An APL Subset Interpreter for a New Chip Set

by

James Hoskin

MSc(Physics), University of Calgary

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
May 1987

Approval

Title of Thesis: An APL Subset Interpreter for a New Chip Set

Name: James D. Hoskin

Degree: Master of Science

Examining Committee:

Chairperson: Dr. W. S. Luk

Dr. R. F. Hobson
Senior Supervisor

Dr. Jay Weinkam

Dr. R. D. Cameron
External Examiner

Dr. Carl McCrosky
External Examiner

April 28, 1987
Date Approved:

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

AN APL SUBSET INTERPRETER FOR A NEW CHIP SET

Author:

(signature)

James (Zeke) Hoskin

(name)

May 14, 1987

(date)

Abstract

The APL language provides a powerful set of functions and operators to handle dynamic array data. Current APL interpreters are hampered by excessive interpretive overhead. The Structured Architecture Machine project has produced a machine architecture (SAM) and an intermediate language (ADEL) intended to allow an APL interpreter to execute almost as quickly as good compiled code for most applications.

This thesis describes a microcoded interpreter for a subset of APL. It differs from earlier programs used for preliminary benchmarks and architecture evaluation in that it contains a complete memory management subsystem and a nearly complete linker, and some of the tables have been moved from dedicated hardware to segmented memory.

Performance measurements show that SAM APL is two orders of magnitude faster than a good microcomputer APL and suggest that the implementation behaves more like a compiled language than an interpreted one. Worst-case performance is not as good as previously predicted. Several approaches to increasing execution speed are discussed, and it is expected that the ultimate performance will meet the goals of the SAM project.

The interpreter developed in this project is intended to become the nucleus of a practical APL system. The final system will require code for the primitives, operators, and data types which were omitted in this version, an error handler, and input/output routines tailored to the final hardware design.

Back in the old days, in 1962
A feller named Ken Iverson decided what to do.
He gathered all the papers he'd been writing for a spell
And put 'em in a little book and called it APL.

Now, writing jots and squiggles is a mighty pleasant task
But it doesn't answer questions that a lot of people ask.
Ken needed an interpreter for folks who couldn't read
So . . . .

from "APL Blossom Time"

J. C. L. Guest

Acknowledgements

I gratefully acknowledge the help and support of the following good people:

- Dr. Richard F. Hobson, my senior supervisor, who knows more about the state of the software than is decent for a hardware specialist.
- Ted Edwards, who initiated me into the mysteries of APL and encouraged me throughout this project.
- Janice Cranna, Don Gamble, John Simmons, Ron Spilsbury, and Warren Strange, who provided much-needed assistance in keeping my part of the project in tune with the rest of it.
- The APL mailbox group on the I. P. Sharp timesharing system, and the Vancouver APL Users Group, for listening to my bad ideas and telling me their good ones.
- My wife, Tamarissa, who helped keep me going

This work has been partially supported by the Science Council of B. C. and the S.F.U. Centre for Systems Science

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. Interpreted Languages

Program developers are traditionally faced with a clear-cut choice between using a compiled language or an interpreted one. Languages that support dynamic data structures are difficult to compile. Also, it is usually impossible to fix and restart a compiled program at the point where an error was detected.

The major disadvantage of interpreted programs has been lack of speed. A traditional interpreter has to do almost as much work per line of code as a compiler would, and does this work every time the line is executed.

One obvious solution is to provide both an interpreter and a compiler for the same language, using the interpreter for development and then compiling the code for production [Bud81 Bud84, GuW78, Per74, Sah78, Wei79, Wgg85]. Both IBM [DrO86, Chi86]and STSC [PMI86] have produced partial compilers for their APL systems. For practical systems, it may suffice to rewrite the most heavily-used code in a compilable language and provide a good interface to compiled modules as part of the interpreter [Wgg86].

Compilation makes sense for programs that will be run many times unchanged. There are many cases, however, where a program may be written to solve a given problem and discarded, or when a program is undergoing modification throughout its useful life. For

example, the interpreter developed in this project is a set of APL programs. It is being fixed whenever bugs are discovered and extended as time becomes available to implement more features. It would be a serious nuisance to face a recompilation cycle every time a change is made.

Another approach is "incremental compilation" [Van77]. A line of code is translated into machine code the first time it is encountered during the execution phase, and the machine code is retained for subsequent executions of that line. Incremental compilation introduces a significant trade-off between speed and generality. The fastest possible object code for a given operation would include such details as the rank, shape, type, and location of the operands explicitly. This kind of code would probably be generated by a good interpreter on any given pass through the corresponding line of source code. The more specific the code produced by an incremental compiler on the first execution of a given line, the greater is the probability that, on a subsequent execution, a property of some operand will have changed too much to allow the code to be used.

The advent of microcoded processors suggested a completely different treatment. If the primitives of the language are written in microcode, then the code generation phase of the interpretive process will be unnecessary, and execution may be speeded up enough to compensate for the time taken by the remaining semantic checking and memory management tasks. [HLL73] and [Zak78] describe two interesting microcoded APL interpreters. The most important example of a microcoded APL is IBM's APL Assist [HaL76], which supplements an existing APL interpreter with some microcoded routines, giving a speedup ranging from slight to a factor of 8. In general, improvements due to microcoding some or all of an interpreter on existing hardware have not given performance approaching the speed of compiled code.

Hobson [Hob82, HGT86] has chosen an architectural approach to the problem of interpretive overhead. In his functionally distributed multiprocessor, one unit handles the interpretation and another does the actual data manipulation. Input/output and user communication are handled by a third processor. Early benchmarks indicate that the performance of an interpreter tailored to such a machine can be almost as good as that of compiled code running on a single-microprocessor system.

The project of developing such an architecture is well under way. This thesis deals with the development of an appropriate interpreter for it.

1.2. Languages and Data Structure

The function of a high-level computer language is to insulate the programmer from the low-level details involved in making a lightning-idiot machine work on a practical problem. There are two main approaches to this insulation: program structure and data structure.

A "structured" language generally has built-in mechanisms for loops and branches to spare the programmer from the details of counting and jumping past alternatives, and some way of preventing unrelated chunks of code from interfering with each other. More advanced languages have ways to link separately-compiled programs, resolve forward references, and so forth

A data structure, such as a Lisp list or a SNOBOL string, is a way of representing a collection of data objects as a single primitive object. All practical computer languages support some kind of data structures, generally arrays. However, many languages, which will here be called "scalar" languages, do not operate on these structures except in a piece-by-piece manner. The languages such as Lisp, SNOBOL, SETL, APL, and Nial which

actually can operate on data structures as data structures are often called "very high-level languages", and have been found to be more straightforward to use (on problems which suit their data structures) than the scalar languages.

APL is the language chosen for this interpreter. APL pioneered the use of "operators" which apply functions to arrays. That is, "+" is the ordinary addition function, and "/" is the "reduction" operator such that +/ARRAY produces the sum along the rows of the array. This approach is more general than simply implementing a "sum" primitive, because the operator can apply to a whole set of primitives producing "rowproduct" with the multiplication function, "rowmaximum" with the maximum function, "columnsum" when used with the "axis" operator, and so on.

The drawbacks of APL, aside from the interpretive overhead addressed by the SAM project, are its unusual character set, its lack of convivial ways to structure code, and its lack of interfaces to external devices. Some approaches to these problems are discussed in in Appendix B.

1.3. Previous and Related Work

At the time this thesis project was begun, the Architecture Support Package (ASP) system already existed in the form in which it was used for developing the current version of the interpreter [Hob83]. Thornburg [Tho83] had already written a model of function call and return using an earlier version of ASP. Gudaitis [Gud85] had written a body of microcode implementing various array accessing algorithms.

The call and return protocol used in the current version of the interpreter is loosely based on Thornburg's. It is not certain to what extent the final set of execution microcode will use Gudaitis' work.

Outside the SAM project, the most closely related body of work is the series of DELtran experiments of Flynn and Hoevel [FIH83, FIH84]. Their work combines the contour model of [Joh71] with a format-based scheme for encoding and executing instructions. The resulting code can be more compact than machine code for traditional architectures, and corresponds more closely to high-level source code. Recent related work by the members of this group includes an execution architecture for Backus' FP language [HHH86] and an extension of the format-based instruction architecture to cover parse subtrees (and directed acyclic subgraphs) of different depths [FJW85].

Several other groups are doing related work. McCrosky [McC85, McJ86] has designed an array-theory-based processor for the Nial language. The SAM architecture is similar to the "decoupled" architectures described in [Hsi85 Lio85, SWP86].

The DEL (Directly Executable Language) stream of computer architecture can be compared and contrasted with DEX (Direct Execution) of source programs [Chu79, ChA81]. By working with a linearized and compact translation of the source program, a DEL implementation gets higher speed and simpler hardware at the expense of retranslation during program editing. Hobson [Hob84] defines a DIL (Directly Interpretable Language) as a linearized form from which the source code can be recovered

Another related field is the design of hardware architectures designed to support functional languages [Veg84 RuM86], production systems [Qui86], and Artificial Intelligence languages [Wah87]. The SAM architecture is not highly specialized for a particular source language and should support implementations of any of these languages more efficiently than conventional uniprocessor configurations. Since SAM's memory architecture is designed to handle arrays efficiently, an array-based AI language like Nial [MGJ84, JGM86] should run especially well.

Chapter 2

The SAM Architecture

The Structured Architecture Machine is based on the SJ16 microprocessor (See Figure 2-1), a microengine designed to be used as a microprogrammed high-level controller in a multiprocessor environment. The SJ16 has fairly conventional data paths, but the microinstructions include unique control features for high-level microprogramming. These control features allow several microoperations to be specified and performed in parallel.

A SAMjr (Figure 2-2) system consists of an SJ16 with a segmented memory controller, a segmented memory consisting of at least one megabyte of dynamic RAM, and zero or more Special Function Units (SFUs).

A complete SAM system (Figure 2-3) consists of an Environmental Control Unit (ECU) connected to two or more SAMjr systems by dual-port memories. The ECU handles the external environment, including the user interface and mass memory. One SAMjr acts as the Program Management Unit (PMU), and is responsible for fetching instructions and maintaining the execution environment. The other SAMjr is the Data Management Unit (DMU), which does the actual data manipulations. There may be more than one DMU. The SAMjr systems are connected to each other by a custom pipeline processor which handles instruction and operand verification while queuing instructions decoded by the PMU for execution by the DMU.

In the system currently under development, the ECU is a stand-alone microcomputer

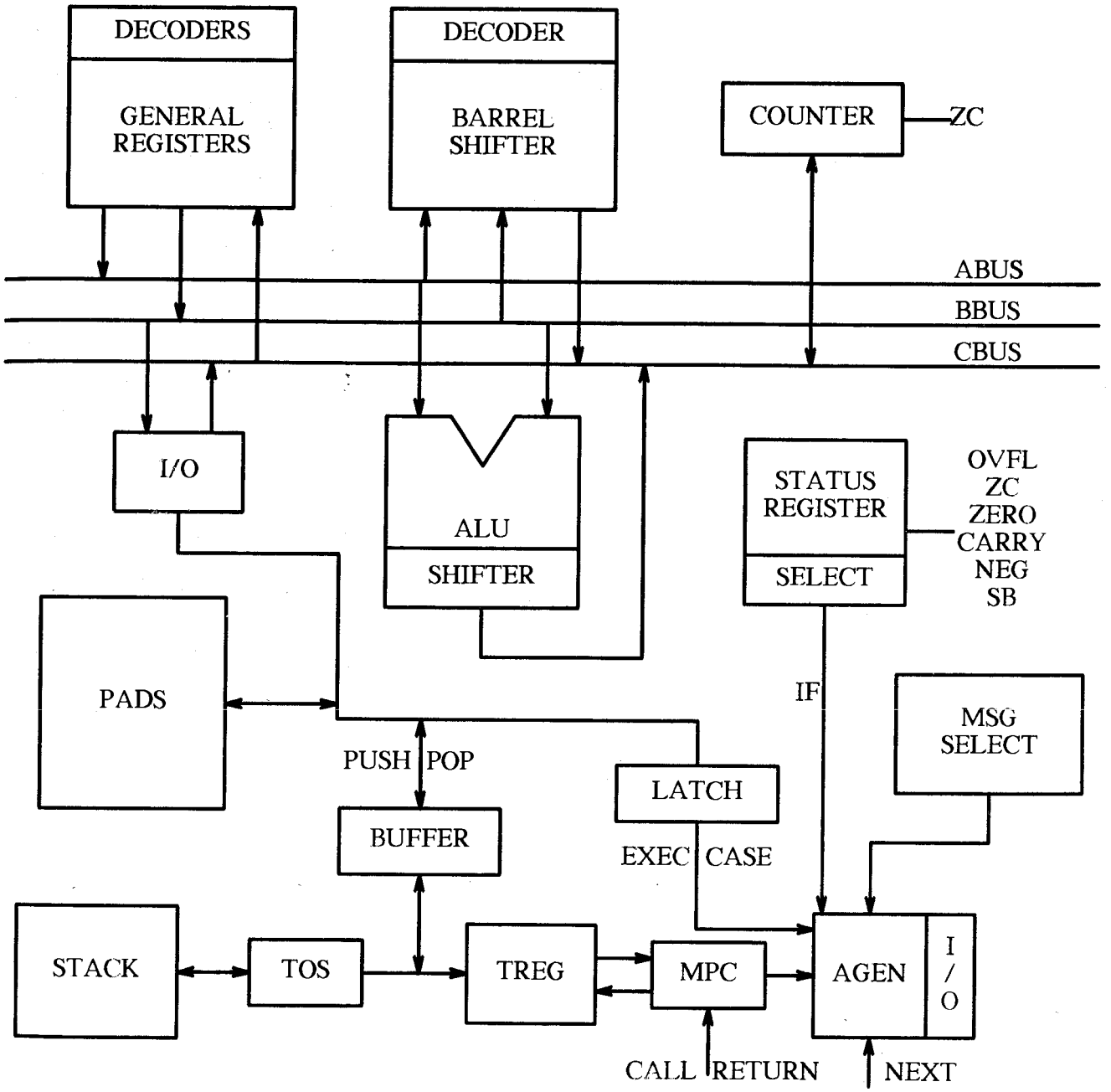

Figure 2-1: The SJ16 Microengine

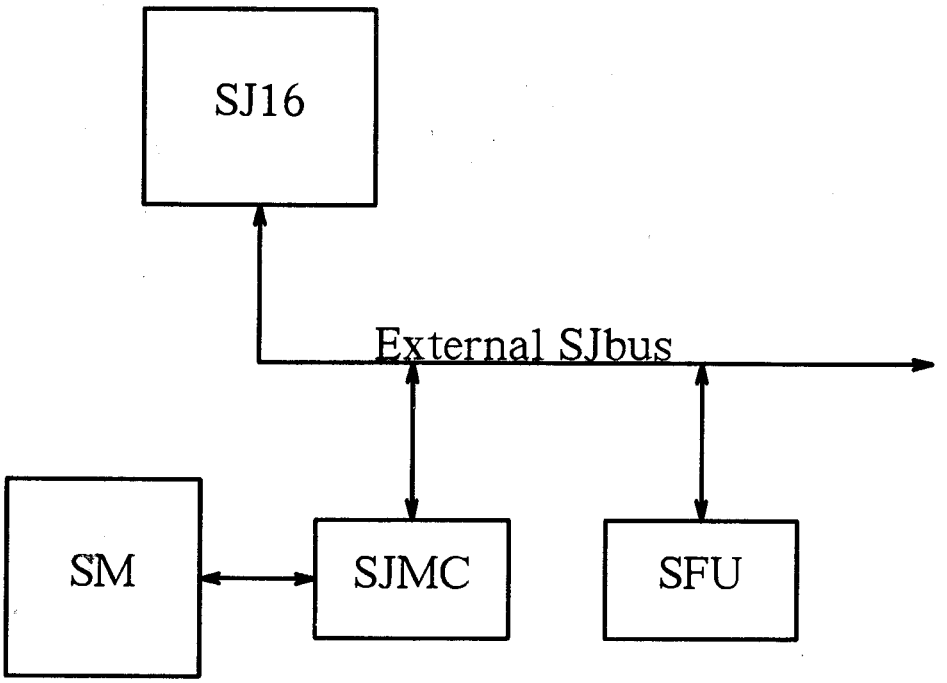

Figure 2-2: A SAMjr system

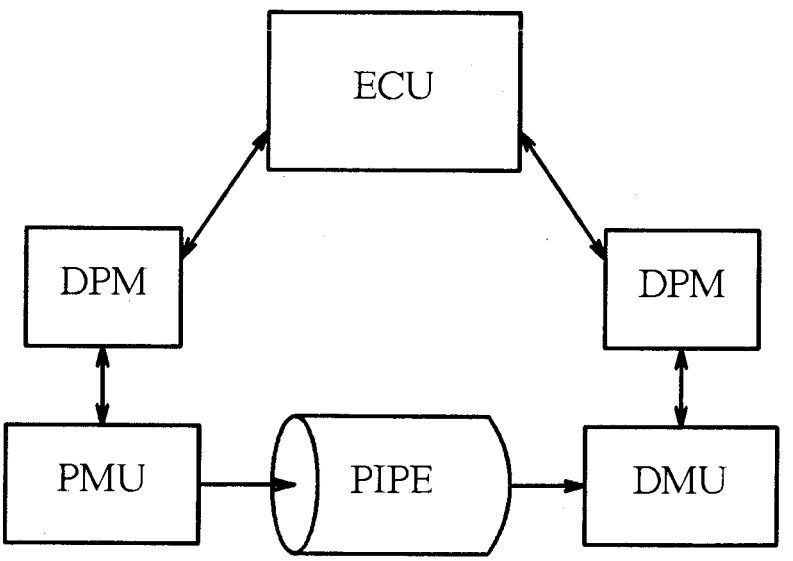

Figure 2-3: A Complete SAM System

workstation (a NEC Advanced Personal Computer), and the only SFU is a Weitek based floating-point processor connected to the SAMjr which is acting as the DMU.

2.1. The ASP Machine Model and Microcode Support Facility

The ASP (Architecture Support Package) system consists of a set of APL workspaces which interpret, measure the performance of, and compile microprograms written in "MicroAPL", a subset of APL plus a set of SAMjr hardware commands. A MicroAPL program is both a valid APL program and a microassembler program for the SAMjr.

2.1.1. Interpretation

Interpretation of MicroAPL programs is handled by an ordinary APL interpreter. MicroAPL commands such as indexing, branching, and assignment are simply the corresponding APL primitives. SAMjr registers, buffers, and memories are modelled by APL arrays. SAMjr microoperations are represented by APL functions. For example, the function SSN operates on the data structures representing the SAMjr segmented memory, buffers, and registers exactly the same way the compiled microprogram "Segment Source Next" is intended to operate on the hardware devices. The next value from the specified segment is placed in the specified register, and the stream buffers are inspected to see if a new memory read operation must be started.

Operations which are to be performed in parallel on the hardware are written on the same line of the MicroAPL function, delimited by the Δ symbol. The present version of ASP does not encode parallel operations except as specifically written by the programmer, and does not reject lines of code which cannot be run in parallel on the hardware.

2.1.2. Measurement

```
    ∇ ALUOUT ← ABUS PLUS BBUS
[1]   STROGEN
[2]   ALUMODE ← ARITHMETIC
[3]   RESULT ← ABUS + BBUS
[4]   ALUOUT ← RESULT + ABUS SETFLAGS BBUS
    ∇
```

Figure 2-4: The PLUS Microprogram

The ASP interpreter can keep track of the number of microcycles used by a program. Every model of a microoperation includes a STROGEN statement which sets some flags to simulate a one level pipeline and increments a clock unless the operation is on the same line of code as the preceding operation (and therefore to be executed in parallel on the hardware). The PLUS microprogram in Figure 2-4 is a good example. Here, line [1] calls STROGEN, line [2] sets the ALUMODE flag, line [3] mimics the hardware addition and sets global RESULT, and line [4] calls SETFLAGS which checks the result and both buses for hardware conditions such as overflow and carry.

This basic clocking scheme has been augmented in various ways. Since a memory cycle takes two machine cycles, the instructions dealing with memory access set a delay flag so that the clock will be incremented an extra tick if the result of a read operation is used before it would be physically available. Trace flags can be set to print out the names and times of functions being executed. And since the model is simply a set of APL programs, arbitrary modifications may be made at any time. At various times, code was added to keep track of ADEL instructions executed, display the contents of the pipe, or suspend execution of selected formats.

Simulation and measurement for the present version of the interpreter were deliberately

done on a version of ASP that models only the SJ16 processors, the SJMC memory controllers, the segmented memories, and the pipe. Facilities for modelling the ECU, the dual-port memories connecting the ECU to the SAMjr systems, and the floating-point processor are being developed separately.

2.1.3. Multitasking

Ideally, the ASP system should run a separate task for each processor being modelled. An earlier version ran as consecutively loaded tasks on MTS APL, requiring manual intervention every time control passed from one processor to the other. Some experiments were done using multiple tasks in Dyalog APL, communicating via a UNIX pipe. The version used in this part of the project works by keeping two complete copies of the data structures. When control passes from one processor to the other, all of the data structures pertaining to the individual processors are renamed. For example, the general register array is named R: when control passes from the PMU to the DMU, R is renamed to PMUR and then DMUR is renamed to R. Since both processors use the pipeline and a set of dual-port registers, the arrays modelling the pipeline and the dual-port registers are not renamed. On every occasion when one processor must await the results of some operation on the other, its clock is set to the timing as of the end of the wait.

2.1.4. Compilation

Valid MicroAPL programs are compiled to SAMjr microcode by a compilation program contained in another APL workspace. The compiler is stricter than the simulator about the inclusion of APL code that is not valid MicroAPL. Compilation therefore provides a useful check on the validity of microprograms.

For reasons of flexibility, the interpreter has made heavy use of named constants, which

are not included in the current definition of MicroAPL. For example, the dual-port register containing the stack pointer is named STKPTR, and only in a single initialization is it mapped to a particular integer. Therefore, the interpreter cannot be compiled in its present form. It will be a straightforward job to transform the source code into a form where such globals are hard-coded as integers, but this job has not yet been done. Possibly it will be feasible to extend the definition of MicroAPL (and alter the compiler) rather than add another step to the translation process.

2.1.5. Modules

APL does not provide a simple way to partition an application into separate chunks of code larger than individual programs. Initially, the entire model was kept in a single APL workspace, ASPSJ16. Now the memory manager, pipeline routines, executable microcode for format and operation syllables, and symbol table management functions are kept in separate workspaces.

All of these workspaces must be copied into ASPSJ16 to be executed. The only advantage to keeping them separately is that maintenance is easier. A set of utilities had to be written to keep track of where functions in the combined workspace belong, and whether they have been modified since the last time they were copied out.

2.2. The ADEL Intermediate Language

The intermediate language used by the SAM APL project is the ADEL (A Directly Executable Language) developed by R. F. Hobson [Hob82, Hob84]. ADEL is a format-based language related to the FORTRAN DEL of Flynn and Hoevel [FIH83, Fly80]. An ADEL instruction consists of a "format" syllable which specifies the valence of the function, zero or more operand syllables, and zero or more function syllables. For example:

$X \leftarrow Y + Z$

becomes

DLR X Y Z +

and

$A \leftarrow B +.\times C \div D$

becomes

SLR C D ÷

DL.S A B + ×

APL code is translated into ADEL in a context which includes a local symbol table. If Y is the 16th identifier used in a given function, then the operand syllable for Y would contain the value 16. All syllables are eight-bit bytes, so a given user-defined function is restricted to using 255 identifiers, excluding comments and primitives. Note that this does not restrict the number of global symbols, as long as no particular function uses more than 255 of them.

DLR	DSR	LLR	SLR	SSR	SLR	SSR
DLS	DSS	LLS	SLS	SSS	SLS	SSS
DL.R	DS.R	LL.R	SL.R	SS.R	SL.R	SS.R
DL.S	DS.S	LL.S	SL.S	SS.S	SL.S	SS.S
DLFR	DSFR	LLFR	SLFR	SSFR	SLFR	SSFR
DLFS	DSFS	LLFS	SLFS	SSFS	SLFS	SSFS
DLMR	DSMR	LLMR	SLMR	SSMR	SLMR	SSMR
DLMS	DSMS	LLMS	SLMS	SSMS	SLMS	SSMS
DLUR	DSUR	LLUR	SLUR	SSUR	SLUR	SSUR
DLUS	DSUS	LLUS	SLUS	SSUS	SLUS	SSUS
D:R	L:R	S:R	D:S	L:S	S:S	L[R]
D:MR	L:MR	S:MR	D:MS	L:MS	S:MS	L[S]
DR	DS	SR	SS	SR	SS	L[]
DUR	DUS	SUR	SUS	SUR	SUS	S[R]
DFR	DFS	SFR	SFS	SFR	SFS	S[S]
DMR	DMS	SMR	SMS	SMR	SMS	S[]
→ N	→ R	→ S	→	∇	Comm	;R
→ N:	→ R:	→ S:	→ :	∇:	R	;S
D ← R	S ← R	D ← ← R	[R]	[S]	SERR	;
D ← S	S ← S	D ← ← S				

Table 2-1: ADEL Formats as of 1984

(See Table 2-2 for Meanings of Characters)

Char	Meaning	Example
D	Destination	DLR A B C + means A is the destination for the result of B + C
L	Left Arg	LLR A B + means A is the left argument. Since an "L" is in "D" position, A is also the destination.
R	Right Arg	Similar to Left Argument
S	Stack	SLR A B +...result is stacked DSR A B +...A gets (stack)+B
.	Product Op	DL.R A B C + × means A gets result of B +.×C
S	Stack Assignment	SLR A B + assigns the result of A + B to the location computed in the preceding sequence of array indexing instructions
F	Derived Fn	DFR A B +/ means A gets the result of +/ B
U	User-Def. Fn	SLUR A B FOO means the user-defined function FOO is run with arguments A and B, and the result is stacked
M	Mixed Fn	DLMR A B C , catenate is a mixed fn
:	Sequence	D:R 3 A B C D E + means A gets result of B + C + D + E Likewise, D:MR 3 A B C D E ,

Table 2-2: Meaning of Format Characters

The ADEL formats defined in [Hob84] are given in Table 2-1. The characters used to describe the formats are defined in Table 2-2. Note that the set of formats is not rigid. Several formats have been added during the development of the front-end translator, and others were added to handle data transfer and program control. The set of formats is intended to be tailored to fit particular applications. For example, a crosstabulation package might reserve a special format for the +.= inner product, and a language extension to include one of the nested array systems would have different operator formats.

Not all of the formats recognized by the translator have been implemented. Table 2-3 contains a list of implemented formats. Since the formats have been implemented as APL functions, it has been necessary to rename them.

FORMAT	ADEL	DESCRIPTION
NOOP	NOOP	No operation
DLR	DLR	Dyadic Scalar Fn, Explicit Arguments and Result
SLR	SLR	Dyadic Scalar Fn, Explicit Args, Stack Result
LLR	LLR	Explicit Args, Dest = Left
DMR	DMR	Monadic Mixed Fn
SMR	SMR	Monadic Mixed Fn, Stack Result
SLUR	SLUR	Dyadic User Defined Function, Explicit Args, Stack Result
SSUR	SSUR	Dyadic U.D.F., Stack L.arg & Dest
SLUS	SLUS	Dyadic U.D.F., Stack R.arg & Dest
SSUS	SSUS	Dyadic U.D.F., Stack Args & Dest
COMENT		Comment
DFR	DFR	Monadic Derived Function
DLMR	DLMR	Dyadic Mixed Function
DLMS	DLMS	" " " , Stack R.arg
SLMR	SLMR	" " " , Stack Dest
DGETSR	D ← R	Explicit Assignment
DGETSS	D ← S	Assignment from Stack
ICTRBZ	new	Initialize Counter, Branch if Zero
INDAS3	new	D[L] ← R
SUR	SUR	Monadic U.D.F., Stack Dest
BIFSTK	new	Branch to Specified Line If Stack =/= Zero
BRABS	→ R	Branch to Specified Line
DECBNZ	new	Decrement Counter and Branch to Specified Line if =/= Zero
NILUFN		Niladic User-Defined Function
NXTL	→ N	Branch to next line
HALT		Stop the Simulator

Table 2-3: Formats Implemented as of January 1987

Chapter 3

The Memory Manager

The first four months of this software project were devoted to developing the memory management subsystem. About five percent of the subsequent work has consisted of keeping track of the information required for the memory manager to allocate space for results and clear out the space used by reassigned variables, popped stack values and expired immediate execution expressions.

Since APL objects are dynamically sized and APL functions are recursive by definition, the execution of an APL program involves allocating and freeing space for arrays both during function call/return and during the execution of primitive functions. The management of memory in the PMU is somewhat less critical, since functions are generally created under immediate execution and are seldom erased. For the sake of consistency, the same memory manager is used by both processors.

3.1. Background

Memory managers are an important part of multiprogramming operating systems and dynamic language execution environments. As such, they are the subject of a considerable body of theoretical treatment. The descriptions of the basic problem, paged and segmented memories, and fitting strategies are based on [PeS83, Dei84].

3.1.1. The Basic Memory Management Problem

An operating system may allow programs to be relocateable, thus allowing memory to be partitioned among several programs. Programs can easily be swapped in and out of fixed-sized partitions. Allowing variable partition sizes raises the basic memory management problem of allocating space in the most efficient way. Memory may become "fragmented", so that free memory is distributed among a set of noncontiguous blocks. It then becomes necessary to choose which free block will be used for a given job, and it may also be necessary to relocate existing jobs in order to merge free blocks.

3.1.2. Paged Memory

One powerful way to deal with the allocation problem is to waive the requirement that a program or a data object be contained in a contiguous chunk of memory. If memory is divided into "pages", and logical addresses consist of a page number and an offset, then a page table can be used to map page numbers into real memory addresses.

The principal differences in simple paged memory systems concern the implementation of the page table: if kept in registers, the table is of limited size; and if kept in main memory, it slows each memory access by a factor of two. The slowdown can be worse in a pipelined machine, since the second memory access cannot be overlapped with the first. A compromise solution is to keep a subsection of the page table in content-addressable registers The SAM architecture keeps the entire table in high-speed static memory

3.1.3. Segmented Memory

In a paged system, pages are of fixed size, so that large objects may occupy several pages and several small objects may share a single page. The segmented memory scheme partitions physical memory into variably-sized segments, with one object occupying one segment. While segmented memory is a more natural way to deal with objects of varying size, it is naturally subject to fragmentation and requires the same kind of allocation scheme as variably sized partitions in a job scheduler.

A segmented memory may be paged, so that a segment need not be composed of contiguous real memory. This usually faces all memory accesses with two levels of indirection, and must be implemented with registers or cache to give reasonable performance. A paged segmented memory can be implemented with a single level of indirection by keeping physical page addresses in the segment table. This is the approach used by the SAM memory controller. The only major drawback to one- level paged segmented memory is that it makes it somewhat more difficult to implement a virtual memory that swaps individual pages.

3.1.4. Fitting Strategies

Given a segmented memory or a variable-partition-size swapping scheme, there will be a list of free blocks to be considered when a new block is to be allocated. The two best-known methods of choosing a free block are first-fit (choosing the first free block big enough to satisfy the request) and best-fit (searching the whole list and choosing the smallest free block big enough to satisfy the request).

In order to make a best-fit strategy perform well, it may be necessary to keep the free block list in a structure amenable to a binary search. If the free list is kept sorted in increasing order of size, the two strategies become identical.

Since searching time grows with list size, it may be advantageous to garbage collect before an unfillable allocation request arrives. For example, the implementers of CDC APL*STAR found that limiting the "Hole Table" to 16 entries caused the proportion of garbage collects triggered by hole table overflow to be much smaller than the proportion triggered by memory fragmentation [Edw87]. Another approach is to ignore freed blocks completely unless they can be merged with the "one big hole", speeding up typical allocations at the expense of more frequent garbage collection.

3.1.5. Buddy systems

It is possible to eliminate the linear search of free block lists, by keeping a set of lists of free blocks of various sizes and looking only in the appropriate list for a given allocation request. Such a system was developed by Knowlton [Kno65] and has since been the subject of continual investigation [PuS70] and development [PaH86].

The basic idea of these "Buddy Systems" is that only a fixed set of block sizes is available. If no free block exists in the size requested, then a bigger block is split, with the left-over part ("buddy") put into the appropriate free block list. When a block is deallocated and its "buddy" is still free, the two blocks may be merged. Buddy systems thus avoid the overhead of list searches, at the expense of block splits and recombinations and of a considerable amount of memory fragmentation, both "internal" (due to the coarseness of the set of block sizes available) and "external" (since the splitting process may leave the free memory as a set of blocks too small to satisfy an allocation request).

The total fragmentation of a buddy system is typically thirty or forty percent [PeN77] of total memory. Given a demand for quick allocation and the availability of cheap memory, this overhead is tolerable as long as the buddy system outperforms the more memory-efficient alternatives.

3.2. The SAM Segmented Memory Model

Any memory management scheme must reflect the hardware support available. For instance, a demand paging system would be awkward to implement on a machine with no page fault interrupt mechanism. The memory manager developed here is based on the paged segmented memory controller forming part of the SJ16 architecture. (See Figure 3-1.)

The most important feature of the model (See Figure 3-2) is the address translation table called the SPT, for Segment Page Table. A logical address consists of a Segment number and an Offset. The logical page number is found by adding the segment number to the integer part of (Offset ÷ Pagesize), and the physical page is given by SPT[Logical page]. The rest of the offset is a byte index into the physical page. A data structure may be distributed in widely scattered physical pages, as long as these pages are pointed to by a contiguous block of the SPT. While it would be possible to keep more than one small object in a single page of segmented memory, "workspace full" problems in existing APL systems are almost always caused by a small number of large objects. In the SAM system, one thousand small arrays would use one thousand pages, leaving at least three thousand pages for larger arrays.

The result of this policy is that no program "cares" where in physical memory a given object resides. When garbage collection is necessary, it is not because physical pages are misplaced, but because there is not a large enough contiguous block of free pointers in the SPT, and the corrective measure involves moving single pointers instead of complete pages of data.

While this memory scheme is in fact a paged segmented memory, the fact that all the

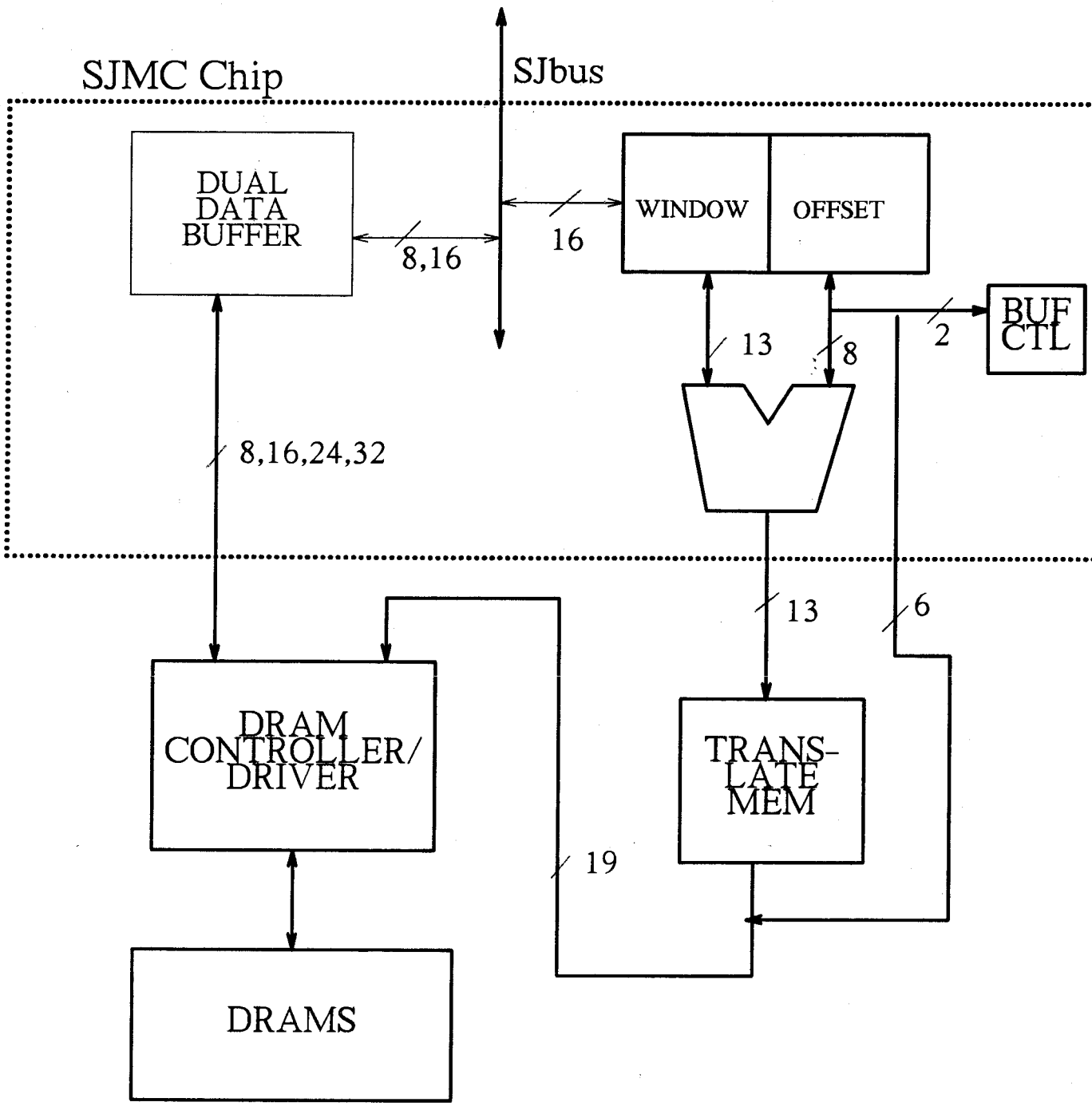

Figure 3-1: The SJMC Memory Controller

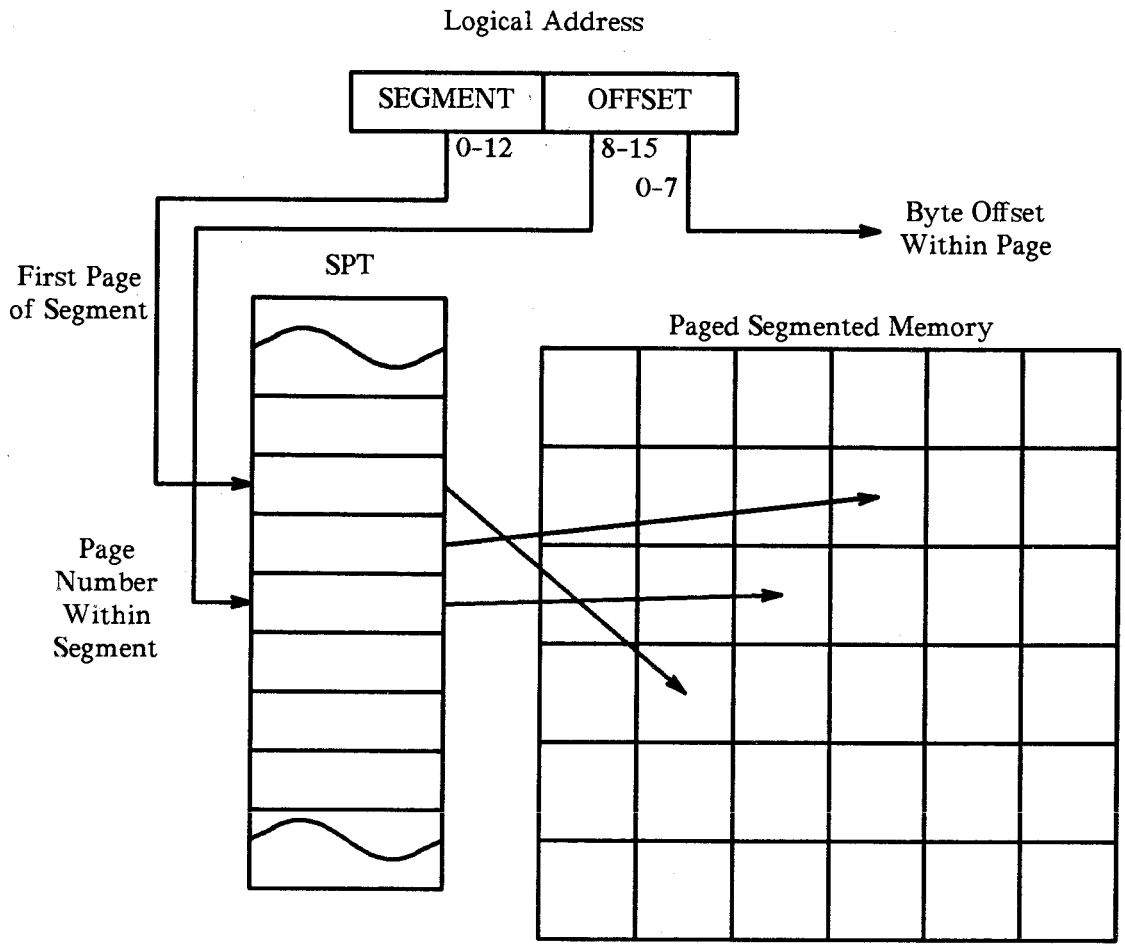

Figure 3-2: The SAM Memory Model

pages of a given segment are kept in a contiguous block of the SPT allows memory accesses with one level of indirection instead of two, at the expense of the page table garbage collect described above. Keeping the page table in fast static memory would slow each access slightly compared to the access time for a "hit" where a subset of the page table is kept in registers. However, in the SAMJr Memory Controller, this address translation can be done during dynamic RAM precharge time, so no performance is lost. This scheme gives the performance of a register-based page table where every access is a "hit" without the special hardware and complex microcode required to deal with "misses".

The other interesting feature of the memory model is that segmented memory is streamed. Once a given address in segmented memory is accessed, the value stored at the next location is read into a buffer and is available to the processor in advance. Memory writes are done to a buffer which is flushed into memory in parallel with other operations. This policy accelerates string and array processing tasks. The tradeoff for any purely array-oriented memory is that scalar accesses require more code than the corresponding accesses on a scalar machine (See Figure 3-3).

The streaming of scalars is handled in microcode by doing the preliminary memory control operations in parallel with other operations. This allows the system to access scalars efficiently without adding hardware and extending the instruction set to include conventional scalar operations.

WDO[Strid] ← R[A]	Set stream "Strid" to the segment specified by R[A]
Strid SRW R[B]	"Segment Read Word": Start stream in word mode at offset specified by R[B]
SBF Strid	"Segment Buffer Fill": Fill the stream buffer for continuous reads. For a scalar read, this instruction could be replaced by an IDLE.
R[C] ← SSN Strid	"Segment Source Next": R[C] gets the next word from the specified stream

Figure 3-3: Streamed Segmented Memory Access

3.3. The Present Model

The memory manager used by the present interpreter is very nearly the simplest possible manager using the architecture support available. The memory manager table (see Figure 3-4) consists of one doubly linked list of free blocks, with each list item containing a block size and each allocated item containing a pointer back to the main data access table. Allocation begins with a first-fit search of the free block list. The block is then either shrunk (if it is more than big enough) or removed from the list (if it is an exact fit). Adjacent free blocks are merged immediately at deallocation time. Garbage collection is tedious but reasonably fast, since only SPT items must be moved. A garbage collect is terminated as soon as a free block big enough to satisfy the current allocation request has been created.

A commented call tree of the implemented memory management functions is given in Figure 3-5.

It seems likely that garbage collection can be made to occur arbitrarily seldom, by keeping a table somewhat bigger than the actual size of the memory. A practical measurement of this prediction has been deferred.

3.4. The Production Model

Before the microcode was written for the current version of the memory manager, a slightly more complicated manager was written in APL. The production memory manager may be based on this earlier version, which used two linked lists instead of one. Free blocks containing a single page are kept in one list, and multiple-page blocks in the other. Since (again, pending measurements) the large majority of data structures will fit into single pages, it is expected that this system will outperform more complex systems such

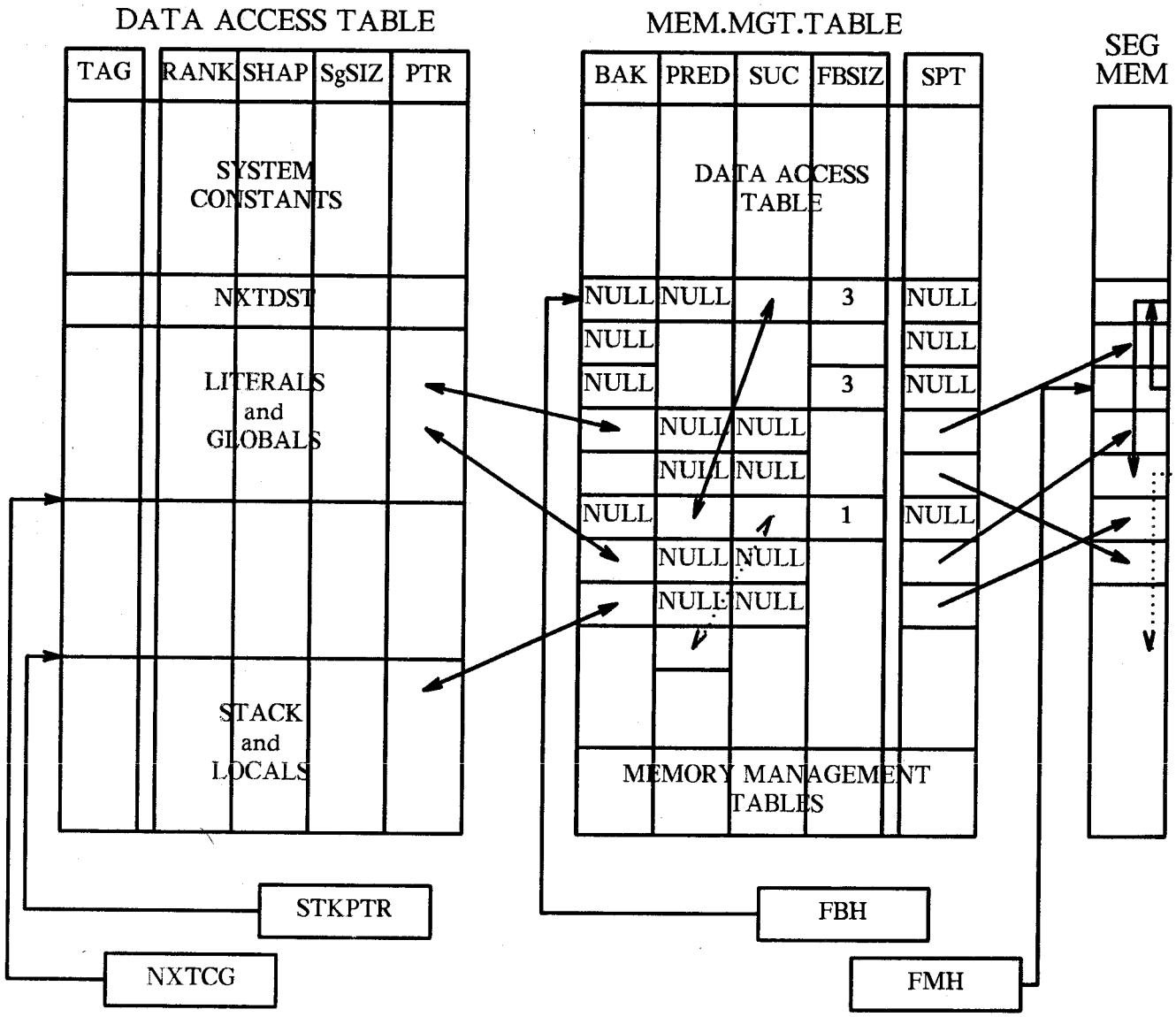

Figure 3-4: Memory Management Tables

```
ALLOCATE          RESERVE N PAGES
  ALLOC_PAGES     GET PAGES FROM FREE PAGE LIST
  ALLOC_WHOLE     FREE BLOCK = SIZE REQUIRED
  ALLOC_PART      TAKE CHUNK FROM BOTTOM OF FREE BLK
FREE              RELEASE N PAGES BEGINNING PAGE P
  FREEPAGES       RETURN PAGES TO FREE PAGE LIST
  MERGE_ABOVE     BLK BELOW IN USE, BLK ABOVE  FREE
  MERGE_BELOW     BLK ABOVE IN USE, BLK BELOW  FREE
  MERGE_BTWN      BLKS ABOVE AND BELOW BOTH FREE
  MERGE_SOLO      ADD ISOLATED BLK TO FREE LIST
GARBCOL           MERGE ALLOC BLKS TO ENLARGE HOLE
SETMMREGS         SET UP REGS FOR MEM MGR
INITMEMM          SET UP SJ16 SM, SPT, and MM TBLS
```

Figure 3-5: Memory Manager - Commented Call Tree

as a buddy system. Eliminating the common one-page blocks from the searches for larger blocks should improve search times by a factor of better than two and reduce fragmentation.

3.5. Theoretical Predictions

While a realistic appraisal of performance must await measurements made with real programs running on a completed implementation, it is not difficult to give a rough but reasonable theoretical analysis of various allocation strategies. In this section, we look at the current single-list strategy, the projected two-list strategy, and a simple binary buddy system. For the sake of simplicity, we will consider only the costs of allocation.

Consider only requests and free blocks larger than a single page. Then the number of blocks searched for a typical request depends on the distribution of sizes in both the free block list and the set of requests. Models based on various distributions gave expected search lengths of 3.5 to 6.0 iterations with an infinitely long free block list and from 3.0 to 3.5 with the free block list limited to 16 elements. For the sake of simplicity, an expected search of 3.5 iterations will be assumed for the rest of this analysis.

Let Pm be the probability of an allocation request being for more than one block. The distribution of free block sizes may differ from the distribution of allocation request sizes due to merges (which lead to larger blocks) and partial allocations (which give smaller free blocks). In particular, the deallocation of a single-page object gives a single-page free block only when neither of its neighbours are free. For the sake of simplicity, assume that this happens about half the time, so that the proportion of multipage free blocks will be 1 - 0.5(1 - Pm), or 0.5(Pm + 1).

3.5.1. Single Free Block List

Using a single list for all free blocks, a single-page allocation will be satisfied by the first block on the list. A multipage allocation will search about 3.5 multipage blocks. Since 0.5(Pm + 1) of the free blocks are multipage blocks, the allocation will search 7/(Pm + 1) blocks. Then a proportion Pm of the allocation requests require 7/(Pm + 1) search iterations, and (1-Pm) of the requests require 1 iteration, giving an expected average of

(1 - Pm) + 7Pm/(Pm + 1) search iterations.

3.5.2. Two Free Block Lists

Using a pair of free block lists, all allocations require a block size check to determine which list is to be searched. Then the multipage allocation requests will search 3.5 free blocks, giving an expected performance of 1 check and (1-Pm) + 3.5Pm or (1 + 2.5Pm) search iterations.

3.5.3. Buddy System

Assume that freed blocks are NOT merged if doing so would give an empty list for a given block size. An allocation will require (log N) checks to choose among N different block sizes by a binary search. (Block sizes are distributed logarithmically, so in principle the correct list can be found by calculation. Unless the hardware supports binary logarithms, it is quicker to do a depth-three binary search.)

For a SAM APL interpreter, N would be about 9, with the checking biased so the first check is for a single block. A multipage allocation will take 3 additional checks to choose among the 8 larger block sizes. Then the average number of checks is 1 + 3Pm at allocation time. One additional check must be made to ensure that the required list is not empty, giving 2+3Pm checks.

There will be a probability Pe that the list is empty, requiring a split. The list to be split may itself be empty, leading to an expected number of splits of Pe/(1-Pe) for a given allocation. The value of Pe is dependent on the recombination strategy. For the purposes of comparison, values of 0.1 and 0.25 will be used.

3.5.4. Comparing the Systems

A search iteration requires one segment startup, two reads, and a comparison, taking altogether about four cycles. (The microcode used in this search is listed in Figure 6-6). A simple comparison requires a test and a branch, costing at least two cycles. Then the average cost of finding a free block to be allocated is shown in Table 3-1 A rough pass at microcoding a split gives about 10 cycles if all list headers are kept in registers, or 16 cycles if headers are kept in memory.

System	Checks	Searches	Splits	Cycles
1-list	0	1-Pm +7Pm/(Pm+1)	0	18-4Pm
2-list	1	1+2.5Pm	0	6+10Pm
Buddy	2+3Pm	0	Pe/(1-Pe)	4+6Pm +16Pe/(1-Pe)
Pm = 0.1				
1-list	0	1.54	0	6.16
2-list	1	1.25	0	7
buddy1	2.3	0	0.11	6.36
buddy25	2.3	0	0.33	9.88
Pm = 0.2				
1-list	0	1.97	0	7.88
2-list	1	1.5	0	8
buddy1	2.6	0	0.11	6.96
buddy25	2.6	0	0.33	10.48
Pm = 0.3				
1-list	0	2.32	0	9.28
2-list	1	1.75	0	9
buddy1	2.9	0	0.11	7.56
buddy25	2.9	0	0.33	11.08

Note: Buddy1 is Pe = 0.1, Buddy25 is Pe = 0.25

Table 3-1: Cost of Finding a Free Block

This simplified analysis suggests that the one-list model may perform as well as the two-list version as long as the proportion of multi-page allocation requests is not more than thirty percent. The buddy system does not outperform the linked-list systems unless the proportion of empty lists can be kept to ten percent.

Chapter 4

Symbol Table Management

The three main management tasks of an APL interpreter are to store APL arrays, to provide a convenient way for executing APL code to access arrays and functions, and to provide a way for a user to get arrays and functions in and out of the system. Each of these tasks is organized around its own table: the Data Access Table for array storage, the Contour Access Table for access by executing functions, and the Global Symbol Table for user access.

4.1. The Linker

The linker is responsible for accepting ADEL functions from the front-end system, writing them into segmented memory, and installing the identifiers used by the functions into the Global Symbol Table.

4.1.1. The User Interface

The user interface to SAM APL is a text editor and translator which run on the workstation that serves as the ECU. To write an APL program, a user edits the text form and invokes the TRANSLATE facility. This produces the ADEL code corresponding to the APL source, plus a local symbol table and enough information to link it.

In a complete SAM system, this internal function form (See Figure 4-1) is then written into dual port memory by the ECU, to be accessed by the linking routine in the PMU. At

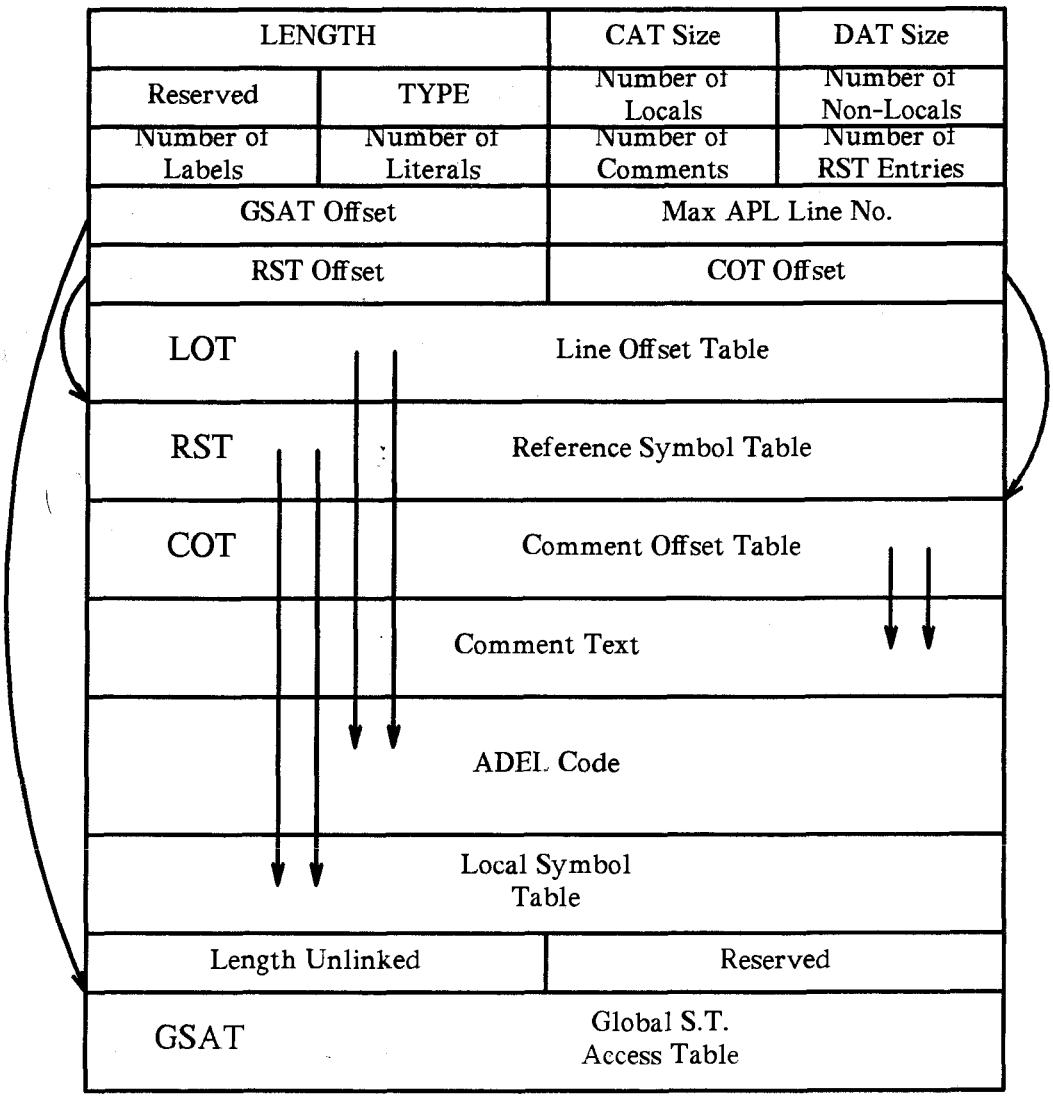

Figure 4-1: Function Internal Form

the current stage of the model, an ADEL assembler on the computer system supporting the ASP model is used to create an APL variable containing the internal function form. Some of the ADEL functions were written by hand instead of generated by translation of source code. This was done because some of the ADEL formats developed during this project are not yet within the scope of the translator. These include the special branching and looping primitives and the simplified indexed assignment format.

Once the ADEL function has been created, the LINK program places it into segmented memory, creates symbol table entries for all of the objects in the program, and creates data arrays for all of the nonscalar literals in the program. The Global Symbol Access Table (GSAT) shown in the diagram for the internal function form is created at link time and added to the function segment by the linker.

To run a line of APL immediate execution (and therefore, to start an APL program), the assembler is used to create a dummy function which is flagged as an immediate execution line in the "Type" field. LINK invokes the execution manager after linking an immediate execution "function", then purges the function from segmented memory after execution is complete. A list of the programs used by the linker is shown in Figure 4-2.

The present procedure for testing APL functions is necessary while the editor and the rest of the model run on different computers without a convenient way of transferring data. In the working SAM system, the front-end computer will place the translated function into dual port memory and the LINK program will read dual-port memory instead of taking an argument.

Changes needed to adapt the LINK program to the finished hardware are minimal, so performance, reliablility, and space figures for the production version of the linker are expected to resemble those of this model very closely.

```
LINK                        HALF THE SAM FRONT END
  BRINGINFN                  NOT uCODE.READ FN INTO PROG SEG
  RUN_SAM                    THE BOSS PROGRAM FOR IFETCH/IEXEC
  STLINKFN                   LINK PGM IN PROG SEG TO S.T.
    CHKNUMV                   PROG IS AT START OF NUM LIT
    STLINKHDR                 BIND OBJS IN FN HDR
      STLINKFNAME              FN NAME UNLIKE OTHER OBJS
        FINDSTPAGE              S.T. PAGE IS NOT EMPTY
          * * *
        HASH_GNRL               HASH ID, LBL, LITERAL
          * * *
        STENTWR                 WRITE S.T. ENTRY
      STLINKID                  LINK 1 ID INTO S. T.
        * * *
    STLINKID                   LINK 1 ID INTO S. T.
      CHKNUMV                   PROG IS AT START OF NUM LIT
      FINDSTPAGE                S.T. PAGE IS NOT EMPTY
        GETCONFPG                GET NEW S.T.CONFLICT PAGE
        STMATCHID                ID AT(PROG,IDOFF)=(STS1,14)?
      HASH_GNRL                 HASH ID, LBL, LITERAL
        HASH_IDL                 HASH ID OR LBL
          HASHMODULO
        HASH_NUML                HASH NUM LITERAL
          HASHMODULO
        HASH_QUOTE               HASH QUOTED LITERAL
          HASHMODULO
      SENDTODMU                 SEND LITERAL TO DMU
      STENTWR                   WRITE S.T. ENTRY
```

Figure 4-2: The Linker - Commented Call Tree

4.2. The Global Symbol Table

4.2.1. Symbol Table Pages

Every identifier in an internal-form function corresponds to an entry in the global symbol table. A sample symbol table entry is shown in Figure 4-3.

The linking process for each identifier consists of hashing the identifier and checking the corresponding symbol table entry. If it is free, the identifier is written into it; if it contains

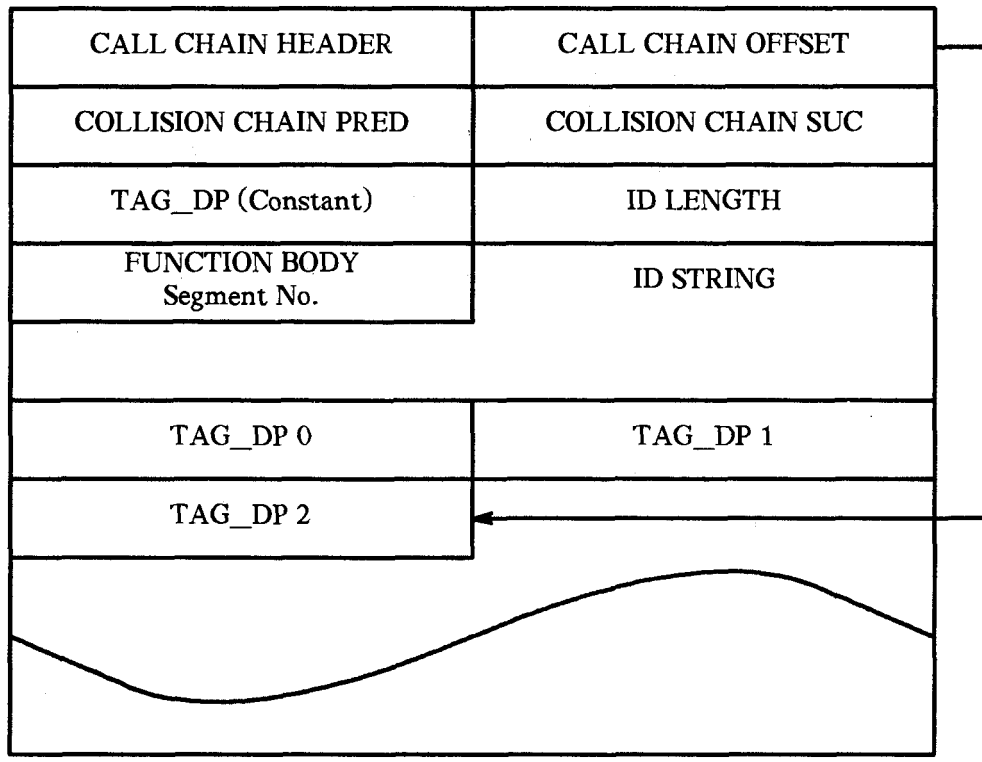

Figure 4-3: Symbol Table Layout

an identical identifier, linking is complete; and if it contains a different identifier, the collision chain is followed until either an identical identifier is found or the end of the chain is reached. If no match is found, the identifier is written into an overflow symbol table page and the new page is linked onto the collision chain.

Once a symbol table entry has been found or created, a pointer to it is written in the internal function segment.

In the usual state of affairs for linking, the system is in immediate execution with no suspended programs. The call stack pointer in the symbol table chain will be pointing to the DMU pointer for the global value, if any. The procedure for function call will be discussed in the following chapter. For now it is sufficient to mention that the linking process is not affected by the state of the call chain. Of course, a newly linked identifier will be tied to the DMU item containing the current value.

4.2.2. Literals

When a numeric or character literal is found in a source program, it becomes an identifier in the ADEL code, mapping to a symbol table entry that can not be hidden by function call. Thus, if several programs happen to use the value 3.6, they will all map to the same symbol table entry. When a literal is linked, its value is sent immediately to the DMU, and a DMU item pointer for the literal value is written immediately into the symbol table page.

The exception to this is the integer scalars 0 through 255. The first 256 DAT entries are reserved for these values, so the DAT pointer can be written into the symbol table page without transmitting a value to the DMU. A few special values like pi and e are kept in known DAT locations for the use of system code, and future version of the interpreter may recognize the equivalent APL expressions and treat them as literals.

Carrying this idea to its logical conclusion, it is possible to execute all expressions involving no variables at link time and link the results as literals. Statistics on the frequency of execution of such "constant expressions" must be gathered before any decision is made about implementing them as literals.

Ordinarily, the symbol table page contains the character string representation of its identifier. In the case of literals, this representation may be too long to fit into a page. When the string length is greater that 40 characters, the "id" section of the symbol table page contains a pointer back to the local symbol table of the function owning the literal. Therefore, if two separate functions happen to have identical long literals, each will have a separate symbol table entry.

Most identifiers are quite short, and since a symbol table page has room for a call chain which will never exist in the case of a literal, a certain amount of space is wasted. Future versions of the interpreter may keep short literals in smaller symbol table pages.

4.2.3. Symbol Table Garbage Collect

Except in the case of long literals, symbol table entries do not contain backpointers to the functions that use them. This makes it difficult to get rid of symbol table entries when their referents no longer exist. Older APL interpreters (MTS APL is a good example) never attempt to clean up the symbol table: when the symbol table is full, the only recourse is to)COPY everything into a clear workspace. An easy but slow way to deal with a full symbol table is to relink every function and global variable, essentially)COPYing the workspace to itself.

To allow symbol table entries to be deleted efficiently, a reference counter could be included in each symbol table page. Linking an identifier to an existing page would increment the reference count, and expunging a defined function would decrement the reference counts of all of its identifiers, with a page marked as free when its reference count drops to zero. This idea will be investigated when the standard interpreter has been completed.

The collision chain pointers in each symbol table page show the predecessor and successor in a chain of symbol table pages corresponding to identifiers that hash to the same value. The forward pointer is used only when linking a new identifier. The back pointer was intended to be used during symbol table garbage collection. Keeping the collision chain as a circular list makes back pointers unnecessary, and the field may be used for a reference count in the next version.

4.2.4. Labels

Function labels are a special case for linking and calling. Unlike ordinary identifiers, they have values and unlike ordinary literals, they have names that may be localized by subsequent function calls. At link time, labels are linked to symbol pages exactly the same way as ordinary identifiers, and their status as labels is indicated by setting the sign bit of the symbol table page address. The item in a function segment following a negative symbol table pointer contains the label value instead of the pointer for the next identifier.

4.3. Related Environment Tables

The environment of an executing program is partly under direct user control but principally a function of the state of execution. In a lexically scoped language, the environment is determined by the block structure of the main program. In a dynamically scoped language like APL, the availability of an identifier is controlled entirely by the most recently called function which localizes the identifier. Thus, this section must be read in conjunction with the treatment of the CALL and RETURN mechanisms in the following chapters.

4.3.1. The Data Access Table

Data manipulation is done in the DMU. Each data item, whether or not it is available to the current scope of the program, is represented by an entry in the Data Access Table, or DAT, kept in DMU segmented memory. Scalars and array descriptors are contained in the DAT entry The entire descriptor of a vector fits into the DAT entry. The length fields for the trailing axes of higher-rank arrays, and the element values of all nonempty arrays, are kept in DMU segmented memory, each segment pointed to by its DAT entry. When code is actually executed in the DMU, the DAT entries are used by the data manipulation routines.

A physically distinct tag memory contains 4-bit type and shape tags corresponding to the entries in the DAT.

As it is currently laid out, the first 256 DAT entries are reserved for the scalar constants 0 through 255. Then a few entries are reserved for constants like "π", "e", and pointers to read-only segments containing the character set and the results of ⍳256 in both index origins. The next DAT entry is a scratch location indexed by system constant NEWDEST, used to write the result of each primitive operation before erasing the previous value of the identifier being assigned.

The rest of the DAT is divided into two stacks. The stack growing forward from NEWDEST is pointed to by NEXTCG (Next Constant or Global) and contains constants and global values that are never erased except by explicit request. The stack growing backward from the end of the DAT is pointed to by STKPTR, and contains values on the execution stack, local variables, and the arguments and results of functions. NEXTCG and STKPTR are kept as mailbox registers in the pipe processor because both the PMU and the DMU must use them.

4.3.2. The Contour Access Table

The Contour Access Table (CAT) for a given environment consists of a list of words containing syntax tags (see Section 5.2.1) and DAT pointers. When ADEL code is executing, each operand syllable is used as a word index into the current segment of the CAT.

The relationship among internal function forms, the symbol table, the DAT, and the CAT is diagrammed in Figure 4-4

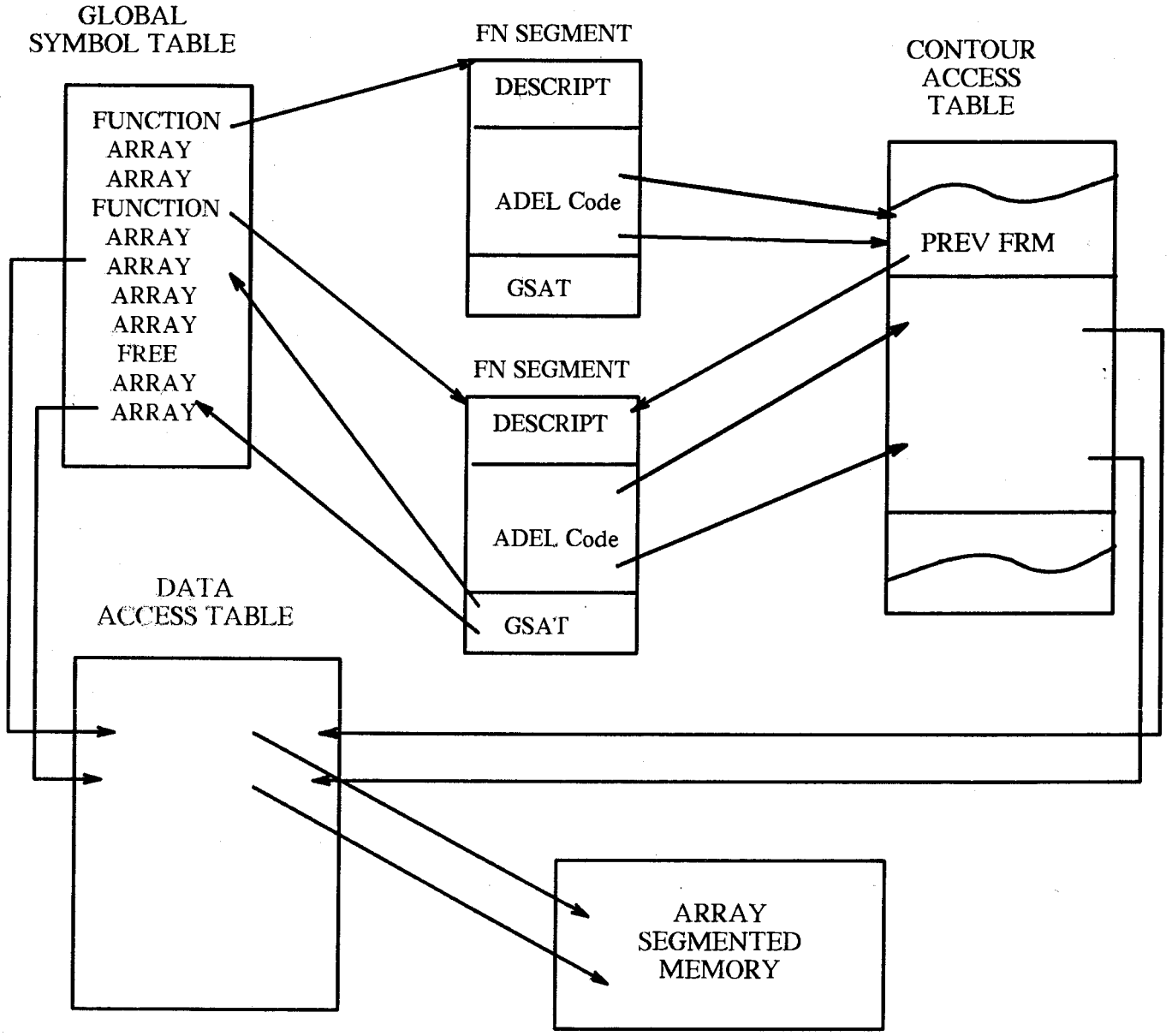

Figure 4-4: Relationship Among Interpreter Tables

Chapter 5

Execution

The paradigm of user interaction with an interpreted language is the Lisp Read-Eval-Print cycle. APL follows this cycle closely, with the important restriction that the user interaction is built into the system instead of being a user-defined program and, as such, modifiable.

The SAM APL user interface is similar to the paradigm. The basic cycle is textedit-execute-print. The details of the editing and printing parts of the cycle are outside the scope of this thesis. The execution part of the cycle begins with linking a line of code as described in the previous chapter. The present chapter deals with the execution of ADEL code once it has been put into program memory and its identifiers have been put into the Global Symbol Table.

5.1. Instruction Fetch

The PMU half of program execution is done by the IFETCH routine. An instruction fetch begins by reading the format syllable of an ADEL instruction. As a side effect, the format is sent to the pipeline. Then the PMU EXECs the format syllable -- that is, executes the microprogram beginning at the address given by the format table indexed by the format. This is equivalent to an instruction decode operation in a standard commercial computer. In the model, this consists of executing the MicroAPL program whose name is in the indexed row of the format table.

Execution of the format microprogram causes the operand and function syllables to be fetched. Syllables denoting primitive APL functions (and also some "hidden primitives" such as *pp* and +/) are sent directly to the pipeline. Operand syllables are used as word indices into the Contour Access Table. The indexed words, which consist of syntax tags and data pointers, are sent to the pipeline. At the end of the format microprogram, the pipe is released if no errors have been detected and the process is repeated as soon as a new pipe becomes available.

Syllables representing user-defined programs are not sent to the pipeline. The PMU microprograms for function call pass the CAT entries indexed by the argument syllables to the pipeline, then do the processing necessary to preserve state information for the current environment and set up a new environment. The details of the CALL and RETURN procedures are described in section 5.4.

5.2. Instruction and Operand Verification

One of the most important features of the SAM architecture is the way instruction verification(syntax checking) and operand verification(preliminary semantic checking) are handled in hardware. A conceptual view of the verification process is shown in Figure 5-1. Each operand is represented in ADEL code as an index into a frame of the Contour Access Table (CAT). This index is conceptually added to a frame pointer (in the current implementation, each frame begins on a page boundary, so the addition is done by the addressing hardware) and the resulting tag/data pointer is read from the CAT. The tags are then used in the instruction verification process and the data pointers are sent to the pipe.

In the DMU, the data pointers are used to index a Data Access Table (DAT). The

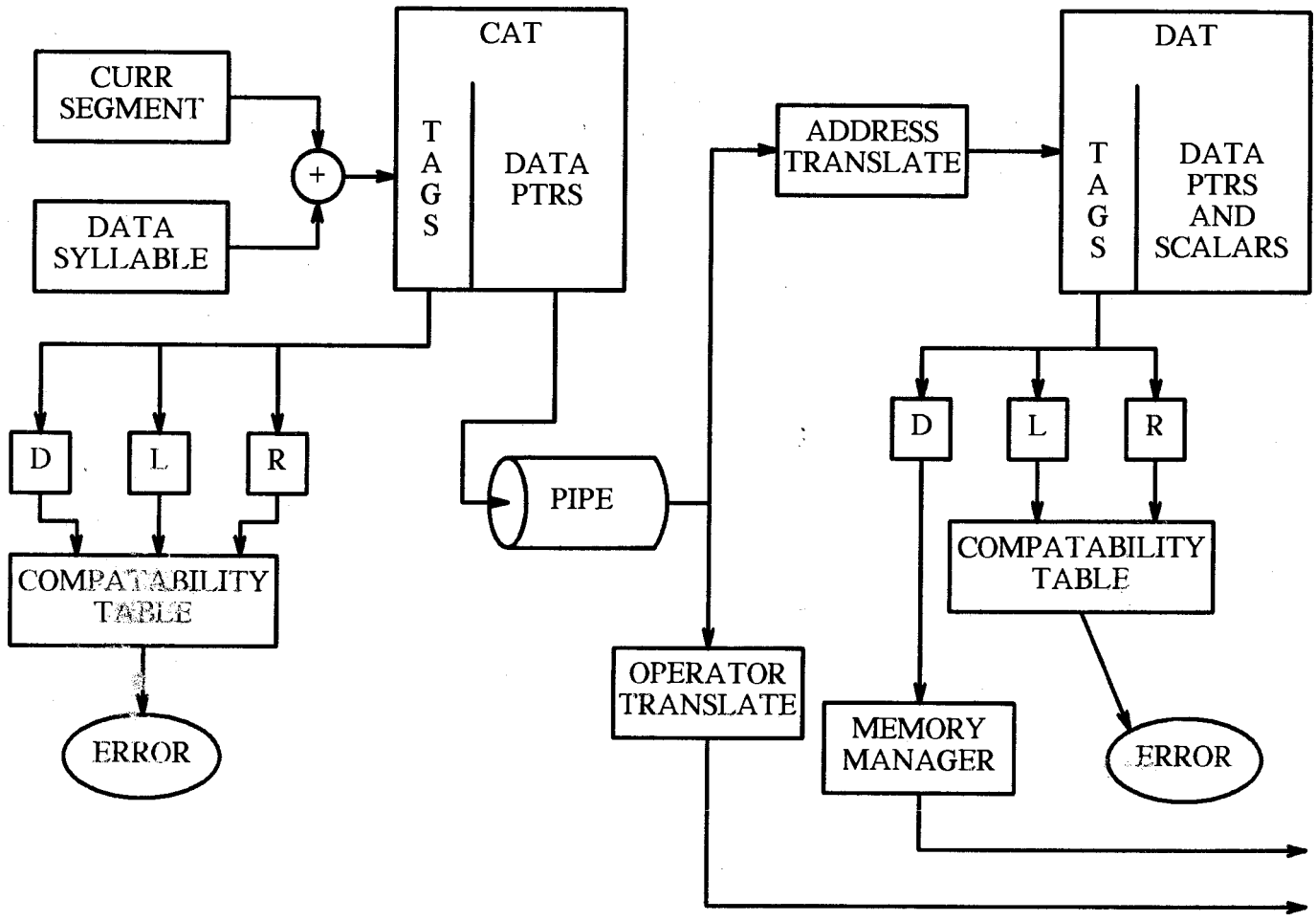

Figure 5-1: Conceptual Diagram of Verification Process

DAT contains data tags, and either the values (of scalars) or partial descriptors(of arrays). The data tags are used for preliminary semantic checking. Finally, the values and/or descriptors are used by microprograms for the format and operator syllables.

5.2.1. Instruction Verification

The Instruction Verify Unit is a compatibility matrix table lookup unit built into the pipeline processor. The tag for each operand is a two-bit value denoting constant, variable, function, or reserved. The table used for instruction syntax verification is shown in Table 5-1.

DEST	VARIABLE	CONSTANT	FUNCTION	OTHER
RIGHT ARGUMENT	v c f o a o u t r n n h	v c f o a o u t r n n h	v c f o a o u t r n n h	v c f o a o u t r n b h
LEFT ARGUMENT				
VARIABLE	1 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0
CONSTANT	1 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0
FUNCTION	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
OTHER	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0

Table 5-1: Syntax Checker Table

When the microprogram executing a format in the PMU is ready to release the pipe, the output is tested. If the result shows that the instruction was syntactically invalid (the compatibility table entry is a zero) an error routine is invoked and the pipe is not released. (At the current stage of the interpreter, the error routines have not been written. When an invalid instruction is detected, the simulator halts and displays an error message.)

RIGHT OPERAND / LEFT OPERAND	NO VALUE	SCALAR	ARRAY	RESERVED
NO VALUE	1	1	1	1
SCALAR	1	0	0	1
ARRAY	1	0	0	1
RESERVED	1	1	1	1

Table 5-2: Semantic Compatibility Table

5.2.2. Operand Verification

The Operand Verification Unit is driven by the pipeline processor. As each syllable is read from the pipe, a tag describing its type (character, boolean, integer, real) and class (scalar, array, reserved, no value) is read from the tag memory into the appropriate tag register.

The preliminary semantic analysis process consists of checking that both operands are valid data types. The compatibility table used is shown in Table 5-2.

Two table-controlled semantic summary values are available from the OVU hardware in the pipeline, but these cannot be used automatically to accept or reject the semantics of a given operation because different functions have different domains. Table 5-3 compares the types of the operands and is used to reject invalid type combinations (bearing in mind that all pairs of types are valid arguments for "equal" and "not-equal") and to select the correct "cast" procedure if the types do not match.

Table 5-4 gives the shape of the operands and the type of the expected result and is used not for semantic checking but for allocating space for the result.

RIGHT OPERAND / LEFT OPERAND	CHARACTER	BOOLEAN	INTEGER	FLOATING
CHARACTER	0	4	4	4
BOOLEAN	4	0	1	2
INTEGER	4	5	0	3
FLOATING	4	6	7	0

Table 5-3: Semantic Summary Table for Types

5.3. Instruction Execution

The basic cycle in the DMU consists of

(1) Read a format syllable from the pipe
(2) Execute the microprogram whose address is the entry of the DMU Format table indexed by the syllable

Each format microprogram begins by fetching the relevant number of operand pointers from the pipe and starting a stream reading the DAT at the offset given in the pointer. As a side effect, the DMU tags are read into the tag registers. When all of the operands have been read, the format microprogram reads zero or more operator codes from the pipe. Next, the microprogram signals that it is ready to release the pipe. At this point the semantic checking hardware verifies that both operands have values. (See Table 5-2)

The format microprograms resemble one another up to the point of pipe release. The next stage consists of additional semantic checks where possible, casts where necessary, and allocating space for the result. The DAT entry indexed by constant NXTDEST is used to hold a scalar destination or the descriptor for an array destination.

SCALAR RIGHT OPERAND	CHAR	BOOL	INTG	FLOAT
LEFT OPERAND				
SCALAR				
CHARACTER	0000	0001	0010	0011
BOOLEAN	0001	0001	0010	0011
INTEGER	0010	0010	0010	0011
FLOATING	0011	0011	0011	0011
ARRAY				
CHARACTER	1000	1001	1010	1011
BOOLEAN	1001	1001	1010	1011
INTEGER	1010	1010	1010	1011
FLOATING	1011	1011	1011	1011
ARRAY RIGHT OPERAND				
SCALAR				
CHARACTER	0100	0101	0110	0111
BOOLEAN	0101	0101	0110	0111
INTEGER	0110	0110	0110	0111
FLOATING	0111	0111	0111	0111
ARRAY				
CHARACTER	1100	1101	1110	1111
BOOLEAN	1101	1101	1110	1111
INTEGER	1110	1110	1110	1111
FLOATING	1111	1111	1111	1111

Note: Each 4-Bit entry describes
Left Shape (0 = Scalar, 1 = Array)
Right Shape " "
Result Type (00 = Character, 01=Boolean)
(10 = Integer, 11 = Real)

Table 5-4: Semantic Summary: Shape and Type

Next comes a CASE decision based on rank and/or type. For formats that include operator syllables, the next step is to execute the operator microprogram from the operator EXEC table indexed by the op code read from the pipe. The primitive microprogram may

be executed once or in a looping structure. Some formats do not have operator syllables, usually because the operator is implicit in the format, as in the INDAS3 format for indexed assignment. (Many ADEL formats without operators never go through the pipe. BRABS, NXTL, NOOP, and so on are examples.)

Finally, the format microprogram calls 'CLR_WRI_DST', a microprogram which deletes the current array pointed to by the destination entry of the DAT, and copies the entry at NXTDEST into the destination position. The DMU then waits until a pipe is available to it, and repeats the process.

The DMU contains two EXEC tables, one for formats and one for functions. The format table is different from the PMU format table, since the DMU does not execute such ADEL operators as NOOP and NXTL, but does contain special formats for data transfer. The function table contains the APL primitives, and a set of "hidden primitives" currently including "+/" and "-/".

5.4. Function Call and Return

5.4.1. Call

After a function-calling format such as SLUR or SUS has processed its operands, the next and final syllable of the instruction denotes the function to be called. The tag part of the CAT entry indexed by the syllable is checked to make sure it is really a function. In parallel with this check, the current set of state variables is copied to the end of the current CAT frame and a stream is opened to the segment pointed to by the pointer part of the DAT entry.

The new function segment begins with a table of descriptors used to verify that the

function and format valence match and to set counters for the number of objects of various kinds in the new environment. While these checks and initializations are being done, streams are opened into the Global Symbol Access Table (GSAT) in the function segment and into a new CAT frame. For each object referenced by the called function, the GSAT entry is used to find the corresponding Global Symbol Table page.

The first item is always the result, and is handled by pushing a "no value" onto the DAT stack. The new top-of-stack pointer is written into the CAT with type tag "Variable". The new CAT entry is also pushed onto the call chain of the corresponding symbol table page.

Arguments that are not already on the stack are pushed onto the stack, with their (new or current) stack positions copied into the CAT with type "Variable". The new CAT entry is pushed onto the call chain of the corresponding symbol table page. If the arguments were not already on the stack, the DMU microcode will make copies of the argument array segments.

The identifier of the function itself is handled by writing the segment number of the function into the CAT with type tag "Function". Since a function name is not a local object, nothing is done to the symbol table. (It is legal in APL to have a local variable with the same name as the function. This is sometimes done to prevent inadvertent recursion. The resulting local variable is usually assigned a character vector containing a message that the function is suspended and should not be rerun without remedial action.)

Local identifiers are listed after the right argument, with the number of locals indicated by an entry in the function descriptor table. For each local identifier, "no value" is pushed onto the DAT stack, and the new top-of-stack pointer is catenated to the "Variable" tag

to arrive at a value that is written into the CAT and pushed onto a symbol table call chain. APL semantics allows local functions, but at this stage of development the interpreter assumes that all identifiers in the local list will be variables.

Since APL semantics do not include "own" or "retentive" variables, it is not necessary to initialize local variables beyond instructing the DMU to set their tags to "No Value" and setting the PMU tags to "Variable" so the first statement that assigns a value to a local variable will be recognized as valid syntax.

The rest of the identifiers in the function are global named objects, literals, and line labels, listed in the order that they were recognized by the translator.

Labels are distinguished at link time by setting the sign bit of the corresponding entry in the GSAT. When a label is encountered during the CALL process, the sign bit is cleared and the symbol table is found as with other identifiers. The value of the label is read from the next GSAT entry. The label value is catenated with the "Constant" type tag to get the value which is written into the CAT and pushed onto the symbol table call chain. Thus a label is effectively a "local constant".

Literals, global variables, and functions are not distinguished in the CALL process. In each case, the symbol table page pointed to by the GSAT entry is consulted to find the end of the call chain, and the value there is written into the CAT. Thus the tag and DAT pointer of each item will be the same in the new environment as in the old. In particular since literals cannot be shadowed (because there is no way to get the translator to accept a literal as an argument, local variable, or label), they will retain their "Constant" type tags and their pointers into the Global/Literal area of the DAT.

5.4.2. Return

Return from a user-defined function is a simpler process than calling. The DMU is told the number of local objects (including arguments but not labels or the result) and clears the values of that many items in the local/execution stack. The process of popping the values is completed by incrementing the stack pointer by the same number. For each local object (including arguments, labels, and the result), the call chain in the corresponding symbol table chain is popped.

The result of popping the name but not the value of labels is simply that the old values become unshadowed. Remember that the value of labels is set at link time, not at call time, so no new values are created by CALL.

The effect of popping the name but not the value of the result is that the old value of the name is unshadowed, and the new object which was created by CALL and given a value during the execution of the called function is now on top of the stack in the calling environment. All of the currently implemented formats involving CALL return the result to the stack this way. Implementing a format like DLUR, which assigns the returned value to a specific variable, would require that the "D" syllable would have to be stored during CALL to be used during RETURN.

When the identifiers have been dealt with, the state information at CALL time is read from the end of the CAT frame preceding the called frame. (Note that the size of this frame is not known yet, so its beginning cannot be located. However, its end is right before the beginning of the called frame.) This information is used to open the old CAT frame, reset the LINENO register, and restart the stream used for executing ADEL code at the syllable following the end of the ADEL instruction which initiated the CALL.

5.5. Primitives

An APL implementation consists of systems to accept input and control the environment, plus the all-important programs that do the work itself. While the primitives are the first thing anybody notices when learning or using a language, in an implementation project they can be the last thing considered.

APL primitive functions can be divided into three groups: the scalar arithmetic and logical functions, the mixed functions, and the array-manipulation functions.

5.5.1. Scalar Functions

A scalar function like "+" applies to scalars, or to the elements of an array, producing a result of the same shape as its arguments. For example, the result of adding a 3-row, 6-column matrix of numbers to another matrix of the same shape will be a new 3-by-6 numeric matrix The principle of "Scalar Extension" allows a scalar to be added to an array producing an array result.

Scalar functions are executed via the DLR family of formats. Arguments cannot be rejected from inspection of the tags except in the case where character arguments are used with a function whose domain is arithmetic. The semantic checking consists of matching ranks and shapes.

5.5.2. Mixed Functions

The mixed functions produce results of different shape from the arguments. For example, "matrix divide" can be used to solve a system of linear equations. It takes as arguments a matrix of coefficient values and a vector of constant values, returning a vector of variable values. "Encode" and "decode" are two more mixed arithmetic functions.

```
24 60 60 decode hours,minutes,seconds
```

produces the total number of seconds represented by the three values, and

```
24 60 60 encode numberofseconds
```

breaks the number of seconds down into hours, minutes and seconds. None of the mixed arithmetic functions have been implemented in the first pass at the interpreter.

Two mixed functions do not deal with arithmetic. Index_Of, a dyadic function symbolized by ι, finds the position of the elements of its right argument in the vector used as its left argument. For example

```
      2 3 4 5 6 ι 3 4
1 2   (in index origin 0)
```

Membership, whose symbol is ϵ, returns a boolean of the same shape as its left argument, indicating whether or not each element is found anywhere in the right argument.

Neither membership nor index_of have been implemented yet.

5.5.3. Array Manipulation Functions

The array manipulation functions generally take a left argument describing the size and shape of the array to be returned, and a right argument containing the array to start with. The most important functions in this group are Reshape, Take, and Index, denoted by $\uparrow$,ρ,and [ ], respectively. Only the Index function is found in most scalar languages.

```
5 6↑ ARRAY
```

returns an array containing the first 6 columns of the first 5 rows of ARRAY, and

```
5 6 ρ ARRAY
```

returns a 5-by-6 array containing the first 30 items of ARRAY taken in row-major order.

In APL, indexing can select arbitrary subarrays as well as scalar elements.

ARRAY[3 4;⍳4]

returns a two-by-four element matrix containing the elements in the first four columns and rows 3 and 4 of ARRAY.

Reshape and Index have been implemented in the first version of SAM APL. Both of these are used with the DLMR family of formats. Semantic checking consists of verifying that the "left" argument is numeric. In order to make indexing consistent with the other array selection functions, X[A] is translated as if it were written A index X.

Indexed Assignment is complementary to indexing, allowing a specified element (in APL, a specified orthogonal subarray of elements) of an array to be respecified. Only indexed assignment of a scalar element has been implemented in this version.

The other APL array-manipulation functions include Drop, Rotate, Transpose, Select (or Replicate in most current APL implementations) and Expand. None of these have been implemented in the first version of SAM APL.

Two more functions can be included in this group: monadic ⍳, the index generator, and catenation, whose symbol is the comma. (Monadic comma, or ravel, is a special case of reshape included in APL for convenience. Its implementation will be easy enough but has been given a lower priority than most of the other primitives.)

The expression "⍳N" generates a vector containing the first N integers, beginning at zero or one depending on the current "index origin". The index generator has been implemented in a restricted form, with a limit of 256 elements and with only origin zero. Source code

using the index generator is translated into ADEL code using the DMR family of formats. Semantic checking consists of verifying that the argument is numeric. The operator microcode simply generates a DAT entry with length field copied from the argument and a pointer to a read-only segment containing $\iota 256$.

Catenation is the most important APL primitive function that has not been implemented in this version of SAM APL, and is the first function scheduled for implementation in the ongoing project.

It should be noted that there need not be a one-to-one correspondence between primitives in the intermediate language and primitives in the source program. For example, the front-end translator is free to recognize common combinations of APL functions like $\times\iota$ and produce a "hidden primitive" in the intermediate program. In the opposite direction, a different translator might accept a source program containing an unimplemented function, say "stringwise_index", and produce ADEL code containing a call to a user-defined function or a sequence of ADEL instructions.

5.6. Derived Functions

Derived functions in APL are formed by applying an operator to functions and data. Examples of derived functions are plus-reduction $(+/)$, inner product $(+.\times)$ and first-axis times-reduction $(\times/[1])$. In the SAM APL project, derived functions are handled in three different ways.

5.6.1. Special Case Microcode

Some derived functions (currently comprising +/ and -/) have the status of "hidden primitives". The APL expression

X ← +/Y

translates to the single ADEL instruction

DFR Y X +/

and is executed by the DFR format microcode and the PLUSRD function microcode. This approach is useful when there is an algorithm that is more efficient than straightforward iteration as in **and**/ and **or**/. Reduction of vectors of length greater than two is rare for the transcendental functions, and reduction by these functions will probably be handled by general-purpose microcode or PMU derived functions.

It is uncertain whether derived functions using the scan (\) operator

```
      +\1 2 3 4 5 10
1 3 6 10 15 25
```

will be implemented with DFR, since the current version of the format hard-codes the rank reduction (i.e., the reduction of a vector is a scalar) associated with the reduction operator.

5.6.2. PMU Derived Functions

Some primitive scalar functions will, at least initially, be implemented as system-resident defined functions. Derived functions which use these primitives will be implemented as defined functions which incorporate calls to the defined "primitives". The consequences of this policy are discussed in the last chapter.

5.6.3. General Microcode

The product operator has a format to itself.

X ← Y +.× Z

would be translated into

DL.R Z Y X + ×

The format microcode then sets up loops and EXECs primitive functions. The scan operator, and some cases of reduction, will also be handled this way.

Note that it is quite possible to implement the commonest products as special cases. It is also possible to use reduction primitives where they exist instead of using only scalar functions in the loops.

Chapter 6

Performance Considerations

6.1. Control Primitives

Many APL programs contain loops, and the standard APL language does not contain an efficient way to deal with them. The typical construction

$$\rightarrow LOOP \times \iota LIMIT > COUNT \leftarrow COUNT + 1$$

involves all of the usual overhead in checking the type of COUNT, performing the addition and comparison, doing whatever arithmetic is involved in the branching idiom used, and finally transferring control.

In the SAM machine, the type checking is done in parallel with the operations, so performance would be improved. However, the transfer of control is a PMU operation, so time would be wasted waiting for the DMU to finish its work and send the result back to the PMU. This waste of time is especially pernicious because the PMU is intended to keep ahead of the DMU. For this reason, SAM APL is extended to include control primitives.

The control primitives developed in the first implementation consist of a branching primitive and a pair of loop control primitives. The ADEL form of the branching primitive is named BIFSTK (Branch IF StacK) The looping primitives consist of a counter initializer with a zero check called INITBZ and an increment-test-and-branch primitive called IBNZ. The ADEL code tested had the form

```
SMR VEC ρ                    Stack ← Length
 INITBZ END                  Counter ← Stack
                             → END if Counter=0
LOOP:
 body of loop
 IBNZ LOOP                   Increment Counter
                             → LOOP unless Counter=0
END:
```

The corresponding APL source code would contain a dyadic system function INITCZ (Initialize and check if zero) and a monadic INCCNZ(Increment and check if not zero), with the syntax

```
 → END INITCZ ρ X            Start Counter and
                             → END if Counter = 0
LOOP:
 body of loop
 → INCCNZ LOOP               Incr Counter and
END :                        → LOOP unless Counter = 0
```

The looping primitives were developed and tested while writing a user-defined version of plus-reduction. The measured results are shown in Table 6-1 Note that the per-element times include the time taken for data manipulation as well as time taken by the control primitives themselves.

Coding	Set-up	Per Element
Microcode	423	2
Looping Primitive	1319	196
Branching Primitive		
(Scalar Test)	2038	297
(Vector Test)	1397	391

Table 6-1: Performance of Control Primitives

Note that using the looping primitive allowed the per-element loop time to be reduced to

196 cycles, as compared to 297 using the branching primitive. Without the branching primitive, the loop time would be slightly longer. This measurement would have required the standard format for computed branch, BSTACK, which has not been implemented yet.

Careful coding is still a useful way of improving speed. Omitting an extra step to ensure that the limit and loop counter are scalar saves 641 cycles of set-up time but raises the per-element time to 391 cycles.

Five different branching techniques are shown in Figure 6-1. The looping primitive IBNZ is the only one that does not require the PMU to wait for a value from the DMU.

The standard branching primitive BSTACK executes a typical APL test-and-branch as three ADEL instructions: the test itself, the APL instruction which uses the destination and the result of the test to place a line number on the stack, and the branch instruction itself. The "Branch-from-Stack" primitives BRANSR and BRANLS combine the branching operation with the preceding operation, so that the APL expression

$$\rightarrow (Y < X) \,/\, LBL$$

can generate the ADEL code

```
SLR X Y < Stack ← Y < X
BRANSR LBL /
```

The present branching primitive BIFSTK could not be generated from APL source code without an IF keyword. BRANSR and BRANLS would generate two ADEL instructions, as BIFSTK does rather than three, as BSTACK does. Since they use an APL function such as "/" or "×⍳" instead of a special **IF** instruction, their performance should be somewhere between BIFSTK and BSTACK. The BIFLR format is based on the realization that the destination syllable of a branching instruction could be used to contain the destination of

the branch itself. By condensing the test, the branch instruction, and the destination into a single format, BIFLR would run slightly faster than BIFSTK.

All of the branching primitives discussed here will be implemented for testing purposes, although only a subset can be kept in the finished interpreter.

Branching Technique	ADEL Code
Looping Primitive	IBNZ Loop
Branching Primitive	SLR X Y < ; BIFSTK Loop
Branch-to-Stack	SLR X Y < ; SSR Loop / ; BSTACK
Branch-from-Stack	SLR X Y < ; BRANSR Loop /
"	SLR X Y < ; BRANLS Loop ×⍳
One-Step Branch	BIFLR X Y Loop <

Note: Semicolon is used here as an instruction Separator

Figure 6-1: Some Possible Branching Techniques

The control primitives in the final implementation may differ from the constructs implemented in this version, which are efficient but not necessarily in keeping with the theme of APL. (In fact, they are not available as APL source but have to be hand-coded into an ADEL assembler).

One common APL idiom that can be handled by a control structure is execution involving a constant string. The expression

execute $(Y=3) / 'Y \leftarrow X \div 2'$

might be compiled to a test-and-branch equivalent to the (illegal) APL expression

$Y \leftarrow X \div 2 \rightarrow (Y=3) \downarrow NEXTLINE$

6.2. Call/Return Mechanisms

Function call and return is one of the principal sources of overhead for programs in most languages. This is particularly true of APL, which allows applications to be built of dozens or hundreds of independent functions, with the dynamic nature of the language making function call expensive on conventional architectures. The SAM architecture has been designed to address the problem of efficient function call and return. The process is fairly similar to DELtran practice [FIH83] and was described in detail in Section 5.4.

To recapitulate, each identifier in a user-defined function is mapped to a symbol table entry. At call time, each entry is inspected, and the corresponding data table pointer is written into a Contour Access Table which corresponds exactly to the numbers used to encode objects (other than primitives) referenced by the user-defined function. If the item is a local variable, the DAT pointer used is top-of-stack after a "no value" has been pushed onto the DAT stack. If the item is a label, the corresponding line number is written into the CAT. For both labels and variables, the CAT entry is pushed onto the call chain for the symbol table entry, shadowing the previously accessible value.

The arguments, if any, are currently handled by copying the DAT entries and the array segments on function call. Array segments for arguments that are from the stack do not have to be copied. Later investigation will determine whether it is worthwhile to implement a "protected-call-by-reference" mechanism to avoid the overhead of making copies of the arguments.

Function return consists of deallocating all of the local variables which have received values and marking the corresponding tags "undefined". The corresponding symbol table page call chains are popped, and corresponding values are popped from the DAT stack.

By popping the result name (in the symbol table page) but not the result value (in the DAT stack) on function return, the value is available as the top-of-stack in the calling environment.

Function call overhead is still considerable compared to that for a simple branch. Several improvements are possible.

- Implementing the formats like DLUR which put the result of a called function into a specific variable instead of the stack will save one pipe cycle by allowing a sequence like SLUR followed by DGETSS to be replaced by a single ADEL instruction.

- It is not necessary to perform a symbol table access for literals, since the DAT pointer is known at link time.

- ADEL keeps literals in the same CAT as named identifiers. It is feasible to keep a separate contour for literal values [FJW85]. To make this work with the current system, new formats would be needed, so "$X \leftarrow X + 31$" would translate to "LLC X 31 +". Note that this would require some way to handle applications with more than 256 literals.

- An edit-time check could determine whether arguments are modified, so the DAT pointer could simply be copied for an unmodified argument, without any need for reference counting and protection flagging in the DMU.

- Special areas of the CAT and DAT could be reserved for "leaf" functions which do not themselves call other functions, allowing a call to be a matter of resetting the environment pointer.

- If the identifiers used by a function are listed in order of use, function execution could be begun before the new environment is complete. More identifiers would be set up as the PMU gets ahead of the DMU, or as they are needed.

- Finally, hardware could be designed to assist the calling mechanism. For example, the CAT and DAT were kept in hardware in an earlier version of the SAM design [HGT86]. Moving these two tables to segmented memory slows down call/return performance by a factor of about two.

```
    ∇    R ← X ACKER Y
[1]      → BIGX IF X > 0
[2]      R ← Y+1
[3]      → 0
[4]  BIGX:
[5]      → BIG2 IF Y > 0
[6]      R ← (X−1) ACKER 1
[7]      → 0
[7]  BIG2:
[8]      R ← (X−1) ACKER X ACKER (Y−1)
    ∇
```

Figure 6-2: Ackermann's Function

			DEPTH	TEST-AND-BRANCH			
X	Y	R		CALLS	SUCCESS	FAILED	COST
0	N	N+1	1	1	0	1	1270
1	0	2	2	2	1	2	1879
1	1	3	3	4	3	3	2946
1	2	4	4	6	5	4	4015
1	N	N+2	N+2	2N+2	2N+1	N+2	--
2	0	3	4	5	4	4	3560
2	1	5	6	14	13	9	8443
2	2	7	8	27	26	16	15461
2	3	9	10	44	43	25	(24616)
2	N	2N+3	2N+4	$2N^2$ +7N+5			
3	0	5	7	15	14	10	9068
3	1	13	15	106	105	59	(58185)
3	2	29	31	541	540	284	(291518)
3	3	61	63	2432	2431	1245	(1303078)
3	N	$2^{N+3}-3$					

COST is measured clock values.
Values in parentheses are estimates.

Table 6-2: Measurements of Ackermann's Function

A good way to exercise the calling mechanism is Ackermann's Function, a doubly-recursive function with the number of calls growing more than exponentially with its arguments. The APL version of Ackermann's Function is shown in Figure 6-2 The

number of calls, branches, and failed branches for various arguments is shown in Table 6-2. The cost of execution is given in the final column. The simulator is too slow to run 3 ACKER 3, but a linear regression of cost versus number of calls, successful branches, and failed branches gives agreement to within 0.1% for all cases up to 2 ACKER 2, and has been used to extrapolate to 3 ACKER 3. The extrapolated speed is compared with various other implementations in Table 6-3. A hardware speed of 8 MHz is assumed.

[HGT86] give a simulator measurement of 1.5 ms for 2 ACKER 2 on 5 MHz hardware. This works out to 7500 machine cycles, over twice as fast as the current simulator. Most of the difference is due to the present implementation of the CAT and DAT in segmented memory instead of in hardware.

LANGUAGE	ENVIRONMENT	TIME (MILLISECONDS)
SAM APL	SIMULATOR	163
SAM APL	Tables in Hardware	80
C	SUN 2	50
C	SUN 3	13
STSC APL	Select AT	15,900
MTS APL	IBM 3081	1014
MTS LISP	IBM 3081	110
Franz LISP	SUN 3	2400
Interpreted LISP	LMI LISP Machine	20,000
Compiled LISP	LMI LISP Machine	92
C	IBM 3081	8

Table 6-3: Comparison of 3 ACKER 3 on Various Systems

6.2.1. The ICKER Function

Since Ackermann's function exercises the calling mechanism of a language implementation, it seemed that it would be interesting to write a purely iterative version of the function. A comparison of the performance between ACKER and ICKER (shown in Figure 6-3) should provide a comparison between calling and branching performance of the implementation. A table of performance measurements is shown in Table 6-4.

```
    ∇   R ← X ICKER Y;STACK;PTR
[1]     STACK ← 256⍴⍳256
[2]     PTR ← 0
[3]  TOP: → BIGX IF X > 0
[4]     R ← Y+1
[5]     → END
[6]  BIGX: → BIG2 IF Y > 0
[7]     Y ← 1
[8]     X ← X−1
[9]     → TOP
[10] BIG2: STACK[PTR] ← X−1
[11]    PTR ← PTR+1
[12]    Y ← Y−1
[13]    → TOP
[14] END: → 0 IF PTR=0
[15]    PTR ← PTR−1
[16]    Y ← R
[17]    X ← STACK[PTR]
[18]    → TOP
```

Figure 6-3: Iterative Ackermann's Function

According to the table, ACKER is more efficient than ICKER on every APL implementation except SAM APL. The measurements for LISP and compiled C were made later in the project and will be discussed later.

My first conjecture was that either SAM APL calls are relatively slow, or its branches are relatively efficient. To check whether the effect was caused by something in ACKER

APL SYSTEM	COMPUTER	CLOCK (MHz)	HARD-WARE	TIME(SECONDS) ACKER	ICKER	RATIO
SAM	Simulator	8	none	.036	.027	0.75
Porta	Fat Mac	8	68000	18	23	1.3
"	Zenith 151	4.7	8088	46	56	1.2
STSC	Zenith 151	4.7	8088	19.9	25.5	1.3
"	Select AT	10	80286	4.7	7.6	1.6
MTS	IBM 3081			.112	.171	1.6
Sharp	Zenith 151	4.7	8088	452	632	1.4
Dyalog	SUN 2		68010	7.1	10.5	1.5
C	SUN 3		68020	.0033	.0025	0.75
C	IBM 3081			.0016	.0008	0.5
LISP	IBM 3081			.024	.032	1.3

Table 6-4: Comparison of 3 ACKER 2 and 3 ICKER 2

or something in ICKER, execution timings with different sets of arguments were compared with the timing on MTS APL. The results of this comparison are shown in Table 6-5.

		---MTS APL--			---SAM APL---			---MTS/SAM----	
X	Y	ACKER	ICKER	RATIO	ACKER	ICKER	RATIO	ACKER	ICKER
0	0	0.44	0.65	1.49	0.155	0.22	1.40	2.84	2.95
1	0	0.67	0.87	1.32	0.23	0.25	1.10	2.9	3.5
1	1	1.06	1.58	1.50	0.36	0.36	0.99	2.9	4.4
1	2	1.46	2.21	1.52	0.50	0.46	0.92	2.9	4.8
2	0	1.31	1.78	1.36	0.44	0.39	0.89	3.0	4.6
2	1	3.16	4.65	1.47	1.05	0.83	0.79	3.0	5.6
2	2	5.83	8.78	1.51	1.92	1.48	0.77	3.1	5.9
3	2	111.5	168	1.50	36.3	27.2	0.74	3.1	6.2

Table 6-5: Comparison of ACKER and ICKER on SAM and MTS for Various Arguments

As seen in Table 6-5, the relative performance of SAM and MTS APL for Ackermann's function is fairly constant. Also, the ratio between the costs of ICKER and ACKER on MTS APL was not dependent on the arguments. But ICKER on SAM APL performed much better than ACKER with larger arguments. Figure 6-4 plots MTS and SAM costs for ACKER and ICKER against MTS ACKER cost on a logarithmic scale.

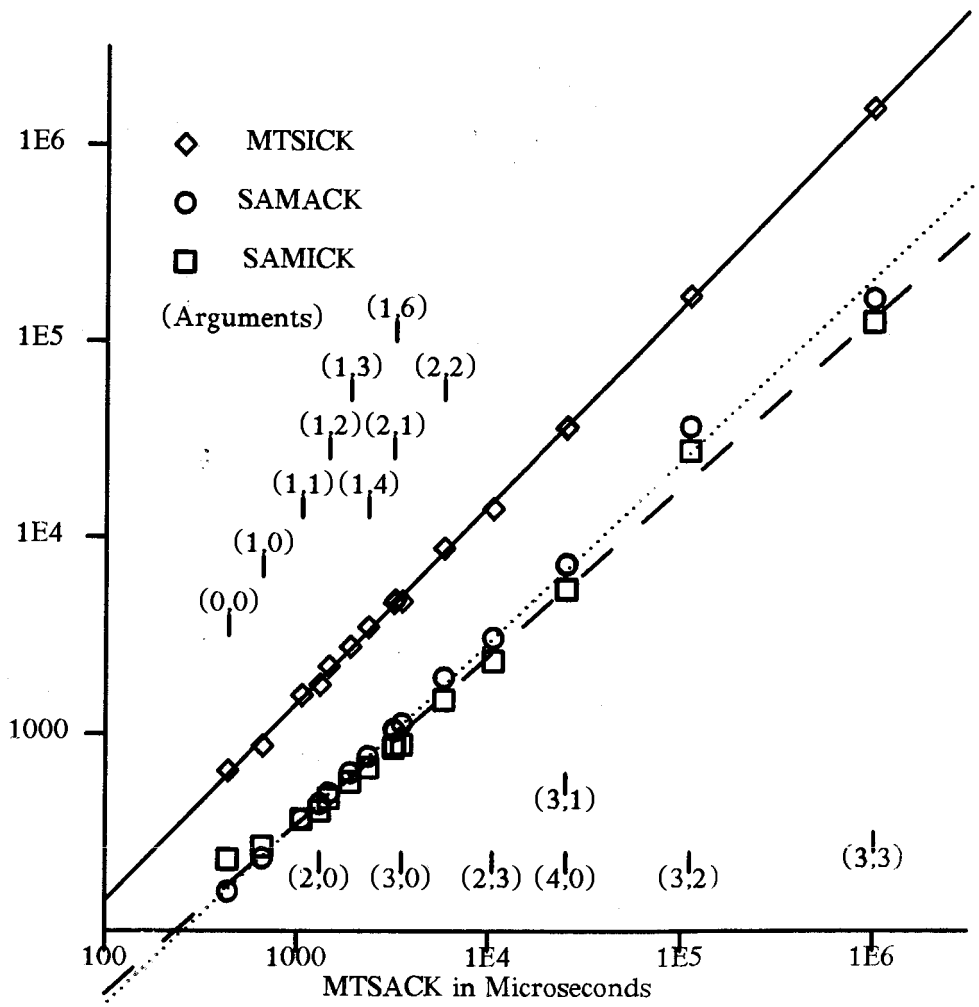

Figure 6-4: Comparison of Recursive and Iterative Performance IN SAM and MTS APL

To test the effect of number of instructions (and to make sure that nothing in the simulated run was compromising the integrity of the clock), the simulations were rerun with an ADEL instruction counter added to the simulator. Clock values were plotted against instruction counts in Figure 6-5. It is apparent that ICKER uses slightly more instructions that ACKER, but the average instruction costs less, and the graph of cost versus instructions for ICKER does not pass through the origin. Since the instruction mix for ICKER contains a single function call, while the proportion of calls to other instructions

in ACKER is nearly constant, it is evident that the performance advantage of ICKER is due to replacing function calls with array references and branches.

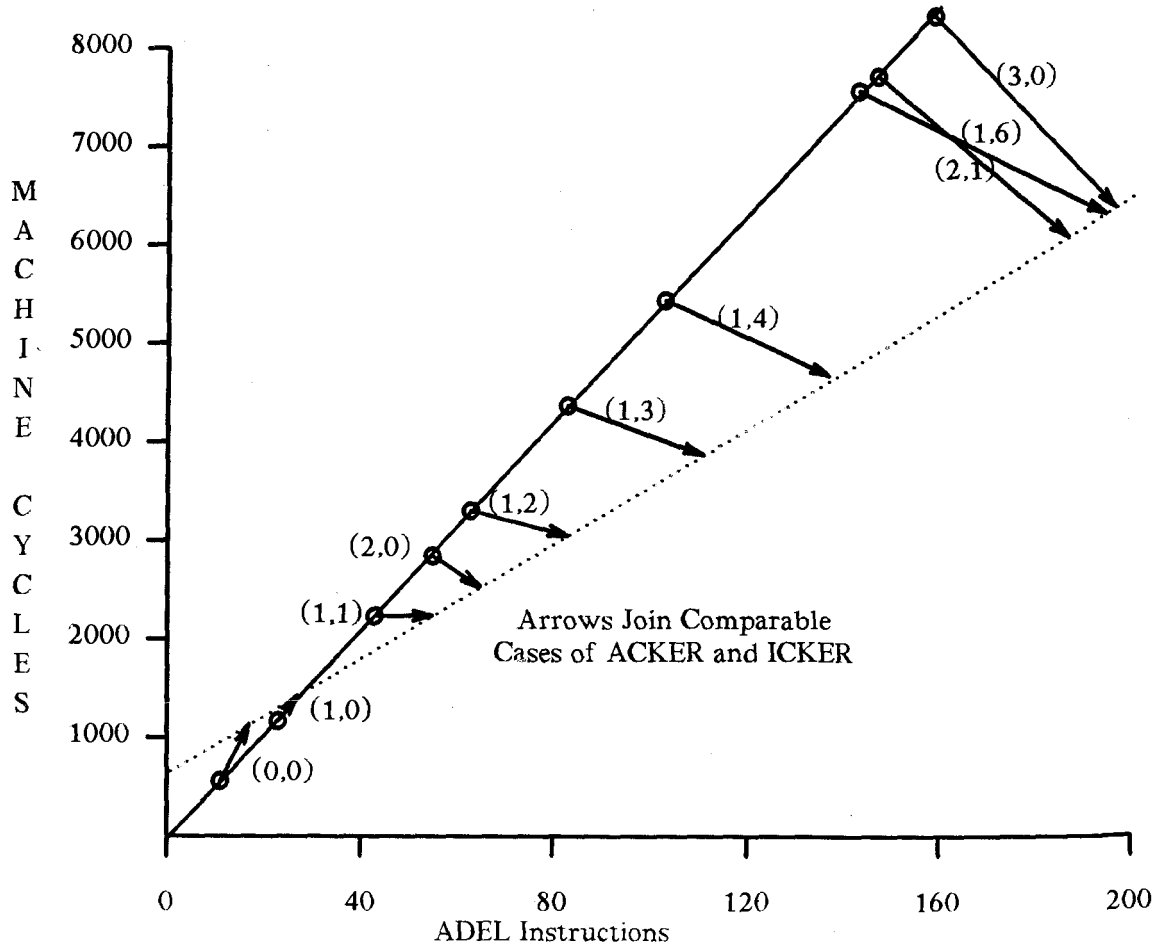

Figure 6-5: Cost Versus Instruction Count in Recursive and Iterative Versions of Ackermann's Function

At this point the pressing question was why other APLs did not show the same behaviour. Either the greater number of high-level instructions executed in ICKER must outweigh the function calls in ACKER, or the array references found in ICKER but not ACKER must cost almost as much as function calls. A comparison of the costs of array references, function calls, and branches is shown in Table 6-6.

SYSTEM	-----TIME IN MICROSECONDS------- TEST AND BRANCH	ARRAY INDEX	INDEXED ASSIGN	FUNCTION CALL
SAM	17.4	8	8	43.5
MTS	78	55	49	71

Table 6-6: Costs of Array References, Function Calls, and Branches

Notice that on MTS APL, test-and-branch, function call, and array references all take the same order of time, 50 to 80 microseconds. In SAM APL, array references are twice as fast as test-and-branch, and function calls are more than twice as slow. This behaviour is more typical of compiled code than interpreted, suggesting that interpretation on SAM will resemble the execution of compiled code in detail as well as in raw performance. This is supported by the figures for LISP and compiled C in Table 6-4. In interpreted LISP, the list references for ICKER outweigh the function calls for ACKER, but in compiled C, ICKER is faster than ACKER by about the same ratio as in SAM APL.

At this stage in the development of the interpreter, only a very rough guess can be made at the ultimate performance of the call and return mechanisms.

At a very rough guess, then, the speed of function call and return might be doubled. The microroutines in this version of the interpreter have not been fine-tuned for performance (see Section 6.6). Implementing the CAT and DAT in hardware and tuning up the microcode would certainly bring the performance up to the predictions in [HGT86] which is double the current speed. Several suggestions for improving the performance of call and return are listed earlier in this section. An interpreter using all of these ideas and with fine-tuned microcode may be twice as fast as the current model without hardware assistance Finally, the technology used in the 68020 chip is at least a

factor of two faster than the technology of an 8MHz SAM system. Raising hardware speed to 16 MHz, moving all tables to hardware, using the fastest algorithms possible, and fine-tuning every line of microcode should speed up call and return by an order of magnitude, giving a speed for Ackermann's Function about the same as compiled C on a 68020 machine.

I have drawn two tentative conclusions from the ACKER-ICKER investigation. First: the poor performance of ICKER on other APLs is due to inefficient array accesses. Second: part of the advantage of ICKER over ACKER on the SAM system may be due to the fact that function call and return are still in their initial correct-but-not-fast incarnations In the final version of SAM APL, ICKER will run faster than ACKER, but probably by about half of the current margin.

It should be mentioned here that the indexed assignment instruction used was special-cased for scalar indices of vectors. (See Section 6.4 for more detail on this instruction.) A more general indexed assignment would be slower. However, there is no reason to suppose that the translator will have any difficulty recognizing scalar indices of vectors and generating the appropriate ADEL instruction even when more general cases are supported. It is unlikely that future developments will make ICKER run more slowly.

6.3. Array Arithmetic

6.3.1. Dragalong and Beating

APL often does operations that appear to be unnecessary. For example, in the expression $A \leftarrow 10 \uparrow B + C$, where B and C are 1000-element vectors, a naive implementation would perform 1000 additions, then discard 990 of the results. In his PhD thesis, Abrams [Abr70] suggested that a more sophisticated APL interpreter should "beat" the addition by "dragging along" a temporary expression until the take operation is complete for both arguments. This approach has a few problems: for instance, suppose B and C were of different lengths. This error could easily be caught. But suppose the operation to be beaten were division instead of addition, and C contained a zero after the 10th position. It is not clear whether an interpreter which did not catch the domain error could be considered to be correct.

In many cases, there is no possibility of an error being introduced by an optimization of this kind. For example, the result of $A \leftarrow B + C + D$ is the same whether two passes are used to create a temporary (C + D) and then add B to it or both additions are done in a single pass. For vectors of length N, 6N memory references are needed for the two-pass algorithm and 4N memory references for the one-pass algorithm. Hobson [Hob84] points out that, given an efficient vector machine, the two vector additions will run faster than the sequence of alternating scalar additions. The implementation of some repeated operations is discussed in more detail in Section 6.5.

6.3.2. Derived Functions

Derived functions can be optimized using the distributive properties of the primitives involved. For example, $SCALAR +.\times VECTOR$ is identical to $SCALAR \times +/ VECTOR$ but $SCALAR +.\div VECTOR$ has no such optimization. The proportion of optimizable cases of this kind in APL applications programs is probably too low to justify the overhead that would be needed to recognize them.

On the other hand, some optimizations are not dependent on special cases. For example, the AND reduction of a Boolean vector

$$X \leftarrow \textbf{and}/\, BOOL$$

reduces to the question of whether or not there is a zero anywhere in the result, and efficient APL implementations do not go on testing elements of the vector past the first zero.

6.3.3. Storage States

One interesting approach to the mixture of arithmetic and selection operations is to use a separate processor for each kind of operation [Sny82]. Snyder's MAPLE architecture handles selection operations as descriptor manipulations whenever possible. This would handle an expression like $A \leftarrow 10\uparrow, B[1;2]$ by generating three successively modified descriptors for B. The array manager would then use the final descriptor to fetch elements of A as needed for arithmetic and logical operations.

Using such storage states speeds up selection expressions at the expense of making data fetches somewhat complicated. A well-designed address calculating mechanism would allow array data fetches at the same speed as ordinary fetches on a scalar machine. The kind of streaming memory access used in the SAM project would be too complicated to justify with current technology.

Another storage state is the arithmetic progression vector, which can be represented by an initial value, a step size, and a length. In the absence of a combinational multiplier built into the addressing mechanism, this representation will slow the use of its elements in any nonsequential fashion. [HaL76] point out that arithmetic progressions are used very heavily in array subscripting operations. It may be desirable to allow the subscripting microcode to generate and use arithmetic progression vectors even if the rest of the interpreter does not recognize them.

There is one place in the SAM interpreter where a storage state approach is used. The index generating primitive saves both space and time, as ⍳N simply generates a descriptor with length field N, pointing to a read-only array segment containing the elements of ⍳256. The principal drawback of this approach is the necessity to check whether a variable is an index vector before deallocating its old value. This problem has been handled by declaring a set of segments to be read-only. The deallocator simply checks whether the segment number is greater than a global READ_WRITE_BEGIN. At the present stage of the interpreter development, this slows down all deallocations with no benefit but a speed-up of the index generator. However, now that read-only segments exist, one part of a nested-array implementation is in place.

A second problem with the descriptor-only approach to the index generator is the existence of two separate index origins. The answer to this problem will be to maintain two separate index-vector segments, setting the pointer to the value active at execution time.

Measurements taken from actual use will be needed to decide whether this approach is optimal.

6.3.4. Arithmetic Coprocessor

For any application involving multiplication, division, or any floating-point operations, an arithmetic coprocessor is necessary for good performance. John Gudaitis [Gud85] shows that the performance of array algorithms is sensitive to the protocol used to access the arithmetic unit(s). If the output of the coprocessor is buffered, it is possible to overlap the bus cycle sending the next arguments with the operation time of the previous cycle.

Inner product requires an accumulation operation in its inner loop. In the general case, it may or may not be faster to use two arithmetic units. The multiply-and-add sequence of ordinary inner product is a processor primitive for the Weitek chip set used in the current SAM implementation. Consider a different inner product, say $\times.+$. Whether or not it is faster to do the vector additions on a different processor than the multiplicative accumulations depends on whether the bus cycles for the extra data transfer outweigh the multiplication time for the processor chosen. In practice, $+.\times$ is expected to be the only common inner product not involving a logical function. The matrix comparison products like **and.**$=$ might use the coprocessor if floating-point numbers are involved.

Two other cases where multiple arithmetic operations can be combined are matrix division and monadic $-.\times$ (the determinant function.) It is important that an APL program dealing with heavily numeric operations such as eigenvectors, Fourier transforms, and the like should be interpreted in such a way as to optimize the streaming through the arithmetic coprocessor. Probably several cycles of implementation will be necessary to get really good mathematical performance. The transcendental APL primitives must be implemented somehow, and it is not at all obvious that the published algorithms will be optimal for the SAM architecture.

6.3.5. Performance Measurements for Array Arithmetic

Problems like Ackermann's Function, with a high proportion of calls and branches to data manipulation operations, are a worst case for an interpreter. Here we will look at the performance of SAM APL in a simple array operation, finding the sum of 1000 integers. In Table 6-1, the microcoded +/ primitive was shown to run with a set-up time of 423 cycles and a per-element time of 2 cycles, giving a predicted time for the addition of 2423 cycles or 0.3 milliseconds. Table 6-7 compares this with some other implementations.

LANGUAGE	HARDWARE	MICROSECONDS
SAM APL	SIMULATOR	300
C	SUN 3	3400
MTS APL	IBM 3081	1800
STSC APL	Select AT	35000

Table 6-7: Time to Add 1000 Integers

For this benchmark, the advantage of compilation is very much reduced. For example, MTS APL is now twice as fast as compiled C instead of seventy times as slow. SAM APL is about six times as fast as MTS APL in both cases.

6.4. Selection Functions

For most applications, the set of APL functions which are used to create, select subsets from, and alter the shape and orientation of arrays are used more heavily than the array arithmetic functions. A considerable body of theory has been developed to combine the selection functions into a set of stepper generators that are used to delay the selection process until all of the steps in the selection have been specified [HaL72, GuW78, TrB82, Ben84]. For example,

```
4 5 ↑ARRAY[2 4 3;5+⍳6]
```

can be treated by generating a "Stepper" or "Grid Selector" for the indexing operation,

apply the "Take" operation to the stepper, and finally apply the resulting stepper to the array itself.

As in the example of combining two additions into a single-pass operation (discussed in Section 6.3.1), this kind of optimization is only useful on a scalar machine. Hobson [Hob84] points out that, given an underlying vector machine, the extra memory references for the straightforward approach are not likely to outweigh the fact that a large number of scalar instructions will be generated instead of a small number of vector instructions. This makes it difficult to justify the extra interpretation needed to form a syntax tree for each expression, combine different operations in the ways discussed above, and attempt to execute the entire tree of operations so formed in one pass.

In the present version of SAM APL, only shape, reshape, indexing, and indexed assignment have been implemented. Indexing and indexed assignment have been restricted to scalar selections into vectors only. With this restriction, an interesting simplification of indexed assignment can be made. Since only three values are used, a single INDASN format can take array, index, and value as its D, L, and R syllables. One possible consequence of this simplification is that the performance of indexed assignment may be unrealistically fast, giving misleading times for the iterative version of Ackermann's Function discussed earlier. At this stage, it does not seem to be too much to expect that the front-end translator can recognize simple scalar-vector indexed assignments and generate the single- operation format, even if the general case will take multiple ADEL operations.

6.5. Sequences of Operations

The use of variable-length instructions with a specific format syllable allows an interesting improvement to repeated executions of the same operation in APL source code. The example most found in application programs is repeated catenation.

```
RESULT ← X , Y , Z
```

In a case like this, especially where X and Y are scalars and Z is a large array, the cost of creating an intermediate result (Y,Z) and then catenating X to this result is much greater than the cost of combining the operations in an intelligent way. The design of ADEL includes formats for repeated operations of this type. The current interpreter does not include any such formats, and their development must be given a lower priority than the many missing features necessary to a robust and complete APL implementation.

Formats that implement repeated executions of a given function are a special case of formats that correspond to more than two levels of a syntax analysis tree [FJW85]. As the authors point out, the number of separate formats needed for all combinations of functions and operands make it impractical to attempt to implement a full set of formats beyond the two-level case (a single function with its arguments and results). Often the result of an expression is used more than once.

```
[1]    X ← Y ← 0
[2]    X ← Y ← W+Z
[3]    R ← X[Y+Z] − W[Y+Z]
```

Example 1 above and other examples of repeated assignment can easily be special-cased. Handling Example 2 in a single ADEL instruction would require a large set of new formats and a redesign of the pipe hardware to handle extra values. A compromise would be to implement a $D \leftarrow D$ instruction, so the example would be translated into the ADEL code

```
DLR Z W X + ; DGETSD Y
```

This format could be implemented easily because a copy of the last value assigned can always be found in the NXTDEST entry of the DAT. Thus multiple assignments are possible without using the stack.

Example 3 is the most interesting, as an example of the kind of optimization that is expected from a good compiler. It would be feasible to implement ADEL formats that would do the job (See Section 5.5.3 for a description of the subscript function):

```
SLR Y Z +
DLMT W R [
SLMS X [
LSL R -
```

where "T" in an argument syllable means that the top element of the stack is used but not popped. There are currently about 100 formats that use S as an operand, and a fully general implementation of this idea would require one or more new formats for most of these. It would not be simple to produce a translator that could recognize that a common subexpression is free of side effects and generate the appropriate code.

6.6. Fine-Tuning the Microcode

SAMjr microcode requires very careful attention to detail in order to use the maximum amount of parallel execution. For example, the microcode used for a search iteration in the memory manager is shown in the first part of Figure 6-6. This code requires 6 cycles per iteration and omits and IDLE instruction needed for proper timing of the memory controller hardware. A more efficient way to microcode the search uses the "mailbox" registers in the pipe to hold temporary results. This code takes 5 cycles the first iteration and 4 cycles per iteration thereafter.

Old Version

```
SEARCH:
    MMS1 SRW R[P]          Start stream
    R[SZ] ← SSN MMS1       Read size
    SF R[SZ] MINUS R[N] Δ R[SUC] ← SSN MMS1
            Compare size and read successor
    → FOUND IF ~NEG Δ R[PRED] ←R[P]
            Test-and-branch, prepare next iteration
    SF ARSHIFT R[SUC]      End-of-list?
    → SEARCH IF ~SB Δ R[P] ← R[SUC]
            Begin next iteration unless end-of-list
```

New Version

```
SEARCH:
    MMS1 SRW R[P]          Start stream for 1st iter
SEARCH1: IDLE
    SF R[N] MINUS M[SAVE] ← SSN MMS1
            Read and save size, compare with N
    → FOUND IF NEG Δ SF ARSHIFT M[SUC] ← SSN MMS1
            Test-and-branch, read successor,check for end
    → SEARCH1 IF ~SB Δ R[P] ← MMS1 SRW M[SUC]
            Start stream for next iter unless end-of-list
```

Figure 6-6: Speeding Up List Search Microcode

This search code takes 5 cycles the first time through and 4 cycles per iteration thereafter.

The best way to apply detailed microcode optimizing techniques to this interpreter will be to obtain some measurements and identify the bottlenecks. Certainly the search loop shown above will be one place where a cycle saved will have a large effect on the performance of the interpreter as a whole.

6.7. Summary of Performance Considerations

The present interpreter gives simulated results about six times as fast as MTS APL on an IBM mainframe and 100 times as fast as STSC APL on a 10-MHZ 80286 machine. Compared to compiled C on a 68020 machine, speed ranges from an order of magnitude slower (for Ackermann's Function) to an order of magnitude faster (for vector arithmetic). Neither of these benchmarks is a realistic model of a real application. For most applications, the performance of the completed interpreter on the present hardware design should more than satisfy the project goal of performance comparable to good compiled code.

Chapter 7

Discussion

7.1. Hardware Feedback

One of the primary purposes of this project was to provide feedback to the team developing the SAM hardware. It would be unreasonably optimistic to expect a hardware designer to anticipate all of the requirements of software. Writing software for a machine that exists only as a simulator leads to the opposite problem, as the software wish list may well include items that are prohibitively difficult to implement in hardware. By having the hardware people inspect and pass on each request as it is made, we hope to avoid both of these problems.

7.1.1. Pipe Clear

The simplest hardware modification to arise from a software request was a facility to clear out the pipeline between the PMU and the DMU.

The PMU fetch-and-execute cycle puts the format syllable into a pipe as a side effect of fetching it. Then argument and operator syllables are put into the pipe by the format microcode. But several PMU format syllables, such as no-op and branch-to-next-line, do not require any DMU action. It is simple enough to write DMU format microcode that does nothing except release the pipe when it encounters these formats. However, this uses up two microcycles of DMU time, and also uses up space in the limited DMU format table. By allowing the PMU to clear out the pipe when it executes a format that does not

require DMU action, the DMU processing is speeded up and the extra formats become available for operations like data transfer, which are not initiated by PMU format fetch.

7.1.2. Triadic Addition

As a contrast to the simple and useful pipeline clear facility, I will mention the Triadic Addition request. After generating a considerable quantity of microcode, I noticed that address calculations and the like often required three values to be added together. Performing two separate additions required two microcycles. Since the major time requirement of addition is carry propagation, I reasoned that three registers could be added, via a carry-save adder in front of the standard adder, in one microcycle.

This particular request turned out to be neither necessary nor feasible. The microcode format is sufficiently parallel that most extra additions can be done in tandem with other operations. The infeasibility derives not so much from the added complexity of the adder as from the redesign of the bus system that would be needed to use it.

7.1.3. Memory Mapping Table Size

Garbage collection is required when there is no sufficiently large block of contiguous free pointers available in the memory mapping table. Of course, garbage collection is useless if there are not enough pages of free memory left to hold the required object. Therefore, by making the number of entries in the mapping table larger than the number of pages in segmented memory, garbage collection should arise very seldom.

It is easy to see that garbage collection could be required using a table less than on the order of the square of the number of memory pages, by executing a long sequence of commands

```
comment N = total number of pages
comment SMALLEST = smallest page to be allocated
SIZE = SMALLEST
while there is room in memory
    ALLOCATE (SIZE)
    ALLOCATE (1)
    FREE(first allocation of this iteration)
    INCREMENT (SIZE)
end
```

Let K be the number of iterations before there is not enough room in the memory. The next allocation will be for SMALLEST+K pages, and there will be K single pages already allocated. Then the size of table required to map all of these free and allocated blocks without garbage collection will be $(SMALLEST \times K)+(K \times (K-1)) \div 2$ and choosing the worst possible value of N/3 for SMALLEST, we find that the mapping table is roughly $N^2 \div 6$ items long.

In practice, of course, a much smaller memory mapping table will be required to make garbage collection exceedingly rare. The size of mapping table needed will be determined by experiments that will be performed when the interpreter has been developed to the state that it can run real APL code. The practical suggestion arising from this theoretical consideration is that the mapping table should be twice the size needed to map all of the real pages of segmented memory.

7.2. A Minimal APL Subset

It is immediately obvious that the APL language contains redundancies. For instance, the ten dyadic logical functions found in APL can be implemented as the NAND function, with preprocessing to translate expressions involving the other nine dyadic logical functions to equivalent expressions using NAND.

Some APL primitives are tempting candidates for replacement by defined APL functions. Matrix divide is the obvious first choice. Encode and decode, grade up and grade down, factorial/gamma and binomial coefficient, and the trigonometric functions could also be replaced. Some derived functions could also be recognized by the preprocessor and implemented as calls to defined APL functions, especially if monadic -.x (determinant) is implemented.

Since it is much easier to write APL functions than microcode, it is of considerable interest to be able to define a minimal subset of APL sufficient for implementing the rest of APL. Some caution is needed here, since a strictly minimal subset would not include such things as addition and negation, since (-X) = (0 - X) and (X + Y) = (0 - (0 - X) Y). What we are after is not a provably minimal subset, but a provably sufficient subset which is comfortable to work with.

7.2.1. Arithmetic and Logical Functions

The arithmetic functions consist of addition, subtraction, multiplication, and division. The transcendental functions can all be implemented as series, especially if the values of e and pi are available. Encode, Decode, Determinant (if it is to be included at all), and Matrix Divide can be implemented as user-defined functions in a straightforward way.

All ten nontrivial dyadic logical functions are included in the subset. Although it is simple enough to get along without most of them, it is even simpler to implement one, then make slight changes in the microcode to arrive at microcoded versions of all ten. It may be necessary to omit complete microcode for some of the logical functions in order to conserve control store.

7.2.2. Array Manipulation Functions

Since an APL interpreter can be written in a scalar language with no array facilities except scalar indexing, it follows that all of the other array manipulation functions in APL are redundant. In fact, [Zak78] implemented transpose, take, rotate, membership, index_of, gradeup, gradedown, expand, drop, and compress as defined functions. SAM APL will implement all of these in microcode, with the possible exceptions of membership, gradedown, and drop, which can be implemented via the identities

```
X∈Y ← → (Y⍳X)<⍴Y  (where Y is a vector)
GRADEDOWN X ← → GRADEUP (-X)
L↓X ← → ((-×L)×(⍴X)-|L)↑X
```

7.2.3. Array Ranks

```
    ∇    R ← X PLUS Y;RANKX;RANKY
[1]      RANKX ← 1↑X
[2]      RANKY ← 1↑Y
[3]      → RANKOK×⍳RANKX=RANKY
[4]      → (RANKX=0)/SCALX
[5]      → (RANKY=0)/SCALY
[6]      SIGNAL'RANKERROR'
[7] SCALX: R ← (RANKY+1)↑Y
[8]      R ← R,(1↓X)+(RANKY+1)↓Y
[9]      → 0
[10]SCALY: R ← (RANKX+1)↑X
[11]     R ← R,(1↓Y)+(RANKX+1)↓X
[12]     → 0
[13]RANKOK: XSHAPE ← RANKX↑1↓X
[14]     YSHAPE ← RANKY↑1↓Y
[15]     → (and/XSHAPE=YSHAPE))SHAPEOK
[16]     SIGNAL'LENGTHERROR'
[17]SHAPEOK: R ← (RANKX+1)↑X
[18]     R ← R,((RANKX+1)↓X)+(RANKY+1)↓Y
    ∇
```

Figure 7-1: Vector APL Function to do Scalar Arithmetic on an Array of Arbitrary Rank

The initial subset contains only scalars and vectors. This restriction requires a demonstration that a user-defined protocol and a set of user-defined functions can mimic all of the APL primitives and derived functions as applied to arrays of rank 2 or higher. The following protocol is one possibility:

The programmer is responsible for keeping track of which values are arrays of rank greater than one. All such arrays are modelled as vectors, with the first (rank + 1) elements containing rank and shape. For character arrays, twice as many elements are used to encode rank and shape as 16-bit integers.

Then all operations involving these arrays are done by defined functions instead of by APL primitives. It is immediately obvious that this can be done, since it is equivalent to writing a subset of an APL interpreter in vector-only APL. The easiest way to establish the feasibility is to show a few examples of such functions. Figures 7-1, 7-2, and 7-3 show how scalar arithmetic, reduction, and inner product can be modelled.

```
    ∇  R←PLUSRED X
[1]     RANKX←1↑X
[2]     SHAPEX←RANKX↑1↓X
[3]     X←(RANKX+1)↓X
[4]     SHAPER←(¯1)↓SHAPEX
[5]     LEN←(¯1)↑SHAPEX
[6]     LIMIT←×/SHAPER
[7]     R←(RANKX-1),SHAPER
[8]     COUNT←0
[9] LOOP:→(LIMIT<COUNT←COUNT+1)/END
[10]    R←R,+/LEN↑X
[11]    X←LEN↓X
[12]    →LOOP
[13]END:
    ∇
```

Figure 7-2: Vector APL Function to do Reduction on an Array of Arbitrary Rank

```
    ∇ R ← X PLUSTIMES Y
[1]     ● PlusDotTimes InnerProduct
[2]     STRIP_R_S              ● set RANK & SHAPE
[3]     → (RANKX=0 1)/SCALX,VECX
[4]     → (RANKY=0 1)/SCALY,VECY
[5]     COL ← −1
[6]     → LERR if (W ← 1↑SHAPEY) ≠ (−1)↑SHAPEX
[7]     NROWS ← ×/ SHAPEX ← (−1)↓SHAPEX
[8]     NCOLS ← ×/ SHAPEY ← 1↓SHAPEY
[9]     R ← (NROWS×NCOLS)⍴100
[10] CLOOP: → END if NCOLS ≤ COL ← COL+1
[11]    ROW ← −1
[12]    CVAL ← Y[COL+NCOLS×⍳W]
[13] RLOOP: → CLOOP if NROWS ≤ ROW ← ROW+1
[14]    R[COL+NCOLs+ROW] ← +/ CVAL×X[(W×ROW)+I+W]
[15]    → RLOOP
[16] SCALX: → SCALX if RANKY ∈ 0 1
[17]    R ← PLUSREDLA RANKY,SHAPEY,X×Y
[18]    → 0
[19] VECX: → SCALR if RANKY ∈ 0 1
[20]    X ← X[(floor⍳⍴Y)×(⍴X)÷⍴Y
[21]    → 0, R ← PLUSREDLA RANKY,SHAPEY,X×Y
[22] VECY: R ← PLUSRED RANKX,SHAPEX,X×(⍴X)⍴Y
[23]    → 0
[24] SCALY: R ← PLUSRED RANKX,SHAPEX,X×Y
[25]    → 0
[26] SCALR: → 0, R ← +/ X×Y
[27] LERR: ... errorprocessing...
[28] END: RANKR ← RANKX+RANKY−2
[29]    R ← RANKR,((−1)↓SHAPEX),(1↓SHAPEY),R
    ∇
```

Figure 7-3: Vector APL Function to do Inner Product on an Array of Arbitrary Rank

7.2.4. Operators

As soon as APL primitives are implemented as user-defined functions, such functions must be included in the domain of operators. Since an APL operator is essentially a frame which determines how a function will be applied to an array, one way to implement an operator that will handle a user-defined function is to write a user-defined function

(that is, an ADEL function which runs in the PMU instead of a microcode function which runs in the DMU) which accepts a user-defined function as an argument. This leads to ALL user-defined functions being in the domain of operators, and with a very small additional amount of work, user-defined operators become available.

In the initial subset, reduction is implemented as hard microcode for all of the microcoded primitives All other operators, and reduction when applied to source-level primitives, are written as APL functions.

7.3. Microcode Size

The present interpreter contains almost all of the environment maintenance routines, about one-third of the executable formats, and on the order of a tenth of the executable operators. DMU format code has been restricted to vectors and scalars only, and the arithmetic operators to integers only.

Altogether, there are 2275 lines of microcode, excluding comments and headers, distributed as follows:

PMU		
Linker	308	
Maintenance(PMU)	321	
Executable Formats(PMU)	254	
Pipe Control(PMU)	73	
Initializations(PMU)	62	
PMU Total		1235
DMU		
Maintenance(DMU)	91	
Executable Formats(DMU)	499	
Executable Operators	375	
Pipe Control(DMU)	34	
Initializations(DMU)	41	
DMU Total		1257
Common		
Memory Manager	198	
Initializations(Mem. Mgr.)	19	

Since both the PMU and the DMU contain a copy of the memory manager, there are altogether 2492 lines of microcode in the system. However, the PMU and DMU do not share control memory, so the individual totals are more important than the combined total.

The microcode for the PMU is complete except for the executable formats. The final version of the PMU microcode should be about 2000 instructions.

The DMU will require a complete set of formats, extended to deal with floating-point numbers and arrays of arbitrary rank, plus code for all of the primitives that have not yet been written. Writing the entire DMU part of the interpreter in microcode would probably require about 8000 instructions. It would be interesting to see if a sufficient subset of APL could be squeezed into 2K DMU instructions, making possible a cheaper machine with most of the primitives and array code written as ADEL routines.

7.4. Conclusions

The implementation of the first subset of SAM APL has raised a number of interesting theoretical and practical problems. While there was no intention of implementing the entirety of APL in this version, the need for operators acting on user-defined versions of primitives made it necessary to do much of the groundwork for important extensions. The initial performance of some control primitives is very encouraging, whether or not they can be expressed comfortably in traditional APL. Difficulties arising from a half-thought-out speed-up to the index generator led to the necessity for read-only segments, which may prove useful in the implementation of nested arrays.

The simulated performance of SAM APL has been very satisfactory, a factor of more than 100 faster than the fastest microcomputer APL tested. Function call and return are

still bottlenecks in SAM execution, and several feasible ways of speeding up call and return must be investigated. Performance measurement of different ways of expressing Ackermann's Function has apparently shown serious bottlenecks in the way other APL implementations handle array references.

It is worth noting that it took about one year's work before the first APL expression

$X \leftarrow 1$

could be executed. Extensions and improvements have been much quicker to define and test given a working nucleus upon which to build. It is probably premature to guarantee that a complete working APL interpreter will be easy to develop from the current state of the project, but at the very least there are good grounds for optimism.

Appendix A

Glossary of Acronyms

ACRONYM	and EXPANSION	DEFINITION
ADEL	A Directly Executable Language	A linearization of APL code allowing complete source code reconstruction. The "assembly language" of a SAM system
APL	A Programming Language	The high-level computer language that is both the host language and the target language of this project
ASP	Architecture Support Package	A software simulator to run and measure the performance of ADEL code
CAT	Contour Access Table	Ordered table mapping ADEL syllables to DAT indices
DAT	Data Access Table	Table of scalar values and array addresses.
DEL	Directly Executable Language	A linearized form of a source program which can be executed directly
DIL	Directly Interprable Language	A DEL which allows the source code to be recovered from the linearized code
DPM	Dual Port Memory	Used to connect the PMU and DMU to the ECU
DMU	Data Management Unit	The SAMjr system reponsible for data manipulations
ECU	Environment Control Unit	The "front-end" system for user and I/O tasks
EXEC	Execute from Table	Execute a microprogram from a table indexed by a function or format syllable
GSAT	Global Symbol Access	List of pointers to Global Symbol Table, kept in

	Table	internal form of function.
HLL	High Level Language	A programming language that does not require explicit references to the hardware on which a program is to be run.
PMU	Program Management Unit	The SAMjr system responsible for program linking, function call/return, and symbol table management
SAM	Structured Architecture Machine	Two or more SAMjr systems set up to distribute the program and data management tasks in executing ADEL code
SAMjr		A component of a SAM system consisting of an SJ16 micro-engine, a memory controller, segmented memory, and zero or more SFUs
SJMC	SAMjr Memory Controller	Streaming segmented memory controller used by SAMjr
SFU	Special Function Unit	A hardware unit (for example, a floating-point processor), attached to a SAMjr to handle a special job
SPT	Segment Page Table	A table of physical page addresses used for segmented memory reference
VHLL	Very High Level Language	A programming language which handles data structures directly instead of element by element.

Appendix B

Extensions to APL

The present project has been limited to implementing a subset of the APL language, with a single small extension in the form of control primitives. The next step will be to implement the entire APL language, or at least a sufficient set of primitives to allow the remainder of the language to be written in APL itself.

The implementation of real extensions to the language must be postponed until standard APL is finished. However, all existing commercial APL interpreters contain significant extensions, and it is proper to discuss them here, if only to get some idea about which should be considered for SAM APL.

B.1. Nested Arrays

Nested arrays are by far the most important extension to standard APL. A standard array consists of a rectangular block of characters or of numbers of a given type. While this is a simple and natural way to represent financial tables and engineering matrices, it is much less flexible than, for example, a Lisp list. Suggestions for "generalized" or "nested" arrays where each element of an array is (or contains) another array, have been circulating since the first useable APL implementations. [Fal73, GhM73]

After about a decade and a half of discussion, three major implementations of nested arrays were completed The first to be released was I. P. Sharp's system. Then came the NARS system of STSC APL, and finally IBM's APL2. Another important nested array APL is the Dyalog APL available for UNIX systems.

Sharp's system differs from the other commercially available systems by being "grounded": that is, an the enclosure of a simple[1] scalar object is distinguishable from the object itself. The other systems are "floating", so that an enclosed simple scalar is identical to the original simple scalar.

The obvious consequence of this distinction is that "floating" APL requires heterogeneous arrays, since the catenation of an enclosed character and an enclosed number is identical to the catenation of the character and number. "Grounded" APL may permit heterogeneous arrays (although no implementations do so), but does not require them.

A less obvious consequence is the pervasiveness of arithmetic and logical functions. Since in a floating system there is no way to distinguish a number from an element of a nested array containing a number, then nested arrays are in the domain of arithmetic and logical functions. In a grounded system, nested arrays may or may not be in the domain of these functions. In practice, a set of composition operators was defined, such that

$(X+\underline{with}\ disclose\ Y)$

means "Go through arrays X and Y element by element. For each pair of elements, disclose them, apply the plus function, and enclose the result as an element of the result array."

The simple provision of a nested array data type, plus the enclose and disclose functions, is only the first step in providing a useful nested array language. Providing a way to apply mathematical and logical functions to the contents of nested arrays is the second step Many more functions and operators must be defined to allow the

---

[1] A "simple" scalar is an ordinary number or character, but not the scalar result of an enclose operation

manipulation of "nestedness" as simply and powerfully as standard APL operators deal with "arrayness".

Writing the microcode to support nested array operations would be a large but fairly straightforward project. Consider the DMU microcode for the DLR format (Figure B-1):

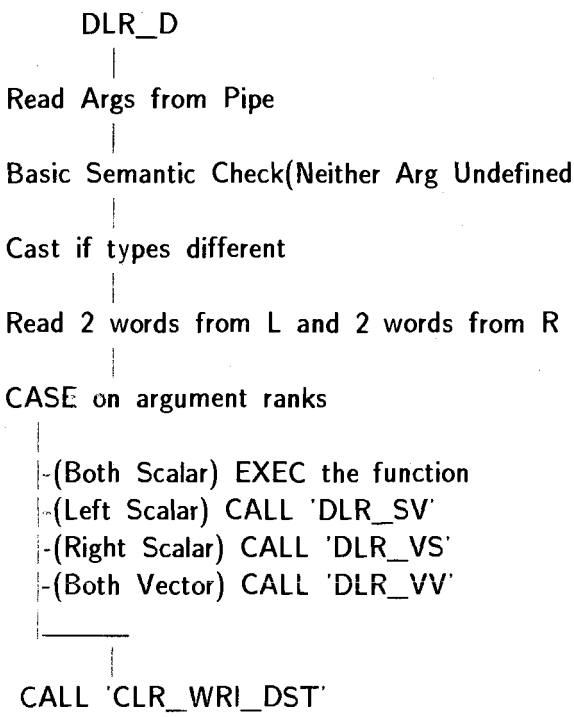

Figure B-1: Broad Outline of DLR Microcode

To handle nested arrays, assume that a floating system is to be implemented, so that $X \leftarrow Y+Z$ might refer to a scalar Y and a nested array Z. Then the DLR routine in the PMU would not have to be changed at all. In the DMU, a test would be inserted after the basic semantic check, as in Figure (B-2). Assume that a nested array has a DMU tag of 1100.

In Figure B-2, the additional test in DLR is shown, followed by a rough outline of the

```
IF L tag is 1100
        CASE on R tag
        |
        |-1100: CALL 'DLR_NN'
        |-10xx: CALL 'DLR_NV'
        |-01xx: CALL 'DLR_NS'
ELSE IF R tag is 1100
        CASE on L tag
        |
        |-10xx: CALL 'DLR_VN'
        |-01xx: CALL 'DLR_SN'
ELSE (Rest of flowchart as before)


      DLR_SN
(DMU Microcode for DLR, Scalar L and Nested R)
        |
Initialize a count for length of R
Open segment containing ptrs for R
        |
      LOOP: Read one pointer for R
            CASE on tag for the pointer
            |
            |-SCALAR: EXEC the function
            |-VECTOR: CALL 'DLR_SV'
            |-NESTED: CALL 'DLR_SN'
```

Figure B-2: Change to DLR Microcode to Support Nested Arrays

DLR_SN microcode that would be called where the left argument is scalar and the right argument is nested. One serious problem that would have to be solved is the recursive nature of nested arrays. If an element of Z is itself a nested array, the DLR_SN function simply calls itself. Since the nature of microcode is not recursive, a different programming style would be needed.

The shape- manipulation functions could operate on nested arrays exactly as if they were arrays of integers. In fact, there is no reason that any of the shape-manipulation code

should 'know' that its argument is a nested array. The problem of 'pad' elements for expand and overtake would disappear if we set up the tables so the value pointed to by 0 is the fill value for a nested array. Presently, the DAT element in position 0 is in fact scalar zero, which is a suitable fill element.

The treatment of nested arrays in this section has necessarily been superficial, and the fact that the examples have been simple is not meant to suggest that the implementation would not be difficult. As the interpreter is developed, I expect to include a very restricted set of nested array operations, possibly limited to Enclose and Disclose. Such a limited nested array system is of little practical use, but it should at least help to prevent implementation decisions that would interfere with a real nested array system in the future.

B.2. Operators

The use of operators to apply functions to arrays is the keystone of APL. Standard APL contains only four such operators: reduction, axis, scan, and product. New operators have been suggested almost as often as new functions. In particular, new operators have been necessary for the manipulation of nested arrays.

Recently, [IPS84] published the beginnings of a formal calculus of operators. The completion of such a calculus might allow the development of a new programming language based upon it, as of nested array theory led to the development of Nial. Such a language might resemble APL only loosely. The extended version of APL proposed by Kenneth Iverson is based on the consistent application of operators [Ive86].

The ultimate operator extension is provided in APL2: user-defined operators. Together with allowing operators to apply to user-defined functions, this completes the process of

using the APL operator to create a language that is totally different in flavour from non-operator languages.

The ADEL intermediate language developed for the SAM APL system encodes the standard APL operators within its format syllables. At present there are no formats for unspecified operators. Adding new operators to an APL implemented via translation into ADEL would require either a new set of formats for each new operator or a set of unspecified-operator formats. New operators could be also realized either by translating them into nested loops during the front-end APL-to_ADEL translation phase, or by trapping to error routines which then call APL programs which do the control operations represented by the new operator. The implementation of operators which apply to source-level representations of some of the APL primitives will make this extension straightforward.

B.3. File Systems

Traditional APL does not have a file system. Early implementations had no way at all of keeping data outside of an APL workspace: everything that was entered or generated in a given session would be stored when the active workspace was saved, but that information could not be used by non-APL tasks, and no more of it could be retained than would fit in an APL workspace. Some ways of using APL workspaces as data bases are discussed in [Soo84]

It was very early obvious to most APL theoreticians that the "right" way to implement APL files was as extra-workspace nested arrays. An early feasibility study [Ive83] suggested that a practical nested array system was ten years and hundreds of man-years away in the early 1970's, but a restricted subset could be implemented in about a year. The result of this effort was the Sharp/STSC component file system.

An APL component file a file is essentially a vector of APL arrays, kept outside of the workspace and accessed by a set of special file functions instead of standard APL array manipulation functions. Component file systems make practical commercial applications feasible, and have been very successful despite their stopgap nature.

The other approach to files in APL was to create a "shared variable" system, wherein a non-APL file AP (Auxiliary Processor) could trade information with an APL task and perform the job of accessing various proprietary file systems and translating data into and out of APL format. Like component file systems, AP-based file systems are commercially practical but clumsy to use. One very useful offshoot of the AP file system concept has been the shared variable facility, which can be used for inter-task communication within and outside of APL.

Since APL will be used on real computers which support other applications, some thought should be given to file interfacing. For example, an APL designed to run on an IBM PC-compatible computer should have some way of dealing with native DOS files. One attractive way to get data base facilities would be to write a useable interface to one of the commercial data base management systems.

B.4. Primitives

Despite the richness of the set of operations created by applying standard APL operators to standard APL functions, it is not hard to define a problem that is most efficiently solved by extending the set of primitive functions. This can be done either by adding new primitives or extending the definitions of existing primitives.

B.4.1. New Primitives

"String_Index" is one example of a function that can easily be written in standard APL but is used heavily enough that an optimized primitive might be desirable. APL supports character vectors, not strings, so that

```
      'THIS IS A SENTENCE'⍳'IS'
2 3
      'THAT WAS A SENTENCE'STRINDX 'AT'
2
```

A primitive String_Index function (and a companion String_Membership function) would simplify and speed up text processing applications, particularly if the underlying hardware includes string-handling facilities.

Various other primitives that have been suggested from time to time include eigenvalue and eigenvector, functions to convert back and forth between matrices and delimited vectors, an assortment of graphics primitives, and functions to handle nested arrays and other data structures.

B.4.2. Extending Existing Primitives

Many features of contemporary APL systems are extensions of functions found in standard APL. For example, the standard sorting primitives do not apply to matrices or to character vectors. Most contemporary APLs have extended the domain of the sorting primitives to matrices and higher-rank arrays, such that

```
RESULT ← MATRIX[gradeup MATRIX;]
```

sorts MATRIX with the first column in ascending numeric order, with ties broken by the second column and so on. An optional left argument contains a collating sequence, so the function can be used to sort a character array.

Another very common extension is the Replicate function. In classical APL.

```
0 1 0 1 1 / 1 2 3 4 5
```

returns 2 4 5. That is, elements of the right argument corresponding to 1's in the left argument are selected. Another way to look at it is the number of copies of a given element of the right argument is equal to the corresponding element of the left argument, either 0 or 1. Extending this second definition gives the Replicate function:

```
1 0 2 0 3 / 5 6 7 8 9
```

returns 5 7 7 9 9 9.

B.5. Strands and Syntactic Variations

It is convenient to be able to denote a single literal consisting of a vector of numbers in the form

```
2 4 6 7 12
```

instead of

```
(2,4,6,7,12)
```

Strand notation extends this convenience to all APL variables instead of just numeric constants. For example:

```
X ← 5 6 7
Y ← 'These are characters'
R ← FOO X Y
```

applies the function FOO to a two-element (nested) vector consisting of X as the first element and Y as the second element. More radically, "strand assignment" can be used to give a value to more than one identifier in one statement. For example `X Y ← Y X` swaps the values of X and Y. All APL implementations with floating nested arrays support

strand notation. Implementations with no nested arrays, or with grounded nested arrays, do not support it.

Strand notation leads to a syntax based on "binding strength" [Bro85]. The resulting syntax is close to standard APL except for the existence of strands and such curiosities as

```
1 2 3[2]  ←→  1 2 (3[2])
```

As imported into APL, strands are at best a mixed blessing. The Nial language is based on array theory [Mor73] with strand notation as the basic input format, and it may turn out that stranded dialects of APL will evolve into a language more like Nial than conventional APL.

B.6. New Environment Features

The standard APL environment is cramped and difficult to work with. APL does not communicate easily with other systems, is not easy to interrupt, and does not provide any easy way to run multiple environments and protect subtasks from each other. Many proposals have been made for minor and major modifications to the APL environment.

B.6.1. Scoping

One very useful proposal was to extend APL's scoping rules [SAL78]. The standard APL name scoping is dynamic with default global. If a function *FOO* calls a function *FEE*, and *FEE* happens to have a loop counter, *I*, which the programmer has neglected to "localize" by including it in the function header, then when *FEE* exits, *FOO* is left with the value of *I* set by *FEE*. If *FOO* also happens to use *I* for some variable, a bug has been created.

Seeds et. al. proposed that the rules be modified to the programmer could choose the default scoping, and that a "strictly local" scope be added such that a local variable could be made invisible to both called functions and the calling function. This simple change to APL would have eliminated about half of the serious errors in my own code over the last seven years.

B.6.2. Event Trapping

The largest extension to the standard APL environment which has actually been implemented on a commercial scale is Event Trapping. When a condition arises which cannot easily be handled by straightforward coding, a trapped APL allows the programmer to specify what action should be taken. Event Trapping has already been implemented Sharp APL, and to a lesser extent on the other timesharing systems. In the Sharp event handler, any function could have a local variable quadTRAP, which contains event codes and APL expressions to be executed when the event arises. For example, "Workspace Full" has event code 1, so if quadTRAP contains "1 RemoveGarbage" and a function encounters Workspace Full, the RemoveGarbage function is invoked. A companion quadSIGNAL system function allows event codes to be raised under program control.

Event trapping makes a large number of problems much easier to solve. For example, if a user has managed to enter bad data into an application program, the program can flush the input, report the situation to the program maintenance file, give the user a more-or-less friendly message instead of a cryptic (to the user) APL error message, and restart in a convenient place.

Extending event trapping to handle interrupts generated from outside the APL task would lead to the possibility of real-time device control programs, more reliable multiple-input-task

systems, and more accurate models of computer systems. A simplified version of this idea was implemented by [ATC77] for sensor-based applications on the IBM System/7 computer.

B.6.3. Namespace Extensions

The global scoping of APL names makes it difficult to hide and protect information in a large system. For example, when a given problem can be solved by using sets of programs in two separate workspaces, the two workspaces can not be copied together without checking for name conflicts and renaming objects to avoid conflicts. A very relevant example is the machine model on which the ADEL interpreter is being developed. The model encompasses two separate processors, and to switch back and forth between them it is necessary to rename all of the data objects in the workspace.

The straightforward way to implement multi-namespace systems in APL is to use an APL which supports multiple tasks, and pass information between them using shared variables. All of the important commercial mainframe APL interpreters (and none of the currently available microcomputer APLs) include a full shared variable facility.

MCM APL [MCMxx] included a one-level structuring system. Objects from one or two of 256 collections of programs and arrays could be accessed at any one time. Using this system, tasks could be integrated with little programming effort, although the performance of the diskette-based hardware was quite slow.

A more flexible system was proposed in [BeD82]. A new set of system commands is used to create, load, and save heirarchical "segments". Whenever an executing function encounters a name which is not in the current workspace, the segment with that name is loaded and the reference is resolved in the new context (which means the segment name must be identical to the name of a new object within that segment).

The Analogic APL machine ([Ber84]) handles name partitioning by threading through multiple "instances" of a workspace with one "prototype". Each workspace is partitioned into "contexts". If a name in a given context is explicitly exported, it may be references in another context, with the value being either a private copy (if the context is an instance) or a permanent stored value (if it is a prototype).

The most drastic namespace handling facility proposed to date is [TaW84]. All APL objects and tasks are subsumed under "The Tree". A complete set of naming and binding procedures is defined, after which different tasks can communicate by using qualified names.

The completed SAM APL may include a modest namespace extension suitable for separating multiple instances of a set of programs while allowing controlled communication between them. The content of the extension will not be worked out at this time. The mechanics of the implementation should be fairly simple. Each context would contain its own set of symbol table pointers, so that the linker but not the execution supervisor must distinguish between different objects with the same name. The details of the extension will require a careful look at what is most useful and easiest to implement.

B.7. New Data Types

B.7.1. Complex Numbers

The numeric domain of Sharp APL and APL2 include complex numbers. It is fairly easy to model complex numbers as arrays with an extra axis of length 2, but it is more convenient to have a complex primitive data type which is in the domains of all of the relevant primitives. A further extension related to complex numbers would be quaternions, 4-tuples of real numbers encompassing complex numbers and 3-space vectors as special

cases. Although quaternions have never been implemented as primitive data types, [Gus86] points out that they be useful for some graphics operations.

B.7.2. Beyond Floating Point Numbers

The APL scheme of booleans extending automatically to integers which extend automatically to floating point numbers handles most mathematical problems conveniently. One common extension is multiple integer types, to increase speed and reduce storage costs for small integers. An "infinity" value can be handled by current hardware and would be useful for catching such problems as the DOMAIN ERROR in (arctan(N×tan(X))) where X approaches 90 degrees. Financial applications would benefit from a fixed decimal storage type. Other possibilities are large integers, rational numbers, and multiple floating-point types to handle overflow/underflow from the default floating-point type.

B.7.3. Graphics

Graphics primitives will certainly be necessary in any new general-purpose language. Efficient graphics applications may require special data types which do not map neatly into arrays. At very least, a set of more than 256 characters will be needed.

B.8. Control Structures

The operator-based approach to arrays make most ALGOL-type nested loops unnecessary in APL. However, instances requiring loops and branches are not supported by anything but the branching and execution primitives. The branch primitive suffers from both high cost and a lack of safeguards against branching into loops, spaghetti code, and "driftwood" (floating branches). Execution is too powerful and general a mechanism to use for a simple IF statement, and implementing it efficiently requires a translation into a test-and-branch or equivalent.

The version of ADEL developed for the first SAM APL supports a BIFSTK branch and a pair of looping primitives to initialize-and-test and increment-and-test. The translator does not recognize corresponding objects in the source code: at the current stage, they are only available in hand-assembled programs.

The idea of introducing ALGOL-style control structures into APL has often been raised and has never been accepted. APLgol [Kel73] is the best-known such implementation. MCM APL contained a way of editing control structures into APL programs. Neither of these implementations has convinced the mainstream of APL programmers that such control structures are useful in APL.

The "EACH" operator found in floating nested array systems is a control structure with an APL flavor.

```
FOO EACH X
```

applies the function FOO to each element of the array X, replacing an explicit loop with an implicit loop. Other operators for program control are described in [Eu85A, Eu85B].

B.9. Arrays of Functions

Once APL arrays have been generalized to contain arrays as elements, a further generalization would be to allow arrays to contain functions. One possible application of arrays of functions would be as a structured environment, with sets of functions needed for a particular purpose kept together in a particular subarray.

A branch of APL theory has developed dealing with the application of function arrays to data arrays [Ben86, Lan86]. At first thought, this kind of operation is too far from the original conception of APL to be integrated smoothly into the language. Arrays of

functions resemble a generalization of the Nial "Atlas" as in the following example from [JGM86].

```
Average IS div[sum,tally]
```

As in the case of strand notation and a calculus of operators, it will probably be necessary to develop a new language to apply arrays of functions in a general way.

References

[Abr70] Abrams, P.
An APL Machine.
PhD thesis, Stanford, 1970.

[ATC77] Alfonseca, M., Tavera, M. L., and Casajuana, R.
An APL Interpreter ans System for a Small Computer.
IBM Sys. J. 16(1):18-40, 1977.

[BeD82] Bergeron, J. and Dubuque, A.
A Structured APL System.
ACM Trans. Prog. Lang. Sys. 4(4):585-600, October, 1982.

[Ben84] Benkard, J. P.
Rank vs Depth for Array Partitioning.
In *APL84 Proceedings*, pages 33-39. ACM, Helsinki, June, 1984.

[Ben86] Benkard, J. P.
Analysis of Function Application of Deep Arrays.
In *APL86 Proceedings*, pages 202-210. ACM, Manchester, July, 1986.

[Ber84] Berry, Michael J. A.
Shared Functions and Variables As an Aid to Application Design.
In *APL84 Proceedings*, pages 57-62. ACM, Helsinki, June, 1984.

[Bro85] Brown, J. A.
A development of APL2 Syntax.
IBM J. Res. Dev. 29(1):37-48, January, 1985.

[Bud81] Budd, Timothy A.
An APL Compiler.
Technical Report 81-17, University of Arizona, 1981.

[Bud84] Budd, T. A.
An APL Compiler for a Vector Processor.
ACM Trans. Prog. Lang. Sys. 6(3):297-313, July, 1984.

[ChA81] Chu, Y., and Abrams, M.
Programming Languages and Direct-Execution Computer Architecture.
Computer 14(7):22-40, July, 1981.

[Chi86] Ching, W.-M.
Program Analysis and Code Generation in an APL/370 Compiler.
IBM J Res Dev 30(6):594-602, November, 1986.

[Chu79] Chu, Y.
Architecture of a Hardware Data Interpreter.
IEEE Trans. Comput. C28(2):101-109, February, 1979.

[Dei84] Dietel, Harvey M.
An Introduction to Operating Systems.
Addison-Wesley, Reading, Mass., 1984.

[DrO86] Driscoll, G. C. and Orth, D. L.
Compiling APL: The Yorktown APL Translator.
IBM J Res. Dev. 30(6):583-593, November 1986.

[Edw87] Edwards, E. M.
Private Communication.

[Eu85A] Eusebi, Edward V.
Operators for Program Control.
In *APL85 Proceedings*, pages 181-189. ACM, Seattle, May, 1985.

[Eu85B] Eusebi, Edward V.
Operators for Recursion.
In *APL85 Proceedings*, pages 190-194. ACM, Seattle, May, 1985.

[Fal73] Falkoff, A. D., and Iverson, K. E.
The Design of APL.
IBM J. Res. Dev. 17(4):324-334, July, 1973.

[FJW85] Flynn, M. J., Johnson, J. D., and Wakefield, S. P.
On Instruction Sets and Their Formats.
IEEE Trans. Comput. 34(3):242-254, March, 1985.

[FlH83] Flynn, M. J., and Hoevel, L. W.
Execution Architecture: The DELtran Experiment.
IEEE Trans. Comput. C32(2):156-175, February, 1983.

[FlH84] Flynn, M. J., and Hoevel, L.
Measures of Ideal Execution Architectures.
IBM J Res Dev 28(4):356-369, July, 1984.

[Fly80] Flynn, M. J.
Directions and Issues in Architecture and Language.
IEEE Trans. Comput. C13(10), October, 1980.
Secondary Reference, not yet verified.

[GhM73] Ghandour, Z., and Mezei, J.
General Arrays, Operators, and Functions.
IBM J. Res. Dev. 17(4):335-352, July, 1973.

[Gud85] Gudaitis, John J.
Evaluation of some Distributed Function Architectures For Array Processing Data Manipulation.
Master's thesis, Simon Fraser University, July, 1985.

[Gus86] Gustafsson, S.
Quaternions and Homogeneous Coordinates.
In *APL86 Proceedings*, pages 71-77. ACM, Manchester, July, 1986.

[GuW78] Guibas, L. J., and Wyatt, D. K.
Compilation and Delayed Evaluation in APL.
In *Fifth ACM Symposium on Principles of Programming Languages*, pages 1-8. ACM, 1978.

[HaL72] Hassitt, A., and Lyon, L. E.
Efficient Evaluation of Array Subscripts of Arrays.
IBM J. Res. Dev. 16(1):45-57, January, 1972.

[HaL76] Hassitt, A., and Lyon, L. E.
An APL Evaluator on System/370.
IBM Sys. J. 15(4):358-378, 1976.

[HGT86] Hobson, R.F., Gudaitis, J., and Thornburg, J.
A New Machine Model for High-Level Language Interpretation.
In *Proc. 19th Ann, Hawaii Intl Conf. Sys. Sci*, pages 132-139. January, 1986.

[HHH86] Huynh, T., Halpern, B., and Hoevel, L. W.
An Execution Architecture for FP.
IBM J Res Dev 30(6):609-616, November, 1986.

[HLL73] Hassitt, A., Lageschulte, J. W., and Lyon, L. E.
Implementation of a High Level Language Machine.
Communications of the ACM 16(4):199-212, April, 1973.

[Hob82] Hobson, Richard F.
A Directly Executable Encoding for APL.
Technical Report TR82-1, Simon Fraser University Computing Science Dept., 1982.

[Hob83] Hobson, R. F.
The SAMjr Microprogramming Guide.

[Hob84] Hobson, R. F.
A Directly Executable Encoding for APL.
ACM Trans. Prog. Lang. Sys. 6(3):314-332, July, 1984.

[Hsi85] Hsieh, J.-T.
Performance Evaluation of the Pipe Computer Architecture.
PhD thesis, University of Wisconsin, 1985.

[IPS84] Iverson, K. E., Pesch, R., and Schueler, J. H.
An Operator Calculus.
In *APL84 Proceedings*, pages 213-218. ACM, Helsinki, June, 1984.

[Ive83] Iverson, Eric.
Private Communication.

[Ive86] Iverson, K. E.
A Concise Dictionary of APL.
I. P. Sharp Associates, Toronto, 1986.

[JGM86] Jenkins, M. A., Glascow, J. I., and McCrosky, C.
Programming Styles in Nial.
In *Proc. 19 Hawaii Intl Conf. Sys. Sci. - Vol II*, pages 267-275.
Hawaii, January, 1986.

[Joh71] Johnston, J. B.
The Contour Model of Block Structured Processes.
SIGPLAN Notices 6:55-82, February, 1971.

[Kel73] Kelley, R. A.
APLGOL, An Experimental Structured Programming Language.
IBM J. Res. Dev. 17(1):69-73, January, 1973.

[Kno65] Knowlton, Kenneth C.
A Fast Storage Allocator.
Communications of the ACM 8(10):623-625, October, 1965.

[Lan86] Landaeta, D. J.
A Notation for Manipulating Arrays of Operations.
In *APL86 Proceedings*, pages 21-29. ACM, Manchester, July, 1986.

[Lio85] Liou, K.
Design of Pipelined Memory Systems for Decoupled Architectures.
PhD thesis, University of Wisconsin, 1985.

[McC85] McCrosky, Carl D.
ACE: The Array-Theoretic Computational Engine.
PhD thesis, Queens, 1985.

[McJ86] McCrosky,C., and Jenkins, M. A.
ACE: The Array-Theoretic Computational Engine.
In *19th Ann. Hawaii Intl. Conf. Sys. Sci.* pages 117-123. ACM. January, 1986.

[MCMxx] Micro Computing Machines.
MCM APL Users Manual.

[MGJ84] McCrosky, C. D.,Glascow, J. I., and Jenkins, M. A.
NIAL: A Candidate Language for Fifth Generation Computer Systems.
Technical Report 84-159, Queens University Dept. of Computing Sci., 1984.

[Mor73] More, T. Jr.
Axioms and Theorems for a Theory of Arrays.
IBM J. Res. Dev. 17(2):135-157, March, 1973.

[PaH86] Page, Ivor P., and Hagins, Jeff.
Improving the Performance of Buddy Systems.
IEEE Trans. Comput. C35(5):441-447. May, 1986.

[PeN77] Peterson, J. L., and Norman, T. A.
Buddy Systems.
Communications of the ACM 20(6):421-430, July, 1977.

[Per74] Perlis, Alan J.
Steps Toward an APL Compiler.
Research Report 24, Yale University Department of Computer Science, January, 1974.

[PeS83] Peterson, J. L., and Silberschatz, A.
Operating System Concepts.
Addison-Wesley, Reading, Mass., 1983.

[PMI86] Pesch, R. H., McDonnel, E. E., Iverson, K. E., Bernecky, Bob, and Allen, D. B.
Minnowbrook APL Workshop, October 1-4, 1985.
APL Quote Quad 16(3), March, 1986.

[PuS70] Purdom, P. W., and Stigler, S. M.
Statistical Properties of the Buddy System.
Journal of the ACM 17(4):683-697. October, 1970.

[Qui86] Quinlan J.
A Comparative Analysis of Computer Architectures for Production System Machines.
In *Proc. 19th Ann. Hawaii Intl Conf. Sys. Sci. - Volume I.* pages 187-193. Hawaii, January, 1986.

[RuM86] Rubinstein, D. L., and Murray, W. D.
Evaluation of Functional Programming as a Basis for Computer Architecture.
In *Proc. 19th Ann. Hawaii Intl Conf. Sys. Sci. - Volume I*, pages 53-58. Hawaii, January, 1986.

[Sah78] Sahl, Harry J.
Considerations in the Design of a Compiler for APL.
APL Quote Quad 8(4):9-14, June, 1978.

[SAL78] Seeds, G. M., Arpin, A., and Labarre, M.
Name Scope Control in APL Defined Functions.
APL Quote Quad 8(4):15-19, June, 1978.

[Sny82] Snyder, Warren.
MAPLE: Multiprocessor APL machinE.
Master's thesis, SFU, February, 1982.

[Soo84] Soop, Karl.
Can an APL Workspace be Used as a Data Base?
In *APL84 Proceedings*, pages 303-310. ACM, Helsinki, June 1984.

[SWP86] Smith, J. E., Weiss, S., and Pang, N. Y.
A Simulation Study of Decoupled Architecture Computers.
IEEE Trans. Comput. 35(8):692-702, August, 1985.

[TaW84] Taylor, S., and Whitney, A.
The One Tree (Breaking Out of the Workspace).
In *APL84 Proceedings*. ACM, Helsinki, June, 1984.

[Tho83] Thornburg, Jonathan.
Function CALL/RETURN model written as undergraduate project.

[TrB82] Treat, J. M., and Budd, T. A.
Extensions to Grid Selector Composition and Compilation in APL.
Technical Report 82-7, University of Arizona Dept. of Computer Science, 1982.

[Van77] Van Dyke, Eric J.
A Dynamic Incremental Compiler for an Interpretive Language.
Hewlett-Packard Journal 28(11):17-23, July, 1977.

[Veg84] Vegdahl, S. R.
A Survey of Proposed Architectures for the Execution of Functional Languages.
IEEE Trans. Comput. 33(12):1050-1071, December, 1984.

[Wah87] Wah, B. W.
New Computers for Artificial Intelligence Processing.
Computer 20(1):10-15, January, 1987.

[Wei79] Weidmann, Clark.
Steps Toward an APL Compiler.
In *APL79 Proceedings*, pages 321-328. ACM, Rochester, 1979.

[Wgg85] Weigang, Jim.
An Introduction to STSC's APL Compiler.
In *APL85 Proceedings*, pages 231-238. ACM, Seattle, May 1985.

[Wgg86] Weigang, J.
MAchine-Oriented Languages in the APL Environment.
In *APL86 Proceedings*, pages 125-131. ACM, Manchester, July 1986.

[Zak78] Zaks, R.
A Microprogrammed APL Implementation.
Sybex, Berkely, Calif., 1978.