COO-614-59

ICR QUARTERLY REPORT

Number 15
November 1, 1967

THE UNIVERSITY OF CHICAGO

THE INSTITUTE FOR COMPUTER RESEARCH

DISCLAIMER

COO-614-59

The Institute for Computer Research

The University of Chicago

ICR Quarterly Report

Number 15
November 1, 1967

Prepared under the auspices of AEC Contract No. AT(11-1)-614.

Victor H. Yngve, Principal Investigator

Staff of the Institute for Computer Research

Naomi Alton
Robert Ashenhurst
Jurgen Bounin
Dolores Brennan
William Brown
Kenneth Burns
Rachel Chakrin
Minja Choe
Elaine Chow
Charles Fischer
Etta Franklin
William Georgas
Robert Graves
Richard Herwitt
Leonardo Herzenberg
Thea Hodge
Jane Jodeit
Paul Kosinski
Hirondo Kuki
George Kreisheimer
Vincent Kruskal
Glenn Manacher
Richard Miller
Lily Monheit
John Munn
Daniel Norman
Alex Orden
Walter Petryshyn
Charles Robinson
Clemens Roothaan
Lyle Settle
John Shepherd
Cheng-kiat Teng
Serena Torres
Nancy Trombetta
Cherie Weil
Michael Williams
Augustine Witt
Victor Yngve

Table of Contents

PROJECT SUPPORT ACTIVITY

It has been customary in Section I-A to report briefly the various developmental activities of the Institute, mainly in the hardware area. In future Reports this will be expanded to cover software activity also, with a somewhat altered emphasis as indicated by the change of section title (from "Maniac III Engineering Activity"). It also has been customary in this section to include a log of Maniac III usage. Due to a late publication date Quarterly Report No. 14 contained this log to October 24; it will therefore be resumed from this date in the next Report.

THE MAN-MACHINE PROJECT*

C.C.J. Roothaan

Progress Report No. 11

(for the period 1 July 1967 through 30 September 1967)

System/360 Mathematical Subroutines

Mr. Kuki has almost finished upgrading the System/360 Math Function Library. While originally about a third of the library was expected to be affected by this work, nearly all of the library ended up being improved. The most significant general improvement deals with the problem of underflow. Intermediate underflow (i.e., temporary underflow which does not affect the answer in any way) is avoided and terminal underflow (i.e., the case when the answer itself is below the underflow threshold) causes a message to be printed if the underflow exception has been enabled. The above work was done by using the new "exploitation" code which was designed to take advantage of the proposed new floating point hardware. The Computation Center is evaluating the effect of this new hardware on the existing FORTRAN Function Library (which does not take advantage of the new hardware).

System/360 Data Management

Messrs. Kosinski, Kruskal, and Williams have been investigating in some detail the internal organization of OS/360 as well as the services it provides. These investigations indicate that it would be impractical, if not

*Work on this project, supported by a grant from the International Business Machines Corporation with C.C.J. Roothaan as principal investigator, is carried on under the joint auspices of the Institute and the Computation Center.

impossible, to incorporate a unified data management system within the current version of OS/360. In particular, the OS supervisor lacks adequate protection facilities to insure a really secure on-line file system.

It is thus contemplated that an entirely new set of supervisory and data management programs will be written to replace the present "primary control program" portion of OS/360. The supervisor would, of course, provide for multiprogramming. It would also incorporate some relatively new concepts in order to provide (hopefully) absolutely foolproof protection of the system against intentional or unintentional interference.

The data management program would provide the unified on-line file system described in another paper entitled Preliminary Proposal for Data Management Development Under the Man-Machine Contract by Paul Kosinski.

Some other goals of this new "control program" are: to provide compatibility interfaces so that it can be used exactly like the current OS control program and to be easily modifiable to take advantage of the additional hardware features of the 360 model 67. Also under consideration is the feasibility of designing the new supervisor in such a way as to be able to supervise other supervisors (including itself), i.e., to allow other operating systems to run under this one as nonprivileged user programs.

CHOP (CHicago Output Package)

In Progress Reports Nos. 8, 9, and 10, this routine was referred to as POOR (Page Oriented Output Routine). The name has since been changed to CHOP (CHicago Output Package).

7094 CHOP: The problem of making CHOP/7094 accessible to FORTRAN users is still being studied. Mr. Eicher has finished the writeup which is being distributed.

360 CHOP: CHOP/360 is being changed to allow the user to specify buffer size so that less than a full page buffer can be used for simpler applications.

CHAP (CHaos Assembly Program)

Miss Monheit was on vacation for six weeks. The assembler is finished. Test programs for debugging the loader have been inserted into CHAOS and debugging on the 7040 has begun.

CHAOS (Chicago Asynchronous Operations Scheduler)

As an interim measure until work is completed on the new re-entrant CHAOS operator request interpreter, the current request interpreter and task setup processor were modified to afford the 1401 operators at the Biological Sciences Computation Center the same query and command abilities available to 7094/7040 operators. The modifications also allow use of the 7040 console typewriter and entry keys as an additional inquiry station.

CHAOS disk file cataloguing techniques are being redesigned. The current cataloguing procedures are oriented toward "task-files" (e.g., job input, job printing, job punching). This "task" orientation is unsuitable for other types of files, such as intermediate files created by the CHAOS Utility Monitor. Uniform cataloguing procedures for all files are necessary to prevent costly redundancy in file catalogue servicing routines. This work has received additional impetus from a pilot project begun to test the feasilibility of allowing users to catalog and access their own 7040 disk files.

A USER'S VIEW OF CAPABILITIES

R. Fabry

ABSTRACT

There are various kinds of resources available to the users of the new ICR multicomputer system. One can use a resource if and only if one has a special address for it, called a capability.

All of a user's information in the system is partitioned into two types: data (including program) and capabilities. Capabilities can never be modified directly by the user.

The only resources accessed directly by machine instructions are core and input/output devices; the hardware must be able to interpret the corresponding capabilities. All other resources are accessed through the supervisor; these might include segments not in core, teletypes, processes, and charge accounts.

INTRODUCTION

The new ICR system will make different kinds of resources available to its users. Some kinds of resources are addressed directly by machine instructions, while other kinds are only addressed through the supervisor. A list of the kinds of resources discussed below includes:

(1) blocks of contiguous core memory words called core blocks;

(2) blocks of logically contiguous disk memory words called disk blocks;

(3) sets of linearly ordered information words called segments (the idea is that a core block or a disk block can physically hold a segment of information);

(4) input/output devices addressed by the hardware as a unit;

(5) individual teletypes (it happens that a teletype is not addressed by the hardware as a unit and is thus not included above);

(6) pieces of work to be or being performed by a processor, which are called processes (to be more specific, each process, at each point in time, is represented by a stateword which contains the information which must be placed into the various processor registers to properly execute the next instruction required to perform the work); and

(7) charge accounts to which the use of the system can be charged by a user of the system.

In a flexible multiprogramming environment a process not only uses resources as it calculates, it also makes calculations about its use of resources. One kind of calculation about the use of resources occurs because as a process runs, its requirements for resources expand and contract. The process is continually requesting and returning resources. It is this fluctuation which provides one of the reasons for multiprogramming: if a large number of processes are running simultaneously, the total amount of resources needed is a function of the average demand of each process instead of the peak demand.

Another kind of calculation about the use of resources occurs when one process is supervising another process. Of course one place such

supervision occurs is within the supervisor. It may also occur when a debugging program is used to supervise the execution of a program being checked out. The sort of calculation about the use of resources which goes on in this situation varies from discovering what resources the supervised process is using to monitoring and passing judgment on its use of resources at every step.

A third kind of calculation about the use of resources arises from the mechanics of sharing when several processes are cooperating as equals. Such cooperation might occur in game playing where each process passes information to the others or in a parallel search if the first process which finds something is supposed to stop the other processes. Even with more conventional problems, the work may be split into several processes in order to make better use of the available resources. This is one of the reasons for parallel processing. For example, consider some work which requires a certain amount of core memory for a certain time when programmed as a single process. If it can be reprogrammed as two processes which, if run together, run in almost half the time of the single process and which together use almost the same amount of memory as the single process, then the total processor time used is almost unchanged but the memory time, and hence the memory space-time, is almost cut in half.

These three cases bring up some interesting questions. How does a process deal with a varying requirement for resources? How does a process grant and retrieve resources when dealing with subordinate processes? How does a process share a resource with another process?

Dennis and Van Horn [1] explored these questions at an abstract, machine-independent level. By developing and unifying the ideas of a number of people, they arrived at an answer based on an entity which they called a capability. Their idea was that a capability is a special kind of an address for a resource. One can use a resource if and only if one has one of these special addresses for it; that is, a capability for it. A process with a varying requirement for resources simply has a varying set of capabilities. A process grants and retrieves resources when dealing with a subordinate process by providing and removing capabilities for the resources. A process can share a resource with another process by giving it a capability for the resource. For such a scheme to work, the user must not be able to create his own capabilities; capabilities must be created only by the programs which allocate resources, namely, the supervisor.

OUR USE OF CAPABILITIES

In the new ICR system, the resource allocation and protection mechanism is based on this idea of a capability. Every resource must be addressed by its capability in order to be used. (It should be pointed out that the concept of addressing via a capability which we use is not the same as that of Dennis and Van Horn. The differences will be examined in a later paper.)

All of a user's information in the system is partitioned into two types: data (including program) and capabilities. In the processor,

there is one set of registers for capabilities and one set for data. In core memory and on the disk there are two kinds of segments: data segments and capability segments. The user can modify data, but data can never cross the boundary and become a capability. Capabilities can never be modified directly by the user. Thus the user cannot create his own capabilities.

Capabilities are used to address many different kinds of resources. For protection, it is necessary that a capability for one resource, such as an input/output device, not be confused with a capability for another resource, such as a segment. For this reason, certain bits in the machine-representation of a capability unambiguously specify the type of resource while the rest specify a part of the resource.

Core blocks and input/output devices are accessed by the hardware during the execution of machine instructions. In all cases the instruction specifies, explicitly or implicitly, a capability which, in its turn, specifies an actual core block or input/output device. Because the hardware must make use of these two kinds of capabilities, one will find a discussion of them in the machine manual. All other kinds of capabilities are for resources which are accessed through the supervisor. In order to use these resources one must call an appropriate supervisor routine, either explicitly as a subroutine or implicitly by a trap. From the users point of view there is no real distinction between resources accessed directly and resources accessed through the supervisor. From the hardware point of view,

however, there is quite a distinction and the following terminology is used: a capability for a resource accessed directly is called a real capability and there are only two kinds; a capability for a resource accessed through the supervisor is called a pseudo capability. The hardware knows nothing about any pseudo capability.

TYPES OF CAPABILITIES

The remainder of this paper will discuss various types of capabilities. The actual types of pseudo capabilities which will be used by the system are not yet determined. The pseudo capability types which are included in the following list are meant to suggest the range of possibilities.

A core block capability specifies a contiguous block of core memory words which are used to contain a segment of information. In the machine representation of this capability the block of core is specified by its beginning address and its length. In addition to specifying the location of the block, this type of capability also specifies whether its contents may be used as capabilities or as data and whether the contents can be read and be modified.

One type of pseudo capability might be a disk block capability, which would specify a logically contiguous block of words on the disk which would be used to contain a segment of information. Such a capability would directly or indirectly specify the block's location on the disk. In addition it would specify whether the contents of the block

could be used as data or as capabilities and whether the contents could be read and be modified. In these respects, the disk block capability would be functionally identical to a core block capability.

It may well be that in our system a user does not have to worry about whether a segment is in memory or on the disk. This is the case if the supervisor agrees to bring into memory any segment on the disk which the user needs in such a way that the user does not explicitly specify the information transfer. In this case, the user could work in terms of a segment capability which the supervisor would provide by using core block capabilities and disk block capabilities. The user would write a machine instruction to access a segment without knowing if the segment would be in memory or not. If the segment was in memory and the user had a core block capability for the segment, then the instruction would execute normally. If the user had a disk block capability, however, the hardware would trap, implicitly calling the supervisor to bring in the segment, to provide a core block capability for it, and to restart the user's process so the instruction which caused the trap would be re-executed.

Input/output instructions are designed to drive input/output devices through a standard plug. Whatever device or devices connect into a single plug is addressed as a unit by the input/output capabilities. Each input/output instruction then specifies an input/output capability.

Since it is planned that all of the teletypes will be connected to the processor through a single plug, it will not be possible to give each user

an input/output capability for his own teletype. No doubt there will be some supervisor routines which can be called to read or write characters on a teletype. The question arises as to how the user will address the particular teletype he wants when he calls these routines. One answer is to invent a type of pseudo capability used for addressing teletypes.

Another type of pseudo capability might be used for addressing processes. When one process calls the supervisor to fork--that is, to start another process--the supervisor could provide it with a capability for the new process. This capability could then be used to specify the process if it is necessary to inquire about its status, stop it, etc.

A type of pseudo capability might also be used for addressing charge accounts. When a process is started it could be given capabilities for the various accounts to which it is authorized to make charges. One of these capabilities would then be presented to the supervisor by the process each time it initiated an operation for which the supervisor makes a charge, in order to indicate the account to be charged.

REFERENCE

1. Dennis, J.B., and Van Horn, E.C., "Programming Semantics for Multiprogrammed Computations." Comm. ACM 9 (March 1966), 143-155.

ON SIMPLIFIED IMPLEMENTATIONS OF THE EOL LANGUAGE

by Leon Lukaszewicz

The data structure of the EOL language as described in [1] and [2] permits one the convenient use of the full EOL language. In many cases, however, only a subset of EOL is used and the data have less general form; for example, the number of characters in EOL words may never be longer than a prescribed limit. In this case the efficiency of the object code may be improved by using a simplified data structure of stacks and files.

Example 1

Let us assume that

1) EOL words are never longer than 8 characters.

2) A computer word contains up to 4 characters.

3) The maximum storage space needed for each stack is known in advance.

4) Constituents can only be added or read from the top of stacks.

5) The current content of stack SN is equal to:

$$SN: \bar{X} = \bar{ALFA} + \bar{GAMMA} \bar{1}$$

Then a possible data structure for stack SN may be the one illustrated as scheme A in the Figure 1 below. The block SBN is the space allocated to the stack SN. Each constituent of the stack occupies two computer words. No free storage list is used. The tables SBEG and SEND contain addresses pointing to the first and the last constituents of the stack, respectively. In this example, addresses in SBEG are variable and those in SEND are fixed.

Example 2

Let us take all the assumptions of Example 1 except point 4 which is now as follows:

4) Constituents can be added or deleted from the bottoms of stacks only. Then a corresponding possible data structure for stack SN may be the scheme B on the preceding figure.

Example 3

Let us take all the assumptions of Example 1 except point 4. Then a possible data structure for the stack SN may be the scheme C.

In the case when new constituents are to be added at the top of a stack and there is no space for them at the beginning of the block SN the pointer in SBEG flips over to the bottom of SN. Similarly the pointer in SEND may flip over to the top of SN (scheme D).

Conclusion

There are many possibilities for simplified implementations of various subsets of EOL. Each implementation may be designed with a particular application in mind.

REFERENCES

[1] Lukaszewicz, L. and Nievergelt, J. "EOL Report", Report No. 241, Department of Computer Science, University of Illinois, Urbana, Illinois, September, 1967.

[2] "The EOL Interpreter", File no, _____, Department of Computer Science, University of Illinois, to appear.

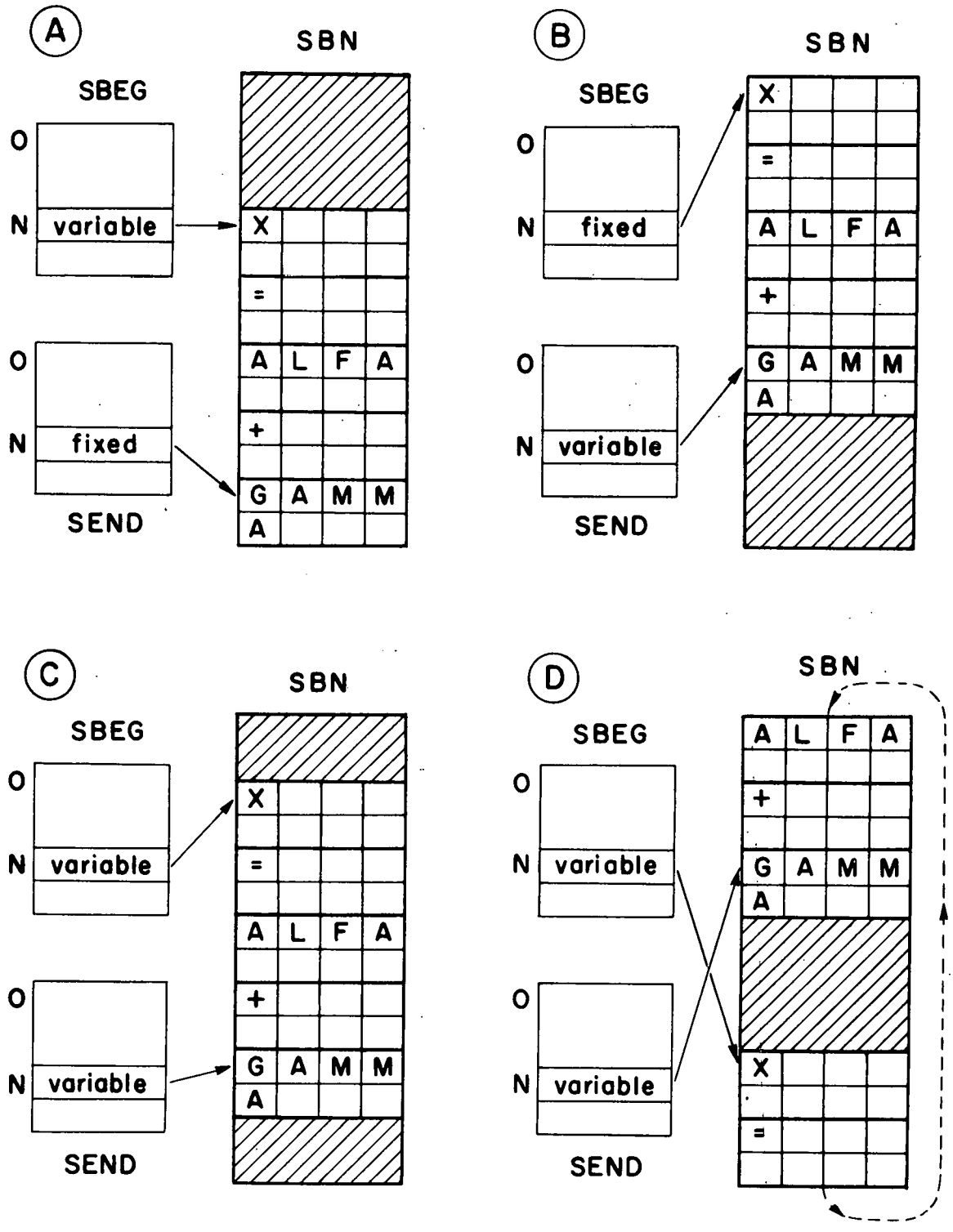
A
SBN
SBEG
0
N
variable
0
N
fixed
SEND
X
=
A L F A
+
G A M M
A
B
SBN
SBEG
0
N
fixed
0
N
variable
SEND
X
=
A L F A
+
G A M M
A
C
SBN
SBEG
0
N
variable
0
N
variable
SEND
X
=
A L F A
+
G A M M
A
D
SBN
SBEG
0
N
variable
0
N
variable
SEND
A L F A
+
G A M M
A
X
=

FIG. 1

SYMMETRICAL STACKS IN EOL

by Leon Lukaszewicz

The present EOL stacks, as described in [1], are asymmetric; that is, constituents can be read or deleted in one direction only -- from top to bottom. Although the last constituent of a stack can be read or tested, it cannot be deleted.

The stack asymmetry is the result of the assumed stack structure, which is now based on uni-directional (asymmetric) lists of two-word cells [2]. Every such cell contains one address, which is pointing to the following cell.

To have the stacks symmetrical, which allows reading and deleting constituents from both ends, the stack structure should be based on bi-directional (symmetric) lists of cells. Each cell of such a list contains two addresses, one pointing to the preceding cell, another to the following cell.

A possible data structure of symmetric lists for EOL stacks is shown in Fig. 1 below. The table of stack pointers contains the addresses pointing to the first constituents of a stack. The last constituent is connected to the first one so that the stack forms a ring*. The first and last cell of each constituent is marked by the bit "1" in the first position in the words containing the last forward and first backward address of the constituent cells.

With symmetric lists use is made of a free storage list in a similar manner as with asymmetric ones.

---

*The circular data structure was suggested by Jurg Nievergelt.

The content of stack Sn in the figure above is:

Sn: ¯ALFA¯BETA¯EPSILON¯...¯GAMMA¯1

Assuming symmetrical stacks, the MOVE instructions, for example, would have the form:

MOVE␣{A|B|Y|Z}<n>, {A|B|Y|Z}<m> [,<number of steps>|<constituent test>]

The letters Y and Z in the first argument have now the following meaning:

Y	read and delete successive constituents from the end of a stack S
Z	read and copy successive constituents from the end of a stack S without deleting them

Let us assume, for example, that

S1: ¯X1¯=¯A¯+¯B¯1

S2: ¯Q¯=¯1

Then the execution of the following MOVE instructions causes the results given on the right

MOVE Z1, Z2, *3	S1: no change S2: ¯Q¯=¯A¯+¯B
MOVE Y1, Z2, '='	S1: ¯Z¯=¯1 S2: ¯Q¯=¯A¯+¯B¯1
MOVE Y1, Y2, '='	S1: ¯X¯=¯1 S2: ¯Q¯=¯B¯+¯A¯1

Below is a set of EOL instructions in their extended form as the result of theo outlined proposal. Dots in square brackets represent the same test argument as in the original instruction (see [1]).

MOVE␣ {A|B|Y|Z}<n>, {A|B|Y|Z}<m> [...]

CLEAR␣ {A|Y}<n> [...]

TEST ␣ {A|B|Y|Z}<n> [...]

{ADD|SUB|MULT|DIV} ␣ {A|B|Y|Z}<n>[,{B|Z}<n>],[...]

WORD ␣ {A|B|Y|Z}<n> [,{B|Z}<m>]

NUMBER ␣ {A|B|Y|Z}<n> [,{B|Z}<m>]

SPLIT ␣ {A|N|Y|Z}<n> [...]

COMPRESS ␣ {A|B|Y|Z} [...]

COUNT␣ {A|B|Y|Z} [...]

WRITE ␣ {A|B|Y|Z}<n>, Q<m> [...]

PUT ␣ {A|B|Y|Z}<n>, {C|D}<m> [...]

RESTORE␣ [{A|B|Y|Z}<n>,] P<m>

{GONA|GOIN|CANA|CAIN} ␣ {A|B|Y|Z}<n>,<label>

SEARCH␣ {A|B|Y|Z}<n>,<label>

SEARCH␣ {A|B|Y|Z}<n>, {B|Z}<m>],<label>

EXTRACT␣ {A|B|Y|Z}<n>, {B|Z}<m>,<label>

To preserve the compatibility of the present and the extended version of the EOL language, the following restriction in the present version should be introduced:

> If in the MOVE or PUT instruction the first argument contains the mode letter "Z", then only the following can be used:
>
> MOVE␣ Z<n>, {A|B|Y|Z}<m>, *1
>
> PUT␣ Z<n>, Z<n>, {C|D}<m>, *1

With the restriction the present version of the EOL language is a subset of the extended one.

REFERENCES

[1] Lukaszewicz, L. and Nievergelt, J. "EOL Report", Report No. 241, Department of Computer Science, University of Illinois, Urbana, Illinois; September, 1967.

[2] "The EOL Interpreter", File no. _____, Department of Computer Science, University of Illinois; to appear.

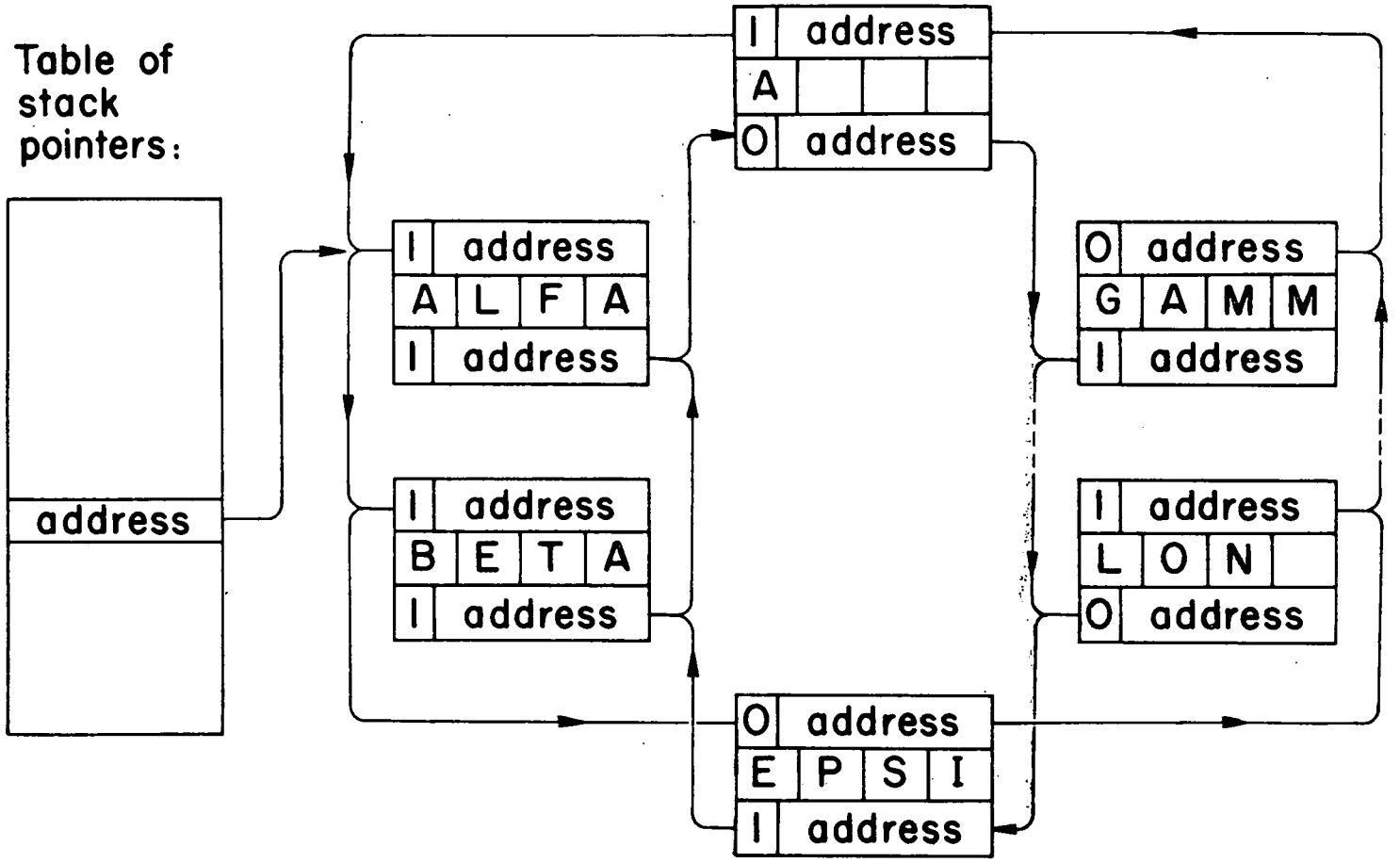
Table of stack pointers:
address
I address
A
O address
I address
A L F A
I address
O address
G A M M
I address
I address
B E T A
I address
I address
L O N
O address
O address
E P S I
I address

FIG. 1

THE UNCLLL PLEX-PROCESSING LANGUAGE: PRELIMINARY DESIGN DESCRIPTION II, and TENTATIVE PROGRAMMER'S MANUAL

R.B.K. Dewar and G.K. Manacher

Abstract

A list language oriented very heavily toward practical applications is described. The data control is very low-level, but the syntax is quite high-level. The design of the language is such as to ensure that sensible use of the features provided will generally result in code whose efficiency is within fifteen percent of that of best-possible code. This implies considerable attention to local optimization. The forms of optimization taken into account are data access, commutative arithmetic, fast-register specification, space allocation, function side-effect, logical operator, and expression evaluation.

A function mechanism equal in power to that of LISP [5] but quite a bit more flexible is provided. In the main it is an extension of the ideas in OEDIPUS [2] and will permit recursive function calls in minimum time. The language is easy to use, and only slightly machine- and implementation-dependent.

The present description supercedes a previous one [4]. Its syntax is greatly simplified, independent of constraints imposed by a macro assembler. Work on this version has not yet begun.

Introduction

For some time it has been apparent that there is a need for a powerful, general, low-level list language, which will at once permit the programmer to manipulate lists in a highly compact, dynamic way, and at the same time to generate code that is as efficient, or nearly as efficient, as code he would generate by hand.

"Low-level" means several things: The control of data structures should be microscopic and precise; the basic operations should correspond to machine instructions; the syntax should permit complex constructions that will compile into very efficient code. Furthermore, the programmer should find that he is never obliged, out of convenience or necessity, to employ a device in the language that will result in inefficient code.

These demands do not in themselves prohibit a high-level, powerful syntax, although they prohibit certain familiar features of well known languages. Among the features that are for our purposes too high-level are ALGOL-style [3] call-by-name, FORTRAN-style call-by-address, dynamic function declaration, program interpretation, SNOBOL-style [6] dynamic parameter handling, indexing loops without internal limitations on index access, run-time data-type checking, and so forth. Furthermore, many of the "convenience" features of familiar high-level compilers such as dynamic array-bound checking, field overflow, etc. must be left out, except possibly under a debugging option.

What is surprising is that even with these restrictions it is possible to put together a language which is syntactically high-level, concise, elegant and almost machine-independent. To be sure, we require a bit more declarative structure than a "normal" language, notably compiler-time declarations of fast registers, but this really presents no burden to the user.

The genesis of the overall syntax we have adopted lies in the observations of Dr. M.V. Wilkes [7], who pointed out that a language such as ALGOL could be looked upon as an "outer" syntax, describing mainly program flow, and incorporating a wide variety of possible data structures described in a relatively microscopic way as an "inner" syntax. The work of K.C. Knowlton [1] at once led us to realize that his low-level list language "L^6" --(it is more fashionable these days to call his structures "plexes" instead of "lists")--contains as its nucleus a plex-describing syntax that can be embedded into an outer "shell" resembling a substantial subset of ALGOL, so that the wedding of the two will indeed conform to Wilkes' Ansatz. The point is that the microscopic description of a list-type threading through a series of plexes is represented in Knowlton's language by a concatenation of letters and numbers. The first of these represents a fixed location called a bug, assumed for these purposes to hold a pointer. The subsequent ones represent certain fixed contiguous bit-patterns within a plex, called fields, which in turn hold pointers; these point to other plexes, and so forth.* The concatenation of a bug name with a series of field names specifies a way of threading

*Our terminology is slightly at variance with Knowlton's.

through a series of plexes. Such a concatenation will be called a sequence.

The embedding of low-level, data-description fields into a high-level, program-flow-oriented syntax is so natural that it seems irresistible. The sequences play the role of symbols in the higher-level language. To be sure, a few additional simple data types are needed, e.g., literals. But these, too, can be fitted in to the syntax as symbols, so that the determination of the nature of symbols can be thought of as lexical, rather than syntactic, in Cheatham's sense [8].

Our language permits the construction of expressions and function calls of arbitrary complexity. Basic facilities include fixed- and floating-point arithmetic, Boolean arithmetic, and bit-manipulation operations. The function mechanism, though quite powerful, is extremely low-level; however, the difference between this mechanism and that of LISP and ALGOL are so specialized that they are for the most part irrelevant. Analogues appropriate to a low-level language of pass-by-address, pass-by-subroutine name, and multiple-exit function failure, as well as exiting a controlled number of levels are provided.

Finally, a set of special precoded functions are included which permit quick access to the space-allocation and function machinery and powerful list-oriented diagnostic printout, if desired.

This language, we believe, represents the most efficient approach to list processing available. Its generality makes it a useful tool in compiler construction, network and flow analysis, theorem proving,

graphics, etc. In addition, it appears suitable as a target language for higher level special-purpose languages.

We have written a detailed preliminary manual which is included. This manual should be read, not in the spirit of digesting details, but rather in connection with its realization, in our opinion, of Wilkes' idea.

Assemblies of programs written in a similar language have already been produced. They fully justify the authors' belief that the language is both low-level in regard to generated code, and high-level in regard to convenience and flexibility of syntax.

1. Bugs and Fields. Literals.

A. Bugs

The objects serving as basic data in UNCLLL, in the sense that the symbol plays the role of basic data in ALGOL are the bug, the field and the literal. The bug is the basic variable in the language, and is specified by writing a single letter followed immediately by an optional integer. An arbitrary number of bugs may be used in a program. A bug is defined by its appearance on the left-hand side of an implicit or explicit assignment statement or as a calling parameter in a function call by returned value (cf. Section 5). Bugs are local to a single program and global to all functions within that program. An implementation dependent feature will allow special fast registers, such as index registers, to be used as bugs. Such fast bugs differ from other bugs in that they are global to all programs. This feature is useful when

several programs are to be linked and run concurrently.

As an example of the use of bugs, the statement

A12 = B + C8

will place the sum of the contents of bugs B and C8 in bug A12. It is thus clear that bugs play the role of variables in a conventional language. In fact, it is possible to write an ordinary program, not involving any list processing, solely in terms of bugs.

B. Sequences

Sequences are used for specifying the threading through a plex structure. In order to see what this means, it is first necessary to define a block, which represents a plex in core. A block is simply a sequence of words in memory. The language provides simple system functions for allocating and returning blocks of various sizes to or from storage at run time. The lowest address of the block is known as the starting address or simply the address of the block. A sequence is defined as a bug concatenated with an optional string of field names. Each field is specified by writing a single letter followed by an optional integer, so that the syntax for field names is identical to the syntax for bugs. The meaning of a field is that of a kind of template applied to a block. Fields are defined explicitly. The simplest way to define a field is by writing a statement known as a compile-time field declaration.

As an example, consider the statement

DEF E12,3,21,15

The effect of this declaration is to declare, at compile time, that the E12 field is defined as representing the field of contiguous bits found in the third word of a block, with a left offset of 21 bits (that is, starting at bit 22, if the first bit is called bit 1), and having a width of 15 bits. Notice that the field refers to no block in particular--it merely specifies what part of the block is being referred to.

A complex sequence will be said to be a sequence that is more complex than a bug, e.g., that involves at least one field name. Thus, for any reasonable machine, the bug is "wide" enough to hold an address. For a complex sequence, it is required that all the fields save the last one be wide enough to hold an address. A complex sequence is computed in the following fashion: The bug on the left is assumed to point to a block. The next field name refers to a contiguous set of bits in that block. If the field name is not the last in the field, it is assumed to be wide enough to hold an address, else an error occurs. The address is assumed right-adjusted in the field, and is assumed to point to another block. The next field name in the field refers to this block. The process continues until the last field name is encountered. This refers, of course, to the block pointed to by the bug or field name just preceding it. The contents of this final field name, right-adjusted, with leading zeroes,

constitutes the value of the entire sequence.

The maximum width of a field is equal to the width of a bug, which is in turn equal to the bit length of a general register or accumulator.* Field names may overlap word boundaries, i.e., the field definition

```
DEF X,1,30,25
```

is permissible on a 36-bit machine. In this example, the field embraces the rightmost 7 bits of the first word and the 18 leftmost bits of the second word.

For some implementations there will be predefined field names, already defined at compile time. These are fields that are particularly useful because of their correspondence with machine instructions, e.g., the address and decrement fields on the IBM 7094.

Fields may be redefined at compile time by having more than one compile-time field-name declaration in the program. Thus knowledge of predefined fields is not necessary since they may be overwritten at compile time.

Actually, three distinct mechanisms are available at compile time for specifying fields. The first is the DEF mechanism explained above. The second is defined by statements like

```
DEFV   E,3,15
```

*Fast bugs, on the other hand, may have smaller widths, but will always be wide enough to hold an address.

The effect is to define a vector of fields, and is identical to having written

```
DEF     E1,1,3,15
DEF     E2,2,3,15
DEF     E3,3,3,15
```

and so forth. In addition, E will be a synonym for E1.

"Bit vectors" can be defined in analogous fashion. The compile-time instruction

```
DEFBV   A,2
```

will define a series of fields one bit length wide spanning the second word of a block. The above instruction is equivalent to

```
DEF     A0,2,0,1
DEF     A1,2,1,1
DEF     A2,2,2,1
.
.
DEF     An,2,n,1
```

where n is the word-length of the machine.

An additional feature for defining fields is the run-time field declaration. Although poorer code will result, it is occasionally useful to define fields in a dynamic manner at run-time. To define a run-time field, it is necessary at compile time to signal one's intention to use such fields. The declarative statement

```
DEFR    name1,name2,...
```

is used, where name1,name2,... are the names of the fields being declared. Before executing a statement involving a field containing a run-time field name, it is necessary to call the DEFLD function (see Section 13).

In case the programmer is unhappy with the letter-then-integer format for bugs and field names, there is a compile time statement, OPTION, which will allow a more flexible syntax for these quantities. After compile-time execution of the statement

```
OPTION NAME
```

the programmer may use bugs and field names consisting of any combination of letters and digits other than a string of digits. In this case, the separator character underline (_) must be used between bugs and field names in the same sequence. When not compiling under OPTION NAME, underlines are ignored. OPTION NONAME reverts to the

default case. A feature allowing the programmer to index a bug or field within a sequence, useful for iterations over arrays, will be included in some implementations, and is described in Section 16.

Literals.

Literals are quantities that stand for themselves. Except for the case of the left side of an assignment statement, they are permissible everywhere in UNCLLL that a sequence is permissible.

Decimal literals (floating point)

These are numbers with a decimal point and possibly an exponent; they may not start with a zero.

Examples: 4.5 .01 76.32E10 +4.0 -3.14159 .0

Decimal literals (fixed point-integers)

These are integers not starting with a leading zero and possibly followed with a binary point place, which gives the bit number which is to correspond to the rightmost bit of the literal.

Examples: 24 12B35 6B34 (these are identical on a 36-bit machine)

-1 -56 +19

Octal literals

These are octal integers preceded by a zero and unsigned.

Examples: 077777 0156 0

Note: Octal 0 is equal to decimal 0.

Hollerith literals

These consist of a series of characters surrounded by quotes. A quote-mark itself is represented by two quotes; thus '' is the literal quote. The resulting literal is right-adjusted characters, left-filled with zeroes.

Examples: 'A''B' ',,, ,0' 'A' "876' '00('

Pointer or Address Literals

These consist of @ followed by a location, bug or function name or the name of a cell used in any other part of the assembly; their value is a pointer to the cell. They are most often used in conjunction with indirect function calls, closed subroutines and formatted I/0 which requires format pointers.

Examples: @LOCL @DEFLD

Hexadecimal literals

These consist of the character # followed by a hexadecimal integer.

Examples: #23AF #E100

Binary literals:

These consist of the cents-mark ¢ followed by binary digits.

Examples: ¢1011000110 ¢0 ¢010

2. Basic Structure of UNCLLL Statements.

An UNCLLL statement consists of two parts. The first part is an optional label, which must appear in card column 1.* Multiple labels, each signalling the same point in the program,may be used. In that case, the colon (:) is used to separate the labels. Individual labels may be followed by a colon. Labels may be composed of strings composed of letters and/or the characters "." and"-" (hyphen).

The second part is the statement body, which must start in some column other than column 1. There are several kinds of statement bodies (which we will also refer to loosely as statements). Let us first consider the simplest kind of statement body known as a simple statement. This consists of a function call, go-to statement, or an expression that can be interpreted as an assignment statement. A go-to statement indicates an unconditional transfer and most commonly consists of the word GOTO, followed by a single label, or just the word STOP, indicating termination of the program. Function calls will be described shortly. Let us pass on to the description of expressions, wherein it will become apparent which expressions are valid as simple statements and what their meanings are.

3. Expressions.

An expression can be either simple or conditional. A simple expression consists of a list of operands separated by binary operators with a unary operator possibly at the end of the list. Thus, typical expressions are:

*Only cols. 1-72 are active.

opd bop opd bop opd

opd bop opd uop

opd bop opd bop opd bop opd uop

opd

opd uop

where opd means operand; bop, binary operator; and uop, unary operator. The scheme of association is strictly right to left, except when shown explicitly to the contrary by parentheses. An operand can be a function call, parenthesized expression, literal or a sequence (cf. Section 1). Of these four types of operands, only one stands for a quantity whose value can be changed, namely, the sequence. Consequently we shall refer to an operand as changeable if it is a sequence.

The principal operator of an expression is defined as the first operator in the expression read from left to right. Thus, in the expression

$$(opd_1\ opd_2\ opd_3)\ bop_4\ opd_5\ uop_6$$

the principal operator is bop_4. The order of evaluation of this expression is as follows: uop_6 operates on opd_5. Then $opd_1\ bop_2\ opd_3$ is computed, and this serves as the left argument of bop_4 while $opd_5\ uop_6$ serves as the right. In other words, the order of evaluation follows the right-association rule.

One exception to the above rules is that if an expression consists solely of a parenthesized expression, it is evaluated as though the

parentheses were absent. Thus, in the expression

$$(opd_1\ bop_2\ opd_3)$$

the principal operator is bop_2.

Thus all binary operators are infix and all unary operators are postfix.

In what follows, we shall use the symbol q, possibly indexed, to represent an operand, and the symbol c to represent a changeable operand. The principal operand is the first operand in the expression.

Some operators are known as assignment operators. These require that the operand immediately to their left be changeable. Thus, the expression (A = 78) employs the binary assignment operator "=" and changes the value of A. An expression with an assignment operator as its principal operator will be called an assignment statement and is a legitimate simple statement. An expression with a principal operator, not an assignment operator, and a changeable principal operand will be called an implied assignment statement and is really a shorthand for an assignment statement, provided that it is used in a context in which a simple statement is appropriate. In such a context, the expression

$$c_1\ op\ q_2\ op_2 \ldots$$

will be interpreted as

$$c_1 = c_1 \text{ op } q_2 \text{ op}_2 \ldots$$

and will therefore result in the assignment of c_1 of the value of the expression.

If an assignment operator is used in a context requiring it to return a value, the value returned is the value of the first operand after the operation has taken place.

For example, multiple replacement may be written:

$$A = BWW = 077777$$

Assignment operators can be mixed into expressions in natural ways. In the expression

$$A + B = C * D$$

the principal operator is +. The result is to multiply C by D, put the result in B, and add the result to A. In addition, if the expression is used in a context in which a simple statement is called for, the result is stored in A.

4. The Structure of an UNCLLL Program.

An UNCLLL program consists of a list of UNCLLL statements, optionally labelled, and separated with semicolons. (In some implementations, notably macro implementations, these may be replaced

with parenthesis pairs).* At the end of the program appears the label END followed optionally by a label indicating the entry point in case the program is a main program. Statements can be either simple (cf. Section 2) or else of the following types: Declarative, conditional, compound, comment, or function-definitional. The remainder of this paper will elucidate these types.

5. Function Calls.

Function calls can be used as operands or simple statements. They have the following form, except for the remarks below:

$$\$name[(q,q,\ldots,/fail1,/fail2\ldots)]$$

(in the case of macro implementations, the left parenthesis may be required to be to the left of $name). Here name is the explicit name of the function, and q are the arguments. If the call is by value (true of all system and most programmer functions) then the call is exactly as above. If, however, an argument q is a sequence preceded by an asterisk *, as in $A(XY,*Q), then the final value of the corresponding formal parameter in the function is copied back on return. This will be termed "call by returned value" and is similar for most applications to call by address. The list fail1, fail2, ... is an ordered list of labels

* Also, for macro-assembler compatibility, the word DO may appear at the beginning of a statement, following the label. It will be assumed meaningless.

to which control is to be transferred if the called function fails. As we shall see, the function definition mechanism includes a means for selecting one of these. If the appropriate failure exit is not present and the function fails, an undetected error will occur.*

It is possible to pass the "name" (e.g., address) of a function into a function. This is accessed by means of an indirect call:

$$\$*c[(q,q,.../fail1,/fail2...)]$$

Here, c is a sequence or bug containing a pointer to the function to be called, typically generated by an '@' literal. The effect of this call is exactly as though the call had been explicit. q and fail1, fail2... have the same meaning as in the direct call.

6. Function Definitions.

Function definitions begin with a FENTRY declarative statement and end with an ENDF declarative statement. The FENTRY statement is of the form (at its simplest)

$$\text{FENTRY name}[(a_1,a_2,a_3,...)]$$

Here name is the name of the function. The $a_1, a_2, \ldots$ are the recursively saved locals. All locals are bugs (rather than sequences). As many of the locals as correspond with the parameters in the call are initialized with the values of the calling parameters. The rest are not initialized

*In the special case of one failure exit and attempted execution of FAIL or FAIL 1, the error will be detected.

to a set value.

Statements appearing within a function block (that is, between the FENTRY statement and the matching ENDF statement) are executed in the usual way. Bugs whose names are identical to the $a_1, a_2 \ldots$ within the function block correspond to the locals.

Certain declarative statements appearing within a function block have special meanings appropriate to the block. The RETURN statement has the form*

RETURN expression

and returns the value of the expression to the calling point. The FAIL statement has the form

FAIL expression

It consults the calling point, taking the failure exit whose ordinal position in the call is given by the value of the expression (observe that this feature can be used as a multiway branch). If there is no such exit, an undetected error occurs.** Both of these function exits of course go up a level of functional recursion. The function exit

*At the lowest level, outside any function block, RETURN is equivalent to STOP.

**See note, last section.

FAIL is equivalent to FAIL 1. The exit RETURN, not followed by an expression, goes back to the calling point with an unspecified value. It is useful when the calling function is a simple statement.

Another declarative statement defined within a function block is the SENTRY statement, of the form

SENTRY name1

The name name1 is distinct from name, the name of the function in which the SENTRY is embedded. This statement declares a subentry point to the function, and defines a callable function in the same fashion as FENTRY. However, it shares the locals with the function in which it is embedded.

The ENDF statement is of the same form as a RETURN statement, thus:

ENDF expression

However, ENDF also serves to delimit the function block.

The function structure is static in that all references to bugs other than the locals of the current function are to global values. Occasionally, however, it will be desired to have dynamically saved locals. Upon entry to the function, a dynamically saved local (bug) has its value just before entry hidden, and is initialized to the value it should have as a formal

parameter. Its old value is restored when the function is exited. Thus, if function F has a dynamically saved local L, and if it then calls function G (defined outside of F) which does not use L as a local, any reference to L in G will refer to F's local L. On the other hand, if L is a static local of F (the usual case), the reference to L in G will be to the global L, which has nothing to do with the local L of F.

The declaration of dynamically saved locals is a bit more expensive than the declaration of static locals. Dynamically saved locals are indicated in the function definition by putting a period before the name of the local. Thus, in

```
FENTRY name(A,B,.C1)
```

A and B are statically saved locals and C1 is a dynamically saved local. Both require the same amount of space on the pushdown list. The handling of static locals is similar to that of OEDIPUS [2].

Static locals have no address and in consequence cannot be pointed to by a pointer literal. Nor can static locals be "fast"bugs. If a static local happens to have the same name as a fast bug, it will nevertheless behave like an ordinary bug. In contrast, dynamic locals have addresses, and can be fast bugs. Thus, the dynamic bug feature is especially useful when it is desired to have a recursively saved fast bug within a function.

There is also provision for recursively saved scratch storage obtained automatically (and returned automatically) by the allocator. Suppose one writes the declaration

```
FENTRY  name(J,K,L(20),M,.N(Q))
```

where Q is an expression. The result will be to declare J,K,L,M, and N to be recursively saved locals. N will be dynamically saved. In addition, on entry to the function (at whatever level), a 20-block (a block consisting of 20 sequential words) will be fetched by the allocator and a pointer to it planted in L. Then Q will be evaluated, and a pointer to a Q-block planted in N. Upon exit from the function, regardless of the values of L or N, these blocks will be returned to storage.

Function definitions may appear within function definitions. In this case, ENDF's match corresponding FENTRY's by virtue of proper nesting, as in ALGOL. Moreover, an inner function shares the parameters of an outer function, in the fashion of the block structure of ALGOL. Unlike ALGOL, however, inner functions at whatever depth of definitional nesting are accessible from anywhere in the program.

Functions must be entered through the normal calling mechanism and may not be "fallen into". However, subentries may be fallen into; they act as though they are "transparent".

All labels within function blocks are inaccessible to go-to's except within the block and at the same level of function-definitional nesting. All labels in the outer level of the program are inaccessible from within a function block.

7. Conditional Statements.

The conditional statement has the form

IF predicate THEN sc [ELSE scc]

where an sc is a comma-separated list of simple statements or compound statements (next section) and an scc is either a comma-separated list of statements or compound statements, or else a conditional statement.*

A predicate consists of an expression and can be either direct or implied. Un like the situation relating expressions to simple statements, any expression may serve as a predicate.

A direct predicate is an expression whose principal operator is a boolean operator. For example, consider the operator EQ which asks whether its operands are equal. The expression

A EQ B + C

is true if the value of B + C is equal to the value of A and false otherwise.

*This comma-in-the-place-of-semicolon syntax provides flexibility with respect to the range of THEN and ELSE. In essence, it provides a single level of compound structure without the necessity of writing a BEGIN-ENDC pair (Section 8).

An implied predicate is an expression having no principal operator or one whose principal operator is not a boolean operator, a boolean operator being one that returns a truth value. In this case, if a predicate is expected, the compiler acts as though the predicate particle 0 NE (not equal) were in front of the expression. Thus, A is interpreted in this context as 0 NE A, A + B as 0 NE A + B, etc. In other words, the expression is treated as though it has a truth value: false if 0, else true. As we shall see shortly, predicate expression (whose principal operator is boolean) have meaning even when used in contexts not expecting a predicate, namely, their natural truth value.

The operation of the conditional statement is simple. The truth value of the predicate is tested. If it is true, the statement following the THEN is executed. If the ELSE is present, the statement following ELSE is not executed. Control then passes to the next statement. If the predicate is false, the reverse occurs, in the fashion of ALGOL.

The conditional statement is also available with IFNOT replacing IF. The meaning is the expected one.

The question mark (?) is a synonym for THEN.

8. Compound Statements.

A compound statement simply of BEGIN followed by any program followed by ENDC. Its use is chiefly in conjunction with conditional statements. Its labels are global, so that if one transfers to a label within a compound statement occurring (say) between the THEN and

ELSE of a conditional statement, the program logic will be as though one had gone there after testing the predicate and taken the appropriate branch of the conditional statement.

9. Declarative Statements.

These have already been accounted for in the main. They consist of a declaration word, e.g., OPTION, FENTRY, ENDF,... followed by some arguments. The legal operands of OPTION will be NAME, NONAME, and several others that will probably be particular to each implementation. For instance, ROUND on the IBM 7094 will cause all floating point instructions assembled from that point on to round their results, NORND will have the opposite effect, and [illegible].

10. Comments.

A statement starting with an asterisk (*) can consist of any string of characters other than semicolon. It will be ignored by the compiler.

11. Compound Expressions.

This is the one type of expression we have not yet looked at. It has the form

simple-expression IF predicate ELSE expression

and has the value of the leftmost argument if the predicate is true, else the value of the rightmost argument if the predicate is false. If the predicate has a principal operator, the predicate must be

enclosed in parentheses. PROV ("Provided that") is a synonym for IF in a compound expression.

The fact that the definition is recursive allows one to construct composites like

se IF predicate ELSE se IF predicate ELSE se

("se" means simple expression). These are similar to LISP-style conditionals and right associate in convenient fashion.

12. Operators.

A. Boolean Operators.

These operators test some condition, then either communicate the result if they are the principal operator of an expression used in a context expecting a predicate, or else return a 1 if the condition is true, or a 0 if the condition is false. None of them change the value of their arguments.

Numeric Relational Operators

All these operators take as arguments fixed or floating numbers or pointers as arguments (the mode on both sides must be the same); the arguments are in the form of literals or sequences or expressions of function calls.

q_1 EQ q_2	q_1 is EQual to q_2
q_1 NE q_2	q_1 is Not Equal to q_2
q_1 GT q_2	q_1 is GreaTer than q_2
q_1 GE q_2	q_1 is Greater than or Equal to q_2
q_1 NG q_2	q_1 is Not Greater than q_2
q_1 LT q_2	q_1 is Less Than q_2
q_1 NL q_2	q_1 is Not Less than q_2
q_1 Z	q_1 is Zero
q_1 NZ	q_1 is Non-Zero

Logical Operators

The arguments to these operators are Boolean quantities; nonzero means true, zero means false. Most often these arguments will be other Boolean expressions but they may be any quantities.

q_1 OR q_2	q_1 OR q_2 (or both) is (are) true
q_1 NOR q_2	neither q_1 NOR q_2 is true
q_1 AND q_2	q_1 AND q_2 are both true
q_1 NTB q_2	q_1 and q_2 are NoT Both true
q_1 XOR q_2	q_1 eXclusive OR q_2 is true
q_1 LEQ q_2	q_1 is Logically EQuivalent to q_2 (both true or both false)
q_1 NOT	q_1 is NOT true

NOTE: with some of these operators (e.g., OR), the result may be evaluated from only one operand. All these operators evaluate their

left operand first and then the second operand is evaluated only if necessary.

Bit Vector Comparison Operators

The arguments for these operators are considered as vectors of bits (up to a full word long).

q_1 BE q_2 — The Bit vectors q_1, q_2 are Equal (identical)

Note: due to multiple representations of numbers (e.g. +0 and -0 on the 7094) this is not the same as EQ.

q_1 BN q_2 — The Bit vectors q_1, q_2 are Not equal

Bit-flag Predicates

These unary predicates should be used for testing flags wherever possible.

q_1 ON — The bits in q_1 are all on.

q_1 OFF — The bits in q_1 are all off.

B. Arithmetic Operators

These operators give arithmetic results.

Fixed Point (Integer)

Any length arguments may be used but only full words can hold negative integers. The result of storing a negative integer in a smaller field is not defined.

$q_1 + q_2$	The sum of q_1 and q_2
$q_1 - q_2$	q_1 minus q_2
$q_1 * q_2$	the product of q_1 and q_2
q_1 / q_2	q_1 divided by q_2 (integer part)
q_1 MOD q_2	the remainder of q_1 divided by q_2

Floating Point

Only full word operands can be used with floating point operations.

q_1 ++ q_2	the floating sum of q_1 and q_2
q_1 -- q_2	q_1 minus q_2
q_1 ** q_2	the product of q_1 and q_2
q_1 // q_2	q_1 divided by q_2

C. Bit Vector Operators

These operators treat their arguments as vectors of bits, extended left with zero bits to full word length.

q_1 BO q_2	Bitwise Or of q_1, q_2
q_1 BA q_2	Bitwise And of q_1, q_2
q_1 BX q_2	Bitwise eXclusive or of q_1, q_2
q_1 L q_2	The contents of q_1 Left shifted by q_2 bits
q_1 R q_2	The contents of q_1 Right shifted by q_2 bits
q_1 RR q_2	The contents of q_1 Right-Rotated by q_2 bits*
q_1 LR q_2	The contents of q_1 Left-Rotated by q_2 bits*
q_1 C	The bitwise Complement of q_1 is If q_1 is a sequence

*The rotation takes place with respect to a full word.

with width W, less than a word-length L, the leading L-W zeroes in its value are also complemented.

Any of the 16 logical operators is available as follows:

Construct a truth table

LEFT	RIGHT	RESULT (e.g. for AND)
0	0	0
0	1	0
1	0	0
1	1	1

The name of the operator is 0001.

D. Assignment Operators

c = q	The value of q is stored in c.
c_1 IC c_2	The contents of c_1, c_2 are InterChanged.
c I q	The value of c Incremented by q is stored in c.
c D q	The value of c Decremented by q is stored in c.

Note that these operators give the changed result of the left operand as a value. Thus

A = 1; B = 2; C = A IC B;

results in C having the value of 2.

Several unary assignment operators are useful:

c CLR	Clear c (preferable to c = 0)
c CB	Clear the bits in c (synonymous with CLR)
c SB	Set all bits in c to 1 (useful for setting flags)
c IB	Invert all the bits in c

E. Pointer or Address Arithmetic.

On some machines, the UNCLLL word-length may be some multiple M of the basic addressing unit. For example, on the IBM 360, M = 4.

This means that care must be exercised in performing arithmetic operations on sequences holding addresses. In order to facilitate the use of arithmetic operators for address arithmetic, the unary (postfix) usage of +, *, 1, and - will be defined to represent addition, multiplication, ... by this multiple M. Thus, if X originally holds a pointer to the last word of a block, after execution of X I 2 * or X = X + 2 *, it will hold a pointer to the third word of the block.

X I will be a shorthand for X I 1 *, and X D will be a shorthand for X D 1 *.

F. Conditional Operators

See Section 11.

13. System Functions

These functions are part of the system and may be called in the normal manner described under function calls.

Storage Allocation Functions

$BLK(size) Allocates a BLocK of length size and returns a pointer to the block.

$REL(pointer, size)	RELeases to the allocator the block of size pointed to by pointer, returns 0.
$CLEAR(p,p,p,...)	CLEAR storage by returning all blocks to the allocator except those pointed to by p,p,p,...; returns 0
$AVAIL(size)	Returns the number of blocks of length size, which are AVAILable after splitting larger blocks.

Note that the allocator can allocate blocks of any size, and that the entire area of the block may be used by the program.

Functions for Manipulation of Function Calls, etc.

$TRACE(fpointer)	Initiates a diagnostic TRACE of the function pointed to by fpointer, returns 0.
$STOPTR(fpointer)	STOPs the diagnostic TRace initiated by $TRACE, returns 0.
$LEVTO(fname, expr)	Jumps up function LEVels until the current function name is that given by fname. A name of zeroes causes return to the highest level. Returns expr.
$LEVUP(nlevels, expr)	Jumps UP nlevels LEVels of function call. Thus ($LEVUP,1) is equivalent to RETURN. It conveys expr to the calling point nlevels up.

$LEVEL	Returns the current function level count (not including itself). Thus $LEVEL at the highest level returns zero)
$FAILUP(nlevels,expr)	Jumps UP nlevels LEVELs of function call, taking the failure exit whose index is given by the value of expr. $FAILTO is defined analogously.

General Functions

$DATE	Returns today's date in the form depending on the implementation.
$TIME	Returns the time of day.
$NONES	Returns the number of one bits in its argument.
$DLS($arg_1$,$arg_2$,nbits, $sizearg_2$)	Double Left Shift arg_2 is a bit vector of length $sizearg_2$. It is positioned to the right of arg_1 and the double quantity is left shifted by nbits bits. The result is arg_1 shifted left with the leftmost bits of arg_2 having entered at the right.
$DRS ($arg_1$,$arg_2$,nbits, $sizearg_2$)	Double Right Shift This is like DLS except that the shift is to the right. The result is arg_2 shifted right with the rightmost bits of

	arg_1 having entered at the left.
$ABS(expr)	Returns the ABSolute value of expr.
$NEG(expr)	Returns the NEGative of the fixed-point value of expr.
$FNEG(expr)	Returns the NEGative of the Floating-point value of expr.
$EXP(A,B)	Returns the floating value of A to the power of the floating value B.
$SQRT(arg)	Returns the square root of floating argument arg.
$MAX(a,a,a,...)	Returns the maximum value of the a's, which are either all floating point or all integers.
$MIN(a,a,a,...)	Returns the minimum value of the a's.
$FIX(a)	Returns, as an integer, the integer part of the floating point number a.
$FLT(a)	Returns the value of the integer a converted to floating point.
$MOVBL($q_1$,$q_2$,size)	Moves the first size words from the block pointed to by q_1 to the block pointed to by q_2.
$NORM(q,width)	Returns the result of shifting q so that its high-order bit is left-adjusted in a field of width width.

$RANDOM(q)	Returns a random integer n in the range $1 \leq n \leq q$.
$RANSET($q_1$)	Initializes the random-number generation to the integer q_1. At the beginning, the random number generator is assumed "initialized" to 1. Thus, random sequences may be repeated or interrupted.
$HASH($q_1, q_2$)	Takes the integer argument q_1 and returns a random number N generated from it in the range $1 \geq N \leq q_2$. Successive calls to $HASH with identical arguments produce the same number N each time.

Field Definition Functions

$DEFLD(name,word,offset,length)

DEfine (i.e., compile) run time FieLD

name is a pointer literal to the name of the field

word, offset, length have the same meanings as in the DEF declaration (see compile time fields).

$SAVDF(name)

SAVe then Field Definition of the field pointed to by name on a special stack used only for this purpose.

$LDFD(name)

$RESFD(name)

REStore the Field Definition of the field pointed to by name from the last entry of the pushdown.

Input-Output Functions

These are described in detail for a particular implementation; in general they will give access to the Fortran formatted and unformatted I/O available as well as allow several applications, such as automatic I/O of linked lists, particularly appropriate to UNCLLL. Definitions have not yet been adopted.

Diagnostic Print Functions

$LDUMP	Dumps all the blocks in use as well as other information such as Bug contents, pushdown levels, etc. Returns the number of blocks in use.
$BDUMP	Like LDUMP but dumps only those blocks pointed to by bugs. Returns the number of such blocks.
$PRBLOK(pointer)	Dumps the block pointed to by pointer. Returns 0.
$PRSTR(pointer,nam1, nam2,...)	PRints the STRucture defined as follows, pointer points to a block, whose fields of nam1,nam2,... point to other blocks. All blocks in the structure are dumped. Loops do not cause duplicate printing.

This list of functions is not intended to be complete, but rather to serve as a model for the kind of features to be implemented.

14. Features Which Result From Optimization.

Since the evaluation of expressions is rather heavily locally optimized, certain features are being left deliberately undefined. In other words, the user uses these features at his peril. This lack of definition is built into the language in order to keep the implementation free of unnecessary restrictions.

1. Operands which act as arguments to logical operators (AND, OR, etc.) may or may not be evaluated, according as they are needed for the evaluation of the entire expression. To the extent that the operands are evaluated, they are evaluated left-to-right.

2. Functions appearing within single expressions may be evaluated in unpredictable order. In general, the propagation of side-effects within an expression is not controllable.

3. The language is designed so that the best code results when the shortest way of writing an expression is adopted, assuming there is a choice. Thus CP SB is better than CP = 1 (assuming the bit-width of the P field is 1). Similarly, an implied assignment statement will sometimes result in better code than a real assignment statement. For instance, to assign to A the result of A BO B, it is better to write A BO B as a statement than A = A BO B, since some machines have an or-to-storage instruction.

4. Other oddities having to do with side-effect propagation may be declared to be undefined as they are unearthed.

15. Programming Examples.

We present here three examples of UNCLLL programming.

1. The function FCT defines the factorial function. Three possible encodings are shown. The left margin of the page corresponds with column 1 of the card.

```
FENTRY FCT(A); IF A GT 1 THEN GOTO R2; RETURN 1;
R2  ENDF A * $FCT(A - 1);
```

```
FENTRY FCT(A);   ENDF 1 PROV (A EQ 1) ELSE A *$FCT(A - 1);
```

```
FENTRY FCT(A);
RETURN 1 IF (A EQ 1) ELSE A * $FCT(A - 1);
ENDF;
```

2. The function XCHM is normally called by value returned. It interchanges its arguments' values if and only if the first is less than the second.

```
FENTRY XCHM(A2,A3);
IF A2 GT A3 THEN RETURN ELSE A2 IC A3;
ENDF;
```

3. A more complex example: The REACH function is called with its first argument serving as a pointer to a block. The block has the

following format. Every word has an A field, a D field, and a P field. The A and D fields are defined as vectors A1, A2, D1, D2, etc. The A and D fields are wide enough to hold an address, and the P field is 1 bit wide. On entry, the P1 field and A1 field contain zero. The D1 field contains the length of the block. The subsequent A fields (A2,A3,...) contain pointers to other blocks which have just the same format. Each block pointed to in turn points to other blocks, etc. The object is to link together all the blocks thus connected but to do it simply, via a single linked list using the A1 fields of each block for purposes of linkage. A bug G, global to the function and initially zero, serves as a header to the linked list. The function is as follows:

```
      FENTRY REACH(X,A); A = X + XD; XP SB; $LINK;
LOOP: X I 1 *; IF X EQ A THEN RETURN ELSE IF XAP OFF THEN
$ REACH (XA);
   GOTO LOOP; ENDF;
      FENTRY LINK; XA = G; G = X; ENDF;
                  ENDF;
```

Observe the function definition within a function definition, with the consequent sharing of variables. Actually, the inner function is unnecessary, as is the ELSE in the conditional statement. An alternative coding is the following, which allows X to be a fast bug:

```
    FENTRY REACH(.X,A);
  A = X + XD; XP SB; XA = G; G = X;
LOOP:  X I; IF X EQ A THEN RETURN; IF XAP OFF THEN $REACH(XA);
    GOTO LOOP; ENDF;
```

16. Dynamically Calculated Pointer Increments.

The fact that bugs and fields both refer to a fixed word in a block during the evaluation of a sequence sometimes makes it awkward to use sequences to address a block used as a linear array. To overcome this problem, a new type of pointer increment may be used. This increment is embedded in a sequence. It immediately follows a bug or field, and takes the form of an expression enclosed in a < > pair. The new syntax for a bug becomes

bug-name < expression>

and for a field,

field-name <expression>

The effect of expression is to modify the contents of the bug or field immediately preceding it by adding to it the contents of expression as a pointer increment, i.e. the value of expression times the number of basic addressing units per UNCLLL word.

17. Generated Code.

In order to see why we were motivated toward this language, we present here some highlights of the 7094 encoding of the last example of the function REACH, assuming that the call uses fast register X dynamically.

The first thing generated is a subroutine call to an internal routine $FALL, transferred to only if the programmer "falls into" a function.

Next the RETURN coding is generated. It will be transferred to when a RETURN is executed. This section always ends with an XEC instruction addressing a location on the stack serving as a return point. The stack location contains an instruction (a TXI in the 7094) which serves to reduce the stack pointer and also to transfer back to the calling point.

In the case of our REACH function, a little more work must be done; in particular, the current value of X must be swapped with the value stored on the stack in the position corresponding to the dynamic parameter represented by X. Since A is static, the same thing need not be done for it. Then, analogous code is generated for FAIL.

Now we are ready for the standard function-entry code, which consists of (1) a word containing the BCD representation of the function, for tracing purposes, (2) a word containing the number of arguments in the function, for tracing purposes as well as aiding the system functions LEVTO, LEVUP, etc., (3) the first word of executable code for the function, which always consists of an STQ that stores the function return word mentioned above on the stack. The remainder of the code is as

follows (Fast bug X is represented internally by index register 1):

```
      CLA     0,1          Get XD
      PDX     ,7
      PXA     ,7           Shift XD to right-adjust it.
      STO     TEMP         TEMP is really on the stack
      PCA     ,1           indexes normally stored complemented
      ADD     TEMP         now we have X + XD
      STO     2,6          2,6 is synonym for A (XR 6 is the
                           stack pointer)
      CAL     PMASK        created by compiler
      ORS     0,1          set the bit
      CLA     G
      STA     0,1
      PCA     ,1
      STO     G            completing the linkage
LOOP  TXI     *+1,1,1      Quick increment of X
      PCA     ,1           Get X
      SUB     2,6          Subtract the stack-synonym for A
      TZE     RETURN       Defined in the header
      LDI*    0,1          Prepare to test XAP
      OFT     PMASK
      TRA     ..001        A created symbol
      CLA     0,1          Get word with XA in address
      ANA     AMASK        Mask address
      STO     3+1,6        Pass parameter to next level
      LDQ     *+3          Begin standard five word calling sequence
      TIX     REACH,6,3    The recursive call
      STR     RETN.,,NARGS Out-of-pushdown-space-transfer, plus
                           some data
      TXI     *+2,6,3      The return instruction
      TXI     $NOFRT,6,3   Failure return (none given, transfer
..001 TRA     LOOP         to external fatal-error routine)
```

18. Miscellany

Spaces are important in UNCLLL and cannot be ignored. A general rule, with some exceptions, is that a space should be present between all operators and operands and in general between distinct terminal symbols of the syntax. The exceptions are found:

(1) in the internal constructions of sequences, of course;

(2) in function calls;

(3) in argument-lists of function definitions, and between the function word and the left parenthesis of a function definition;

(4) in connection with some operators, which can be thought of as "carrying their own blanks with them." In general, the operators composed of nonalphanumeric characters are in this category.

Continuation cards have no special syntax, since the semicolon delimits a statement. However, statements may be broken over cards only where a blank is permissible. Column 1 of the continuation card should not be used.

Indirect goto's are handled by goto statement

JTH expression

Here, JTH is a mnemonic for "Jump through", and the value of expression is an address to which control is to be passed. Its main use is for jumping out of small nonrecursive subroutines.

19. Debugging.

One of the most difficult problems programmers will encounter in a language such as this is debugging. The language, being low level, has very little protection built into it. Thus, for example, returning the same block, twice, to the allocator via release function $REL may result in parceling it out twice much later on, with attendant confusion owing to double concurrent usage. As another example, making

a slight mistake in indexing over a block may permit one to write into the next block contiguously in storage. Since there is no way for the programmer to tell what significance the next block has, any errors that occur may occur in completely unpredictable ways. These difficulties may in some cases be palliated by the diagnostic features that will be built in, but they will not be cured.

One line of attack that seems promising is to have a (slow) interpretive version which is similar to the original, with less allocatable storage and perhaps slightly fewer features, together with additional declarations valid only for the interpretive version. The several types of data (floating, fixed, pointer, etc.) would be tracked at run time to make sure that they were suitable operands for the various operators and system functions. In the case of integers, an operand would be flagged as a legitimate pointer only if (1) it were set by an assignment to a value returned by the BLK space-allocating function, (2) it were assigned a value already so flagged, or (3) it were changed from a pointer within some block to a pointer within the same block via some addition or subtraction. Blocks returned to storage would be checked to verify that their length is correctly given, and that they are currently in use. In these ways, many annoying and hard-to-detect mistakes would come to light.

20. Macros.

We believe that macro facilities similar to those proposed for PL/1 [9] should be useful for UNCLLL. In addition, we hope to provide

for the expression of new operators in terms of machine code.

21. Formal Syntax.

The accompanying syntax gives UNCLLL statements as we have presented them, but is simplified at a few points.

(Note: "*" means "common lexical definition". "**" means a long disjunction of terminal symbols; see text. The symbol ε represents the void string).

main program	→	program end-statement ;
program	→	labelled-statement-list [; program]
labelled-statement-list	→	label-list [DO] statement-list
label-list	→	label^{+} [: label-list] | null-label^{+}
statement-list	→	statement [; statement-list]
statement	→	general-statement | function-definition | ε
general-statement	→	uncond. -statement | cond. -statement | comment
uncond. -statement	→	simple-statement | decl. -statement
simple-statement	→	unit-computation | goto-statement | compound-statement
goto-statement	→	GOTO label | RETURN [expression] | FAIL [expression] | JTH expression | STOP

+must start on col. 1 of card. A null label is blank.

unit-computation	→	function-call | (unit-computation) | assignment-statement | implied-assignment-statement
assignment-statement	→	sequence assignment-binary-op** expression | sequence assignment-unary-op** | sequence IC sequence
implied-assignment-statement	→	sequence binary-op** expression | sequence unary-op**
expression	→	simple-expression | conditional-expression
simple-expression	→	data-unit binary-op** expression | data-unit unary-op** | data-unit | assignment-statement
function call	→	$ fname [(arglist)]
arglist	→	arg [, arglist]
arg	→	[*] expression
data-unit	→	function-call | (expression) | sequence | literal
label	→	xas
fname	→	xas
xas	→	string-of-letters-or-period-or-hyphen*
decl.-statement	→	DEF field-name , word , offset , length |
		DEFV field-name , offset , length | DEFBV field-name , word | DEFR field-name-list | OPTION option-list
field-name	→	letter* [integer*]
word	→	integer*
offset	→	integer*

length	→	integer*
field-name-list	→	field-name[, field-name-list]
option-list	→	option[, option-list]
option	→	ROUND|NAME|...* +
compound-statement	→	BEGIN program ENDC
comment	→	* any-characters-but-semicolon*
cond.-statement	→	ifn predicate THEN simple-statement-sequence [ELSE simple-or-conditional-statement]
ifn	→	IF|IFNOT
simple-or-conditional-statement	→	simple-statement-sequence|conditional-statement
simple-statement-sequence	→	simple-statement [, simple-statement-sequence]
predicate	→	direct-predicate|implied-predicate
direct-predicate	→	data-unit boolean-op** expression
implied-predicate	→	data-unit|data-unit nonboolean-op** expression
conditional-expression	→	simple-expression if predicate ELSE expression
if	→	IF|PROV
end-statement	→	END++ label
function-definition	→	function-header ; interprogram ENDF [expr]

+other options may be added later.
++starts in column 1.

function-header	→	FENTRY fname [(formal-parameter-list)]
formal-parameter-list	→	generalized-bug [, formal-parameter-list]
generalized-bug	→	[.] bug [(expression)]
interprogram	→	program [SENTRY name ; interprogram]
sequence	→	complex-bug [field-name-list]
complex-bug	→	bug-name [< expression >]
bug-name	→	letter*[integer*]
field-name-list	→	complex-field-name [field-name-list]
complex-field-name	→	field name [< expression >]
literal	→	floating-number*|hex-literal|octal-literal|binary-literal|pointer-literal|fixed-decimal-number*|hollerith-literal
hollerith-literal	→	' characters-other-than-quotes '
octal-literal	→	0 octal-number*
pointer-literal	→	@ name*
hex-literal	→	# hex-number*
binary-literal	→	¢ binary-number*

22. Acknowledgements.

We wish to thank Messrs. K.C. Knowlton, M.D. McIlroy, W.S. Brown, V. Kruskal, J. Pomeranz and W.S. Worley for insightful comments that led to many of the design and implementation features of this language.

REFERENCES

1. Knowlton, K.C., A Programmer's Description of L^6. Comm. ACM, Vol. 9, No. 8, Aug., 1966, pps. 616-625.

2. Brown, W.S., An Operating Environment for Dynamic-Recursive Computer Programming Systems, Comm. ACM, 6, 1965, pps. 371-377.

3. J. W. Backus, et al. Report on the Algorithmic Language Algol 60, Numerische Mathematik, 2, 1960, pps. 106-136.

4. Dewar, R.B.K., and Manacher, G.K., The UNCLLL List-Processing Language: a Preliminary Description. ICR Quarterly Report No. 13, University of Chicago, May 1, 1967.

5. McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, part I, Comm. ACM 3, April, 1960, p. 184.

   See also: LISP 1.5 Manual, MIT Press, 1965

6. Farber, D.J., Griswold, R.E., and Polonsky, I.P., The SNOBOL3 Programming Language. BSTJ XLV, 6, July-Aug., 1966, pps. 895-944.

7. Wilkes, M.V., The Outer and Inner Syntax of a Programming Language, Technical Memorandum No. 67/4, University Mathematical Laboratory, Cambridge (July 1967).

8. Cheatham, T.E., Jr., The Theory and Construction of Compilers. Report No. CA-6606-0111, Computer Associates, Inc., Wakefield, Mass.

9. Any version of IBM's PL/1 report containing a description of macro facilities.

SIGNAL-TO-NOISE RATIO IN FRINGE VISIBILITY MEASUREMENTS WITH A MICHELSON STELLAR INTERFEROMETER

R. H. Miller

(Part of a Report to be published in the Proceedings of the Summer Study on Optical Aperture Synthesis)

At the National Academy Summer Study on Optical Aperture Synthesis, D. Currie (University of Maryland) and P. Bender (Bureau of Standards, Boulder) proposed a means of detecting fringes and measuring their visibility that uses fluctuations of optical phase between two beams due to astronomical "seeing" as the modulating mechanism. The detector can be a single-element interferometer, rather like half of a Mach-Zehender interferometer. This note contains a description of how the fringe visibility might be measured with such a detector, and a study of the accuracy with which it may be measured in the presence of background and shot noise.

The signal-to-noise problem for fringe detection is expected to be most severe at very low intensities (faint objects). If the problem is then manageable, it should be tractable with brighter objects.

With faint objects the number of counts obtained with a photomultiplier or other quantum detector, operating for a specified time, will be Poisson distributed.* Poisson distributions may be governed by a parameter that is a function of the time, and the time dependence of the number of counts will be controlled by the time dependence of the parameter. The parameter in the Poisson distribution is simply the integral of the (time-dependent)

*If it were not, a Hanbury-Brown-Twiss measurement could be made.

expected count rate over the counting interval:

$$p_N(n) = \frac{N^n e^{-N}}{n!} \qquad (1)$$

with

$$N = \int_{\text{Counting Interval}} dt' \, R(t') \, . \qquad (2)$$

(This is easily demonstrated; it also appears in Mandel's article. [1]

The fringe detection scheme suggested by Currie involves considerations similar to those obtained with other detector schemes. Two detectors are arranged so that, with a certain relative phase of the two beams, all the light goes to one of them, while if the relative phase differs by π, all the light will go to the other. At intermediate phase-differences, the light will be shared by the two detectors. The phase difference will fluctuate because of the seeing. Suppose the expected counting rates for the two detectors are

$$R_{0,1}(t) = \alpha \pm \beta \cos \phi(t) \, , \qquad (3)$$

where the fluctuating phase enters as the cosine of a variable angle; this angle may be as large as several times 2π. The terms α and β include the corresponding intensities and detector efficiencies. *

*Photomultiplier dark current and other time-independent background noise can be included in α. If so, the relation (4) for fringe visibility must be modified, as in Eq. (22).

Correlations in the counts registered by detectors 0 and 1 arise from the relations between the expected counting rates. The fringe visibility is

$$\mathcal{V} = \frac{I_{max} - I_{min}}{I_{max} + I_{min}} = \frac{\beta}{\alpha} . \qquad (4)$$

Currie points out that fringes may be recognized by looking at sequential pulse pairs because one of the detectors will count faster than the other. If a signal produced by detector 1 followed by one produced by detector 0 be denoted by 10, then tally a count in channel D (for direct) if $11 \vee 00$; a count in channel S (for switch) if $10 \vee 01$. Inasmuch as this can be done for each pulse pair, the number of counts $n_D + n_S$ is the total number of counts registered in both detectors (irrespective of origin) ($n_D + n_S = n_0 + n_1 - 1$).

It is characteristic of the assumed statistical properties of the counts produced by detectors 0 and 1 (Poisson distribution) that (1) the probability for a count to appear at one detector is independent of whether one has occurred at the other, just as it is independent of the precise time the last count in the same detector (more formally, the probability that no count has appeared up to the present time is an exponential distribution), and (2) the probability for a count to appear in either detector (inclusive or) is itself a Poisson distribution whose parameter is the sum of the parameters for the individual detectors. In all these Poisson distributions, the expected count rate may be

time-dependent, and correlations may appear through different expected rates.

Starting from an arbitrary time origin, the probability to have a count in detector 1 during the time interval t to t + dt without having previously had a count in either detector 1 or detector 0 is

$$p_1(t)\,dt = e^{-(N_0+N_1)}\,R_1(t)\,dt, \tag{5}$$

the probability to have a count in detector 0 with no previous count in either detector is

$$p_0(t)\,dt = e^{-(N_0+N_1)}\,R_0(t)\,dt, \tag{6}$$

and the probability for a count in either detector is

$$[p_0(t)+p_1(t)]\,dt = e^{-(N_0+N_1)}\,[R_0(t)+R_1(t)]\,dt. \tag{7}$$

Here, the expected number of counts is Eq. (2):

$$N_0 = \int_0^t dt'\,R_0(t') \tag{8}$$

$$N_1 = \int_0^t dt'\,R_1(t'). \tag{9}$$

At this point, an essential feature of the present calculation emerges that makes it tractable: there is no time-dependence of

$[R_0(t) + R_1(t)] = 2\alpha$ (Eq. 3). Thus $N_0 + N_1 = 2\alpha t$,

a fact that greatly simplifies the calculation (the random function of the time has cancelled out).

The expected numbers of counts in the channels n_D and n_S may now be calculated. If the counts are tallied over the time interval $0 - T_0$ then

$$n_D = \int_0^{T_0} R_1(t)\,dt \int_t dt'\, e^{-(N_0+N_1)} R_1(t') + \int_0^{T_0} R_0(t)\,dt \int_t dt'\, e^{-(N_0+N_1)} R_0(t'), \tag{10}$$

where N_0 and N_1 are given by integrals running from t to t', and the inner integral in each case may be considered as describing the process of awaiting a second pulse each time one is registered. These form the sequential pairs of pulses in Currie's scheme. A similar pair of double integrals describes n_S :

$$n_S = \int_0^{T_0} R_1(t)\,dt \int_t dt' e^{-(N_0+N_1)} R_0(t') + \int_0^{T_0} R_0(t)\,dt \int_t dt'\, e^{-(N_0+N_1)} R_1(t'). \tag{11}$$

The upper limits on the inner integrals have purposely been left unspecified. Currie has suggested inserting a cutoff at a time $t + \tau$, representing a gating time during which a second pulse is accepted following the first, or the introduction of some weighting function in the inner integral. Neither of these is really necessary--the exponential

is enough of a weighting function--but an upper limit $t+\tau$ will be used. There is a possibility that $t+\tau$ could exceed T_o, requiring use of $\min(t+\tau, T_o)$ as the upper limit: since it may be expected that T_o will greatly exceed τ, this may be ignored (terms $O(\tau/T_o)$ are then being ignored).

When the assumed expected count rates (Eq. 3) are inserted into these integrals, the cross-terms ($\alpha\beta$ terms) will have contributions of order 1 while the other terms are of order T_o. Thus, for very long times (large T_o), the $\alpha\beta$-terms are $O(1/T_o)$ relative to the others and, in the spirit of the usual autocorrelation argument, may be neglected. The integrals can be regarded as double integrals, whose order can be interchanged after a change of variable to $t'' = t' - t$. Since very long times (effectively infinite) are to be considered, the limits may be simply handled. The integrals with a coefficient α^2 can be done immediately and give $\alpha(1-e^{-2\alpha\tau})T_o$; those with β^2 give:

$$\pm 2\beta^2 T_o \int_0^{\tau} dt'' \, e^{-2\alpha t''} \left[\frac{1}{T_o} \int_0^{T_o} dt \, \cos\phi(t) \cos\phi(t+t'') \right], \qquad (12)$$

which involves the autocorrelation of $\cos\phi(t)$, integrated over the window-time with a weighting function. In the limit $T_o \to \infty$, $\tau \to \infty$, the Laplace transform of the autocorrelation appears; the time independent part of the counting rate (2α) is the Laplace transform variable s. The integrals yield some function of α, τ, and of the parameters describing the seeing phase, $f(\alpha, \tau, \bar{\bar{\phi}})$. Finally, the expected number of counts

$$n_{D,S} = [\alpha(1-e^{-2\alpha\tau}) \pm 2\beta^2 f(\alpha,\tau,\bar{\bar{\Phi}})]T_o . \qquad (13)$$

This system can be solved to yield β and α in terms of the number of counts, n_D and n_S ; this will be taken up later.

A modification of this scheme is to count ordered n-pulse sequences. In principle, more information is available by making use of these n-tuples. These might be separately tallied or included with n_D and n_S with suitable weightings. Their probability distribution is not simply multinomial because the sequence is studied. The expected number of counts of a specified n-pulse sequence will be given by an n-fold integral of the form

$$n_{ijk\ldots n} = \int_0^{T_o} dt\ R_i(t) \int_t^{t+\tau} dt'\, e^{-2\alpha(t'-t)} R_j(t') \times$$
$$\times \int_{t'}^{t+\tau} dt''\, e^{-2\alpha(t''-t')} R_k(t'') \times \ldots\ldots \times \qquad (14)$$
$$\times \int_{t^{(n-1)}}^{t+\tau} dt^{(n)}\, e^{-2\alpha(t^{(n)}-t^{(n-1)})} R_n(t^{(n)}) .$$

Here the upper limits on the inner integrals correspond to a sharp cutoff that has been inserted to require the entire n-pulse sequence to arrive in a time interval of duration τ after the first pulse of the sequence; the upper limits might be changed to correspond to other conditions. The indices i, j, k... take on values 0 or 1 referring to the detector

number. Suitable linear combinations of these integrals, such as used in Eqs. (10) and (11), go together to describe the number of counts in a certain category (the triple combinations 111 and 000, for example). It is clear that these integrals will yield higher order autocorrelations of $Cos\ \phi(t)$. Some cross-terms between α and β bring in the lower order autocorrelations. This matter will not be pursued further; there is no point of principle that is unclear.

Some insight into the character of the autocorrelation function might be sought by guessing convenient behaviors for $\phi(t)$. The autocorrelation will give:

$$\frac{1}{2T_0}\int_0^{T_0}\left\{Cos\left[\phi(t)+\phi(t+t'')\right]+Cos\left[\phi(t)-\phi(t+t'')\right]\right\} \qquad (15)$$

and the first part averages to zero in the usual spirit of autocorrelation integrations. The second piece cannot vanish for $t''=0$; so it must be non-zero for some range of values of t'' . Although the integral is a time-average and the prescription by which it was obtained clearly calls for time-averaging, it may be approximated by an ensemble average. To do this, imagine an ensemble of systems, with an ensemble averaged probability to have a certain phase difference at a certain time-difference, t'' . This probability must be a function of t'' that reduces to a δ -function at $t''=0$. A convenient form for this is suggested by considering that the phase difference might be described by a random walk; the asymptotic diffusion solution may be appropriate. [2]

$$W(x,t'')\Delta x = \frac{1}{2(\pi D t'')^{1/2}} \exp\left\{-\frac{x^2}{4Dt''}\right\}\Delta x, \qquad (16)$$

in which x is to be treated as the phase-difference $\phi(t) - \phi(t+t'')$. Then the autocorrelation can be obtained from

$$\frac{1}{2}\int_{-\infty}^{\infty} \cos x \; W(x,t'')\,dx = \frac{1}{2} e^{-Dt''}. \qquad (17)$$

The diffusion or random-walk solution is not physically admissible for all times because it grows without bounds and is of infinite extent. However, it is probably a reasonable approximation. Other probability distributions can be easily invented and the autocorrelation can as easily be worked out for them. Returning to the Gaussian, the integral for $f(\alpha, \tau, \bar{\bar{\Phi}})$ is:

$$\frac{1}{2}\int_0^{\tau} dt'' \, e^{-2\alpha t''} \, e^{-Dt''} = \frac{1}{2(2\alpha+D)}\left(1 - e^{-\tau(2\alpha+D)}\right). \qquad (18)$$

In the limit $\tau \to \infty$ this reduces to the Laplace transform, as noted earlier.

With this value for $f(\alpha, \tau, \bar{\bar{\Phi}})$, expressions can finally be written down for n_D and n_S :

$$n_{D,S} = \alpha T_0 \left(1 - e^{-2\alpha\tau}\right)\left[1 \pm \frac{\beta^2}{\alpha(2\alpha+D)} \; \frac{1 - e^{-(2\alpha+D)\tau}}{1 - e^{-2\alpha\tau}}\right]. \qquad (19)$$

These numbers η_D and η_S are expectation-values that may be inserted into the generalized Poisson distributions of Eq. (1) for the number of counts of each kind to be expected in a measurement. The factor $(1-e^{-2\alpha\tau})$ is the probability to get at least one count within the gating time following a given count.

Amplitude scintillation can be introduced into this analysis; then α of Eq. (3) is also a function of the time. The sum of $\mathcal{R}_0(t)$ and $\mathcal{R}_1(t)$ is no longer constant; however, with the approximation that it is treated as constant inside the exponential, an identical formalism will lead to the Laplace Transform of the autocorrelation of the (random) amplitude function. This point will not be pursued further.

It has been noted earlier that it is not necessary to use a finite value of τ in Eq. (19). Because further analysis is simpler in that case, let $\tau \to \infty$. Then $\eta_{D,S}$ involve β^2/α^2 if $2\alpha \gg D$ and involve $\beta^2/\alpha D$ if $2\alpha \ll D$, with a crossover at $2\alpha \approx D$. A value for D can be estimated from the fact that Michelson and Pease were able to observe fringes. This means that a spread of the seeing phase of about a radian cannot have occurred much faster than 1/10 second. In Eq. (16) this sets $D \sim 2.5\ \text{sec}^{-1}$, and thus places the crossover at about $\alpha = 1$ to 2 detected counts per second. Since this is well into the counting rate at which night sky background and photomultiplier current will dominate the noise picture, it is probably safe to use the β^2/α^2 relation to estimate the variance in the determination of the fringe visibility, V. If the phase-independent part (α) contains some background or detector

noise, it can be written as the sum of contributions from the object being studied (α_o) and from the background plus detector noise (α_B): $\alpha = \alpha_o + \alpha_B$. The visibility can be written as β/α_o ; the accuracy with which α_B is known will enter the equations. For simplicity of notation we shall take α_B the same for both detectors.

In principle, exact sampling distributions could be written; in practice, these would be very complicated. Since both n_D and n_S may be fairly large numbers, the first and second moments of the Poisson distribution will be used; this implies the asymptotic normal approximation. Then $\langle n_{D,S} \rangle = n_{D,S}$, $\langle \delta n_{D,S}^2 \rangle = n_{D,S}$, and $\langle \delta n_D \, \delta n_S \rangle = 0$. (This last relation is a consequence of the assumptions made at the outset: the sampling errors in n_S do not depend on those present in n_D .)

The background rate, α_B , can be determined by directing the interferometer off the star to a (star-free) field of the same sky brightness, and counting there for a time, T_B . Then $\alpha_B = n_B/T_B$, where n_B is the number of counts obtained in the background run, and $\langle \delta \alpha_B^2 \rangle = n_B/T_B^2$. There is no correlation between α_B and n_D or n_S . In an ordinary first-order theory:

$$\langle \delta V^2 \rangle = \left(\frac{\partial V}{\partial n_D}\right)^2 \langle \delta n_D^2 \rangle + \left(\frac{\partial V}{\partial n_S}\right)^2 \langle \delta n_S^2 \rangle + \cdots$$

$$+ 2\left(\frac{\partial V}{\partial n_D}\right)\left(\frac{\partial V}{\partial n_S}\right)\langle \delta n_D \, \delta n_S \rangle + \left(\frac{\partial V}{\partial \alpha_B}\right)^2 \langle \delta \alpha_B^2 \rangle + \cdots \qquad (20)$$

+ cross terms between n_D , n_S , and α_B .

From Eq. (19) with $2\alpha \gg D$, $\tau \to \infty$,

$$\frac{\beta^2}{\alpha^2} = \frac{n_D - n_S}{n_D + n_S} . \tag{21}$$

Then

$$v = \sqrt{2}\sqrt{\frac{n_D - n_S}{n_D + n_S - 2 n_B T_0/T_B}} \tag{22}$$

and

$$\langle \delta v^2 \rangle = \frac{16}{v^2 (n_D + n_S - 2 n_B T_0/T_B)^2} \times$$
$$\times \left[(n_S - n_B \frac{T_0}{T_B})^2 n_D + (n_D - n_B \frac{T_0}{T_B})^2 n_S + \frac{v^4}{4} n_B \frac{T_0^2}{T_B^2} \right] . \tag{23}$$

The problem of optimum division of observing time between measurements and the determination of background rates is readily formulated from this expression. In the case where the background count is unimportant, the variance in the fringe visibility determination goes as v^{-2} ; small visibilities are much more difficult to measure. On the other hand, the background calibration is less important with small visibilities.

Particularly because some background and detector noise will always be present, it is not necessary to introduce a cutoff time $(\tau \to \infty)$. This should simplify the pulse handling logic.

The formalism developed here is satisfactory for time-dependent backgrounds (including scintillation due to the "seeing"); it follows through

all the way if the background varies little in a time $1/\alpha$ (where α includes the background).

A 1P21 photomultiplier, properly selected and refrigerated, can give a detector background of $\sim$ 10 to 50 counts per minute when adjusted to count single photoelectron pulses. This is slow enough to reach the limit $2\alpha \sim D$, which might force the inclusion of a cutoff (τ not $\to \infty$); signal count rates of 5-10 sec^{-1} are enough to be in the "high-rate" regime of Eq. (23). Using Johnson's [3] calibrations and a 10% quantum efficiency for the photomultiplier, a band 1 Å wide at 4400 Å through a 3-inch aperture would give $\alpha \sim 10\ sec^{-1}$ at a (B-)magnitude of 7.1. Then $\langle \delta v^2 \rangle$ of 10^{-2} could be reached in about 10 sec at $v=1$, but would require 1000 sec (20 min) at $v=0.1$. These rates are probably optimistic by about a factor two because mirror inefficiencies, losses in a Mach-Zehender interferometer detector, imperfect filter overlap and transmission will all reduce the available light. Sky backgrounds are negligible at these levels.

REFERENCES

1. Mandel, L., 1963; Progress in Optics, Vol. II, pp. 181-248, Eq. 69 and Appendix B.

2. Chandrasekhar, 1943; Rev. Mod. Phys. 15, 1-89; Reprinted in N. Wax Ed. Noise and Stochastic Processes (Dover: New York, 1954) Eq. 16).

3. Johnson, H. L., 1966; Ann. Rev. Astron. and Astrophys. 4, 193-206; Table IV.

A SCHEME FOR STUDYING ASTRONOMICAL "SEEING"

R. H. Miller

1. Introduction

While many studies of "seeing" in astronomy are directed toward image stability and scintillation, the fluctuations of optical phase should provide a very sensitive measure of the phenomenon. A method is proposed for monitoring the instantaneous phase difference between two (or more) separated beams of light coming from the same star. Summaries of the phase-difference history at various separations will then give the structure-functions (or correlations) of the optical phase due to seeing. The frequency-dependence can also be determined. These functions, as studied for the optical phase, contain the corresponding information concerning scintillation and image stability.

The commonly accepted theoretical examinations relate the phase, scintillation, and image stability to refractive-index fluctuations caused by atmospheric turbulence. The theory is developed in several places; for example, Chandrasekhar (1952), Stock and Keller (1960), and Tatarskii (1961). With these models, the optical path length along the light ray fluctuates; this causes the incoming wavefronts to be distorted away from their usual plane wave shape. The shape of the incoming wavefronts changes irregularly with the time, although it has a short-term memory (of the order of tenths of a second), as is indicated by studies on the power spectrum of scintillation (for example, Mikesell, 1955).

From the standpoint of studies of atmospheric turbulence the method proposed is sensitive, but unfortunately gives a result related to the integral of the turbulence functions over the light path. Thus it bears only indirectly on questions of the universality of the turbulence functions and of the altitude dependence of the strength functions. Astronomically, however, this is just the information needed in phenomenological studies of seeing.

The discussion begins with the details of the optical phase detector, and then proceeds to show how this detector might be made into a practical instrument.

2. The Optical Phase Detector

Suppose that, by some means (to be explained later), light beams sampled from two separated (small) areas on an incoming wavefront (from a star) are conveniently available, each made parallel and with the two beams travelling parallel to each other. As long as the two sampling points on the incoming wavefront are not so widely separated that the star source is resolved (in the sense of a Michelson Stellar Interferometer), the two beams are coherent. Then, if they are put into an interferometer, the phase difference between the beams manifests itself. With reference to Figure 1, a detector

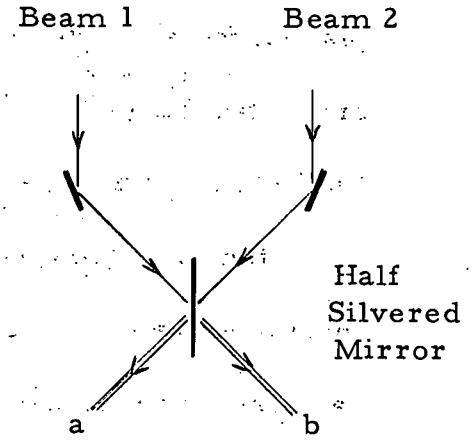

Figure 1. Beam-splitter optical phase detector.

on one side (a) will receive all the light with a certain phase-difference between the beams; if the relative phase is changed by π, all the light would go to the other detector. Intermediate phases would cause a division of the light energy between the two, the exact division depending on the relative phase.

Now, suppose that a quadrature phase could be obtained by some means: that half of the light in one of the beams could be shifted in phase by $\pi/2$, and that a separate measurement is made of the relative phase of Beam 2 phase shifted by $\pi/2$ and that of Beam 1. Then four detectors would give information on $(\phi_1 - \phi_2)$, $(\phi_1 - \phi_2 - \pi)$, $(\phi_1 - \phi_2 + \pi/2)$, and $(\phi_1 - \phi_2 - \pi/2)$. This result may be put into a standard fringe-counting type of apparatus, such as is used in laser length measurements, for example. This kind of equipment functions as follows: these four quantities may be combined in pairs to tell whether $\text{Cos}(\phi_1 - \phi_2) > \text{Cos}(\phi_1 - \phi_2 + \pi)$ (a) (a statement that is either true or false), and $\text{Cos}(\phi_1 - \phi_2 + \pi/2) > \text{Cos}(\phi_1 - \phi_2 - \pi/2)$ (b). Now, the "truth-values" of these two statements pass through a sequence of possible values in a certain order as the phase difference, $(\phi_1 - \phi_2)$ increases $(ab \to a\bar{b} \to \bar{a}\bar{b} \to \bar{a}b \to ab)$, and passes through this same sequence in the opposite order as the phase difference decreases. It is a simple matter to design logic circuitry that will generate a "count-up" signal for any change in the direct sequence and will generate a "count-down" signal for any change in the reversed sequence. If these are tallied in a reversible counter, the number of quarter-cycles of optical phase change between the two beams is stored

in the reversible counter at any time. A readout of this counter at various times records the history of the phase difference between the two beams.

There are some complications that must be allowed for in any practical embodiment of this device. These will be discussed later.

An experiment to study the phase-relations of the two beams, as it relates to the astronomical "seeing", consists of taking histories of the phase difference between beams obtained at various separations of the beams. These can later be analyzed to provide such quantities as the maximum phase excursion between the two beams, the cross-correlation of the phases, the power spectrum of the phase difference, the maximum rate of change of the phase difference, the rms phase difference, and so on.

As was noted earlier, this fringe-counting technique is standard in laser length measurements; the important difference is the amount of available light and the degree of coherence in the present experiment. Studies of phase changes in the propagation of laser beams in turbulent air have been reported by Buck (1967); again the amount of light and the degree of coherence are quite different.

3. Obtaining the Quadrature Phase

The quadrature phase might be obtained by putting a $\lambda/8$ step in the middle of the full mirror that bends one of the beams. To operate correctly, this beam must be split into two portions of

approximately equal intensity. Because the phases due to seeing might be different in geometrically separated regions, it is preferable to produce two beams differing in phase by some other means.

Another way of producing two beams of nearly equal intensity is to pass a beam through a quarter wave plate. The quarter-wave plate analyzes the beam into two linear polarization states and introduces a relative phase shift of $\pi/2$ between the two states. The quarter-wave plate can be placed in one of the beams and can be compensated by a neutral plate in the other beam. The beams emerging from the half-silvered reflector can each be analyzed into the two linear polarization states characteristic of the quarter-wave plate before passing on to the detectors.

4. Sampling the Incoming Wavefront

Light for studying the physics of seeing can most conveniently be obtained from a star. A standard astronomical telescope may be used to collect starlight and to track the star. Two beams can be obtained by an appropriate pupil somewhere away from the focus. The detailed optical arrangement for obtaining and analyzing the beams is shown in Figure 2. Convergent light from a telescope enters from the top and reaches a focus at F, where a field-defining aperture and color filter may be placed. The beam then diverges to the desired size and is made parallel by a field lens L. A collimator might be placed between F and L to shorten this part of the optical path.

Light from the telescope primary is split into two halves whose beams travel in opposite directions by the movable mirror M, and spots to be sampled from the incoming wavefront are selected by the two aperture-plates A. Moving M in the direction indicated changes the spacing between the sampled points on the incoming wavefront without changing the size of the spots. The size of the spots is determined by the opening in the aperture plates, and may be adjusted according to the brightness of the star being observed.

The two mirrors R redirect the sampled beams to cross at a half-reflecting surface in the split prism H. On the way, one beam passes through a quarter-wave plate, Q, to be made into two beams out of phase with each other by 90°. The quarter wave plate is compensated in the other beam by a neutral plate Q'. The beams emerging from the half-silvered reflector H are analyzed into (linear) polarization states that correspond to the characteristic states of the quarter-wave plate by the two analyzing prisms (Nichols or Wollaston prisms) N. Finally, the intensity of each of the four light beams so produced is measured by one of the four detectors $D_1, \ldots D_4$. The differences $(D_1 - D_2)$ and $(D_3 - D_4)$ are compared to obtain the signals for fringe-counting.

5. Size of the Sampling Apertures

The sampling apertures should be as large as possible to admit as much starlight as possible. However, if the apertures are too large, the phase of the entering light will not be well-defined, because it will be jumbled by the seeing. The permissible aperture size may be determined by the

arguments of this section. The design suggested does not fix the aperture size--it may be empirically adjusted. But an estimate is useful to permit signal-to-noise studies for a determination of the limiting magnitude.

A wavefront of the incoming starlight is not plane. It is uneven--something like the surface of water with waves on it--with a peak-to-peak amplitude that depends on atmospheric conditions. An upper limit to this peak-to-peak amplitude may be estimated for routine observing conditions from the fact that Michelson and Pease were able to observe fringes; this implies that the phase-difference between well-separated beams does not appreciably exceed a coherence length of the light as seen by the human eye--about 10 wavelengths. The "wavelength" scale describing the size of the variations can be estimated from the observation that with small telescopes (4" or less) the "seeing" usually produces a reasonable Airy disc that moves around by amounts corresponding to angle-of-arrival changes on the order of a second of arc. This implies that curvature of the wavefronts is less than a wavelength over a 4 inch aperture; but the slope of the wavefront may be around 1 arcsecond, which corresponds to about a wavelength across the aperture. Over larger apertures (8" or more) the curvature may be important, as indicated by the loss of the Airy disc. This means that a 3-inch aperture should not show appreciable wavefront curvature, and, near the extrema of the equiphase surface, should always have small enough phase change across the aperture that the number of cycles is well defined. Unmatched slopes of the wavefronts

across the two apertures that are being compared leads to a loss of the effective fringe visibility. This makes the measurements more difficult.

The loss of visibility may be estimated as follows. Each point of the half-silvered mirror combines light from corresponding points in the two incoming wavefronts. The corresponding points are determined in a ray-optics sense. (This statement is only approximately true because of diffraction effects; these are not important in this problem.) The total intensity from all of the sampled part of the incoming wavefronts may be obtained by summing the intensities resulting from an elementary area on the half-silvered mirror as if these were independent.

Let R be the reflection-coefficient of the half-silvered mirror, and T be its transmission (as measured for the wave amplitudes). Let a_1 and a_2 be the amplitudes of the two incoming waves, ϕ_1 and ϕ_2 be their phases at the half-silvered mirror. Let α be the phase-difference between reflected and transmitted waves emerging from the half-silvered mirror. Then, for one of the detectors, say D_1, the contribution of this elementary area to the total recorded intensity is

$$dI = dA\left[R^2 a_1^2 + T^2 a_2^2 + 2RTa_1 a_2 \cos(\phi_1 - \phi_2 + \alpha)\right]. \qquad (1)$$

If the two light beams have a reduced coherence due to some other cause, the cross-term may contain a factor V $(0 \leq V \leq 1)$ that indicates imperfect fringe visibility. The quantities R, T, a_1, a_2, and α are assumed not to vary over the half-silvered mirror.

The phase difference $(\phi_1 - \phi_2)$ will vary from place to place over the sampled incoming wavefront. The phase-difference might be expanded in a Taylor series with only linear variations retained; this is equivalent to assuming that each of the wavefronts is locally plane and that curvature terms may be ignored. The coefficient of the linear term may be related to the direction of the normal to each of the incoming wavefronts $(\nu_1, \varphi_1, \nu_2, \varphi_2)$ (spherical angle coordinates). Then for ν_1, ν_2 small (they are of the order of 10^{-5}!), the phase difference is

$$\phi_1 - \phi_2 + \frac{2\pi\varpi}{\lambda}\left[\nu_1^2 + \nu_2^2 - 2\nu_1\nu_2\cos(\varphi_1 - \varphi_2)\right]^{1/2}\cos(\psi + \beta), \quad (2)$$

where now ϕ_1 and ϕ_2 are the phases of the two beams at the center of the (circular) sampling apertures at the half-silvered mirror, (ϖ, ψ) are two dimensional polar coordinates of a point in the aperture, λ is the wavelength of the incoming light, and β is an unimportant angle related to the azimuths of the two wave-normals.

When the intensity is integrated over the (circular) apertures, the integrals are just those appearing in Fraunhofer diffraction theory; the only difference is that the squaring has already been done. The

results can be written down immediately (see for example, Born and Wolf, 1959, Sec. 8.5.2): The effective visibility is reduced by a factor

$$\left[\frac{2J_1(kR)}{kR}\right] \qquad (3)$$

over the case where the two beams are parallel, with

$$k=\frac{2\pi}{\lambda}\left[\alpha_1^2+\alpha_2^2-2\alpha_1\alpha_2\cos(\varphi_1-\varphi_2)\right]^{1/2}. \qquad (4)$$

Similar integrals may be worked out for other aperture shapes; they will usually contain an awkward dependence on the azimuth angles of the wavenormals (φ_1, φ_2, and β). This is why a circular aperture is to be preferred.

If the visibility were to fall to zero, the fringe counter would not record changes in phase that might actually occur. A worst-case criterion should be applied--corresponding to $kR \sim 4\pi\alpha R/\lambda$ for some largest expected value of α. Taking this value to be 1 second of arc (5×10^{-6} radians) at $\lambda = 5000$ Å, the visibility function would fall to 1/2 at about a 1.5-inch diameter aperture. In any case, the apertures should be made as small as possible to get the required amount of light--but two to three inches is an expected maximum permissible size.

The formal similarity of this calculation to that of Fraunhofer diffraction is to be expected. The difference is that while in Fraunhofer

diffraction a coherent superposition is formed of the wavelets from various parts of the aperture, here an incoherent superposition is formed because the light is not brought to a focus. Inclusion of curvatures would result in terms that are formally similar to Fresnel diffraction.

This result may be used to derive a probability distribution for the fringe visibility after certain assumptions are made concerning the probability distributions of α_1, α_2, and $(\varphi_1 - \varphi_2)$. This point will not be further pursued here.

In making this calculation, other sources of low fringe visibility have not been explicitly included. A principal contributor to this loss would be a large phase-difference $(\phi_1 - \phi_2)$, if that carried the beams beyond a coherence-length. This may be avoided by using band-limiting filters. Filters of about 100 Å passband should allow fringe detection to about 50 wavelengths of phase difference between the two beams.

6. Signal-to-Noise Ratio in the Fringe Counts

It is commonly conceded that there are frequency-components in scintillation at frequencies extending as high as 1 kilocycle. A design goal of 1 kc has been adopted for preliminary signal-to-noise estimates.

Suppose each of the detectors generates a standard pulse for each recorded photoelectron. It will also generate identical pulses for dark current noise from a photomultiplier. If these standard pulses are fed into an RC integrating circuit with decay time constant τ, then, since the

standardized pulses arrive in some kind of Poisson process with an expected arrival rate of ν per second, the expectation value of the current is $\nu\tau$ times that of a single pulse, and the variance is $\nu\tau/2$. It is not straightforward to derive the distribution of the output current, but many of the features were discussed by Rice (1944); these results are drawn from that work.

The difference of two such signals may now be taken, and the condition that the expected difference current should exceed the (square root of the) variance in the difference leads to

$$\delta|\sin\varphi| \sim \frac{1}{2V\sqrt{\nu\tau}}, \qquad (5)$$

where V is the expected visibility and φ is the phase angle sought (measured from a suitable origin). Thus, if $\nu\tau \sim 100$ (100 counts in an integration time), the phase angle is fairly well determined except if it is within about 3° of zero phase (taking $V = 1$): at $\nu\tau \sim 10$, the phase error is about 9°.

With the design goal of 1 kc, $\tau \sim 1$ millisec, and $\nu\tau \sim 10$ at $\nu \sim 10^4$ photoelectrons per second. This will typically be large compared to the dark current even of room-temperature photomultipliers. Using standard astronomical figures, at 100 Å passband, 10% quantum efficiency for the photomultiplier, a 3" diameter aperture should provide 10 kc for ν at about magnitude 2.5. One magnitude has been allowed for other losses in the optics and for atmospheric transmission losses. There are quite a few stars this

bright to use as light sources. Thus, the apertures and bandwidth goals seem realistic.

It is interesting that frequency components that lead to phase changes much less than the 1/4-cycle seen by the phase detector may be measured because the other larger-amplitude components modulate them, and reduce the problem to the detection of a sine wave in the presence of a great deal of noise. Such detection is routine in radio and radar astronomy applications.

The use of the differences $(D_1 - D_2)$, $(D_3 - D_4)$ frees the detection scheme from amplitude scintillation; only second-order amplitude terms enter in the determination of these differences.

7. Circuits and Recording

The difference-signals generated by the circuits described in the previous section may be fed into a set of logic circuits that compare a change in the state of the assertions, a, $\overline{a}$, b, $\overline{b}$ of Sec. 2 with the present state to generate a "count-up" or "count-down" signal. The difference may be tallied in a reversible counter. The reversible counter may occasionally change at a faster rate than the 1-kc design goal because of the stochastic character of its inputs. Because the logic depends upon comparing changes with present states, it will have a certain dead time (likely just a few nanoseconds with present-day circuits); again, the stochastic inputs might cause generation of both "count-up" and "count-down" signals within this resolving time. If so, the circuits should warn the observer that the phase count may be in error by a quarter-cycle after such an event. The circuits

are being designed to generate an "error" signal should this happen. These will be monitored along with the reversible phase count.

A small computer such as a PDP-8 may be used to keep a record of the phase difference count and of any error conditions. The computer can read the reversible counter and the error counter at about millisecond intervals. Magnetic tape transcriptions of the readings will provide the history of the phase change referred to earlier.

Post-run summaries of the phase changes will permit reduction of the histories to a usable form. For each of the various separations, we expect to work out the various quantities mentioned in Sec. 2.

REFERENCES

1. Buck, A. L., 1967 Applied Optics 6, 703.
2. Chandrasekhar, S. 1952: Mon. Not. Roy. Astr. Soc. 112, 475.
3. Mikesell, A. H. 1955, The Scintillation of Starlight (U. S. Naval Observatory, Washington, D. C.)
4. Rice, S. O. 1944, Bell Syst. Tech. J. 23, 282, Reprinted in Selected Papers on Noise and Stochastic Processes N. Wax, Editor, (New York; Dover Press, 1954).
5. Stock, J. and Keller, G., 1960 in Telescopes, G. P. Kuiper and B. M. Middlehurst, eds. (Chicago: University of Chicago Press), p. 138.
6. Tatarskii, V. I., 1961 Wave Propagation in a Turbulent Medium (Translated) (New York: McGraw-Hill Book Company).
7. Born, M. and Wolf, E. 1959 Principles of Optics (New York: Pergamon Press).

FIGURE CAPTION:

Figure 2: Arrangement of the optical phase detector. See text (Sec. 4) for detailed description.

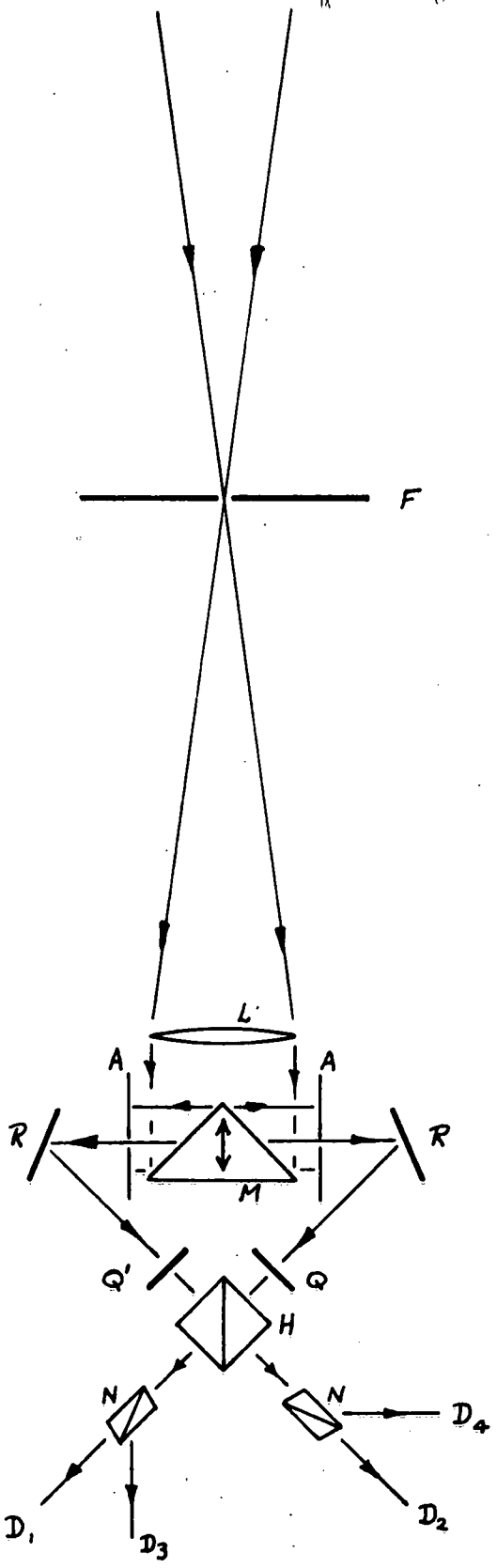

Fig. 2