage structure necessary in the abstract machine for supporting h. This structure would have been necessary even if in-line code had been generated. General purpose registers do not form part of the abstract machine, it being a stack machine - in any case their treatment would not

have differed from that in the Algol R implementation.

Organisation of Store.

In the abstract machine for h, store is not necessarily considered to be one contiguous, continuous sequence of storage locations. Store is divided into several areas. The areas are :-

Code Area

Here the abstract machine code is located.

Literal Pool

This is an area which contains all the literal values which cannot be made immediate (contained in instructions as operands). At the head of the Literal Pool are the Procedure and Structure Tables containing static information about each procedure body and kind of structure in the program. These will be discussed later.

Heap This is an area in which space is allocated by explicit requests and reclaimed when a request cannot be fulfilled. Only space which is no longer accessible from the entities in existence is reclaimed. If the request still cannot be fulfilled the program terminates (abnormally). Frames, structures, displays, vectors and strings are kept on the heap. They are called heap objects and have space within themselves for storage reclamation information. At any point in the execution of the program, the first frame and last frame in the dynamic chain are known and are referred to from special purpose registers.

In the execution of a program, values arise as a result of expression evaluation. These are stored on one of three stacks each of which contains values of a certain class and organisation. Two of these stacks contain values which contain pointers into the heap, and are searched when marking heap objects during garbage collection.

Pointer Stack

This is a stack which contains temporary values of entities which point into the heap (pointers, strings, vectors). That is they contain the addresses of heap objects.

Procedure Stack

This stack contains temporary procedure values. It is implemented as a separate stack for the same reason as we have a separate pointer stack. This allows procedure values to be separately identified when garbage collecting. A procedure value contains a pointer part and a non pointer part, thus could not go on the pointer stack.

Main Stack

This stack contains temporary values of entities which are not eligible for placement on the other stacks (booleans, characters, files, integers, enumerateds).

The implementation of routine values.

We now consider in detail the implementation of routine values, first in general then specifically their implementation in h. By the term "routine" we mean either a procedure or a function, that is, the parameterised abstraction of a statement or an expression respectively. We will see that problems arise in the implementation because when a routine is assigned around in the same manner as say an integer, a straightforward implementation based on a stack is not possible. We emphasise that the binding of identifiers in h is static. This requires a different implementation from that employed by languages such as LISP [McCa62] which have dynamic binding.

Routine Values.

The implementation of routines as assignable entities is best approached by considering what we mean by a routine value. Again we refer

to Weizenbaum [Weiz68] for a more detailed explanation of the nature and implementation problems of procedure values. Traditionally, routines are considered to be control structures not data. In fact they are a hybrid control and data structure. As a control structure, routines are bodies of code which may be called from points in the program. There is however, a data structure implicit in the call. Each routine on a call requires housekeeping and local variable space. Let us assume for simplicity that this data space is a block of storage, called a frame, allocated on the call which disappears on the return since it no longer is required. Such space could of course be allocated on an individual basis for each variable.

Routines may call other routines in a manner controlled by the static embedding of their declarations and the language's scope rules. We obtain a frame for each pending routine call. These make up the dynamic chain. The most recently called routine may access data in those frames of pending calls of statically surrounding routines and its own local frame. The static chain is where the local and non-local variables and constants of the routine reside. The most recent frame need only remain in existence until its owning routine returns. Because of the static nature of routine declarations, a routine may only be called when its surrounding routines have been called, that is when their frames are in existence. Thus, a routine may only be called when it is guaranteed to have a complete non-local environment. Because of the LIFO nature of calls and frame allocation/deallocation, a stack is traditionally used to implement routines, the frame of the most recently called routine being allocated on top of the stack. This topic is well treated in the literature [Rand64] and we will not elaborate it here.

A routine value is therefore both a control structure (a callable body of code) and a data structure (its non-local environment). Taken together these two have been called a closure. [Land64] The environment is

augmented on a call of a routine by the local data frame. A routine value closure then can only come into existence each time its declaration is met, that is, when its non-local environment has been built up. This is done by a history of calls terminating in a call of the deepest routine body (statically) containing that declaration. If certain limiting conditions of use are met, (non-assignable procedures) such a value may be called as normal (even passed in as a parameter to another) and the implementation may then make use of a stack. Morrison [Morr77] has shown how such closures may be simply created, called and passed as parameters in a stack based implementation.

Assignable Routine Values.

Having considered the nature of a routine value, we have in h given it full "civil rights". It may therefore be created and freely stored like any other value, complex in nature or simple. However, it is the ability to store such a value with complete freedom which causes implementation problems. In the traditional Algols where routine values are not freely assignable, a routine value comes into existence at its declaration. It then remains in existence only as long as its creator's frame is in existence. Even if it is passed as a parameter, the routine using that parameter has to return, losing the value, before the creator routine returns. Thus a routine value in such languages has a lifetime shorter than the time its creator takes to execute. It is this fact which allows a stack based implementation of frame allocation.

However when a routine value may be freely stored as in h, its lifetime may exceed that of its creator's execution time. For example, a routine value created on a call of a deeply nested routine may be assigned to an outer level variable. Despite the return of those calls which had culminated in its creation, the value still exists, living in the outer level variable.


```
e.g.
let p := ...
... proc{ ... proc { ... p := proc { ...
```

But what of the frames making up its non-local environment? If we had a stack based implementation then they would be long gone, having been popped from the stack on the returns of their owners. This presents a dilemma, because a routine value cannot be called unless its environment exists (otherwise where would the free variables be?). Some method of retention [John71, Weiz68] of its environment is therefore essential. This can be achieved in a straightforward manner, borrowing from the stack implementation, the display or the static chain mechanism, or for singly allocated variables, the environment list of the SECD machine. [Land64] Frames now must be allocated from a heap, where retention is possible. The natural process of garbage collection of a heap can mark those frames forming part of a routine value which is accessible in the program at that point in its execution. Anything not marked is made available for re-use.

To get round this problem of environments of procedures, languages are sometimes restricted. One restriction is to allow procedure values to access any free variable in the frame for the outer block and local variables in its own frame since these frames are always accessible. [Rich69, Gunn80, Turn76] Space for a frame can then be allocated in a LIFO manner in exactly the same way as when procedures were not values. Another restriction is not to allow assignment of a procedure value to objects which may live longer than its creator's frame (Algol-68). That is, it can only be called while the frames in its environment are on the static chain. Furthermore, on exit from a call, the frame corresponding to that call (the head of the dynamic chain) can be thrown away giving the same LIFO implementation as when procedures were not values.

Our simple implementation although sufficient, can be optimised to save space retained in environments. One of the advantages of a stack is

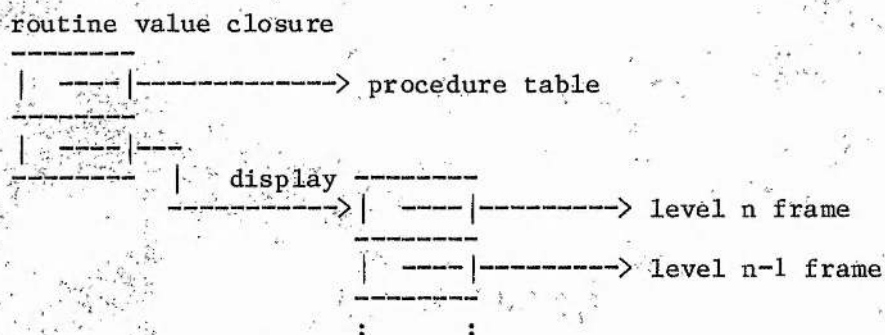
that the frames are dynamic in length, only being as big as they need be at each point in the program. We can also superimpose expression evaluation on the same stack. Unfortunately in our implementation, frames must be of fixed size and have a unique cell for every local variable (including the parameters). Whereas with a stack, variables could be allocated only as necessary and share the same space.

```
if ... then
begin let x <- 0 ; ... end
else
begin let y <- 0 ; ... end
```

In the above example, "x" and "y" could share the same space on a stack based implementation. With a heap implementation of a language supporting routine values this may not be possible. In the example assume the ellipsis code of each branch created a routine value respectively accessing "x" and "y" globally. The frame containing "x" and "y" would form part of the environment of each of these routine values. "x" and "y" must therefore exist independently in the frame since they may each be accessed by the routine values created in their respective branches.

This procedure value in the abstract machine for 'h' is a pair consisting of

- (1) the address of a table which contains information relevant to the execution of the procedure such as the start address of the procedure segment,
- (2) a pointer to an object called a display which contains the addresses of the frames comprising the procedure value's environment.



Variable and Constant Addressing

Each variable and constant, whether it be local to a routine or not must be individually addressable. The compiler allocates for each one a unique position (for reasons given above) in its local frame. Since each routine body is at some static level of nesting, the code generator knows statically the location of all variables or constants accessed by the routine. These are addressed by pairs made up of a lexicographic routine level difference and a position within the frame. The difference in routine level between a use of a non-local variable and its declaration in a surrounding routine, is used at run time as an index into the current display to access the corresponding frame. The level difference for local variables is of course zero. This is similar to the corresponding situation in a stack based implementation of a traditional Algol. [Rand64]

Contents of Procedure Tables.

The use of tables is a device to save space and eliminate duplication of information and its associated movement between copies. The space which the information would have occupied is now reduced to a reference to a table. Each procedure table contains,

- (1) Size of procedure frame.
- (2) Frame layout information (that is, where the variables of each type are stored).

- (3) Procedure segment start address.
- (4) Information relevant to the optimising of frame space.

Procedure Value Assignment.

This is done by copying the closure value.

Procedure Calling.

Calling involves four stages.

- (1) Which procedure to call? This is simply established by an appropriate procedure value pushed on the closure stack.
- (2) Obtain local data space for the procedure. This is creation of a frame. The details needed are obtained from the procedure table whose address is in the first part of the closure on top of the procedure value stack. Fill in the relevant house-keeping information such as the dynamic link, display pointer and procedure segment start address.
- (3) Evaluate the parameters to the procedure in the current environment and assign them to their respective positions in the newly created frame.
- (4) Transfer control to the procedure, saving the return address in the frame and make the change to the new environment by making the new frame the 'current' frame.

(2)-(4) are handled by the intermediate code instruction "call". The code generator produces an equivalent abstract machine instruction which is interpreted.

Procedure Value Creation.

A closure is created by the "makeproc" instruction whose argument is the number of the procedure table corresponding to the procedure body. This is an abstract machine instruction which the code generator has produced from an equivalent intermediate instruction. The table is

accessed and its address is made the first part of the closure. The second part is a pointer to a display. The display is a block of pointers, one for each outer procedure value. The non local environment of the created procedure value is simply the non local environment of the creating procedure (that is, the one containing the makeproc instruction) plus the frame of the creating procedure. This frame is on top of the dynamic chain. The non local environment is held in the housekeeping area of the creator's frame. Thus the display is one longer than that of the creating routine.

Optimisation.

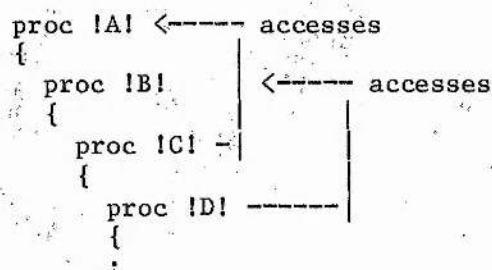
We reconsidered the above scheme, which, while having simplicity in its favour, suffered from the inefficiency that significant amounts of space could be used. It could be optimised to save some space. What we intended was to eliminate the retention of unneeded variable space in environments. Davie's approach [Davi79] does away with procedure frame allocation of space and allocates space for each block. A block descriptor consists of a pair of references, one to a block frame containing the variables of that block, the other to a frame containing references to the non local variables of outer blocks. On block entry space is allocated for these and the references filled in. This ultimately means that only the variables needed are retained, not whole frames although "a viable garbage collector ... will have to be moderately sophisticated". It seems to be an attractive solution but has the space overhead of additional housekeeping information for the garbage collector. It has the time overhead of allocation and garbage collection, for a greater number of allocated units and operations. With cheap store now available the space problem is not so critical as it might have been. Bobrow [Bobr73] puts forward a proposal for a stack implementation. A stack is more efficient in terms of space allocation/deallocation when this takes place at the top of the stack. Bobrow's proposal however involves copying of frames on the stack and is

based on complex data structures. We feel that any advantage gained by using a stack in that manner is outweighed by the time overheads involved in copying frames. His method is extremely difficult to follow.

We agree with the spirit of these space optimisations but adopt the attitude that a straightforward, easy to implement partial solution is a good compromise. Our scheme only keeps those frames which are actually needed. We feel that this is acceptable for the following reasons.

- (1) Store is always becoming cheaper and larger amounts are available on machines.
- (2) Programmers are encouraged to keep procedures short. If procedures are say 30 lines long at most then they are unlikely to have many local variables. (an interesting statistic from [Broo82] is that a third of variables in Pascal programs are declared at the outer level.)
- (3) The implementation should not be designed with "worst case" in mind, that is, monolithic procedures with many variables, especially if this means a solution which is more complex or burdensome in resources than a solution for the "typical" case.

For a particular procedure value we may view its non local environment as a collection of frames, one for each procedure within which it is nested. On execution of the body of code for this procedure value certain locations in this collection will be accessed. If a particular frame in this environment is never accessed then it need not be referred to in the display. That is, its corresponding display entry could be the nil pointer. The frame would then be garbage collected. For example, let A,B,C,D be procedure values nested within each other, D being the deepest.



If C's code does not access any variables declared in B, then by the above reasoning a pointer to B's frame perhaps need not appear in C's display when procedure value C is created. However if D does access B's variables, a pointer to B's frame must be part of D's display. But, D's display is made up from C's local frame plus C's display, thus C's display must contain a reference to B's frame, even though it does not access it itself.

Similarly, A's display (let us assume it is not at the outer level) must contain all those frames referred to by itself, B, C and D. If no entry was created in this closure's display for each global frame accessed by these nested procedure values, then when these values came into existence their environment could not be completed.

Now we may generalise and formulate some rules about display creation in a makeproc instruction. The display created for a procedure value P is formed from the display of the creating value C plus possibly the frame of the creating value. This latter is only needed if P or nested procedures within it access C's local variables. P's display uses only those frames in C's display which it or procedures nested within it, need to access. Thus,

P's display = some or all of C's display +
possibly C's frame

The following important condition is arrived at. A procedure's frame must be part of every display created for procedure values nested within that procedure, upto the most deeply nested procedure which accesses local data within that frame.

Display Creation.

On procedure creation, making up the new display part of the value, there are two actions to be taken with regard to frame pointers.

- (1) possibly add the creator's frame pointer to the new display and
- (2) copy certain frame pointers from the old display to the new.

So for each procedure value at creation time we need to know, (A) whether to put the creator's frame into the new display, and (B) which frames in the creator's non local environment to copy into the new.

At compile time this static information must be built up for each procedure value. The compiler keeps information about each procedure value. (A) above is a boolean flag, the copy creator frame flag. One flag is needed for each level of nesting. (B) is a boolean vector, the copy display entry flags. One vector is kept for each level of nesting.

```

proc
{
  let x <- 0      ! level D !
  proc
  {
    ... proc
    { .. x .. ! level U !

```

Thus when the compiler looks up a variable, used at procedure level U, declared at procedure level D it does,

```

copy.creator.frame( D + 1 ) := true ;
! level D frame is needed !
for l := D + 2 to U do
  copy.display.entry( l, D ) := true
  ! D frame must be retained up to level U !

```

The information is put with other relevant data in the procedure table.

Summary.

We believe that h brings together features usefully employed in a programming language. For example strings have been limited entities implemented as character arrays. Some languages employing user defined

structures claim to be completely compile time type checked. In h this is actually the case having the advantage of making the program more readable and efficient. The procedure can successfully be an assignable value and add to the power and expressivity of a language. Its implementation can be very straightforward, space probably being a limiting factor. However we have shown that a simple optimisation can reduce this problem. We have introduced the idea of a constant location which may be initialised at run time and not subsequently updated. This is a useful addition to a language since it is the case that many "variables" are not updated.

CHAPTER 6

The Polymorphic Programming Language ns1.

h had primarily investigated the incorporation and implementation of first class routines in a general purpose programming language. One further area of languages and their implementation which we felt deserved further investigation was that of type polymorphism. That is, the ability of programs to process values dynamically according to the type of the value. The polymorphism refers mainly to the ability of locations to contain values of more than one type but also to a minor extent to the ability of operators to take operands of differing types.

The ns1 programming language.

A process control language called PROTOCOL [Har181] had been developed which employed polymorphism and it was decided to develop a new language based on h, developing the polymorphism of PROTOCOL. This new language, called ns1, also attempts to rationalise and improve the data structuring facilities in h. ns1 was designed in collaboration with D.M.Harland, the latter also writing the compiler for it. The abstract machine implementation of ns1 is presented as the final part of this thesis. We stress again that the language design was embarked upon as a vehicle for investigation of implementation techniques and is of minor relevance. An overview of ns1 is given in this chapter, concentrating on those features which distinguish it from its predecessors. It is very similar to h in its control structure therefore we will illustrate the language by means of example except where novel features or constructs different from h are employed. The control structure uses sequences, declarations, choice and repetition constructs similar to h.

Polymorphism.

Most of today's programming languages force the user to rigidly specify the types of value which a location may contain. Other languages impose no checking on the programmer, treating a value as a bit-pattern to be interpreted according to the operation in which it is employed. We feel that there is a middle course in languages where the user at some times wishes to be restricted in the types he manipulates and at other times wishes to be free. Whichever the case, he still must be prevented from nonsensical operations such as adding a logical to a function. Milner [Miln78] states

"A widely employed style of programming ... entails defining procedures which work well on objects of a wide variety ... flexibility is almost essential in this style of programming"

We refer to the ability of a location to hold values of more than one type as polymorphism. It has also been interpreted to mean the ability of procedures to take parameters of different types on different calls. Strachey [Stra67] considered polymorphism in two ways, ad hoc and parametric.

ad hoc

Here there is no single systematic way of determining the type of the result of an operation from the type of the arguments. He gives all the ordinary arithmetic operators and functions as examples stating "there may be several rules of limited extent ... but these are themselves ad hoc both in scope and content". We interpret this to mean polymorphism in its most general form.

Parametric

This is a more limited form of polymorphism with regular rules. Strachey illustrates it by example. If we have a function

$f : \alpha \Rightarrow \beta$

which maps values of type α into values of type β and a list of values of type α

$l : \alpha\text{-list}$

then the application of the function map,

$\text{map}(f, l) \rightarrow \beta\text{-list}$

gives a β -list. Now this is the case whatever the types α and β . Thus we may say that the type of map is

$\text{map} : (\alpha \Rightarrow \beta, \alpha\text{-list}) \Rightarrow \beta\text{-list}$

The type of the function contains named parameters. Its polymorphism is of a simple parametric kind, α and β could be regarded as type-variables standing for the actual types on a call. This is a restricted form of the more general polymorphism because it does not handle the case where say the list is made up of values of different type. Some work [Miln78, Burs80, Barb80] has been done on parametric polymorphism and we do not consider it further except where it arises in the discussion of our polymorphism.

Strachey goes on to say "Polymorphism of both classes presents a considerable challenge to the language designer but it is not one which we shall take up here". We consider ad hoc polymorphism by allowing types to become first class citizens of a language. This approach has also been adopted by EL1 ("The inclusion of modes among the legitimate values in a language allows modes to be computed, providing a very powerful definitional capability" [Wegb74]) and the purely functional language of Barbuti and Martelli. [Barb80]

Primitive types.

The types of ns1 may be split into two, the primitive types and the data structures, for discussion purposes. All values, of whatever type in

`ns1`, have the same rights. The primitive types are, integer, character, logical, user defined enumerated ⁺, procedure, function and type. The data structures supported are lists, vectors and user defined structures. These are discussed in a later section. Note that characters have not disappeared as was suggested in the `h` section. This is because strings can be simulated, not by vectors, but by lists of characters as in SASL. [Turn79] Also procedures and function types are not qualified by their parameter and result types as in `h`. This simplifies the language considerably without reducing its power. This is also the approach taken by the polymorphic language SASL. [Turn79] The introduction of type "type" is a simple way of allowing polymorphism so that the type of a value may be determined dynamically and processed accordingly. Thus we have elevated type to first class citizenship. With the introduction of type "type" comes operators for dealing with such values. For example, there is the monadic operator "typeof" which takes any value and returns a value of type "type". We may then switch on this type value to an appropriate piece of code to handle values of that type. This is similar to what happens during pointer type restriction in `h`, but is not a special case and is treated uniformly by existing choice constructs. We do not need to introduce an analogue of the `h` "test" nor do we need a plethora of predicate functions (`isint`, `ischar` etc.). For example, consider a polymorphic procedure in `ns1` which "prettyprints" values. A list must be printed enclosed in brackets and its elements comma-separated.

⁺ a discussion of user defined enumerated types in `ns1` may be seen in [Har182]

```

let pretty const :=
  procedure ( v )
  begin
    case typeof v of
      int : ...
      char : ...
      list :
        begin
          out( '(' )
          if ~ nil v do
            repeat
              begin
                pretty( hd v )
                v := tl v
              end
            until nil v do
              out( ',' )
            end
          out( ')' )
        end
      ...
    end
  end

```

As will be seen, the implementation supports type "type" in a very straightforward manner.

Polymorphic Variables and Constants.+

Polymorphism extends to locations. In ns1, every location, whether it be in a data structure or in-line, has attributes affecting its constancy and the type of the values it may contain. We have defined these attributes to be value constancy and type constancy respectively. We give the scheme below using in-line declarations as examples, but stress that the attributes belong to all locations. In keeping with h, all locations are initialised on creation, this being reflected in the declaration syntax.

We define a location (also referred to as a cell) which is non-updateable after initialisation to be value-constant. This is like the constant in h. We define a cell which is restricted to containing values of a specified type to be type-constant. We see that four kinds of cell may be declared in ns1 depending on the type and degree of constancy.

+ the material on polymorphic variables and constants has been published in [Gunn81]

These are,

- (1) A general purpose cell with no attributes of constancy.

e.g. let c := exp()

"c" may be updated, by values of any type. It is a polymorphic variable. It is the variable of GEDANKEN [Reyn70] and Euler. [Wirt66]

- (2) A type-constant cell.

e.g. let c int := exp()

The cell is an integer variable which must be initialised and updated with only integer values. It is the variable of such languages as Pascal. In fact, the declaration is more powerful than at first appears. The type which appears after the "let" is a type literal but may be any type-valued expression. This is illustrated by the (slightly contrived) example -

let c (if i = 6 then typeof q else int) := exp()

Depending on the value of "i", "c" will be allowed to contain only integers or values the same type as that in "q".

- (3) A value-constant cell.

e.g. let c const := exp()

The cell may be initialised by a value of any type, but once initialised cannot be subsequently updated. This is the dynamic constant of h, except that the type of the initialising value may not be known until run time.

- (4) A type- and value-constant cell.

e.g. let c const int := exp()

This cell may only be initialised by a value of the type specified (either statically or dynamically) and subsequently can not be updated.

It may be seen that our scheme differs from other approaches to polymorphism, for example the union of Algol 68. [Wijn75] We have adopted the approach that a cell may be restricted to a single type or be free to contain any type. This contrasts with the idea of cells being restricted to certain subsets of types (unions). We think our scheme is simpler and does not lose expressivity through its generality.

Type Checking.

At this point it is perhaps worth examining the static and dynamic nature of our polymorphism with respect to type checking. We demonstrate our method of type-checking with some examples. First we introduce the type "any". "Any" is our compile time notion of polymorphism, that is, it is the type of a value whose type is not known until run time. We use a type rule associated with a syntactic construct to define how types are treated. These rules determine the types required in and produced by such a construct. We give a rule for an nsl "if" expression whose syntax is similar to that for h.

```
"if" { boolean } "then" T1 "else" T2
    => if T1 = T2 then T1 else { any }
```

"T1" and "T2" may be regarded as type variables. The rule states that the clause after the "if" must be of type boolean, since this may not be known at compile time, the compiler checks that it is of type any. If it is a type other than these a compile time error is given. If the type is any, then a run time error is given if the type turns out not to be boolean when the expression is evaluated. The result type is T where both arms are of type T, or any where they differ. Our scheme detects type errors at compile time where it can, otherwise at run time. For example,

```
let t char := if a or b then
                'y'
                else
                'n'
```

The compiler can tell that this produces a character at run time. If "a"

and "b" are known to be booleans then there will be no run time type check at all. Furthermore, the compiler need not generate a run-time check that the value produced before "then" is boolean because the use of "or" will ensure this.

We believe our approach is simpler although not as powerfully compile time type checked as others. [Barb80,Miln78,Deme80] In these approaches the above "if" clause would be illegal if the types of each arm were different. Checking at run or compile time is a design tradeoff. The authors mention above do not have any run-time type checking since they wish type errors to be detected as early as possible. "a further advantage of static type checking is ... computational efficiency, since run time checks are no longer necessary". [Barb80] In return for this seeming advantage they acknowledge that their polymorphism has to be restricted.

"it is very important to be able to perform type checking at compile time, even if this entails some restrictions on the type structure" [Barb80]

"our view of types precludes run-time type-checking. Thus we have been led to devise language restrictions" [Deme80]

"everything concerning types is done at compile time ... although it does impose constraints on the use of types" [Miln78]

We, like the authors of EL1, [Wegb74] do not restrict the programmer to only performing actions type checkable at compile time only. Dynamic checks are imposed where insufficient information is available at compile time to deduce and check types. We refer to Strachey [Stra67] who says

"The decision in CPL to make types a manifest property of expressions was a deliberate one of language design ... The

opposite extreme is also worth examining. ... This scheme of dynamic type determination may seem to involve a great deal of extra work at run time, and it is true that in most existing computers it would slow down programs considerably. However the design of central processing units is not immutable and logical hardware of the sort required to do a limited form of type determination is relatively cheap. We should not reject a system which is logically satisfactory merely because today's computers are unsuitable for it."

We also feel that such a scheme should be considered, then perhaps in the light of experience using it, lessons may be learnt about its usefulness.

Data Structuring in nsl.

nsl provides three kinds of data structure each of which has its own particular characteristics but all of which are treated in a uniform manner in other areas. They are vectors, user defined structures and lists. The former two are descendants of h's data structures, the latter is an addition to the language. The list has been implicitly used in a restricted form in other languages. Its worth has been recognised and exploited in nsl. In addition to being able to access data structures in the "usual" way (e.g. integer subscripts, fields) we also allow the elements of any data structure to be accessed by its (integer) position. Normal access is called selection to distinguish it from positional access. There is a syntactic differentiation between these kinds of access.

e.g. p{ fl } - selection

p[| 6 |] - position

Vectors.

Vectors consist of polymorphic variable locations (i.e. neither type- nor value-constant) selected by integers within certain bounds.

Lists.

Lists consist of a series of polymorphic values (i.e. elements are not updateable since they are not locations). Although lists are characterised by head and tail operations in the manner of SASL [Turn79] we also allow selection by integer in the range one to the length of the list.

Structures.

User defined structures differ from h in that

- (1) they add to the types of the language
- (2) they may contain type- and value-constant locations. Type constancy must be manifest to the compiler.
- (3) field selectors are values of a new type with all the rights and privileges of other values.

Consider the following example of a structure definition.

```
structure tree ( element ; left, right tree )
```

This introduces two new types, tree (a pointer to a tree structure) and fld.tree (field of tree). Each tree structure has three fields, two of which are type constant and must be tree's. Five literals are introduced.

```
tree, fld.tree of type type
```

```
element, left, right of type fld.tree
```

Note we prefer that new literals share a syntax and scope with names. [Har182] Also type values are underlined by the compiler. Thus we may write expressions such as,

```

let fixity const fld.tree :=
  if postfix then right else left ;
tree.ptr{ fixity } := ...

```

Here a field selector initialises a type- and value-constant location. We know of no other Algol which allows field selectors of user defined structures to be first class citizens.

Null Data Structures.

Each data structure type has a single null value of that type, that is there is a null vector, a null list and a null for all of the user structures. The section on enumerated initialisation shows how they may be written.

Creation of Data Structures.

Creation and initialisation of data structures is by one of two constructs common to all types of data structures. These specify what kind of data structure is to be created and provide initialising values for each element. Vectors need additionally to have their bounds specified. A missing lower bound is taken to be one. Because the creation constructs may be used to build lists and user structures, if bounds are specified for these, the lower bound must be one. The upper bound must be the number of fields for a user structure or else is interpreted as the length of a list. These constructs are,

Enumeration

The provision of an individually calculated initialising value for each element in the data structure. This construct needs a list of initialising expressions and a value of type "type" which must be a data structure type. A lower bound may optionally be specified. Should the initialising list be empty then this is taken as the single null value of the data structure.


```
e.g. vector at m [ a, a + a, a * a ]
( if x then list else vector ) [ m, q, 1 + 2 ]
tree[ 6, tree[ 4, tree[], tree[] ], tree[] ]
list [ 's', 't', 'r', 'i', 'n', 'g' ]
```

Lists of characters have a convenient syntax. The above example may be written "string".

```
list []
vector []
tree []
```

These are the null data structures for the corresponding types.

Replication

Each element is initialised using a single expression. There are two forms. The first is where the expression is evaluated once only, its result initialises every element of the data structure. In the second, the initialising expression is re-evaluated for each element. In both the lower bound may optionally be specified as above after "at" and the upper bound is optionally evaluated after "size" or "upto". Examples are now given of the first kind.

```
vector at m upto n value a + b
list size 20 value ' '
tree size 3 value tree[]
```

Examples of the second kind are,

```
vector at 1 upto n eval f()
let i := 1 ; let j := k
h := list size x eval begin i := i * j -> i end
```

In the latter example "h" is assigned a list of the powers of j. A variation of this second form of replication allows the position of the element to take part in the initialising expression.

e.g. list size 5 with k eval k * k.

This produces the list of squares 1, 4, 9, 16, 25. For each element integer constant "k" is initialised with its position.

e.g. v := vector at m upto n with p eval p + m - 1

For this latter vector "v", $v\{i\} = i$, $i = m, n$.

There are a variety of operators on data structures such as,

upb, lwb - upper and lower bounds of a vector

sizeof - size of a data structure

nil - a predicate testing for a null data structure

hd, tl, append, prefix, join - for lists.

Lists may also be "sublisted", that is, a list may be extracted from another.

In ns1, general purpose routines may be written to handle all the data structures in a program. As an example we give a function value which returns true if all values in an arbitrary data structure are of the same type.

```

let issametype :=
  function( ds )
    begin
      let top const int := sizeof ds
      let ty const type := typeof ds{ 1 }
      let pos int := 1
      let same bool := true
      while pos < top and same do
        begin
          pos := pos + 1
          same := ty = typeof ds{ pos }
        end
      -> same
    end

```

The function takes a parameter which is any data structure (we have omitted checks for it being a data structure type for clarity.). The constant "top" is initialised with its size and "ty" with the type of its first element. "pos" is a counter which is used to determine which

position in the data structure we use in our check for type. The loop consists of stepping up the data structure comparing the type of each element with that of the first until we reach the end or one is found which differs. The value of "same" is returned as the result of the function. Note that "issametype" may be applied to any data structure.

For example,

```
issametype( list[ 1,2,3,4 ] ) gives true
structure itree( e1 : int ; l, r : itree )
:
let p := itree( 6, p1, p2 )
issametype( p ) gives false
```

Routine values in ns1.

These are similar to h in that they are values in their own right. However unlike the myriad of possible procedure types in h because of differing parameter and result types, there are only two in ns1, procedure and function. Values of the latter return a value when called. Since functions are polymorphic and can return any type, the type of the result does not form part of function type. That is, a function may return different types on different calls. Similarly, the parameter type does not form part of the routine type. This is the approach adopted by SASL. [Turn79] Routine values take zero or one parameter, however this parameter may be a data structure if several values are to be passed in. Parameter passing, as in h, is call-by-value. Some examples are now given,

```
procedure ( ) z := z + 1

function( z const int ) y + z

let sum :=
  function( ls const list )
  begin
    let s int := 0
    for p := 1 to sizeof ls do
      s := s + ls{ p }
    -> s
  end
```

Multiple Parameters.

A convenient syntactic form of a list which is an actual parameter is to separate the elements with commas.

e.g. `sum(1,2,3)` is equivalent to
`sum(list[1,2,3])`

However if a single value is passed it is of course not made into a list. On entry to the routine value type checking takes place if the formal parameter is type-constant.

Similar to the actual parameter list syntax sugaring is one for formal parameters. If it is desired to pass in fixed length actual parameter lists, then a familiar syntactic form of a formal parameter list may be given. For example,

`procedure (q ; m const ; t int ; x const char)`

This heading means that the actual parameter value must be passed as a 4-list. Four local cells called "q", "m", "t" and "x" are declared with the attributes stated, each initialised with the corresponding element of the list. Type checking takes place on initialisation. This mechanism we believe allows the programmer complete flexibility or rigidity, whichever he wants.

Type checking is performed on the call if the formal parameter is type constant. A point to note is that the type constancy of the formal parameter may also be calculated on the call. This allows a form of parametric polymorphism. For example,

`let ty type := int
let f := function (par ty) ...
f(6) ; ty := char ; f(7)`

The second call of `f` would fail because the type constancy expression result (the evaluation of "ty") would give character. Another example is,

let fred := function (t type ; v t) ...

The type of "v" is determined by the actual parameter passed to "t".

To complete this section we give an example of an nsl program which provides the stack abstract data type given in the h section.

```
! a polymorphic stack !
let new.stack :=
  function( s const int )
    begin
      let stack const := vector size s value 0
      let sp int := 0
      -> list
      [
        function ! tos !()
          if sp > 0 then stack{ sp } else 0 ,

        procedure ! push !( x )
          begin
            sp := sp + 1
            if sp > s then
              begin end else stack{ sp } := x
          end ,

        procedure ! pop ! ()
          if sp > 0 then sp := sp - 1 else begin end
      ]
    end
  begin
    let tos, push, pop := new.stack( 15 )
    push( 999 )
    let a := tos() + 1
    push( a )
  end
```

A call of "new.stack" takes a limit and returns a list of the stack maintenance routines. Note that there is no need to declare a structure in which to return them, a list is all that is necessary. Such a list is returned in the call above and stripped, its elements initialising three routine variables.

CHAPTER 7

The Tagged Architecture Machine Implementation of ns1.

This section describes the implementation of ns1 from the standpoint of those features rarely found in other languages. These are constancy, polymorphism, user-defined types, orthogonal data structures, and field selectors and routines as assignable values. It examines the characteristics of these features in order to subsequently show how they are implemented. They greatly influence the architecture of the underlying abstract machine which is described. Finally it highlights several important instructions such as those for routine calling. Appendix C additionally describes instructions for data structure management. It also shows how the list as a data type in a language can be exploited.

Main Design Influences.

The machine was designed for the implementation of Algol-like polymorphic programming languages in general and of ns1 in particular. Its instructions therefore reflect the high level operations in such languages, and are not primarily intended to be hand generated. The nature of the machine draws from traditional architecture (especially stack machines for expression evaluation) but is mostly influenced by the characteristics and data space requirements of polymorphism, routine values and data structures. These mainly show their influence in the structure of the store which is not a totally linear store but is segmented in the manner of a heap.

In character, the machine is related to the SECD machine [Land64] however it attempts to minimise some of the inefficiencies inherent in that machine when it is realised on current architectures. The SECD machine implements applicative languages with no assignment whereas ours is for Algol-like languages in which assignment and the store play a major part.

Our tagged architecture model (TAM) is an abstract machine implemented by an interpreter. The compiler generates this code directly because it is radically different from today's architectures and could not realistically be generated in-line by a code generator. The following were major considerations in the store design.

Routine Values

As we have seen for h, a routine closure comprises a table, a display and the body of code. Space for these must be allocated from a heap.

Data Structures.

Broadly speaking, a data structure may be considered to be a collection of locations or values. Again these data structures need to be allocated from a heap.

Run time type- and constancy-checking.

A program written in a polymorphic programming language, by its very nature may not be able to be completely type checked at compile time. This means that any machine supporting that language must be able to check types at run time, that is, types must be identifiable at run time. This is usually performed by some kind of tagging [Ilif68, Myer78, Feus73, Feus72] mechanism where a value carries round an indication of which type it belongs to. The compiler may also not be able to check whether a constant location is being assigned to or not, whether it be type- or value-constant. In this case some run-time check must be performed by the implementation.

The above considerations mainly reflect the aspects of a polymorphic programming language which differentiate it from traditional architectures. Although much of ns1 is traditional it was felt that its implementation should be designed from the position of those aspects considered to be novel. This approach proved encouraging because the traditional aspects of the language fitted easily on top. We thus designed an ideal machine

taking heart from Strachey's comment [Stra67] "We should not reject a system which is logically satisfactory merely because today's computers are unsuitable for it". Myers [Myer78] is of a similar opinion saying

"The similarity of the architectures of today's systems to that of earlier systems can cause us to become complacent about the subject; we look about us and see ... that the architectures of current systems are virtually the same as those of earlier systems. ... As a result, the architecture of future systems remains the same."

Feustel [Feus73] encourages self defining data and says

"all data elements in a computer memory [should] be made to self-identifying by means of a tag ... such a machine architecture may well be a suitable replacement for the traditional von Neumann architecture."

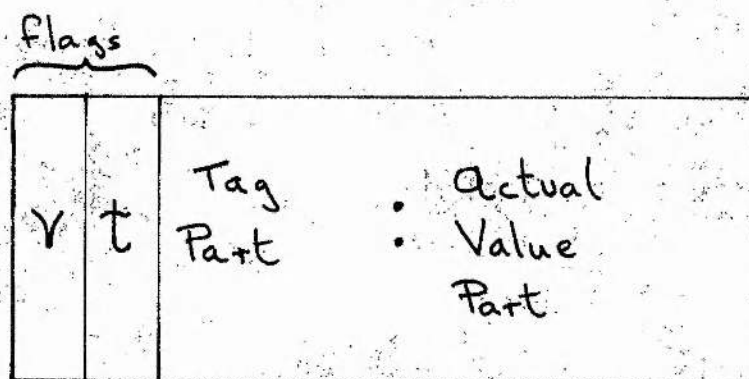
Store Organisation.

Fundamental to the design of the machine is the organisation of its store. With its simple organisation it is found that many high level language features may be supported. The store design was based on the observation that data could be held as collections of locations, each of which needed to be individually accessed. The store is segmented into allocatable, arbitrary length blocks of cells.

Cells.

Cells are all the same size and contain type-tagged values. Each cell is divided into three individually accessible components, two of which are one bit flags, the other containing a value. Figure 1 shows a cell and its components. The two flags are the value-constant (v) flag and the type-constant (t) flag.

Figure 1.

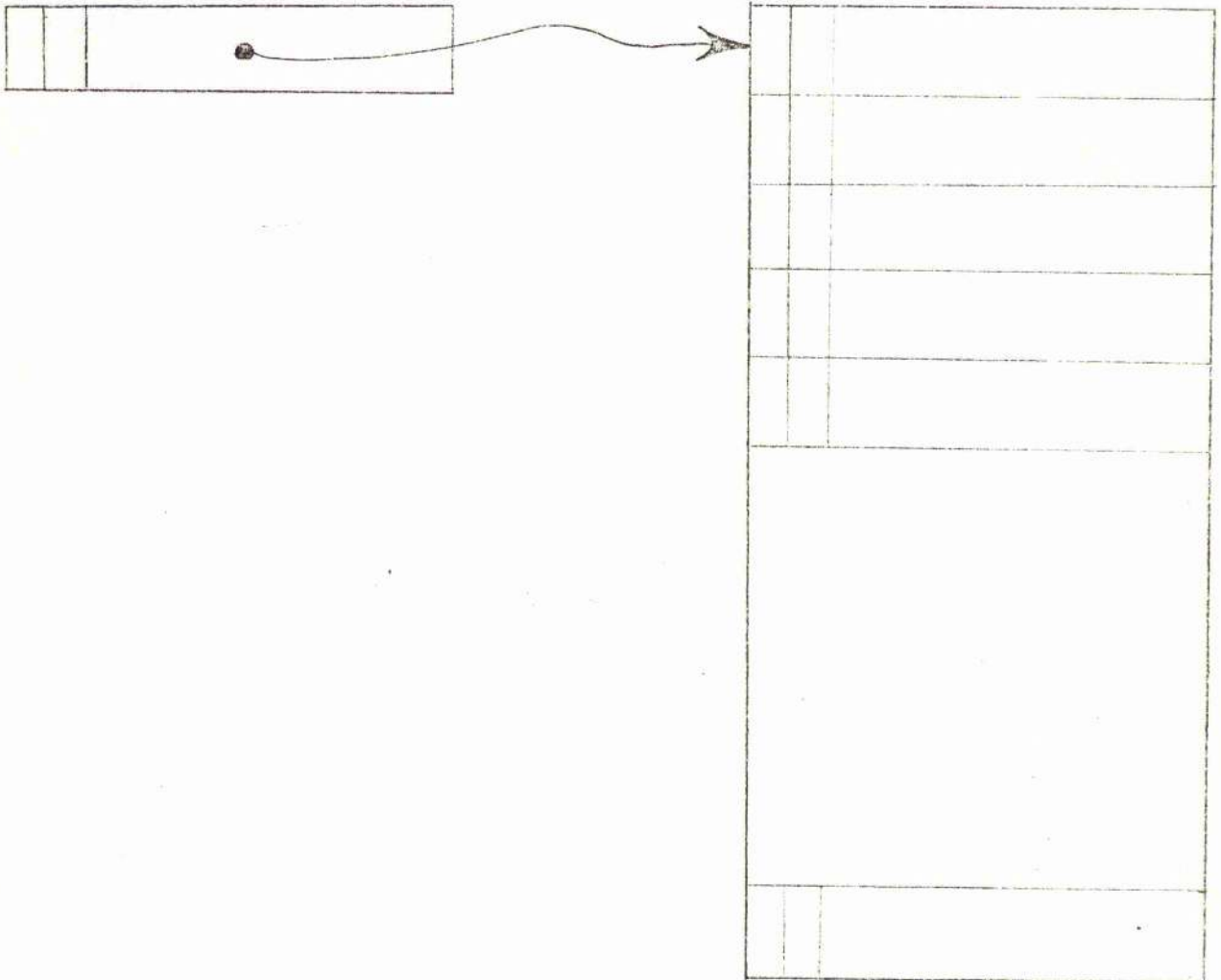


If the value-constant flag is set then the cell value contents may not be updated by certain instructions. If the type-constant flag is set the cell value may only be updated by another value of the same type as that of the value already in the cell. Each value belongs to a single type. This type is carried round as part of the value. All values are the same size. Thus a value is made up of an actual value part and a type tag. We depict them separately in figure 1 but emphasise that they are inseparable.

Blocks.

A block is a linearly ordered sequence of cells. Blocks are accessed using values whose tags must be one of a special subset of the types supported by the machine and whose actual value part is a reference to the block in store. There are machine instructions which result in the creation of a block together with an appropriately typed reference value. This block reference value must be held in some cell. Figure 2 shows a block.

Figure 2.



Blocks disappear (via garbage collection) when no value references them. Cells within a block may only be accessed by specifying a block reference value and a position within the block or a selector value. These two values constitute an address in the TAM machine. There is no way to refer directly to any cell within a block.

Values.

Values are made up of a type tag and an actual value. During certain instructions and operations this type tag may be checked for consistency. In non-tagged machines, the type of a value is determined solely by the

context of its use. A tagged machine however can determine that the type is valid in the context of its use and furthermore can support polymorphic instructions which use that tag to determine the course of action of an instruction. The machine supports a wide range of implicit types, some of which are essential for its own operation, others are the types provided for use by the polymorphic programming language. The user may also define his own types by means of the Structure and Enumerated Tables described in a later section. We differentiate between the non block reference types and the block reference types.

Non-Reference Types.

These are the values whose actual value part is not a block reference, that is the value is totally contained in one cell and is not a reference to a block. Some are straightforward such as "boolean", "character", "integer", "position" and "type". A value of type "position" can be considered to be an unsigned, non-zero integer which is the position of a cell in a block. The first cell is at position one. Other tags are provided for the internal consistency of the machine ; these are "error" and a special type "any". New data types may be defined. This allows the supported language to generate new tags and provide tables to support the new types. For a discussion of these see later.

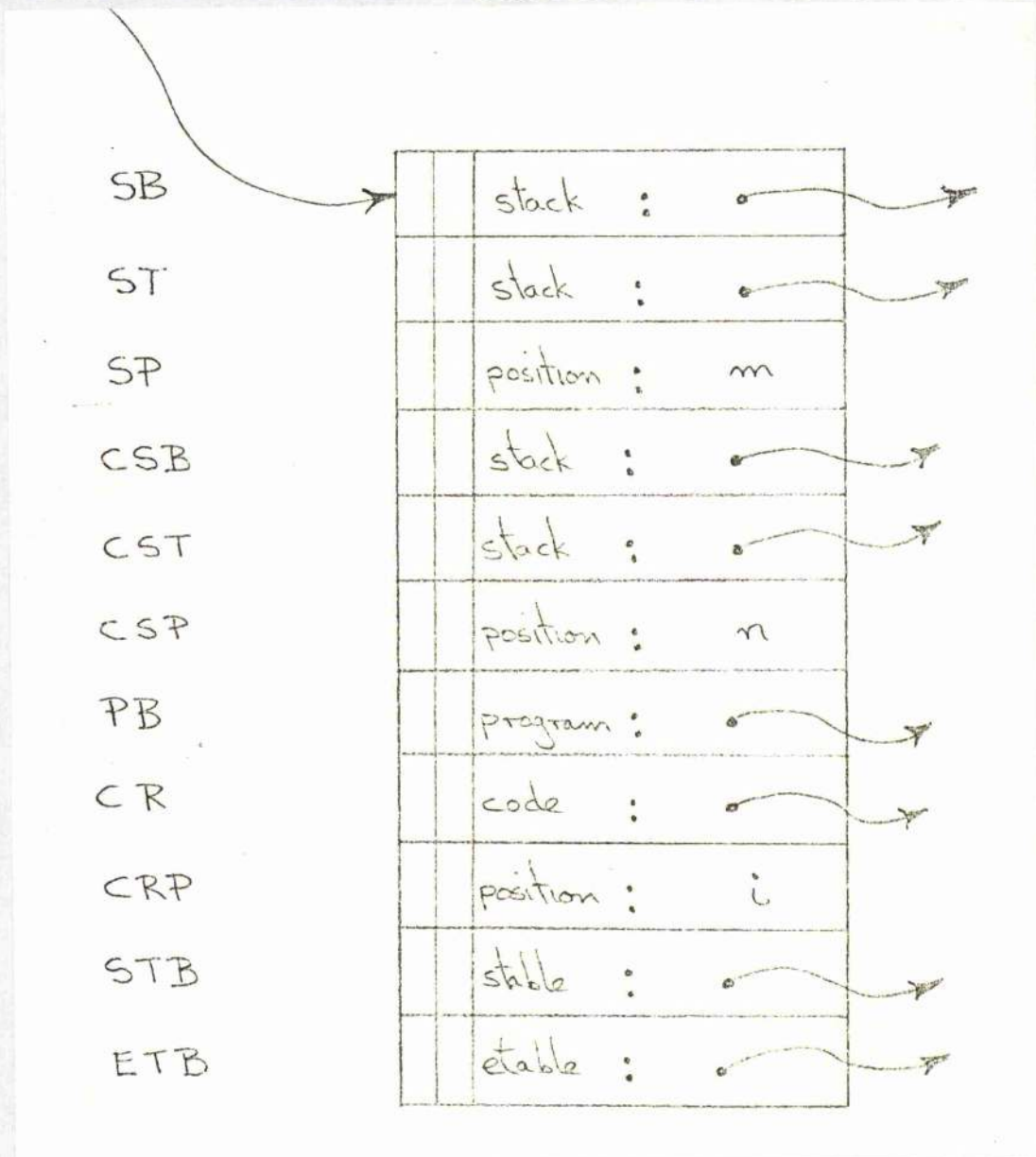
Block References.

As mentioned above, a number of values have an actual value part which is a block reference (depicted as a pointer in the figures). These reference values have type tags reflecting the use for their corresponding blocks. These values are solely internal to the machine in that the user is not aware of them, such as "frame", "stack" and "environment", or they are the implementation of user data structures, such as "list" and "vector". A special actual value, nil, is used where no block is referenced.

Registers.

The program in execution is represented by a single block reference value. This points to a block, the same size for every program, containing values used as implicit operands in certain instructions. That is, each cell within this block has a special function in the operation of the machine. We will call these cells, registers and give them mnemonic names. There are three groups of register, namely, for stack handling, for program operation and for data description. Figure 3 shows the Register Block for a program.

Figure 3.



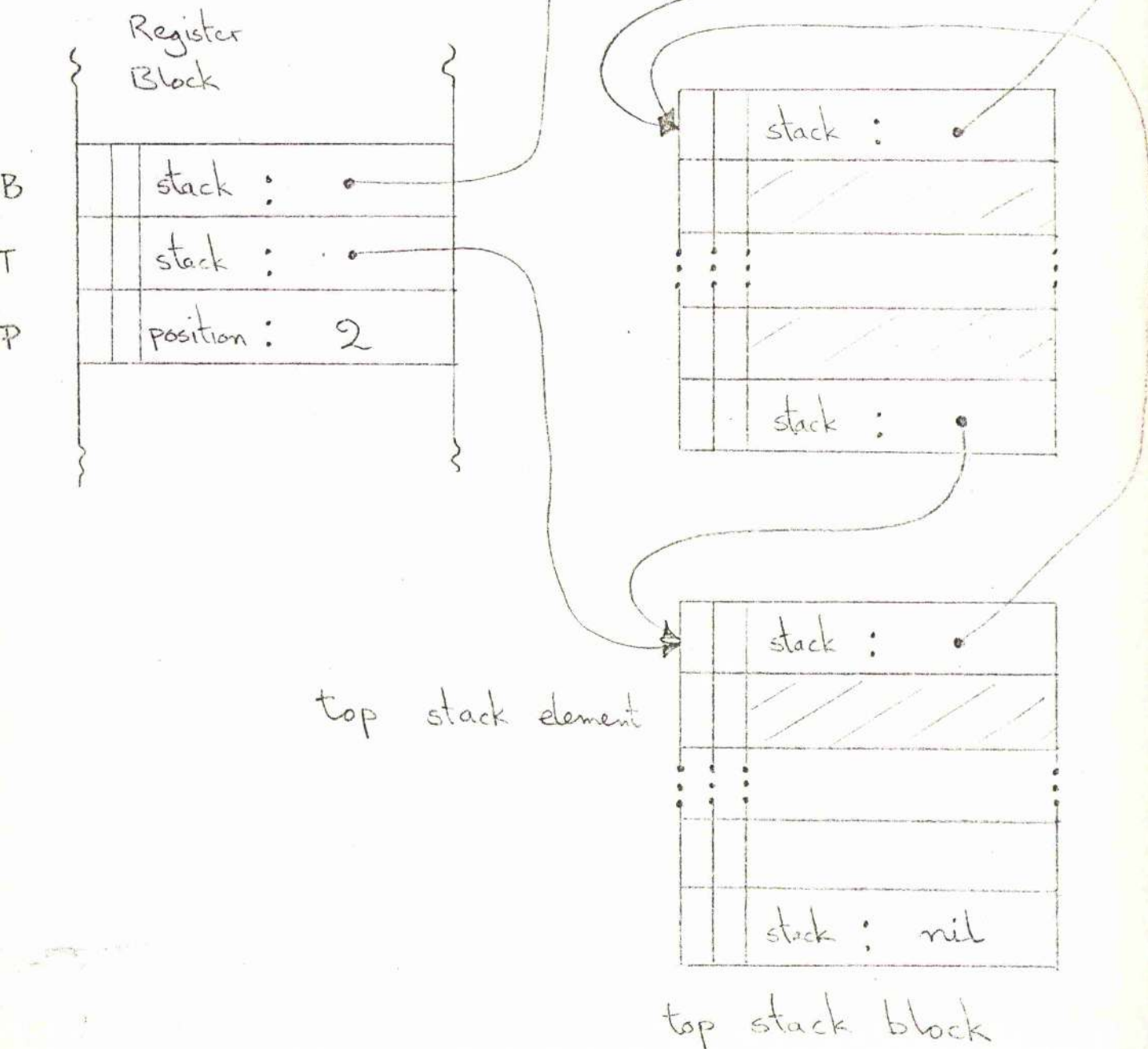
The Stacks.

There are two stacks, one for expression evaluation called the evaluation stack, and one for routine execution housekeeping called the control stack. A stack is a doubly linked list of blocks. Each cell in the blocks (apart from a linkage cell at either end) contains a value on the stack. Each stack is maintained by three registers. These are :-

SB	:	evaluation stack base (type "stack")
ST	:	evaluation stack top ("stack")
SP	:	evaluation stack position ("position")
CSB	:	control stack base ("stack")
CST	:	control stack top ("stack")
CSP	:	control stack position ("position")

The stack base register refers to the lowest block of the stack. The stack top register refers to the highest block. All blocks within a stack are the same size, although the evaluation and control stack block size may be different. All blocks except the top stack block are considered to be full. The stack position register contains the position within the top block of the highest used cell. The lowest cell in each block contains a reference to the previous block. The highest cell in each full block refers to the next block in the stack. Figure 4 shows the evaluation stack.

Figure 4.



When a block is about to overflow, that is, before a push there is only one free cell left in the block, a new block is allocated and the appropriate housekeeping cells and registers are updated. Similarly, when the top block empties, it disappears and the second top block becomes the new top.

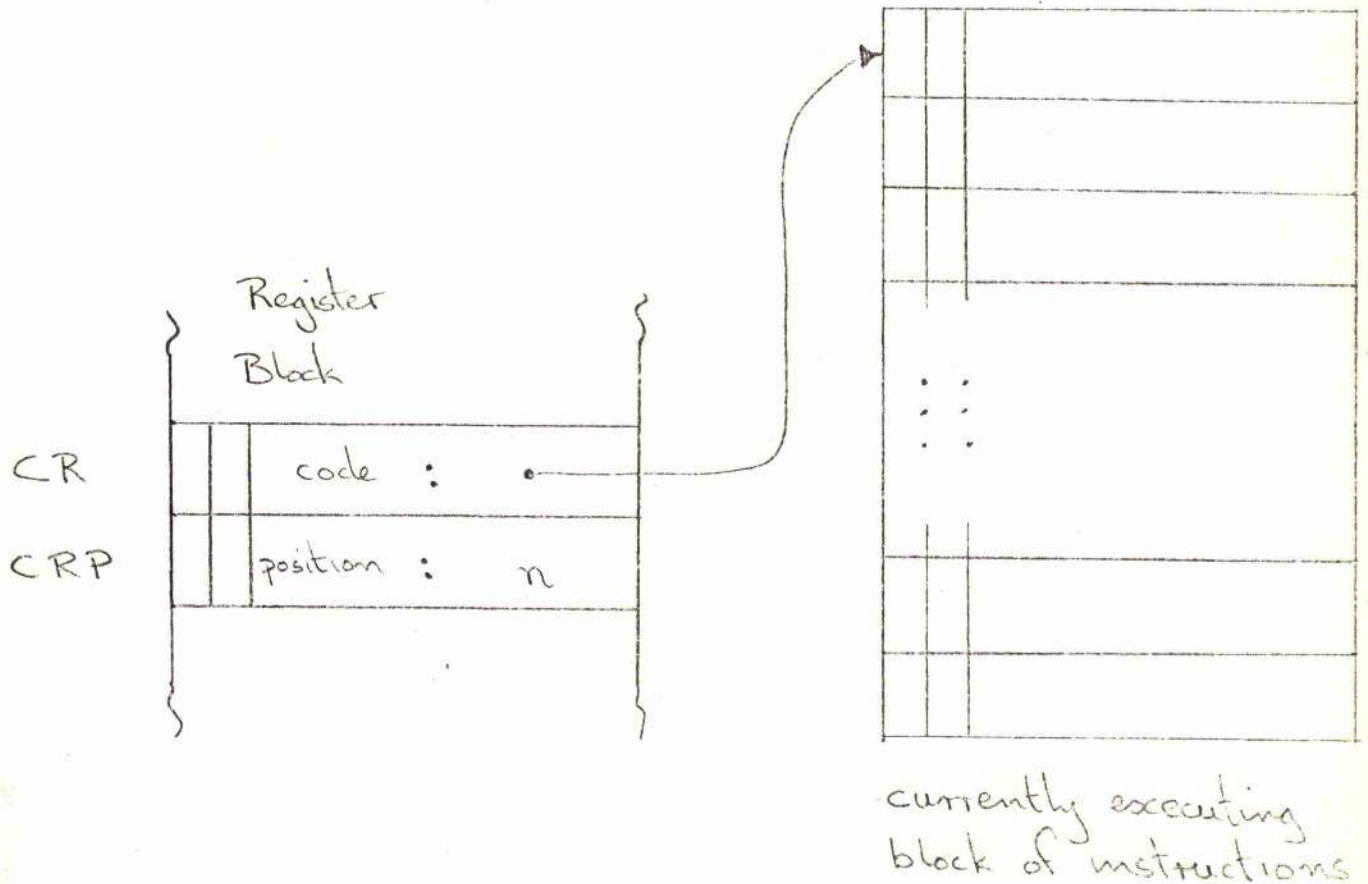
The Program.

The code executed by conventional machines and their data are both seen as bit patterns residing in locations. Code may be treated as data and vice versa since the treatment of the contents of a location is determined by context. That is, if the machine program counter is pointing at it then it is an instruction, whereas if it is to be added to an accumulator then it is integer data. Our machine differs from conventional machines in that its tagging mechanism is exploited to give a high degree of protection. It is also orthogonal in that a body of code is treated as a typed value. In fact, it is treated much like the data structure it may be considered to be. Such a body resides in a block and is accessed by a reference of type "code". The instructions within it are themselves values which may only be executed. It is this alone which differentiates them from other values. There is no way that code in a "code" block can be changed - the cells containing it are made value constant. Nor can it be accessed as data - there are no instructions to treat it in such a manner. Furthermore, the machine will only execute the contents of "code" blocks. A "code" block contains values which are instruction fields. An instruction may extend over several cells. Each code block corresponds to a single source language routine body. The main program is considered to be a routine called at the end of the loading phase.

Orthogonality demands that a program definition in the machine is itself a block whose cell values are references to blocks containing code. This contains as many values as there are routines. This reference is held

in register PB, the program block. Its type tag is "program"; in fact it is the only value of that type during execution. All accesses to routines are performed by means of their position within the program block. At any time, only one routine is being executed. A reference to its code block is held in register CR, the current routine register. The position of the current instruction within this block is held in register CRP, the current routine position (see Figure 5).

Figure 5.



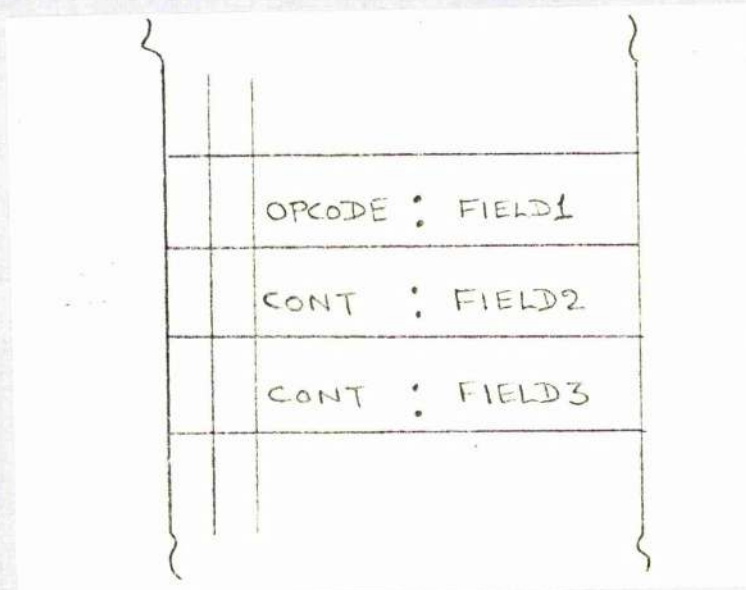
The first cell in the program block is a reference to a code block known as the "boot" segment. On starting execution the machine sets CR to point at the boot segment. This code block is actually just a normal procedure call to one of the other code blocks which is the main body of the program.

Routine bodies are accessed by means of a position in the Program Block, allowing the compiler to generate code using them without knowing where they will lie at run time. This organisation lends itself to a form of segmentation. The code for a routine body could remain on disk until that routine is required for the first time. A nil block reference in its corresponding Program Block cell would indicate that the code is not in store and must be pulled in. Thus routines not called would not take up any space. Although the store seems to be infinite, this is achieved by a garbage collection model on a finite store. Even if this pseudo-segmentation was not extended to include the freeing of hardly used code blocks, the limitations of a finite store and the fitting in of variable sized blocks would still be present.

Instruction Tags.

The two components of a value, the tag and actual value parts, are distinct but inseparable. The machine sees an instruction as a series of values in successive cells. The tag part of the first value of the instruction is the operation code for the instruction. The parameters, if any, for the instruction lie in the actual value part of this and following cells. These other cells each have a tag type "continuation of instruction" (the mnemonic is "CONT"). The machine will not execute this so it cannot start executing "between" operation codes. Figure 6 illustrates a three field instruction.

Figure 6.



Some instructions have literal values following the operation code cell rather than operand fields. These values are the same as those appearing elsewhere in the machine except they are generated directly by the compiler. This means of course that they are manifest to the compiler and not created at run time. That is, all literal values planted in the code are not block references. Figure 7 shows a "load literal" instruction (mnemonic "LDL") where the first operand is a count of the literals following.

Figure 7.

			LDL : 3
			integer : -10
			char : 'A'
			boolean : true

Thus instructions, being made up of sequences of values, are treated in exactly the same way as other data within the machine.

Jumps are simply instructions whose operand is a new position value within the same block of code. The type checking mechanism of the machine does not allow anything except a valid op-code to be executed so it is impossible for bad jumps to take place to the middle of instructions. It is still possible however to jump to the wrong instruction!

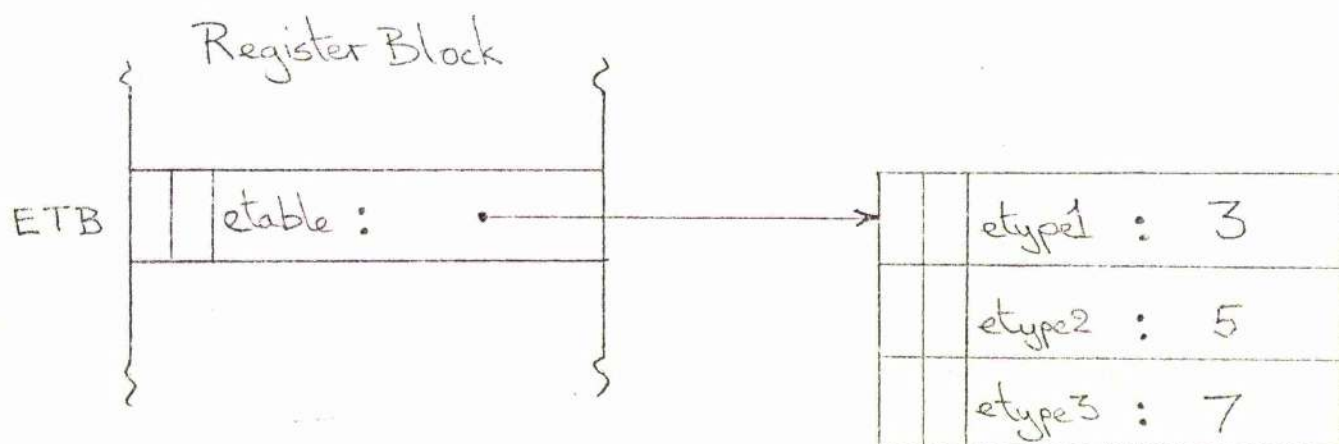
Data Description.

A feature of the TAM machine is that it allows the user to define his own scalar and structure types. These are handled uniformly due to the orthogonality of design and are not special cases. Two tables may be constructed to implement these and ranges of tags are reserved. We will deal with each kind of user definition in turn.

Enumerated Types.

There is a range of tags which defines new scalar types, specifiable by the supported language. These tags are called the enumerated types and are basically similar in character to the enumerated types of Pascal. Associated with them is a block of cells called the Enumerated Table. A reference to it is held in register ETB. The machine treats the actual value of an enumerated type as an unsigned integer. Values of the enumerated types are ordered. These run from one to the user defined maximum. Each cell in the Enumerated Table contains the maximum value of each defined enumerated type. The compiler passes on to the machine how many enumerated types are needed and what their limiting values are. It allocates a new enumerated tag for each enumerated type declared by the user. Figure 8 shows a typical enumerated table for the nsl declarations in the next example.

Figure 8.



```

datatype colour( red, green, blue )
:
datatype symbol( plus, minus, times, divide, power )
:
datatype staff( fred, jim, arthur, john,
                patrick, peter, louis )
:
minus has the value < etype2 : 2 >
staff has the value < type : etype3 >

```


The only allowable operations on enumerated types are relational operations and the addition and subtraction of an integer value. The latter two operations check that the resulting value of the same enumerated type is still within range. The same addition instruction is used for adding an integer to an integer or an enumerated type. This polymorphic add uses the tags to differentiate between them.

Structures.

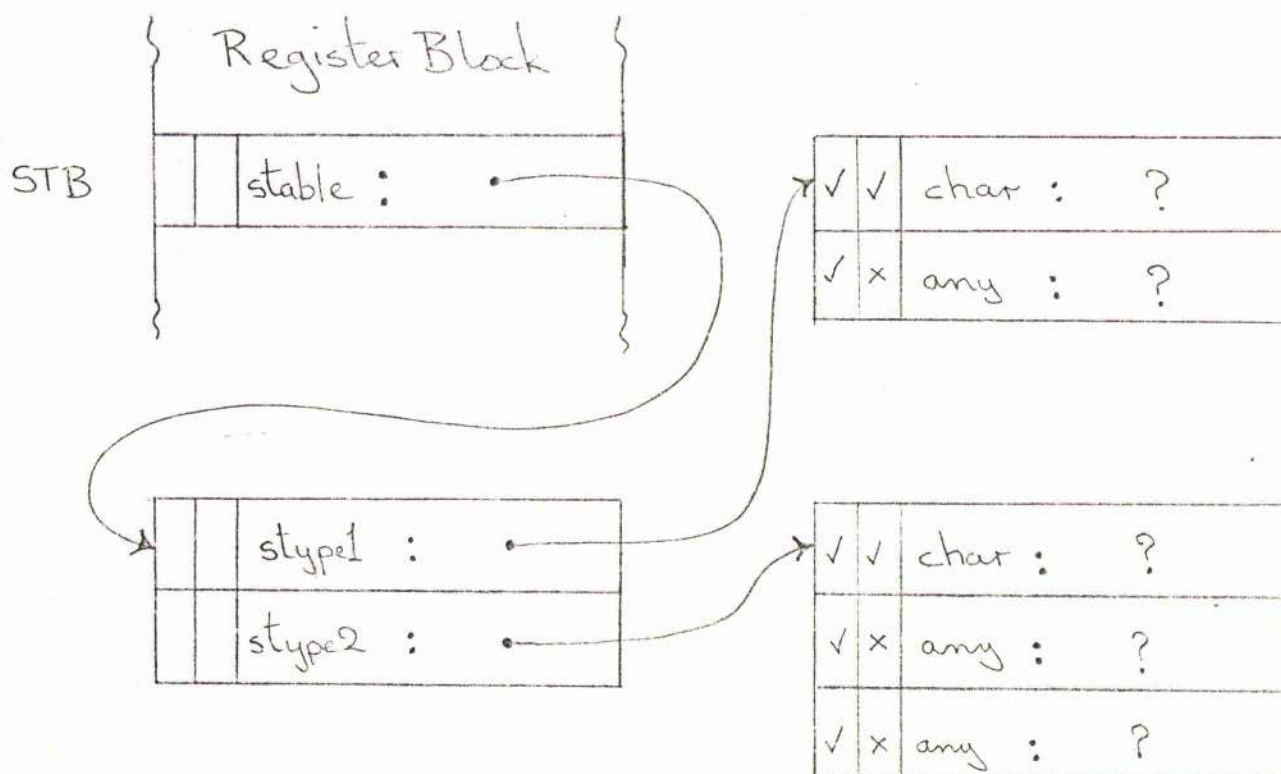
The high level language user defines different classes of structure. In ns1, each class is a value of type "type". It is the type of all references to incarnations of structures belonging to that class. The machine supports a structure type and a field type for each class. Two ranges of tags related to each other are reserved and specifiable by the compiler. One range is the structure tag range. The other is the field tag range. These ranges are of the same length and each field tag is associated with a unique structure tag. A source language structure is implemented as a block of cells and a field is a component cell of a structure. A structure value is a reference to such a created structure. A field value is a position within a structure and is used to select a component field. Like the enumerated types, there is a Structure Table, a reference to which is in register STB. There is one value with a unique structure tag in this table for each structured type (structure class) specified by the user. These each point to different blocks of cells called structure templates. A template is a pattern for creating new structure values of the corresponding type. The flags and the type tags are filled in for each template by the compiler according to the user's definition. The type tag "any" is used if no type constancy was specified, otherwise it will be the specified type. Figure 9 shows a typical structure table for the ns1 declarations the next example.

```

structure unary( urator const char ; urand const )
:
structure binary( brator const char ;
                  brandl, brandr : const )
urand has the value < ftype1 : 2 >
binary has the value < type : stype2 >

```

Figure 9.



A "unary" structure has a block reference as its actual value and stype1 as its tag value.

Routine Implementation.

Here we wish to focus on the implementation of the routine values in ns1 on the TAM machine. We will consider five topics.

- 1) Compile time information.
- 2) Run time information.
- 3) Routine value creation.
- 4) Routine value assignment.
- 5) Routine value calling.

Compile Time Information.

The compiler must provide the information needed to create and call routine values, and to access variables and constants in their environments. The information necessary may be summarised as follows.

Frame Size

This is needed for the allocation of a frame block on a call. It is readily obtained by the compiler on a single pass through the source. It becomes part of a routine value when that value is created.

Variable and Constant Addressing

These are addressed by pairs made up as in h, from a lexicographic routine level difference and a position within the frame.

Routine Bodies

The compiler does not know at run time where the body of code will reside, so it allocates a unique segment number to each routine value. This is used as a position in the Program Table to find the body of code on closure creation. These table entries will be set up dynamically on loading.

Run Time Information.

Call Information

Although frames must be allocated from a heap, the mechanics of routine calling is still LIFO. We may thus make use of a stack to handle the housekeeping of routine calls. This is the control stack.

mentioned above. There are five registers used to implement routine calls.

CST - the control stack top

CSB - the control stack base

CSP - control stack position

CR - the current routine code block

CRP - the current routine position

The first three are the usual stack housekeeping registers similar to the evaluation stack. The control stack consists of a series of triples, three values pushed on it during a routine call. On return from the call these are popped. The control stack therefore behaves like the traditional stack implementation, only here environmental information is retained, it being off stack as part of routine values. In fact the evaluation and control stacks could be combined but we prefer to use a separate stack. CR and CRP determine the current instruction. Together they form the TAM machine's program counter.

Environment Information

A new kind of block associated with routines is an environment block. It is referred to by a block reference of type "environment" and is a block of cells containing values of type "frame". "Environment" blocks hold the non-local environments of routine values, forming part of these values. They also hold the complete environments of the currently executing routine and of the pending calls of routine values. References to these are on the control stack.

Routine Value Creation.

A routine value is created by executing a "make-procedure" or "make-function" instruction (mnemonics "MKP" and "MKF"). These instructions have two operands. The first is the segment number of the corresponding

routine body. The second is the size of the frame to be created to hold the local variables of that routine on its call.

The run time representation of a routine value is a block of three cells ; its code, non-local environment and local frame size. (Its full environment includes its own local frame which does not come into existence until the value is called.) All but the environment may be copied or derived from the instruction operands. The code reference is obtained by using the segment number as a position in the Program Block. The non-local environment of the routine value being created is made up of all the frames currently accessible. This latter environment is already held on top of the control stack as the total environment of the current routine (which of course contains the "make" instruction). As will be seen on calling a routine value, its non-local environment and its local frame are made into a new complete environment which is pushed onto the control stack.

The routine value is a block reference with a type tag "procedure" or "function", referring to the block of three cells. As a result of the make instruction, the value is pushed on the evaluation stack.

Routine Value Assignment.

This may be done freely in the same contexts as other values, for example straight assignment, data structure initialisation, parameter passing and so on. There are no restrictions or special cases. The value is treated the same as any other.

Routine Value Calling.

All routines require a single actual parameter on a call. This simplifies the calling mechanism yet does not reduce the power or flexibility of parameterisation. The nsl compiler will generate a value of type "error" if no source actual parameter is given. This will be examined on routine entry which means that the type checking can detect the lack of

an expected parameter. The parameter must lie on top of the evaluation stack above the routine value to be called. The instruction executed is a "call-procedure" or "call-function" (mnemonics "CLP", "CLF"). This has a code operand which is the number of actual parameters supplied. If this is greater than one then the instruction pulls them off the stack, creates a list from them and pushes the list value. This becomes the actual parameter. The second top value is then checked to verify it is a procedure (for a "call-procedure") or a function (for a "call-function"). The return address is pushed onto the control stack. This is just the contents of CR and CRP.

A new completed environment must be created for the routine. It may access local and intermediate free variables, so the full environment is made by creating a local frame and storing its reference and the non-local environment references supplied as part of the routine value. The non-local environment, as has been seen, is the complete environment of the routine which created the called value. Note this need not be the currently executing routine. The main program has an empty non-local environment. The new (complete) environment is thus one cell larger than the creation (non-local) environment. This completed environment is pushed on the control stack. It determines the frames currently accessible, that is, while the called routine's code is being executed. Those instructions which access cells in frames will make use of it.

The actual parameter is copied into the first cell in the local frame. If the routine was called with no parameters then the compiler will have generated an "error" value as the actual parameter.

The last act of the call is to set CRP to the first position and CR to point at the code block of the routine value. This is in effect a jump to the first instruction of the routine. This first instruction has the simple task of checking the consistency of the actual/formal combination, that is, whether a parameter was expected and one was supplied. The return

instruction is the same for procedures and functions, since a function will have left its result on the evaluation stack. The return simply pops off the environment on the control stack and restores CR and CRP popping them as it does so.

We see, therefore that the implementation of routines as values in an Algol-like language is quite straightforward with a suitable architecture. The TAM machine demonstrates the advantages of the orthogonal treatment of blocks and type tagging.

Other TAM Instructions.

We leave further discussion of the remaining TAM instructions to Appendix C. These cover data structures, accessing of cells and list exploitation.

Summary.

We have presented an architecture which implements a powerful polymorphic programming language. This machine makes use of facilities little seen in the architectures of present day computers and must be implemented by means of an interpreter. We have exploited lists to aid the implementation of parameter passing and multiple assignment. To support routines as values and data structures we use a storage handling mechanism which is orthogonal in its treatment of both and which simplifies the accessing of values contained in routine variables and constants, or reside in data structures. This same storage mechanism is also used to organise the internal "housekeeping" of program execution. We have introduced two constancy flags with each location which with tagging allow a high degree of data protection and consistency checking.

In addition to the instructions described here are a number which exploit the information available at compile time. For example where the type of a value being stored matches the type restriction of a variable,

both being known at compile time, the compiler can generate a version of the "store-frame" instruction which does not perform type checking. Similarly an add operation need not be polymorphic where the types of its integer operands are known at compile time. These "sugared" instructions are faster but add to the size of the interpreter although not to its complexity. The advantages of "large" high level abstract instructions as advocated by Myers [Myer78] may be summarised as follows ; there are less instructions overall to implement resulting in smaller interpreters, and the more compact code is simpler to generate resulting in smaller compilers. We believe that the implementing of powerful languages such as nsl requires new architectures which can handle and exploit polymorphism. They should manipulate tagged data and at least have the simple storage organisation described above.

CHAPTER 8

Summary, Conclusions and Further Work.

Our work has led us to propose an approach to language implementation based on an abstract machine interface between the LDT and MDT.

The Overall Technique.

We have developed a methodology or technique consisting of the following actions which may take place in some logical order or in parallel. The details have been covered in the previous chapters and the appendices should be referred to for example material. We concentrate here on the overall approach. The methodology may seem somewhat simplified in doing this. It is our belief that any methodology should, apart from producing the desired results, attempt to be intuitive. That is, the user should feel confident with it even to the extent that it seems obvious. The measure of success is whether it can be applied easily and perform its task. Unfortunately, like a methodology for language design [Morr80], the results of its application initially are subjective until such time as it is applied by others when it may be objectively judged. We have in this work developed the technique and shown its applicability to increasingly more powerful languages. We see the tasks involved as -

- Analysis of the source language to determine its abstract properties
- Design of the intermediate language
- Examination of the target machine configuration
- Decision on the appropriate mapping
- Writing the compiler, code generator/interpreter

Analysis of the language.

The user of a language (i.e. the programmer) is concerned to a large part with its concrete syntax. The implementor has available to him well proven methods of analysing the concrete syntax and we exclude them from

our discussion. The implementor must consider the language in a very much more abstract form so that its properties, character and so on stand out. He must try to eliminate concrete syntax yet must be able to easily convert from concrete syntax to a much more abstract form. In particular we feel the language should be examined from the following points of view, these being most important to the implementation.

a) Data Types.

These are of particular importance being the properties of the objects manipulated by the programmer. Data types are best split into two areas, primitive and structured. In the main primitive types are straightforward to implement. Examples are numbers and characters. They have simple storage requirements and are produced by monadic or dyadic operations in expressions. The implementor must decide which are the primitive types, what their storage requirements are and what operations are valid with them. For the Algols at least storage for primitives will be either on a stack or within a data structure allocated on a heap.

At this point in the methodology, the implementor is concerned with building some kind of abstract model with which he may understand the underlying storage requirements of objects in the language. He is perhaps attempting to formulate some kind of operational semantics which will be of some use to him when actually implementing the language on some real machine. To a lesser extent he is becoming familiar with the operations which may be performed on objects. Of these the most important are those related to the storage structure namely those concerned with the addressing of objects and the assignment of objects.

For example an integer in a language may be considered thus -
 Storage - requires an atomic cell (i.e. is always accessed as a unit) large enough to hold the required range of values. This may be allocated within a data structure or for the duration of a routine.

Operations - $i + i \rightarrow i$ (addition)

$a[i] \rightarrow$ base type of a (array indexing)

i must be within the bounds of a etc.

Data structures are separately treated because they have more complex storage requirements and operations. Data structures can be considered under the headings of storage creation, accessing and operations.

Storage - A data structure requires storage to hold its collection of data. This will remain in existence for a period determined by the rules of the language. Usually storage is reserved on entry to a procedure and released on exit. Otherwise storage is reserved explicitly at some point during execution and is automatically released when not referred to.

Creation - the implementor must consider what information is available at the time of creation of a data structure. This will be used to determine perhaps the size of the data structure and its initial values.

Accessing - Data Structures comprise of collections of data which may be either values or cells containing data. The data may be primitive or structured.

For example a Pascal record may be considered thus :

Storage - the record consists of locations each of which hold primitive values. It may have variants determined by the value in one of its fields. Its total storage requirements depend on these values.

Creation - a record may be created as part of the local data of a routine or as an array element. No information about the values in the field are available. A record may also be created dynamically where some of the field values are available.

Accessing - a record may be accessed in total or have its fields individually accessed. Accessing of a dynamically allocated record is by means of a pointer to it.

Operations - a record may be assigned.

At this point the implementor will have some kind of 'feel' for the underlying storage structure for his abstraction. There would appear to be two requirements in an abstract model for storage of high level language objects. Either storage is allocated in a stack-like manner or a heap-like manner. These models are well proven and understood. Given our intention to make the implementation straightforward we find them useful at this stage to help the implementor think about the language more abstractly. We may be criticised in that an implementation model is forced too early. This is partly justified because a combined stack/heap storage mechanism is the basis of the majority of Algol-like language implementations. The stack model could be eliminated and expressed in terms of a heap. It could then be reintroduced at a later stage, say when deciding on the run-time environment, as a more efficient way of storage allocation for a particular class of objects. (This is what happened in the implementation of ns1).

Closely related to the abstract storage model is the addressing structure of the language. We suggest the view that an object resides in one of several 'spaces', a space being some kind of abstract collection of objects related by one or more properties. The language may be said to impose rules upon which spaces may be accessed at any point in the execution of the program. For the Algols in particular, the collection of variables in a block may be regarded as a space. They are related in that they disappear on exit from the block. They do not all necessarily come into existence at the start of the block. This may even be regarded as a subspace of the space of all the variables allocated for a particular invocation of a procedure. Each data structure may be regarded as a space because it contains a collection of related variables. One important space is the heap itself. It is a collection of objects which remain in existence according to the rules of the language.

A simple way of addressing these objects is to specify which space and which object. For a variable this will mean identifying the block and the

variable within the block. For a field of a data structure this means identifying which structure and which field. The implementor at this stage may choose to separate out the spaces associated with blocks from that which is the heap (i.e. the space of dynamically allocated data structures). This was done with Algol R but with h and ns1, which needed to retain variables in blocks, all spaces were subspaces of the heap space.

b) Control structures.

At their lowest level control structures may be modelled with jumps and boolean tests. However we have noted that doing so too early in the implementation means that information may be lost which could have been useful in later optimisation. It is better to abstract over the control structures, ridding them of concrete syntactic differences and to categorise them into families of sequence, choice, repetition and abstraction.

Choice constructs may be separated into 1-, 2- and n-armed (e.g. if and case).

Repetition constructs may be boolean controlled with the test at the start, middle or end of the loop. It may be also be range controlled (e.g. for loop).

Procedural or functional abstraction is also an important control structure.

With this abstraction over each kind of control structure it is important to identify the overall properties of the family. This may assist in simplifying the implementation. For example with boolean controlled repetition the loop may be considered to be constructed from three parts, a sequence, a test and another sequence. In order to simplify the three possible boolean controlled loops we say that the first sequence is empty in the case where the test is at the start of the loop and the second sequence is empty where the test is at the end of the loop. We then, in the

abstract form have only one kind of boolean controlled loop. This simplification extends to 1- and 2-armed boolean controlled choice in a similar manner. The reason for this kind of simplification is that it simplifies the abstract machine language into which the source will be compiled.

Design of the Intermediate Language.

Having considered the source language thoroughly the next step is to design the abstract machine language (AML) which will be produced by the compiler. Its purpose is to convey the data and algorithmic information in the source to the next stage of processing. This intermediate language is envisaged as the input to a code generation pass however the implementor may already have decided on a simulation mapping. Even if this has been chosen, the abstract machine language for a translation mapping should be designed first. The abstract machine language for a simulation mapping is at a lower level than that for a translation mapping. It should be designed as a refinement of that for a translation mapping. This is because the same analysis and design which goes towards a higher level AML also should go toward one which is aimed specifically at direct interpretation. The two languages would be very closely related, the lower level language having derivable constructs and operations from the higher. We have noted that the lower level interpreted language is easily produced from the higher. Also at a later date the mapping may be changed to a translation one or the source language may be put onto a machine where a translation mapping is more appropriate. The compiler need not produce the higher level code but may directly output abstract machine code for interpretation however this restricts implementations on other machines to be simulation mappings.

The abstract machine language is a descriptive language primarily in that it is a set of instructions to a code generator. It may be read as the machine code of a very high level abstract machine between Klint's type 2

and 3 described in the introduction. However such a machine is at too high a level to be implemented thus we translate the AML into a lower level AML or to real machine code.

One function of the AML is to convey information about the data used by the program, another is to represent the algorithm of the program. None of the information content of the source program should be lost in producing this AML. The data description may be separated from the algorithm. This may make the code generator easier to write however at the cost of a slightly more complex compiler. On balance it is probably better to have the structure of the AML program isomorphic to that of the source program.

For the Algols, the idea of scope and related variable lifetime must be conveyed in the AML. It is best to use similar nested possibly bracketed constructs as is used in the source (e.g. begin-end pairs). The data descriptions are generated when declarations are met in the source. The data description for a variable should consist of the same information as is contained in the source but in a form much more convenient for later processing by a code generator. The name of a variable is not relevant in the execution of the program but may be carried over to be available during run time debugging. Each named entity in the language is best referred to by a number. This need not be unique if the data descriptions are contained in nested structures reflecting the scope/lifetime of the source entities. Should this bracketing not be used then references to named entities must be distinct and some means must be provided in the AML of specifying the lifetime of these entities.

Each named entity then should be described in the AML with a reference number and information about it. This is usually the type associated with it. For a procedure (or function) it is the list of parameter types (and result type). For an array it is its base type and its index type. All uses of these entities will make use of the reference number, as a parameter to

the use. Examples of these descriptions are given in Appendices A and B. These detail the relationship between the source and the AML.

In translation to an abstract representation, control constructs must retain their structure and not be broken down into lower level jumps. Although they usually have a fairly simple syntax at the source level, at the AM level the syntax should be as simple and regular as possible.

Expressions, data structure accessing and assignment should be converted into reverse Polish. This higher level AM is inherently stack and heap based. The stack is used only for expression evaluation and does not restrict the code generated to using a stack. Chapter 3 shows why this is an excellent representation. A simple but effective technique called pseudo evaluation may be employed to generate optimised code. This technique forms the basis of our proposed code generator.

The AML will be very similar to that used in h. One exception is that the data descriptions should be embedded in the algorithm code as was done in the AML for Algol R. This is easier to produce by the compiler in that separate files need not be maintained for code and data parts of the AML program. The code generator may then employ a stack of descriptors which represent the data currently being described at any point in the program, as opposed to a vector of descriptions of each separate entity in the whole program.

Decision on the appropriate mapping.

The choice of mapping is determined by a number of factors. These include -

- availability of tools
- desire for portability
- storage available on target machine
- whether it has a stack
- speed of machine
- number of registers
- word size
- addressing modes
- runtime facilities desired
- ease of implementation
- efficiency of desired implementation
- level of features in source language (e.g. 1st class procs/types)

Writing of compiler, code generator/interpreter.

We feel that an effective implementation can be produced by using a recursive descent one pass compiler outputting the AML code which is passed to a pseudo evaluation code generator. This second pass may either produce real machine code or a lower level AML which will then be interpreted. The compiler may even produce this interpreted code directly but this approach is less flexible. The design of this code should however be a refinement of the higher level AML which is no longer produced by the compiler.

The reasons for choosing the technique of pseudo evaluation and to extend it were

- a) it was successfully used in a working compiler
- b) it was readily understandable
- c) it satisfied the need of a code generator to simulate the run time environment of a language
- d) it fitted in with our intuitive expectations of an abstract machine for an Algol-like language
- e) it seemed ideal for its originator's professed purpose - expression code generation
- f) it could be extended and developed to deal with all our high level abstract machine language constructs, not just expressions.

Appropriateness of the Technique.

The technique has been applied particularly to three members of the Algol family. Each language employed different, additional or more powerful features than the previous. The characteristics of the Algol family have been outlined in Chapter One.

The technique has also been applied to a purely functional language, SASL [Turn79], insofar as an AML was designed for interpretation. No difficulty was found in doing this possibly because the characteristics of the functional language were mainly those of expressions. That is, many of the features of the Algols such as assignment and control structures were missing from the language.

We feel therefore that if a language embodies similar characteristics to those outlined in the introduction then the technique should be applicable to its implementation. These characteristics cover a very wide range and belong to many languages. The language may contain other data types, operations and data structures. This would not exclude the use of the technique since the AML is high level. The operations and data management only become relevant when the code is being generated. Even at that stage the pseudo evaluation technique is well suited to easing the problem because of its organisational and descriptive properties. The technique may also be applicable to other language features not covered such as parallelism, however this has not been done.

Application of the Technique.

The technique may be applied when a quick yet effective and easily portable implementation is required. The first task is to see if the technique is appropriate for the language concerned. The language must at least be analysed and the decision based on whether it has similar features to those which have been covered.

It is our experience that a code generator without much optimisation would take as much time to write as the compiler. Should a greater degree of optimisation be required then a more complex internal description of the data could be employed by the code generator. This is the most time consuming part of the implementation.

The internal descriptions are the most crucial part of the code generation and effort spent in their design is well rewarded. It is recommended strongly that a very simple code generator be written first even though the intention is to have a highly optimised implementation. This will have the advantages that an implementation can be provided reasonably quickly, and familiarity with the code generation technique will make easier the writing of a second one.

It is also recommended that the compiler produce an AML suitable for a code generator. Even if the mapping will be a simulation on the proposed machine, at a later date it may be desired to use a translation mapping on the same or another machine. The code generator to produce a lower level AML for interpretation should be very straightforward. This is a simple application of Poole's hierarchy of abstract machines.

Assessment of the Technique.

The main advantage that the technique offers is its inherent simplicity and appeal to the intuition of the implementor. The abstract machine language is readily designed and produced from the source. It requires very little in the way of experience on the part of the implementor. Most of the work has been involved in showing that it may be applied to a range of Algol-like languages. The measure of its effectiveness has been the ease of implementing a language without recourse to specialised knowledge or training.

It is difficult to quantify the results of this technique. Robertson [Robe81a] has shown that the code produced by a similar approach to code generation from a high level intermediate language is efficient. We have not done any comparative studies because of time, the lack of available implementations of similar source languages and because of Robertson's encouraging results. We feel that our technique produces acceptable results, this observation being based upon experience with other high level

languages on the machines upon which the work was undertaken.

It has been our intention to simplify language implementation not to develop a technique which will guarantee highly optimised code even though this is possible. The results are subjective in that the technique has not been applied by others. In order to measure its success or failure we feel that an Algol-like language should be implemented using this technique and others. The criteria for such a comparative study of implementation techniques might be time to write and get working correctly the compiler/code generator, its size and speed, and its workspace. The resulting code from these implementations should also be compared from the points of view of space and speed. An additional consideration should be how portable the resulting implementation is. Such a study has not been undertaken as we feel that not enough time was available both to develop the technique and measure it effectively.

We make no claim that it enables the implementor to handle language features which cannot be handled by other techniques. It has been our intention to show that it is effective for those features which characterise the Algol family. Again, subjectively we feel that it has done so and has coped with features not usually associated with the Algols such as first class procedures and first class types. We have no measure as to how well it has done so because of the lack of availability of other languages with these features on the machine on which our languages were implemented (PDP11). In the case of procedures as values, another two Algols, Algol-S [Turn76] and IDEA [Davi79] with that feature were available on another machine. One eliminated the problem by a language restriction which reduced the implementation problems to those of a straightforward Algol. The other used less space at run time than h but at the expense of more complex runtime information and garbage collector.

In summary, the technique is simple, straightforward and highly organisational. It can be used to develop optimised code should this be

desired, but at the expense of greater effort in the code generator. A quicker implementation can be produced by either a simple code generator or an interpreter. The language would be portable by writing a new code generator/interpreter for each machine. No comparative study has been undertaken as to the quantified effectiveness of the technique.

Direction for Further Work.

At present the design of the AML depends on the implementor. Analysis of source languages shows that although the concrete syntax, data types and structures may differ, at the abstract level many of these differences either are eliminated or become less apparent. Thus it would appear that the AMLs for these languages would be very similar. This may suggest that a more formal study take place with a view to designing a framework AML which could be used as the basis of AMLs for languages with similar characteristics. This is not quite a return to the UNCOL philosophy or even that of Janus. It would be a very much more high level abstract machine. Its main purpose would be a guide to the implementor.

We believe that languages of the nature of nsl should be examined. nsl itself was a vehicle to investigate language implementation. In this sense it was experimental, and perhaps too different from the more popular Algols to be readily acceptable. It should be redesigned with emphasis on the following.

- (1) Routine types should include the types of parameter and result. We still feel that only one parameter to a routine is necessary, given the flexible data structures and syntactic sugaring for parameter lists implemented in nsl. The current types proc and fn would then be proc(none), proc(any) and fn(any -> any) where none is the type of the empty object.
- (2) The data structures vector and list could have types associated with them. For example, vector(int), list(char) and list(any).

- (3) This still leaves us with the problem of specifying the type of a parameter list. One possible solution is to have another list type say a fixed list where the types of each element and the number of elements are known. Then the type of

```
function ( c char ; x ; m, n int -> int )
is
fn( flist( char, any, int, int ) -> int )
```

- (4) At the moment structure field types in declarations must be manifest to the compiler. This makes for a simple implementation (see Structure Tables) but there is no reason why they should not be dynamically specified.

```
e.g.
let ty type := ...
:
structure tree ( element ty ; ... )
```

As far as the abstract machine is concerned, such source language changes may involve a tag perhaps becoming a reference to a block containing a description of the type. The storage structure of the abstract machine should also be re-examined with respect to whether it is more efficient (time and space wise) to allocate small single size blocks such as "cons" pairs, or stay with the current scheme. This would involve an investigation of fragmentation.

Acknowledgements.

I would like to thank everyone who helped in some way with this work and thesis. Special thanks are due to the authors of the UNIX operating system, its editing, filing, referencing and text formatting facilities, without which this thesis would have taken longer and been much more difficult to produce. I would also like to thank my wife, Yvonne, who looked after me, and David Harland who provided both intellectual stimulus and moral support. I appreciate and acknowledge the financial support of the Science Research Council of Great Britain.

APPENDIX A

Algol R Intermediate Code.

In this appendix the intermediate code is described with reference to the source and associated abstract machine intermediate code.

Descriptions.

The "declare" instruction in the intermediate code is used to inform the code generator of the attributes of variables, procedures and structures of the source program. It gives a number with which to refer to the entity described. Each such entity has a type and a name. The types of Algol R are extended by "procedure" and "structure" for the purpose of code generation. The "declare" instruction takes the form;

"declare" number name type

This is a directive to the code generator to build a descriptor for the entity. The numbers are allocated by the compiler in order and are used in all references to the described entities. The "declare" statement corresponds to a declaration in the source. These appear at the head of blocks. For example,

<u>Source</u>	<u>Code</u>
begin	block
< declarations >	< declares >
	enddecl
< statements >	< code for statements >
	free n m
end	endblock

The structure of the program is preserved by delimiting blocks by the "block ... enddecl ... endblock" construct. Immediately preceding an

"endblock" may be a "free" instruction. This specifies an upper and lower limit of the descriptor numbers allocated in that block. This directs the code generator to throw away the corresponding descriptors since the numbers will be reallocated for subsequent "declares". The "free" instruction is also used at the end of procedure code to deallocate the descriptors of the formal parameters.

The name in the "declare" is that given in the source. It need not be used by the code generator. The type is in a similar format to the source. The types may be grouped in four sections and we give examples of each.

Primitives

<u>Source</u>	<u>Code</u>
integer a, b	declare 1 a integer
	declare 2 b integer
real r	declare 3 r real
boolean b1	declare 4 b1 boolean
char c	declare 5 c char
struct s2	declare 6 s2 struct

Arrays

An array type is "array" followed by its base type and dimensionality. An array creation instruction, `iliffe.op` may precede the declare if the bounds are specified in the declaration. Its parameters are the base type of the array, its dimensionality, the number of specified dimensions and the number of such arrays to be created.

Source

```
integer array( E1 :: E2 ) a1, b1
```


Code

(E1)

(E2)

iliffe.op integer 1 1 2

declare 7 a1 array integer 1

declare 8 b1 array integer 1

The bound list in the source specification may have asterisks in it representing unknown bounds. If the bound list is all asterisks then no "iliffe.op" is produced since only space need be reserved but no array is to be built. Following the type, dimensionality and specified dimensions of the "iliffe.op" is the number of such arrays to be built. If the bound list has some but not all bounds specified then these specified bounds are stacked for use by "iliffe.op". For example,

Source

char array(*) string

Code

declare 9 string array char 1

Source

struct array(1 :: 5, *, *) ptr

Code

stackconst i 1

stackconst i 5

iliffe.op struct 3 1 1

declare 10 ptr array struct 3

Procedures

A procedure type is represented by "procedure" followed by the parameter types followed by "->" and the result type. Procedures which are not functions have "void" result type. The procedure description is terminated by an indication of whether the body follows or not. The procedure declaration may be "forward" in which case the actual declaration appears later. The same declare statement, apart from the terminators "forward" and "present" appears at both declarations. With a present procedure, the body immediately follows. This is bracketed with "segment ... enddecl ... endsegment". The declares for the formal parameters lie before the "enddecl".

Source

```
integer procedure pl
( integer a ; struct point )
: source code for body
```

Code

```
declare 20 pl procedure integer struct
-> integer present
segment
declare 21 a integer
declare 22 point struct
enddecl
: code for body
endsegment
```

Example of forward procedure :

Source

```
procedure p2 ; forward
```

```
:
```

```
:
```

```
procedure p2 ;
```

```
  : Source code for p2
```

Code

```
declare 12 p2 procedure -> void forward
```

```
:
```

```
:
```

```
declare 12 p2 procedure -> void present
```

```
segment
```

```
enddecl
```

```
  : code for p2
```

```
endsegment
```

Structures

A structure definition defines the template. The structure class name and field names are used in the source and must have descriptors built for them.

Source

```
structure jim
```

```
  ( struct result, list ;
```

```
  integer array (*) size ;
```

```
  boolean isempty )
```

Code

```

declare 12 jim structure
declare 13 result struct
declare 14 list struct
declare 15 size array integer 1
declare 16 isempty boolean
endfield

```

Use of descriptor numbers.

When a variable, procedure, field, or structure is required for some purpose, it is referred to by its descriptor number. A descriptor number is assigned on a "declare" statement. On lexical exit from a block or segment, those descriptor numbers assigned in that block or segment are made available for use again, by means of the "free" statement. Thus the descriptor number is a convenient way of referring to any variable, procedure, field or structure which is in scope in the source text. Wherever a described entity is needed, the "stack" instruction is used. This has a single parameter which is the descriptor number of the entity.

<u>Source</u>	<u>Code</u>
---------------	-------------

x	stack < descriptor number for x >
---	-----------------------------------

Array or structure element values.

<u>Source</u>	<u>Code</u>
---------------	-------------

E1(E2)	(E1) (E2) sub.op 0
----------	------------------------

Constants.

Constants are loaded using the "stackconst" instruction. This specifies the type of the constant and its value.

<u>Source</u>	<u>Code</u>
-3	stackconst i -3
3.01	stackconst r 3.01
true	stackconst b true
false	stackconst b false
'%'	stackconst c %%
'a'	stackconst c %a
"abc" " "	stackconst s 5 %abc'
nil	stackconst n

A single character is preceded by a percent. Strings are preceded by their length and a percent marks the start of the string.

Expressions.

Let E with or without a subscript stand for a source expression, then (E) is the code generated.

Unary operators.

<u>Source</u>	<u>Code</u>
+E	(E)
~E	(E) not.op

And similarly for the following :-

<u>Source</u>	<u>Operator</u>	<u>Code</u>	<u>Operator</u>
-		neg.op	
abs		abs.op	
code		code.op	
decode		decode.op	
upb		upb.op	
lwb		lwb.op	
float		float.op	n
truncate		truncate.op	

If the integer with "float" is "2" float the second top of stack otherwise float the top.

Binary operators.

<u>Source</u>	<u>Code</u>
E1 + E2	(E1) (E2) plus.op

And similarly for the following :-

Source Operator Code Operator

-	minus.op
div	div.op
rem	rem.op
/	divide.op
*	times.op
=	eq.op
~=	ne.op
<	ls.op
>	gt.op
<=	le.op
>=	ge.op
is	is.op

"eqs.op" is used for equality of strings.

Exceptions :-

E1 isnt E2	(E1) (E2) is.op not.op
E1 or E2	(E1) or.op (E2) endor
E1 and E2	(E1) and.op (E2) endand

Note the latter two are hybrid control structures.

Assignment.

The left hand side of an assignment denotes the l-value (address) of a variable. Thus for subscripting on the left hand side of an assignment the last subscript operation must specify the address of the resultant variable. This is done by specifying a different parameter to "sub.op" from that when the value is required.

Source

x := E

Code

```
stack.address < desc.nr. of x >
```

```
( E )
```

```
ass.op
```

Source

```
EO( E1, ... , En ) := En+1
```

Code

```
( EO )
```

```
( E1 )
```

```
sub.op 0
```

```
:
```

```
( En )
```

```
sub.op 1
```

```
( En+1 )
```

```
ass.op
```

Procedure calls.Source

```
EO( E1, ... , En )
```


Code

```
stack < desc.nr.of proc E0 >
```

```
mark.stack
```

```
parlist
```

```
( E1 )
```

```
ep
```

```
:
```

```
( En )
```

```
ep
```

```
endpar
```

```
apply.op
```

Source

```
E0
```

Code

```
stack < desc. nr. for proc E0 >
```

```
mark.stack
```

```
apply.op
```

Structure creation.

This is done by specifying the structure (by means of its descriptor number) and each of the field values. E.g. using the structure declaration shown previously.

Source

```
jim( void, nil, limit, false )
```

Code

```

stack < desc. nr. for structure jim >
formvec
stack < desc. nr. for struct void >
ef
stackconst n
ef
stack < desc. nr. for array limit >
ef
stackconst b false
ef
formend

```

If statement and expressions.Source

```

if E1 then E2 else E3

```

Code

```

if
( E1 )
endbool
then
( E2 )
else
( E3 )
endif

```

Source

```

if E1 do E2

```

Code

```
if ( E1 ) endbool
```

```
do ( E2 ) endif
```

While statement.Source

```
while E1 do E2
```

Code

```
while ( E1 ) endbool
```

```
( E2 ) endwhile
```

For statement.Source

```
for i := E1 to E2 by E3 do S
```

Code

```
for
```

```
( E1 )
```

```
to
```

```
( E2 )
```

```
by
```

```
( E3 )
```

```
declare i integer
```

```
( S )
```

```
endfor
```

Case statement.Source

```
case i of 1,2,3 : S1 ; 0 : S2 ; default : s3
```

Code

```
case
```

```
( E )
```

```
of
```

```
( S1 )
```

```
endswitch 3 1 2 3
```

```
( S2 )
```

```
endswitch 1 0
```

```
( S3 )
```

```
endswitch 0
```

```
switch.op
```

```
endcase
```

An integer giving the number of constants precedes their values. The constants may be integers, characters or strings and have the same representation as stacked constants. (see before).

APPENDIX B

h Intermediate Code.

The intermediate code may be considered to be a medium-to-low level language for a vaguely specified stack and heap based computer. The instructions effect their results by means of manipulating descriptors of their data.

STOP - halts execution of the program.

STACK number - Push the descriptor referenced by the number onto the evaluation stack.

ADDR - Consider as address. The descriptor on top of the stack is used only to get the address of the entity described not the value.

ASS - Assign the value described on top of the stack to the entity described beneath it. Pop both descriptors.

STACKL number - Push the descriptor of the literal referred to by the number onto the evaluation stack.

CLOSE - Close the stream described on top of the stack. Remove its descriptor.

READ - Replace the stream descriptor on top of the stack by a descriptor of a character input from that stream. The character may have been output by a WRITE or WRITEB instruction since for characters the effect is the same.

READI - Replace the stream descriptor on top of the stack by a descriptor of an integer built from characters input from the stream.

READBI - Replace the stream descriptor on top of the stack by a

descriptor of an integer input from the stream and output by a corresponding WRITEB instruction.

READBS - Replace the stream descriptor on top of the stack by a descriptor of a string input from the stream, which was output by a corresponding WRITEB instruction.

POP - Pop the top descriptor on the stack.

WRITE - Pop the top two or three descriptors on the stack. These describe an entity output to the data stream described beneath it on the stack. On top is the width field in which to right justify the representation of the entity. All entities whatever their type are output as a stream of one or more printable characters.

WRITEB - Pop the top descriptor on the stack which describes an entity (char, integer or string) output to the data stream described beneath it on the stack. The entity is output in a form readable only by READ, READBI or READBS instructions.

Binary Operators.

The two descriptors on top of the stack are replaced by a descriptor of the result of the operation. These operators are as follows.

PLUS, MINUS, MULT, IDIV, REM, CAT, SUCC, REPL, SSTR, OPEN, RELOP relop

The relops are "=", "~=", ">", ">=", "<" and "<=". Note SUCC takes a range end limit as well as its normal operand.

Monadic Operators.

The descriptor on top of the stack is replaced by a descriptor of the result of the operation.

ABS, CODE, DECODE, UPB, LWB, NOT, EOF, STL, PRED, ORD, STOV, STOC, VTOS,

CTOS, ESTR

There are a number of control structures which consist of a combination of operators and streams of instructions amongst which may be other control structures.

OR - Perform a boolean inclusive 'or'.

OR stream

On top of the stack is a descriptor of a boolean, if the value is true do not evaluate the stream but carry on with the next instruction after the stream. If the value is false, pop the stack and evaluate the stream pushing its result on the stack.

AND - Perform a boolean 'and' operation.

AND stream

On top of the stack is the descriptor of a boolean. If it is false, do not evaluate the following stream but carry on with the next instruction after it. If it is true, pop the stack, evaluate the stream and push on the result.

IF - This control structure causes execution of one or other of two streams depending on the result of a boolean stream.

IF streamb stream1

ELSE stream2

Streamb is evaluated. The boolean described on top of the stack is tested and its descriptor popped. If the result is "true" then stream1 is executed and control commences after the code for stream2. If the result is "false", control commences with the code for stream2.

LOOP - This causes repeated execution of code until a boolean evaluates

to false. The boolean controlling the exit from the loop lies between two streams of code.

LOOP stream1 streamb stream2

Stream1 is executed. Streamb is evaluated and the boolean result described on top of the stack is tested and its descriptor popped. If the test is "false" execution commences after stream2, otherwise stream2 is evaluated and execution recommences at the start of stream1. This continues until streamb evaluates to "true".

FOR - This is another loop-type clause. The condition for exit is an arithmetic comparison.

FOR istream sstream lstream number cstream

The number is the descriptor number of a control variable for the loop. Istream is evaluated and assigned to the control variable popping its descriptor. Sstream is evaluated and saved in a 'hidden' (to the user) location. Its descriptor is popped. Lstream is evaluated and also saved in another hidden location. Its descriptor is popped. The control variable is compared with the result of lstream. If the value of sstream is positive then, if the control variable is less than or equal to the result of lstream execution continues after the for statement. I.e. it is exited. If the value of sstream is negative then, if the control variable is greater than or equal to the result of lstream, execution continues with cstream, otherwise execution continues after the FOR statement. After evaluation of cstream the control variable is incremented with the result of sstream. Execution recommences with the comparison of the control variable and the result of lstream as described above.

CASE - In the CASE statement the value of a stream determines which one of several streams of code is executed.


```

CASE sstream
cstream ... cstream : bstream
:
:
cstream ... cstream : bstream
DEF bstream

```

The sstream is evaluated and compared for equality with the results of evaluating the cstreams in order of appearance. Should it match one then the following first bstream is evaluated to give the result of the whole CASE. If none match then the bstream following DEF is evaluated to give the CASE result.

TEST - This is exactly the same as CASE except the cstreams are STACKL's of a structure trademark and the sstream evaluates to a pointer to a structure. The comparison is on that structure trademark and the literal's value.

MAKEPROC number - Push on to the stack a descriptor of the procedure value created and referred to by the number.

ATVEC - Puts values into a vector.

ATVEC streamlb streaml ... streamN MAKE RD

The stream after the ATVEC is the lower bound of the vector to be created. An vector is created of the correct size and each element starting with the one at the lower bound is initialised with the result of the corresponding stream. The RD may not be present. If RD follows the elements of the initialised vector are only allowed to be read.

BUILDVEC - Creates a vector initialised by repeated evaluation of a section of code.

BUILDVEC stream1 streamu streami RD

The first two streams after the BUILDVEC are the lower and upper bounds of a vector of the type specified, which is created. Each element of the vector is initialised by evaluating afresh the third stream. The RD may not be present just as in ATVEC. If RD follows then the elements of the initialised vector may only be read.

SEG - This statement describes a self-contained body of code called a procedure or segment.

SEG number stream

The number is the table number for this body of code or procedure. SEG is always reached by a procedure call and together with other house-keeping operations, saves the place to which control must return on exit, i.e. after execution of the stream. A segment may not contain embedded segments.

CALL - Set up local data space for a procedure and call it.

CALL number stream ... stream

Each call of a procedure evaluates the stream in that procedure's corresponding SEG statement. However new local data space must be found onto which the local variables are mapped. The number is that of the description of parameters (offsets within their type areas) which this frame will contain. If there are parameters, then for each one, the result of evaluating the stream is assigned to a parameter in the frame created. Which parameter depends on the number of parameters in the table whose number follows the CALL. The first value is assigned to the first parameter and so on. The descriptor on the stack resulting from evaluation of each parameter is popped. After all parameter streams have been processed the procedure whose descriptor is on the stack is called. I.e.

execution starts at the SEG statement for the procedure using the new data frame created. After the execution of the procedure control recommences after the CALL instruction. The procedure descriptor is popped. If it returns a result then the result descriptor is pushed on to the stack.

SUBV - Subscript vector.

On top of the stack is a descriptor of an entity of type scalar, integer or character. Beneath it is a descriptor of a vector. The top element is converted to an integer if it is not already and used to subscript the vector beneath it. The two descriptors are replaced by a descriptor of the element of the vector accessed by the subscript operation.

SUBS - Subscript structure.

On top of the stack is a descriptor of a field of a structure. Beneath it is a descriptor of a ptr. They are replaced by a descriptor of an entity of the same type as the field which describes the field accessed by the subscription operation.

MAKESTR - Build a structure.

MAKESTR number stream1 ... streamN

The number is the descriptor number of the structure. It is followed by one stream per field in the structure. A descriptor of the ptr pointing to the structure created and initialised by the streams is pushed onto the stack

Syntax of Intermediate Code.

```
code = segment { segment }
segment = "seg" number opt.stream
stream = element { element } ";"
opt.stream = { element } ";"
element = single
```

```

| .snum number
| "or" stream
| "and" stream
| "if" stream opt.stream
  "else" opt.stream
| "loop" opt.stream stream opt.stream
| "for" stream stream stream number opt.stream
| ( "case" | "test" ) stream
  stream { stream } ":" opt.stream
  { stream { stream } ":" opt.stream }
  "def" opt.stream
| "atvec" stream stream { stream } "make" [ "rd" ]
| "buildvec" stream stream stream [ "rd" ]
| "call" number { stream }
| "makestr" number stream { stream }
single = "addr" | "stop" | "close" | "ass" | "pop"
| "subv" | "subs" | "read" | "readi" | "readbi"
| "readbs" | "write" | "writeb" | "succ" | "ord"
| "plus" | "minus" | "mult" | "idiv" | "rem"
| "cat" | "repl" | "sstr" | "open" | "abs" | "code"
| "decode" | "upb" | "lwb" | "not" | "get" | "eof"
| "stl" | "stov" | "stoc" | "vtos" | "ctos" | "estr"
snum = "stack" | "stackl" | "mass" | "relop" | "makeproc"

```

Relationship between Syntax and Code Generated.

In the following, the angle brackets "<" and ">" in the code generated section represent "code generated for" the syntactic entity between them.

syntax: sequence **"**"**

code: < sequence > **"stop" ";"**

syntax: ["constant"] name "<-" clause

code: < clause > "stack" number "addr" "ass"

The numbers after the "stack" instructions are the descriptor numbers corresponding to the names.

syntax: "if" clause "then" clause "else" clause

code: "if" < clause > ";" < clause > ";" "else" < clause > ";"

The corresponding single branched "if" statement is :

syntax: "if" clause "do" clause

code: "if" < clause > ";" < clause > ";" "else" ";"

syntax: "while" clause "do" clause

code: "loop" ";" < clause > ";" < clause > ";"

syntax: "repeat" clause "while" clause "do" clause

code: "loop" < clause > ";" < clause > ";" < clause > ";"

syntax: "repeat" clause "while" clause

code: "loop" < clause > ";" < clause > ";" ";"

In the three above syntactic forms, if the "while" is replaced by an "until" then in the code generated a "not" is inserted after the "until" clause and before the semicolon.

syntax: "for" name "<-" < clause > "to" < clause >

"by" < clause > "do" < clause >

code: "for" < clause > ";" < clause > ";" < clause > ";"

number < clause > ";"

When there is no "by" clause i.e. :

syntax: "for" name "<->" < clause > "to" < clause > "do" clause

"by" 1 - is assumed in the source.

syntax: "case" clause "of" case_list "default" ":" clause

where

```
case_list = clause { "," clause } ":" clause ";"
           [ case_list ]
```

code: "case" clause ";"

```
< clause > ";" { < clause > ";" } ":" < clause > ";"
:
:
:
< clause > ";" { < clause > ";" } ":" < clause > ";"
"def" < clause > ";"
```

syntax: "test" clause "is" is_list

"isnt" ("use" name "in" | ":") clause

where

```
is_list = name_list ( "use" name "in" | ":" ) clause ";"
           [ is_list ]
```

This gives the same structure of code as the "case" clause. However "test" instead of "case" is generated. The code generated for a structure name in the name list is

stack1 number ";"

where number is the number of the descriptor of the trademark corresponding to the structure name.

```
syntax: "write" clause "," clause [ ":" clause ]
        { "," clause [ ":" clause ] }
```

```
code: < clause > < clause > < clause > "write"
      { < clause > < clause > "write" } "pop"
```

If the first clause, that is, the one specifying the stream, should be absent then code to stack the file for standard output is generated. If any of the width clauses, that is those following the colon, should be missing then code to stack the appropriate width value is generated. The width value used is determined by the type of the clause to be output.

```
syntax: "writeb" clause "," clause { "," clause }
```

```
code: < clause > < clause > "writeb" { < clause > "writeb" } "pop"
```

```
syntax: ( "read" | "readi" ) "(" clause ")"
```

```
code: < clause > ( "read" | "readi" )
```

If no clause appears then code is generated to stack the variable containing the file descriptor for standard input. The corresponding operation is generated for each of the read types.

```
syntax: ( "readbi" | "readbs" ) "(" clause ")"
```

```
code: < clause > ( "readbi" | "readbs" )
```

```
syntax: "open" "(" clause "," clause ")"
```

```
code: < clause > < clause > "open"
```

```
syntax: "close" "(" clause ")"
```

```
code: < clause > "close"
```

```
syntax: lvalue "!=" clause
```

code: < lvalue > "addr" < clause > "ass"

syntax: "null"

code: does not generate any code

syntax: "at" clause "make" ["constant"] "[" clause { "," clause } "]"

code: "atvec" < clause > ";" < clause > ";" { < clause > ";" }
 "make" ["rd"]

The "rd" is generated only if "constant" appears.

syntax: "vector" clause "::" clause ["constant"] "val" clause

code: "buildvec" < clause > ";" < clause > ";" < clause > ";" ["rd"]

The "rd" is generated only if "constant" appears.

syntax: "procedure" "(" [proc_spec_list] "->" [type] ")" clause

code: "makeproc" number

Code for the clause is generated and output at this point.

The code generated and output for the procedure is :

code: "seg" number < clause > ";"

The number in both cases is one with which to refer to information about the procedure.

syntax: exp "or" exp

code: < exp > "or" < exp > ";"

syntax: exp "and" exp

code: < exp > "and" < exp > ";"

syntax: exp relop exp

code: < exp > < exp > "relop" < relop >

The code generated for a relop is as follows.

Source	Code
=	"="
~=	"~="
>	">"
>=	">="
<	"<"
<=	"<="

syntax: [addop] exp addop exp { addop exp }

code: exp [< addop >] < exp > < addop > { < exp > < addop > }

The code generated for the optional addop preceding the first exp in the syntax is as follows.

Source	Code
"+"	no code generated
"_"	"neg"

The code generated for the dyadic addops are :

Source	Code
"+"	"plus"
"_"	"minus"

syntax: exp multop exp { multop exp }

code: < exp > < exp > < multop > < exp > < multop >

The multop codes are :

Source	Code
"x"	"mult"
"div"	"idiv"
"rem"	"rem"
"cat"	"cat"
"repl"	"repl"

syntax: "succ" exp

code: < exp > "stackl" nr "succ"

The number after the stackl is the number of values in the scalar type of the scalar expression.

syntax: monop exp

code: < exp > < monop >

The monop codes are :

Source	Code
"abs"	"abs"
"code"	"code"
"decode"	"decode"
"upb"	"upb"
"lwb"	"lwb"
"~"	"not"
"stl"	"stl"
"pred"	"pred"
"ord"	"ord"
"stov"	"stov"
"stoc"	"stoc"
"vtos"	"vtos"
"ctos"	"ctos"
"eof"	"eof"

```
syntax: simple { "(" [ clause { "," clause } ] ")"
```

```
    | "[" clause { "," clause } "]"
```

```
    | "{" clause { "," clause } "}"
```

```
    | "|" clause "|"
```

```
    | "[" clause "," clause "]" }
```

```
code: < simple >
```

```
{ "call" number [ < clause > ";" { < clause > ";" } ]
```

```
  | < clause > "subv" { < clause > "subv" }
```

```
  | < clause > "subs" { < clause > "subs" }
```

```
| < clause > < clause > "sstr"
```

```
| < clause > "estr" }
```

Depending on the form and type of simple one of the above is generated. After the "call" is the number of the description of the parameter types. There is one special case of the above viz.

```
syntax: simple "{" clause { "," clause } "}"
```

```
code: "makestr" number < clause > ";" { < clause > ";" }
```

This is the case where the "simple" is a "name" ; i.e. the name of a structure. The number is the trademark number of that structure.

The code for "simple" is as follows.

Source	Code
name	"stack" number
	"stackl" number
literal	"stackl" number
"(" sequence ")"	< sequence >
"begin" sequence "end"	< sequence >

For the "stack" instruction the number is that of the descriptor for that name. For the "stackl" (stack literal) the number is that of the descriptor for the literal.

Where lvalue syntax matches the above syntax, the same code is generated.

Syntax of Data Information.

```
data_info = lex param_desc literal_desc "!"
lex = "lex" number { info } "endlex" type
      flag number { flag }
```



```

flag = "t" | "f"
info = type | struct_desc | lex
type = simple_type | proc_type | vector_type
simple_type = "int" | "char" | "bool"
            | "ptr" | "string" | "file" | "void"
            | "scalar" number
proc_type = "proc" type
vector_type = "vector" simple_type
            ( simple_type | proc_type )
struct_desc = "structure" type { type } "endstructure"
param_desc = number { number type { type } }
literal_desc = number { literal }

```

Relationship between Syntax and Data Descriptions Generated.

```
syntax: sequence "***"
```

```

data: "lex" integer < sequence > "endlex" "void"
      ( "t" | "f" ) number { "t" | "f" }
      param_desc lit_desc "!"

```

The main program is treated as a routine value. The compiler outputs details of the parameter lists and literals used in the program. Only one copy of each different one is output. The parameter lists are a space saving device used by the code generator (see CALL instruction).

```
syntax: "procedure" "(" [ proc_spec_list ] "->" [ type ] ")" clause
```

```

data: "lex" integer [ < proc_spec_list > ] < clause > "endlex" < type >
      ( "t" | "f" ) number { "t" | "f" }

```

The integer after the "lex" is the routine value's identifying number. The first flag is the copy creator frame flag. Then follows a count and the copy display entry flags.

```
syntax: [ "constant" ] name { "," name } ":" type [ proc_spec_list ]
```

```
data: < type > { < type > } [ < proc_spec_list > ]
```

Each variable or constant parameter type is output. This is also the case for in-line variables and constants or structure fields.

```
syntax: "structure" name "be" "{" structure_spec_list "}"
```

```
data: "structure" < structure_spec_list > "endstructure"
```

```
syntax: name { "," name } ":" type [ structure_spec_list ]
```

```
data: < type > { < type > } [ < structure_spec_list > ]
```

```
syntax: [ "constant" ] name "<-" clause [ "coerce" type ]
```

```
data: < type >
```

If within the same sequence a name has already been specified in a "forward" declaration then nothing is generated for that name otherwise < type > is generated for each such name. The type is the same throughout for this. It is the type after the "coerce" if present, the type of the clause if not.

```
syntax: [ "forward" ] name { "," name } "be" type
```

```
data: < type > { < type > }
```

```
syntax: "for" name "<-" clause "to" clause [ "by" clause ]
```

```
data: type
```

```
syntax: "use" name "in"
```

```
data: type
```

Code generated for types is as follows.

<u>Type</u>	<u>Output</u>
integer	"int"
char	"char"
boolean	"bool"
ptr	"ptr"
string	"string"
file	"file"
scalar	"scalar" integer
procedure	"proc" type
vector	"vector" type type
void	"void"

The integer after the "scalar" is the number of the scalar type.

The type generated after the "proc" is the result type of the procedure.

The types following the "vector" is first the subscript type then the element type.

Literals are output as follows.

<u>Type</u>	<u>Output</u>
integer	"int" integer_value
string	"string" string_value
scalar	"scalar" integer integer_value
char	"char" char_value

The literal value in the case of integer is the integer itself.

The literal value in the case of string is the length of the string, followed by the characters of the string.

The integer in the scalar is the number of values in the scalar type.

The literal value in the case of scalar is the number of the scalar literal in the list of names describing the scalar type.

The literal value in the case of char is the character value itself.

No literals are output for "nil", "true" or "false".

APPENDIX C

Other nsl Abstract Machine Instructions.

These are instructions of the TAM machine not covered in chapter 7.

Data Structure Implementation.

The TAM machine supports a wide range of data structures. These are all blocks and thus are treated in a similar manner. Some of the data structures, like "stack" and "code" blocks are purely for internal working but others are provided for the implementation of data structures of a polymorphic programming language. We will concentrate on the latter. There are three, directly reflected in nsl. They are the list, the vector and the structure.

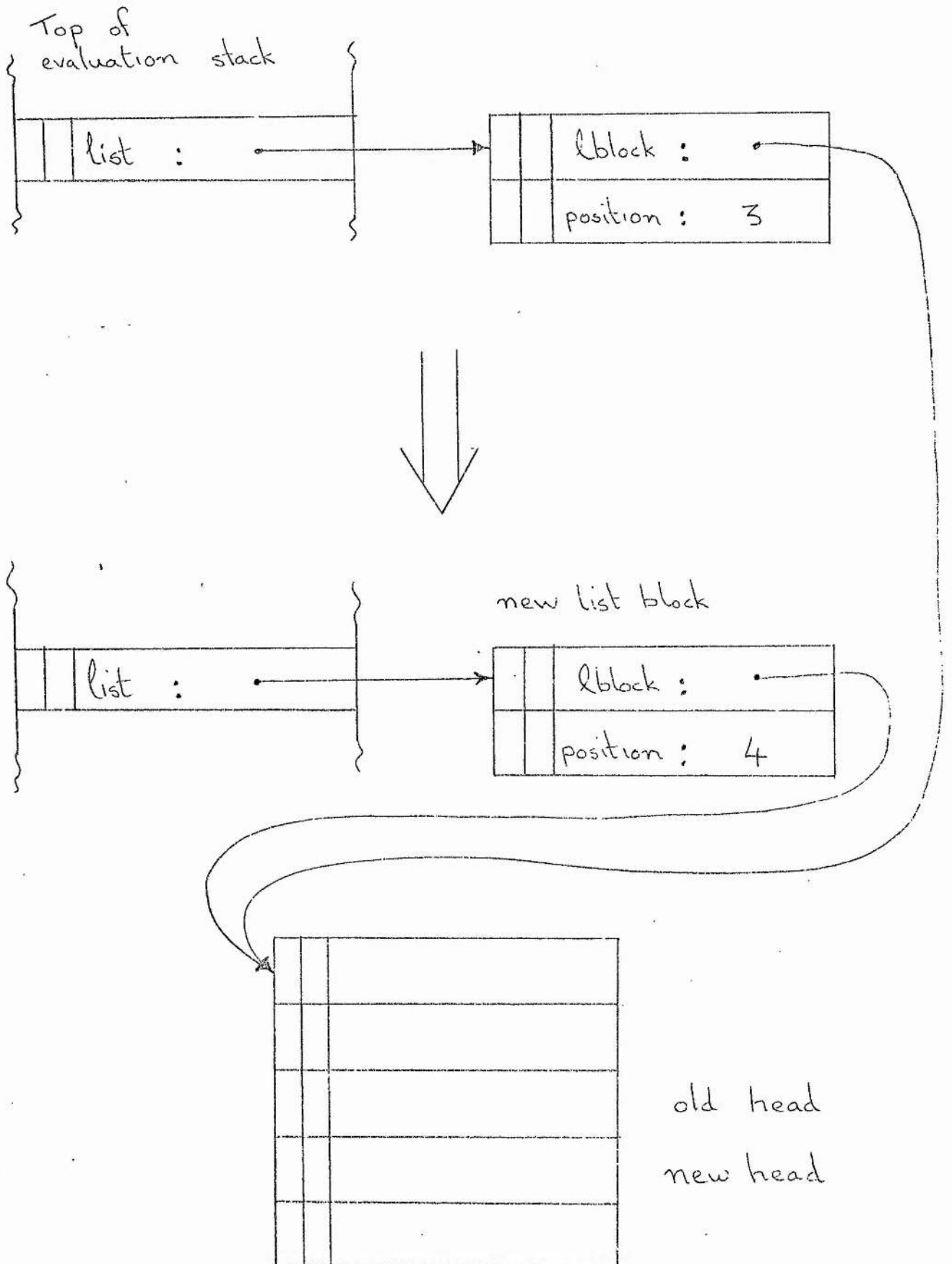
Lists.

A list is an ordered collection of heterogeneous objects characterised by "head" and "tail" operations. One of the conventional ways to implement them is by a "cons" [McCa62] pair. In our machine, this would mean that each member of the list needs two cells in a block, the first being the head, the second being the tail, a list. In fact we have chosen a different organisation more in keeping with the abstractions over all data structures and slightly more efficient for certain list operations. We believe that list manipulation in nsl will exploit the iterative constructs supporting creation rather than the recursive addition of elements to lists.

A list value is a reference to a block of two cells. The first cell value is a reference of type "list block" to a block of value-constant cells containing values of any type. The second is a position. The position is that of the first element of the list (which need not be the first in the block of values). This and following cells in the list block

comprise the members of the list. A "head" operation on this list will push that element on the evaluation stack (having popped the list value). A tail operation creates a new two cell list with the position value one greater but the same list block value. An empty list does not have a block reference as its actual value but a nil value. List head and tail operations are not expensive since no copying of the list members is done, and they are less wasteful of space than if "cons" pairs were used. List elements may also be accessed of course by using positional accessing. Accessing an arbitrary member of a list by position would be more restrictive if "cons" pairs were used. Note that the constancy flags on the element cells ensure list cells are not updateable. This is because nsl interprets lists as being collections of values and not locations. Figure 10 shows a "tail" operation on a list.

Figure 10.



Vectors.

A vector is an ordered collection of heterogeneous objects characterised by selection of individual cells by an integer selector limited by defined bounds. A vector value is either nil or is a block reference. The first cell in this block is the lower bound, the second is the upper bound. Bound values are of type "integer". The remaining cells are the vector elements each corresponding to a selector value.

Structures.

A structure value is a reference to a block of cells each of which is selected by a unique value of a field type directly associated with the structure type. Every created structure is based on a template held in the Structure Table. A template block defines the types and flag settings of the component cells. A structure value is a reference to a copy of such a template block, but with the actual values filled in. The field cell tags determine how the fields may or may not change their value and type, and whether the initialising and subsequently assigned values supplied for the fields are the correct type. Each structure template represents a unique structure type. Structures modelled on a particular template may only have their fields selected by a value of a unique selector field type. Values of this selector type may not select cells of any other type of data structure. In particular they may not be used to select a field of another structure value.

Data Structure Space Allocation.

Data structure space allocation and initialisation is performed in a single instruction. However we separate these processes only for the purposes of discussion.

To allocate a data structure there must be provided a type, and possibly a lower and an upper bound. The type is that of the desired data

structure. This is of course a value of type "type". The lower and upper bounds are values of type "integer". Specification of bounds is not necessary in all data structure creations. Where they are provided, in the most general of the data structure creating instructions, they must conform to what one would expect. Vectors need both, to determine the size of vector to create and the range of allowable selecting values. Lists and structures must have a lower bound of one. The upper bound is used to determine the number of elements in the list. For structures the bounds are not actually necessary although, if specified, the lower bound must be one and the upper bound must match the size of the corresponding template. This is really a check that all the fields will be initialised. Their number is determined by the corresponding template.

Vector and list creation simply involves the creation of appropriately sized blocks. Created structures are copies of the template thus have their flags and tags filled in where specified. In all creation instructions, when a zero sized block would result, the actual value part is made a nil block reference giving a nil data structure as the result of the operation.

Initialisation.

There are several initialisation schemes common to all data structures. These characterise the data structure creating operations and each is performed by a single instruction. Operands for creation and initialisation either reside on the evaluation stack or in the code. In some cases they may be implicit. We will now describe the schemes and their instructions.

Enumeration.

The "data-structure-enumerated" ("DSE") instruction takes as an operand in the code, the number of values on the stack to be used in initialising the data structure. On the stack, beneath these initialising

values lie the lower bound above the data structure type. The upper bound for vectors is determined from the size and the lower bound.

A data structure of the specified type is created. Its cells are initialised in turn from the lowest initialising value on the stack. Should the number of initialising values be zero, then a nil data structure value will result. The type of each initialising value is checked against the corresponding type restrictions (if any) of the data structure. All values down to and including the type are popped and the data structure value is pushed. The DSE instruction would be generated for each of the constructs in the next example.

```
list [ k, l, m ]
( if x then dsty{ i } else vector ) at 1 [ "label", 5, m*4 ]
binary[ '+', exp( 4 ) ]
```

Literal.

The "data-structure-literal" ("DSL") instruction is provided for data structures where the data structure type and its initialising values are known at compile time. These are therefore generated in "code" blocks as literals by the compiler. This saves pushing the literal values on the stack and then creating a data structure. The initialising values are extracted directly from the code. The operand in the code is a count of the literals following the instruction in the code. Immediately preceding these is the data structure type as a type literal. The lower bound is implicitly one and the upper bound is the number of literals. The data structure is created and initialised with the usual checking being performed. A DSL instruction would be generated for the nsl construct "abc", that is, a list of literal characters.

Replication.

The "data-structure-replicate" ("DSR") instruction uses a single value which is replicated over the whole data structure. That is, each cell is initialised with that value. It takes no operands in the code but on the stack lie the initialising value above an upper bound, above a lower bound, above the data structure type. The data structure is created with each cell initialised with the specified value. The instruction would be generated for the ns1 constructs in the next example.

```
list size a + b value init()
```

```
vector at -m upto +m value 0
```

```
binary size 2 value '+'
```

Position Controlled.

A newly created data structure is completely initialised on creation (since it is not sensible to allow use of the data structure until this has been done). This is a philosophy recognised in some of the later languages [Morr79] which impose it on data structures and variables. The simple default initialisation of the creating instructions given above can be subsequently updated to "re-initialise" a complete data structure with an individually calculated value for each element. This re-initialisation is available to the ns1 programmer in the form of a powerful ns1 construct as shown here.

```
list size 40 with pos eval pos * pos
```

This example builds the 40 element list :

1, 4, 9, 16, 25, 36,

This construct is supported by a loop determined by a set of three instructions which take an already existing data structure and control the assigning of values to its elements in order of their position. An ns1 data structure will have been created with the DSR instruction and a dummy

initialising value. A position value determines the current cell to be "re-initialised". The position value starts at one and steps up by one to the number of elements in the data structure. It is held in a value-constant cell of the local frame. Throughout the operation of these three instructions the data structure value will be on the stack.

The "data-structure-initialise" ("DSI") instruction has the position in the local frame of this controlling cell, as an operand in the code. The function of the instruction is to initialise the control cell to one, that is, the position of the first element in the data structure.

The "data-structure-jump" ("DSJ") instruction has as code operands a label (a position in the current code block) and the position of the control cell in the local frame. On top of the stack is the partly-reinitialised data structure value. The position value in the control cell is compared with the number of elements in the data structure. If it is greater than it then a jump takes place to the specified label. The completely reinitialised data structure is left on the stack from where it make take part in any valid operation such as assignment.

Following this instruction will be code to generate, for the current cell in the data structure, a new initialising value on the stack. Then comes the "data-structure-store" ("DSS") instruction which has a label and a control cell position as code operands. The value above the data structure on the stack is popped to the cell in the data structure whose position is given by the value in the control cell. This value is incremented by one. A backward jump is then taken to the label at which will be a data-structure-jump instruction. The example shows where code is generated for a position-controlled initialisation.


```

list size s ! generate DSR replicating 0 !
with i      ! generate DSI initialising i !
eval        ! L1 : DSJ i, L2 !
f()          ! code for call leaving result on stack !
             ! DSS i, L1 !
             ! L2 : !

```

Loading and Storing of values.

Values residing in blocks may be transferred to the evaluation stack and vice versa. A cell in a particular block is accessed by a selection or position value, and a value whose actual value part is a block reference. There are load and store instructions which do this, calculating these values from implicit values, code operands and values on the stacks. A load is the moving of a value from a cell in a block onto the evaluation stack. A store is the reverse operation. The load and store instructions fall into two groups. One group assumes implicitly that the loads and stores involve frames of the current environment. The other makes no such assumption but relies on block reference values on the stack, or in other blocks. Because of the orthogonality of the storage structure, both groups utilise the same internal "micro-operations". Frames are really data structures supporting routines.

Frame Cell accessing.

On top of the control stack lies the current environment. Each of the cells in this environment block in turn contain references to the frames of calls of routine values statically surrounding the current one in the source. These contain the accessible variables and constants. The first cell in the environment block contains a reference to the current local data frame. The "load-frame" ("LDF") instruction has two code operands. One is the number of frames down the current environment from the top to find the required frame. The other is the position within that frame of the cell. If the cell is in the local frame of the current routine then the number of frames down will be zero and so on. To obtain the particular

frame the machine converts this value to a position in the environment block. The contents of the cell at the specified cell in the particular block are pushed on the evaluation stack. Note that the uniformity of the machine means that frame cell accessing is the same as data structure positional accessing (see later) apart from the conversion of a routine-level difference to a block reference. In fact, all cell accessing is ultimately coerced into positional accessing for processing by internal "micro-operations".

The "store-frame" ("STF") instruction does the opposite of this. It has the same code operands and the particular cell is located in the same way. The value on top of the evaluation stack is popped into this cell. A check is made that the final cell is not value constant. If it is type constant then a check is made that the type of the value being stored is the same as that already in the cell.

Initialising Frame Cells.

A special case of storing is the initialising of local frame cells. This corresponds to the declaration of ns1 variables and constants. It combines the storing of a value with a setting of the type and value constancy flags. The "store-frame-initialise" ("STFI") instruction has two code operands and a possible third. One is the position of the cell. The current frame is implicitly taken to be the one containing the cell. The second operand is the constancy flags in an encoded form. The flags in the cell are set according to this operand. If the type-constant flag was set then a type value is expected specifying how the cell is to be made type constant. This is either a third operand in the code (statically determined by the compiler) or is already on the stack (dynamically evaluated) underneath the initialising value. The machine exploits its tags to determine whether there is a third operand. If so, there will be a value of type "type" in the cell following the other two operands,

otherwise there will be an instruction. Thus the type, if needed, may be specified statically or dynamically. Whichever it is, the machine checks that the value supplied is of type "type".

General Block Loads/Stores.

When addressing a cell within a block two values are needed, a block reference and a means of specifying the cell within the block. This is either selection or positional access. In fact selectors are converted into positions for use by internal "micro-operations" during the actual access. Checking is performed, ensuring that a selecting value is valid for the block reference value. For example, integers select cells in vectors, checking against the bounds, and field values select cells of matching structures.

It often happens that we want to string together several successive selections or positional accesses terminating in a single cell access. For example, a vector element may be a structure value, one of whose fields is a vector, one of whose elements we wish to access. The process of accessing the last cell is one of repeatedly using a block reference, selection/position value pair to extract from a cell a new block reference value. This value is then used as the next block reference together with the next selector/position value. Initially, a block reference and a series of selector/position values are necessary; intermediate block reference values are extracted from cells determined by all but the last selection/position operation performed. The last block reference value and selection/position value address the cell required.

The "load-stack" ("LDS") instruction performs precisely this function. It takes as a code operand the number of selector/position values on the evaluation stack. Beneath these will be the initial block reference. These are popped and the value extracted from the last cell is pushed.

The "store-stack" ("STS") instruction is the corresponding store. It differs in that above the selector/position values on the stack is the value to be stored. The block accessing takes place as before except that the value on top of the stack is stored in the last cell, rather than its value being extracted. As with other stores, a check is made that the cell being accessed is not value-constant and that the value being stored matches the type constancy of the cell.

Exploiting Lists.

The load and store instructions are adequate for the implementation of a polymorphic programming language like nsl. However, by taking multiple values residing in lists, instead of single values, shorter, faster instructions may be used in certain cases. The following instruction descriptions may be divided into the four families, general list stripping, multiple initialisation, multiple stores and parameter stripping. These involve operations on several values possibly held as a list.

General List Stripping.

It is necessary to be able to take a list value on the stack and replace it with its component values, these being pushed in order of appearance in the list. The instruction "strip" ("STR") does this, and takes as a code operand the expected size of the list. The actual size of the list must match this.

Multiple Initialisation.

This occurs when a several initialisations are performed together. On the stack will be a number of values. These may have been pushed there by a number of successive expression evaluations or by the use of the "strip" instruction on a single list value. These values are used to initialise adjacent local cells by the "multiple initialise" instruction (mnemonic "MI"). It takes a count and a starting position as code operands. The

number of stacked values given by the count are stored in turn in the cells of the local frame starting at the specified position. The lowest value is stored first and so on.

Cells initialised by this instruction have their flags set in batches by repeated use of the "set-flags" ("SEFL") instruction. It takes at least three code operands, a count of the number of cells, a starting position and the encoded flags for each cell. If the type-constancy flag is set the type appears on top of the stack or as a fourth operand as previously explained in the "store-frame-initialise" instruction. The cells determined by the count and starting position have their flags set according to those specified by the instruction. If the type-constancy flag is set, the type tags in the cells are checked against the specified "type" value. The following example shows where these instructions are generated for a piece of nsl code.

```
let a, b, c const int := 1
! code to push list 1 !
! STR !
! MI !
! SEFL 3, pos, fl, < type : int > !
```

Multiple Stores.

In the same way that several initialisations of local cells may be performed by a single instruction, several assignments may be performed at once. Each individual assignment needs an address and a value. The "multiple-store" ("MS") instruction has as a code operand an integer count. On the stack are a number of values above the same number of addresses. The number of each is the same as the count operand. A cell address reflects the way a cell is accessed. The address on the stack is a pair, a base value which is a block reference, beneath a selector value. Note there is no concept of pointing directly into the middle of a block. These addresses are generated by instructions similar to the "load-frame" and "load-stack" instructions except they push the addresses of the cells

not their contents. The values on the stack are stored in the cells specified by the corresponding address pairs, the lowest value going into the cell addressed by the lowest pair on the stack. The MS instruction would be generated for the ns1 code in the following example.

```
a, b, c := c, a, b
```

Parameter Stripping.

Every routine takes exactly one parameter. This resides in the first cell of a local frame. When several values are to be passed in, the user may explicitly pass in a data structure. Once inside the routine, using the general load instruction described above, he may extract its components, explicitly initialising locally declared cells as in the next example.

```
let p := procedure ( v : const vector )
    begin
        let a, b int := v{ 1 }, v{ 2 }
        :
    end
;
p( vector[ m, n ] )
```

This however may be done implicitly by high level stripping and initialisation instructions.

On a call instruction, the machine determines whether more than one parameter was supplied by inspecting the operand. It automatically builds a list if there is more than one, this being passed as the actual parameter. Inside a routine there can be planted an instruction which automatically breaks this list up. This is the "strip-fill" ("SF") instruction which takes as a code operand a count of the expected length of the actual parameter which must be a list. The parameter is checked for being a list of the required length and is stripped apart, each element value being stored, in turn, into the cells following the first in the frame. These cells are in fact being initialised and still must have their

value and type constancy subsequently specified by repeated use of the SEFL instruction. The following example shows these instructions generated for a piece of nsl code.

```

let p := procedure ( a, b int )
    begin
        ! The instructions are generated at entry !
        ! SF 2 !
        ! SEFL 2, 2, < type : int > !
        :
    end
:
p( m, n )

```

References

Aho77

A. V. Aho and J. D. Ullman
Principles of Compiler Design
Addison-Wesley 1977

Amma73

U. Amman
"The Method of Structured Programming as applied to the Development of
a Compiler"
International Computing Symposium, North Holland 1973

Bail80

P. J. Bailey, P. Maritz, and R. Morrison
"The S-Algol abstract machine"
Technical Report CS/80/2 Dept. Comp. Sci., Univ. of St. Andrews
1980

Barb80

R. Barbuti and A. Martelli
"Static type checking for languages with parametric types and
polymorphic procedures"
Lecture Notes in Computer Science, Springer Verlag International
Symposium on Programming 83 pp1-16 1980

Barr63

D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey
"The main features of CPL"
Comp. J. 6 pp134-143 1963

Baue73

H. Bauer, S. Becker, and S. Graham

"Algol W Implementation"

Technical Report CS98 Stanford University 1973

Bell73

J. R. Bell

"Threaded code"

C.A.C.M. 16 6 370-372 1973

Berr71

D. M. Berry

"Introduction to Oregano"

Proc. Symp. on Data Structures in Programming Languages, Sigplan

Notices 6 2 1971

Berr79

D. M. Berry and R. L. Schwartz

"United and discriminated record types in strongly typed languages"

Information Processing Lett. 9 1 13-18 1979

Bobr73

D. G. Bobrow and B. Wegbreit

"A model and stack implementation of multiple environments."

C.A.C.M. 16 10 pp591-602 1973

Bohm66

C. Bohm and G. Jacopini

"Flow diagrams, Turing machines and languages with only two formation rules"

C.A.C.M. 9 pp366-371 1966

Bril72

P. C. Brillinger and D. J. Cohen

Introduction to Data Structures and Non-Numerical Computation

Addison-Wesley 1972

Broo82

G. R. Brookes, I. R. Wilson, and A. M. Addyman

"A static analysis of Pascal program structures"

Software - Practice and Experience 12 10 pp959 1982

Brow69

P. J. Brown

"Using a macroprocessor to aid software implementation"

Comp. J. 12 4 pp327-331 1969

Brow72

P. J. Brown

"Levels of Language for Portable Software"

C.A.C.M. 15 12 pp1059-1062 1972

Burs80

R. M. Burstall, D. B. MacQueen, and D. T. Sannella

"HOPE: an experimental applicative language"

Internal Report CSR-62-80 Dept. Comp. Sci., Univ. of Edinburgh 1980

Catt80

R. G. Cattel

"A Survey and Critique of some Models of Code Generation"

Technical Report Computer Science Dept., Carnegie-Mellon University

1980

Cole74

S. S. Coleman, P. C. Poole, and W. M. Waite

"The mobile programming system Janus"

Software - Practice and Experience 4 pp5-23 1974

Cree69

B. A. Creech

"Architecture of the B6500"

COINS-69 Third International Symposium 1969

DEC71

DEC

PDP 11 Processor Handbook

DEC 1971

DEC81

DEC

VAX Architecture Manual

DEC 1981

Dah172

O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare

Structured Programming

Academic Press 1972

Daki73

R. J. Dakin and P. C. Poole

"A mixed code approach"

Comp. J. 16 3 219 1973

Davi79

A. J. T. Davie

"Variable access in languages in which procedures are first class citizens"

Technical Report CS/79/2 Dept. Comp. Sci., Univ. of St. Andrews
1979**Davi81**

A. J. T. Davie and R. Morrison

Recursive Descent Compiling

Ellis Horwood 1981

Daws73

J. L. Dawson

"Combining interpretive code with machine code"

Comp. J. 16 3 216 1973

Deme80

A. J. Demers and J. E. Donahue

"Data types, parameters and type checking"

Proceedings 7th Annual Principles of Programming Languages Symposium

pp12-23 1980

Dijk68

E. W. Dijkstra

"GOTO Statement considered harmful"

C.A.C.M. 3 pp147-148 1968

Evan68

A. Evans

"PAL-A language designed for teaching programming linguistics"

Proc. ACM 23rd Nat. Conf. pp395-403 1968

Feus72

E. A. Feustel

"The Rice research computer - a tagged architecture"

A.F.I.P.S. S.J.C.C. 40 pp369-377 1972

Feus73

E. A. Feustel

"On the advantages of tagged architecture"

IEEE Transactions on Computers C-22 7 pp644-656 1973

Grie71

D. Gries

Compiler Construction for Digital Computers

Wiley 1971

Gunn78

H. I. E. Gunn

"h reference manual"

Dept. Comp. Sci., Univ. of St. Andrews 1978

Gunn79

H. I. E. Gunn and R. Morrison

"On the implementation of constants"

Information Processing Lett. 9 1 pp1-4 1979

Gunn80

H. I. E. Gunn

"hil reference manual"

Technical Report CS/80/3 Dept. Comp. Sci., Univ. of St. Andrews
1980

Gunn81

H. I. E. Gunn and D. M. Harland

"Degrees of constancy in programming languages"

Information Processing Lett. 13 1 pp35-38 1981

Gunn82

H. I. E. Gunn

"Compile time type checking of structure field accessing"

Information Processing Lett. 14 1 pp22-25 1982

Har181

D. M. Harland

"The application of message passing to concurrent programming"

Ph.D. Thesis Dept. Comp. Sci., Univ. of St. Andrews 1981

Har182

D. M. Harland and H. I. E. Gunn

"Another look at enumerated types"

SIGPLAN Notices 17 7 pp62-71 1982

Horn74

J. J. Horning

"Structuring Compiler Development" in

Advanced Course on Compiler Construction, Tech. Univ. of Munich
1974

Hunt81

R. Hunter

The Design and Construction of Compilers

Wiley 1981

IBM70

IBM

IBM System /360 Principles of Operation

IBM 1970

Ilif68

J. K. Iliffe

Basic Machine Principles

Macdonald 1968

Jens65

J. Jensen

"Generation of Machine Code in Algol Compilers"

BIT 5 pp215-245 1965

Jens74

K. Jensen and N. Wirth

PASCAL - User Manual and Report

Springer-Verlag 1974

John71

J. B. Johnston

"The contour model of block structured processes"

Proc. Symp. on Data Structures in Programming Languages SIGPLAN

Notices 6 2 pp55-82 1971

Klin81

P. Klint

"Interpretation Techniques"

Software - Practice and Experience 11 pp963-973 1981

Korn80

P. Kornerup, B. B. Kristensen, and O. L. Madsen

"Interpretation and code generation based on intermediate languages"

Software - Practice and Experience 10 pp635-658 1980

Land64

P. J. Landin

"The mechanical evaluation of expressions"

Comp. J. 6 4 pp308-320 1964

Land66

P. J. Landin

"The next 700 programming languages"

C.A.C.M. 9 3 pp157-164 1966

Lisk74

B. Liskov and S. Zilles

"Programming with abstract data types"

Proceedings Symposium on Very High Level Languages, SIGPLAN Notices 9

4 pp50-59 1974

McCa62

J. McCarthy

LISP 1.5 Programmer's Manual

MIT Press Cambridge Mass. 1962

McKe65

W. M. McKeeman

"Peephole Optimisation"

C.A.C.M. 8 7 pp443-444 1965

McKe74

W. M. McKeeman

"Compiler Construction" in

Advanced Course on Compiler Construction Tech. Univ. of Munich 1974

Miln78

R. Milner

"A theory of type polymorphism in programming"

Journal of Computer and System Sciences 17 3 pp348-375 1978

Morr76

R. Morrison

"The Algol R Abstract Machine"

Dept. Comp. Sci., Univ. of St. Andrews 1976

Morr77

R. Morrison

"A method of implementing procedure entry and exit in block structured high level languages"

Software - Practice and Experience 7 4 1977

Morr78

R. Morrison

"Algol R"

Technical Report CS/78/1 Dept. Comp. Sci., Univ. of St. Andrews
1978

Morr79

R. Morrison

"S-Algol reference manual"

Technical Report CS/79/1 Dept. Comp. Sci., Univ. of St. Andrews
1979

Morr80

R. Morrison

"On the development of Algol"

Ph.D. Thesis Dept. Comp. Sci., Univ. of St. Andrews 1980

Myer78

G. J. Myers

Advances in Computer Architecture

Wiley 1978

Naur63

P. Naur and others

"Revised Report on the Algorithmic Language Algol 60"

C.A.C.M. 6 1 pp1-17 1963

Naur63..

P. Naur

"The Design of the GIER Algol Compiler"

BIT 3 pp124-140,145-166 1963

Newe72

M. C. Newey, P. C. Poole, and W. M. Waite

"Abstract Machine modelling to produce Portable Software - A Review and Evaluation"

Software - Practice and Experience 2 pp107-136 1972

Nori74

K. V. Nori, U. Amman, K. Jensen, and H. H. Nageli

The PASCAL (P) Compiler Implementation Notes

E.T.H. Zurich 1974

Pool74

P. C. Poole

"Portable and Adaptable Compilers" in

Advanced Course on Compiler Construction, Tech. Univ. of Munich
1974

Rand64

B. Randell and L. J. Russell

Algol 60 Implementation

Academic Press 1964

Reyn70

J. C. Reynolds

"GEDANKEN-A simple typeless language based on the principle of completeness and the reference concept"

C.A.C.M. 13:5 pp308-319 1970

Rich69

M. Richards

"BCPL:A tool for compiler writing and system programming"

Proc. A.F.I.P.S. S.J.C.C. 557 1969

Rich71

M. Richards

"The portability of the BCPL compiler"

Software - Practice and Experience 1 pp135-146 1971

Robe77

P. S. Robertson

"The IMP-77 language"

Internal Report CSR-19-77 Dept. Comp. Sci., Univ. of Edinburgh 1977

Robe81

P. S. Robertson

"The production of optimised machine code for high-level languages using machine-independent intermediate codes"

Thesis CST-13-81 Dept. Comp. Sci., Univ. of Edinburgh 1981

Russ65

D. B. Russell

"On the implementation of SLIP"

C.A.C.M 8 pp263 1965

Sib161

R. A. Sibley

"The SLANG system"

C.A.C.M. 4 75 1961

Site71

R. L. Sites

"Algol W Reference Manual"

Technical Report STAN-CS-71-230 Stanford University 1971

Smit71

G. D. Chesley and W. R. Smith

"The hardware-implemented high-level machine language for SYMBOL"

Proc. A.F.I.P.S. S.J.C.C. pp563-573 1971

Stee61

T. B. Steel

"UNCOL the myth and the fact"

Ann. Rev. Aut. Prog. 2 1961

Stee61..

T. B. Steel

"A first version of UNCOL"

Proc. A.F.I.P.S. W.J.C.C. 19 371 1961

Stra67

C. Strachey

"Fundamental concepts in programming languages"

Oxford University Press 1967

Tane82

A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson

"Using Peephole Optimisation on Intermediate Code"

A.C.M T.o.P.L.a.S 4 1 1982

Tenn77

R. D. Tennent

"Language design methods based on semantic principles"

Acta Inf. 8 97-112 1977

Turn76

D. A. Turner and R. Morrison

"Towards portable compilers"

Technical Report CS/76/5 Dept. Comp. Sci., Univ. of St. Andrews
1976

Turn77

D. A. Turner

"Error diagnosis and recovery in one pass compilers"

Information Processing Lett. 6 4 pp113-115 1977

Turn79

D. A. Turner

"SASL reference manual"

Technical Report CS/79/3 Dept. Comp. Sci., Univ. of St. Andrews
1979

Wait70

W. M. Waite

"Building a mobile programming system"

Comp. J. 13 28 1970

Wait73

W. M. Waite

Implementing Software for Non-Numerical Applications

Prentice-Hall 1973

Wait74

W. M. Waite

"Code Generation" in

Advanced Course on Compiler Construction Tech. Univ. of Munich 1974

Wegb74

B. Wegbreit

"Procedure closure in ELI"

Comp. J. 17 1 1974

Weiz68

J. Weizenbaum

"The funarg problem explained"

M.I.T. Cambridge, Mass. 1968

Wijn75

A. van Wijngaarden and others

"Revised report on the algorithmic language Algol-68"

Acta Inform. 5 ppl-256 1975

Wilk64

M. V. Wilkes

"An experiment with a self compiling compiler for a simple list-processing language"

Ann. Rev. Aut. Prog 4 1964

Wirt66

N. Wirth

"Euler: a generalisation of Algol and its formal definition"

C.A.C.M. 9 1 and 2 1966

Wirt68

N. Wirth

"PL360 'A Programming Language for the /360 Computers'"

J.A.C.M. 15 37 1968

Wirt77

N. Wirth

"What can we do ... "

C.A.C.M. 20 11 pp822-823 1977