AFS FUNDAMENTAL CONCEPTS AND SYSTEM LANGUAGE

THIRD EDITION

March 8, 1971

IBM CONFIDENTIAL

This document contains information of a proprietary nature. All information contained herein shall be kept in confidence. None of this information shall be divulged to persons other than: IBM employees authorized by the nature of their duties to receive such information, or individuals or organizations authorized by the Systems Development Division in accordance with existing policy regarding release of company information.

AFS FUNDAMENTAL CONCEPTS AND SYSTEM LANGUAGE

TABLE OF CONTENTS

Part 1: Introduction

Part 2: Basic Concepts and Structures

Part 3: System Concepts and Facilities

Part 4: The Man-Machine Interface in AFS

Part 5: A Logical Implementation

Appendices

PREFACE

This is the third edition of the AFS logical architecture by the Poughkeepsie Advanced Systems Group. It is a refinement and extension of the second edition and is presented as a basis for further work and as a vehicle for communication between the several groups working on AFS. Although the design effort has concentrated on the conceptual level, it is being supported by concurrent implementation studies that are discussed in the AFS System Architecture Manual.

GUIDE TO READING THIS REPORT

People with different backgrounds will find it expedient to approach the study of this material in different ways. This guide suggests a reading sequence for engineers, programmers, and system analysts.

1)	All:	read	1.1.1	A One-Page Summation
2)	All:	read the rest of	1.1	Executive Summary
3)	All:	study	2.1.1	Storage
			- 2.1.2	Processes
			- 2.1.3	Objects
			- 2.1.4	Access Machines
			- 2.2.1	Key Concepts
4)	All:	skim the rest of	2	
5)	engineers:	study	5	A Logical Implementation
	programmers:	study	4.3	A Summary of Basic Infix Form
	system analysts:	study	3	System Concepts and Facilities
6)	All:	study	2	Basic Concepts and Structures
7)	All:	study		The rest

Part 1

INTRODUCTION

Since AFS is developing a new approach to computer system design, some background information is necessary to place the concepts in perspective and to ease the transition to novel lines of thought. Chapter 1.1 presents an overview of the new concepts, the relationship between AFS and other developments by IBM and competitors, and the objectives and requirements that AFS is trying to meet. Chapter 1.2 discusses underlying assumptions that motivate and direct the design effort. Finally, chapter 1.3 presents the notation and syntactic conventions used throughout the remaining parts of this manual.

Chapter 1.1

EXECUTIVE SUMMARY

1.1.1 A One-Page Summation

AFS, Advanced Future System, is proposed as an alternative to compatible extension of System/370. It is intended to meet FS Market Requirements by Advanced Systems Planning and Evaluation. Basic elements of AFS include self-describing data, reference to data by symbolic names rather than addresses, dynamic attribute examination, automatic storage hierarchy, network function transparency, and a high-level machine language called SL, the System Language. Such a functional base will provide a significant gain in system usability. This document presents a new conceptual foundation, and describes SL and the associated system facilities. A companion document, the AFS System Architecture Manual, discusses implementation and presents additional detail.

The conceptual foundation for AFS is a synthesis of advances in Computer Science. It is modeled formally using the Vienna definition methods. It provides a framework for multiprocessing, data independence, data base structures, source/sink and network communications, modular control system structure, uniform resource management, and migration from System/360/370 including coexistence and dynamic interchange.

The number of AFS constructs is minimized by exploiting each fully. For example, assignment is the universal means to put something somewhere, whether assigning a value to a number, sending information to a printer, or filing a new program under some name. Similarly, an "object" has the same formal structure whether it represents numeric data, a data structure, a virtual device, a program environment, a function activation, an access authorization, a communication port, or any other system entity.

SL is a complete language, whose functions include those necessary to represent programs written in contemporary high level languages, as well as all system control facilities. SL statements are constructed with these functions just as arithmetic expressions are constructed with arithmetic operators. A customer may use COBOL, PL/I, FORTRAN, APL, or RPG as if each were the actual machine language. SL is extendible: new functions and data structures are readily accommodated. Furthermore, the AFS design is such that facilities beneath the external interface may be redefined with SL functions.

1.1.2 Background

The conceptual foundation results from a fresh examination of fundamental data and control structures in light of the past decade of progress in computer science. The approach differs from earlier ones in that provision is made from the outset for essential FS ingredients such as multiprocessing, data independence, data base structures, coexistence of multiple architectures (such as System/370), network communications, applications subsystems, and unified system resource management.

The SL design also differs from earlier approaches in basic character: The conceptual framework provides a basis for an architecture which can grow gracefully, rather than one which is tightly circumscribed. Extensions and modifications can be defined in SL itself in such a manner that system discipline and integrity pervades all levels of redefinition; user programs are written as though the extensions were an integral part of the system.

This type of design is called a Recursively Extensible Architecture. It offers users the ability to extend or specialize subsystems for their particular requirements, system architects the ability to develop the architecture without impacting customer programming investments, and IBM product designers the opportunity to build hardware to support either general or specialized functional extensions.

1.1.2.1 Historical Foundation

Design of the data and control structures required for a complete, functioning system has historically been the task of programmers. In the process of building increasingly complex systems, a systematic body of programming knowledge has developed. Central to this body of knowledge is an understanding of fundamental structures and algorithms which occur throughout all programming practice. Work in programming languages over the past ten years has to a large extent consisted of developing notations with which one can conveniently employ various subsets of these basic elements. The SL approach has been to survey the fundamental structures, determine a minimal set of basic concepts, and design a total external interface based upon this set.

1.1.2.2 Related IBM Activities

There are a number of current activities that relate directly or indirectly to AFS. System A in Research is examining an external interface similar to SL: System A is designed to run on an NS symmetric multiprocessor system, and programs at the external interface level will either be compiled into System/370 code or be interpreted in an intermediate language similar to SL. The Endicott Advanced System Group has worked on a similarly motivated design effort during the past several years. Their work through 1970 is summarized in a February 1, 1971, report entitled HLS-Prototype Project Report. More recently, Endicott ASG representatives have worked both with the SL designers and with Ray Larner, who has formulated a proposal for a high level interface called ML (Machine Language). Several individuals in the San Jose Research Center have been actively participating in AFS areas. The Palo Alto Scientific Center has microcoded a Model 25, and now a Model 145, to interpret APL code directly. They have also conducted related studies concerning the performance of microcoded APL machines vs. conventional instructions and compilers. Much of the work on data base organization is pertinent, especially the PROP/DB prototype in Poughkeepsie. The New York Programming Center is studying the significance of an AFS-like architecture for the principal programming languages, and the broader classes of languages and language building tools which may become possible. Prototype PL/I work done in Hursley, in conjunction with the functional memory program, has shown several opportunities for significant performance improvement. Work to date on the FPS project has considered similar concepts, and it seems that some commonality with the eventual FPS direction is likely.

1.1.2.3 Competition

Numerous university and industrial investigators are exploring AFS-like directions. Some are exploring these directions with the intent of developing more efficient microcode for existing hardware. Examples can be found in papers emanating from universities. Some manufacturers are producing microcodable hardware which lends itself to providing higher level interfaces. Examples are the ICL and Gemini machines. There is considerable discussion of APL-like machines; CDC claims that the STAR system directly performs APL-like functions. McFarland's paper in the 1970 FJCC describes TPL (The Programming Language), for which direct hardware support is discussed. Iliffe's Basic Machine and Rice's "PL/I" machine are further examples of machines which offer direct support of higher level external interfaces. By far the most experienced competitor to date is Burroughs: The B5000 in the early sixties and the more recent B5700, B6700, and B7700 all support a higher level interface directly. Their architecture readily offers support, such as virtual memories and multiprocessing, which poses serious difficulties for OS or DOS. Their design has permitted construction of an operating system in a higher level language. Further development in AFS directions should be anticipated from Burroughs.

1.1.3 Objectives and Requirements

AFS is intended as an alternative to a compatible extension of System/370 for the FS time frame. AFS must therefore meet official FS Market Requirements rather than generate new ones. In the event that any of these requirements are not achievable, AFS has the objective to equal or exceed the best FS proposal with System/370 compatible hardware.

SL is the machine language of AFS and therefore inherits the above requirements and any other AFS requirement that has a language implication. At present, these requirements are stated in a memo, "AFS Requirements and Objectives" Jan. 19, 1971, to C. J. Conti and A. A. Magdali from R. B. Bennett and W. D. Wilson. A brief summary of the requirements from the SL point of view is given below:

SL must allow the user to interact with AFS in a high level language and suffer neither the isolation from the machine caused by compilers today nor the inefficient execution caused by interpreters. This is to be accomplished in two ways: on the one hand, the machine language itself will be a high level language exploiting current language technology; on the other hand, the user will be able to act as if the machine language were any one of five favored languages--COBOL, FORTRAN, PL/I, RPG, and APL--and he must not suffer a serious performance penalty for ignoring machine language.

To meet this requirement, SL must faithfully interpret the five favored languages: Under AFS, the conversational user must be able to interrupt execution, make changes, resume execution, execute incomplete or defective code as long as it makes sense to do so, and get the full benefits of a really good interpreter of the language without paying the performance penalty normally associated with interpretation.

SL must be an appropriate object language for the interpreters mentioned above and for compilers from the current principal high-level languages, extensions that will be made to them, and new programming languages that may become popular in the FS time frame.

Security, privacy, and system integrity must be provided to protect one user from another and to protect the system from the users.

An objective of SL is to fulfill the above requirements by, among other things, designing a system with self-describing data. To this end, attribute examining hardware should enhance both security and system integrity and fulfill the additional requirement of making it possible to restructure data without invalidating programs.

The design of SL must allow more efficient implementation with LSI than would be possible if the high-level source language were translated to a low-level machine language implemented with LSI.

SL must be extendible to accommodate new operators, new data types, and new devices. It must also enforce constraints that encourage more disciplined use.

SL must accommodate programs that exploit new market areas: particularly data base systems, data communication systems, transaction-based applications, and interactive use. These new areas must be accommodated without losing ground in what will continue to be a major market, batch computation in established applications.

AFS must emulate System/370 with twice the cost/performance. When the customer makes the transition to native mode AFS, there must be a four to one gain in price performance over System/370. The customer must be able to make the transition in a piecemeal fashion. The part of an application that has been translated to AFS native mode must exhibit AFS properties; for example, translated parts must exhibit user security and system integrity that is unachievable in System/370.

To aid a customer's transition, PL/I, FORTRAN, COBOL, RPG, and APL as executed by AFS must meet standard specifications for the languages.

1.1.4 Design Principles

SL has been constructed with a number of specific design principles in mind. They are each discussed in Section 1.2.5. They are:

Minimum Number of Basic Concepts
Completeness of Basic Concepts
Rigorous Control and Access Disciplines

Maximum Hardware Design Freedom
Network Function Transparency
Bit Code Independence
Modifiability
Extensibility

1.1.5 Basic Concepts

Key èlements of a high level interface and of a machine that directly supports the interface, have been described in several earlier reports, such as the McPherson task force report and the Endicott HLS Prototype reports. The machine is partitioned into functional units for processing, storage management, and source/sink and network communications. The interface includes self-describing data, generic operators, separation of storage from communications I/O, and provision for coexistence and interaction of data and program material produced for dissimilar architectures (such as System/370, System/3, 7090, 1401, etc.).

Producing a design capable of integrating these key elements requires more than simply defining a particular external interface. A formal conceptual foundation must first be erected in which it is possible to exhibit basic elements, structures, mechanisms, and key processes with which one can realize and prove proper behavior not only for computational processes, such as arithmetic expression evaluation, but also for essential system functions such as coexistence, multiprocessing, data base, networks, and dynamic resource management. To date, most of these aspects have simply been left for the system programmer to solve. Experience has made it clear that system design cannot continue to ignore such matters. This is especially true for systems such as AFS.

The conceptual foundation for SL consists of three basic elements: Process, Storage Cell, and Object; three basic structures: Accessibility Graph, Environment Tree, and Dependency Graph; two classes of basic mechanisms: inter-object communications protocol and inter-object request/response handling; and five key processes: translation, expression interpretation, symbol resolution, procedure activation, and resource management.

A process designates an algorithmic activity. It consists of a motive force called an interpreter, a procedural description, and a set of status information called the PSR (Process Status Record). A storage cell is the basic unit of storage. It is identified by a unique internal identifer called a Cell Name, and it contains exactly one object. An object is an entity used to represent every logical and physical resource of the system. It has an active subelement, a process called an Access Machine, and a passive subelement, called an Owned Resource. Every reference to the owned resource is accomplished by activation of the access machine. This model permits uniform representation and handling of all system resources.

The accessibility graph defines the paths by which objects may be reached. It contains a subgraph, a tree called the Ownership Tree, which defines ownership among objects. The environment tree defines the context in which symbols appearing in program modules are resolved to particular objects. The dependency graph records dynamic dependencies among objects. It includes a subgraph called the Activation Tree, and it is used by resource management.

The names of the basic mechanisms and key processes directly suggest their respective roles.

By using the above constructs, a conceptual foundation of the necessary type has been defined. The definition methods developed by the Vienna Laboratory (VDL) were employed to ensure formal completeness. SL represents a particular interface definition within the conceptual framework.

1.1.6 System Concepts

Part 3 of this document discusses the manner in which the SL conceptual foundation serves as the basis for a total operating system that meets FS market requirements. Of particular concern has been consideration of resource management, user environment, system control, and functional capabilities.

Resource management encompasses handling of both nonunique resources such as storage and unique resources such as particular data elements. A resource management policy is adopted which will ensure completion of all jobs submitted to the system. The system can be so structured that it is possible to prove that resource conflicts never occur in vital portions of the system. Errors occurring elsewhere are prevented from propagating to other parts of the system. Individual users are offered the option of avoiding deadlocks altogether by stating resource requirements in advance, or of dynamically requesting resources at the cost of possibly having to back out of deadlock situations.

! WHAT CAN 'CONFLICT' MEAN HERE!

The AFS system effects a modular handling of user environments. All resources of the system, including ports to the outside world, are owned by the resource manager. The operating systems, defined as subsystems in SL, through which a user may wish to avail himself of AFS facilities are also owned by the resource manager under the subsystem landlord. Each subsystem claims, and is allocated if available, a package of resources which it may control and allocate to the user via its own subsystem resource manager. Some operating systems may be granted a "semi-permanent" (e.g. "IPL" to "shut-down") status in the system, existing for long periods of time and servicing many users; such dedicated subsystems may have direct, implicit control over a set of ports. Thus, a user entering the system via any of these ports sees only that operating system and feels as though he were running on that subsystem's host architecture; this is the logical equivalent of virtual machines and permits users of, e.g. OS/370, to run as though they were on System/370. Users entering the system through ports not directly controlled by dedicated subsystems first encounter the initial interpreter, through which they may request the creation of a free subsystem for their private or shared use. The subsystems thus established are transient and are granted access to resource packages minimally including the active port and the user's files. Once running under a subsystem (SL itself is an example), the user may request the dynamic creation of additional subsystems for concurrent or consecutive, interactive or batch, dependent or independent, execution. A user job, in the classical sense, is thus initiated at port sign-on times and terminated with sign-off; dynamic subsystems created in the interim may become jobs at the user's explicit request.

The system control structure is based upon partitioning system activity into functional and server configuration levels. Work flow on the functional level handles initiation, coordination, and termination of communication, data entry, data retrieval, and computation functions. On the server level, which is beneath the SL level, control is concerned with orderly flow of work through the system, including control and synchronization of both logical and physical resources.

Consideration of system functional capabilities includes particular concern regarding data base, data communications, and coexistence.

SL objects and data structures provide convenient representations for the data aggregates and indices required for either ring-structure or entity-set data organizations. Access machines and the accessibility graph can be used jointly to enforce privacy and security.

At the SL level the user deals with processes involving data communications by use of objects known as Ports. The access machines of Ports provide the bridge to deeper levels of communication control. The deeper control levels include one which performs device independent formatting, and another which handles device function dependent and inter-system protocols.

Data tranmission protocols for line control and network (path) management are handled in the communications unit beneath the SL level.

The access machine also provides a possible basis for coexistence and interchange of (virtual) devices and other systems written under differing architectures. The access machine is a process which is activated whenever a request is made upon the object of which it is a part. The interpreter and procedural description of an access machine need not be of the same architecture as the process making a request upon the access machine. SL code can therefore call System/370 code in a rigorously disciplined manner, and vice versa. This mechanism also enables one software subsystem to access data in another, even if the subsystems have different architectures.

1.1.7 The External Interface, SL

Part 4 of this document describes the basic infix form of the SL language. It is this form which constitutes the primary man-machine interface of the AFS system. Each SL function is described separately, along with examples of its use and discussion of its side-effects. (This level of description of SL is only partially complete in Edition 3.) Examples of translation of high level language constructs to SL are also presented.

1.1.8 A Logical Implementation

Part 5 of the document presents a logical implementation of AFS. The definition methods developed by the Vienna Laboratory (VDL) were employed, in order to insure formal consistency and completeness. This approach turned out to be particularly effective for this level of design work. The presentation in Part 5 is an English transcription of the formal implementation rather than one which utilizes the VDL notation.

The logical implementation of AFS describes the way the system operates on an abstract machine which models the concepts SL presents to an AFS machine language programmer. Any physical implementation that produces the same observable behavior is a proper concrete representation of AFS. System designers are free to realize the AFS system in the most economical fashion for each particular market. Slavish copying of the logical model would probably result in an inferior physical implementation. Such an implementation, therefore, is not recommended.

Chapter 1.2

DESIGN PRINCIPLES

1.2.1 Rationale of the AFS System

There is considerable evidence that a Von Neumann architecture is inadequate for future IBM systems: such an architecture is a poor target for compilers, the coding conventions are inefficient in the information theoretic sense, and the units of work encoded are not optimal for either large or small machines. Furthermore, the property of data independence, which is clearly required for future systems, is impossible, or at best prohibitively expensive, with an architecture in which attributes of data are sprinkled throughout every instruction that references the data. There is also a serious question as to whether a system based upon Von Neumann instructions can guarantee the security and integrity that future systems must provide.

Another problem that must be corrected is that present hardware/software systems require the user to understand much more than he needs to know to do his work. A solution to this problem in a limited context has been provided by certain conversational systems like JOSS, CPS, and APL. In these systems, the user is not required to learn unrelated languages like machine language or JCL in addition to the language in which he writes his program. Furthermore, he has good conversational access to what is going on: if he does something wrong, he is likely to find out forthwith. With new architecture, these advantages will extend to the full range of problems that computers solve without incurring the performance penalty of a software interpreter.

During the past decade, considerable practical and theoretical work on programming languages has been done. Although centered around language, this work has analyzed structures that are fundamental to all forms of computation: the structures are common to many types of languages and appear throughout operating system design. The time is ripe, therefore, to focus upon these basic structures, to implement them directly in hardware, and to construct the architecture of an entire system upon the foundation they form.

1.2.2 Interface Levels

At present, five basic architectural levels have been identified:

1) Physical Components
2) Hardware Boxes
3) System Control
4) System Language, SL
5) General User

This document discusses the logical aspects of the interface between levels 3 and 4. The AFS System Architecture, of course, must define the details of all interfaces. Several observations should be made on the interface between SL and System Control.

An AFS system, logically, makes available to a user through the SL interface a set of system services in data communications, data entry/retrieval, and data manipulation and computation. Beneath the SL level, the control and synchronization of system work flow is under the control of a System Control program. The System Control program is architected to consist of a number of functional control modules, Terminal Control, Data Communications Control, Data Control, Monitor Control, and Command Control. The Command Control module has the responsibility to coordinate work flow activities on both the logical and physical levels. On the physical level System Control functions are mapped onto a physical structure which consists of three basic engineering subsystems, PPS(Program Processing Subsystem), SMS(Storage Management Subsystem), and SSS(Source/Sink Subsystem). Each of these units requires its own logical control program, which will be called an ECP (Engineering Control Program). The SL/System Control interface is common across all AFS installations. Within the System Control level, the SCP interacts with the interface provided by the respective ECP's. This interface will be called the EI (Engineering Interface).

1.2.3 Logical and Physical Interfaces

In early computer systems, logical and physical interfaces were identical: programming manuals included a rough sketch of hardware organization, describing registers, data paths, and CPU clock cycles. In System/360, IBM introduced a family of computers with identical logical interfaces, but totally different physical organizations and data flow. Software developments removed the programmer even further from hardware: with pseudo-devices in HASP and virtual machines in CP/67, programming interfaces became purely logical, with no direct relationship to physical devices.

A lesson from history shows the importance of separating logical and physical interfaces: On the IBM 704, all I/O went through the MQ register in the CPU; a programmer could overlap I/O and computation only by complex programming techniques involving delicate timing considerations. The IBM 709 added channels to allow I/O transfers to proceed without interfering with computation, but each type of I/O device required a different set of control instructions. System/360 simplified the logical interface by adding control units that responded to the same type of command for an entire class of devices, but the proliferation of channels and control units increased the number of hardware devices and hence total system cost. To reduce cost, small models like System 360 Mod 25 used CPU logic to perform the functions of channels and control units. After a decade of progress, physical interfaces on the Mod 25 were the same as on the 704, but logical interfaces were totally different: because of functional differences between I/O and computation, computer architects had defined logical interfaces that separated channels and control units from the CPU; on the assumption that every logical interface requires a physical interface, they had designed different hardware devices for every functional unit; to improve cost/performance, engineers eventually found ways of doing all the functions on a single unit. The moral is that logical interfaces are programming aids, physical interfaces are engineering approaches to better cost/performance, and any similarity between the two is purely coincidental.

Not so!

The AFS project involves a critical analysis and redefinition of all interfaces in an information handling system: the programmer's interface should be a purely logical one with all the aids that can simplify his task and with no housekeeping details; the physical interface should be designed for optimum performance at a given cost with no unnecessary constraints from the programming interface.

1.2.4 Facilities Beneath the Logical Interface

Before considering what features future systems should have, let us contemplate the state into which current systems have evolved. For our hardware, assume a hypothetical Model 195 with relocation features and a modified CP/67 system to run on it. Then imagine a PL/I program using disk I/O running under OS/360 running on the modified CP/67 running on the hypothetical Model 195. Storage management on such a system is fantastic: First, the PL/I program must manage transfers between its own storage and the disk file. Beneath the PL/I interface, the compiler inserts storage management routines to suballocate storage faster than OS/360 can with GETMAIN and FREEMAIN. On the next level, OS/360 allocates space to the program and parcels it out in response to

GETMAIN's; it also allocates space on its virtual 2311 disk and does housekeeping for I/O requests. On the next lower level, CP/67 creates the illusion of storage and disk for OS/360: it busily allocates space in core, moves virtual pages to meet the demand, and conjures up a 2311 out of space in core, drum, and 2314 disk. Meanwhile, hardware allocates blocks of space in the high-speed buffer and moves data to anticipate future use; it also allocates space in various registers invisible to the programmer: instruction buffers, data buffers, and reservation stations that effectively replace the floating-point registers with a set of virtual registers. The point of this example is that storage management occurs at every level of current systems: allocations done at one level are frequently undone at the next; most of the allocations are done by software; and storage allocation by hardware is about two orders of magnitude faster than allocation by software.

As the preceding example showed, storage management by operating systems is inefficient compared to management by hardware and is inadequate to eliminate further management by problem programs. Processor allocation and task dispatching can also be performed by hardware: super computers like the Model 195 or MPS have built sophisticated multiprogramming algorithms into hardware; even a small machine like the Model 25 does hardware dispatching every time the CPU converts itself into an I/O channel; and multiplexor channels are hardware units designed to appear like many channels by internal multiprogramming. A control block is a kind of descriptor that is processed interpretively; Burroughs has been building machines for the past decade that do much, but not all, of descriptor processing by hardware. Compilers, linkage editors, JCL interpreters, indexed sequential access methods, and thousands of problem programs all do symbol resolution and linking, and they could all do it much more efficiently with hardware assistance. Establishing a new environment is done by hardware at every change of PSW and whenever a CPU becomes a channel; Burroughs systems also use hardware to switch environments for procedure calls. On modern systems, these functions occur more frequently than floating point multiplies and divides and are more fundamental to overall system operation. For optimum cost/performance, these functions should be reduced to a set of primitives that are as firmly supported by hardware as floating point arithmetic.

SURE, EVERYTHING SHOULD BE THE SUBJECT OF HWE/SFTWRE TRADEOFFS.

1.2.5 Design Principles

In order to design a system of the greatest possible utility, a number of design principles have been adopted as objectives. Ideally, the AFS system should exhibit properties derived directly from these principles:

1) Minimum number of basic concepts: Current systems suffer severely from constructs that are seemingly pulled out of the air with little regard for consistency or uniformity. Every effort is being made to design SL with a minimum number of basic concepts.

2) Completeness of basic concepts: Although few in number, the basic concepts must encompass all structures required for the AFS System. Separate operating system or command constructs, such as the system structure built around the APL language, must be obviated.

3) Rigorous control and access disciplines: The AFS design must make it possible to prove that system disciplines required for security and integrity are enforceable.

4) Maximum hardware design freedom beneath SL: The design should avoid constraining the manner in which hardware interprets it since different AFS machines may employ quite distinct internal representations.

5) Network function Transparency: The architecture should ensure functional transparency to user application programs and most system facilities of the physical network location - virtual (co-existent), local, or remote - of devices and other systems. Further, it should easily allow data and functions to be logically transparent to users.

6) Bit code independence: The internal bit codes used to represent SL should not be defined as part of the architecture. A standard representation for compiler output will be defined, but all bit structures within the system will be generated by execution of SL operators. Inverses of these operators are necessary to display internal structures for analysis and debugging.

7) Modifiability: The architecture should contain provision for user redefinition of system operators. The user should be able to incorporate suitably disciplined procedures in place of those normally supplied by the system. Architecturally, this requires that system primitives are themselves redefinable in terms of the system. Fully generalized, this principle requires the architecture to be recursively extensible.

8) Extensibility: The user should be able to define new operators that operate within his own contexts and to extend the definition of old operators to new classes of data.

Chapter 1.3

LEVELS OF LANGUAGE DESCRIPTION

1.3.1 Levels of Syntax

Three levels of SL are significant to the user. These are all symbolic in the sense that actual addresses and other machine oriented quantities are not accessible to the user; they are only represented in SL by symbols.

Strict SL is a machine oriented level that is most convenient for compilers to generate. Basic infix SL has the same operators as strict SL, but it has a format that is more congenial for people and can be mapped almost one-to-one into strict syntax. Following is an expression in strict syntax:

```
     stow(quotient(sum(A;B);sum(C;D));E)
```

In basic infix, the example becomes

```
     ((A+B)÷(C+D))->E
```

or

```
     A+B÷(C+D)->E
```

Extended infix is the most fully developed SL syntax. It incorporates basic infix as a proper subset. Extended infix will be supported by a software translator that will map it to strict syntax. The purpose of extended infix is to provide a flexible programming tool for those who wish to work directly with AFS data structures.

APL and LISP are expression oriented languages: the result of every operation is a value that can be used as input to another operator; consequently, experienced APL programmers often write subroutines consisting of a single expression with dozens of functions and variables; in LISP, an entire program is normally one long expression. The syntax of APL or LISP has both advantages and disadvantages: its advantages include simple syntactic rules with only one statement type and freedom from arbitrary conventions, a context free structure that allows any operand to be replaced by an expression that computes the same value, and a consistency that makes programs a subset of the list structures allowed for data; a possible disadvantage of such syntax is that it sometimes leads to long statements that are hard to read. Although long statements may obscure the programming style, they arise from the great modularity of languages that can combine small expressions in an endless variety of ways. Rather than restricting the power of the system, AFS will provide a general expression oriented language together with programming aids that encourage a clear, disciplined style.

As an example of the power of generalization and the expression oriented structure, consider a program to read records indexed by the variable CURRENT from files JOE and SAM and then write the smaller of those two records on the file TOM indexed by CURRENT. PL/I requires the following four statements to perform the task:

```
     READ FILE (JOE) INTO (TEMP1) KEY (CURRENT);
     READ FILE (SAM) INTO (TEMP2) KEY (CURRENT);
     TEMP1=MIN(TEMP1,TEMP2);
     WRITE FILE (TOM) FROM (TEMP1) KEYFROM (CURRENT);
```

The first observation we might make about these statements is that although they perform actions very similar to the fetching and storing of single elements of vectors, PL/I syntax obscures the similarity. The second observation is that PL/I chops expressions into statements that force the user to create unnecessary temporary variables as targets of the READ's. In SL, the similarity between indexed vectors and indexed sequential files is reflected in the language, and the fact that every expression has a value allows all four PL/I statements to be condensed into one SL statement:

```
     JOE[CURRENT] min SAM[CURRENT] -> TOM[CURRENT];
```

1.3.2 Strict Syntax

Although bit encoding of the machine language is not a primary topic of this report, a concrete notation is necessary for giving examples and stating definitions precisely. Therefore, all definitions will be stated in a form called the AFS strict syntax. This form is a direct mapping of the tree structure of the abstract syntax and is isomorphic to the class of bit encodings that will be executed directly by hardware. Following are production rules for the strict syntax in the IBM standard metalanguage:

```
     group ::= { s-expr [; s-expr] ... }

     s-expr ::= symbol [argument-list] | group | constant

     argument-list ::= ( s-expr [; s-expr] ... )

     symbol ::= letter [letter|digit|underscore] ...
```

An s-expr is an expression in the strict syntax. More general expressions in the extended syntax are defined by their mapping into s-expr's. A group is a collective object whose elements are complete expressions; it corresponds to BEGIN-END or DO-END blocks in PL/I and to procedure and function bodies. The group is more general, however, because it returns a value and can be used in place of an ordinary variable or constant; furthermore, it has the structure of a list and can be indexed or concatenated with other groups. A complete expression forming one element of a group is called a statement; following is an example of a group with two statements:

```
     {stow(sum(sin(X);exp(cos(Y)));Z);sum(difference(A;B);C)}
```

The first statement saves the result of the computation in Z, and the temporary value is discarded when execution moves on to the next statement. The second statement computes (A-B+C), which is returned as the value of the group. This form of syntax has a structure that is good for compilers, but bad for humans; the extended syntax is an infix form that is good for humans and directly mappable by compilers.

1.3.3 Extended Syntax

Although the strict syntax presented above is mathematically elegant, it suffers from the LISP unreadability syndrome: it uses too many parentheses, prefix notation is harder to read than infix, and arithmetic expressions are not written in familiar forms. The sample expression given in section 1.3.2 may be written in infix form as:

```
     {sin X+*cos Y->Z;A-B+C}
```

To improve readability, extra blanks and parentheses may be inserted, familiar mnemonics like 'exp' may be used instead of single character operators, and comments in French quotes may be inserted anywhere blanks may appear:

```
     {sin X + exp cos Y -> Z; (A-B+C) <<value of group>> }
```

The extended syntax will also include additional forms that are familiar from other programming languages such as if expressions and do-loops. Since a group is a list of expressions, an if expression can be constructed by indexing. For example, all three of the following expressions

```
     if A=B then X+3->Y else Y-3->X end

     {Y-3->X;X+3->Y}[A=B]

     A=B select {Y-3->X;X+3->Y}
```

can be converted to the strict syntactic form

```
     select(eq(A;B);{stow(difference(Y;3);X);stow(sum(X;3);Y)})
```

1.3.4 Character Set

The character set for a programming language must be a reasonable compromise among many conflicting constraints:

1) Ease of program entry,
2) Readability,
3) Use of familiar conventions,
4) Availability of existing and future I/O devices.

For good readability and an esthetically pleasing text, a large character set is important: studies of reading speed show that average readers can read lower case text much faster than text printed in upper case only, and mathematicians use a large character set to reduce long formulas to a size that can be more easily encompassed by the eye. APL has had considerable success in introducing a number of special characters for various functions, but rigorous adherence to the convention of single character operators leads to absurdities like "1 circle X" for sin(X) and "I-beam 20" for time. A large character set can unfortunately introduce problems in program entry: the reversal operator in APL requires 5 key strokes--upshift, O, backspace, upshift, M--and takes more typing effort that a three-letter word. I/O devices for 88-character keyboards are common, and even larger keyboards will become practical with CRT devices, while limited character devices like keypunches will be less common in the FS time frame. Nevertheless, character sets with about 80 or 90 symbols will still be more accessible than those with upwards of 150 symbols. Therefore, SL should assume that the basic form of input will be with a character set of 88 symbols, but it should make provision for devices with a smaller set and take advantage of future devices with larger character sets.

The proposal currently being considered for the SL external syntax is the set of conventions adopted by PAL: all user defined symbols are either single lower case letters or alphanumeric strings beginning with an upper case letter; reserved words and system defined symbols are either special characters or strings of two or more lower case letters. This convention includes the APL conventions as a special case, but it also provides an infinite number of words with mnemonic significance like sin, cos, time, date, if, and then. Furthermore, every special character would have a corresponding symbol like 'sum' for '+' so that devices without that character could still use the function; for devices without lower case letters, an escape character could be used to indicate reserved words.

Part 2

BASIC CONCEPTS AND STRUCTURES

This part of the manual describes the logical structures that are visible to system programmers and to user programmers who choose to code in SL. Although SL is the machine language for AFS, its concepts reflect the structures of compilers and operating systems much more than details of typical von Neumann machines. Three characteristics distinguish the following presentation from the principles of operation of other machines: the absence of bit representations, a theoretical style of definitions and theorems, and the basic assumption that traditional software functions of storage allocation and process dispatching are performed at the engineering level.

IN OTHER WORDS, ENGINEERS SOLVE MY PROBLEMS!

Chapter 2.1 begins with a discussion of objects: their residence in storage cells and their nature as processes. All the objects in the system make up the object base in which three directed graphs embody all interrelationships: the accessibility graph, which includes all possible paths for accessing one object from another; the environment tree, which defines paths for symbol resolution; and the dependency graph, which includes all outstanding requests by objects for services by other objects. Further discussion shows how these graphs interact with various types of objects, program structure, and resource management. The final chapter in this part discusses the built-in functions provided with the system.

Chapter 2.1

OBJECT BASE

2.1.1 Storage

A fundamental concept of AFS is that all storage internal to the system is managed automatically: the programmer refers to data and other objects by symbolic names rather than by physical addresses. Storage management would extend over levels from high-speed registers and monolithic memories up through Comanche files, optical storage devices, and even cataloged off-line storage such as tape libraries. Logically, all such storage is an integral part of the system, distinctions between levels are invisible to the programmer, and it is considered almost unlimited in size.

When independent formulations of a problem give rise to similar concepts, those concepts probably contain an essential element of the problem that is invariant under change of notation or frame of reference. The problem of distinguishing between objects and the mechanism for referencing them is a fundamental one that every computer system, programming language, and theory of computation must face: In von Neumann machines, a special type of data called an address is used to refer to other data; although addresses have the useful properties of numbers, they are bound so tightly to physical storage that their logical properties are inextricably confused with problems of allocating storage and devices. In the definition of CPL, Strachey distinguished L-values and R-values according to whether the value could appear on the left or the right of an assignment statement; the target of an assignment had to be a value with location-like properties. ALGOL 60 can be formally defined without the concept of storage only because it has a relatively small number of basic concepts; to deal with pointers and to formalize concepts of assignment, ALGOL 68 introduced the concept of a reference, which is like an address pointing to a cell capable of holding a given type of object. In his analysis of APL, Abrams distinguishes selection operators and computational operators: the value of a selection operator is linked to the storage of one of its operands and can transmit changes back to it; the value of a computational operator has no connection to the storage of its operands and cannot transmit changes back to them. One of the design principles of AFS is to search for the essential elements underlying all programming languages and to build a new system upon them; the concepts of object and storage cell are fundamental and require careful definition to support a general treatment of assignments, synonyms, ownership, and argument passing to functions.

For defining indices and pointers, storage addresses are useful, but the housekeeping they entail far outweighs their usefulness. The storage cell in AFS is a logical location capable of holding any object or collection of objects, no matter how large: its characteristics of a location simplify the definition of indices and pointers, but it involves no housekeeping burden because the storage management system makes the cell appear as large as necessary and automatically moves it to any device that may need to process its contents.

Definition: A storage cell is a logical location identified by a unique cell name. Each storage cell contains one and only one object; there is no upper limit on the size of a storage cell. The cell name is an internal identifier (abbreviated iid) whose representation is invisible to the user.

This definition is non-constructive: it defines a storage cell by axioms or characteristics that are visible to the programmer, not by an explicit construction from something more primitive. The advantage of non-constructive definitions is that the implementer has maximum freedom in his choice of representations and hardware design. The disadvantage of such definitions is that they don't prove that an efficient implementation (or even any implementation) is possible. To remedy that situation, the informal notes between definitions will illustrate the abstract concepts with a sample implementation; since the illustration will not necessarily be the optimum engineering solution, the implementers are free to use any design that satisfies the axioms.

Definition: A buffer is a temporary storage cell created for the purpose of holding an object until it can be processed or moved.

Buffers are intimately related to the mechanism for passing messages between objects such as arguments to functions and results from functions: Normally, what is passed is the cell name of some storage cell containing the message; in computing X[I], for example, the select function returns the cell name for the storage cell containing X[I]. However, when the sum function computes (A+B), there is no permanently allocated storage cell containing the result; therefore, the interpreter that is interpreting the function obtains a temporary storage cell, called a buffer, to hold the result. Buffers correspond to I/O buffers in current systems as well as to registers in the CPU or on a pushdown stack.

A particular implementation of the storage cell concept is discussed in the System Architecture Manual. The Storage Management Subsystem (SMS) described there provides spaces identified by unique space numbers; each space is linearly addressable by an offset from the beginning of the space. A collection of storage cells can be implemented as a space divided into a number of fixed length blocks holding object images, also known as DAPOVs (Descriptor And Pointer Or Value). The cell name corresponds to the space number and offset to the object image; the uniqueness of space numbers guarantees the uniqueness of cell names. If an object image is very large, the block identified by space number and offset only holds part of the image and contains the space number of another space holding the overflow. Since spaces can be chained together if necessary, there is no fixed bound on the size of objects.

2.1.2 Processes

The concept of process is fundamental to all levels of an information handling system: CPU, channels, operating systems, and external devices. A rationally designed system must have a precise concept of process and of the possible interactions between processes. In AFS, the definition of process is based on the well developed foundation of automata theory and is designed to facilitate the implementation of multiprocessing systems.

Definition: A process is an automaton with a set of states S and a set of states W contained in S in which it waits for input. Processes can be best described by assuming they have three parts:

1) A process status record (abbreviated PSR) containing the current state, input, and contents of buffers used for working storage. There is a one-to-one correspondence between processes and PSR's.
2) A procedural description that encodes a finite set of information defining the states and permissible transitions between those states. Some procedural descriptions may be shared by many processes.
3) An interpreter that performs state transitions for a process: it examines the procedural description and the PSR and sets the PSR to its next state. An interpreter may be time shared among a number of processes.

The process status record keeps track of all information that defines the current state of a process. In automata theory, a PSR is analogous to the instantaneous description of a Turing machine. In a System/360 CPU, a PSR is analogous to the program status word together with the contents of the fixed and floating registers. In the CDC 7600, the exchange jump package is the equivalent of the PSR. In the Burroughs 6700, the pushdown stack together with control words that may be stored in it form the equivalent of a PSR.

Above the SL level, a procedural description could be a read-only program. Beneath that level, procedural descriptions may be in microcode or hard wiring. The reason for separating the procedural description from other parts of a process is to allow a number of re-entrant processes to use the same description simultaneously. For primitive objects, the hardware may take shortcuts during high-speed execution and not separate the three parts of a process; for error logouts or responses to a diagnostic programmer, however, the system must generate a PSR that effectively represents the current state of an object.

The interpreter is the motive power that causes a process to move from one state to the next; it is the logical abstraction of active servers like CPU's and channels, but is more general since it includes software interpreters as well as special devices that may be attached as RPQ's. The AFS logical architecture has deliberately avoided the concept of a CPU; instead, the more general concept of process allows the engineer greater freedom to build distributed execution units, special purpose devices, and multiple processing units to improve performance without changing any logical interfaces.

The definition of process sets the stage for later discussion of wait states, exceptions, and suspensions: When a process needs input, it stays in one of its wait states indefinitely; a waiting process is considered asleep, and sending it input corresponds to a wake-up call. Exceptions are unusual conditions like arithmetic overflow or violations of access rights; when an exception occurs, the process in which it occurs generates a message for another process called a monitor and then goes into a wait state until it receives a message from the monitor. A suspension occurs when the motive force, the interpreter, is removed from a process, and the process naturally stops because there is nothing to make it go; suspensions result from time sharing the interpreter among many processes so that only one can be running at any given time, but they can also occur when a process has run out of money (using too much time or space) or when it is stopped because of some other event like an attention signal from the programmer who started it.

Processes occur at all levels of a system. When concepts are not clearly defined, engineers and programmers working on different levels may be unaware that they are facing similar problems and duplicating functions performed on other levels. In System/370, for example, there are processes executing in channels and I/O units, in microcode in the CPU, and at the instruction level for subroutines and tasks. The concepts, terminology, and data formats at the various levels completely obscure any similarity between these processes: records of processes in channels and I/O units are maintained in channel status words; the record of a process at the microlevel is logged out by the DIAGNOSE instruction; the record of the architecturally defined CPU status is in the program status word and register contents; and the record of a process as viewed by OS/360 is in the task control block. Not only does System/370 use awkward terminology for the various processes, it also uses awkward means for switching status: for subroutine calls, the BAL instruction does only half the job since it only modifies part of the PSW and it doesn't save registers. To call a program with different status, an SVC instruction must be used with considerable overhead from the operating system. The rest of the status, the registers, are at the mercy of the called routine to save or destroy. If the called routine is re-entrant, the simple BAL instruction, which takes one microsecond on a Model 65, must be supported by two SVC instructions to get and free temporary storage, at a cost of over 200 microseconds. In AFS, PSR's maintain the status and working storage for all processes at all levels. Although data formats beneath the SL level are CPU dependent structures and cannot therefore be identical to formats above that level, the same concepts and terminology are used to emphasize the relationship between similar problems on different levels of the system design.

2.1.3 Objects

In AFS, the object is a generalization from two sources: descriptor/value pairs and resource/process associations. Descriptors are maintained with data in data management systems, APL, EULER, and the dynamically varying parts of PL/I. The type field in a descriptor can be interpreted as the name of a machine for accessing the value part. Although the few bits that describe a floating point number don't exhibit many characteristics of a procedure, the generality of an access machine or procedure is valuable for complex arrays and structures and is essential for the intricate relationships in a large data base. The association of a process with every resource derives from Dijkstra's approach in T.H.E. Multiprogramming System and from Ole-Johann Dahl's approach to objects in SIMULA 67. Dijkstra associates a process with every resource in his system; the process is solely responsible for allocating that resource and acts as a central clearinghouse for all accesses to it. Chapter 2.5 shows that all objects in AFS have the properties of Dijkstra's resources and naturally fit into a general scheme of resource management. Alan Perlis suggested that simulation languages might provide a suitable basis for an operating systems language since they have the best developed concepts of event and process; the AFS concept of objects as processes is a generalization of the objects in the simulation language SIMULA 67.

Definition: An object is the basic entity in the system; it has an active part called an access machine and a passive part called an owned resource. Its active part responds to requests by other objects and may in turn generate requests of its own.

1) There is an input queue of cell names that specify buffers containing requests for the object.
2) The access machine is a process that waits in one of a set of states called ready states when it is ready to respond to input requests. When a cell name for a request appears on its input queue, it assumes ownership of the buffer containing the request, performs whatever action is appropriate, returns a buffer containing the answer, and returns to a ready state.
3) The owned resource is data that is accessed only by the object's access machine; for objects like clocks or printers, however, the resource may interact with events outside of the system.

Since this definition is general enough to accommodate source-sink I/O devices as well as objects as powerful as a Turing Machine, it can include any conceivable device within the standard accessing and allocating method. For a floating point number, the implementation could specify a fixed length bit string as the resource and a few bits to identify a hardware unit as the access machine. For I/O devices, the object internal to the system would be called a port whose resource would be a logical connection to the external device and whose access machine could be a hardware or microcoded control unit. Since the internal structure of an object is invisible to the caller, an object implemented in hardware or microcode on one system could be implemented in software on another: as in SIMULA 67, a software access machine is a procedure that defines a potentially infinite class of activations; an object corresponds to a process executing in one such activation; a ready state is a point in the procedure where the process waits for input; and the owned resource is a set of automatic variables used by the activation. Logically, all objects are processes; even a floating point variable is a process that is normally waiting, but must occasionally answer requests to deliver a value or to stow one away.

Definition: A primitive object is one that cannot be constructed from other objects in the system: the PSR, interpreter, and procedural description that make up its access machine are not objects formally defined in the logical architecture.

Somewhere underneath all the logical data structures, there must be primitive building blocks from which everything else can be constructed by software. Although the logical definitions of primitive objects are parallel to the constructions of other objects, their substructure is visible only to the engineers and diagnostic programmers.

Definition: A reducible object is one that can be constructed from other objects: the PSR, interpreter, and procedural description of its access machine are AFS objects that can be manipulated by SL.

Primitive objects are defined axiomatically in terms of their effects on other parts of the system. Sometimes, reducible objects are defined axiomatically, but most reducible objects are defined by an explicit construction in terms of primitive objects. All primitive objects are implementation defined; many reducible objects are implementation defined, and others can be user defined. For efficiency, reducible implementation-defined objects may be built out of hardware or microcode even though they can be constructed out of more primitive objects. Logically, however, all reducible objects have the same status whether they are implementation defined or user defined.

Definition: The primitive object nil has an access machine with only one state; for every request, nil returns a copy of itself. For operations on lists, nil has the properties of a zero element list.

In APL/360, the empty vectors are similar to nil, but they have additional type information: the empty character vector has a descriptor that indicates that it is of type character, and it expands into blanks; the empty numeric vector is of type numeric, and it expands into zeros; nil is of type any, and it expands into a list of undefined objects.

Definition: The primitive object undef has an access machine with only one internal state. For every request except destroy, undef raises an error exception.

Logical storage cells can never be empty. If nothing else has been put in them, they contain an undefined variable object. The object nil is a general neutral element; it responds without error exception to any request, although some functions such as + or - may themselves raise error exceptions when given a nil operand. The object undef is a general undefined element; it always raises error exceptions except when being copied or destroyed.

Primitive objects are so basic to the structure of the system that they cannot be constructed by software. Hardware devices may not be primitive in the same sense because a disk drive, for example, could be simulated by a software routine that duplicates its interface and uses the storage management system to perform the same functions; but there is no sequence of instructions that could create a new disk drive in the corner of the machine room and physically attach it to the computer. Therefore, certain objects must be built in from the beginning, and others may be attached as the system expands or removed when they fail. As long as the physical interface provides circuitry that matches voltage levels and makes the device look like a procedure, the logical interface can make room for it in the object base and can define synonyms and access machines that make it respond to any protocol expected of it.

Definition: A port is an object that communicates with the world outside the system: its access machine handles the interface, and its owned resource is a logical connection to a physical device.

Since ports are objects, they have the same interface as all other objects: they have a well defined status with respect to the accessibility graph, environment tree, and dependency graph; and they respond to requests in the same way as other objects. Therefore, it is always possible to replace a port with a software object that has the same interface; programmers can create logical printers, simulated 2314 disks, and even simulated networks of machines. If a graphic device has an unusual interface, the real port to the device can be replaced by a logical port that behaves like a printer, but that contains a program to massage control information passed with a request and send it to the graphic device in the appropriate format. To make network communication more transparent to the user, the system will provide identical interfaces for a virtual System/370 emulated inside the system and for a real System/370 at the far end of a telephone line.

If communications with a system were in the character format of typewriters and printers, the internal representation of an object would be of no concern to programmers and could remain totally invisible. But since data may be interchanged between systems, either conversationally or by removable storage media, there must be a standard representation of an object that can be recorded on an external medium and reconstructed on a different system. This standard representation is called an object image; every system is free to use its own internal forms, but they must all be directly mappable to the standard form for an object image.

Definition: An object image is an external representation of an object. The object image has two parts corresponding to the two parts of the object: a descriptor that specifies the access machine and a representation of the owned resource.

1) If the object is primitive, the descriptor indicates that it is primitive, and the representation is a

bit string specifying which object it is.

2) In general, the descriptor specifies the complete access machine by indicating the PSR (which may contain zero bits of information in some simple cases), the object image of the procedural description of the access machine, and the interpreter of the access machine.
3) If the owned resource contains storage cells holding other objects, the representation includes the object images of all those objects.
4) If the object is a synonym containing the cell name of some storage cell, the object image must contain a path name (see section 2.1.5) for reconstructing the cell name by indexing from some standard vertex of the accessibility graph.

The object image is an external form of the DAPOV (Descriptor And Pointer Or Value) discussed in the System Architecture Manual. Although a DAPOV on a small system may be different from a DAPOV on a large one, the object images will be the same for all. The object image may be considered as the DAPOV for an abstract implementation of AFS; it may turn out to be identical to the internal DAPOVs of one or more actual implementations, or it may be a compressed encoding of the internal DAPOVs.

Definition: The <u>object base</u> is the set of all objects in the system.

The term object base is more general than the term data base since it also includes the logical interfaces to hardware resources. Because of the generality, all hardware devices have descriptors and can have synonyms defined upon them. Whenever a device breaks down, its descriptor can be changed to point to another device or a software simulator that can replace it. All of the advantages of late binding therefore apply to devices as well as data: instead of doing a SYSGEN for every configuration, implementers can provide standard logical facilities, make descriptors for non-existent facilities point to substitutes, and keep the logical appearance constant as descriptors are changed one by one to reflect the current configuration.

The definition of object given above implies that all objects are serially reusable resources. Non-reusable objects can be implemented by making the access machine destroy the object after its first (or n'th) use; no requests can bypass this check since the object cannot be used except through its access machine. Re-entrant procedures and time-shared devices correspond to a potentially infinite class of serially reusable objects: by subdividing storage, a single re-entrant procedure can provide automatic variables for as many activations as requested; by subdividing time, a time-slicing routine can provide multiple logical devices that all perform the same function as a single physical device. The AFS view of objects as processes treats the problem of resource management as a problem of interprocess communication.

Definition: A request on an object is a triple (T;P;D), where T identifies the request type, P is information proper to that type, and D is the destination or object that is to receive the answer. Normally, the access machine of the object will execute the request and return a result to the object D. In some cases, the access machine will cause an event called an exception; see section 2.4.1 for a definition of exceptions and the ensuing events.

Definition: The dependency graph is a structure defined over the object base: If an object x has a request on its input queue that specifies an object y as its destination, then y is said to depend on x, and (y,x) is an edge of the dependency graph.

Later chapters will bring out implications of the dependency graph in resource management, process dispatching, and deadlock determination. Chains of subroutine calls form a subgraph of the dependency graph known as the activation tree: if x is an activation of a program that calls a subroutine y, then x is dependent on an activation of y until it returns.

2.1.4 Access Machines

Since every object has an access machine, it always has an active element available to perform necessary functions. A typical function is that of monitoring: During debug mode, the programmer may wish to monitor all accesses to a particular variable and then perform a specific action such as recording the access, calling some procedure, or waiting for instructions from the terminal. For sensitive data, all requests on an object may cause its access machine to check the identity of the caller and to notify a security officer of an access attempt by an unauthorized user. For proprietary software on lease, the access machine might destroy the object after a thousand uses. All these applications rely on the invisibility of an object's internal structure--when an ordinary variable is replaced by one that is being monitored, its normal interface remains unchanged.

Definition: An access machine has the following external interface:

1) It must have a set of ready states in which it waits for requests with arguments (T;P;D); after processing a request, it must return to a ready state.

2) The argument D specifies the destination for the response to the request.
3) The argument P specifies further information proper to the request type.
4) The argument T specifies one of the following request types:

Authorize: Request to obtain a synonym to the storage cell containing the object (see section 2.1.5).

Copy: Request to obtain a copy of the object. If the object may not be copied, the access machine raises an error exception. If the argument P is nil, then the entire object is copied; otherwise, P must specify some subpart to be copied.

Delete: Request to delete a storage cell of a collective object. The argument P must be the index of the cell to be deleted (see section 2.1.6). The object contained in the cell is not destroyed, but is returned as the response to the request.

Destroy: Request to destroy an object. If the object is non-destructible, its access machine raises an error exception. If it is a collective object, it makes destroy requests upon all of its elements before finally destroying itself.

Evaluate: Request upon a simple data object to deliver a value or upon a more complex object to generate a value. The argument P is nil for ordinary data objects, but must be a list for functions (see below).

Identify: Request to obtain a description of the access machine and structure of an object. If the argument P is nil, the response includes all identifying information; otherwise, P specifies the information requested (see below).

Insert: Request upon a collective object to insert a new storage cell into its owned resource (see section 2.1.6). P specifies the index to be mapped onto the new cell by select requests; if P is nil, the new cell has no index.

Select: Request upon a collective object to map P onto its storage cells: P must be a set (possibly nil) of elements in the index set of the object; the response is a set of cell names selected by those indices (see section 2.1.5 for further discussion of indexing).

<u>Start</u>: Request upon an activation of a function to begin interpretation of the procedural description associated with the function. The argument P is a list of objects to be bound to the formal parameters of the function.

<u>Stow</u>: Request to stow the value P in the owned resource of an object. The access machine will either perform data conversions to make P comply with its conventions or raise error exceptions if P cannot be converted properly or if the current value cannot be modified.

5) The access machine always reserves the right to tell lies about itself and its resource; this right is essential to data independence because it must always be possible to replace an object with another object that may be different in structure, but appears the same.

Definition: In order to specify requests, a primitive <u>request constant</u> is defined for each of the request types; the names of the request constants are formed by adding 's' to the corresponding request name: <u>authorizes</u>, <u>copies</u>, <u>deletes</u>, <u>destroys</u>, <u>evaluates</u>, <u>identifies</u>, <u>inserts</u>, <u>selects</u>, <u>starts</u>, and <u>stows</u>.

Simple data objects like floating point numbers and character strings very seldom make requests upon any other objects. The objects that normally make requests are functions: primitive functions make requests upon arguments passed to them in the initial evaluate request, and reducible functions are user defined programs whose very nature is to make requests upon data objects, upon primitive functions like sum, difference, product, or stow, and upon other user defined functions. The following definition of function presents the external interface of a function: it describes the action of a function as seen by the caller or by the rest of the system, but does not describe the internal processes and structures of the function. Chapter 2.2 describes the internal interface of user defined functions and the method of constructing them.

Definition: A <u>function</u> is an object that responds to evaluate requests by creating an <u>activation</u> and then making a start request upon the activation to compute the value to be returned.

1) If F is a reducible function, the activations are objects distinct from F that reside in storage cells with distinct cell names.
2) If F is a primitive function, its activations are not objects and cannot be manipulated by SL expressions. When the distinction is relevant, activations of primitive functions are called <u>quasi-activations</u>.
3) The argument P in the evaluate request upon a function F must be a list of the number of arguments required by F. If F takes no arguments, P must be nil, and F is called <u>niladic</u>. If F takes 1, 2, 3, 4, or n arguments, it is called <u>monadic</u>, <u>dyadic</u>, <u>triadic</u>, <u>tetradic</u>, or <u>n-adic</u> respectively.

The distinction between a function and its activation is essential: Since evaluation of a function may take a long time, it would be undesirable to keep the function tied up and unable to respond to any other request during the entire time of evaluation; many users on a system may want simultaneous access to a function such as a compiler, an editor, or a trigonometric function. Even more fundamental are recursive functions whose entire structure depends on the ability for one activation of a function to call another activation of the same function. On the other hand, it would also be undesirable to have many copies of the function, since the code can be shared. Therefore, a call upon a function causes it to spin off an activation which contains its own temporary storage, but which uses the same read-only code as all other activations of the function: an activation is a process whose PSR is unique to it, its procedural description is the read-only code which is shared, and its interpreter is the decoding mechanism that may be shared with other activations of the same function as well as with other functions written in SL. For consistency, primitive functions are considered as activations of hardware or microcoded procedural descriptions, but the activations are invisible to the programmer since they are defined at a level beneath his view.

Definition: The triadic function <u>request</u> makes requests upon objects and returns the value passed back by the access machine of the object; request(T;P;X) makes a request of type T with argument P upon object X.

The request function provides a general way of making requests upon objects. Certain requests, however, occur so frequently in specific contexts that special functions are provided to make those requests.

Definition: The monadic function <u>evaluate</u> makes an evaluate request upon its argument and returns the value that it delivers. For any object X, evaluate(X) is equivalent to request(evaluates;nil;X).

Definition: The dyadic function stow makes an evaluate request upon its left argument to obtain a value P. It then makes a stow request upon its right argument with P as the proper argument for stow. The value returned by the function is P. For any objects X and Y, stow(X;Y) is equivalent to request(stows;evaluate(X);Y).

The stow function is one of two types of assignment functions in AFS. The other assignment is the replace function discussed in section 2.1.6. The distinction between stow and replace is that the stow function makes a request upon its target to stow away the value, whereas the replace function makes a request upon its target to destroy itself and then replaces it with a totally new object. The special character symbol for stow is a single arrow, and for replace a double arrow; these symbols suggest the fact that the stow function normally changes only the owned resource of the target, but that the replace function changes both the access machine and the resource parts.

2.1.5 The Accessibility Graph

Previous sections defined objects and requests upon them; this section defines the possible paths for reaching one object from another. The structure that defines these paths is the accessiblity graph, which is a union of two subgraphs: the ownership tree that links collective objects with their elements and chains of synonyms that form links across the tree. Although neither the ownership tree nor the chains of synonyms allow circuits, the accessiblity graph can and must have circuits to support various types of list and ring structures. As later discussions show, the accessibility graph has the generality necessary for various structures, but it also has sufficient restrictions to prevent infinite looping in copying lists or resolving references.

Definition: A synonym is an object that behaves like a cell name; if x is an object and y is a synonym to x, then y has the following properties:

1) The resource of y contains a set called the rights to x which defines permissible requests on x.
2) The resource of y also contains either the cell name of the storage cell containing x or the cell name of an object from which x is accessible together with a path name from that object to x (see the definitions of path name and accessibility later in this section).
3) In response to requests, the access machine of y checks the request type; if the type is in the set

of rights to x, it passes the request to the object x; otherwise, it processes the request itself.

Cell names are not objects and cannot be stored and manipulated like objects. Synonyms are cell names with an access machine that can respond to requests and with an interface that gives them the same status as other objects. In a sense, synonyms are invisible objects because they don't answer requests themselves, but pass requests on to some other object. The rights define the requests that can get through to the object that the synonym points to. For some requests not in the set of rights, the synonym raises an error exception; for others, like destroy requests, it may make the response itself, i. e. by destroying itself instead of the object it points to.

Definition: The dyadic function <u>authorize</u> makes an evaluate request upon its left argument to return a list of request types and then makes an authorize request upon its right argument to obtain a synonym with the list of request types as the rights of the synonym. If X is an object and L is a list of request types, authorize(L;X) is equivalent to request(authorizes;evaluate(L);X).

Definition: The monadic function <u>syn</u> makes an authorize request upon its argument X and returns a value S that is a synonym to the storage cell containing X. The access rights of S do not include rights to make destroy and copy requests upon X; in response to such requests, S destroys or copies itself. The remaining rights in S are the ones granted by the access machine of X in response to the authorize request. If a request on S is not in the set of rights and is neither a destroy nor a copy request, the access machine of S raises an exception. If X is any object and L is a list of all request types except copies and destroys, then syn(X) is equivalent to authorize(L;X), which is equivalent to request(authorizes;evaluate(L);X).

A data base may sometimes have synonyms defined upon other synonyms; because of the implicit following of pointers in synonyms, there is danger of the system getting into an infinite loop if there is a circuit in the synonym graph. Since circuits of synonyms can only arise as a result of replace assignments, the replace function (defined in section 2.1.6) must have built-in checks to insure that the target of the assignment is not along a chain of synonyms extending from the source of the assignment. If the system is initially without circuits of synonyms, then such checks will guarantee that no circuits can arise.

Theorem: If a request of type T is made on an object through a chain of synonyms, then T must be in the intersection of the rights of all synonyms in the chain.

This theorem guarantees that safeguards placed on a synonym can never be weakened by other synonyms with a more permissive set of rights: the rights are a kind of filter that only permits certain types of requests to pass through; another filter can reduce the number of types that pass through, but it can never make any other filter more transparent.

Definition: A metonym is an object whose resource contains an enclosed synonym (see section 2.1.7). Since the synonym is enclosed, the automatic following of the pointer is inhibited, and a disclose operation must be made to obtain the synonym.

Although synonyms are adequate for list processing and data base applications, they can't be used for pointers in PL/I because they are almost indistinguishable from the objects they point to. Metonyms are objects that are recognizably different from the ones they point to and require a special operation to reach them. Suppose X is a floating point number, S is a synonym to X, and M is a metonym to X; then (S+1) and (X+1) would produce the same result, but (M+1) would raise an error exception. The disclose function must be used to produce a synonym from a metonym: the result of (X+1) could be obtained from M by the expression (disclose(M)+1).

A synonym is an object that represents or indirectly addresses one other object; the most complicated structures that can be built out of synonyms are linear chains. Trees represent the next level of complexity: a list whose elements may also be lists forms a tree; a vector in APL is a tree whose leaves are one level removed from the root; workspaces in APL are trees of heterogeneous objects such as functions, scalars, vectors, arrays, and groups; libraries, files, tables, and pools of devices all represent collections of objects, which may in turn include collections of other objects. In AFS, all these concepts are expressed by the general notion of a collective object that has other objects as elements; together, the collective objects form a tree, called the ownership tree, that includes everything in the object base.

Definition: A collective object is one whose owned resource is a set of storage cells for containing other objects; the collective object is said to own the storage cells in its resource.

Definition: If x is a collective object and y resides in a storage cell owned by x, then y is an element of x.

Definition: An elementary object is one that owns no storage cells: it is an element of a collective object, but it has no elements of its own.

Definition: The ownership relation between collective objects and storage cells has the following properties:

1) No object owns the storage cell it resides in.
2) The <u>system root</u> R is a unique object whose storage cell is not owned by any object.
3) No storage cell is owned by more than one object.
4) If S is a set of objects containing R and if S includes all objects that are elements of objects in S, then S includes all objects in the system.

Theorem: Every object except the system root is an element of one and only one collective object.

Theorem: The ownership relation defines a tree structure over the object base: the system root is the root of the tree, collective objects are at branching nodes, and elementary objects are at leaves of the tree. Call this tree the <u>ownership tree</u>.

The ownership tree provides a basic organization over the object base that resembles the typical tree structure of catalogs. The entire Library of Congress catalog is a tree structure: it is divided into 26 categories, which are subdivided into 26 categories, which are subdivided into 10 categories, which are subdivided into 10 categories, etc. The table of contents of every book is a tree structure; its index is a tree structure. The Yellow Pages of any telephone book form a tree structure. Unfortunately, tree structures are not adequate for all needs: almost every index, catalog, and phone book has cross references; and in complex cases, the number of basic entries may be far outnumbered by the cross references. AFS provides both types of referencing mechanisms: the ownership tree includes all objects; some of those objects may be synonyms that skip across the tree to objects along other branches. The union of the ownership tree and chains of synonyms forms the accessibility graph; to the programmer, a path that follows synonyms can be used exactly like a path that only indexes down the ownership tree.

Definition: The <u>index set</u> of an object x is a set of objects mapped onto the elements of x by select requests on the access machine of x. The index set of an elementary object is empty.

Definition: A <u>list</u> L is a collective object with the following properties:

1) If L has no elements, then L is identical to the object nil.
2) If L has N elements, then its index set is the set of integers 0, 1, ..., (N-1).

Lists are the most primitive collective objects: they are ordered sets of possibly heterogeneous objects. Although the usual formulations of set theory consider unordered sets to be more primitive than ordered sets, linear ordering appears to be fundamental for a theory of computation: Common storage devices (including the books in which set theory is formulated) force a linear ordering on all representations of sets. If a set is defined in terms of a predicate P, then one might maintain that "the set of all x such that P(x)" defines a set without defining a representation; in reply, we could answer that only recursive predicates are meaningful in a theory of computation and that hence the set must be recursively enumerable.

Definition: The monadic function ilist makes an identify request upon an object to obtain its index set: ilist(x) is a list whose elements are copies of objects in the index set of x.

If X is a vector in APL, (RHO X) is the length of X, and (IOTA RHO X) is equal to ilist(X). In AFS, however, the index set of a generalized collective object may not be computable from a single integer. In JOSS, for example, the programmer can define a vector with valid indices 1, 2, 5, and 9; although RHO of such a vector is undefined, the function ilist returns the list 1, 2, 5, 9. Similarly, AFS allows objects indexed by character strings; although IOTA and RHO of such objects are not defined, ilist would produce the list of valid character strings.

Definition: The dyadic function select takes an object x for its right operand and an element i of ilist(x) as its left operand; select(i;x) makes an evaluate request on i to obtain its current value and then makes a select request on x with the value of i as the argument. The value returned by select(i;x) is the cell name of the storage cell that the access machine of x associates with i.

The select function performs the ordinary operation of indexing by integers that is common in many languages as well as the more general indexing by character strings and other objects. The method for doing the indexing is left to the implementer: integer indexing will probably be done by hardware or microcode; indexing by character strings may be done with an associative memory, a microcoded search algorithm, or a hashing algorithm; indexing by more exotic objects would undoubtedly be done by a software access machine.

Definition: An object x is directly accessible from y if either x is an element of y, or x is an element of an object z which is directly accessible from y.

Definition: An object x is indirectly accessible from y if either y is a synonym for x, or there exists an object z that is a synonym for x and z is indirectly accessible from y.

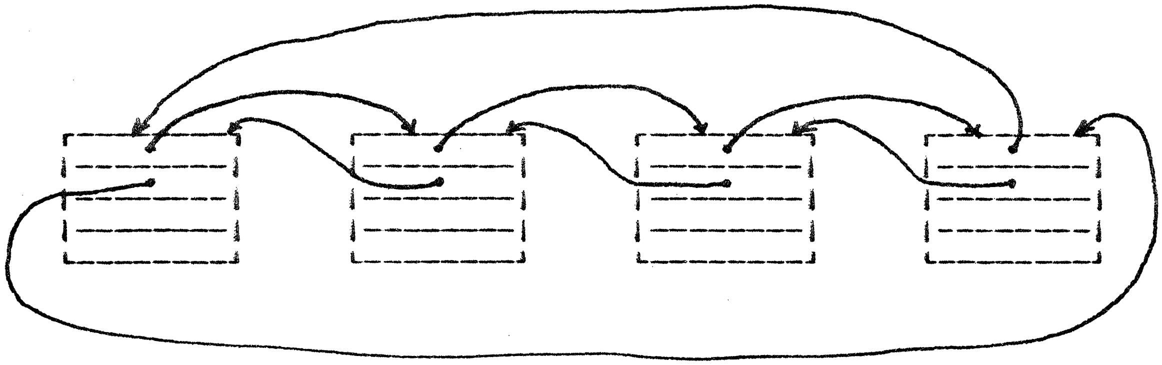
An object x is directly accessible from y if it is on a branch of the ownership tree that hangs down from y. Synonyms in AFS are analogous to indirect addresses in conventional systems: x is indirectly accessible from y if there is a chain of synonyms leading from y to x.

Definition: An object x is accessible from y if x is either directly accessible from y, indirectly accessible from y, or accessible from some object z which is accessible from y.

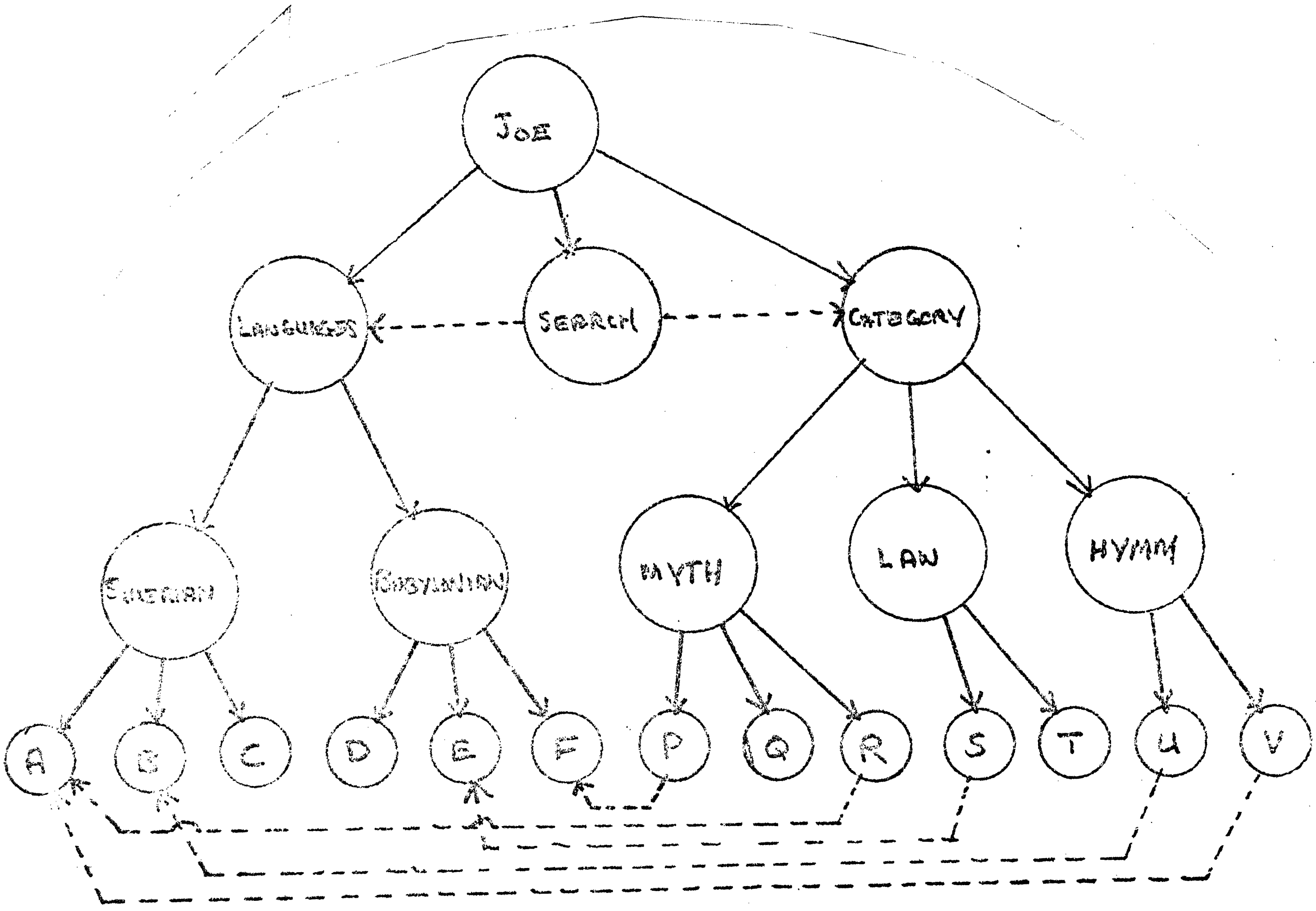
Direct accessibility is a relationship isomorphic to the ownership tree. Indirect accessibility corresponds to chains of synonyms and the objects they point to. The accessibility graph is a union of the graphs for direct and indirect accessibility. An object x is accessible from y if there is any path from y to x, some parts going down the tree and others going across chains of synonyms.

Definition: The accessibility graph is a union of the ownership tree and the chains of synonyms: (x,y) is an edge of the graph if either x is a synonym for y, or y is an element of x.

The accessibility graph will have circuits whenever there are ring structures or general cross references. Consider a structure of collective objects, each with four elements: the first element is a synonym that points forward to the next object, the second element is a synonym that points backward to the previous object, and the remaining two elements are data of some sort; then suppose that the objects are linked in a ring so that the last object is considered the predecessor of the first:



Consider the following example:



Suppose a philologist named Joe has a data base consisting of ancient Near Eastern texts. Each text could be a collective object whose elements are lines; each line would be a collective object whose elements are words. Although the division of a text into lines and words is straightforward, there are many ways of grouping texts into larger collections: one way is to put all Sumerian texts in one collective object, all Babylonian texts in another, and so on for Akkadian and Ugaritic; another grouping would put all texts on myths and legends from all the languages in one category, all hymns in another category, codes of law in a third, and business records in a fourth; many other bases for grouping are equally possible--chronological, geographical, etc. By means of synonyms, the accessibility graph can exhibit all the relations simultaneously. The diagram above shows part of Joe's data base: The node labeled JOE is a collective object with elements whose indices are 'LANGUAGE', 'CATEGORY', and 'SEARCH'. Under the collective object JOE.LANGUAGE are collective objects for each language Joe is working with; under each language are the texts written in that language. But if Joe is doing a comparative study of myths in Sumerian and Babylonian, he may find it easier to use JOE.CATEGORY.MYTH, which is a collective object containing synonyms to all the texts that relate myths in any of the languages. In this example, MYTH.P is a synonym for BABYLONIAN.F, LAW.S is a synonym for BABYLONIAN.E, and HYMN.U is a synonym for SUMERIAN.B. Therefore, the node B is directly accessible from the nodes SUMERIAN, LANGUAGE, and JOE, is indirectly accessible from the node U, and is accessible from the nodes HYMN and CATEGORY.

The relations expressed by synonyms do not have to be built into the structure from the beginning: when Joe adds a new text to his collection, he can insert it under the appropriate language; at any later time, he can define synonyms for it in any existing categories or even define new categories. Some texts may belong to several categories: SUMERIAN.A can be accessed via synonyms MYTH.R or HYMN.V. And in all cases, a running program does not need to know if it is accessing an object directly or via synonyms. For even greater flexibility, Joe can hire a computer science student to write some user-defined access machines to create special objects that have the same interface as ordinary collective objects, but that execute elaborate search procedures. For example, the object SEARCH may look exactly like an ordinary collective object; but internally, it has synonyms to LANGUAGE and CATEGORY and has an access machine that searches down those trees. If Joe wants to find the text of a myth about Gilgamesh, he could request SEARCH.MYTH.GILGAMESH.TEXT; then the access machine would look through all the texts accessible from the node MYTH to find one about Gilgamesh.

If x is a collective object, its index set must have enough indices to select every element of x. If y is an element of x and n is the index that selects y, then n is called a simple name for accessing y from x. If y happens to be a synonym for some other object z, then n is also a simple name for accessing z from x, because operations on y are automatically passed on to z. In the above example, 'A' is a simple name for accessing A from SUMERIAN, and 'V' is a simple name for accessing A from HYMN. If x is accessible from y by some complex path, there must be a list of simple names for each stage of the path. In the example, A is accessible from JOE by three different path names: LANGUAGE.SUMERIAN.A, CATEGORY.MYTH.R, and CATEGORY.HYMN.V. This example does not show any circuits in the accessibility graph; but when there are circuits, there are an infinite number of paths and hence path names for accessing some objects. (Note: this example used unique simple names for every node to make the discussion easier to follow; in general, elements of different collective objects may have the same simple names without causing ambiguity.)

Theorem: If x is an element of y or if x is indirectly accessible from some object z which is an element of y, then there exists an element n in the index set of y such that x=select(n;y). Call n a <u>simple name</u> for accessing x from y.

Definition: A <u>path</u> from an object y to an object x is a list of objects, the first of which is y and the last x; from an object u in the list to the next object v, there must be a simple name for accessing v from u. The list of simple names is called the <u>path name</u> from y to x.

Theorem: If x is accessible from y, then there exists a path name from y to x.

The path names provide a way of indexing down the ownership tree and skipping across the synonym chains. Before using a path name for accessing an object x, the system must find the object y from which x is accessible by that name. The environment tree described in chapter 2.3 defines a search procedure for finding the starting point from which the path name leads to the object. When a program is executing, the interpreter resolves names by searching up the environment tree until it finds a node that recognizes either the entire path name or at least the first one or more simple names in it; then the interpreter can make select requests with the remaining simple names until it reaches the object x.

2.1.6 <u>Manipulating Storage Cells</u>

Most operations on objects make requests on the access machine of the object. Certain operations performed on collective objects are intended to modify the storage cell containing an element. Although such operations are intended for manipulating storage cells, they can have side effects of destroying an object or moving it to a new storage cell.

Definition: Let x be a collective object, and let i be an object which is not in the set ilist(x), but which is acceptible to the access machine of x for addition to ilist(x). Then the dyadic function <u>insert</u> makes insert requests on a collective object to insert a new storage cell and index: if x already has i in its index set, insert(i;x) raises an error exception; otherwise, it has the side effect of adding a new storage cell to the resource of x, placing a copy of undef in the new cell, adding i to ilist(x), and causing the access machine of x to map i onto the new cell. The value returned by insert(i;x) is identical to the value of select(i;x).

Definition: The dyadic function delete makes a delete request on a collective object to remove a storage cell from its resource and to remove the index to that cell from its index set; delete(i;x) has the side effect of removing the storage cell containing x[i] from the resource of x and of removing i from ilist(x). The cell name of the old cell may not be used to identify any other cell ever to be created in the system. The value returned by delete(i;x) is the object in the storage cell before the cell was deleted.

Every function returns a value: the value of insert is useful as the target of an assignment for initializing the new object; the object returned by delete is useful to allow a cell to be deleted and its contents moved somewhere else in a single statement. If the expression delete(i;x) occurs alone in a statement, the cell containing x[i] is deleted by the function delete, and the value of x[i] is destroyed when execution moves on to the next statement.

Definition: The monadic function remove removes the contents of a storage cell without deleting the cell: remove(x) has the side effect of placing a copy of undef in the cell containing x; the value of remove(x) is the old value of x unchanged.

Definition: The dyadic function replace destroys the object contained in a storage cell and replaces it with a copy of another object: replace(x;y) makes a copy request on x to make a copy of itself, makes a destroy request on y to destroy itself, and places the copy of x in the storage cell formerly occupied by y. If y refuses to destroy itself, it remains unchanged, and an error exception occurs. If y is indirectly accessible from x, then an error exception occurs, and the target is not changed. The value of replace(x;y) is a copy of x.

Theorem: No circuits of synonyms can arise by execution of replace; any attempt to form such a circuit raises an error exception.

The replace function is a type of assignment used primarily for moving objects and placing initial values into new storage cells; its use in initialization is the basis for executable declaration statements. For normal assignments, the stow function makes a request upon the access machine of an object to perform the action and make necessary conversions.

When a storage cell is deleted, synonyms and metonyms containing its cell name are not destroyed; but any use of them raises an error exception. Since cell names are never reused, there is no danger that a new cell could be accessed via invalid synonyms. The four functions insert, delete, remove, and replace have important side effects on synonyms: suppose x is a collective object whose i'th element is y; then the statements

    {i delete x; i insert x}

leave a copy of undef whose direct accessibility is the same as y's, but whose storage cell has a new cell name that is different from the cell name in previous synonyms to y; the operations remove(y) and replace(undef;y) cause the undefined object to have the same accessibility as y, even for synonyms. If y is a collective object, any storage cells it owns are part of its resource and are moved with it; consequently, any synonyms to elements of y remain pointing to the same values even though a synonym to y itself may point to a copy of nil in the old storage cell.

Theorem: Let x be an object directly accessible from y[i] and indirectly accessible from z. After the operation delete(i;y) or remove(y) is executed, but before the object y[i] is destroyed, x will still be indirectly accessible from z.

These definitions can be implemented efficiently: removing an object involves moving a single descriptor from a space and replacing it with a descriptor for undef; the rules for synonyms to elements of a collective object follow immediately from the fact that the space containing the elements is not changed.

The function replace is defined as making a copy of its left argument; in a later section on program execution, the copy rules are modified to eliminate unnecessary copies. In particular, no copy is required when the object is the result of certain functions, which include remove and delete. Therefore, the following expression does not destroy the object A.B.C, but simply invalidates all its old names and renames it R.F.G:

    replace(delete('C';A.B);insert('G';R.F))

In infix form, the above expression may be written:

    'C' delete A.B => ('G' insert R.F)

A major advantage of the current design is that it has the flexibility of general list processing systems without the overhead of garbage collection or reference counts. Systems like LISP and SNOBOL keep data available as long as there is a reference to them; although such a property is often convenient, it seriously impairs efficiency: In LISP, for example, the standard method of garbage collection is to stop all computation, start at the topmost node of the system, and trace all data elements to see if any are unreferenceable; only after all nodes have been traced can the system throw any data away, and only then is there any space to resume execution. The method of reference counts replaces massive garbage collections at infrequent intervals by increments and decrements to a count field whenever synonyms are copied and erased. Although most objects have a count field of one, all objects must maintain such a field with provisions for letting such values grow arbitrarily large. On a storage hierarchy system, reference counts can become quite inefficient since a local action of copying a pointer can require the reference count of a distant object to be modified. The AFS approach is to destroy objects upon explicit request and to allow synonyms to destroyed objects to become invalidated; for ordinary FORTRAN and PL/I programs, this approach is the most efficient. If an application requires reference counts, they can always be added by causing the access machine of a collective object to keep counts of references to its elements, to issue special synonyms that report back whenever they are copied or erased, and to delete the elements when their reference counts go to zero; thus, the power is available when needed, but most objects don't have to pay for it.

Much of the library and cataloging facilities of current systems can be handled by the functions introduced so far: The DD cards in OS/360 are used to create synonyms between external and internal devices; for example, if SYSOUT is the name for a collective object whose elements are logical output devices and if A is the index for selecting logical printers, then the DD card

```
    //SYSPRINT DD SYSOUT=A
```

is equivalent to the expression

```
    syn SYSOUT.A => SYSPRINT
```

In OS/360, DD cards also specify physical characteristics of devices and request a type of allocation such as shared use or exclusive use for modification; in AFS, physical parameters are totally unnecessary, and the system provides much finer control over dynamic resource allocation (see chapter 2.5). In APL/360, system commands are outside of the language and cannot appear in functions; following are the AFS forms of some APL system commands:

```
   )LOAD 10 LOGIC       LIB10.LOGIC => Current
   )SAVE 10 LOGIC       Current => LIB10.LOGIC
   )CLEAR               ClearWS => Current
   )ERASE JOE SAM       delete JOE; delete SAM
   )COPY 10 LOGIC WFF   LIB10.LOGIC.WFF => ('WFF' insert Current)
   )LIB                 llist Mystuff -> SYSPRINT
```

The APL/360 system makes copies of workspaces because it has no way of sharing read-only objects and no way of defining synonyms to objects in other workspaces. Under AFS, a subsystem would be free to make copies or define synonyms as it chose.

2.1.7 Structure

The elements of a general collective object have only one thing in common: they reside in storage cells that all have the same owner. Special types of collective objects may impose more conditions either on the elements or on the admissible index set. Typical conditions restrict the index set to integers, pairs of integers, or character strings; other conditions restrict the elements to have the same access machines or representations. Although conditions restrict generality, they may improve efficiency and simplify enumeration of all elements. If all elements have the same access machine, the descriptor of the entire collective object need specify the access machine only once for all elements; such savings are especially obvious for bit vectors.

2.1.7.1 Lists

We have already defined a list in section 2.1.5. A list is a member of a special class of collective objects with particular index sets. The indexing capability of SL provides a mapping between a set called the index set and a set which comprises the objects in the storage cells of a collective object. The elements of the index set are called index objects. As the use of the word "set" implies, no structure is imputed to either set by the indexing mechanism itself. The most primitively structured collective object is the list. A list is a collective object whose index set is the set of integers less that N for some integer N. For example, a list of ten objects has for its index set (9, 3, 5, 1, 7, 4, 2, 8, 0, 6). A list in particular, and any indexed object in general, acquires its structure, if any, from the inherent structure of the indexing objects themselves. This structure must come from something other than indexing. In the case of the integers, initial segments of which are popular index sets, that structure is provided by the arithmetic functions which apply to them. These operations, ultimately definable in terms of the Peano postulates, are the basis for most index sets. Accordingly, we may clarify the definition of a list to say that a list is a collective object whose index set is an initial segment of the integers. We intend to imply that the ordering of the integers is a part of the definition of a list. For convenience, we introduce the following

Definition: A <u>primitive</u> <u>index</u> <u>set</u> is an initial segment of the non-negative integers.

Usually the term "index set" will be used in place of "primitive index set" when the context permits. Lists form the only special class of collective objects which is primitive to the system. There are no restrictions on the elements of a list. They may be scalars, closures, arbitrary collective objects, or other lists.

2.1.7.2 Structures

Since the elements of a collective object may themselves be collectives, it is possible to build tiered structures of arbitrary complexity and indexing depth. It is useful to have some definitions to talk about these objects.

Definition: A *structure* is a collective object some subset of whose owned objects is composed of collective objects together with all objects accessible by iterated indexing from the given object.

Definition: An *indexed structure* is one all of whose collective objects are indexable.

Definition: A *list structure* is a structure all of whose collective objects are lists.

Definition: The *shape* of a list is the number of elements in it.

Shape is a general term which also applies to arrays. When referring to lists or to vectors the term length will sometimes be used. One of the important characteristics of a structure is the number of tiers that have been defined. One can retrieve any one of the elements of a list with a single indexing operation. To specify an element of a list of lists, the indexing operation must be repeated.

Definition: The *depth* of a structure is the maximum number of times the indexing operation can be performed on the structure before reaching a scalar or an object already reached.

A scalar has depth zero. A simple list has depth one. One can simulate arrays at the programing level with list structures of depth two, ie., with lists of lists.

One may wish to define a depth two structure of lists whose elements are indexable. Unfortunately, the depths of these elements will be added to that of the structure and any attempt to determine the depth with ordinary functions will yield the wrong result. To handle such situations the encapsulate function is provided. It conceals any arbitrary structure within a scalar so that it can be placed in a structure without increasing its depth. The original structure can be recovered by using the uncover function.

For convenience in defining the locate function for lists we introduce a related type of indexed object. It is not primitive to SL.

Definition: A *pseudo-list* is an object whose index set consists of integers.

2.1.7.3 Arrays

For a number of reasons it is desirable to provide indexing with an arbitrary number of objects in a single level of indexing. The facility is provided by most high level languages in use today. It provides much of the flexibility of a list structure without incurring the inefficiency of multiple calls on the indexing operation to retrieve a single object. Furthermore, it is easier to rearrange objects within the structure since it is not necessary to shift them from one collective object to another.

This desirable facility is provided in SL as in other languages by arrays. In keeping with the spirit of SL, arrays are basically defined in a general way. They differ from other indexable objects in that a rigid framework has been provided in which their index objects reside. This framework is defined with the aid of a list structure called the base list or the base list structure of the array. No restrictions are placed on the index objects themselves, or on the elements of the array.

Arrays are not primitive to SL. It is thus an implementation decision whether the hardware will construct vectors of vectors to describe arrays or not.

Definition: the *base list* or *base list structure* of an array, A, is a list structure of uniform depth 2. The i-th sublist is called the *i-dimension index* set of A.

Definition: An *array*, A, of rank r is an object whose index set consists of lists of length r. The i-th element in each index object list is chosen from the i-dimension index set of A. The *rank* of A is the shape of its base list. An array of rank r is called an *r-array*. The *shape* of an array is a list of the sizes of its i-dimension index sets for all applicable i.

The monadic function ibase applied to an array produces its base list. The composite function shape ibase produces its rank. For any array, A, the following identity holds:

shape A ≡ shape map ibase A.

The elements of the index set of A are members of the augment outer product reduction of the base list of A. In standard terminology, this is the Cartesian product reduction.

Definition: A vector is a 1-array.

Definition: A matrix is a 2-array.

We shall refer to a vector with k elements and a list with k elements as a k-vector and a k-list, respectively. In particular the empty list and the empty vector are the 0-list and the 0-vector, respectively. Note that a scalar can be considered as an array of rank zero as well as a list structure of depth zero.

For a general array there are no restrictions on the elements of the sublists of the base list. In fact there is no restriction on the lengths of the sublists. For example, the integer generator can produce a potentially infinite list, which can be indexed with any integer. This list is the only entry in the base list for the corresponding infinite length vector.

An array may be indexed by characters, lists, other arrays, etc. If all the k-dimension index sets are finite, then the array is finite. If all the index sets comprise only integers, then the array is indexed by lists of integers. This is the most general type of array usually handled. A particularly important subclass of finite, integer indexed arrays is the following:

Definition: A primitive array is one in which the index set in each dimension is a primitive index set.

In order to provide the kind of flexible restructuring through indexing which is available in, for example, APL we permit the substitution of certain arrays within the list which constitutes an indexing object. These substitutions define an infinite set of structures which the select function will accept for indexing arrays.

Definition: The basis for the index set of an array is the Cartesian product reduction of the base list of the array.

This is what is usually called the index set of the array. The function ilist on an array produces the basis of the index set.

Definition: The complete index set of an array is derived from the basis for the index set. For any position or set of consecutive positions in an index list may be substituted any array. The elements of the array must be lists of the same length as the partial list the array replaces. The returned object will be an array. The base list of the returned array is the catenation of the base lists of the participating arrays.

Since the phrase "basis for the index set" is usually shortened to "index set", the word "complete" must be expressed when imported to prevent confusion. The base list for an array defines its structure in complete detail even for arbitrarily indexed arrays. The information required to determine the index structure for a primitive array is much less. It is simply the length of the index set in each dimension. The shape function applied to an array will return this information in the form of a list. The function igenerator applied to a scalar returns the index list for a list of corresponding length. The function igenerator applied to the shape of a primitive array generates the index base for that array by function distribution.

The relationships between the various types of arrays and lists can be described by the results of applying the various structure determining functions to them. The information is summarized in the following table.

	list	scalar	0-vector	vector	r-array
ilist	list	0-list of lists	0-list of 1-lists	list of 1-lists	list of r-lists
index object	scalar	0-list	1-list	1-list	r-list
ibase	list	0-list of lists	1-list of lists	1-list of lists	r-list of lists
shape	scalar	0-list	1-list	1-list	r-list
shape of shape	0-list	0	1	1	r

For convenience in defining the locate function for arrays we make the following definition.

Definition: The *index object array* of an array A is the primitively indexed array with the same shape as A whose elements are the respective index objects of A.

Note that the relationship between a pseudo-list and its list of indices is analogous to that between an array and its index object array.

Chapter 2.2

PROGRAM STRUCTURE AND INTERPRETATION

The access machine of every object is a process. This process is derived from a procedural description by adding some local storage and causing an interpreter to begin executing this description. This chapter presents the form and execution of programs, that is, procedural descriptions written in SL. The chapter begins with an overview of the concepts which are important to interpretation and program structure. After that the form of a program is given. This is given as a data structure in SL. Then, the constraints that this form implies on the external syntax are given.

The remainder of the chapter is devoted to the interpretation of the text of the program. The interpretation of an expression is developed in detail. The protocols for calling other functions are presented in a form suitable for using functions written in a foreign (non SL) architecture. Then, the interpretation of functions with multiple expressions (i.e., statements) are described. Finally, various operators for varying the order of interpretation are discussed.

2.2.1 Key Concepts

This section introduces at an overview level the key concepts which are required to represent and execute an SL program.

2.2.1.1 The Form of the Language

In SL there are two forms in which programming may be done: an external syntactic form and a machine-oriented data structure form. The reason for this dichotomy is that there is no single form which is adequate for both human beings and machines. Humans expect clarity of expression and readability. They often find it easier to manipulate programs in textual units such as strings. On the other hand, machines work better with fairly rigid data structures. Then, the machine can use the fixed information to provide a more compact program representation and to optimize execution.

There is, however, another reason for having two representations for a program. This is exemplified by LISP. In LISP, it is possible to input and display acyclic list structures in an external syntactic form. However, it is also possible to write LISP programs to build and modify list structures using the LISP functions. Since a program in LISP is a list structure, this has the important consequence that it is easy to write programs that write or modify other programs.

The flexibility to construct programs as data structures is very important. It makes it possible to write compilers with greater ease. It also helps when program modification is required as in a sort generator. Finally, it allows programs to respond to requests by constructing another program to do the work. This type of behavior will become more popular as data query systems grow.

The external syntactic form is therefore designed to give the best possible human interface to the system. It provides extensions of the strict machine form to better support naive users. The external form will be translated into the machine form by an incremental, statement-by-statement translator whose existence can be ignored by most users of the external form. The machine form is defined in terms of data structures which can be constructed and manipulated in SL. It is designed to maintain the information needed to do faithful interpretation and to be convenient to manipulate.

Programs are expressed in groups of statements. Each statement is a string of symbols. A symbol is one or more characters which is clearly delimited. The symbol strings represent infix expression in the external form. In the machine form, the symbol strings represent the Polish prefix form of an expression. In either case, a statement is any legal SL expression. Each group of statements is represented in the machine form by a module which contains a list of the statements.

2.2.1.2 The Execution of the Language

Program text, even a module, is really only a representation of an algorithm description. It is only by executing the text or module that the intent of the algorithm is carried out. In SL there are a number of steps in the process of executing an algoritnm or procedural description. These steps form a phased history of the life of a module.

Definition: A module goes through a number of <u>phases</u> as it entered, prepared for execution, executed and finally discarded. These phases in order are:

Translate: Converting the program into a module.

Load: Establishing a new copy of the module with its associated load oriented (static) storage in the user's current context.

Activate: Creating an object which contains information associating parameter symbols with arguments and contains a new generation of the activation-oriented (automatic) storage.

Execute: Interpreting the body of the text of the module.

Deactivate: Possibly releasing the generation of automatic storage if it can no longer be accessed.

Unload: Releasing all storage associate directly with the loaded module being unloaded.

At each phase the form of the module changes. Up to the execute phase more and more information is added. After that, information is discarded. It is quite possible to use a phase of a module as the basis for several different instances of the next phase. For example, only a single load is required for many different activations of a module. Similarly, a single copy of the text of the module can be shared by many loads.

The load and unload phases are developed in detail in Chapter 2.3. In this chapter the emphasis will be on the transition from the load phase to the activate phase, and onto the execute phase and, finally, through the deactivate phase.

Definition: The transition from the load phase into the activate phase and onto the execute phase is called activating a function.

The process of function activation includes building up a new object from the loaded module by adding some automatic storage, passing arguments and causing the independent execution of the new object. It begins when an evaluate request is made on a loaded module. When the activate phase is entered, the interpretation of the text is begun.

Definition: The interpretation of a function is performed by scanning the text of the function module and making requests on the objects associated with the symbols that are encountered.

In the terms of formal logic, meaning is given to a purely syntactic form by associating objects from a universe with each of the symbols in the form. Then, the form can be evaluated using the rules of combination for the objects associated with the symbols in the form. In SL, the symbols are associated with storage cells which hold the objects that give the symbols meaning.

Definition: Symbol resolution is the mechanism which associates with each symbol the cell name of a storage cell in the object base.

As will be seen in Chapter 2.3, it is possible to separate symbol resolution into a number of stages. Each stage inserts information which is fixed with respect to all succeeding stages. This factoring of symbol resolution can greatly improve the performance of the machine since potentially repetitive work is done only once.

As the interpreter moves through the module text, it will need to keep some status information in the PSR for the activation which is being interpreted. One of the major pieces of information that must be saved is the status of evaluating the operands of an operator. Because expressions can be nested to an arbitrary depth, an undetermined number of operators may be in the process of operand evaluation simultaneously. Therefore, a special part of the PSR is distinguished to hold operator evaluation information.

Definition: An evaluand is a collective object which holds the information about the status of evaluation for one operator and its operands. The evaluand is part of the PSR.

An additional portion of the PSR is used to retain which statement is currently being interpreted. This corresponds to the instruction counter on classic machines.

Definition: The statement index is a portion of the PSR which holds the index of the statement currently being interpreted. If there is no such statement, then the value of the statement index is undef.

When the execution of the module text is completed, the activation is destroyed. The storage associated with that activation may or may not be destroyed depending on whether or not references to symbols associated with that storage are still legal. In PL/I such references are not legal so the storage may be released. However, in LISP references are legal and the storage may outlive the activation.

2.2.2 Internal Program Representation

The basic unit of program construction is a stretch of text where each symbol in that text has only one meaning. Internally, this is represented by a module.

Definition: A module is a primitive collective object consisting of two components: the module text and the dictionary. There is one entry in the dictionary for each symbol which occurs in the module text. This dictionary entry also holds the information for symbol resolution.

The fact that each symbol has only one association within the module makes a module suitable for the minimum unit of translation into internal form. The symbols can be factored into a separate dictionary, and their occurrences in the text can be replaced by offsets into that table. Then, symbols can be resolved by associating storage cells with the entries in the dictionary. This encoding reduces the size of the program text and the complexity of decoding it. Once program text is encoded, however, it is meaningless without the associated dictionary. Therefore, whenever program text with symbol associations can be selected as a separate unit, the corresponding dictionary must be available to define the meaning of offsets in the encoded text.

Definition: The dictionary is composed of three component structures: the symbol table, the linkage table, and the attribute table. There is a 1-1 correspondence between the entries of each table. The symbol table has the character representation of the symbol. The corresponding entry in the linkage table has the association (if any) for the symbol, and the attribute table entry has information about the symbol.

The dictionary is logically indexed by the symbols. Hence, the symbol table acts as the index list of the dictionary. However, within the text of the module, the symbols are represented by symbol references. Symbol references are logical indices into the three parallel tables. Therefore, the symbol references are alternate indices for the dictionary. The symbol references correspond to the symbolic names used in the system architecture manual.

Definition: A symbol reference is a logical index into the dictionary. When used, it selects the component corresponding to the symbol it represents. It is valid only within the module in which it was created.

Definition: The tetradic function insert symbol causes a new symbol to be added to the symbol table of the designated dictionary and the corresponding entries in the linkage table and the attribute table to be filled in. The result of insert symbol (I;L;A;X) is the symbol reference of a new entry in the dictionary X with the value of I as the symbol entry, the value of L as the linkage entry, and the value of A as the attribute entry.

The insert_symbol function is much like the normal insert function. The major difference is that two additional arguments, the linkage information and the attribute information, are provided. Also, the result is not the cell name of the added cell, but is the symbol reference which will select the new entry. Using a special operator to add to the dictionary makes it possible to discipline the use of the dictionary.

The attribute component is arbitrary and may be used to store information required by the language being translated. Hence, it may serve as a compile-time dictionary and as a place to hold initializing information at run time. The form of the linkage information will be discussed in Chapter 2.3. Basically, it consists of an indication as to whether the symbol is defined within the module or that it is defined in some other module. In the latter case, it contains the information on how to find the defining module. It also contains information on the storage class, since this affects linkage.

Whenever the same symbol occurs in two different modules, the occurrences may or may not be associated with the same storage cell. One possible approach is to define the symbol-storage cell association to be the same for all the modules in any collection of modules. Then, a different symbol would be needed for every distinct storage cell to be referenced in the collection. This is annoying for one user and almost impossible to handle when two or more users are combining their programs. Therefore, it must be possible to define a context in which a particular symbol-storage cell association is to hold. It is then possible to have more than one association in a set of modules.

Definition: A symbol is _defined_ in a module if the storage cell associated with the symbol inside the module is different from the storage cell associated with the symbol in the surrounding context of the module. The linkage information corresponding to the symbol in the dictionary indicates when the symbol is defined.

Definition: A _local symbol_ is a symbol which is defined within the module in which it occurs.

The local symbols of SL correspond to the local symbols of APL and the declared internal symbols of PL/1. They are also known as bound symbols in mathematics. Local symbols are important because storage cells are allocated for local symbols when the module is used. All other symbols are just references to storage cells allocated outside the module.

Definition: A symbol which is not local to a module is a _free symbol_ or a _parameter symbol_. The linkage information for such symbols indicates how to find the definition of the symbol and the associated storage cell.

The symbol-object association for free symbols is derived from the surrounding context of the module. The method for determining this association and the surrounding context will be discussed in Chapter 2.3. The free symbols correspond to those symbols in PL/I procedures and APL functions which are not declared within the procedure or function. The resolution of parameter symbols is discussed in section 2.2.4.

Definition: The text component of a module is a list of statements. Each statement is a list of symbol references.

The list of symbol references represents an expression (see the next section) in Polish prefix form. Treating statements as a list makes it possible to select the statements by a simple integer index. This makes editing the text much simpler. It also provides a clean definition of local labels.

Definition: A local label prototype is a symbol defined in a module and associated with the index of one of the statements in the text.

The prototype is made into a local label by adding to the prototype information which indicates which generations of local storage were active when the label was created.

2.2.3 Syntactic Form of Program Text

This section sets down the constraints on the external syntax that are conceptually required. It is not to be interpreted as a specification of the syntax, but only of the form of the syntax. Many concrete syntaxes or external representations are compatible with these properties; one such representation is the external form presented in Chapter 4.3. The external syntax is designed to be suitable for human use. It is intended that an incremental translator will build the program representations described in the previous section. Where it is relevant, the machine form will be discussed with the syntactic constraints.

2.2.3.1 Symbol Lists

Definition: Program text is a string of symbols.

This is an important difference between AFS and existing systems. Unlike the bit encodings of System/370, bit encodings in AFS and physical addresses of hardware devices are known only to the implementation. Bit encodings are never displayed to programmers in hex dumps and can never be modified by them; instead, all communication is in the form of character strings defined in the logical architecture.

Definition: A symbol consists of one or more characters treated as a single unit; the implementation must include appropriate delimiters or character counts to indicate the extent of a symbol.

Intuitively, symbols correspond to the tokens of PL/I and APL. They include denotations for constant, single and multiple character operators, and identifiers. There will be rules for determining the extent of symbols so that the last character of a symbol is obvious to a symbol parser (lexical analyzer).

There are two classes of symbols: operator symbols that represent operators requiring operands to be evaluated, and elementary symbols that represent objects that do not require operands to be evaluated. These classes are distinguished so that it is possible to syntactically preprocess the text: Operators must be syntactically distinguished from elementary symbols if syntax checking or parsing is to be done. In APL, variables are syntactically indistinguishable from user-defined operators; therefore, the only way to tell if a symbol is an operator or a variable is to find out what the symbol represents at execute time.

Definition: An elementary symbol is a symbol without any syntactically-associated operands. Two subclasses are distinguished: The first subclass, literal symbols, consists of symbols whose form identifies the objects they represent; the second subclass, representative symbols, consists of all the remaining elementary symbols.

These two subclasses correspond to the classes of constants and identifiers respectively. Examples of literal symbols are 'XYZ, 3.4, 2+4I. Examples of representative symbols are X, VARIABLE_ONE. The rules for resolving representative symbols are given in Chapter 2.3. However, literal symbols can be resolved at translate time to a special constant table which is an extension of the dictionary. Each literal can be replaced by a special internal symbol reference to this table.

Definition: An operator symbol is a symbol which has operands that are syntactically associated.

There are at least two ways to distinguish operator and elementary symbols. One way is to enclose the arguments of an operator symbol in parentheses as is done with PL//I function references. The second way is to put a description of the number and location of the operands before or after the operator in the program text.

These definitions cause niladic functions to be considered to be elementary symbols because they have no operands. However, there is no need to parse variables and niladic functions in a different way.

Definition: A *simple expression* is either a single elementary symbol or an operator symbol, together with the correct number of operands. Each *operand* is a simple expression.

This defines expressions recursively beginning with elementary expressions such as constants, variables, and niladic functions. These may be used as operands for operator symbols to build one level expressions. Then, two level expressions may be built from these one level expressions or simple expressions. This allows arbitrarily deep nesting of operators.

Definition: When an expression has the form of an operator together with a set of operands, the operator is called to *top operator*.

This definition reflects the fact that the syntax of an expression is really a linear representation of a tree. The non-terminal nodes of this tree are the operators, the terminal nodes are the elementary symbols. The branches in the tree correspond to the operands of the operator to which they are attached.

2.2.3.2 Special Operators

There are cases where it is necessary to use an operator symbol as an operand of another operator. One example of such a use occurs with the inner product operator in APL. It takes two operators (e.g., + and *) and two arrays and produces a result. This is written A+.*B where + and * are not operators with operands, but are elementary symbols used with the dot operator. Because of the syntactic rules given above, it must be possible to syntactically distinguish the two different uses of + and *.

Definition: There is a prefix symbol *quote* which syntactically converts the occurrence of the symbol following it into an elementary occurrence.

Hence, the APL inner product would be written in the strict syntax as inner (quote plus;quote xpn;A;B). In the extended syntax, a simpler expression similar to the APL form might be adopted, but such a form would be a syntactic macro whose expansion in strict syntax would have to use elem. Note that the APL form requires a precedence relation in conjunction with the dot operator to override the normal use of + and *.

If quote is used with elementary symbols, it has no effect since it only indicates how to parse the program and not how the access to a symbol is to be interpreted. This is covered below in discussing the evaluation of operands of operators.

Another problem occurs when languages like APL are translated to SL. It is necessary to represent the program text for APL in a partially parsed form. In those cases where it is impossible to tell syntactically how to parse the symbol string, the delayed parse operator is used.

Definition: The dyadic delayed parse function takes two operands. There are three legal combinations of operands:

	first operand	second operand
1)	niladic object	monadic function
2)	niladic object	dyadic function
3)	partial dyadic fcn	niladic object

All other combinations are illegal. The result of delayed_parse in each case is:

1) a niladic object which is the result of applying the monadic function to the niladic object.
2) a partial dyadic function which has as its first argument the niladic object. Before the dyadic function can be evaluated, the second argument must be obtained.
3) a niladic object which is the result of evaluating the partial dyadic function with the niladic object as the second argument.

This allows the APL text to be represented and the parsing to be completed at execute time. The APL expression A B C D E would become

```
dpar(dpar(dpar(dpar(E;D);C);B);A)
```

where dpar stands for delayed_parse.

2.2.3.3 Grouping Expressions

The simple expression is too restrictive a format for all programming. It is necessary to group expression which are executed only for their side effects and not for the final result. These correspond to sets of lines in APL or a set of statements in PL/I.

Definition: A group is a segment of program text beginning with an initial marker (e.g., left brace), continuing with expressions separated by a marker (e.g., semicolon), called the statement marker, and ending with a final marker (e.g., right brace).

Definition: The initial and final markers are called group markers.

A group represents a "module constant". That is, the translation of a group yields a module. Hence, a group is very similar to a literal symbol. This fact makes it reasonable to allow groups to occur where an elementary symbol can occur. This leads to syntactically embedding groups within groups. To accommodate this possibility, a group was defined over expressions rather than simple expressions.

Definition: An <u>expression</u> is either a simple expression or a group or an operator symbol, together with the correct number of operands. Each operand must be an expression.

Definition: An expression which is one of the components of a group is called a <u>statement</u>.

The syntactic rules given above allow a group to be syntactically embedded within an expression and, hence, within another group. This is purely a syntactic convenience. Each group is translated to a separate module which does not contain the embedded groups. Instead, it contains internally-defined symbols which are associated with the modules for the embedded groups. This process is analogous to the handling of literals. The procedure for resolving and connecting the separate modules is discussed in Chapter 2.3.

The following definitions are inserted to clarify which symbols are in the dictionary of a particular module.

Definition: A symbol which is part of the text enclosed by the group markers is <u>contained</u> in the group defined by the markers.

Definition: A symbol which is contained in a group A but is not only contained in groups textually contained in A is <u>directly contained</u> in A.

Only those symbols which are directly contained in a group are put in the dictionary for the module generated by that group.

A typical group is the set of statements which exchange the contents of two variables, A and B. This requires a temporary location and three statements;

```
{stow(A;TEMP);stow(B;A);stow(TEMP;B)}
```

2.2.3.4 Declarations

If all programming were done in the machine form of SL, then declarations would unnecessary. All declarations could be done by executing the insert_symbol function on the appropriate module dictionary. However, it is necessary to have a way of indicating in the external syntactic form that certain symbols are being defined and that others are free or parameter symbols.

Therefore, the external syntax must have declarations. A declaration will be treated as a notation for one or more uses of insert_symbol on the dictionary of the module which results from translating the group in which the declaration occurs. See Chapter 4.3 for the syntax of declarations.

2.2.3.5 Functions

One of the most powerful aspects of mathematical notation is the ability to abstract upon an existing expression to define a new function. An n-adic function can be defined from an expression by designating n of the symbols occurring in that expression as being parameter symbols. When the new function is applied to a set of n values, these values are associated with the corresponding parameter symbols in the expression. The result of the function is the result of evaluating the expression in the context of these parameter symbol associations.

It is important to note that the module produced by translating the group is a niladic function. An evaluate request is required to cause an activation of the module to be created. The result of such an activation is the result of evaluating the text of the module. Therefore, the group brackets act to delay the evaluation of the text in the group until an evaluate request is made. Hence, the group represents the text, not the evaluation of the text. It is, in fact, a module or niladic function constant.

Since a module already represents a function, it is relatively easy to create an n-adic function from it. All that is required is to modify the linkage information of the symbols to be treated as parameter symbols. This can be done with insert_symbol. However, it is convenient to have a syntactic form which clearly shows the functional abstraction.

Definition: The dyadic operator lambda takes as its right operand a module and as its left operand an ordered list of symbols which are not local to that module. The result of the operation is a parameterized module. The symbols given in the left operand are marked as parameter symbols. The parameter symbols will be resolved in the order in which they occur in the left operand of lambda.

The parameter symbols must be resolved when a function is activated (see section 2.2.4) since the arguments may differ from use to use. However, the remaining symbols may have been previously resolved. For the rest of this chapter, it is assumed that all symbols other than the parameter symbols have already been resolved by an unspecified algorithm. This restriction is removed in Chapter 2.3.

A good example of the use of the lambda operator is to define a function which doubles its argument. Let X be a local variable. The expression 2•X yields a value which is twice the value of X. This can be made into a function by making X a parameter symbol. The expression

lambda (X; {product(2;X)})

yields a function which gives twice its argument whenever it is applied. It is assumed that the symbol 'product' is externally defined to be the multiply operator. The literal symbol '2' represents the object 2. Although the only local symbol in this function is a parameter symbol, it is possible to have other local symbols as well as free symbols in a function module.

2.2.4 Activating a Function

A function is used by making an evaluate request on it. The evaluate request contains the arguments to be used by the function. The function may or may not do the work to compute the result itself. If the function is to be reentrant, it creates a new object with new local storage to compute the result. This allows the function to process other requests "simultaneously". If the function does not create a new object to compute the result, then the function automatically becomes serially reusable because of the request queue in the storage cell it resides in. See Chapter 5.4 for further details on function activation.

Definition: An evaluate request on a SL function performs the following actions:

1) A new activation of the function being called is created by the object receiving the evaluate request.
2) The argument list is passed to this new object via a start request. The start request causes the interpreter for the new activation to begin.
3) The interpreter first associates the parameter symbols in the new activation with the storage cells of the arguments in the argument list.
4) The text of the function is then interpreted.

Definition: Each evaluate request creates an activation of the function which is being interpreted.

The matching of arguments to parameter symbols is left to the interpreter in the access machine as is the interpretation of the body of the operator object. This allows flexibility in the definition of the evaluation of the operator. The operator may be a SL function, as defined above. However, it may also be a primitive operator or a procedure in some other programming language. For primitive operators, the system will access the argument list and the result of the operation is defined axiomatically. In the case of procedures written in other languages, the access machine contains the interpreter for those procedures.

2.2.5 Expression Interpretation

Consider a single function being applied to a set of numeric values. For example, the expression 2+3 indicates the application of the sum function to the operands, 2 and 3. The evaluation of this function is relatively simple. The values of its arguments are already computed. Therefore, to evaluate the function, it suffices to associate the arguments, 2 and 3, with the appropriate parameter symbols in the code for the sum function and to begin interpreting that code.

This small example already shows several aspects of the interpretation process. If we assume that the sum function is not primitive, for example, it might be defined in terms of operations using the Peano axioms for arithmetic. Then, we see that evaluating an operator may cause additional expressions to be interpreted. There are three steps in the interpretation of the sum operator in the above example. First, the two operands are collected into a list of operands. Then, the function representing the operator is activated. The activation of the function causes the parameter symbols to be associated with the storage cells holding the operands. Finally, the expression which forms the body of the function for sum is interpreted. The result of the operation is the value computed by the interpretation of the body.

In the example above, the operands were elementary symbols. The syntax allows the operands to be expressions. In this case, the arguments are not the expressions themselves but are the values represented by those expressions. That is, the function is applied to an argument list which is constructed from the results of evaluating the expressions. Thi s complicates the interpretation of a function. The argument list cannot be constructed until each of the expressions forming the set of operands is evaluated. For example, in the expressicn, sum(2;times(3;5)), the subexpression times (3;5) must be evaluated before the sum function can be evaluated.

Definition: The occurrence of a literal symbol in the program text is replaced by an association to a read only storage cell which holds a copy of the object the literal represents. <u>Evaluation</u> of a literal symbol yields the cell name for that cell.

Literal symbols are treated as expressions to be evaluated at "compile time". This is in fact what is done in most programming languages. A good example of this is the handling of vector constants in APL.

Definition: The evaluation of an elementary symbol results in the cell name of the storage cell associated with that symbol.

Since symbols are always associated with storage cells, this is the most general result which could be computed. It is clear that the contents of a storage cell can be obtained if the internal identifier for that cell is known. However, it is not possible to determine the cell name of the cell which held an object when only the object itself is known.

One problem with having the cell name be the result of evaluating an elementary symbol is that it is often the contents of the cell or even the result of evaluating the contents of the cell which is desired. Therefore, operators are provided in SL to force the further evaluation of the contents of a cell by making calls on the object stored in the cell.

Definition: The interpretation of an expression which consists solely of an elementary symbol is the evaluation of that symbol.

An operator symbol cannot be evaluated without its arguments. Hence, it is necessary to simultaneously define the interpretation of an expression and an operator symbol. The interpretation is begun at the top of the tree representing the expression. The main reason for this is that it allows a context to be provided for the evaluation of the operands. This context can be used to perform dragalong, as defined by P. Abrahms. It can also be used for the type of optimization used in the Boulder PL/I compiler.

Definition: The interpretation of an expression which consists of an operator, together with a set of operands, is done in stages.

1) The object in the storage cell associated with the operator symbol is accessed with an identify call to obtain its sttributes. If it is a function or procedure and the required number of arguments agrees with the number of operands given, then stage 2 is begun. Otherwise, an error exception is raised.
2) Each expression in the operand set is interpreted. The results are stored in a set of buffer cells associated with the evaluation of the operator. When all the operands have been evaluated, the argument list, a vector of storage cells containing copies of the results, is constructed and stage 3 is

begun.

3) The operator is evaluated using the argument list. The result of the expression is the result of the interpretation of the operator.

The order of evaluation is defined to be left to right to be consistent with the actual implementations of most programming languages and to make it possible tp predict the order in which side effects will occur. It is not felt that any freedom for parallel evaluation can be effectively exploited at this level. The advantage of predictability seems to outweigh any improvement due to parallelism.

The arguments are passed by reference. This is required to implement such primitive functions as replace. Replace must have access to the storage cell to be modified if it is to operate correctly. This is only possible if the cell name is the argument to the function. Call by value can be implemented by having the called function copy the contents of the cells referenced in the argument list. Call by name is slightly more difficult, but can be implemented by passing references to niladic functions. Then, these functions would be evaluated at each use of the call by name parameter symbol within the text of the called function.

Definition: The evaluation of an operator symbol and an argument list is performed by making an evaluate request on the object contained in the storage cell associated with the operator symbol. The argument list is passed as the argument of the evaluate request.

The definition of interpretation shows that beginning interpretation of an operator causes other operators to also be interpreted. In particular, each operand of an operator will be interpreted. When the operator has a function body, then that expression is also interpreted. Thus, many operators may be in some stage of the evaluation process.

Definition: The state of evaluation of each operator symbol being evaluated is kept in a collective object called an evaluand. This collective object keeps track of the current action being performed and the partial results which have been completed.

The evaluand holds the results of evaluating the operands prior to constructing the argument list. An evaluand serves much the same function as the Mark Stack Control Word used in the Burroughs architecture. However, it controls the building of the argument lis, as well as the call on the operator. It is so named because it represents a part of the expression being evaluated. It can be used to provide status information for debugging requests.

Because evaluation of expressions is strictly left to right, the evaluands for the set of operator symbols which have not yet completed the evaluation of their operands form a chain. This chain of evaluands corresponds to the stack segments of the Burroughs machines. This chain is anchored in the PSR and ends with the evaluand for the symbol being currently evaluated by the interpreter.

The evaluation of a function generates a new activation which has its own PSR. The interpretation of this new activation may create additional evaluands attached to the new PSR. These are indirectly connected to the evaluands in the PSR of the activation making the evaluate request by the dependency graph. The request causes the requestor to become independent on the respondent. These links in the dependency graph form a chain through a set of activations.

Definition: A <u>activation</u> <u>chain</u> is a subgraph of the dependency graph. Each edge (X,Y) of the activation chain has the property that X is an activation which has made an evaluate request which caused Y to become the respondent to that request.

The activation chain contains the history of function invocations. It can be used in conjunction with the evaluands it links to provide the status information when a process is suspended. The activation chain is also used to identify generations of activation oriented (automatic) storage.

2.2.6 Sequential and Parallel Execution

A module has a list of statements which can be interpreted in two different ways. The default evaluation of a module causes statements to be interpreted in strict left to right sequential order. In the transition to the next statement, the previous statement result is destroyed. The result of the group is defined to be the result of the last statement executed in the group.

An alternative is to use the parallel function. This function evaluates the statements in an arbitrary order. This may mean actually in parallel if more than one processor is available or interleaved execution. The result in this case is a list made up of the results of each statement.

Definition: The monadic function <u>parallel</u> takes as its argument a module and yields the list formed by concatenating the results of interpreting each of the statements in the module. The order which the statements are interperted is undefined.

When dealing with groups, two additiona components are needed to define the current point of interpretation. The cursor specifies which module is currently active. The statement index indicates which statement within that group is active. Since evaluation of the parallel function causes several statements or groups may be simultaneously active, there can be multiple activation chains. These chains form the activation tree.

The syntactic group markers (braces) have the function of stopping the normal evaluation algorithm. That is, they leave the group unevaluated. If, however, the group occurs in a context where a "value" is needed, the group will be evaluated sequentially. Such a context can be created by the evaluate function, or by other value-oriented functions such as stow or sum. The delay function is used to override a value context.

2.2.7 The Apply Function

When expressions or groups can be the result of a function, it is not possible to use the implicit invocation mechanism. For example. it might be necessary to select one of two functions to apply depending on a truty value (TV). This might be written as

```
(If TV, then quote sin else quote cos)(.5)-}X
```

in the extended syntax. This becomes

```
TV select {quote cos;quote sin} apply list .5 stow X
```

in the basic syntax. The strict syntax for this expression is

```
stow (apply(select(TV;{quote cos;quote sin});list(.5));X)
```

Therefore, an e plicit apply function is needed to associate a function with its operands. If there are no operands, apply reduces to an evaluate function.

Definition: The dyadic function apply makes an evaluate call on its first argument with its expression X(Y). second argument as the argument list. Apply(X;Y) will yield the same result as the expression X(Y).

2.2.8 Selective and Repetitive Control

A powerful, yet disciplined system, requires the abilities for control to flow to one of several alternatives and to provide for repeated execution of a group. The former facilities is provided by the select function, which extracts statements from a group. The repetitive facility is provided by the repeat function which causes a group to be repeated until an iteration condition is satisfied.

It is possible to terminate a group anywhere during the sequencing of the group. The exit function causes the current group to be terminated and yields the value of its argument as the result. When it occurs within a group that is being repeated, it causes the termination of the current repetition. When it occurs in the predicate, it terminates further repetitions.

There are times when it is desirable to conditionally exit from a group with a value. This capability is provided by the conditional function. It takes as operands a predicate and a group. If the predicate yields 0, then the group is not executed and the result of the expression is nil. If the predicate yields 1, the effect is tye same as executing an exit function with the group as its argument.

Gotos are supported but only indirectly. The goto function causes a sequence exception. The standard system action is to reestablish the environment of the label which is the argument of goto. However, the user may field the exception and reject the goto if he desires.

Chapter 2.3

ENVIRONMENT

This chapter discusses and presents the rules for resolving symbols to storage cells in the object base. The various times at which symbols may be resolved are described. The method for providing a context for free symbols is presented. The structure of a procedure is completed.

2.3.1 Phases of Program Execution

The concept of code which is executed at well defined times in the life of an executing program is presented. These time periods are called phases. Phases define when instances of variables may be created. The phases are:

translate
load
activate
execute
deactivate
unload

2.3.2 Local Symbol Resolution

At any point in time, each symbol is associated with a storage cell by a resolution map. Each module may have many activations for every load. Because the contexts of these activations may differ, each activation must logically have its own unique resolution map. Each separate resolution map will be called an environment.

Instead of redoing the whole resolution map, the part of it which remains constant is factored into a common mapping schema. This schema associates each local symbol with a phase identifier and an offset into the storage for that phase. The mapping of local symbols is completed by indicating which instance of each phase corresponds to the desired enviroment. The mapping of free symbols is discussed in the next section. When a resolution map is restricted to the local symbols it is called a local environment.

When each phase is executed, storage is reserved by creating a collective object for that phase. All allocations within that phase become part of that collective object. Therefore, given the offset and the identification of the correct instance of the phase, the mapping is well determeined.

The storage allocated during the loading phase corresponds to PL/I STATIC storage. PL/I AUTOMATIC storage corresponds to the storage allocated by the activate phase. If the deactivate phase does not explicitly destroy the collective object owning the storage, it will remain. This permits coroutines and passing functions up the activity chain.

2.3.3 Context for Free Symbols

Local symbols are resolved to instances of storage cells connected with some phase of the procedure in which they are defined. Free symbols are resolved to local symbols in some other module. This section defines the method for determining which local occurrence is used.

A simple resolution rule is to use the first occurrence of the symbol found by searching the local environments of the modules on the activity chain. This may give access to too many symbols, so the stop function can be used to hide a symbol from the search. A symbol is visible to the search if the stop function was not applied to it in some module in which it is visible.

This rule does not provide for unique local environments, however, so the connect function can be used to define a particular module in which to begin the search. If connect is executed within an active module then the particular instance of storage to be used is also defined. The environment of module "A", which uses connect to bind module "B" is called the predecessor environment of module "B". If a symbol is not found in the predecessor environment then its predecessor is checked, etc. The search terminates when no predecessor exists. The set of predecessors form a chain called the environment chain. Since many activations can exist, these chains form a tree called the environment tree. This is a tree defined on the ownership tree.

When an appropriate local symbol occurrence is found, the resolution map is extended by a process called linking. A reference to the storage cell which is associated with the found symbol occurrence is placed in the resolution map position which corresponds to the free symbol.

2.3.4 Alternate rules for Free Symbol Resolution

The search rules given above must be extended to handle PL/I EXTERNAL scope. What is needed is a method for specifying where in the environment or activity chain the search is to begin and end. This storage for a module in which the free symbol is to be may be defined in terms of relative back references along either the activity chain or the environment chain. It may also be provided by a reference to an existing local environment. Similar conditions could be used to terminate the search.

2.3.5 Modifying Attributes and Values

Having established how symbols are resolved to storage locations, it is necessary to indicate how the contents of these locations are set. There are several functions for this purpose.

The object contained in a location may be changed using the replace function. If only the owned resource component is to be modified then the stow function is used. Each object may have set up constraints on the values it will allow as own resources, so conversions may be caused by the stow operation.

The create function is provided to allow the user to build new objects, given a description of the desired format and an existing object from which to obtain the components of the new object. The description may be a data description or it may be a user defined access procedure. If the existing object is incompatible with the description, a conversion is required to build the new object.

2.3.6 Review of Program Data Structure

Given the rules of chapters 2.2 and 2.3, a program module becomes a complex object. It is a collection of text lists, each of which corresponds to a phase in the life of the program. There is also a table of all symbols directly contained in the module. These are partitioned into local, free, and parameter categories with the restriction that parameter symbols occur only in the execute phase modules. Labels of statements are also in this symbol table, along with references to the phase in which the label occurs.

Each module is basically an ordered structure, where some of the component statements may be unindexed. The index set is the set of line labels or statement labels. The elements of the structure are ordered by line label values. This allows replacements and changes to be made easily.

Multiple entry points are allowed. They are represented by parameters to a common entry point. This entry establishes the argument-parameter symbol correspondences and then branches to the appropriate starting point in the execute module.

Chapter 2.4

MULTIPLE CONTROL STRUCTURES

This chapter treats the problems of exceptional conditions and explicit creation of processes. Both synchronous interupts such as overflow, and asynchronous interrupts such as I/O are defined. The mechanisms for identifying and handling such interrupts are given.

Processes (tasks) may be explicitly created and their execution may be monitored and temporarily suspended. It is through these mechanisms that debugging will be implemented. The data structure of the control tree is described to show how status information may be obtained.

2.4.1 Exceptions and Synchronous Interrupts

When a primitive function is evaluated, conditions which are not built into the language interpreter may occur. These conditions are called exceptions. They cause interrupts which are synchronous with the evaluation of the function. These interrupts are processed by creating a function call which is stacked onto the activation chain including the function causing the exception.

The function for which the exception occured is located in some module "A". The procedure to handle the exception is found by one of three possible rules. Within each module it is possible to define a set of procedures to be used when particular exceptions occur. The first rule is to require the exception handling procedure to be defined in module "A". If it is not then the system action is used instead. The second possible rule is to search back up the activation chain in which the module resides for a definition of the exception handler. This is what PL/I does. The third rule is to search back up the environment chain for the exception handler.

It must be possible to simulate the occurrence of any exception under program control to facilitate debugging. There is an signal function which causes the exception given as its operand. The exceptions will be values in the language so they be used as arguments to functions or combined into sets.

2.4.2 Changing Sequential Flow

In chapter 2.2 the interpretation of a sequential group proceeded in strict left to right order. Most of the programming languages to be supported allowed transfers in the flow of control. Therefore a sequence exception is defined to stop the normal sequence of evaluation and to provide an argument which specifies where the evaluation is to continue. This allows the user to field this exception if disciplined programs are desired.

As a aid to the user there is a cell for each group which remembers the point from which the last sequence exception transfered control. This is an aid to debugging programs with gotos.

The question of transfers of control outside a module is more complex. It is necessary to designate an environment to resume as well as a statement to continue the execution at. This means that, in general, a label has two components. It has a statement index and an environment reference. The environment reference has in it the information on which module to resume.

In the above discussion there was no dependency on the environment to resume still being active. This permits coroutines and the environments of functions which were passed upwards to be "reactivated".

2.4.3 Processes and Monitors

The parallel function does not provide sufficently flexible multipqrgramming facilities. The reason is that the number of processes to be created must be known when the parallel function is executed. The create function is provided to give finer control over the creation of new processes. It causes a new process in the suspended state to be created and attached as subordinate to some process in the activation chain leading to the process executing create. The result of create is a cell name for the new process.

A subordinate process may be activated by applying the start function to its cell name. It may be stopped temporarily with the suspend function. The process which starts a suspended process may continue to run in "parallel" with the started process. When a process has completed, it may terminate itself by the destroy function. It may also be terminated externally by destroy.

If process "A" knows the cell name for process "B" then process "A" is a controling process for process "B". A controlling process can monitor the actions of its subordinate processes. The monitor function suspends the process executing it and starts the process given as an operand. The other operand is a set of events, called intercepts, which can occur in the monitored process. When an intercept occurs, the monitored process is suspended and the monitoring process is restarted. The result of monitor is the intercept designator for the intercept which caused the switch. Breakpoints may be handled by monitoring the execution of the statements with the breakpoints on them.

Monitoring may be undone with the ignore function. It causes the monitoring process to be reactivated with a special indication that it is to ignore the process it was monitoring. The result of the ignore function is nil.

Once a process is suspended, it may be temporarily activated using the inject function. This function is used to execute an expression in the environment of the suspended process. It is used to change that enviroment, investigate the values of variables, etc.

There are cases where it is necessary for one process to be able to suspend a second process only at well defined points in the second process. For example, it is desirable that attention signals interrupt the runing function on statement boundaries. This capability is provided by the priority function which also can be used to give information to the resource manager.

2.4.4 Asynchronous Interrupts

The above interrupts are all synchronized with the execution of the procedures. There are other events such as I/O completion and attention signals which occur asynchronously with respect to the execution of the program text. These may also be handled by a monitoring process. However, in this case the event being monitored may have already occured before the monitoring action is attempted. Therefore, it is necessary to save the event information in case it will be monitored. Setting up the initial value of an event variable is a problem.

There are two ways to treat multiple occurrences of a monitored event. These can occur easily in asynchronous events and in processes which have parallel activation chains. The monitor can be treated as a serially reusable resource and the occurrences beyond the first can be queued. Alternatively, a new copy of the monitoring process can be made to handle each new interupt. This allows a potentially infinite number of copies of the monitor to be created. Currently restricting monitors to be serially reusable seems to be more reasonable.

2.4.5 The Data Structure of Control

The activation tree is a data structure which contains the status information that determines the flow of control. Each activation in the activation tree contains a cursor (group identifier,statement index and expression offset),the process id for the chain in which it resides, and the user identifier. These may be accessed for debugging information like the APL SI vector and to do validity checking on accesses to protected objects. A particular activation may be identified by selection operations on the activation tree. The branches are ordered by their order of creation so numeric indices may be used. It is unlikly that the information in the activation tree can be modified using the normal data structure operations because it would undermine the system discipline.

Chapter 2.5

RESOURCE MANAGEMENT

2.5.1 Summary of the Problems

In an ideal system, all data would be accurate, and no error could be generated anywhere within the system. In the real world, errors occur due to program bugs or hardware bugs. Even if perfection could be achieved, it wouldn't necessarily be marketable since such a system would probably cost too much to produce and run too slowly to be salable. In designing a system, it is vital to specify the techniques to be used in handling the various type of errors that can occur.

One way to contain the effect of an error is to partition the system into a set of levels such that an error at one level cannot propagate to the next higher level in the system. The most obvious such partitioning is that between user data and system data. The following discusses error handling in each of these two categories.

User data can be put into two general categories, private data and public data. A job whose data is all private and which suffers an unrecoverable error may simply be re-run. If the job is run frequently and if errors are common and if it is uneconomic to re-run the job in its entirety, then the job should be temporally segmented. That is, the job should be broken into distinct time segments. In case of an error during one segment, the job is begun again at the end of the previous segment. This is simply the familiar mechanism of checkpoint-restart.

A job that only uses public data has a different set of problems, of which the update-in-place problem is the most obvious. The update-in-place problem is solved by defining a mechanism for gaining exclusive control of a portion of public data, but this solution opens the door to the problem of deadlocks, and it can also cause large quantities of data to be made unavailable to other users while under the exclusive control of one user. Furthermore, if an error occurs so that it is necessary to terminate a job that had exclusive control of an entire data set, it is not clear which, if any, portions of the data set were left in an invalid state. A technique that reduces the scope of data potentially affected by an error, as well as tending to reduce the occurrence of deadlock, is to segment the data into smaller units such as records or fields. One might call this approach error control via physical segmentation as contrasted to temporal segmentation. A job to be performed on a public data set would be be broken into a number of small operations to be performed on all or selected segments of the data set. In case an error occurred, the segment being operated on at the time would be the only segment to contain a possible error. Therefore, the segment could be flagged and the circumstances regarding the error incident could be reported to the Data Base administrator who would see to it that whatever steps were necessary were taken to correct the error.

Errors in system data are another matter. While it may be possible for errors to occur in the system data pertaining to individual users with no more regrettable effect than the termination of some subset of the users on the system, it is not tolerable for any errors to occur in the information the system has about its own structure. For example, it is not permissible for a queue element to be incorrectly deleted from a queue or for the queue to become intertwined with another queue. Errors in this class of data can potentially go undetected for some considerable period of time, a period of time sufficient for them to propagate themselves throughout every nook and cranny of the system. Such an error can compound itself so that it is not possible to know what information in the system is valid and what is invalid. Some approaches to the problem of guaranteeing the validity of system data, as well as of attempting to ensure but not to guarantee the validity of user data, are outlined in Section 2.5.4 on Resource Management.

On batch systems, users were offered in effect two separate sets of functions with which to implement a solution to a problem: those provided by the compiler at compile time and those provided by the control program at execution time. On interactive systems, users frequently intermix compilation and execution. And on systems like APL/360 with excellent debugging facilities, the user may suspend execution at any time to change his programs and then resume execution. Such systems, which allow fluctuating resource requirements for each user, raise problems that cannot be met by the batch-oriented algorithms of OS/360.

An individual writing a program can control the resources available to him in such a fashion as to accomplish the assigned function. The writers of a control program, on the other hand, are faced with the fact that no one can predict all the combinations of functions that can be requested by every statistically aberrant group of users in any given time period, where each function requested implies some resource usage that the user has neither knowledge of or control over. Since the user is not aware of the resources required to accomplish a function he has requested, he cannot assist the control program in anticipating resource usage, and so the control program must constantly be prepared to handle all worst case situations.

Holt, in his recent thesis on deadlock, has distinguished usable resources from consumable resources. Consumable resources refer, for all practical purposes, to the type of interaction between processes typified by the WAIT-POST logic of OS/360. Processes may interact through operations on consumable resources just as they may interact through operations on reusable resources, and therefore, both types of interactions can contribute to the occurrence of deadlocks. There is an important difference, however. A user process may interact on a consumable resource with either a system process or another process within his own job. His process would not interact on a consumable resource with another process in a distinct job. Therefore, the user can hurt only himself through the invalid or badly timed use of a consumable resource. The system also has the choice of waiting on either a user process or a system process. The former case should be strictly outlawed, since it jeopardizes system security. The latter case is normal and is to be expected. The point to be noted is that dependencies between system processes interacting on consumable resources are known at design time, and therefore deadlock possibilities can be handled at design time. Consumable resources should not be a deadlock consideration for system processes.

The following diagram describes a situation noted by R. M. Smith. It illustrates a potential invalid timing interaction between two CPU's which no amount of locking will avoid. The example is

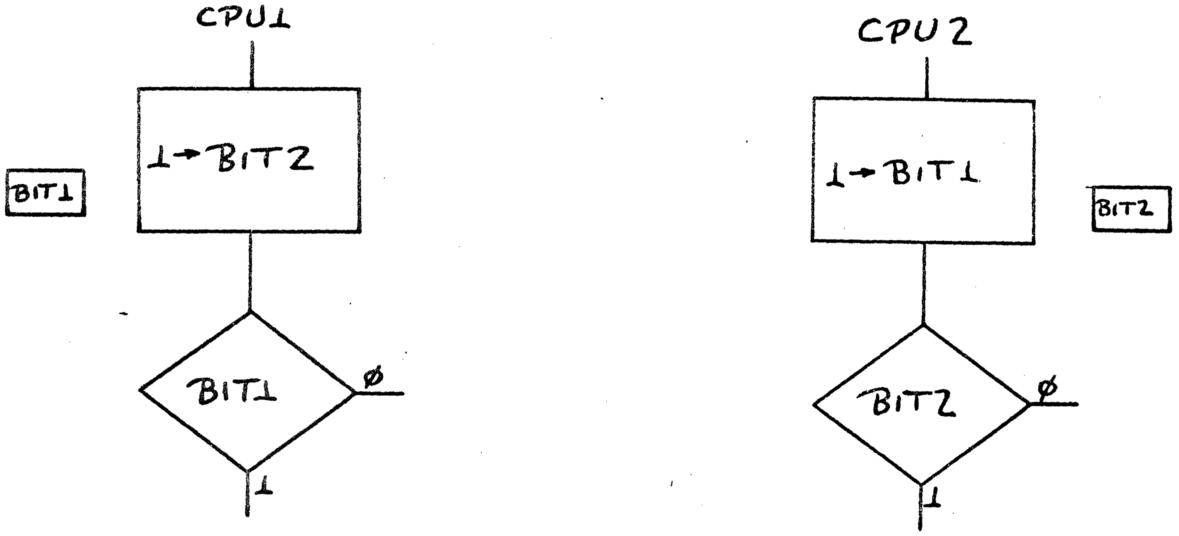


Figure 2.5.1-1

specifically stated in terms of CPU's. It illustrates the sort of timing interaction that must be considered in the design of any multiprocessing control program such as AFS.

In diagram 2.5.1-1, CPU 1 sets bit 2 to one and then tests bit 1, while CPU 2 sets bit 1 to one and then tests bit 2. Both bit 1 and bit 2 are assumed to have been initialized to zero. Bit 1 is physically close to CPU 1, while bit 2 is physically close to CPU 2. If timing interactions are ignored, that is, if it is assumed that all operations are completed instantaneously, then it is apparent that at least one and perhaps both of the two CPU's will emerge from the test of bit 1 or bit 2 having found that the tested bit was set to one. It is possible though that each CPU could send a signal to change the value of one of the bits and then test the other bit before the signal setting the other bit to 1 had been received, so that the two CPU's could find the bits both set to zero.

2.5.2 Classes of Resources

The most fundamental resources in the system are space and time: in the physical implementation, space means storage in the Storage Management Subsystem (SMS) as defined in the System Architecture Manual, and time means execution time on a Program Processing Unit (PPU). Since all objects reside in storage cells, they all require some space in the SMS; and since all objects are processes, they all require some execution time on a PPU in order to respond to a request. By definition, the SMS manages all internal storage, and the PPU's service the requests on the queues for various objects.

On conventional systems, space and time have been managed by software control programs, with the exception of some space management by hardware on buffered machines like the 370/165; on AFS, such control functions will be performed completely beneath the level of SL programming. Because of this increase in hardware control functions, the engineering design must solve a number of problems normally faced only by programmers: For example, if off-line storage is treated as a logical extension of SMS, then the data path for requesting the operator to mount tapes must be dedicated to the SMS; otherwise, a deadlock might arise if the operator was using the console for a non-SMS function that caused paging in the SMS that caused an overflow of on-line storage that required the mounting of a new tape that required a message to be sent to the console that was still busy with the original request. Other possibilities for deadlock could arise if dispatching a PPU required space in SMS and allocating space in SMS required some processing by a PPU; even if normal cases of deadlock were completely eliminated, problems might arise if standard protocols were relaxed when a hardware error occurred and recovery procedures made the SMS dependent on a PPU for emergency measures. If treated systematically, these problems are solvable by a series of levels like those discussed in section 2.5.1: the SMS must be the most fundamental part of the system and can never be logically dependent on services by anything outside of itself. Logical dependencies can be eliminated even in emergencies by dedicating certain resources, such as a special log-out area in a PPU, that could allow a physical PPU to become a logical part of the SMS for a certain period of time. On small machines, such procedures could be used to allow a single PPU to perform all functions: just as the same hardware on a 360/25 can behave alternately like a CPU, a channel, and a control unit, a single PPU could switch hats and act either as a logical SMS or as a logical PPU.

For the remainder of this chapter, we shall assume that space and time are allocated by hardware: the SMS provides a practically limitless amount of storage upon request, and the PPU's are queue driven boxes of hardware that dispatch themselves to service the logical processes. These are big assumptions that imply a lot of engineering design to make possible and even more to make practical. See the System Architecture Manual for more detail about the hardware design and various simulation studies.

Some resources, such as ports, correspond to physical devices that have an independent existence. Other classes of resources are constructed by suballocating space and time: the access machines of objects require time on a PPU to respond to requests; data representations, internal identifiers, procedural descriptions, and PSR's take up storage space in the SMS.

Definition: Every object is a resource that belongs to one of the following classes:

1) Finite: there is a limited number of objects with an equivalent status and ability to respond to requests.
2) Unique: there is only one object with a particular status and ability to respond to requests.
3) Unbounded: the object belongs to a potentially infinite class of equivalent objects; upon demand, a new object of the class can be created by suballocating space and time if available.

Finite objects are ones like printers, where the total number is fixed, but any one or several may be equally capable of satisfying a request. Almost all data objects are unique; copies of read-only objects may be acceptable in some cases, but tables and records like airline reservation or payroll files must have a single updatable copy. Unbounded resources correspond to function activations where a new one may be created for every call upon the function.

One way to increase the apparent number of finite resources is to create function activations that have the same logical properties as the limited resource. For example, a multiprogramming system with only one printer can provide many logical printers by creating multiple activations of a spooling program: each activation may respond to requests exactly like a printer; after receiving a complete document, the activation will compete with other activations for service on the physical printer.

2.5.3 Subsystems

A hierarchical structure for a system is essential to a good design: Each level of the system can be designed and debugged independently. Errors arising in one level cannot propagate to higher levels. And the growth in the total number of possible interactions between objects is linearly proportional to the number of objects, not exponential as in an unstructured design.

The AFS concept of subsystem is the basis for operating systems, user jobs, and networks of systems. A subsystem is a subset of a system in which all interactions with objects outside of the subsystem are channeled through a single resource manager. From the outside, a subsystem behaves like a single object; from the inside, the rest of the system is only visible through the top.

Definition: A subsystem is a subset of the object base with the following properties:

1) There is a single object called the subsystem root from which all other objects in the subsystem are directly accessible (i.e. the subsystem forms a subtree of the ownership tree with the subsystem root as its root).
2) The subsystem root has an element called the resource manager that is a collective object whose elements are synonyms to all external objects used by the subsystem.
3) The subsystem also forms a subtree of the environment tree with the subsystem root as its root.
4) No object inside the subsystem is dependent on any finite resource except the ones whose synonyms are held by the resource manager.

2.5.4 Resource Management in AFS

Resource allocation in AFS basically follows Habermann's algorithm (CACM, July 1969) extended to meet the needs of the AFS system environment. Habermann's algorithm requires that each user define at job initiate time the maximum usage of each resource required by his job. This maximum usage is called the claims specified by the job. During the running of the job, the user requests resources as needed up to the limit of his claims. Upon receiving a request for resources, the system tests to determine (1) whether or not the resources are available, and (2) whether or not a safe sequence exists. If the resources are available and a safe sequence exists, then the request is granted immediately. If one or the other of the two conditions is not true, then the request is not granted until the two conditions have become true. If the request exceeds the claim, then the request is refused.

Definition: A sequence of jobs, JOB1, JOB2, ..., JOBN, is called a safe sequence provided that if every job in the next instant requested all the resources it claimed at initiate time, then JOB1, using the resources it now holds plus those currently free, can run to completion and so free up the resources it now holds, and then JOB2 using the resources it now holds plus those currently free plus those held by JOB1 can run to completion, and so then JOB3 . . . .

It is unreasonable to require the user at the terminal to specify at logon time all the resources that he might use in the coming session. In order to permit the user to request resources which he has not claimed previously, Barry Goldstein has suggested an important modification to Habermann's algorithm. Goldstein's algorithm allows the user to request resources which he has not previously claimed. In response to a request for resources, the system, as in Habermann's algorithm, tests to see whether or not the resources are available and whether or not a safe sequence exists. If both conditions are true, the resources are granted immediately. If either condition is false, then the user has to wait unless making him wait would create a deadlock. The result is that a batch user who never exceeds his claims will never encounter a deadlock and therefore need never prepare for handling deadlocks. On the other hand, a terminal user can dynamically request resources that had not previously been claimed at the cost of occasionally having to program his way cut of the deadlock.

There are conflicting demands made by the two needs to avoid deadlocks in allocating resources and to allocate resources in a network. Avoiding deadlock requires that there exists a single centralized allocator with complete knowledge of all the processes in the system and all the resources assigned to those processes. Running a network, on the other hand, requires that each installation in the network enjoy a measure of independence from the other installations. If centralized resource allocation were to be performed in a network, then every request for resources would have to be referred back to the single specific node in the network that contained the resource allocator. Since this is unfeasible, a method must be found for allocating resources at each node in a manner that is as independent as possible from the resource allocation decisions made at other nodes. This form of resource allocation can be accomplished providing that additional constraints are placed on the safe sequences maintained by the resource allocators in the network.

Let the system be composed of disjoint sets of resources and for each set of resources define a resource allocator. Assume that the resource allocators are all at the same level, and on top of them define a tree structure of resource allocator coordinators. The particular tree structure is arbitrary but is fixed for any given network.

Local jobs are ones that only use resources in one of the disjoint sets of resources. Distributed jobs are ones that use resources from two or more of the sets of resources. A job can enter the system at any node. A local job is transmitted to the node at which it will execute (if it wasn't submitted at that node). A distributed job may enter the network at any node but will be passed up the tree of resource allocator coordinators and possibly back down some other branch of the tree until it arrives at the lowest level resource allocator coordinator (or RAC) that has jurisdiction over all the resources claimed by the distributed job. The job is then broken up into subclaims tagged with the following field:

COUNTER.TIME.RACID

which specifies the position in the safe sequence relative to other distributed jobs that the current incoming distributed job is to occupy. Generally the idea is that distributed jobs should be processed in FIFO order. The problem is to determine the meaning of FIFO in an environment in which time scales may not be synchronized. A simple time stamp does not suffice, since different RAC's using different clocks could stamp requests for different jobs to be sent to the same safe sequence with the same time. Consequently, JOB1 might precede JOB2 on one safe sequence, while JOB2 preceded JOB1 on another safe sequence. To avoid this and other timing problems, the claims sent down to the resource allocators are tagged with the value COUNTER.TIME.RACID. TIME is the value of the RAC's time stamp, RACID is the identification of the RAC sending the request down, and COUNTER is the value of a counter maintained by the highest level RAC and sent down to all lower RAC's. This counter value acts as an artifical but uniform time scale for all RAC's in the system. Since all distributed jobs maintain the same relative ordering with respect to each other in all safe sequences in the system, no deadlocks occur in the network.

Holt has pointed out (CACM, January 1971) the possibility of jobs becoming effectively blocked in a safe sequence. Such a situation could occur if a sequence of high priority jobs continually occupied so much core that a a low priority job never had its request for a large amount of core satisfied. Consequently, the low priority job would be blocked indefinitely and could not be guaranteed to complete in any given time. To assure that every job will eventually complete, Holt proposes that jobs in the safe sequence be tagged with a time value that indicates the length of time they have been waiting in the queue. Then construction of the safe sequence is biased to favor those jobs that have been waiting longest.

Shoshani (CACM, November 1969) has described the problems of permitting simultaneous access to the elements of a list structure. While it is not clear that any of the specific approaches that he recommended should be adopted, AFS must provide solutions that are at least as effective.

The THE System as described by Dijkstra (CACM, May 1969) contained a very attractive approach to the problem of avoiding deadlocks in the system. The system was structured into six levels. Level 0 consisted of a clock and dispatcher. Level 1 consisted of the paging controller. Level 2 was the message handler. Level 3 handled source-sink input/output. Level 4 held the problem programs, and Level 5 was the user. One inviolate rule of the system was that no process at a lower level could wait for a process at a higher level, though processes at a higher level could wait for a process at a lower level. Consequently, deadlocks were avoided partly through the enforcement of this simple rule. Some such structuring should be undertaken for AFS not only to prevent deadlocks, but also to reduce the level of complexity of the system to a more manageable degree, and thereby allow a more complete and accurate design to be formulated.

Chapter 2.6

FUNCTION SET

2.6.0 Introduction

The operators of SL are the basis of the system. The elaborate structure of dyadic objects and operators to work on them is intended to implement an attribute examining system. The operators are the lowest level active element which can be programed. In this respect they are like S/360 instructions. The detailed function of an operator depends in part on the attributes of the operands at the moment of execution. In this respect they are like APL functions. The operators are also responsive to the environment in which they are executing as determined by explicit program declaration statements and the activation chain. This aspect of operators is the contribution of SL.

The operands of an SL operator are objects residing in the storage cells associated with the operand symbols in the expression containing the operator symbol. The operator symbol itself is associated with a storage cell which contains the function object to be activated. This last relationship enables easy operator redefinition when necessary. The dyadic nature of the operands complicates the definition of the operator at the object level as compared to that of a simple system. The purpose is to simplify the description at the program level. In analogy, the description of floating point operations are more complicated than those of the corresponding fixed point operations; the existence of these operations, however, simplifies program statement by eliminating the need for scaling.

2.6.0.1 Arguments

Part of the definition of a function is the specification of the number and type of its arguments. For monadic and dyadic functions written in infix notation, the arguments can be recognized by having their symbols appear next to that for the function. This technique is used by APL to distinguish between monadic and dyadic functions. In the prefix form of notation the function must contain sufficient information to specify the number of arguments. The spelled out forms of the functions, which are different for monadic and dyadic forms, must, accordingly, be used in the prefix notation. Functions which require more than two arguments will be described as monadic, with their operands taking the shape of lists of three or more members. The symbols which are assigned to functions may do double duty in the sense of being used for both a monadic and dyadic function. These symbols can only be used for infix notation, where the distinction can be made syntactically.

Not all functions have single character symbols assigned to them yet. In some cases in which this has not been done we have indicated which pairs of one monadic and one dyadic function should share the same symbol. As a first principle one might try to define the monadic form to be related to the dyadic through some sort of default, ie., having the monadic form equal the dyadic with some special value for the missing argument. The trouble with this is that for most symmetric operators the natural special value makes the function into a no-op. For example, monadic _plus_ is a standard no-op. To get maximum mileage out of the basically limited number of single characters the monadic function is not usually defined in terms of the dyadic for symmetric functions. An attempt to be reasonable is made, however, in many cases following the example of APL.

In addition to the number of arguments which a function expects one must specify the type of argument.

Definition: A function may place certain restrictions on the types of its arguments. Any argument meeting these restrictions is called _primitive_ to the function. The action of the function on an argument of such a type is determined entirely by the definition of the function and not by function distribution.

For example, numbers are primitive to the arithmetic operations. Zero and one are primitive to the logical operations. A mcre subtle example is _select_. Any object can be primitive as a left argument. Any indexed object is primitive on the right. If, however, the right argument of _select_ has a restricted index set, say it is a list, then the primitive objects on the left beccme restricted, respectively, to integers.

2.6.3.2 Function Distribution

It has gradually been accepted in programing languages that distribution of functions over structures of operands should be automatic as in APL rather than requiring explicit loops as in early FORTRAN. Since our structures are very general, cur definition of function distribution must be so too.

We shall discuss function distribution for dyadic functions. The situation for monadic functions is, in fact, simpler and can be deduced from the dyadic case. Suppose that a function appears between two objects neither of which is primitive. The function examinies the two objects to see if they are two collective objects with identical index sets. If not, an error has occurred. If the condition is satisfied, the function is applied iteratively to the elements of the structures producing an identically indexed collective object as the result. If any pair of objects is not a pair of primitives, the analysis is executed recursively. If at any stage of the recursion one operand is primitive and the other not, the primitive operand is imbedded by replication in a collective object matching that of the other operand and the function is evaluated.

Note that function distribution applies only over indexed structures, most usually, in practice, over lists and arrays. Objects of type closure not primitive to the function being distributed are not uncovered for distribution. Stopping distribution is one of the functions of encapsulation.

Enclosure can also be used, in conjunction with function definition, to modify, as well as simply to control distribution. Suppose, for example, that one wished to carry out rational arithmetic with proper fractions kept as integer pairs. One would wish for a function, ratsum, which is sum for integers and defines the arithmetic sum for rationals. One defines rational numbers as enclosed collective objects consisting of two integers and an identifying field. Objects of type closure are made primitive to ratsum. When a closure is encountered, the function itself analyses the object to decide what to do with it.

Function distribution can also be explicitly controled by certain functionals, as discussed in 2.6.5.

2.6.1 Program Structuring Operators

This section has some symbols which are not properly operators. That is, they are not encountered at execute time. However, they are included for completeness. The operators given here are used in constructing a runable procedure from symbol strings.

2.6.1.1 Parsing operators

These operators are used to make it possible to break the text into units and to build a parse tree.

quote (cf 2.2.3.2)
delayed parse (cf 2.2.3.2)
braces (cf 2.2.3.3)

2.6.1.2 Scope Building Operators

These operators allow the user to define symbol occurrences as being local, parameters, or free and to build the context in which the free symbols will be resolved.

<u>insert_symbol</u> (cf 2.2.2)
<u>lambda</u> (cf 2.2.3.5)
<u>stop</u> (cf 2.3.3)
<u>connect</u> (cf 2.3.3)
<u>load</u>

2.6.1.3 Unique Name Creation

These operators allow the user to create unique names from existing names. For example, they can be used to create local temporaries. These unique names are not normally printed when the symbol table is dumped.

<u>unique-name</u>

2.6.2 Object Composition

This section includes the operators which are used to construct objects from the primitive ojects of the system.

2.6.2.1 Descriptor Defining

These operators are used to build up the components of an object description from the built in access mechanisms or attributes. The result of these operators is an access machine.

2.6.2.2 Object Constructor

The object constructor <u>create</u> takes as operands an access machine and an existing scalar or collective object and produces an object which is a copy of the existing object converted to be consistent with the given access machine.

2.6.3 Structure and Index Operators

This section contains the operators which are used to build complex data structures. It includes such categories as storage management, index sets, structural combination, and explicit structure linking.

2.6.3.1 Index Set Operators

These are the basic operators which make use of the indexing facility and alter and examine index sets.

<u>select</u>

The dyadic operator <u>select</u> takes for its second operand an indexed collective object and for its first operand an index object of its second operand. The result is the corresponding element of the collective object.

<u>ilist</u>

The monadic function <u>ilist</u> takes an indexed collective object for its operand. The result is a list of the index set for the collective object. Since the index sets for common objects may be quite large, this operator must be used with caution.

Structures of arbitrary complexity may be built from collective objects since their elements may themselves be collectives. Because of the generality there is no way in the strict syntax to index into subobjects other than by repeated use of the indexing operator. For example to refer to an element on a sublist of a sublist of A one writes:

```
4 sel (1 sel (2 sel A)).
```

<u>ibase</u>

The monadic function <u>ibase</u> takes an array or list for its argument and returns its base list. This is a list structure of depth 2 or 1 respectively which describes the structure of the argument.

<u>shape</u>

Monadic <u>shape</u> takes an array or list as its argument and returns the shape of the argument. This is a list structure of depth 1 or 0.

<u>igenerator</u>

The monadic function <u>igenerator</u> takes a scalar number for its argument. It returns a list whose shape is given by the argument and whose elements are its own index set.

For details on the preceding functions, refer to the table in section 2.1.7. Note that shape is the rho operator of APL. For primitive arrays, igenerator shape yields ibase.

Additional operators are:

name_value

The dyadic function name_value takes as its arguments a value and an object that is to be treated as the index for that value in any collective object in which the result of name_value occurs. It can be used to pass keyword arguments in an evaluate request.

2.6.3.2 Storage Management

These operators are used to add and delete components of collective objects by inserting and deleting storage cells in the collection own by the object.

insert
delete

2.6.3.3 Stuctural combination

These operators are used to piece separate structures together to form a single structure. There are several operators because of the different ways that structures may be combined. The simplest structure is a list. There is an element of indirection in a list which must be carefully controled. For example, let

A = (,a,b,c),

and

B = (,d,e,f).

We must distinguish between the lists

C = (,a,b,c,d,e,f)

and

D = (,A,B).

We introduce four list constructing operators which enable us to construct C and D from A and B, as well as to perform other operations.

catenate

Catenate takes two operands, each of which is a list. The result is a list comprising the elements of the two lists.

augment

Augment is a dyadic operator. The left argument must be a list. The right argument is added to the list.

list

List is a monadic operator. It accepts any object as its operand and forms a one element list with the argument as the element.

ravel

The monadic operator ravel takes as argument an indexable object Q. It produces the result

ilist Q sel Q.

We now observe that the list C can be formed by A cat B. The list D is formed by list A augment B. It is possible that catenate may be redefined to perform limited type conversion so that vectors and lists can be combined. In particular, by using the 0-vector as a left argument for cat a vector can be created from its elements by first forming a list and then converting.

The next two operators permit the formation of general arrays from lists and the reshaping of existing arrays.

reshape

The dyadic operator reshape takes for its left argument a shape, i.e., one of the types of object in the shape row of the table in 2.1.7. The left argument is raveled and then inserted in odometer order into a structure described by the left argument. The right argument is truncated or replicated as necessary.

This operator is the dyadic rho of APL. The order of entry of elements into the structure is as in that language. We refer to this order as the odometer order.

rebase

Rebase is a dyadic function. Its left argument must be an index base list. The right argument is raveled and inserted into the appropriate structure in odometer order.

2.6.3.4 Operators for Composing and Decomposing Scalars

These operators are used to build an object which is to be treated like a scalar from a set of components and to obtain the components of an existing scalar object.

enclose

The monadic operator enclose creates a scalar object whose owned resource is the operand. The resulting object is a scalar of type closure. An enclosed synonym is a metonym.

disclose

The monadic function disclose takes as argument a scalar of type closure. The result is the object which is the resource of its operand.

2.6.3.5 Explicitly Linked Structuring

These operators are used to build structures on components which are not owned by the structure but are only referenced by it. These references may be explicitly followed or they may be implicitly followed when that referencing component is selected.

- <u>access</u>
- <u>point</u>
- <u>ultimate</u>
- <u>synonym</u>

2.6.3.6 Implicitly Defined Data Structures

These operators really define data structures but may be used when the data structure is not finite. They define a rule which completely determines the value of each component when the operands of the operators are given. They act like encodings of the data structure. This is similar to the implicit definiton of a set using a predicate the elements of the set must satisfy.

- <u>igen</u>
- <u>step</u>
- set notation

2.6.4 <u>Operators for Modifying Objects</u>

This section describes the operators which are used to modify either the own-resource component of an object or the contents of a storage cell.

- <u>stow</u>
- <u>replace</u>
- <u>remove</u>

2.6.5 <u>Control of Function Distribution</u>

In addition to the <u>enclose</u> and <u>disclose</u> operators, which provide indirect control of the distribution of functions over collective objects, a number of functionals provide direct control. This explicit control is only provided for lists and arrays, since those are the only collective objects whose structure is explicitly defined.

<u>reduction</u>

The monadic function <u>reduction</u> takes a list of three elements for its argument. The first is a dyadic function. If the second argument is an array the definition of reduction is as in APL, with the third argument replacing the APL subscript. If the second argument is a list the result is the same as for a vector with the same entries. If the third argument is omitted the default is as in APL.

<u>inner product</u>

This functional is defined as in APL for arrays. If the arguments are lists the result is the same as for the corresponding vectors.

<u>outer-product</u>

This functional is defined as in APL for arrays. If the structures are lists, the result is not a matrix but a list of the expected elements in odometer order.

One desirable feature of arrays is the ability to treat them as scalars in one or more dimension so that they can distribute in those dimension as scalars do. This feature is provided in APL by treating 1-vectors as identical to scalars and similarly treating a length of one in any dimension. This achieves one desirable feature at the expense of another, viz., maintaining the distinction between scalars and other arrays. We believe that this distinction is worth maintaining and that arrays must be of identical structure for function distribution to occur. To provide the flexible matching, we introduce another kind of object.

Definition: A <u>partial array</u> is like an array except that one or more entries in its base list may be scalars.

A partial array is indexed exactly like the corresponding array in which scalar entries in its base list have been replaced by one element lists. The index set in scalar dimensions is the entry in the base list for that dimension. The function ibase applied to a partial array produces the list structure of mixed depth described above. The function shape applied to a partial array produces an error.

<u>map</u>

The dyadic functional <u>map</u> occurs between a function and an argument which is a collective object. It forces the function to refuse to accept the object as a primitive and to distribute over its elements.

2.6.6 <u>Element and Pattern Searching</u>

Two operators are used to provide the reverse operation to indexing.

index

The dyadic function *index* takes for its left operand an array or list. The right operand is any object. The result is the index of the first occurrence of the object in the array or list if it exists

It is desirable to be able to search for a sub-collection of objects. By this we mean searching for one array or list imbedded in another. For this imbedding to be defined a way of combining arrays of index objects must be defined. In the typical case of primitive arrays, arithmetic plus, together with its distribution properties can be used.

locate

There are two cases of locate, depending on whether the operands are arrays or lists.

a) Let A and S be arrays with I the index object array of S. Let the ranks of A and S be equal. Let comb be a function defined on the index objects of A and S. Then A loc S is B, an index object of A, such that

B comb I sel A <-> S

b) Let A be a list and S a pseudo-list. For lists comb is arithmetic sum. In this case *locate* is defined so that the Coppola identity,

A loc S + ilist S sel A <-> S,

is satisfied.

2.6.7 Computational Operators

This section describes the operators for numerically oriented computation. They create a new object using their operands as input to a rule of combination. Hence they always cause copying to occur. The operands of an operator are objects; the result, another object. The operands, to enable implementation of standard languages, must contain several kinds of information in their descriptors. There must be the information necessary to interpret the string of bits or whatever is in the machine as a number. In addition there must be the information which the programer associates with the operand through his declaration statements. For our present purposes a number or value is a concept not indigenous to SL in terms of which we describe the functions of the operators. We assume that a value is something understandable to a user so that defining operations in terms of values makes sense. We shall define the value to be used in operations in terms of a fixed radix representation with radix ten. We assume approximately the goals of PL/I but not the achievement of any particular implementation. We assume that the user can designate range and precision or precision only (fixed point and floating point, respectively) of his stored operand. During expression evaluation the machine will keep at least the declared precision and, usually, no more than N digits, where N depends on the machine. Running expressions on a machine with larger N will not yield less accurate results. N is currently a machine dependent parameter. In the case of SL it will presumably be declarable as part of the program ambience.

The storage operands are held to the precision specified by the programer. During expression evaluation greater precision will generally be held in temporary storage cells. This is analogous to the extra guard byte of precision held for floating point numbers during S/360 instructions. Except for division the precision retained will be at least as great as that maintained by PL/I.

A further complication to operator specification is language dependency. A classic example of this is the FORTRAN divide. If the variables being divided are integers, the FORTRAN result is the integer obtained by truncation of the correct answer, regardless of the result destination.

a ÷ b (FORTRAN) <--> (xT) x floor abs T<—a ÷ b

Naturally the scope of variability in this area is huge. If every programer chose to define differently the results of all possible ordered pairs of input descriptors the language dependence is SL would be unmanageable. Our goal is to provide enough flexibility to provide a reasonable set of alternatives for future growth. Special glitches for today's anomalies will be provided. It is hoped that they will wither away.

For the nonce we will define the logimetric operators over the range of fixed and floating decimal numbers with base 10. PL/I notation will be used to designate the current descriptors, i.e., (p) or (p,q) denotes precision. We assume that numbers are kept in signed true form. We also assume that the program has given some specification of the ambience of execution. This is usually derivable from the data description for the entire expression.

When an operator executes it knows the following:

the values of its operands, including the location of

significant digits
the declared or computed precision of its operands
the value of "N", the maximum precision to be used in the present environment
the precision required in the present expression, determined from the controling assignment
the number, K, of operators in the present expression

The machine can use run-time values to help with precision problems, since it runs interpretively. The value of an operand may contain more digits than its specified precision. This happens typically after a divide operation. A reasonable value is specified for the precision of the result. Additional digits up to the precision available may be kept since they will increase the accuracy of the result, but their loss will not cause an exception.

The "precision required" will derive from the declared precision of destination objects for assignment operations. In general a number of digits appropriate to the final assignment in a statement will be kept. This assignment will sometimes be called the controling assignment.

2.6.7.1 Operators with Identical Domain and Range

These operators are associative and may be used where associativity is required to make good use of the operator. The reduction functional is an example where associativity is needed.

2.6.7.1.1 Numeric Operators

The primitive arguments for the numeric operators are numbers in all cases.

plus and minus
: The monadic functions plus and minus return the same value and significance and precision as the input, with the sign changed in the case of minus.

signum
: Signum returns a single digit with precision (1,0) and value 1, 0, or -1, according as the input is positive, zero, or negative, respectively.

recip (÷)
: Reciprocal returns a value equal to one divided by the value of the input. The result precision is (p,q), wherein p is equal to the precision of the input, and q is chosen to place the first significant digit of the result value at the

left of the field. In addition, if the operation is followed in the expression by multiplication or exponentiation, the value is kept as a rational number if the absolute value of the denominator is 256 or less. Further, if the division is not exact in the specified field, additional digits up to N are kept, but not considered crucial; i.e., their truncation will not cause an exception.

ceiling and floor

If the precision of the input is (p,q), with $q \geq 0$, ceiling and floor return the appropriate integer with precision (p-q,0). If $q < 0$ a domain exception occurs.

There are additional monadic operators, to wit:

exp
ln

There are the dyadic forms of these operators:

sum
difference
product
quotient
max
min
power
log

The quotient function returns a single scalar result, the quotient of its arguments. Two additional dyadic operators are related to this one. Together they provide the functions provided by two different definitions of the function which has been called "mod" in some languages. It seems desirable to get away from the name "mod" altogether to avoid further confusion. The names we have chosen are, unlike "mod", consistent with mathematical usage.

quotient_remainder

The dyadic function quotient_remainder returns a two element list. The first element is the same value as that returned by the quotient function. The second element is the remainder.

residue

The dyadic function residue is defined as in APL.

magnitude

The monadic function magnitude yields the absolute value of its argument.

2.6.7.1.2 Logical operators

These operators return domain errors unless the input values are within acceptable limits of 0 or +1. If the restriction is met, the return is the appropriate single digit number with value +1 or 0. The result precision is (1,0). Note, for example, that ¬¬ may not be an identity operation.

not (¬)
and
or
nand
nor

2.6.7.2 Operators with non-uniform Domain and/or Range

These operators are not associative at all times. In some cases the range may be a subset of the domain so when the domain is restricted to that subset the operator is associative.

2.6.7.2.1 Range and Domain Differ

These are primarily comparison operators but the elementary search operators are also included in this class.

eq
ne

Equal and not-equal take numbers and strings as primitives. With string primitives their definitions are the and and or reductions, respectively, of the result for ordinary lists.

Additional operators are:

le
lt
gt
ge
member

2.6.7.2.2 Dyadic Operators with Heterogeneous Domains

These are not exactly computational operators since they really build new structures from existing ones. However, they are grouped here because their inputs are computational.

expand
compress
base-value
representation

2.6.8 Selection Operators

This section consists of the operators which only define an access path to an object. They do not make a copy of the selected object nor do they modify that object.

select
take
drop
rotate
reverse

2.6.9 Control Operators

This section describes the operators for directing the flow of control. It is divided into two parts. The first part is concerned with a single control path. The second part has the operators for multiple control paths.

2.6.9.1 Sequential Control

These operators are used to start control flowing through a program and to modify and control the sequencing of the text of that program.

evaluate (cf 2.1.4)
apply (cf 2.2.7)
exit (cf 2.2.8)
repeat (cf 2.2.8)
goto (cf 2.2.8)
delay (cf 2.2.6)
conditional (cf 2.2.8)
signal (cf 2.4.1)

2.6.9.2 Multiple Control

These operators allow parallel execution of expressions and are used to create, control, and monitor independent processes.

parallel (cf 2.2.6)
create (cf 2.4.3)
destroy (cf 2.4.3)
suspend (cf 2.4.3)
start (cf 2.4.3)
monitor (cf 2.4.3)
ignore (cf 2.4.3)
inject (cf 2.4.3)
priority (cf 2.4.3)

2.6.10 Resource Coordination

This section describes the operators used for input/output, information flow between processes and for the allocation of resources. These operators are a separate class because they interact heavily with the arbitrator.

2.6.10.1 Information Flow

These operators are used both to syncaronize independent processes and to provide the means for information transfer between two processes. There functions are discussed in section 3.4.2.

<u>send message</u>
<u>wait message</u>
<u>send answer</u>
<u>wait answer</u>
<u>introduce</u>

2.6.10.2 Resource Allocation

These operators are used to make preliminary claims and to acquire resources known to the context in which they are executed. They also are used to release the resources. See chapter 2.5.

<u>claim</u>
<u>free</u>
<u>acquire</u>
<u>release</u>

2.6.11 Edit and Search

This section introduces the operators for editing and searching. The approach to be used is to encode the transformation for a finite state transducer which takes as input the encoding, and the string to edit or search and produces as the output the result. The machine must be at least a generalized FSM but even more power may be required.

2.6.11.1 Dyadic Translate

The form is

translate(m;k),

where

m is a translate machine
k is the translate subject.

The purpose of translate is to translate the elements of the translate subject into an output form, subject to the constraints, manipulations, and transformations specified by the translate machine. The types of translations which can be specified range from the finite state operations of the 360 EDIT, EDMK, TR, and TRT through Fortran FORMAT and PL/I PICTURE processing to interpreting/compiling programming languages and recognizing/transducing formal languages.

The translate subject is a collective object which is generally of type list. The elements of the list are the objects (e.g. tokens, characters, symbols) upon which the translate machine is to operate.

The translate machine is a collective object which owns two objects: the initialization function and the translate table. The translate table is a collective object of matrix extraction. Its index set is the cross product of the set of distinct elements I in the input and a set of states S. The elements of the translate table are objects which respond to an execute request with a value of nil, undef, or an element of S. Hence, these elements may be functions (generally triadic), lambda-expressions, groups, variables, or constants, whose evaluation may entail side-effects (such as modifying a push-down stack or adding to an output list). The initialization function is basically similar to an element of the translate table. In response to an execute request, it may perform some housekeeping (or pre-processing, such as stack initialization) duties as a side effect, and return a value which is that element of S (together with the first element of the translate subject) at which translation is to commence.

Translate operates in the following manner: After initializing the output(o) to undef and the input marker(i) to 0 (i.e., the zeroth position in the translate subject), the initialization function is called. The resulting initial state, along with the current input, determine an element of the translate table to be executed. This, in turn, produces a new state as its value which, together with the next translate subject element, can again be used as an index into the translate table object. This procedure is applied iteratively until either

a) the new state is nil

or

b) the new state is undef .

Case (a) occurring concurrently with exhaustion of the translate subject signifies a successful translation, in which case the output(o) is returned as the value. Any other termination condition indicates that an error has occurred, in which case an exception is raised and the value to be returned is a collective object consisting of the uncompleted output(o), the input marker(i), the state(s) at the time of the error, and the complete translate subject.

The input marker and output may be manipulated by the elements of the translate table, thus providing extended finite state operations. By suitable specification, translate has all the power of a (simulated) Turing machine. In addition, recursive calls on translate permit the simulation of tree-like automata and non-deterministic automata.

In SL-like terms, the function could be written as

```
      (m,k) lambda
       {dcl new (i,s,o);
        undef->o;
        0->i;
        apply(sel(1,m);(o,i,k))->s;
        repeat(s≠nil∧(s≠undef);
                {apply(sel(
                          (sel(i+1->i;k),s);
                          sel(2,m));
                      (o,i,k))->s});
        sel(s≠nil;{o;(o,i,s,k)})
      }
```

where sel stands for the select function. As an example of translate, consider the following SL program segment:

```
                       .
                       .
                       .
 0->n;
 (o,i,k)λ{A}=>T[0;A];
 repeat(  n+1->n<10;
          {(o,i,k)λ{o cat ('$',n)->o;
                    test(o;i;k);
                    B}=>T[n;A];
           (o,i,k)λ{o cat list n->o;
                    test(o;i;k);
                    B}=>T[n;B]});
 syn T[1;B]=>T[0;B];
 nil=>T[nil;B];
 (o,i,k)λ{sel(shape(k)-2=i;{;o cat list '.'->o});
          sel(3 res (shape(k)-5-i)=0;{;o cat list ','->o})}=>test;
 translate((A,T);'00 10 2345678')->result;
                       .
```

.

.

where sel is the select function, res is the dyadic residue function, and cat is the catenate function.

The value of result would be the object
$1,023,456.78

2.6.11.2 Monadic Translate

The monadic form of translate

translate(k)

takes as argument only the translate subject k. It is assumed that k is a SL string in external syntax form. The result of translate is the internal, functional SL equivalent of the external syntax.

Part 3

SYSTEM CONCEPTS AND FACILITIES

Part 2 described the fundamental structures at the basis of AFS. These structures provide the equivalent of a "bare machine" that executes SL directly. This part describes system facilities built on top of that basis to provide a rich set of user oriented functions.

An important concept of AFS is that there are no privileged instructions, only privileged resources. Since all operations available to the operating system are therefore available to any user, special purpose systems as well as IBM standard systems can be designed and run like ordinary jobs.

Chapter 3.1

SYSTEM DESIGN CRITERIA

The system design criteria are enumerated in terms of applications, operational environments, and service modes. These are derived from FS market requirements. The object for the exercise is to classify the requirements into three somewhat mutually independent categories. Criteria from each category are used as the basis to begin one aspect of the system design effort.

Topics to be described in this section of the report give an indication of the current effort to satisfy the application and certain of the operational environment criteria.

3.1.1 Applications

Criteria included in this category are focused on the types of user applications which are prevalent during this time frame. Important applications include:

- Data Entry/Data Retrieval,
- Data Manipulation and Computation, and
- Data Communications.

3.1.2 Operational Environments

Operational Environments are concerned with the aspects of physical demands and constraints on a system relative to performing user applications. Examples are:

- Reliability, Serviceability, Availability
- Size of data base
- Number of lines and terminals
- Geograpaical distribution of:

- Users
- Terminals
- Data Bases
- System nodes (in a network)

- Traffic rate/message mix
- Response Time
- Security/Privacy

3.1.3 Service Modes

Service modes are concerned with the manner in which system services are exercised by a user in order to satisfy his application requirements when subjected to the appropriate operational environment constraints. Service Modes include:

- Transaction based (Routine processing),
- Interaction based (Non-routine processing),
- Event-triggered,
- Batch, and
- Message Switching.

Chapter 3.2

ENVIRONMENT MANAGEMENT

The principal functions performed by an operating system are to set up the environments required by procedures and to execute them within those environments. Current systems have normally assumed a "standard" environment, in which all user code is to operate. This environment is usually different from that required by components of the operating system itself. Software tools for the user, especially compilers, are designed under the assumption that the generated code is to operate in the "standard" environment. Most tools, therefore, are unusable in other contexts, and the "standard" environment is often unsuited to more ambitious user subsystems. When faced by non-standard circumstances, therefore, the user or systems programmer is on his own and usually resorts to the worst kinds of trickery -- by necessity, not by choice. As is well known, OS/360 is full of such ad hoc solutions.

Since environment construction and management primitives are included directly in SL, either IBM or user systems may set up environments which are well suited to their needs. This implies a new outlook on software tools, which no longer are allowed assumptions about the environment in which generated code is to operate. In addition, as seen in Part 2, steps are now required to imbed a procedure within its operating context. Software subsystems and virtual systems follow as corollaries of this approach. The disciplines which insure integrity, privacy, and security pervade the system, including all subsystems.

System control and the command language is also embedded in SL: user commands would translate to expressions in SL. This view of control is similar to the approach taken in the July 1970 draft of the CCL report. In CCL the control functions were programed in CCL procedures. This approach gives the user more flexibility in defining work flow than a static language like JCL. Nesting of control procedures is also allowed. Any IBM standerd command language will be provided as part of the system support as well as the SL control functions.

3.2.1 General System Environments

The root of the system ownership tree is the system resource manager, which has the responsibility for resource control. One such resource is the subsystem landlord.

3.2.1.1 System Specification

The subsystem landlord is a collective object that owns all the highest level operating (sub)systems initially accessible to the user; these include systems like DOS, CP, CMS, OS, and TSS, as well as user-defined operating systems and SL subsystems. Insert requests may be made on it to add new operating systems, or to delete existing ones; suitable access rights for this capability will be installation-definable and presumably secure. General users may not make modifications to dedicated operating systems owned by the subsystem landlord.

Interaction among operating systems, such as with CP/CMS, may be achieved via accessors assigned by the subsystem landlord.

3.2.1.2 Resource Control

The resource manager is a collective object which controls the allocation of space, time, and external devices, and the sharing of library resources global to multiple operating systems. It may logically (i.e., via synonyms) group elementary resources (such as ports, storage devices, or libraries) into new generic collective objects called resource packages, and permit access to them by other system objects via the accessor mechanism. In this manner, operating systems may be given control over specific groups of terminals, device classes, channels, libraries, etc. Additionally, judicious assignment of access rights may permit more detailed delineation of resource availability.

For resources which are shared by or accessible to more than one operating system, the resource manager will monitor the accesses and guard against lockups and deadlocks.

3.2.1.3 Initial Interpreter

All ports to the outside world are controlled by the resource manager. Some subsets of these ports may be combined within a resource package and accessors to that resource package given to a dedicated subsystem. Such ports are called dedicated ports; a user signing on to one of those ports is immediately confronted by the subsystem to which it was dedicated (e.g., a port dedicated to TSS would require the user to enter the usual LOGON message). No other kinds of jobs save those meaningful to the controlling operating system may be entered from that port.

Free ports, on the other hand, are initially unassigned by the system. These ports fall under the control of the initial interpreter which is part of the resource manager. The initial interpreter responds to user commands entered via a free port, and creates the corresponding subsystem under the subsystem landlord; such subsystems may include, for example, user-tailored versions of OS and DOS, or SL subsystems. Initial resource claims for the new subsystem may also be honored by the initial interpreter before control is transferred.

3.2.2 Operating System Environments

An operating system is a subsystem whose access machine is a process called the control program and whose resource consists of a set of jobs and a collection of libraries called the system input. Operating systems may be either dedicated (bound) or free. Dedicated subsystems enjoy a semi-permanent status within the system; that is, the resource packages assigned to them (in particular, groups of ports) may be allocated for extended periods and the subsystems themselves would typically be generated by the system operator or the installation. These subsystems would include IBM-supplied operating systems such as OS with TSO. Free subsystems are created by user request and are hence transient; they normally will be destroyed when the user signs off. These subsystems will primarily comprise user-initiated and -tailored OS and other local operating systems, as well as all SL subsystems.

3.2.2.1 Resource Control

Each operating system may be given accessors to resource packages via the resource manager; this may occur at the time the operating system is added to the subsystem landlord, or dynamically when the operating system is invoked. In the former case, this permits an operating system to be sole accessor of a set of ports or lines; these resources are thus dedicated to that operating system until either the operating system is deleted or the resource manager, which still owns the resources, rescinds accessibility. In the case of dynamic allocation, ports and lines are accessible to the operating system on a request basis. In either case, resources such as disks, on-line or virtual printers and card-readers, and system libraries in resource packages accessible to operating systems are not shared by other systems; hence, management of these resources becomes the responsibility of the operating system control programs, via the subsystem resource managers.

A dedicated operating system may have the only accessor (as assigned by the system resource manager) to its system input; the libraries contained therein may be accessed by other systems provided the subsystem resource manager of the dedicated operating system permits an accessor to be established.

3.2.2.2 Subsystem Resource Manager

Each subsystem contains its own resource manager to control the allocation of resources from the resource package. Requests for additional resource allocation beyond that in the resource package are made through the subsystem managers, which communicate directly with the system resource manager. It is the responsibility of the subsystem managers to monitor resource requests and perform local deadlock prevention and control.

The capability in the system of nesting subsystems entails the recursive application of subsystem resource management. Each subsystem may permit a subsystem nested within it accessors to part or all of its resource package (as would be the case when running a free user-optioned OS under a free user-optioned CP); in that event, the nested subsystem's resource manager would control those resources directly. However, the higher-level subsystem may elect to maintain control over the resources, in which case lower-level subsystem resource managers would have to route resource requests through it (such might be the case when running an incremental PL/I compiler under a SL subsystem).

3.2.2.3 Control Program

The control program contains the routines necessary to service the system input. This includes inserting and deleting jobs, scheduling, priority assigning, interrupt handling, managing elements of the resource package(s), providing for job initiation/termination, and performing system maintenance. In terms of current operating systems, the control program refers to the job, task, and data management in OS/360; the task, data, and program management in TSS/360; and the virtual machine management in CP/67.

3.2.2.4 System Input

The operating system resource consists of the jobs which it is running or queueing, together with the appropriate libraries and other resources contained in the resource package allocated by the resource manager. The libraries contain language processors, maintenance and accounting files, language-associated subroutines, and any other semantic information required to define a job context. The job contains control information required by the operating system, along with program text or modules; in the case of the SL system, this information is all part of the program itself. Job integrity and security is provided via the allocation of time and space by the resource manager upon request by the operating system.

3.2.3 Job Environments

A job in the SL sense is a subsystem owned by the subsystem landlord.

3.2.3.1 Jobs in the SL Environment

SL jobs may be created by the initial interpreter at the user's request from a free port, or they may be created dynamically by existing SL subsystems. In the former case, a user entering the system through a free port informs the initial interpreter of his desire to run in SL mode. This initiates the creation of a SL subsystem by the initial interpreter via insert requests on the subsystem landlord. The resource package for the subsystem contains, minimally, an accessor to the free port plus accessors to any of the user's files; in requesting a SL subsystem, the user may specify additional resources to be allocated. Hence, the job is logically created at sign-on time and, if no subsystems with autonomous control are created in the interim, logically destroyed with sign-off.

Jobs may also be initiated from within a SL subsystem by insert requests on the subsystem landlord. The resources of the created job must initially be a subset of the resource package granted to the creator. If the two subsystems are then to run in parallel, only one may have access to the entry port, and resource control (for shared resources) must be arbitrated by the system resource manager. If they are to run nested, then either or both may retain access to the port; however, resource requests by the nested job's subsystem resource manager must be reflected back to the higher-level job. In both cases, mother-daughter jobs (as well as other autonomous SL subsystems) may communicate via shared data files or message transmission.

Each SL subsystem, once created, is free to make use of the entire SL language facility: subtasks may be created for serial or parallel execution, elements of the resource package utilized, etc. The SL user has the most freedom in employing the system for his problem solution, but does not have the capability of impugning the system's integrity or security.

3.2.3.2 Jobs in Non-SL Environments

Jobs under the control of foreign operating systems will maintain the identities they would have as if they were running under those systems and within those systems' host architectures.

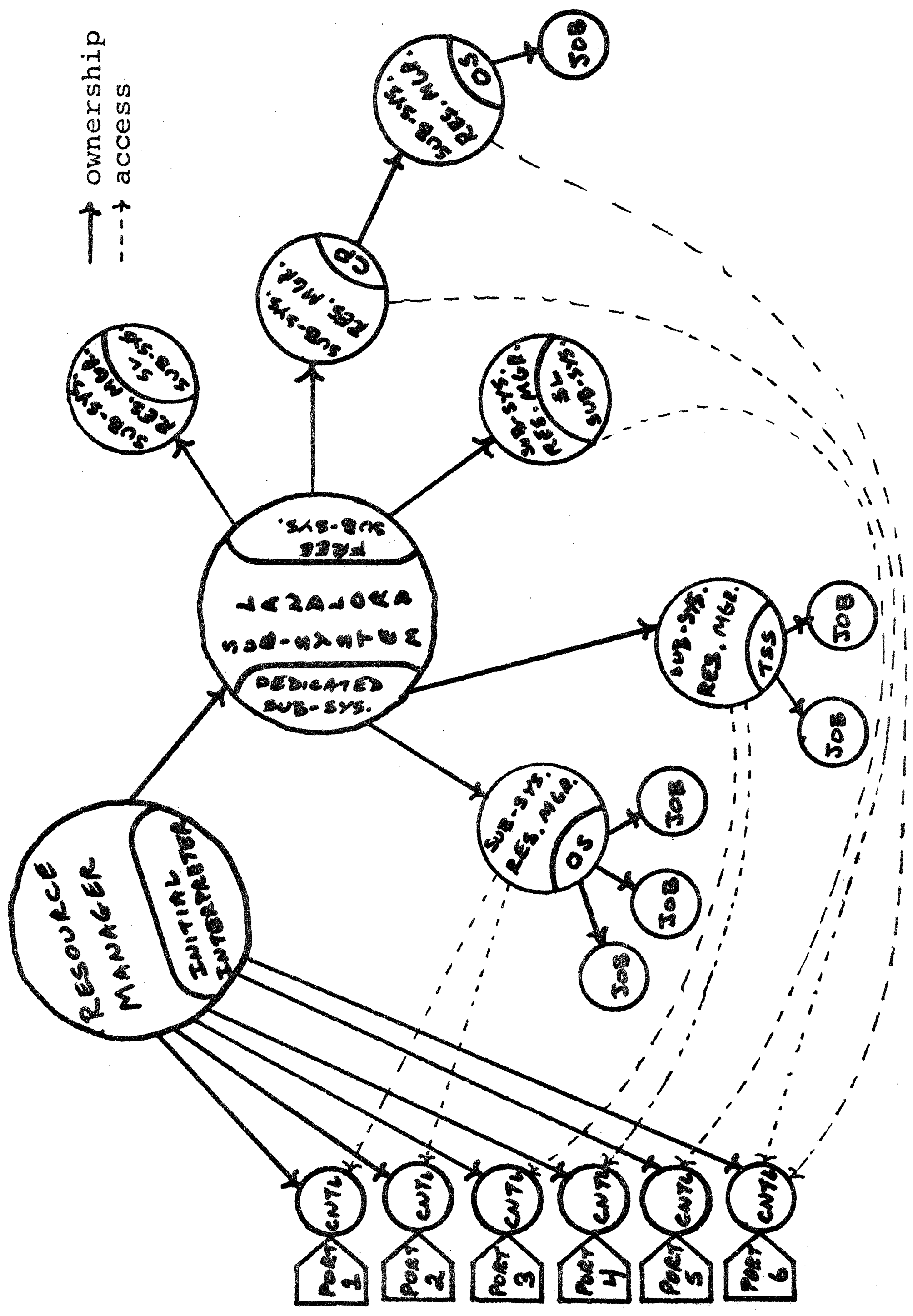
3.2.3.3 Resource Control

A SL job may utilize any resource available in its subsystem's resource package, or it may request the resource manager to create additional resources for it through the subsystem resource manager. However, requests for increased or modified resources by nonconversational jobs (i.e., those without a port in their resource package) will result in message transmission to the monitoring subsystem (or cancellation of the job if no higher-level job exists) if the request cannot be granted. Hence, SL libraries, workspaces, files, etc. fall directly under the control of the resource manager; this provides additional checking facilities for multiply-accessible file usage, as an example. The user's own libraries and workspaces may be made accessible to the subsystem either at the time the subsystem is created, or as the user requires them.

Resources for jobs under foreign operating systems are assigned to the operating systems in the manner previously stated (section 3.2.2.1). Whereas the accessing mechanism may be similar to that above, users of other operating systems can view resource availability in the manner to which they have become accustomed.

3.2.3.4 Example of System Configuration

The following illustration is a static, logical representation of a system with 6 ports: ports 1 and 2 are dedicated to an OS/370 subsystem running one background and two interactive jobs; ports 3 and 4 are dedicated to a TSS subsystem running two interactive jobs; port 5 was free, but has been assigned by the initial interpreter (upon user sign-on) to an SL subsystem; and port 6 was free, but has been assigned to a CP subsystem, which has in turn initiated a private OS/370 subsystem with access to the same port. In addition, there is an SL subsystem which is apparently running in the background after having been initiated by another SL subsystem no longer active:


ownership
access
RESOURCE MANAGER
INITIAL INTERPRETER
LANDLORD
SUB-SYSTEM
DEDICATED SUB-SYS.
FREE SUB-SYS.
SUB-SYS. RES. MGR.
SL SUB-SYS.
SUB-SYS. RES. MGR.
CP
SUB-SYS. RES. MGR.
OS
JOB
SUB-SYS. RES. MGR.
SL SUB-SYS.
SUB-SYS. RES. MGR.
TSS
JOB
JOB
SUB-SYS. RES. MGR.
OS
JOB
JOB
JOB
CNTL
CNTL
CNTL
CNTL
CNTL
CNTL
PORT 1
PORT 2
PORT 3
PORT 4
PORT 5
PORT 6

Chapter 3.3

SYSTEM CONTROL

3.3.0 INTRODUCTION

System Control is concerned with that set of data structures, processes and control mechanisms required to support and control the work flow operations in the system on two levels:

- Functional

- Server configuration

On the functional level, System Control is concerned with the initiation, coordination and termination of system functions in response to external (e.g., on-line user) stimulii in

- Data Communications
- Data Entry and Data Retrieval
- Data Computation and Manipulation

On the server configuration level, System Control is concerned with the control and synchronization of system work flow and the allocation/deallocation of resources.

Functionally speaking, System Control can be further partitioned to consist of System Command Control and System Monitor Control. System Command Control is concerned with the control and management of normal system operations involving system functions and resources in response to external (e.g.; User) stimulii. System Command Control operates in line with system work flow. On the other hand, System Monitor Control is responsible for the monitoring, detecting and handling of exceptional conditions occurring in the system. System Monitor Control operates in parallel with both the system work flow stream and System Command Control.

In the present report, emphasis is focused entirely on System Command Control, while capabilities characterizing System Monitor Control will be enumerated in a later section (3.4 System Functional Management).

Concepts fundamental to System Command Control will be presented from the following perspectives:

- The Faculty Concept. Here the system is viewed from the eyes of the system itself. The various areas of functional responsibilities are identified and grouped into autonomous functional partitions -- Faculties.

- The Work Flow concept. Here the system is viewed from the point of view of a user demand as the demand travels through the system. During this system walk-through, certain system functions are brought into focus as a matter of system overhead while others are invoked explicitly as a result of interpretation and execution by the system of the user's demand.

- Basic Control Structures and Mechanisms. Here the system is viewed also from the vantage point of the system itself. An extension of the key concepts (Part 2) in Object Base, ownership tree, and program structure resulted in the formulation of the basic structures and mechanisms for System Control.

3.3.1 A Functional System Structure

The functional system structure (Figure 3.3.1-1) is described in terms of its two major constituent components:

- The five functional Faculties

- The Queue mechanism for inter-Faculty interaction

3.3.1.1 Five Faculties

A partitioning of the total system functions into functional partitions based on areas of responsibilities resulted in a five-Faculty system structure . A summary description highlighting the roles of each Faculty is given below:

1. The Terminal Faculty

   A wide range of terminal capabilities are provided in a modular fashion, which can be configured by the user to provide him with a selective combination of terminal functions to meet his specific application, operational and service mode requirements.

2. The Data Communications Faculty

The Data Communications Faculty is concerned with the transportation of data into and out of the system. The Data Communications functions are:

- Transmission dependent
- Terminal dependent
- Message dependent

3. The monitor Control Faculty

This Faculty is responsible for the detection and handling of exceptional conditions occurring in the system. Also, it is responsible for system support and administrative support type of operations.

4. The Command Control Faculty

This Faculty is the central hub of control for the system. It is responsible for the initiation, coordination and termination of system services in response to user demands. Also, it is cognizant at all times of system work flow activities and system resource availability status.

Through a number of tables which Command Control maintains, it is cognizant of:

- The global working contexts about a user,
- The particular working contexts about a user during a specific instance of user/system interaction,
- System work activity status
- System resource status

5. The Data Control Faculty

The responsibilities of this faculty cover the management and control for all system resident data. Functions include:

- The accommodation of multiple logical structures
- Security control for private and shareable data
- Exclusive control for concurrent access of shared data

- Historical versions of data

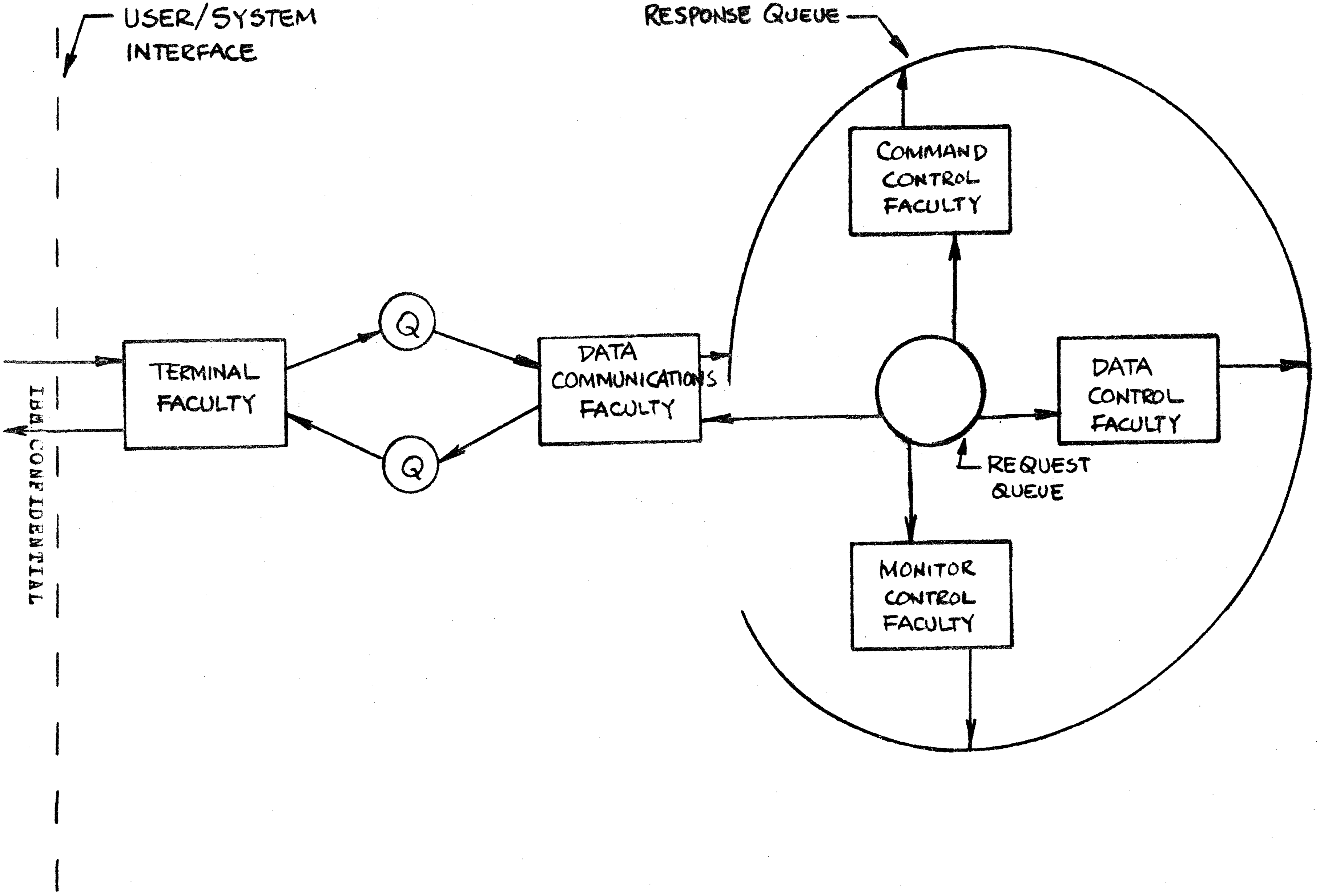


FIGURE 3.3.1-1 A FUNCTIONAL SYSTEM STRUCTURE

3.3.1.2 Queue Mechanism for Inter-Faculty Interaction

In response to an external stimulus at the user/system interface (Figure 3.3.1-1), one or more of the Faculties must collaborate to perform the necessary work. Inter-Faculty interactions are accomplished via the system Request and Response queues.

A unified message structure for interaction is employed by all Faculties. Pertinent information to be exchanged is assembled into a standard message structure. This information consists of:

- Identifications -- Requester and Responder ID's,
- Interaction types -- Request and Response types, and
- Parameter data.

Furthermore, on a conceptual level, once a Faculty is activated, it will perform the work as specified until a logical conclusion point is reached.

3.3.2 Work Flow

Once a functional system structure is postulated, the next step in bringing the role of System Control into focus is to scrutinize work flow activity through the system and identify which Faculties would come into play at which points in the work flow process. This is accomplished by a technique known as functional threading. This technique involves the tracing of external (user) demands through the sequences of system Faculty initiations, coordinations, and terminations. The object is to develop functional sequences of interacting Faculties in response to specific user stimulii. To be responsive to market requirements, the scenario for user stimulii must be developed based on user applications for the FS time frame.

An order hierarchy is required to specify the meaningful levels of control that must be established in the system. These levels of control should be defined on the Work Session, Job, and Faulty levels. System Control utilizes this control hierarchy to establish and maintain successive levels of context for control relative to the execution of a user work demand.

3.3.3 Basic Control Structure and Mechanisms

The Faculty and Work Flow perspectives address the roles of System Control from a gross point of view and present a picture of the system structure in terms of functional aggregates. This is an "Outside-in" approach--in that the system is described starting from the application level and ending up inwards at a functional partition level.

An entirely different approach to describing the roles of System Control is an "Inside-Out" approach. Here the emphasis is placed on a description of the structures and mechanisms which are basic to System Control.

3.3.3.1 A Multi-Server System Environment

From the point of view of System Control, work is performed by a combination of active and passive system elements. An active element is a system server (e.g., a program processing unit) that is capable of doing work. The passive element is the program module(s) which contains algorithm(s) indicating how the work is to be performed. This is an iterative definition in that a combination of active and passive elements may appear to be the active element to a second passive element,etc.

The external program structure for all programs (either system supervisor or user application programs) follows the standard PL/I static nesting structure (Figure 3.3.3-1). The external program structure for the system supervisory programs for the Faculty system structure concept (section 3.3.1.1) is shown in Figure 3.3.3-2. Logically, the supervisor control program can be thought of as a single procedure which in turn consists of five basic procedures.

Similarly, the internal program structure also assumes a uniform structure. The properties of this structure are:

1. All programs are re-entrant.

2. One or more program modules make up a program.

3. A program module consists of two components:

    - A program text component

    - A symbol dictionary component

The active system elements which are capable of doing work operate in a multi-processing environment.

Important concepts for System Control in this area include:

1. Multiprocessing is the normal mode of system operation.

2. Server pool concept-- System servers are organized into pools of resource by type. All server pools are interconnected to one another through an interaction network(Figure 3.3.3-4).

3. The concept of floating supervisor control-- No master/slave relationship exists among server elements in the server pool. The supervisory control program which is executed by every available server is considered to be the master.

4. All server elements have identical processing capabilities and are equally qualified of performing work (either supervisory control or user application work). No server is vested with any special processing roles.

5. A queue-driven system concept-- server's interface for work assignment is via work queues. Requests for work are always enqueued onto the appropriate work queues.

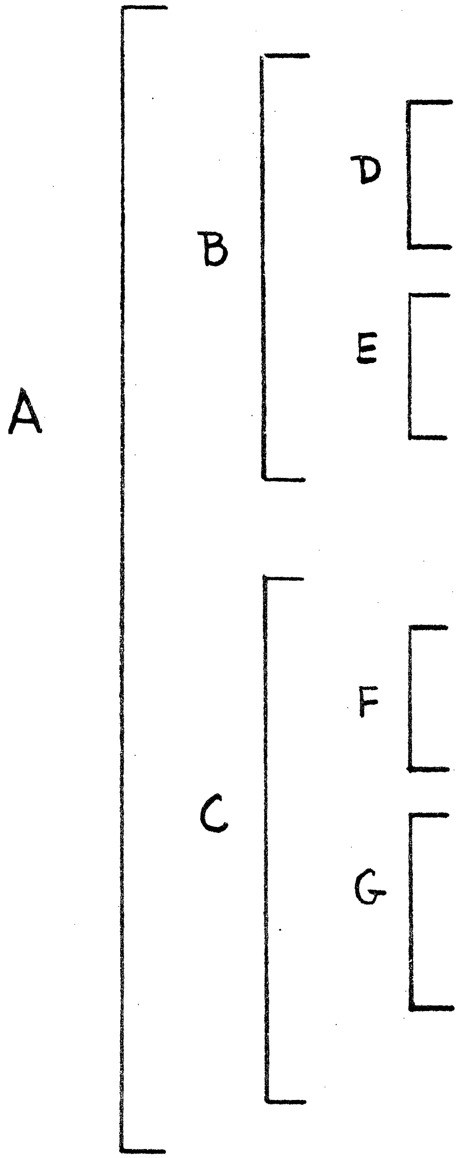


FIGURE 3·3·3-1 A PL/I PROGRAM STRUCTURE

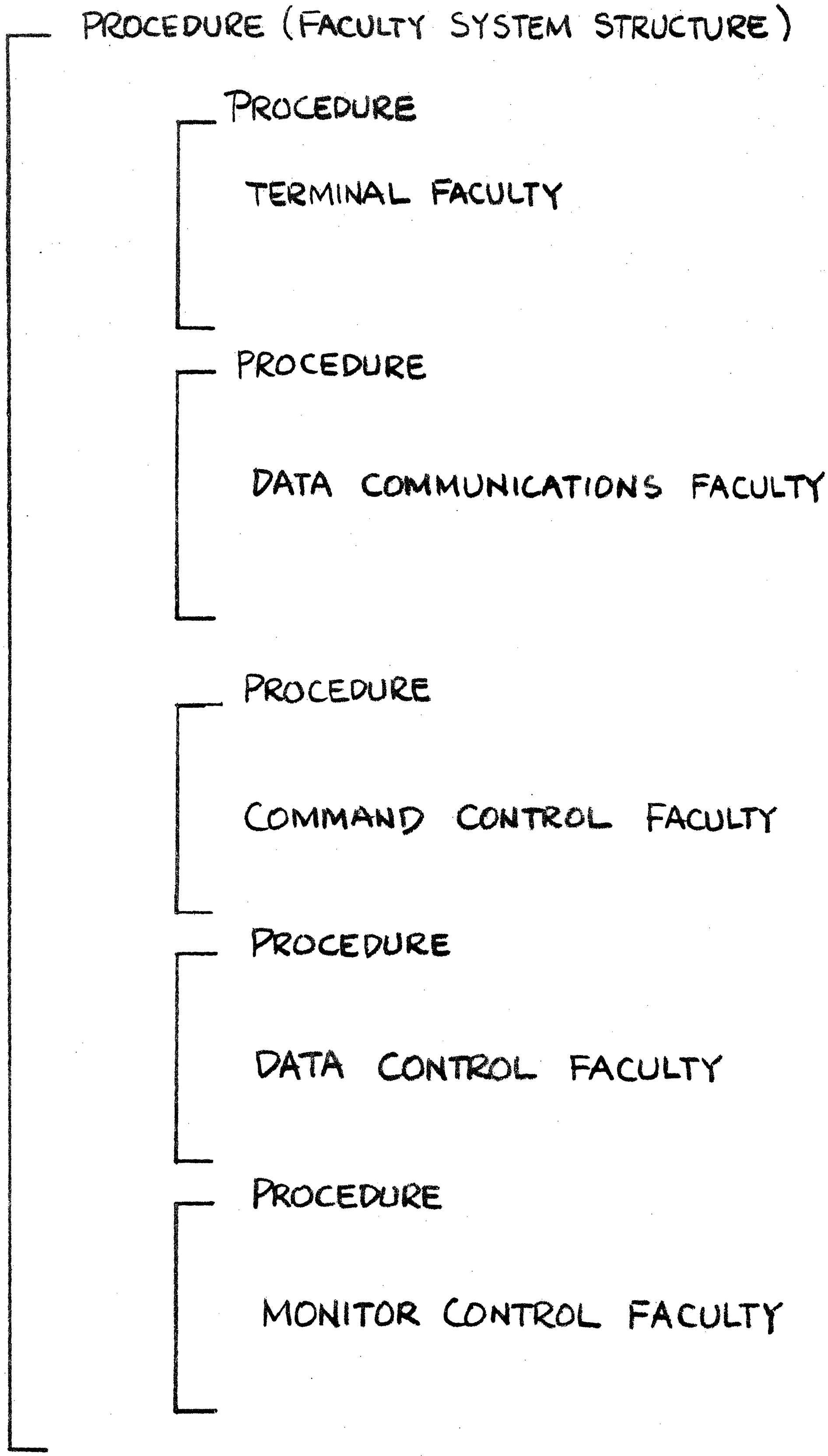


FIGURE 3.3.3-2: PROGRAM STRUCTURE FOR THE FACULTY SYSTEM STRUCTURE

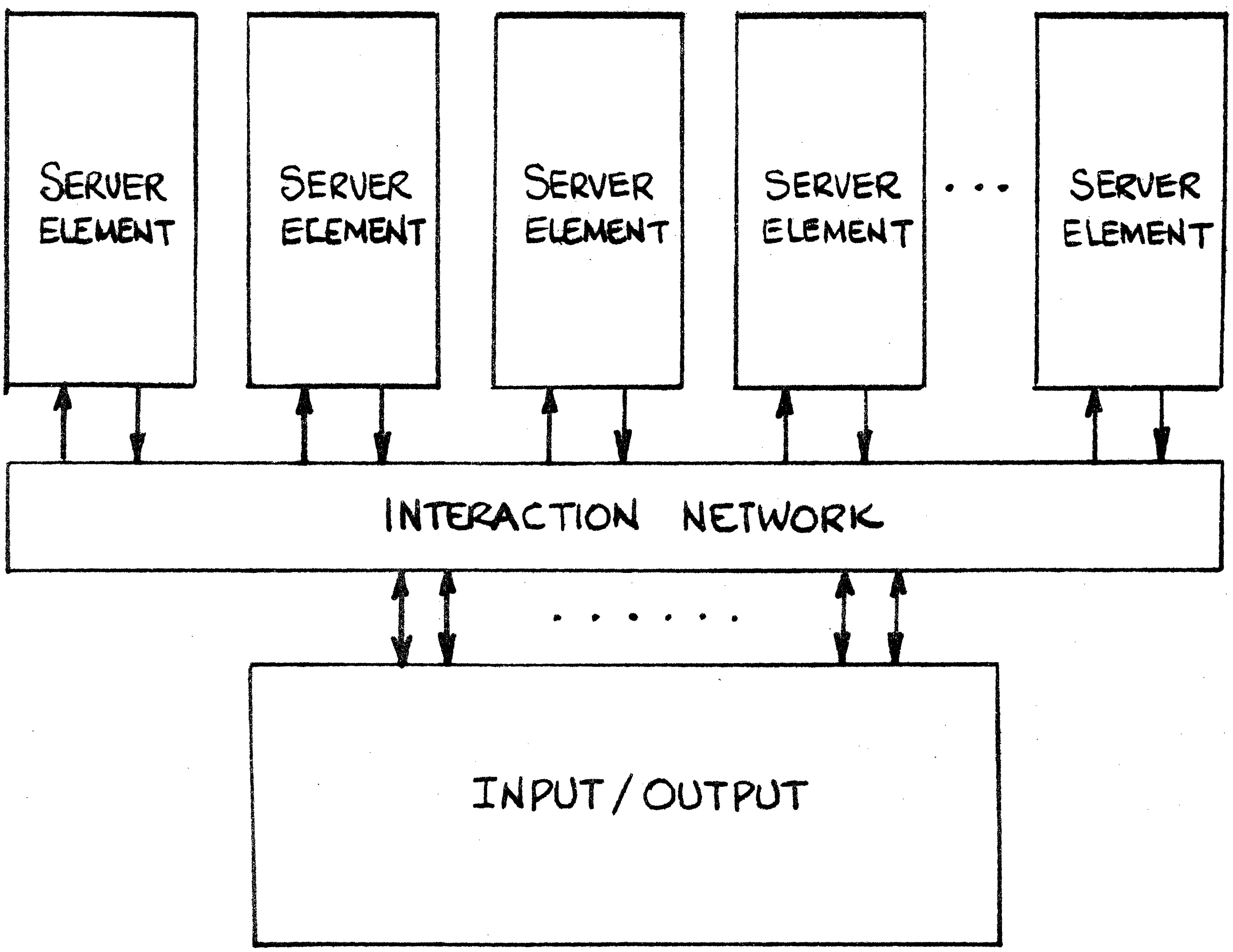


FIGURE 3.3.3-4 : A MULTI-SERVER CONFIGURATION

3.3.3.2 System Interactions

System Interactions are required between active elements to accomplish control and management of system functions and resources to be responsive to an external stimulus. System interactions take place due to:

1) Problem interaction: These relate to logical dependencies within a program. Synchronization between concurrently executing instruction streams is required.

2) Supervisory interaction: The supervisory interaction is concerned mainly with the allocation of server resources and with the job of dynamically tuning the system.

3) System interaction: Active system elements interact with one another to verify the validity of system control data, to dynamically reconfigure the system due to load balancing or malfunctions, etc.

Of these interactions, System Control's involvement in the supervisory function of task assignment and server element selection will be described in some detail to furnish some insight into the problem.

The control algorithm on task assignment and server element selection is based on the concept that all system resources are executing the most important tasks as determined by the environment. In the system (Figure 3.3.3-4), as a server completes execution of the work specified by a task (a unit of work specification),it executes the task assignment algorithm of the supervisory program and dequeues a new task from an appropriate work queue. Thus, tasks must be assigned to server elements so that work can be performed, and server elements must be selected from a pool of server elements to take on the tasks. The role of interaction network is to facilitate in this assignment and selection process in order that an optimum system operational environment is establisned and maintained.

Tasks include both supervisory and user tasks. New tasks are generated due to new job introductions, task splittings, or I/O interrupts. All tasks are assigned priority numbers. Similarly, an "availability index" is associated with each server element which is executing a task. The "availability index" is derived directly from the priority of the task which is being executed by the server element. The "availability index" is a measure used to determine the relative degree of a server element's readiness to take on a new task. An idle server element has the lowest "availability index" (i.e.;most ready to take on a new task).

When a new task is being introduced into the system, it is assigned a priority number and is enqueued onto the appropriate work queue. An idle server element is selected to take on the task. In the event no idle server elements are available, an active server element must be selected to take on the task. To make the selection, a comparison is made between the priority of the ready task and the availability index of each and every active server element. Those with lower availability indices are all available. The one server element with the lowest availability index, however, is deemed to be most eligible and will be selected to take on the task. The lower priority active task which was being executed prior to the selection will go into a dormant state and will be enqueued onto the work queue. Should there be more than one eligible server element with identical availability indices (i.e.; all are executing tasks with equal priority numbers), a tie-breaking algorithm will have to be executed.

Chapter 3.4

SYSTEM FUNCTIONAL MANAGEMENT

A description of the important concepts from some of the key system functional areas is presented in this chapter to give an indication of the directions being followed.

3.4.1 Data Base Management

Data base management is concerned with the accessing of data by multiple terminal users from an on-line centralized data base. A terminal user's access to the data base may be for the purpose of:

- Read-only data entry and retrieval,and
- Read/Write data entry and retrieval:
  - Data insertion
  - Data modification
  - Data deletion

Accessing takes place in a concurrent and independent manner in either the transaction processing mode (routinized processing) or interaction mode (non-routinized processing).

Data base functions to be addressed for the AFS logical architecture must be responsive to these types of user requirements. Accordingly, topics to be addressed in this section touch upon all of the following:

- Data Independence
- On-line availability
- Convenient data entry and retrieval
- Multiple user data structures
- Symbolic data access
- Authorization to private and shared data
- Exclusive control to concurrently shareable data
- Data Base recovery
- Historical versions of data
- Transaction audit trail

3.4.1.1 Data Base Management: An Overview

A logical representation of the major data base components and interfaces is given in Figure 3.4.1.1-1. User activities in data entry/ retrieval, data manipulation and data base maintenance are presented to the system as application and system programs. The procedural specifications of the programs are defined independently of the data descriptions. Functional capabilities in each area are made available to the users via the Data Manipulation and the Data Description languages. Definition of multiple logical data structures on the same system resident data is allowed to accommodate the many views which independent users may elect to see data. The entity record set concept in terms of entity attribute description of external things is used as the vehicle for logical data structure representation. All data accesses are subject to system data exclusive control which is responsible to act as a filtering function to resolve the contention problem due to concurrently shareable data requests. Data base address space is a multi- linear symbolic address space. In addition, data recovery constitutes an integral part of the total data base management function.

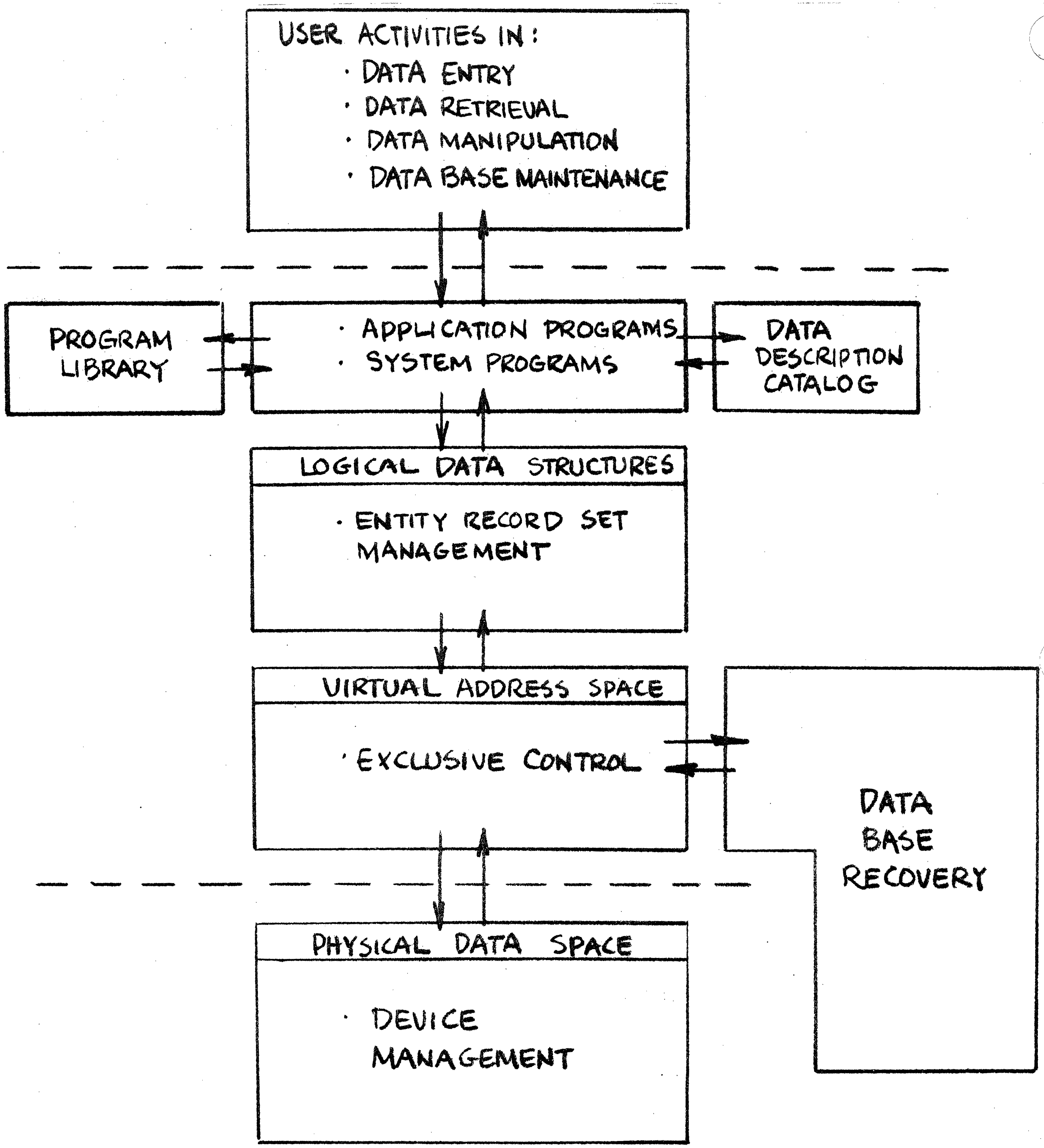


FIGURE 3.4.1.1-1: A LOGICAL REPRESENTATION OF DATA BASE COMPONENTS & INTERFACES

3.4.1.2 Data Base Language Capabilities

The Data Description Language (DDL) is the language used to define an Entity Record Set. An Entity is a person, place, or thing. The things may be real or abstract. An entity is that about which a user wishes to record information in the data base. An entity record set is a collection of similar entities (Figure 3.4.1.2-1). To completely describe an entity record set, it is necessary that:

The attributes which describe the entity are described.

The entity records making up the entity record set are described.

The data names and their synonyms are described.

In addition, the DDL can be used to describe the physical characteristics of data in the data base. However, these capabilities are available only to the system installation manager.

The Data Manipulation Language (DML) is the language which enables the user to manipulate the logical data in his application program. Data manipulation implies data entry and retrieval as well as computation and processing. Both of these capabilities will be supported in TSL as operators. Since a user may wish to converse using any of the five languages: PL/I, FORTRAN, COBOL, RPG, and APL, DML must rely on a host language to provide the computational capabilities. DML ,in turn, will provide the language interface between the program and the data base. Therefore, all calls to and from the data base to retrieve data, to enter data, to modify data, or to delete data are invoked via DML operators.

3.4.1.3 The Entity Record Set

An entity record set is a two dimensional array representation of data structures in terms of Entities and Attributes (Figure3.4.1.2-1). An entity is a person, place, or thing. Attributes are the property classes which characterize an entity. An entity record set is a collection of similar entities. Also, associated with each entity record set is an attribute whose values have a one-to-one relationship with the entities (i.e.; the unique identifiers). Thus, an entity record is that collection of attribute values which describe an entity.

Attributes for an employee entity record set are:

1. Unique Identifier
2. Employee name
3. Social security
4. Sex
5. Birthdate
6. Data of hire
7. Department assignment
8. Division assignment
9. Education record
10. Marital status as of date
11. Position as of date
12. Perf. Evaluation as of date
13. Salary as of date

Attribute type 1 is the unique identifier attribute. Attribute types 2 through 13 are facts about the employee entity. A fact is a relation - a correspondence between members from two sets. Attribute types 2 through 6 establish a one-to-one relationship between a member from the employee entity set and the respective attributes. For instance, there can be only one "date of hire" attribute value for an employee. On the other hand, however, attribute types 7 through 13 establish more complex relationships. No one-to-one relationships exist. Furthermore, each of the relationships can be qualified by a time parameter. Thus, an employee can be assigned to work in more than one department as of a certain date.

The internal system organization of the data for an entity record set must be such that it is responsive to the many ways a user may elect to view the data. One way to express an entity record set is in terms of a collection of relation sets (Figure 3.4.1.3-1). A relation set is an entity record set which has only a pair of attributes - - one of these being the identity attribute. Thus,
data required for the employee entity record set can be materialized from the twelve relation sets. Note that the identity attribute has been replicated twelve times to provide the connectivity required to link together the pertinent relation sets.

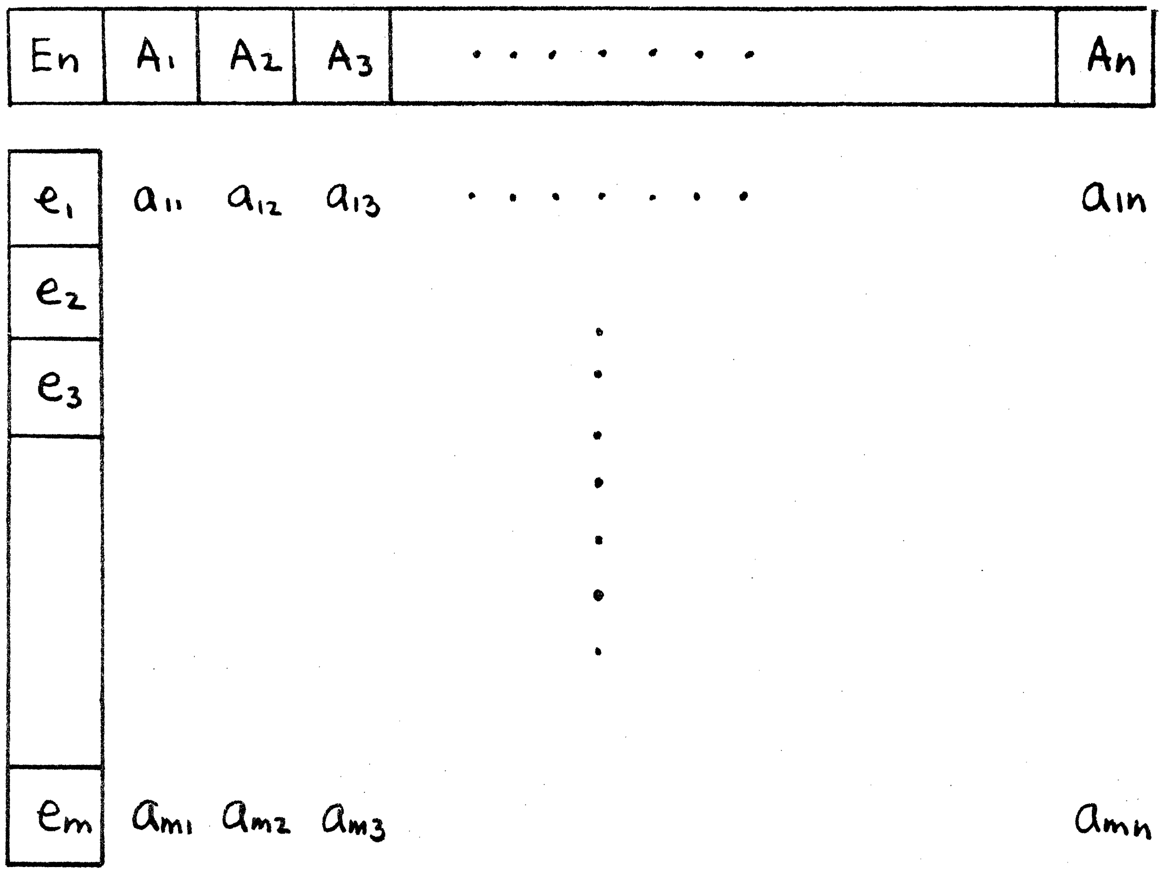
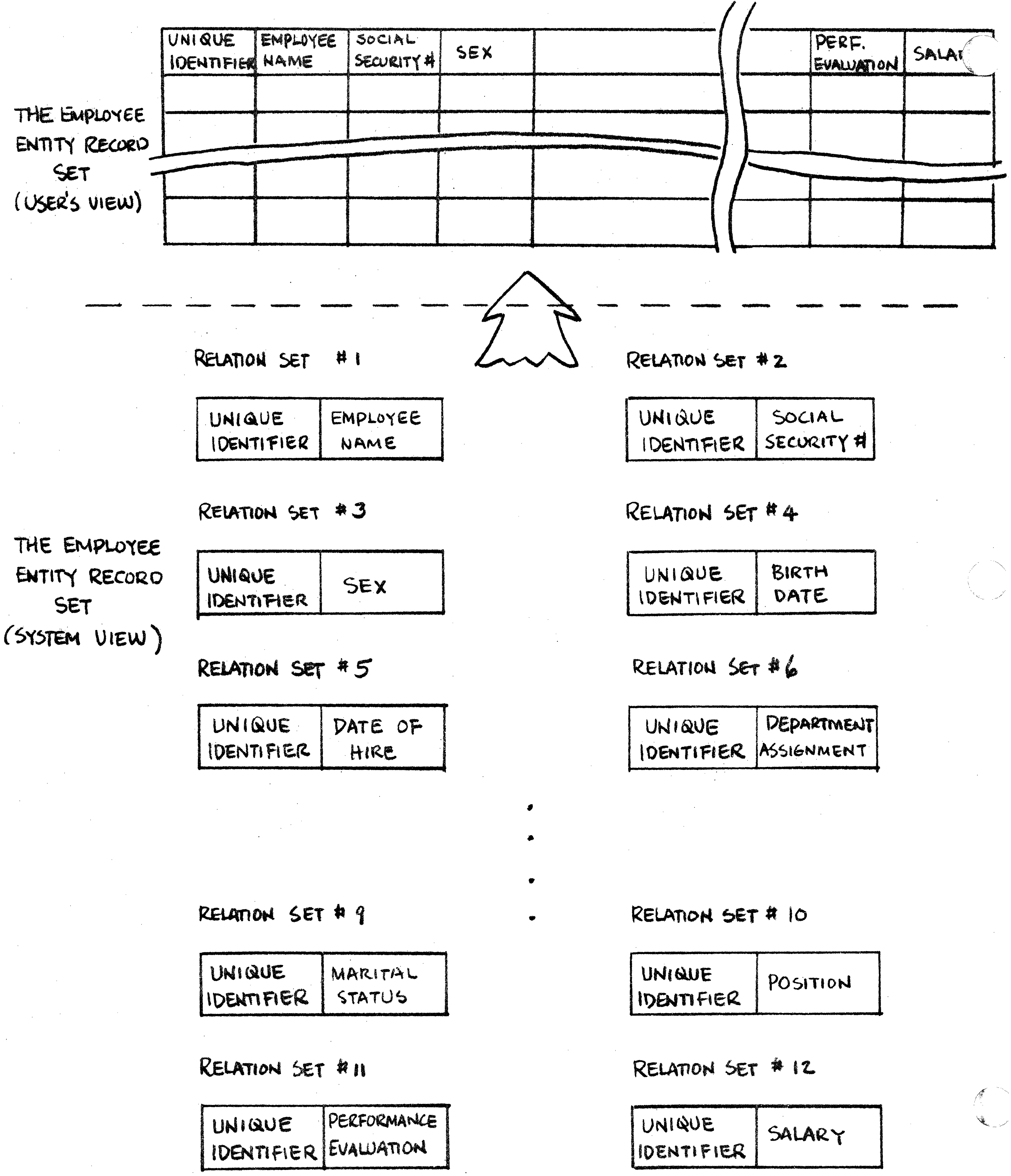


FIGURE 3.4.1.2-1: ILLUSTRATION OF AN ENTITY RECORD SET

FIGURE 3.4.1.3.1: USER & SYSTEM VIEWS OF AN ENTITY RECORD SET



3.4.1.4 Data Integrity

Data integrity is addressed based on the way system exercises exclusive control to resolve contention, modification and update problems due to independent concurrent data accesses on data from a centralized on-line data base. Also, data integrity is concerned with the way data recovery is handled to checkpoint pertinent data base information so that operations may be restarted in case of a catastrophic data base failure.

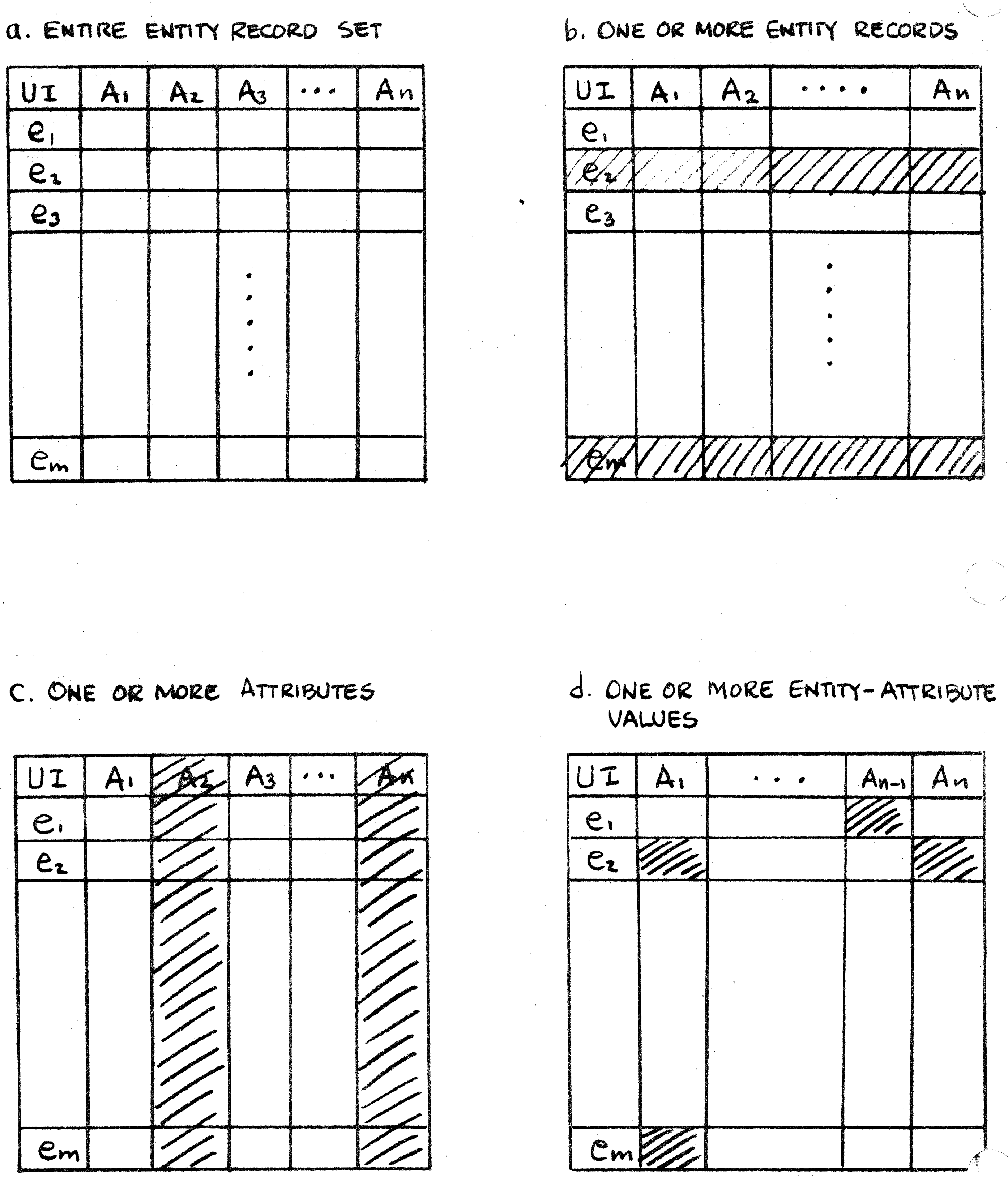
3.4.1.4.1 Exclusive Control

All system resident data can be classified as:

Read Only	Private
Read/Write	Shareable

All possible ways in which a user may choose to access data on an entity record set are shown in Figure 3.4.1.4-1. However, when two or more independent users are making simultaneous data access requests on the same entity record set, exclusive control must be exercised. Parameters which the system must consider in performing the exclusive control function are determined by the type of entity record set involved (i.e.; Read/Write or Read-Only), and by the type of data access requests(i.e.;Read-only or R/W).

POSSIBLE DATA ACCESS REQUEST TYPES:



a. ENTIRE ENTITY RECORD SET

UI	A_1	A_2	A_3	...	A_n
e_1					
e_2					
e_3					
			⋮		
e_m					

b. ONE OR MORE ENTITY RECORDS

UI	A_1	A_2	....	A_n
e_1				
▨ e_2	▨	▨	▨	▨
e_3				
			⋮	
▨ e_m	▨	▨	▨	▨

c. ONE OR MORE ATTRIBUTES

UI	A_1	▨ A_2	A_3	...	▨ A_n
e_1		▨			▨
e_2		▨			▨
		▨			▨
e_m		▨			▨

d. ONE OR MORE ENTITY-ATTRIBUTE VALUES

UI	A_1	...	A_{n-1}	A_n
e_1			▨	
e_2	▨			▨
e_m	▨			

FIGURE 3.4.1.4-1: CONDITIONS SUBJECT TO EXCLUSIVE CONTROL

3.4.1.4.2 Data Base Recovery

The second aspect of the data integrity problem will be addressed by focusing attention to the functional organization of a specific mechanism for data base recovery -- a Journal organization. System roles which can be fulfilled by a Journal include:

1. Data Base Recovery-- All data access requests which cause data modifications to take place will cause the modifications to be reflected in both the on-line data base and the Journal.

2. Historical Versions of data -- Data in the Journal is checkpointed periodically to give a snapshot in time of the data base content.

3. Transaction audit trail--All data modificiations must be captured in the Journal in the exact manner as the modifications are made.

A Journal organization which fulfills these basic principles is given in Figure 3.4.1.4-2. There are two types of Journal Records: the Checkpoint Journal records (Journal Records 5 and 41); and, Transaction Journal records (Journal Records 6, 14, 16, 18, and 36). Also, there are three types of Journal threads: the Daturn thread, the Transaction thread, and the Attribute Transaction thread.

The Transaction Journal record is created in the Journal whenever a transaction takes place. The Checkpoint Journal record, on the other hand, is a system assembled Journal record which reflects the status of data as of the time the record is created.

The threads are the mechanism by which to connect together all those Journal records which are generated within particular contexts.

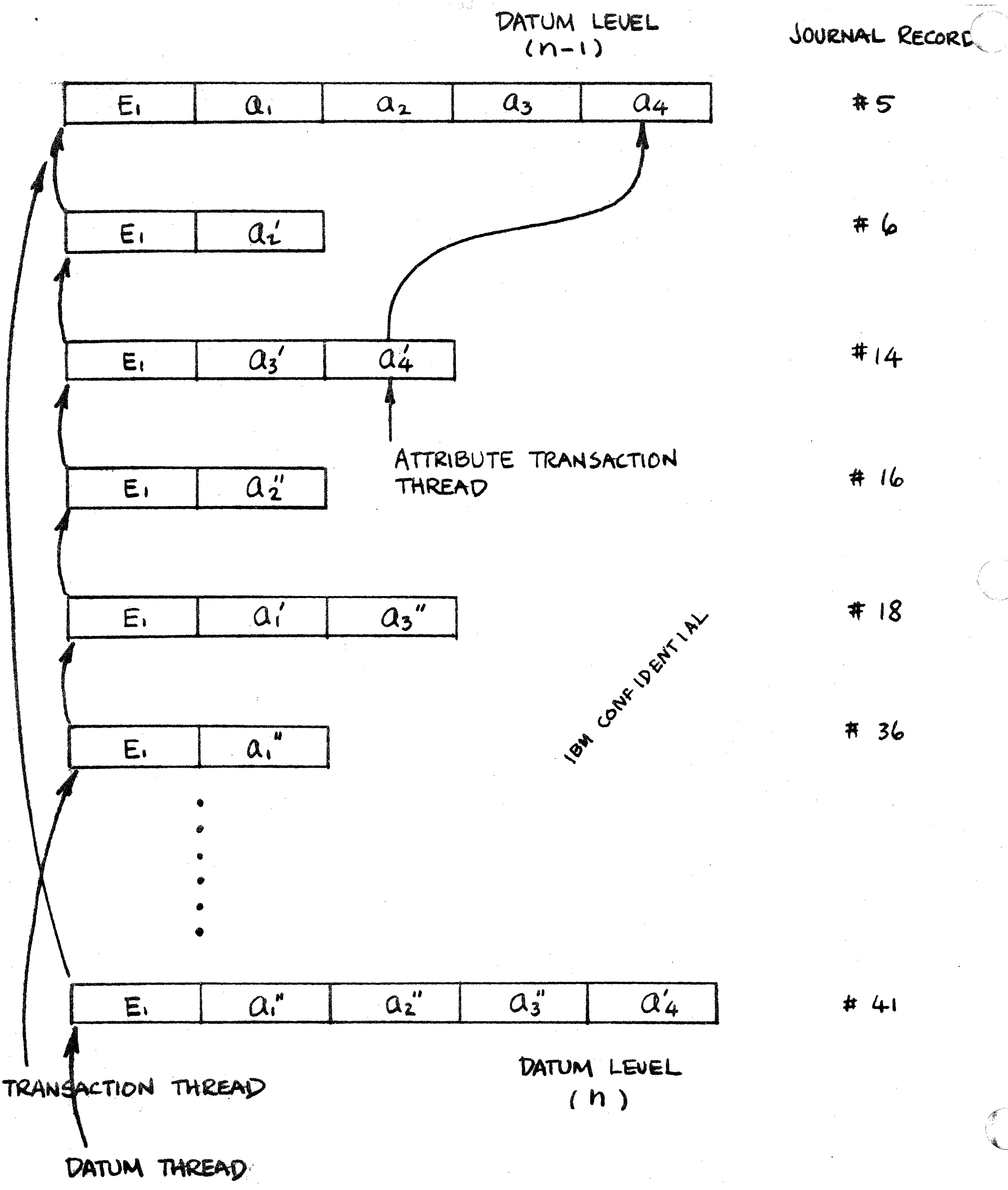


FIGURE 3·4·1·4-2 SCHEMATIC OF A FUNCTIONAL JOURNAL

3.4.2 Data Communications

3.4.2.1 Background

The data communications area, in comparison to the other major functional areas of a data processing system, is in an earlier stage of evolution. As a result special attention is required to architect a system structure that provides this area with the flexibility to properly evolve during the product life of AFS without compromising the other areas of the system and the overall system structure itself.

The basic tenets of AFS postulates, with good technical justification, the availability of Storage Management Units (SMUs) with the capability of providing viable random access to an essentially infinite logical addressing space opaque to the individual performance /capacity characteristics of the various storage devices in a SMU. They further postulates the availability of Program Processor Units (PPUs) with a functional capability of providing a high-level System Language (SL) interface, described in Part 2 in this document, and be opaque to the number and individual performance characteristics of the various program processors in a PPU. In some sense, these could be thought of as "ultimate" interfaces to these units - or at least ones at a very advanced conceptual level.

A comparible level is not anticipated for the data communications area at the time AFS is introducted, however, enough is known to allow an architectural structure to be developed which can be evolved with low impact to the system and neglible impact to application programmers.

What then are the characteristics of the data communications area that contrast its architectural status to that of the SMU and the PPU?

An (operating) system essentially simultaneously services many users typically at a centralized facility. On the other hand, data communications must in general deal with hardware devices that interface with a set of individual users at distributed locations. The former allows for highly functional interface levels and short well- controlled internal IBM data paths. The latter typically necessitates low costs at the device and needs to append functions of a system in a time-sharing manner in order to provide the desired user interface. In addition the long data paths, generally external to IBM products (telephone lines, etc.), create significant additional problems in themselves.

Since the advent of LSI will allow for expanded device function, the increase in the data communications market will bring about dramatic changes in the technology and pricing of communication paths, and the requirement will grow for more application program independence of device characteristics, a system architectural framework must be established which is both flexible to such changes yet provides guidelines to allow them to properly evolve.

Further complicating this area are the market requirements to allow present systems and devices to co-exist in an emulated (virtual) mode under AFS; to provide a means for dynamic interchange with those systems (as well as ones in separate installations) and devices; and to allow for most hardware devices to transcend the introduction of AFS. Essential to achieving a smoother transistion from both an internal IBM programming and engineering viewpoint as well as an external customer viewpoint will be an early common and coordinated recognition of FS goals, and tradeoffs within all present products during the interim toward those goals. This is discussed in more detail in Chapter 3.5

3.4.2.2 Basic Concepts

A set of basic concepts have been identified for establishing long-range criteria for data communication tradeoffs:

- All physical I/O (external) to source-sink devices and other systems will be handled by the communication unit (CU) of the installation. This includes unit record and sensor devices as well as typical communications terminals.

- Logical I/O, i.e. as seen from an application program and most of the AFS control program, will have virtual/local/remote transparency. This includes any dynamic interchange with other virtual systems. Physical I/O, i.e. as viewed from the Source/Sink (S/S) subsystem of the control program, will have local/remote transparency.

- The SL and hence HLL (Higher-Level Languages) and other PP interfaces to application programs will be by means of a minimum set of device classes. The FDL (Field Descriptor Languages) for pre-formating data structures on complex devices such as graphics will both simplify application programs and increase their degree of device independence.

- Both the terminal user and the application programmer have functional interfaces - independent of their locations or path connecting them. The logic to accomplish these functions from either end should look like it is satisfied either by the other or the terminal device in between them. By terminal here, is meant either a single terminal on a cluster or common terminals with a central controller (compound terminal). Cost tradeoffs have dictated, and are expected to continue to dictate, that improved cost/performance can be achieved if some of this apparent terminal device logic is implemented in the AFS control program. This logic has two parts: one is the formating field descriptors mentioned earlier which must be specified by the customer, and the other is simply good hardware/software tradeoffs. It is important to keep these logically separate from the path functions required to connect the terminal with the system. These path functions are to be performed in the CU and the other network management units between the terminal and the system.

- A general AFS control program queuing mechanism for passing work to be done between processes will allow resource management to tune the system for a range of response requirements. The interface to the CU will be a consistent extension of this queuing mechanism.

3.4.2.3 Types

Data communications with the system need to be examined at the logical I/O interface and of the physical I/O interface. Because of the basic AFS concepts of distributed (network) data and programs as well as virtual devices and systems, there is generally not a 1:1 relationship between these two interfaces.

At both interfaces, however, information is considered to be consummable, i.e. once sent, it can not be obtained again, and once received, it cannot be requested again.

3.4.2.3.1 Logical I/O

Logical I/O is defined to be explicit operations made by a program to communicate outside the logical closed entity or environment containing its known authorized data, programs, and system services. Such communications are called messages if they represent original information being sent or received and answers if they are requesting a response to a previously sent message.

Inter-AFS Jobs - These messages provide for interchange between normally independent AFS environments that want to establish local communication paths. Full supporting system services will include dynamic establishment and validation of authority and ability for controlled sharing of data and programs.

Source/Sink - These messages provide for interchange with areas outside AFS. These areas are either devices or other operating systems (networks). By means of a well structured data communications path, the SL interface to these areas will be made almost completely free of device dependencies and absolutely free of physical path dependencies.

The general formats for logical I/O functions are as follows:

introduce (arg1; ....; argn) ---> name

- This provides a means for naming a source/sink port object having the characteristics defined by the argument list. A name may represent a collective object thus allowing for broadcasting to all elements of that object.

send-message (name(arg, .. etc.); msg) ---> msgid

- This sends a data object, msg, to the object called name some of whose characteristics may be temporarily modified by the argument list.

- The msgid is returned by the system to allow for subsequent reference to this message if an answer is later desired or an error condition results.

wait-message (name (arg, ..etc.)) ---> (msg;msgid)

- This allows the program to specifically wait for a message from the source object, name.

- Again the msgid returned allows for a subsequent answer to be returned.

send-answer (msgid) ---> answer

- Requests an answer to the message previously identified by msgid.

wait-answer (msgid) ---> (answer;msgid)

- Only one of the msgid arguments is to be used. If the left or input argument is non-void, the process plans to wait until only that message is answered. If the left argument is void, then the system will return the first answer it receives and identify it with the msgid specified by the system earlier when the message was sent.

3.4.2.3.2 Physical I/O

Physical I/O is defined to be the data communication interface between the system and the CU which in turn interfaces with the real devices or other operating systems known to the AFS system. Whatever the source of a message, its format at this stage is in BDUs (Basic Device Units). Each message (or answer) can be represented by a set of fixed length BDUs with embedded sequencing information. Functionally they contain a device name, priority information, and a string of bits which is logically opaque to the CU. Correspondingly the system is unaware of the external location of the device/system or the path(s) to them.

The BDUs reside on queues of port objects in the logical address space of the SMU. These represent a consistent extension to the normal queuing mechanism for passing information between objects for processing purposes.

3.4.2.4 Architectural Considerations

The purpose of data communications is to send and receive consummable messages between two or more devices, systems, or application programs. These messages may be explicitly initiated by a device/another system or application program or they may be implicitly initiated within the AFS control program to provide network transparency.

Explicit messages are essentially those between users either at devices or as a result of writing application programs. Another system can be thought of as just another type of device. Two things can effect a message: its path and the functions performed in between the sender and the receiver. It is the responsibility of AFS to make the path virtual/local/remote transparent. The functions are dependent on the characteristics of the sender and receiver. For example, if both are just AFS application programs (inter-AFS job communications) then the functions in between are essentially zero, i.e. just normal expression processing. On the other hand if one of the end points is a graphics device then there are considerable functions required to translate the data to an application program from the grid of the tube and perhaps its light pen. While logically these functions appear to be done in the device, cost/performance reasons may require that some of these functions be done in the AFS control program.

In order to make the path of the message transparent, the system must handle various situations depending upon whether one end point relative to the other is in a native AFS job, in a virtual device or operating system, or whether it is locally or remotely attached. The first two situations are handled within the AFS control program. The last two are physical I/O and are handled transparently to the control program by the CU.

The following sub-sections translate these message function and path aspects into the services performed by the major areas of AFS.

Before proceeding it should be stated that AFS must be flexible enough to dynamically add/delete devices, and its associated CU to correspondingly be able to make on-line changes in device types and the paths to them.

3.4.2.4.1 System

The standard system functions for messages are those provided for all expression evaluations. These are such things as name resolution, attribute examination, and validation of authority.

In addition, a unique system message identifier (msgid) is created for each new message. It is retained by the system only while it has responsibility for the message and forgotten after delivery of the message to either an application program or a port object queue which interfaces with the CU or a virtual device/system.

Standard inter-AFS job communications within the same system are independent of the Source/Sink (S/S) subsystem. In the case where network processing is required, the cooresponding subsystem desiring the information interfaces with the S/S subsystem is the same way (except for different authorization) as an application program.

Users of the S/S subsystem are unaware of whether the device or other system is co-existing in the same AFS system or not, and if not, whether it is local or remote.

3.4.2.4.2 Source/Sink Subsystem

This subsystem provides a uniform interface to all communications outside its system. Its responsibilities are to process the data so that it is in a suitable logical form for the eventual receiver - device, operating system or application program. In the case of a device, it may mean special format processing and/or internally cost/performance implementation of device functions. In the case of an operating system, it means the protocol for communications - which by the way should be trivial if it is to another AFS system. In any case, its internal system interface is in the form of BDUs (Basic Device Units) which are the logical interface to any particular device in question.

At this point far down the processing path, the S/S subsystem finally resolves the question of virtual attachment. Its answer simply determines the port object queue the BDUs are to go on or come from.

The BDUs fundamentally only have a device/system name, a priority to aid algorithmic scheduling, and a string of bits represent the data information. In addition, they will probably have a fixed block format thus requiring some additional imbedded sequence number. The bits of data information will be logically opaque to the CU.

The message queues for the port objects will be located in the logical address space of the SMU, and the mechanism and interlocks with the CU will be essentially identical to that between other objects in the system. One difference, however, is that since the information is being moved out of the SMU logical address space, the cell name for that will no longer be a suitable means of identification and, if going to another system, will have to be replaced by a prescribed network symbolic name.

3.4.2.4.3 Communication Unit

The CU is the interface of the system to the physical communication network. Its responsibility is to get BDUs to and from devices/other systems for its own system.

To do so it must know what devices are connected, the paths (lines) to them, and the protocol for those paths. In addition, it must determine the optimal transmission block size, termed BTU-Basic Transmission Unit.

Opaque to the contents in the BDUs, it may employ various compaction algorithms in conjunction with associated communication unit facilities on the network if it can improve cost/performance.

Like the PPU, the CU may be a multiprocessor. Furthermore, it may be connected to more than one system and conversely a system may be associated with more than one CU. In the latter case, an additional small amount of physical network awareness may get back into the system design in order that it may have to decide what device goes with what CU.

3.4.3 System Monitor Control

System Monitor Control is responsible to monitor all system operations and to cause recovery actions to begin in the event of system failure(s). In addition, Administrative Control (e.g.; statistics collection and customer billing, etc.) and System Support Control (e.g.; dynamic system reconfiguration control, etc.) constitute important system roles of System Monitor Control.

The following is an enumeration of the Monitor Control categories:

3.4.3.1 User Orientation

-Control of terminal user activities

-Assignment of terminal user priorities

-Degree of user/system interaction

-System Operator and Data Base Administrator support

-User Integrity

-System service support

-Billing
-Performance analysis
-Verification of proper system operations

-Assigning passwords

3.4.3.2 System Configuration Control

-Startup and Shutdown of system

-Set Priorities

-Dynamically change priorities

-Provide warning alarms on exceptional operating conditions

-Line or specific terminal load exceeds pre-assigned maximum load

-Terminal outage

-Low priority messages are not being processed

-Data access requests are not being honored

-Unusual number of accesses to data base

-Erroneous password

-System monitoring support on specific system components

-Server allocation

-Gather and output user statistics

-Terminal load by time of day

-Line load by time of day

-Errors by line and terminal

-Number of message by type by time of day

-Response time by message type

-Response time by time of day

-Processing time by message type

-Data access, deletion and insertion statistics on data base

3.4.3.3 Checkpoint Restart

-Automatic checkpointing of the entire system based on a pre-defined criteria.

-Checkpointing initiated explicitly by the System Administrator.

-Requested selective checkpointing on specific Entity Record Sets initiated explicitly by the System Administrator. All active and/or pending processing requests involving the Entity Record Sets should also be checkpointed.

-Restart capability (warm start) which involves the rest to initial state the Entity Record Sets updated and the reconstruction

-A Restart capability (cold start) after a catastrophic failure.

3.4.3.4 Terminal Network

-Enabling a line or terminal

-Disabling a line or terminal

-Selective termination of message handling

-Selective termination of message processing programs

-Transmission control

-Path Control

-Message Delivery Control

-Alternation of intermediate station characteristics

-Alteration of Port Profile

-Shutdown of a terminal

-Enabling and disabling of terminal(s) in exclusive mode

-Security lock and unlock of terminal(s).

3.4.3.5 Data Base

-Physical attribute descriptor table definition

-Physical groupings of attribute values into Entities and Entity Record Sets

-Physical data organization, access methods required and storage media spanned

-Physical index tables to be maintained.

-Dynamically establishing new complex logical data structures

-Selectively inhibiting the use of specific Entity Record Sets

-Batch-mode of data base maintenance and re-organization

- Control based on the data content

- Control based on data access operations on data

- Security classification of Entity Record Set types

- Security classification of Entity-Attribute fields

- Security classification by level and by association

- Control over concurrent data access

- Historical versions of data access

- Transaction audit trail on historical and/or current versions of data.

Chapter 3.5

MIGRATION, CO-EXISTENCE, INTERCHANGE

3.5.1 Background

This subject is probably the most difficult strategic issue: to understand the relationship of a new, yet undefined, system to that of present, and changing, systems. Because its impact is so broad - engineering, programming, customers - there is a tendency to delay decisions which, like ecology problems, unconsiously translate themselves into a default decision of incremental improvements until eventually the panic of crisis forces a major change.

The Company's goal is to make profit on a continuous basis, both yearly and long range. It predominantly makes that profit from engineering products, hence this is the major migration factor - and not programming. Obviously, programming is important to making the engineering products attractive, and thus indirectly affects profitability. Since programming is the primary user interface, it is also important to separate it as logically as possible from the engineering to allow for easy introduction of new engineering products.

The point being stressed here is that programming migration from one operating system to another is a lesser, albeit, important factor than that of engineering product migration. It is essential to understand what feasibly can be done to aid programming migration, and what cannot. New system attempts in the past have burdened themselves with so many compatibility constraints that they lost their capability to introduce the new concepts that justified having a new system in the first place.

There is another facet of these self-defeating myths, namely the one that says that anything conceptually new is too far out (i.e. ad tech) because it is so difficult to even extend present products - witness OS and DOS. What is generally forgotten is that it is not the new functions that are conceptually difficult, but it is the unsuitable system structure, present low functional engineering interface levels, and the lack of programming interface control that are the primary inhibiting factors.

A product ship "window" can be foreseen around 1977 for an opportunity to make a major system architectural change with the combination of the ending of S/370 CPU/memory product lives and the advent of LSI components. The subsequent portions of this section attempt to define the major issues involved in taking advantage of that "window" to introduce a system base which at this time has the possibility of being an "ultimate" one - from technical intuition, ability to adjust to both user functions and introduction of new engineering products, and from the eventual "defined by enertia" effect. These factors coupled with the increasing obvious "aging" of present operating systems to changes should give rise to serious management reflections if we do not take advantage of the FS "window".

3.5.2 Issues

There are a number of issues that need to be realistically appraised to best understand the tradeoffs over time that need to be made to get AFS introduced into the marketplace.

- First of all, a thorough evaluation effort for AFS from all facets of IBM is essential to gain the best system base possible. In particular, a strong central system architecture group will be required to ensure that a consistent set of tradeoffs is made to maintain for new market requirements and technology.

- AFS will have a new program SCPI (System Control Program Interface), which will be different from OS and DOS. It should be realized that even a new S/370 - based FS operating system would also need a new SCPI. As a result, program migration must be as a result of at least re-compilation. If agreement on the common intersection of the feasibly possible user interfaces (HLL, CCL, FDL, and DDL) was obtained (in 1972?), then emphasis could be made to direct users to that common set during the interim. A corollary of this which needs to be accepted is that many, probably the majority of programs, will not be easily moved by re-compilation. These in particular include the major system efforts to take full advantage of S/360.

- Because of the marketing factor that FS PPUs must replace S/370 CPUs, and because of the level of incompatibility between the two (in spite of the above HLL, etc. compatibility efforts), co-existence of present operating systems is essential. Furthermore, a basic co-existence capability is required (with a 2:1 cost/performance) which still allows for an attractive performance lure to AFS. The tradeoffs between these two are some of the most critical needed to be made.

- Second generation IBM systems (14xx, 70xx) should only have to be simulated on S/360 under OS or DOS and hence have no direct impact on either the SL or engineering units of AFS.

- An unresolved issue is the ability/need to have co-existence of GSD systems.

- Another unresolved issue is the ability/need to have co-existence of non-IBM systems.

- The ability to dynamically interchange information logically between AFS and other operating systems should only be by means of a formal networking protocol. This will provide native (co-existence), local, remote transparency to users of these systems as well as limit the impact of co-existence of the other systems on the structure of the AFS control program.

- Co-existing non-AFS data, along with programs and operating systems, must also be controlled by AFS. Logically this data is owned by their own operating systems and requested via the networking interface if used by a job running on the AFS control program. Physically, the data may be stored in the SMU or via a S/370 interface to individual storage devices. Individual devices will only be used by native non-AFS systems. They are of two classes: those that can also work in the SMU and those that cannot. Non- AFS data can be moved in an application user transparent manner off the possible SMU devices into the SMU after which the devices can be added to the SMU. The older storage devices including tapes, which are not possible to be put in the SMU, can remain until their cost/performance is low enough at which time their data can also be moved into the SMU and these devices removed from the system.

- Source/sink equipment, with the proper interim product plan, should be able to directly connect to AFS via a 27RN-like Communication Unit (CU). Present operating systems should evolve as much as possible towards the data communications architecture concepts outlined elsewhere in this document. In particular, native systems should act as if they had a CU attached to them - thus providing a clearer interface to the Source/Sink (S/S) subsystem of the AFS control program.

3.5.3 Course of Action

The general course of action at this time is to develop the broad technical understanding of AFS architecture; realistically appreciate what can be done to aid migration and their tradeoffs; and then seek to take advantage of the interim time to prepare both our spectrum of engineering and programming products and the customer community to ease the transistion of introducing AFS into the marketplace.

Part 4

THE MAN-MACHINE INTERFACE IN AFS

This part of the report is to become a description of AFS in terms of the basic infix form. The user who wants to learn to use the system without probing into its inner workings may do so by reading only this part. At present, only two chapters have been started. Chapter 4.3 describes the functions and syntactic markers, and Chapter 4.5 presents examples of SL programs.

Chapter 4.3

SUMMARY OF BASIC INFIX FORM

4.3.1 Introduction

In this chapter, the functions and syntactic markers are described as they are used in the basic infix form of SL. This is the form that people usually want to see and to think about. Compilers will usually produce the strict form, so a few people will be interested in seeing strict form. The following expression is written each way:

```
(a+b) ÷ (c+d) stow e

stow(sum(quotient(a;b);sum(c;));e)
```

The basic infix form is described in terms of the strict form in which the primary description of SL has been given. Eventually, this chapter will become a programming manual and will contain a partial repetition of a description of the semantics of SL so that a programmer who chooses not to delve below the basic infix level will not have to do so. For the present, however, only enough semantics is given here to guide the reader who has read the previous chapters at least cursorily.

In particular, the syntactic form of program text is discussed in 2.2.2. Some readers will find it helpful to review that section before reading the following descriptions of functions.

Some syntactic markers have the form of functions, so they are included in this exposition without further ado since they have syntactic properties like those of true functions. Included also for completeness are certain other syntactic markers which are quite different: parentheses, braces, semicolon. These are listed in alphabetical order with the other syntactic markers and with the functions. It may be helpful to read these first.

The following examples explain the rules used to translate n-adic function and syntactic marker definitions from strict form to basic infix form:

```
        f(x)      becomes   f x
        f(x;y)    becomes   x f y
        f(x;y;z)  remains   f(x;y;z)
```

If the function name is alphabetic, blanks must be used to delimit it.

Blanks may be used freely throughout SL in most reasonable places. They may be placed before or after any non-alphanumeric character that represents a function or syntactic marker, or they may be omitted. At least one blank must be used to separate adjacent alphanumeric symbols. Wherever one blank may appear, any number of blanks may be used. Blanks must not appear in a symbol, in a function represented by something produced with more than one key stroke, or in a constant that is not a character string.

At present, evaluation is left to right, and there is no precedence except that semicolons, parentheses, braces, and brackets are considered to delimit expressions. More precedence relations may be introduced in subsequent editions.

There are two classes of symbols: function symbols that represent functions requiring arguments and elementary symbols that represent objects that do not require arguments in order to be evaluated. In the strict form, the syntax of the expression in which the symbol occurs indicates the class to which the symbol belongs. In the basic infix form, the notation is more concise and the class of a symbol is not indicated by the syntax of its use. Instead, the class is recorded in the dictionary of the module, and it is determined by the definition of the symbol. If it is defined by a lambda expression with one or more arguments, then it is a function symbol. If it is defined by a functional that has function symbols as arguments, then it is a function symbol. Otherwise, it is an elementary symbol. (Ref. 2.2.2)

Eventually, many functions and syntactic markers will be expressed by single characters. For this exposition, however, most of them are represented by mnemonic names or abbreviations.

In certain cases a familiar character has been used (like + or -).

4.3.2 Common Abbreviations

The form used to describe a function or syntactic marker includes a "where" section that defines notation, variables, etc. Certain very common abbreviations are defined here for once and for all, and the definitions are omitted in the many operator definitions. The following are syntactic variables that stand for instances of classes of character strings. Two instances of an abbreviation in a single expression do not necessarily stand for the same string. If they do, a digit will be appended (e.g., stmt3), and the same digit will be appended in two instances that refer to identical strings.

abbreviation	stands for
expr	expression
stmt	statement

4.3.3 Alphabetical Listing of Functions and Syntactic Markers

Eventually, this section will become a programming reference manual with every function and syntactic marker described. At present, the functions listed in section 4.3.4 are not described. However, the reader who is familiar with APL can understand them well enough from their names and from the introductory remarks in 4.3.4.

Section 4.3.5 lists functions that are defined elsewhere or not defined in this report.

Section 4.3.6 summarizes the situation.

Section 4.3.7 gives a preliminary rough analysis of the complexity of SL, judging it in terms of the number of functions and syntactic markers required.

The functions and syntactic markers are arranged alphabetically with the names in the bottom title.

The examples given at the top of each page are intended to be exhaustive and to cover all possible uses of the symbol being defined. This goal has not been achieved in Edition 3.

examples: g apply a

where: a is an ordered list of symbols like (x;y), or it is a single symbol.

g is an expression whose value is an unevaluated expression or unevaluated group of statements.

value: The value of the last expression evaluated.

side effects: None

use: In the examples: The dyadic apply applies g to a.

comment: The implicit invocation mechanism is occasionally inhibited by built-in mechanisms to prevent ambiguity. Sometimes, the programmer intentionally inhibits the invocation mechanism so as to be able to manipulate an expression or group rather than just its value. When this is the case, it is clear from the definitions of the operators involved. The purpose of apply is to execute code whose invocation has been inhibited. The dyadic apply function also associates parameters with the function it invokes. The monadic eval performs this function without associating parameters.

References: 2.2.6, 2.6.9.1 , eval

example: r authorize x

(evaluates & copies) authorize x

where: x is an object.

r is a rights expression. The allowable rights are the present tense third person singular verb forms of the names of the requests that may be made on an object: authorizes, copies, deletes, destroys, evaluates, identifies, inserts, selects, starts, and stows. To authorize all rights available in the right argument, specify "all".

value: A synonym that provides authorization for access to x.

side effects: The synonym, an object, is created.

use: A synonym is like a pointer, but it has safeguards so that it cannot be used except by requests with the proper authorization. Unlike a pointer, a synonym automatically passes all authorized requests to the object to which it points, whereas a PL/I pointer requires a further operation on it to produce a value.

comments: Synonyms and metonyms are accessors. It is not possible to convert any other data type into an accessor. This protects the system integrity from incursions of the sort that can be accomplished by adding integers to PL/I pointers in OS/360.

It is possible to convert a synonym to a metonym by the enclose function, and vice versa with a disclose function.

The authorization conveyed by a synonym is the authority to use functions that use requests corresponding to the rights in the rights expression. Notice that the names of the rights are the first person singular verb forms of the corresponding request names.

An authorize expression that attempts to convey rights not possessed by the object will raise an error exception.

Synonyms and metonyms are needed by data base applications.

References: 2.1.5, 2.1.4, 2.6.3.5, syn

examples:

{stmt;stmt;stmt}

f{stmt;stmt;stmt}

{expr}

value: The value of a group of statements delimited by braces and semicolons is a collective object (a list) whose elements are the statements.

side effects: None

uses: A pair of braces delimits a portion of code and inhibits the implicit evaluation mechanism.

A specific use of a pair of braces is to delimit a group of statements in order to use the group as an argument of a function.

Another specific use is to enclose an expression so as to inhibit the action of the implicit evaluation mechanism.

Comments: A pair of braces may be used in SL to perform the function of BEGIN;....;END; or DO;....;END; in PL/I. A new environment is created for a group when it is invoked if and only if some function in whose arguments the group appears or scme statement in the group requires an allocation of storage that is local to this invocation.

Braces can only be understood if one understands semicolons and parentheses. See first the page on delimiters and then the pages on semicolons and parentheses.

Braces are syntactic markers that do not appear in the code that the machine executes.

References: 2.2.2, delimiters, semicolon, parentheses examples: p conditional expr

where: p is a predicate, an expression whose value is true, 1, or false, 0.

value: When the value of the left argument is 1, the value of the expression is the value of the right argument. When the value of the left argument in 0, the value of the expression is nil since it is not executed.

side effects: If the left argument is 1, control returns from the group. This is like the PL/I RETURN-statement which causes control to return from a block. If the value of the left argument is 0 the expression has no side effects.

use: To terminate the evaluation of a group conditionally.

comments: The conditional provides the means to express conditional expressions. It will probably be represented by a single character. In this case, nested PL/I IF THEN ELSE statements, and LISP conditional statements will be handled concisely and elegantly.

References: 2.2.7, exit

example: p create x

where: p is a procedure description.

x is an object.

value: An internal identifier of the object it constructs.

side effects: It constructs an object which has, in its access machine, p as its procedural description, and a process status record and interpreter that are appropriate to p. The resource of the object is a copy of the resource of x, translated to fit the new access machine.

use: To construct objects using software procedure descriptions.

Reference: 2.6.2.2

examples:

```
declare x y z stop static;
                   p unique;
a b c new automatic
{stmt;stmt;stmt;stmt;stmt}
```

```
declare d g
```

where: d is a list of scope and storage class

g is a group of statements. Among these may be statements that affect access machines, in other words, declarative statements, other than those that affect scope and storage class.

value: The value of the group, in other words, the value of the last expression evaluated before control exits from the group.

side effects: The variables listed in the space between the declare marker and the group have the attributes mentioned.

use: To make scope and storage class declarations.

comments: The need to separate declarations of scope and storage class from other declarations is a result of the fact that SL is a machine language and the basic infix form is not rearranged before being executed. In the extended infix form declarations will probably be more like those in PL/I.

examples: i delete x

where: x is a collective object.

i is a member of the index set of x.

value: The erstwhile ith member of x

side effects: The storage cell corresponding to i is removed from its index set.

use: To delete storage cells from the resource of a collective object.

Reference: 2.1.6

examples:

{stmt;stmt;stmt} This is the external representation of a collective object, a list of three unevaluated statements.

f(stmt;stmt;stmt) This triadic function is interpreted as a monadic function that takes as its argument a list of the values of the three statements.

g{stmt;stmt;stmt} This is a monadic function that takes, as its argument, a list of three unevaluated statements.

a ÷ (b+c) The denominator is the value of the sum.

{expr} The braces inhibit the implicit invocation mechanism.

uses:

A pair of parentheses delimits a portion of code and does not inhibit the implicit invocation mechanism.

A pair of braces delimits a portion of code and inhibits the implicit invocation mechanism.

Semicolons delimiting the constituents of a portion of code, delimited by braces or parentheses, indicate that the constituents are the elements of a list.

comment:

These delimiters are shown together on this page to illustrate the symmetry. For details, see references.

References: braces, parentheses, semicolon examples: disclose m

disclose n + 1

disclose x

where: m is a metonym for an object y.

n is a metonym for a floating point number x for which there is a synonym s.

y is a collective object.

x is enclose y.

value: a synonym for y, if the argument is a metonym. y if the argument is enclose y.

use: A metonym is a pointer and disclose is used to get at the value it points to.

comments: In the second example, n+1 would raise an error exception, whereas s+1 would compute the sum correctly. Note that (disclose n+1) = (m+1), and that (m+1) =(x+1) .

For any object x, disclose enclose x = x.

References: 2.1.5, 2.1.7, enclose

examples: enclose x

where: x is an object

value: If x is a collective object, the value is a scalar object that contains the collective object x.

If x is a synonym, the value is a metonym.

uses: In the first case, it is used to make it possible to compare characters instead of bit vectors or to compare words instead of charactor vectors.

In the second case, it is used to make metonyms which are like PL/I pointers.

comments: For any object, x, disclose enclose x = x.

References: 2.1.5, 2.1.7, disclose

examples: evaluate g

where g is an unevaluated group of statements or and unevaluated expression.

value: The value of the last expression evaluated.

side effects: none

use: To execute an expression or group when the implicit invocation mechanism has been inhibited.

comment: See the comment under apply.

References: apply

examples: exit expr

value: The value of exit is the value of the expression that is its right argument.

side effects: If the exit statement occurs in a group, control returns from the group. This is like the effect of the PL/I RETURN-statement which causes control to return from a block.

use: To terminate the evaluation of a group.

References: 2.2.7

example: goto s

where: s is a symbol that has been used as a label or an expression that has the value of such a symbol.

value: The goto statement is not a normal expression and does not have a value.

See comments below.

side effects: The goto generates a sequence exception which is handled by the monitor. The next expression to be executed is the one whose label is the right argument of goto.

use: To perform the function of goto or branch in object code produced in translating from other languages.

comment: goto is not necessary for programs written in SL. Many users will prefer to eliminate it from the repertoire of functions available.

If it is feasible to label expressions as well as statements, and if it is feasible for a goto statement to have the value of the last previously evaluated expression, then the goto will provide a particularly powerful tool. However, this capability will not be added if it implies significant cost increase or performance degradation.

Reference: label

example: ibase x

where: x is an array

value: The index base of the array x which is a list of lists. The ith sublist is a list of the values that the ith element of a member of the index set may take, and they are listed in order of increasing value.

side effects: The list of lists is created.

use: To generate the index base.

comments: The abbreviation ibase stands for index base.

Reference: 2.1.7.3

example: igenerator s

igenerator (0;1;2)

where: s is a list of positive integers.

value: A list of lists of integers. Each sublist is a primitive index set (i.e., 0, 1, 1,....,n), and the number of elements of the kth sublist is the kth element of s.

side effects: The list of lists is created.

uses: To generate the index base for a primitive array from the shape of the array.

examples: ilist x

where: x is a collective object.

value: A list that is a copy of the index set of x.

side effects: Production of the copy.

comments: If x is a vector, ilist x is the same as the APL expression iota rho x.

The abbreviation "ilist" stands for "index list".

Reference: 2.1.5

example: i insert x

3 replace (i insert x)

where: i is an object, not in the index set of x but suitable to be added to it, or it may be the object nil.

x is a collective object.

value: An implicitly defined synonym of the i component of x.

side effects: (1) A new storage cell is added to x.

(2) i is added to the index set of x.

(3) i is mapped onto the new storage cell.

(4) A copy of undef is placed in the cell.

use: An important use is the one illustrated in the second example which adds an object, a storage cell to put it in, and a member of the index set of access it with.

comment: The new member is added to the end of the index set, if the index set is ordered. To move it elsewhere, a subsequent application of rotate will do so.

If i is already a member of the index set of x, or if i is not nil and not in the admissible set of indices for x, an error exception is raised.

References: 2.1.7, 2.6.3.2

example: s:expr

where: s is a symbol.

value: The value of s:expr is the value of expr.

side effects: The symbol s becomes a label of expr.

use: To attach labels to expressions so that they may be the target of a goto function.

comments: The colon is a syntactic marker that indicates that some symbol is a label and indicates the expression that it labels.

A label is a read only value initialized at compile time.

Labels appear to be useful primarily for object code created by translators from other languages and not for native mode SL programming.

Possibly, it will be found that only statements can be labeled, and that it is too costly to be able to label expressions inside statements.

examples: a lambda g

(x;y) lambda {stmt;stmt;stmt}

where: a is an ordered-list of symbols.

g is a group

x and y are symbols.

value: An n-adic function where n is the number of symbols in the left argument.

side effects: None

use: A lambda expression may be assigned to a symbcl, making it a function symbol. Alternatively, the lambda expression may be used in place of a function symbol in an expression.

comments: The lambda expression is the means, in SL, to extend the functions available. SL may be extended in data types by defining new access machines. To accommodate new data types, cld functions must be redefined by assigning to them the value of an appropriate lambda expression.

The names of the arguments of the function are given in the symbol list in the order in which they must appear in an expression that uses the function.

References: 2.2.3, 2.6.1.2

examples: parallel g

parallel {stmt;stmt;stmt}

where: g is a group.

value: A vector whose elements are the values of the statements comprising the group.

side effects: None

use: To state that the statements comprising the group may be executed in parallel or in any order the machine selects.

References: 2.2.5, 2.6.9.2

examples:

```
f(stmt;stmt;stmt)

a/(b+c)
```

value: Parentheses do not inhibit the implicit invocation mechanism so the value of a portion of code delimited by parentheses is either the value of an expression or a list of values of statements.

side effects: None

use: A pair of parentheses is used to delimit a portion of code without inhibiting the implicit invocation mechanism.

One specific use of parentheses is to control the order of execution of functions in an expression.

Another specific use of parentheses is to delimit the argument list of an n-adic function when n is greater than 2, and when, as is usually the case, the arguments are to be evaluated before the function is evaluated. In SL, such a function is interpreted to be a monadic function that takes the argument list as its argument.

comments: To understand parentheses, it is necessary to understand the semicolon and braces. Read first the page on delimiters and then the pages on parentheses, braces, and semicolon.

References: delimiters, braces, semicolon example: remove x

where: x is an object.

value: x

side effects: A copy of undef is placed in the storage cell so that if x is evaluated again, an error exception is raised.

use: To remove the contents of a storage cell without destroying the cell.

Reference: 2.1.6

examples:

```
p repeat g

1 stow i;(i<10)repeat {i+1 stow i;stmt;stmt;stmt}
```

where:

p is a predicate, an expression that evaluates to 1 or 0

g is a group of statements

i is an integer

side effects: None

value: The left argument is evaluated. If its value is one, the argument on the right is evaluated. Then the left argument is reevaluated and the cycle is repeated. If the value of the left argument is zero, execution ends. The value is the value of the last expression executed in the right argument. If the right argument is not evaluated at all, the value is nil.

use: The second example is equivalent to , in PL/I:

```
DO I=1 TO 10;stmt;stmt;stmt;END;
```

The extended infix form will probably have a DO-statement of this sort.

References: 2.2.7, 2.6.9.1

examples: x replace y

where: x and y are objects

value: The value is a copy of x.

side effects: The object y is destroyed, unless y refuses to destroy itself. In this case, y remains unchanged and an exception occurs.

use: Usually replace is used when one argument or the other is an expression that has the value of an object. Then it is possible to make an object that is a copy of a component of another object or by using insert, to add a copy of an object as a new element of another object.

comments: Notice that replace changes the whole object, both access machine and resource. Stow, on the other hand changes only the resource.

References: 2.1.7, 2.6.4

examples:

```
i  select  x

i  select  x stow y
```

where:

x is a collective object.

i is a member of a synonym for a member of the index set of x.

y is an object whose access machine is suitable.

value:

An implicitly defined synonym for a member of the right argument whose index is the left argument.

comment:

Select does not create a copy but merely identifies some part or parts of the collective object that constitutes the right argument. To create a copy, it is possible to use a stow function as in the second example. In this case, the target y must have an access machine that is suitable for the ith element of x.

References: 2.1.5, 2.6.3.1

examples: {stmt;stmt;stmt}

f(x;y;z)

value: Semicolon does not have a value.

side effects: The semicolon is a delimiter whose precedence is lower than any function or functional. If two expressions are adjacent, one must be an operator and the other must be one of its arguments. However, if a semicolon intervenes, they become two elements of a list. As such, they are called statements.

The difference between a statement and an expression is that a statement is a member of a group of statements and then, when the group is evaluated, the value of a statement is discarded after the execution cursor passes the semicolon and before evaluation of the next statement begins.

comments: The semicolon is used to delimit the arguments of an n-adic function when n>=3. The comma is not used because it is reserved to be used as the name of a function.

To understand the semicolon, it is necessary to understand braces and parentheses. Read first the page on delimiters and then the pages on braces, parentheses, and semicolon.

References: 2.2.2, delimiters, braces, parentheses example: shape a

where: a is an array

value: A list of integers of length r where r is the rank of the array (the member of dimensions) and the ith element in the list is the size of the ith dimension of the array.

side effects: The list is created.

examples: x stow y

where x and y are objects or are expressions that have the value of objects.

value: The value is an object that has the access machine of y and the resource of the value of x.

side effects: The left argument is evaluated. Then, the right argument is evaluated. Finally, the resource of the value of the right argument replaces the resource of the value of the left argument.

uses: This is the normal assignment that takes place in languages like PL/1.

comments: To produce the kind of assignment that appears in APL see replace.

References: 2.1.4, 2.3.5, 2.6.4

example: syn x

where: x is an object.

value: A synonym that provides authorization for access to x.

side effects: The synonym, an object, is created.

use: A synonym is like a pointer, but it has safeguards so that it cannot be used except by requests with the proper authorization. Unlike a pointer, a synonym automatically passes all authorized requests to the object to which it points, whereas a pointer demands a further operation on it to produce a value.

comments: It is not possible to convert data of any other kind to a synonym. This protects the system integrity from incursions, such as can be accomplished by adding integers to PL/I pointers.

To generate a synonym with fewer rights than one already has it is necessary to use the authorize function.

Synonyms are needed for data base applications.

References: 2.1.5, 2.6.3.5, 2.1.4, authorize

4.3.4 APL Functions to be Implemented in Hardware

From the preceding discussion and a knowledge of APL, the approximate meaning of the following will be obvious. There are a total of 59 functions in this category. Notice that some APL functions are defined elsewhere and are not listed here. The hyphen indicates when a dyadic and monadic function are related. When the SL name differs from the APL name, it is shown in parentheses.

Monadic		Dyadic
plus	-	plus(sum)
reciprocal	-	divide(quotient)
negative(minus)	-	minus(difference)
signum	-	times(product)
ceiling	-	maximum
floor	-	minimum
exponential(exp)	-	power
nat log(ln)	-	log
magnitude	-	residue
sin		and
cos		or
tan		nand
arcsin		nor
arccos		less(lt)
arctan		not greater(le)
sinh		equal(eq)
cosh		not less(ge)
tanh		greater(ge)
arcsinh		not equal(ne)
arccosh		
arctanh		
not		take
membership		drop
		reshape
ravel	-	catenate
reverse	-	rotate
transpose	-	transpose
grade up		compress
grade down		expand
pi times		outer product
reduction(reduce)		inner product

Someone might argue that the circular functions are just one and not 13 functions. From one point of view, they are 13 functions with hard-to-remember names.

4.3.5 SL Functions Defined Elsewhere

The following 38 functions are defined or identified elsewhere in this report: aquire, augment, base value, claim, connect, copy, delay, delayed parse, destroy, free, identify, ignore, index, inject, insert symbol, introduce, list, load, locate, map, member, monitor, name value, point, priority, quotient_remainder, release, representation,send answer, send message, signal, step, suspend, translate(dyadic), translate(monadic), ultimate, unique name, wait answer, wait message.

4.3.6 Summary of Functions So Far Indentified

Defined in 4.3	28
Identified by APL	59
Indentified elsewhere	38
	125

4.3.7 A Measure of SL Complexity

The complexity of SL can be measured roughly by comparing it to APL which performs a much more constrained function but has large areas of similarity. To do this The APL functions that remain will be dientified.

There are 8 APL functions that have clear conterparts among the SL functions mentioned: branching(goto), function definition(lambda), local variable identification(declare), specification(replace), size(shape), trace control(monitor and ignore), label(label), indexing(select), comma (augment).

There are 50 more APL functions to do things that SL will do but the relationship is not direct either because the details have not been worked out or because the work is done differently. In some cases the work is actually done by functions already identified. These are: editing control, editing mark, display controls, locked function, stop control, terminal input, character input, 34 system commands, 9 system dependent(I-beam) functions. In SL all of these things will be done with the kind of functions so far identified. There will not be the diversity seen in APL/360.

Finally there are 9 APL functions that will probably be programmed:

encode	decode
factorial	binomial coefficient
roll	deal
three square roots of sums of squares	

No decision has been made as to which of these marginal APL functions belong in the basic infix level of SL and which should be programmed. It may be, for example, that none of the circular functions will be in the machine language. However, all of them will be in the extended infix form and all of them will be supported where they appear in the various favored high level languages. With this information, the languages can be compared as follows:

APL/360 functions with direct SL counterparts	67
APL/360 functions SL will cover	50
	117
APL functions to be programmed	9
	126
SL functions with clearly defined APL counterparts	67
Other identified SL functions	58
	125

Clearly more functions will be added to SL. However, it seems clear that SL will be only a little more complicated that APL/360 while providing much more capability.

Chapter 4.5

EXAMPLES OF SL PROGRAMS

This chapter demonstrates the suitability of SL as a target language for the translation of programs from PL/I, COBOL, FORTRAN, APL, RPG and LISP. For each of these languages, typical program constructs are illustrated (along with contextual information when appropriate) and followed by an equivalent SL construct. The SL examples given are written in the basic infix notation (refer to Section 4.3).

Programs written in SL to accomplish these same purposes would be much simpler since they would not involve the complexities of the various source languages.

4.5.1 Translations from PL/I

Example of simple case of PL/I DO statement:

```
DO I = 1 TO 10; statement_list; END;
```

The SL code for the above is:

```
0->I;{I+1->I≤10}repeat{statement_list}
```

In a somewhat more complicated case with various data types involved in the iteration calculation, there can be rounding problems that prohibit the simple initialization used above. Furthermore the value of the iteration limit can be changed during the iteration so there must be a temporary. In this case:

```
DO I=1 TO N;
      statement_list
   END;
```

Equivalent SL group:

```
  {declare C unique;
   {0 stow C;
    {eval{C select {{1 stow I                 };
                    {I sum 1 stow I le N stow C}
                   }
```

```
         }
    } repeat {statement_list}
   }
  }
```

The general case of the PL/I DO statement is much more complicated than the ordinary user realizes, or can utilize often. A full explanation of the interaction of the TO and BY clauses with the WHILE option, with more than one specification present, may be found in the PL/I Language Specifications manual (Y33-6003-1, pp 144-146) or the PL/I (F) Language Reference Manual (C28-8201-2, pp 364-367). This general case can be programmed in SL using a single skeleton (with the possibility of repeating one section as the multiple specifications require), substituting for the names of the variables used in the DO statement. An example of this is shown below:

```
  DO I=J1 TO K1 BY L1 WHILE(E1),
       J2 TO K2 BY L2 WHILE(E2),
                ...            ;
         statement_list
     END;
```

Equivalent SL group:

```
  {declare U V W C BODY TEST unique;
   syn{statement_list} stow BODY;
   syn{signum V is 0 select{I le U;I ge U}} stow W;
   {0 stow C;
    syn{eval W and E1} stow TEST;
    {eval{C select {{K1 stow U;L1 stow V;J1 stow I;TEST stow C};
                    {I sum V stow I;TEST                         }
                   }
         }
    } repeat BODY
   };
   {0 stow C;
    syn{eval W and E2} stow TEST;
    {eval{C select {{K2 stow U;L2 stow V;J2 stow I;TEST stow C};
                    {I sum V stow I;TEST                         }
                   }
         }
    } repeat BODY
   };

   .
   .
   .
  }
```

Note that the first three lines are the setup code which need be present only once regardless of the number of specifications appearing in the original PL/I DO statement. These are followed by a pattern group which is repeated once per specification, separated by semicolons as necessary, and terminating with the final right brace.

This general skeleton can be simplified substantially by a compiler if the original DO statement does not contain all of the most general options. For example, if the expressions in either the TO or BY clauses are constants, the corresponding temporaries U and/or V can be eliminated. If the BY expression is a constant, then the entire expression "signum V is 0" can be evaluated at compile time, and the result can be used to chose the expression to be substituted for W. Sufficient evaluation of constant expressions at compile time can result in the reduction of the general case to a much simpler program, like the one shown for the simple case of PL/I DO.

4.5.2 Translations from COBOL

Example of EXAMINE, IF and ALTER statements:

```
        EXAMINE INPUT-RECORD TALLYING ALL ','.
        IF TALLY IS EQUAL TO 0
           THEN ALTER SWITCH TO PROCEED TO EXIT.
        .
        .
        SWITCH. GO TO.
```

Equivalent SL statements:

```
     elem sum reduction (INPUTRECORD member ',') stow TALLY;
     eval({;{EXIT} stow X}[TALLY eq 0]);
         .
         .
     goto X;
```

4.5.3 Translations from FORTRAN

Example of ARITHMETIC IF statement:

```
   IF (E) 12,56,13
```

Equivalent SL statement:

```
   goto(signum E select{12;56;13});
```

4.5.6 Translations from LISP

Example of LISP conditional statement:

```
    COND((P1   E1)
         (P2   E2)
              .
              .
         (Pn   En))
```

Equivalent SL statement:

```
  eval{P1 condition E1;P2 condition E2; ... ;Pn condition En};
```

Part 5

A LOGICAL IMPLEMENTATION

A logical implementation of the system is being defined using the Vienna Definition Method. Initially the logical implementation will be presented in English. In later versions of the document, the formal notation will be introduced.

Chapter 5.1

BASIC STRUCTURE

An object construct is a storage cell and its contents.

A storage cell is named by an iid and contains queue(s), a queue manager and an object. Iid's are unique, not reused. An iid is the internal representation of a cell name. A storage cell is known as a buffer when the ownership conventions are suspended, e.g. a request is always sent in a buffer because ownership of the object construct is retained by the sender until the recipient accepts it.

A queue contains the iid's of buffers which represent messages being sent between object constructs. Queues are organized in a FIFO fashion. There are request queues and response queues.

A request queue queues the iid's of requests intended for processing by the access machine associated with the object of this object construct. Every storage cell has at least one request queue.

A response queue queues the iid's of responses for processing by the access machine associated with the object of this object construct. A storage cell may have none, one, or more response queues.

A message whose iid is placed on a queue is the communication link to and from object constructs. There are request messages and response messages.

A queue manager is associated with the queues of each storage cell. It is the communication interface between other object constructs and the object of this object construct. As soon as an object construct is created, the queue manager can begin to handle incoming requests. The queue manager of each storage cell can handle messages in parallel with the queue managers of all other storage cells in the system. Each queue manager handles its messages sequentially. The logic of queue managing is written extralingually, e.g. in micro-code.

An object contains an access machine and a resource.

An access machine is associated with the resource of each object. It is the processing interface between the queue manager and the resource of this object. As soon as an object construct is created, the access machine can begin to process incoming requests. The access machine of each object can process messages in parallel with the access machines of all other objects in the system. The access machine process is described by three components. These are a procedural description, an interpreter of the procedural description, and a process status record (PSR). The procedural description describes the processing logic. The interpreter provides the actual motive force for the process by interpreting the procedural description. The PSR is an area of storage in which the interpreter records the current state of its interpretation of the procedural description. The logic of an access machine is written either extralingually or in SL. If the logic is written extralingually, the object is said to be primitive. If the logic is written in SL, the object is said to be reducible.

A primitive object is an object whose access machine is written extralingually. All requests sent to the queue manager associated with the storage cell containing a primitive object are passed by the queue manager to the access machine of the primitive object for processing. In fact, since the logic of the queue manager and of the access machine are both written extralingually, the functions of the queue manager can be merged into the functions of the access machine for primitive objects. This is being done in the logical definition. Further, since the procedural description, the interpreter, and the PSR for a primitive object are all extralingual entities, these components are not separately denoted, but are jointly denoted by the object type, e.g. a LIST-type object.

A reducible object is an object whose access machine is written in SL. All requests sent to the queue manager associated with the storage cell containing a reducible object are passed by this queue manager to the interpreter of the SL code which by being interpreted will process the requests. Since the procedural description, the interpreter, and the PSR for a reducible object are all SL entities, these components are separately denoted by their three iid's.

A resource contains the undifferentiated data value owned by the access machine.

When communicating with foreign architectures, it is not meaningful to transmit the iid of a storage cell containing the object of interest. It is necessary to transmit the object part of a storage cell as a piece of data. An object image is the representation of the object part of a storage cell as data.

Parts of the logical definition of SL require representing certain hardware boxes. It is advantageous to represent them as far as possible as object constructs. Instead of being located via an iid, a quasi-object construct is located via its qid.

Qid's are unique, not reused, and are distinguishable from iid's. In all other respects quasi-object constructs are treated like object constructs. A quasi-object is a representation of an entity requiring service when service is provided by multiplexing a finite number of servers over a potentially infinite number of such entities. For example, the QSLINT-type object (quasi-SL interpreter) represents the requirement for hardware multiplexing of a finite set of I-boxes over all ready processes. The QEVAL-type object (quasi-evaluant) also represents the requirement for hardware multiplexing of a finite set of I-boxes over all ready processes. The QSUM-type object (quasi-sum) represents the requirement for hardware multiplexing of a finite set of adders over all ready processes.

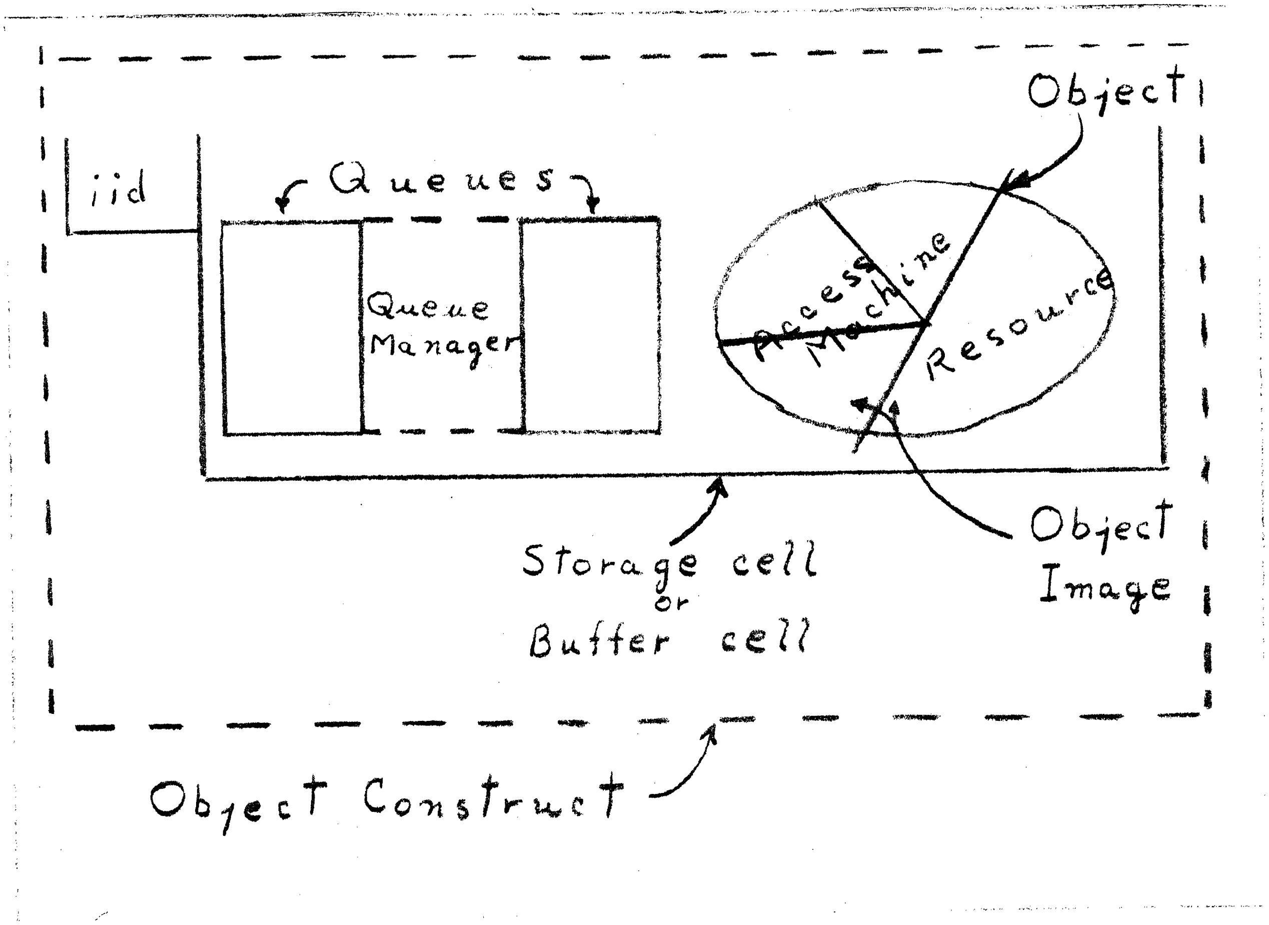


Figure 5.1-1:Structure of a Storage Cell and its Contents

The user who writes a strict syntax SL program deals with syntactic operators and syntactic simple operands. When his text is interpreted, the names he used for his syntactic operators and simple operands will be resolved to some iid. Informally, both operators and operands are represented by objects. To the user an operator represents an object which he wants to invoke, to pass some arguments, to have it operate on the arguments, to send back an answer, and to quit. To the user, an operand represents an object which the user wants to pass as an argument to some operator.

The name, e.g. sum, syn, create, for a primitively defined syntactic operator resolves to an iid of a storage cell of an object construct whose object's object-type is PFUNCTION (for primitive function) and whose resource part contains an indication of the primitively defined operation to be performed, e.g. addition, synonym creation, object construction.

The name, e.g. translate, sin, for a reducibly defined syntactic operator resolves to an iid of a storage cell of an object construct whose object's type is FUNCTION and whose resource part contains the iid of the SL text, the iid of SL symbol table, the iid of the SL link table, and the iid of the outstanding activation table. The interpretation of the SL text defines the operation to be performed, e.g. program translation, sine computation.

The name for a primitively defined syntactic simple operand resolves to an iid of a storage cell of an object construct whcse object's object type could be INTEGER, LIST, SYN, FUNCTION, ... In the case of an INTEGER-type object, the resource part is the integer itself.

The name for a reducibly defined syntactic simple operand resolves to an iid of a storage cell of an object construct which contains a reducible object. The resource part of the reducible object contains the iid of storage used by SL text as it is being interpreted.

Chapter 5.2

BASIC MECHANISMS

5.2.1 Introduction

Some of the most important basic mechanisms are those permitting message communication between object constructs and those permitting message handling by an object construct.

The queue manager associated with the queues of an object construct can invoke the message communication mechanisms. They are: send request mechanism, forward request mechanism, and send response mechanism.

The queue manager can also invoke the message handling mechanisms. They are: wait for request, read request, wait for response, read response.

The access machine associated with the resource of an object performs the actual processing of the requests and the responses.

Conventions for the format of a message are introduced. The creator of the message, the queue manager or the access machine, uses these conventions. They are: request format convention and response format convention.

In describing the mechanisms in English, the logical steps are listed sequentially. In fact some of these steps will occur in parallel, and will be so noted when we describe the mechanism in VDL notation.

5.2.2 Message Communication Mechanisms

5.2.2.1 Send Request Mechanism

Queue Manager

1. passes the following parameters to the send request mechanism: the iid of the recipient of the request, the recipient's request queue number, the iid of the buffer representing the request message, the iid of the sender of the request, the sender's response queue number.

Send Request Mechanism

2. produces a unique msgid. /* A msgid is a unique identifier used to tag a request for the purpose of responding to it.*/

3. completes the request message by adding the msgid to the request message. The iid of the request message is the iid of a buffer containing a LIST-type object. It replaces the first subobject of this LIST- type object with a MSGID-type subobject whose resource part contains the msgid produced for this request message.

4. adds an entry to the System Communication Table. Each entry contains the following information: the msgid, the iid of the sender of the request, the sender's response queue number, the iid of the recipient of the request, the recipient's request queue number, and the iid of the request message. /* the first two pieces of information are essential to message communication. By keeping all these information pieces we depict the Dependency Graph, thus aiding resource management, system restoration, and system verification */.

5. puts the iid of the request message on the specified request queue of the specified recipient.

6. passes back to the queue manager the msgid.

unique msgid	iid of sender of request	number of sender's response queue	iid of receiver of request	number of receiver's request queue	iid of request

Figure 5.2.2-1:System Communication Table Entry Format

5.2.2.2 Forward Request Mechanism

Queue Manager

1. passes the following parameters to the forward request mechanism: the iid of the recipient of the request, the recipient's request queue number, and the iid of the buffer representing the request message.

Forward Request Mechanism

2. the iid of the request message is the iid of a buffer containing a LIST-type object. Using the msgid in the resource part of the MSGID-type subobject of this LIST-type object, it locates the appropriate entry in the System Communication Table.

3. updates the iid of the recipient of the request and the recipient's request queue number with the specified new recipient and new request queue number.

4. puts the iid of the request message on the specified request queue of the specified recipient.

5. returns to the queue manager

5.2.2.3 Send Response Mechanism

Queue Manager

1. passes the following parameter to the send response mechanism: the iid of the buffer representing the response message.

Send Response Mechanism

2. the iid of the response message in the iid of a buffer containing a LIST-type object. Using the msgid in the resource part of the MDGID-type subobject of this LIST-type object, it locates the appropriate entry in the System Communication Table. The entry in the SCT specifies the iid of the recipient of the response and the recipient's response queue number. /* the response goes back with the same msgid used to tag the request to which it is a response*/.

3. puts the iid of the response message on the specified response queue of the specified recipient.

4. deletes the entry from the System Communication Table

5. returns to the queue manager

5.2.3 Message Handling Mechanisms

5.2.3.1 Wait Mechanism

Queue Manager

1. passes the following parameters to the wait mechanism: the queue number to wait on or a list of queue numbers to wait on where the list determines the priority order of message retrieval.

Wait Mechanism

2. waits for an iid to appear on the specified queue. /* Note that the one wait mechanism allows waiting on a request queue or on a response queue/*.

3. when an iid appears, it passes back to the queue manager the queue number on which the iid appears.

5.2.3.2 Read Request Mechanism

Queue Manager

1. passes the following parameter to the read request mechanism: the queue number containing the iid of the buffer representing the request message.

Read Request Mechanism

2. removes the first iid from the specified queue.

3. deletes the iid from the specified queue.

4. verifies that indeed the buffer represents a request message. The iid of a request message is the iid of a buffer containing a LIST-type object. It checks that the second subobject of the LIST-type object is a REQUEST-type object.

5. if yes, it passes back to the queue manager the iid of the buffer representing the request message.

5.2.3.3 Read Response Mechanism

Queue Manager

1. passes the following parameter to the read response mechanism: the queue number containing the iid of the buffer representing the response message.

Read Response Mechanism

2. removes the first iid from the specified queue

3. deletes the iid from the specified queue

4. verifies that indeed the buffer represents a response message. The iid of a response message is the iid of a buffer containing a LIST-type object. It checks that the second subobject of the LIST-type object is not a REQUEST-type object.

5. if yes, it passes back to the queue manager the iid of the buffer representing the response message.

5.2.4 Message Processing

Queue Manager

1. passes the following parameter to the access machine: the iid of the buffer representing the request or response message.

Access Machine

1. /* the request processing logic provided by an access machine involves 'if ...then' logic: if request so and so, then perform such and such, where such and such varies by object type. For example, what a FUNCTION-type object does to process an execute request is far different from what a FLOAT-type object does to process an execute request. The details of what actions each object type does as a function of receiving any possible request, has yet to be defined in this model*/

5.2.5 Message Format Convention

5.2.5.1 Request Format Convention

Queue Manager or Access Machine

1. If the send request mechanism is subsequently going to be invoked , the creator of the request message constructs in a buffer a LIST-type object. The first subobject must be an UNDEF-type object. The second subobject must be a REQUEST-type object whose resource part contains the name of the request. The remaining subobjects must be object types appropriate to each of the parameters of the request. A request need not have parameters but if it does then, for example, if a parameter is the iid of some storage cell, the subobject would be an ACC-type object. If a parameter is some integer, the subobject would be an INTEGER-type object. /* A buffer acquired when some request was sent to the queue manager could be used as the buffer in which to construct the request */.

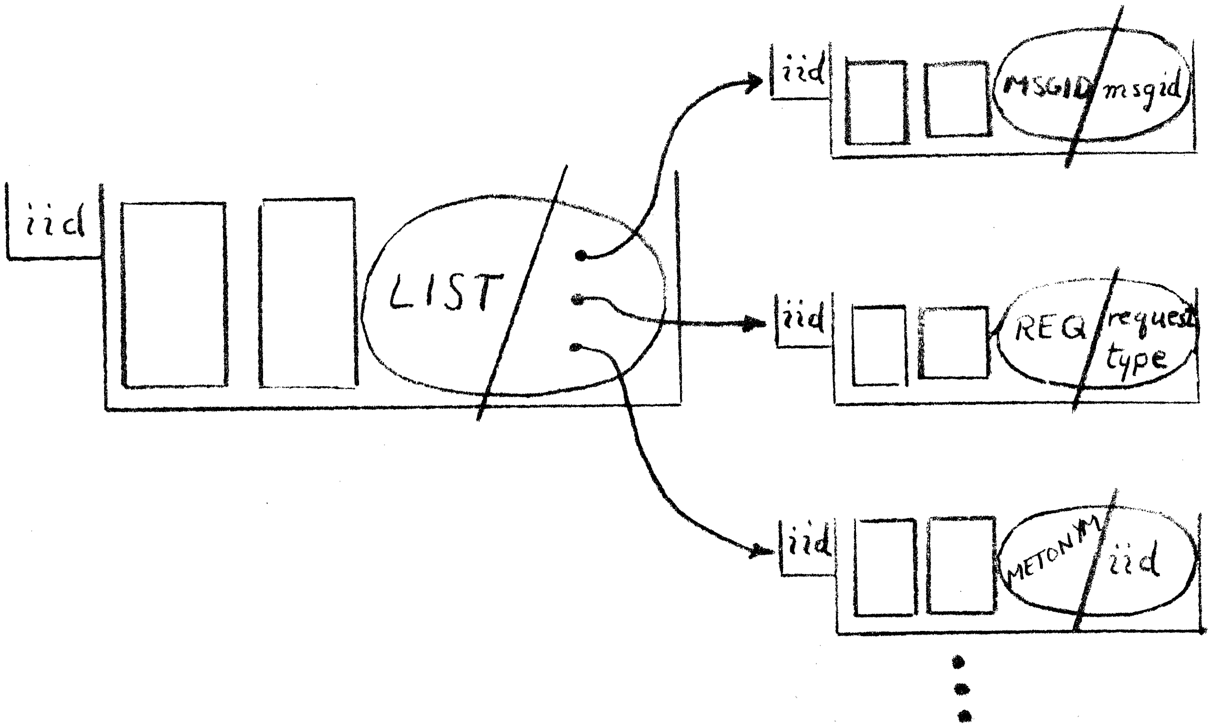


Figure 5.2.5-1: Format of a Request

5.2.5.2 Response Format Convention

Queue Manager or Access Machine

1. If the send response mechanism is subsequently going to be invoked, the creator of the response message constructs in a buffer a LIST-type object. The first subobject must be a MSGID-type object whose resource part contains the msgid that came over with the request message to which this is a response. The remaining subobjects must be object types appropriate to each of the components of the response. /* The buffer representing the request to which this response message is a response should be used as the buffer in which to construct the response. The correct msgid is already there*/.

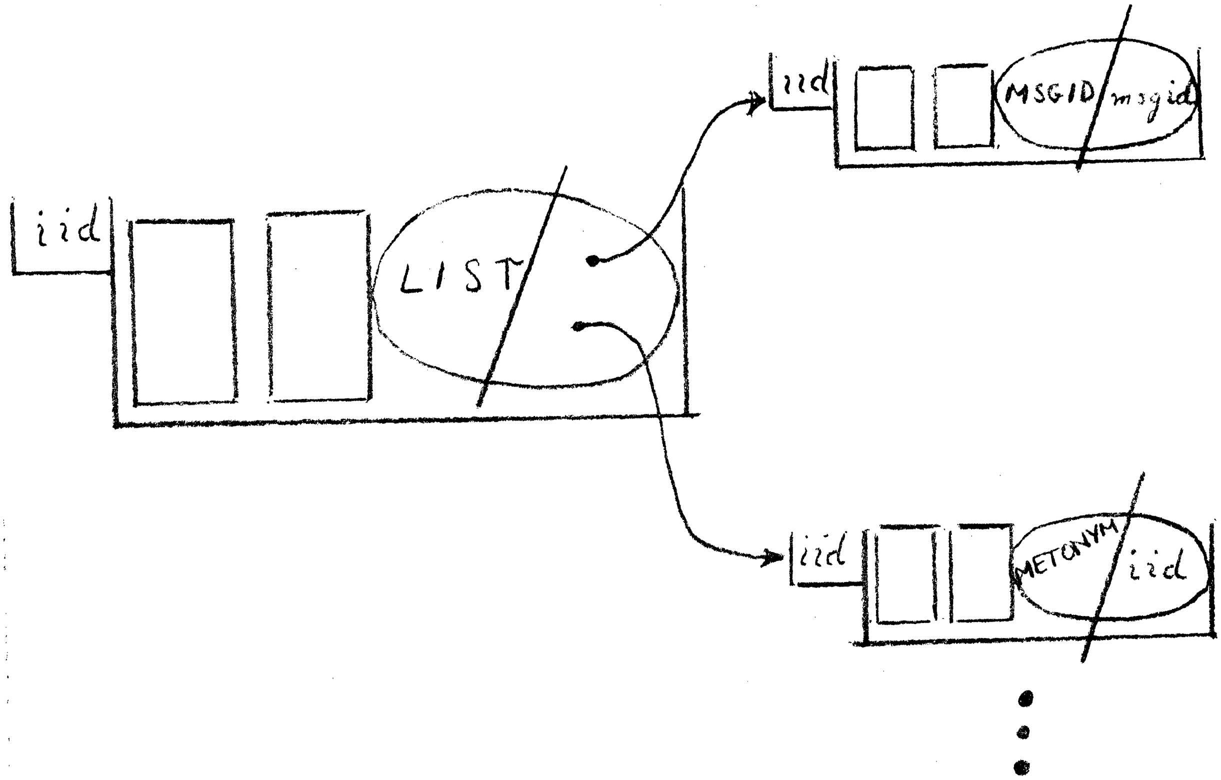


Figure 5.2.5-2: Format of a Response

Chapter 5.3

KEY PROCESSING ACTIVITIES

5.3.1 Introduction

The definition of the basic mechanisms of the queue manager and the definitions of the request and response processing activities of each access machine type is essentially a logical definition of SL. Certain access machine processing activities are especially important. Some of them are translation, expression evaluation and symbol resolution. The definition of expression evaluation is described is described below.

5.3.2 Expression Evaluation

Each reducible object will cause one QSLINT-type quasi-object to be spun off for the interpretation of all the statements of the syntactic group associated with the reducible object. Each QPARALLEL-type quasi-object will cause one QSLINT-type quasi-object to be spun off for the interpretation of each statement of the syntactic group associated with the QPARALLEL-type quasi-object. A QSLINT-type quasi-object is known as an interpreter.

Each QSLINT successively spins off one QEVAL-type quasi-object for each statement in the statement group it is processing. A QEVAL-type quasi-object is known as an evaluant.

Each QEVAL, not handling a simple operand, spins off a QEVAL-type quasi-object for each operand in the expression it is processing.

Access Machine of the QSLINT-type quasi-object (the interpreter)

1. /* Assume that the following parameters were passed to QSLINT if it were called by a reducible object:

    (1) the iid of an object construct which has a FUNCTION-type object /* this iid is in the acccess machine of the reducible object */. Located in the resource part of the FUNCTION-type object is the iid of an object construct which has a LIST-type object. This LIST-type object represents the statement group. Located in the resource part of this LIST-type object are the iid's of object constructs which represent statements. A statement may either be a

simple operand (SYMBOLREFERENCE-type object) or a complex operand (LIST-type object). A complex operand is a LIST-type object representing an operator and its operands. Located in the resource part of a LIST-type object, representing such a complex operand, is the iid of an object construct which has a SYMBOLREFERENCE-type object. The resource part of this SYMBOLREFERENCE-type object contains a symbol number. This SYMBOLREFERENCE-type object represents the operator. Also located in the resource part of a LIST-type object, representing a complex operand, are the iid's of object constructs representing the arguments to the function. These object constructs can have a SYMBOLREFERENCE-type object or a LIST-type object.

(2) the iid of an object construct which has an UNDEF-type object /* this iid is in the access machine of the reducible object */. This UNDEF-type object represents the interpreter workarea (IWA) which is part of the PSR.

(3) the iid of the object construct representing the storage used by the SL program being interpreted /* this iid is in the resource part of the reducible object */.

(4) the iid of the object construct representing the actual arguments intended for the function.

Assume that the following parameters were passed to QSLINT if it were called by a QPARALLEL-type quasi-object:

(1) the iid of an object construct which has a LIST-type object representing a nested statement group

(2) the iid of an object construct which has an UNDEF-type object /* this iid is in the resource part of a LIST-type object representing the interpreter workarea of the predecessor interpreter */. This UNDEF-type object represents a nested interpreter workarea (IWA).

(3) the iid of the object construct representing the storage used by the SL program being interpreted.

(4) the iid of the object construct representing the actual parameters intended for the function. */

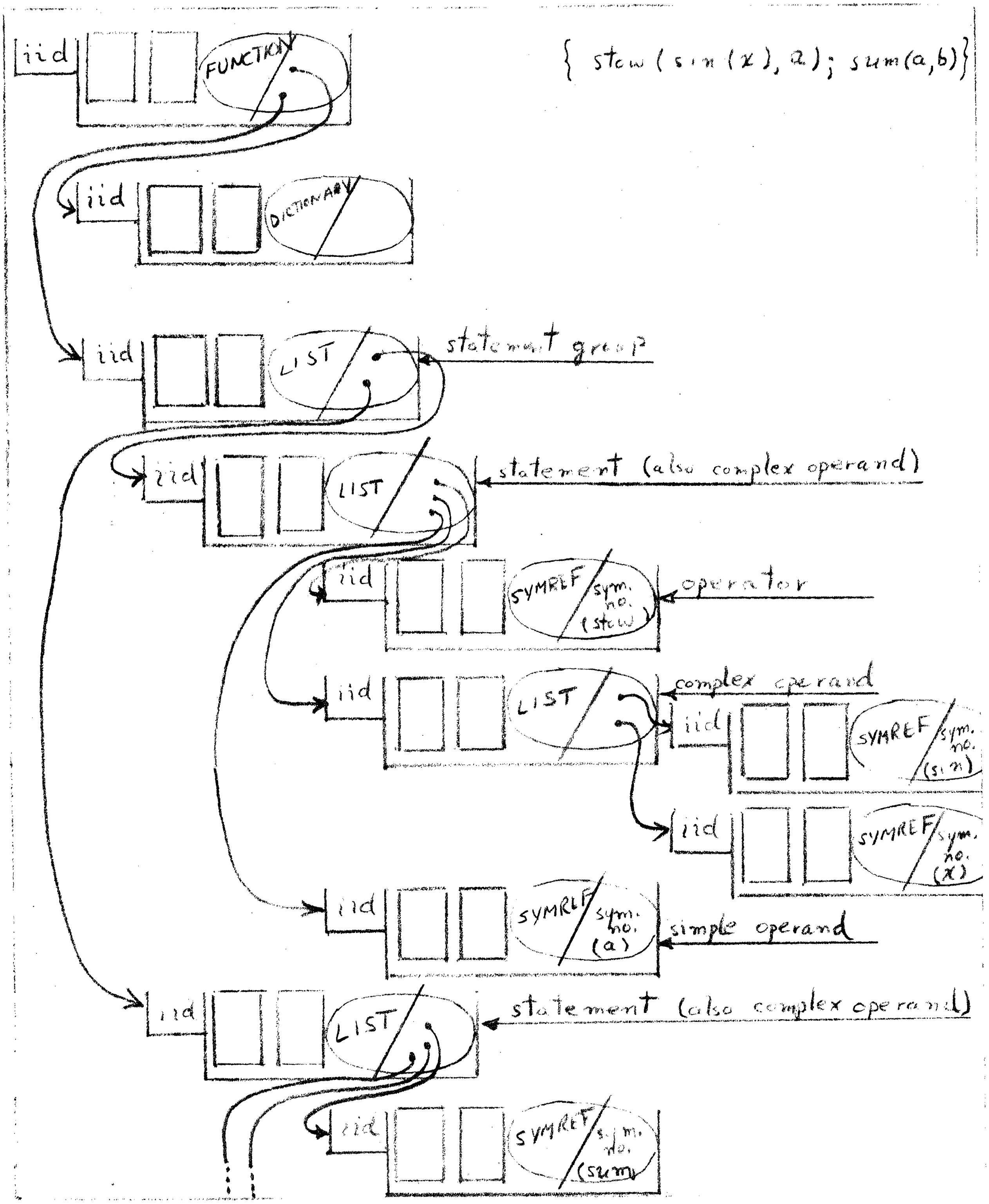


Figure 5.3.2-1:Structure of a Sample Function

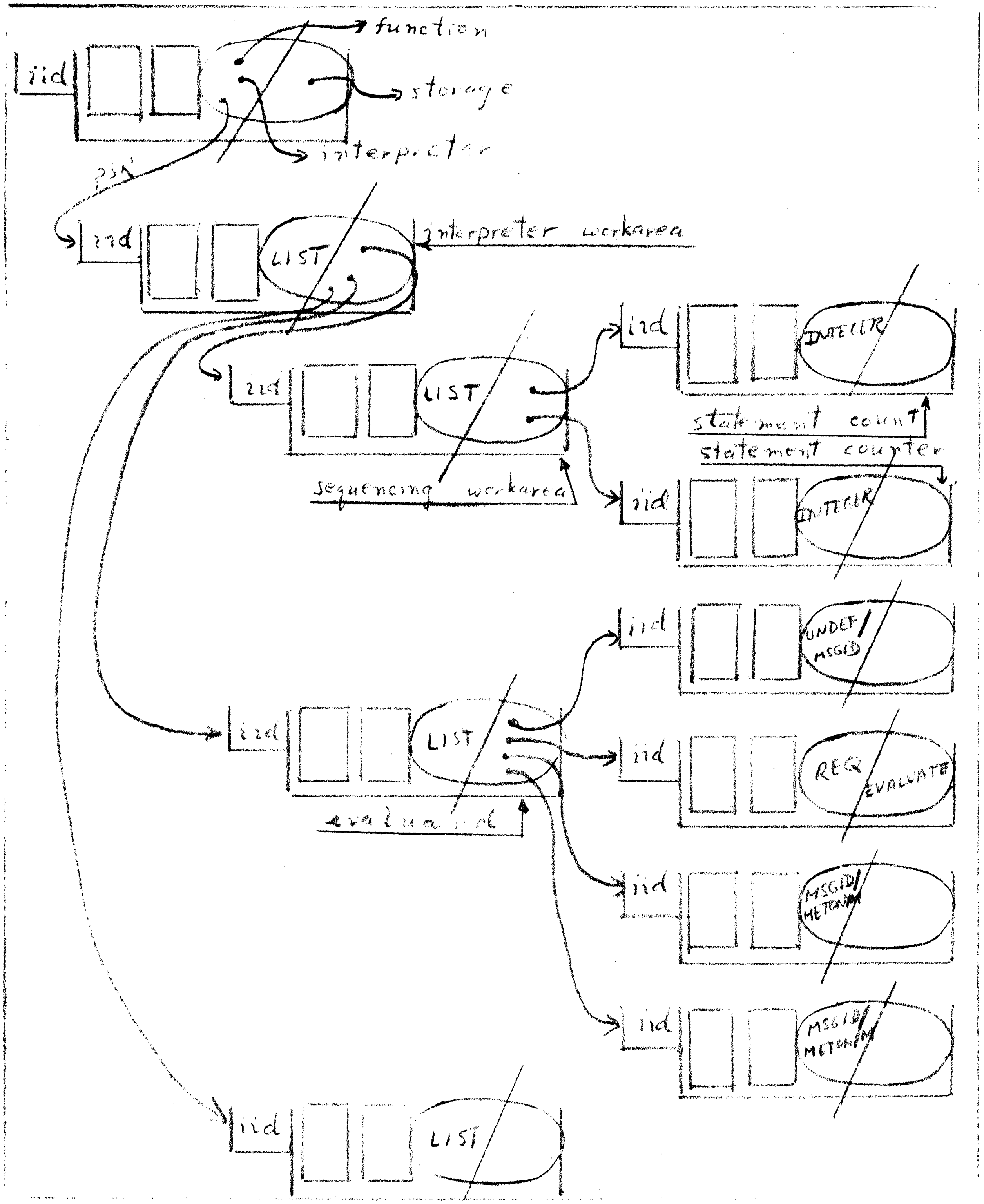


Figure 5.3.2-2:Structure of a Sample PSR

2. replaces the UNDEF-type object representing an IWA with a LIST-type object. This LIST-type object represents the (nested) interpreter workarea.

3. augments this LIST-type object, thus creating an UNDEF-type object.

4. replaces the UNDEF-type object with a LIST-type object. This LIST-type object represents the sequencing workarea.

5. augments this LIST-type object twice, thus creating two UNDEF-type objects.

6. replaces each UNDEF-type object with an INTEGER-type object. The first INTEGER-type represents the statement counter. The second INTEGER-type object represents the statement count.

7. if it were passed the iid of an object construct which has a FUNCTION-type object, it retrieves the LIST-type object representing a statement group; else it was passed the iid of a LIST-type object representing a (nested) statement group.

8. uses the request format convention and the send request mechanism to send an identify request to the LIST-type object representing the statement group. It needs to know the number of statements it is to interpret.

9. uses the wait mechanism to wait for the response.

Access Machine of the LIST-type object representing the statement group

10. uses the read request mechanism to read the identify request

11. /* Details of how a LIST-type object processes the identify request are not described now*/

12. uses the response format convention and the send response mechanism to pass back a response to the QSLINT-type quasi-object The response indicates the number of statements to be interpreted by the interpreter.

13. uses the wait mechanism to wait for the next request.

Access Machine of the QSLINT-type quasi-object (the interpreter)

14. uses the read response mechanism to read the response.

15. stores the number of statements in the resource part of the INTEGER-type object representing the statement count.

16. stores zero in the resource part of the INTEGER-type object

representing the statement counter.

17. if it were passed the iid of an object construct which has a FUNCTION-type object, it binds the parameters and handles the prologue if any.

18. creates a quasi-object construct with a QEVAL-type quasi-object.

19. augments the (nested) IWA, thus creating an UNDEF type object. This object will represent the evaluand.

20. uses the request format convention and the send request mechanism to send a start request to the QEVAL-type quasi-object just created. The parameters to start are the iid of an object construct which has a LIST-type object representing a (nested) statement, the iid of an object construct which has a LIST-type object representing the symbol table, the iid of the (nested) IWA just created, and the iid of the object construct representing storage.

21. uses the wait mechanism to wait for a response.

Access Machine of the QEVAL-type quasi-object (the evaluant)

22. use the read request mechanism to read the start request.

23. if it were passed an SYMBOLREFERENCE-type object representing a simple operand reference, it performs steps 24-33. If it were passed a LIST-type object representing a complex operand, it performs steps 34-65.

If the evaluant were passed an SYMBOLREFERENCE-type object representing a simple operand, then it -

24. uses the symbol resolution mechanism to locate the iid of the storage cell of the object construct represented by the simple operand. The symbol number in the resource part of the SYMBOLREFERENCE-type object indicates the symbol table entry which corresponds to the simple operand.

25. uses the request format convention and the send request mechanism to send an authorize request to the object construct just located. It wants a pointer to the object construct represented by the simple operand.

26. uses the wait mechanism to wait for a response.

Access Machine of the object construct just located

27. uses the read request mechanism to read the authorize request.

28. /* Details of how the simple operand processes an authorize request are not described now */

29. uses the response format convention and the send response mechanism to pass back a response to the QEVAL-type quasi-object. The response indicates the iid of an object construct which has an METONYM-type object.

30. uses the wait mechanism to wait for a request.

Access Machine of the QEVAL-type quasi-object (the evaluant)

31. uses the read response mechanism to read the response.

32. uses the send response mechanism to pass back a response to the interpreter (QSLINT) or evaluant (QEVAL) that invoked it.

33. destroys itself.

If the evaluant was passed a LIST-type object representing a complex operand, then it -

34. replaces the UNDEF-type object representing the evaluand with a LIST-type object. This LIST-type object represents the evaluand.

35. augments this LIST-type object twice, thus creating two UNDEF-type objects.

36. replaces the second UNDEF-type object with a REQUEST-type object whose resource part contains the name evaluate.

37. uses the symbol resolution mechanism to locate the iid of the storage cell of the object construct represented by the operator. The symbol number in the resource part of the SYMBOLREFERENCE-type object indicates the symbol table entry which corresponds to the operator.

38. uses the request format convention and the send request mechanism to send an identify request to the object construct just located. It must know if the object construct just located represents a function and if so, if the number of operands syntactically supplied is equal to the number of actual parameters semantically required by the function.

39. uses the wait mechanism to wait for a response.

Access Machine of the object construct just located

40. uses the read request mechanism to read the identify request.

41. /*Details of how the object processes the identify request are not described now */

42. uses the response format convention and the send response mechanism to pass back a response to the QEAUL-type quasi-object. The response indicates whether or not an evaluate request will be processed and the number of semantically required parameters.

43. uses the wait mechanism to wait for a request.

Access Machine of the QEVAL-type quasi-object (the evaluant)

44. uses the read response mechanism to read the response.

45. uses the request format convention and send request mechanism to send an identify request to the LIST-type object, representing the complex operand, sent to it as a parameter. It wants to know the number of actual parameters syntactically supplied.

46. uses the wait mechanism to wait for a response.

Access Machine of the LIST-type object

47. uses the read request mechanism to read the identify request.

48. /* Details of how the object processes the identify request are not described now*/

49. uses the response format convention and the send response mechanism to pass back a response to the QEVAL-type quasi-object. The response indicates the number of subobjects augmented from this LIST-type object.

50. uses the wait mechanism to wait for a request.

Access Machine of the QEVAL-type quasi-object (the evaluant)

51. uses the read response mechanism to read the response.

52. subtracts one from the number sent back in this response and verifies that the number of syntactically supplied parameters equals the number of semantically required parameters.

53. for each parameter it creates a quasi-object construct with a QEVAL-type quasi-object; it augments the (nested) IWA, thus creating an UNDEF-type object representing an evaluand; and it uses the request format convention and the send request mechanism to send a start request to the QEVAL-type quasi-object just created. The parameters to start indicate the expression to be

interpreted, the symbol table, the (nested) IWA just created, and the storage. For each parameter it augments the LIST-type object, representing its evaluand, thus creating UNDEF-type objects; and it replaces these UNDEF-type objects with MSGID-type objects whose resource part contains the msgids of the various start requests. /* The order of the MSGID-type objects in the evaluand reflect the order in which parameters will be passed to the function*/

54. uses the wait mechanism to wait for a response.

Access Machine of QEVAL-type quasi-object

55. uses the read response mechanism to read the response.

56. uses the msgid of the response to locate the appropriate MSGID-type object in its evaluand.

57. replaces the MSGID-type object with the object whose iid was passed back in the response.

58. deletes from the LIST-type object representing its IWA, the LIST-type object representing the evaluand of the evaluant which just returned the response.

59. determines if its evaluand contains any outstanding messages. If it does, it uses the wait mechanism to wait for a response, and repeats steps 55-59 as necessary

/* If individual operand evaluation should be done in sequence rather than in parallel, the evaluant performs all the steps 53-59 for each operand */

60. uses the send request mechanism to send the evaluate request to the object construct represented by the operator located via the symbol resolution mechanism in step 37. /* The request format convention was adhered to in the construction of this request, since the evaluant built up the request in the evaluand.*/

61. uses the wait mechanism to wait for a response.

62. /* Details of how a function processes its parameters are not described here -- see scenarios 1 and 2 */

63. uses the read response mechanism to read the response.

64. uses the send response mechanism to pass back the response to the interpreter (QSLINT) or evaluant (QEVAL) that invoked it.

65. destroys itself

Access Machine of a QSLINT-type quasi-object

66. uses the read response mechanism to read the response.

67. deletes from the LIST-type object representing his IWA, the LIST-type object representing the evaluand of the evaluant that just returned the response.

68. determines if there are more statements in the group to be processed by comparing the statement count with the statement counter. If there are, it adds one to the statement counter, and goes back to step 18.

69. uses the send response mechanism to pass back a response either to the reducible object or the QPARALLEL-type quasi-object that called it.

70. destroys his IWA

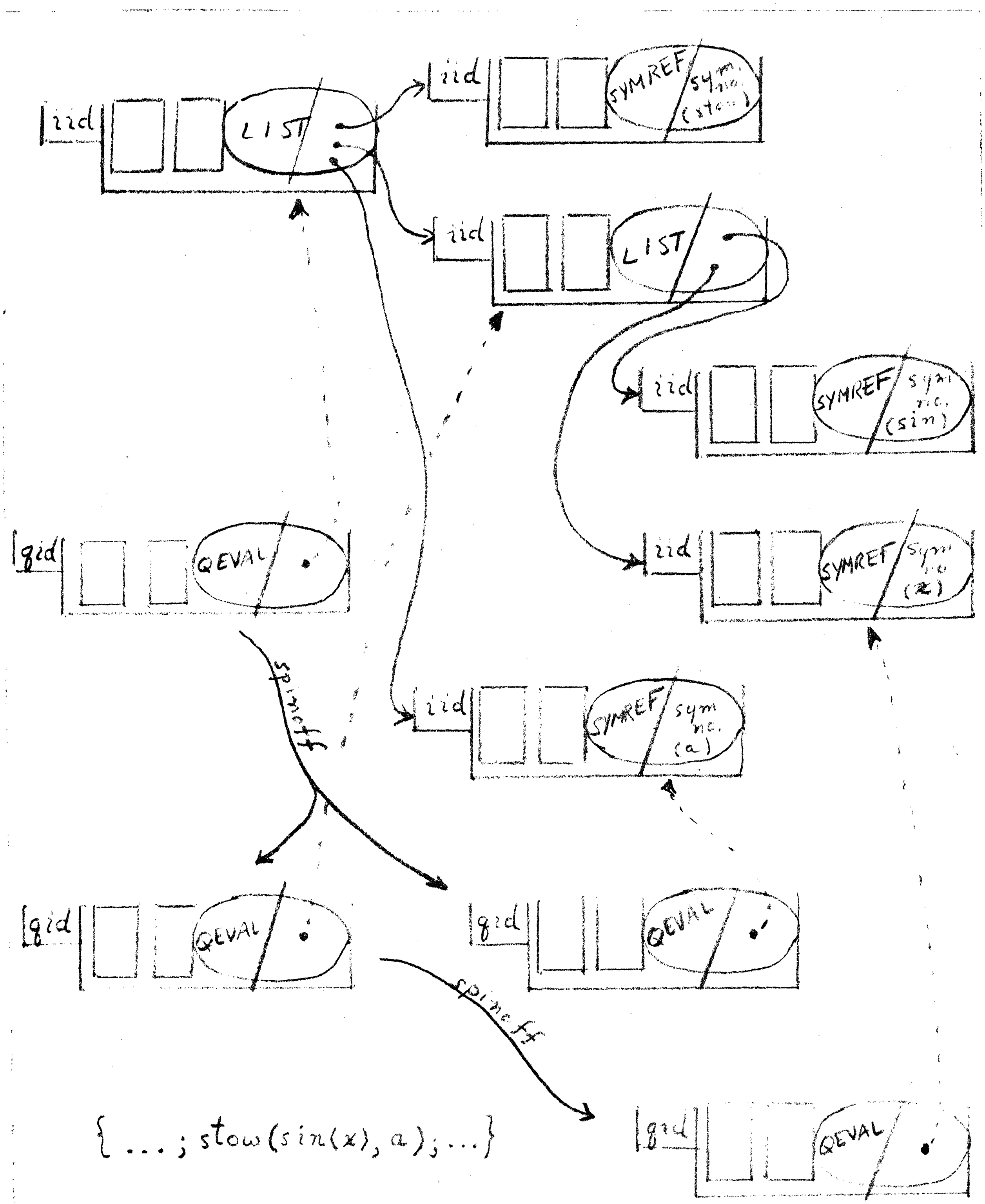


Figure 5.3.2-3:QEVAL Spinoff for Subexpressions

Chapter 5.4

SCENARIOS

5.4.1 Introduction

The scenarios are examples chosen to tie together ideas presented under Basic Structure, Basic Mechanisms, and Key Processing Activities.

5.4.2 Using a Primitive Syntactic Operator

```
{...;sum(a,b);...}
```

Expression Evaluation

1. /* Assume that the expression evaluation mechanism has reached the point where it is ready to invoke the sum function, passing it the evaluated simple operands a and b as actual parameters */

2. uses the send request mechanism to send the evaluate request to the object construct named sum which was located via the symbol resolution mechanism. The parameters to evaluate are the iid's of the object constructs named a and b.

3. uses the wait mechanism to wait for a response.

Access Machine of the PFUNCTION-type object (the sum function)

4. uses the read request mechanism to read the evaluate request.

5. creates a quasi-object construct with a QSUM-type quasi-object.

6. uses the request format convention and the forward request mechanism, /* no new msgid */, to forward a start request to the QSUM-type quasi-object just created. The parameters to start are the iid's of the object constructs named a and b.

7. uses the wait mechanism to wait for the next request. /* The PFUNCTION-type object is completely severed from the QSUM type quasi-object*/

Access Machine of the QSUM type quasi-object

8. uses the read request mechanism to read the start request.

9. /* Details of precisely how QSUM does the addition of a and b are not described now */

10. uses the response format convention and the send response mechanism to pass back a response to the expression evaluation. The response consists of the iid of the object construct which represents the result of adding a and b.

11. destroys itself

Expression Evaluation

12. uses the read response mechanism to read the response.

13. /* Refer to the expression evaluation mechanism for details of response handling */

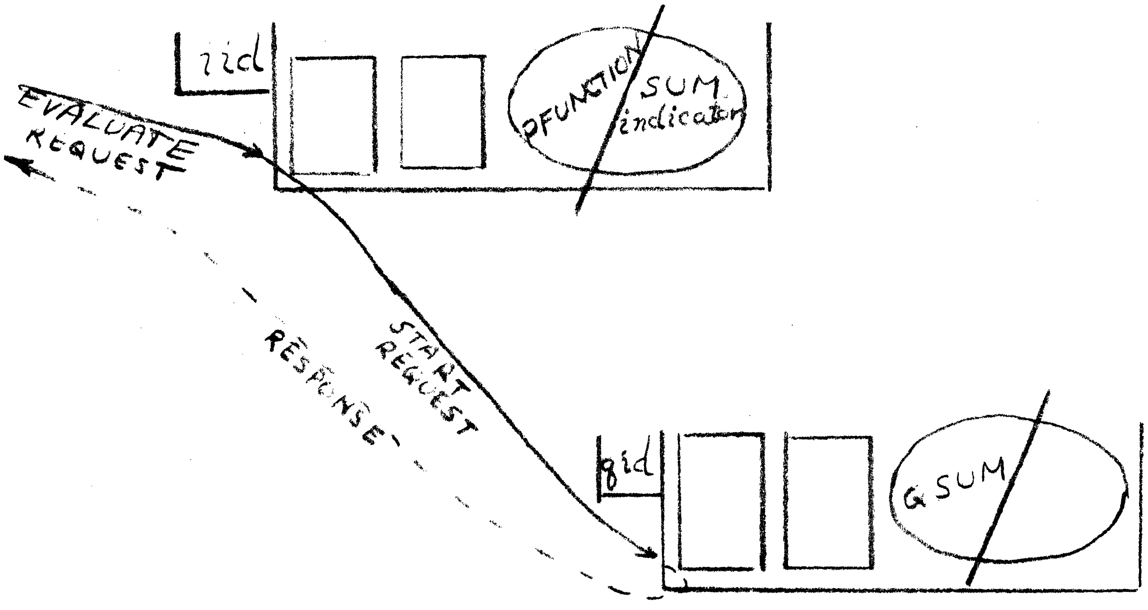


Figure 5.4.2-1:Primitive Operator Flow

5.4.3 Using a Reducible Syntactic Operator

{...;sin(x);...}

Expression Evaluation

1. /* Assume that the expression evaluation mechanism has reached the point where it is ready to invoke the function sin, passing it the evaluated simple operand x as an actual parameter */

2. uses the send request mechanism to send the evaluate request to the object construct named sin which was located via the symbol resolution mechanism. The parameter to evaluate is the iid of the object construct named x.

3. uses the wait mechanism to wait for a response.

Access Machine of the FUNCTION-type object (the sin function)

4. uses the read request mechanism to read the evaluate request.

5. creates an object construct with an UNDEF-type object.

6. replaces the UNDEF-type object with an object whose access machine contains three iid's: the iid of the object construct named sin which has a FUNCTION-type object /* the SL interpreter will need access to the SL text and symbol table */; the iid of the object construct named slint which has a PFUNCTION-type object /* this PFUNCTION-type object will spin off an SL interpreter */; and the iid of an object construct which has an UNDEF-type object /* this is the PSR and will be used by the SL interpreter for its workspace */. Such an object is a reducible object.

7. uses the request format convention and the send request mechanism to send a start request to the reducible object just created. The parameter to start is the iid of the object construct named x.

8. adds an entry to its Outstanding Activation Table. The entry contains the msgid of the evaluate request just processed, the msgid of the start request just sent to the reducible object, and the iid of the reducible object. /* Each FUNCTION-type object must keep a record of all spun-off reducible objects still active so that it can block change requests (a request to change the SL text) until all spun-off reducible objects have terminated or suspended */.

9. uses the wait mechanism to wait for the next request or response. /* The FUNCTION-type object is effectively severed from

the reducible object since the FUNCTION-type object may now process new requests or replies. */

Queue Manager of the Reducible Object

10. uses the read request mechanism to read the start request.

11. uses the request format convention and the send request mechanism to send an evaluate request to the object construct named slint which was located via the iid in the access machine of the reducible object. The parameters to evaluate are the iid of the object construct named sin which has a FUNCTION-type object/* this iid is in the access machine of the reducible object*/; the iid of an object construct which has an UNDEF-type object /* this iid is in the access machine of the reducible object */; the iid of the object construct used for storage by the interpreted SL program /* this iid is in the resource part of the reducible object */; and the iid of the request sent to it by the FUNCTION-type object (the sin function) /* this request contains a start request type and the iid of the object construct named x */. /* The queue manager associated with a reducible object always packages up the requests sent to it and sends them on without examination for their interpretation by SL text */

12. uses the wait mechanism to wait for a response.

Access Machine of the PFUNCTION-type object (the SLINT function)

13. uses the read request mechanism to read the evaluate request.

14. creates a quasi-object construct with a QSLINT-type quasi-object.

15. uses the request format convention and the forward request mechanism to send a start request to the QSLINT-type quasi-object just created. The parameters to start are identical to the parameters of the evaluate request discussed in step 11.

16. uses the wait mechanism to wait for the next request. /* The PFUNCTION-type object is completely severed from the QSLINT type quasi-object */.

Access Machine of the QSLINT-type quasi-object

17. uses the read request mechanism to read the start request.

18. since the object representing the process status record (PSR) is an UNDEF-type object (i.e. it is initialized), the QSLINT-type quasi-object knows that it is not resuming a suspended interpretation, but is beginning a new interpretation. Therefore, it binds the parameters intended for processing by SL

code, and it augments the storage named in the resource part of the reducible object. It binds the parameter, the iid of the object construct named x, as follows: it locates the symbol number of the formal parameter in the SL symbol table. The property of being a formal parameter has been associated with the symbol number. It then locates the SL link table entry using the symbol number as offset, and inserts the iid of the object construct named x into the iid slot of the entry.

19. /* Details of interpreting SL text representing the sin operation are not described now. Refer to the expression evaluation mechanism for details on interpreting SL text */.

20. uses the response format convention and the send response mechanism to pass back a response to the reducible object. The response consists of the iid of the object construct computed by the interpretation of the SL text representing the sin function.

21. destroys itself

Queue Manager of the reducible object

22. uses the read response mechanism to read the response.

23. uses the send response mechanism to pass back the response to the FUNCTION-type object (the sin function). The response consists of the iid of the object construct computed by the interpretation of the SL text representing the sin function.

24. since the PSR indicates that an SL return function had been interpreted, it destroys itself.

Access Machine of the FUNCTION-type object (the sin function)

25. uses the read response mechanism to read the response.

26. uses the msgid in the first subobject of the LIST-type object representing the response to search the Outstanding Activation Table for the appropriate entry, retrieves the msgid of the original evaluate request for use in step 27 and deletes the entry.

27. uses the send response mechanism to pass back the response to expression evaluation. The response consists of the iid of the object construct computed by the interpretation of the SL text represented by the sin operator.

28. uses the wait mechanism to wait for the next request or response.

Expression Evaluation

29. uses the read response mechanism to read the response.

30. /* Refer to the expression evaluation mechanism for details of response handling */.

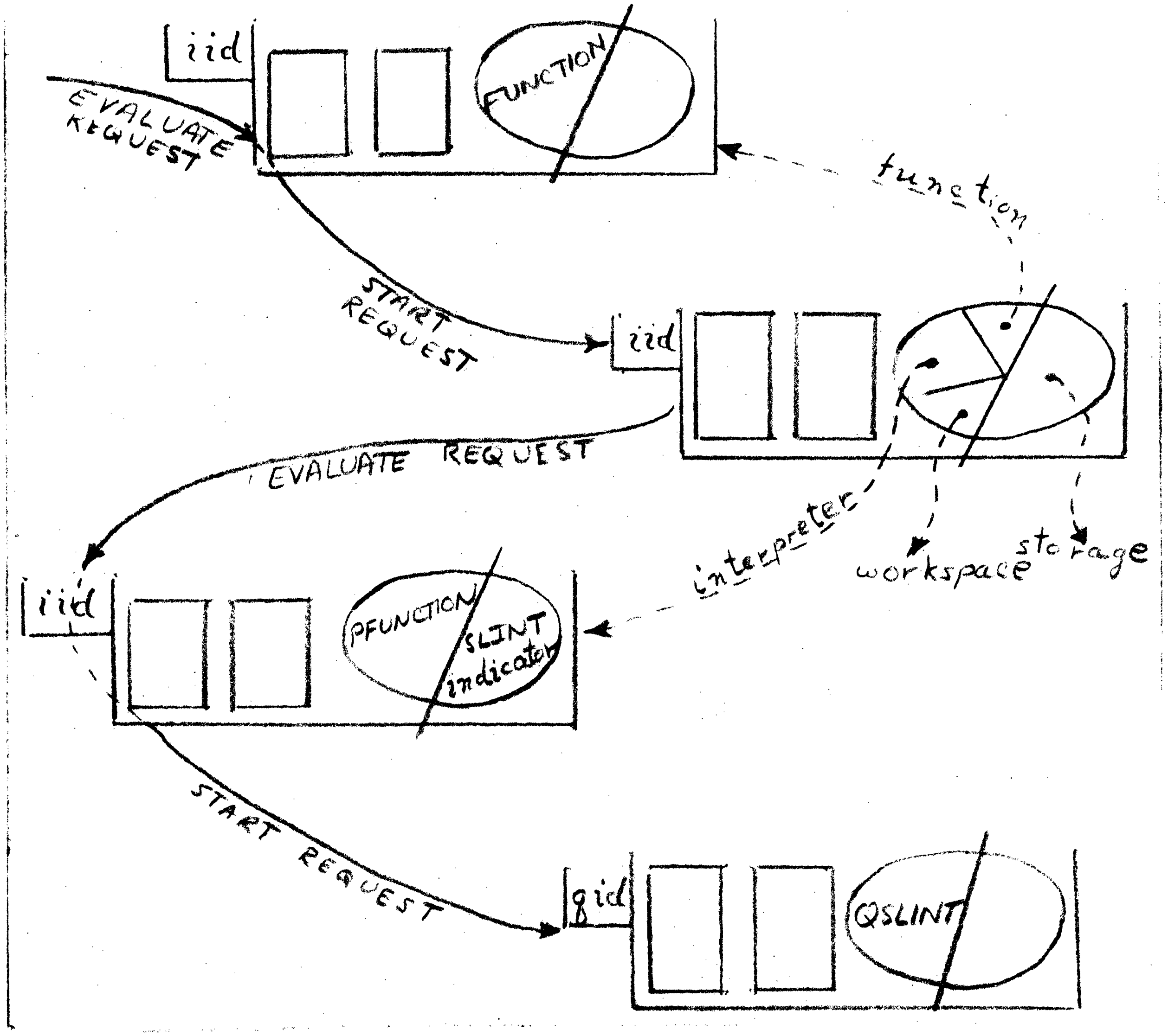


Figure 5.4.2-2:Reducible Operator Flow

Appendix 1

GLOSSARY

The following words and phrases include terms formally defined in the logical architecture together with important terms in the informal discussions. Words beginning with lower case letters are built-in objects, either constants or functions. Numbers in parentheses indicate the section in which the term is defined. The letters (GT) indicate terms from graph theory.

Access machine (2.1.3) The active part of an object that responds to requests upon the object.

Accessibility graph (2.1.5) A graph of all paths for accessing objects. It has two major subgraphs: the ownership tree and the chains of synonyms.

Accessible (2.1.5) An object x is accessible from y if there is a path in the accessibility graph from y to x.

Activation tree (2.2.5) A tree linking activations of functions to the activations of functions they called. It is a subgraph of the dependency graph.

Admissible index set (2.1.5) A set of objects admissible as indices to the access machine of a collective object.

Argument (2.2.5) The result of evaluating an operand for a function.

Assignment (2.1.4) An informal term for referring to the stow and replace functions.

authorize (2.1.5) A dyadic function that makes an authorize request upon an object in order to obtain a synonym to the object with a given set of rights.

Buffer (2.1.1) A temporary storage cell used for holding an object or shipping it somewhere else.

Cell name (2.1.1) An identifier that uniquely specifies a storage cell.

Chain (GT) A graph whose edges define a strict linear ordering of the vertices. It is both a tree and a rooted tree.

Circuit (GT) A path whose first and last vertices are identical.

Collective object (2.1.5) An object that owns storage cells containing other objects.

Connected graph (GT) A graph in which for any two vertices x and y, there exists an undirected path from x to y.

create (2.1.4) A dyadic function that creates a new object by activating an access machine and providing it with initial values for its owned resource.

Deadlock (2.5.1) A state of the system in which a set of queued requests can never be resolved. It results from a circuit in the dependency graph.

delete (2.1.6) A dyadic function that deletes storage cells from the owned resource of a collective object.

Dependency graph (2.1.3) A graph of outstanding requests upon objects: if x is waiting for a request on y, then (x,y) is an edge of the dependency graph.

Descriptor (2.1.3) An implementation defined representation of an access machine: it contains a PSR and specifies the interpreter and procedural description.

Dictionary (2.2.2) For each module, the dictionary maintains information about all symbols: character representation, linkage, and initial attributes.

Directly accessible (2.1.5) An object x is directly accessible from y if there is a path in the ownership tree from y to x.

Edge (GT) An ordered pair of vertices in a graph.

Element (2.1.5) An object residing in a storage cell owned by a collective object.

Elementary symbol (2.2.3) A symbol in program text without any syntactically associated operands.

Elementary object (2.1.5) An object that does not own any storage cells; all elementary objects are scalars.

Environment tree (2.3.3) A rooted tree that defines search paths for symbol resolution.

evaluate (2.1.4) A monadic function that makes an evaluate request on its argument to deliver or generate a value.

Exception (2.4.1) A response by an access machine indicating that the normal response cannot be made.

Extended syntax (1.3.3) An infix notation that includes macro facilities to be mapped into strict syntax.

Forest (GT) A graph consisting of one or more unconnected trees.

Function (2.1.4) An object that responds to evaluate requests by creating an activation that computes an object as result.

Generator (2.1.7) A collective object whose elements are computed upon demand instead of being stored in the SMS.

Graph (GT) A set of points called vertices and of ordered pairs of vertices called edges. Only directed graphs are used in the discussion.

Group (2.2.3) A list of statements enclosed in braces. A group is the external form of a module.

identity (2.1.4) A monadic function that asks an object to identify its access machine.

ilist (2.1.5) A monadic function that returns the index set of a collective object.

Incoming edge (GT) An edge (x,y) is an incoming edge with respect to the vertex y.

Index set (2.1.5) The set of objects mapped by select requests onto storage cells of a collective object.

Indirectly accessible (2.1.5) An object x is indirectly accessible from y if there is a chain of synonyms from y to x.

insert (2.1.6) A dyadic function that inserts new storage cells in the owned resource of a collective object.

Interpreter (2.1.2) The motive force behind a process: it examines the PSR, decodes the procedural description, and puts the PSR in its next state.

lambda (2.2.3) A function that creates a new function by binding formal parameters to a module.

List (2.1.5) The most primitive type of collective object. Its elements are indexed by consecutive integers starting at 0 and may be of different types.

Metonym (2.1.5) An encapsulated synonym. It is used for pointers in PL/I to conform to restrictions in the language definition.

Module (2.2.2) The machine form of a group: it contains the text for the group together with a dictionary of all symbols in the group.

nil (2.1.3) A primitive object that has the properties of a zero element list.

Object (2.1.3) Basic entity in the system; it has an active part called an access machine and a passive part called an owned resource.

Object base (2.1.3) Set of all objects in the system.

Object image (2.1.3) An internal representation of an object: it contains the descriptor of its access machine and a representation of the owned resource.

Offset (2.1.1) A displacement from the beginning of a table. This term is not a formal part of the definition.

Operand (2.2.3) An expression in program text that evaluates to an argument for a function.

Operator symbol (2.2.3) A symbol that resolves to a function and that has syntactically associated operands.

Outgoing edge (GT) An edge (x,y) is an outgoing edge with respect to the vertex x.

Owned resource (2.1.3) Passive part of an object that is managed by the access machine.

Ownership tree (2.1.5) A tree defined over the object base by the ownership relation between collective objects and storage cells.

parallel (2.2.5) A monadic function that causes the statements of a module to be executed in parallel.

Parameter (2.2.3) A symbol local to a module that is resolved to an argument every time the module is activated.

Path (GT) A sequence of vertices of a graph G such that if x and y are adjacent vertices, (x,y) is an edge of G.

Port (2.1.3) An object whose access machine and resource connect to a data path through the Source-Sink Subsystem (see the System Architecture Manual).

Primitive object (2.1.3) An object that cannot be constructed from other objects defined in the logical architecture.

Procedural description (2.1.2) Encoded information that defines the states of a process and permissible state transitions.

Process (2.1.2) An automaton that has three parts: a process status record (PSR), a procedural description, and an interpreter.

Process status record (2.1.2) The record of the current state of a process, its input, and its working storage.

Program text (2.2.3) A string of symbols.

PSR (2.1.2) Abbreviation for process status record.

quote (2.2.3) A syntactic marker that suppresses automatic evaluation of a function.

Ready state (2.1.3) State of an access machine when it is ready to respond to a request.

Reducible object (2.1.3) An object that can be constructed from more primitive objects in the logical architecture.

remove (2.1.6) A monadic function that removes an object from a storage cell without deleting the cell.

replace (2.1.6) A dyadic function used for assignments that replace the target completely.

Request (2.1.3) A pair of parameters passed to an object to request some service.

Reserved word (1.3.4) A string of two or more lower case letters used to designate system defined objects and various constructions in the extended syntax.

Resource manager (2.5.3) The object in a subsystem that obtains rights to objects outside of the subsystem and allocates the rights to other objects within it.

Rights (2.1.5) A set of requests that a synonym passes on to the object it points tc.

Root (GT) The distinguished vertex of either a tree or a rooted tree.

Rooted tree (GT) A connected graph in which there is a distinguished vertex with no outgoing edges and all other vertices have exactly one outgoing edge.

Seed (GT) A tree with one vertex and no edges.

select (2.1.5) A dyadic function that makes select requests on a collective object to map indices onto storage cells.

Sequential synonym (2.1.8) A synonym that can be sequenced through successive elements of a collective object.

SMS (2.1.1) Abbreviation for the Storage Management Subsystem (see the System Architecture Manual).

Space number (2.1.1) A number identifying a logical space in the SMS. This term refers to the implementation rather than to the formal definition.

Statement (2.2.3) A complete expression used as one element of a module.

Storage cell (2.1.1) A logical location large enough to contain any object.

stow (2.1.4) A dyadic function that makes a stow request on the target to perform assignments. It makes a less drastic change than the replace function.

Strict syntax (1.3.2) A prefix notation that is mapped one-to-one into the internal machine code.

Strongly connected graph (GT) A graph in which for any two vertices x and y, there exists a path from x to y.

Structure (2.1.7) A subtree of the ownership tree together with all objects accessible from objects in the tree.

Subsystem (2.5.3) A subset of the object base having only one point of connection with the graphs linking the rest of the system.

Symbol (2.2.3) A string of one or more characters.

Symbol resolution (2.2.1) The act of resolving symbols to cell names of storage cells containing objects.

syn (2.1.5) A monadic function that makes an authorize request to obtain a synonym that responds to copy and destroy requests itself.

Synonym (2.1.5) An object that automatically passes requests to the object whose storage cell it names.

System root (2.1.5) The object at the root of the ownership tree;

all objects in the system are directly accessible from the system root.

Tree (GT) A connected graph in which there is a distinguished vertex with no incoming edges, and all other vertices have exactly one incoming edge.

under (2.1.3) A primitive undefined object.

Undirected path (GT) A sequence of vertices of a graph G such that if x and y are adjacent vertices, then either (x,y) or (y,x) is an edge of G.

Vertex (GT) A point on a graph.

SDD - Poughkeepsie
D/B11, B/706-2
Extension 3-2589

March 15, 1971

Memorandum to: Recipients of Advanced Future System Proposal

Subject: Index to SL Report

Enclosed is an index to the "Fundamental Concepts and System Language" Report. Page numbers correspond to the third edition, dated March 8, 1971.

John F. Sowa

John F. Sowa

JFS:dc

Enclosure

INDEX