IMM 397
May 1973

Courant Institute of
Mathematical Sciences

ASL: A Proposed Variant of SETL
(Preliminary)

H. S. Warren, Jr.

Prepared under Grant No. NSF-GJ-1202X2
with the National Science Foundation

New York University

IMM 397
May 1973

New York University

Courant Institute of Mathematical Sciences

ASL: A Proposed Variant of SETL

(Preliminary)

May 1973

H. S. Warren, Jr.

Results obtained at the Courant Institute of Mathematical Sciences,
New York University, with the National Science Foundation,
Grant No. NSF-GJ-1202X2.

PREFACE

When the Algorithm Specification course was first given in the spring term of 1971, Professor Schwartz asked the class for criticisms of SETL. Here, some two years later, is my response.

I have tried to be as constructive as possible. Rather than simply say "this is wrong" and "that is wrong", I have tried to say "here is a better way to do it."

There were a few things that struck me as "wrong" in SETL right from the beginning. This includes the SETL treatment of numbers, tuples, sets used as maps, and the meager I/O facilities. In considering how these facilities should be changed, I soon found myself contemplating changes to just about every data type, operator, statement type, and syntactic construction in the language. The criticism has snowballed into a complete respecification of SETL.

In fact, since I am questioning every detail of SETL, I might as well question its name, too. Not that SETL is so bad a name, but it seems to me that it would be preferable for the name to reflect the true essence of the language. The essence of SETL is that it is an executable algorithm specification language. Since that is a bit too much, I suggest the name Algorithm Specification Language, or ASL. The fact that SETL and ASL achieve the objective of permitting a concise but humane specification of algorithms in part by using sets is merely incidental. In fact, it happens

that ASL uses sets to a greater extent than SETL does, because the SETL data types of Boolean strings, character strings, and tuples are all sets (maps) in ASL.

In spite of the fact that I am suggesting changes to practically every detail of SETL, this respecification is not really that big a change. The goals of ASL are the same as those of SETL: it remains a tool for furthering computer science by pushing the complexity limit of what can be programmed, a vehicle for communicating algorithms to people for the purposes of research or classroom instruction, and a practical means of prototyping complex software systems.

The key features that give SETL its great expressive power are present in ASL. These features are, in my opinion, (1) the set of ordered pairs of arbitrary objects used as a map, (2) the total absence of storage management dictions, (3) the "value oriented" (rather than "pointer oriented") character of SETL, (4) the nearly total absence of required declarations, and the consequent existence of programs that operate on data structures independently of their elemental nature, (5) the arbitrary set and tuple data types, (6) the universal and existential predicate expressions, and (7) the set former expression.

The syntax of ASL remains very close to SETL, in spite of the fact that some change is suggested to just about every syntactic construction. At the end of this paper are a few algorithms coded in both SETL and ASL, and a glance at these reveals how similar the languages are. ASL is

still SETL just as FORTRAN IV is still FORTRAN and Algol 68 is still Algol.

If ASL is so similar to SETL, then why bother with this respecification? I believe that this is worth doing just as FORTRAN IV and Algol 68 were. This is particularly true because SETL is still young and there is not yet any user community with an investment in the existing SETL. So if there's anything wrong with SETL, it's much easier to change it now than later. Section 15 summarizes what I think can be improved in SETL, and contrasts it with ASL.

This respecification is at present very incomplete. All that is done here is the easy work: the respecification of the data types, expressions, statements, procedure linkage, etc. A glaring deficiency is that no work is done here on conversational and real time aspects. This seriously limits ASL's use for most of the largest programming projects, which is the very area where ASL's potential is greatest.

Other major deficiencies are that no work is done here on language extension facilities (such as user-defined data types), a preprocessor, implementation considerations, and the intermediate language.

But incomplete as this is, it is hoped that it will be a positive contribution to the development of SETL.

CONTENTS

*: Not supplied yet.

*: Not supplied yet.

*: Not supplied yet.

0. Overview

Data Types

Number	1, 1.2, 1.3E100, 3/5
Character	'a'
Boolean	TRUE, FALSE
Pointer	↑expr
Procedure	fun, SQRT
Set	{1, 2, 3}

Special Cases of Sets

Relation	{(a, b, c), (d, e, f), ...}
Map	{(a, b), (c, d), ...}, first components unique
Array	{((1, 1, 1), a), ((1, 1, 2), b), ...}
Matrix	{((1, 1), a), ((1, 2), b), ...}
Vector	(a, b, c) = {(1, a), (2, b), (3, c)}
Character string	'abc' = ('a', 'b', 'c')
Boolean string	'101'B = (TRUE, FALSE, TRUE)

Operators

Equality test	a = b, a ≠ b
Existence test	∃e
Arithmetic	+, -, *, /, **, |x|, !
	These apply to maps, matrices, etc., in the usual mathematical way.
Numerical compare	<, ≤, >, ≥
Boolean	&, ∧, ∨, ⊻, ¬, ⇒, ⇏, ≡, ≢
Pointer	↑e, ↓p
Set	#, x ∈ S, x ∉ S, ∪, ∩, -, ⊂, ⊆, ⊃, ⊇, ∃S
Relational (binary)	𝒟, ℛ, ℒ (Domain, Range, and Inverse)
Array and Vector	LI, HI, LENGTH, ¢, @

The user may define additional binary infix operators.

Itemized Map Referencing

If f = {(1, 2), ('a', 3), ((4, 5), 6)}
then f(1) is 2, f('a') is 3, f(4, 5) is 6, and f(7) is undefined.
After "f(1) = 8; f(9) = 10;", f is {(1, 8), ('a', 3), ((4, 5), 6), (9, 10)}.

Procedures

May be invoked recursively.
May be invoked from either right-hand or left-hand sides.
Call by value; for right-hand calls no argument return.
Variable and arbitrary number of parameters is possible.

Set Formers

Enumeration	{a, b, c}
Iterative	{e(x), $\forall x \in S \mid C(x)$}
	{e(x, y), $\forall x \in S, \forall y \in R \mid C(x, y)$}
	{e(i), $1 \leqslant \forall i \leqslant n \mid C(i)$}
	{e(i), $\forall i = 2, 4, \ldots, 10 \mid C(i)$}
Null set	$\emptyset$

Vector Formers

(a, b, c)
(e(x), $\forall x \in S \mid C(x)$), etc.

Search Expressions

$x \in S : C(x)$	$i > 0 : C(i)$	$x = 2, 5, 6 : C(x)$
$m \leqslant i \leqslant n : C(i)$	$x = 2, 4, \ldots, n : C(x)$	

Universal Predicates

$\forall x \in S : C(x)$	$\forall i > 0 : C(i)$	$\forall x = 2, 5, 6 : C(x)$
$m \leqslant \forall i \leqslant n : C(i)$	$\forall x = 2, 4, \ldots, n : C(x)$	

Existential Predicates

$\exists x \in S : C(x)$	$\exists i > 0 : C(i)$	$\exists x = 2, 5, 6 : C(x)$
$m \leqslant \exists i \leqslant n : C(i)$	$\exists x = 2, 4, \ldots, n : C(x)$	

Cross Sections

f(a, *) is $\{(x(2), y), \forall (x, y) \in f \mid x(1) = a\}$
f(*, a), f(a, *, b), etc., have similar definitions.

Subarrays

f(m:n) is (f(m), f(m+1), ..., f(n))
f(m:), f(:n), f(a:b, c:d), etc., have similar definitions.

IF Expressions

```
y = IF x ⩾ 0 THEN x ELSE -x;      Valid on left-hand side also.
```

DECLARE Statement

Used mainly for efficiency enhancement and machine-readable commentary; ASL is basically declaration-free.

```
DECLARE (x, y) NUMERIC, S STATIC,
        M MAP ((CHAR, ...), INTEGER),
        epsilon 0.001;
```

Assignment Statement

```
x = expr;   f(x) = expr;
f(g(x)) = expr;   means   t = g(x); f(t) = expr; g(x) = t;
(x, y, z) = (1, 2, 3);
x = y = z = expr;
expr;   means   expr = TRUE;   Example: x ε S; means S = S ∪ {x}.
```

Input/Output Statements

```
READ x, y, z;
READ i, a(i) FORMAT(N(5), A(12));
READ x, y, z FILE f COPY g POSITION(NL);
READ x, y STRING c;
READ x, y NAMES;
WRITE x, y+3 FORMAT(PAGE, A, N(5));
PAGE FILE f MARGIN(11, 80) AT 51 DO
                           WRITE PAGENUM(f) FILE f ...
                           FORMAT(LINE(53), X(42), N);
                           WRITE FILE f POSITION(PAGE);
                           END
FORMAT[NL, A(10), 6 N 4, S, V];
```

Control Statements

```
RETURN; QUIT; CONTINUE; STOP; EXECUTE L; GO TO L;
null.
```

Statement Brackets and Headers

```
DO ... END, BEGIN ... END, ( ... ), [...]
INITIALLY DO k = 0; WRITE POSITION(PAGE); END
IF c THEN s1 ELSE s2

IF    c1,  'TTF'
      c2,  'TFT'
THEN(s1    'X  '
      s2   'XX '
      s3   '  X')
ELSE s4

CASE c(i)
   '(': '[': n = n + 1;
   ')': ']': n = n - 1;
   '|':      (n = n + sw;  sw = -sw);
             END CASE c(i);

STARTING s1 for-specification WHILE c DOING s2 s3
∀x ε S | C(x) DO ... END
1 ⩽ ∀i ⩽ n | C(i) DO ... END
∀i = 2, 4, 9, n+3 DO ... END
∀i = 2, 4, ..., 100 | C(i) DO ... END

Set of first 100 primes:
STARTING (S = ∅; k = 1) ∀p ⩾ 2 | [2 ⩽ ∀i < p : REM(p, i) ≠ 0]....
   WHILE k ⩽ 100 DOING k = k + 1;  S = S ⋃ {p};

FUNCTION f(x, y, z); ... END f;
OPERATOR x .OP. y; ... END .OP.;
```

Name Scoping

Variables are local unless explicitly declared as external/shared (two-way cooperation is required).
Variables are stacked on recursion unless explicitly declared as STATIC.

Preprocessor

Similar to that of PL/I.

Elaboration Facilities

Machine readable commentary, e.g.:
DECLARE 1 ⩽i ⩽1000 INTEGER,
f(x, y, z) ASSIGNS (x, z),
g PROCEDURE MAP(NUM, CHAR);

Debugging Facilities

DECLARE 1 ⩽i ⩽1000 INTEGER VERIFIED;
TRACE(x, y); STOP TRACE(x); (like the PL/I CHECK prefix).

Input and Listing Control Commands

EJECT; MARGIN(i, j); NEATER; (=INDENT + OVERPRINT).

Built-in Procedures

SQRT(x), SET(x), PAIR(x), POW(S), COMPILE(text), etc.

1. Concepts

ASL is an executable algorithm specification language. It is intended for use as a research tool for developing and experimenting with complex algorithms, as a teaching aid in communicating algorithms to others, and as a vehicle for constructing prototypes of large software systems.

ASL is a revision of the SETL (Set Theoretic) language developed at New York University. It is a procedural language, and has been influenced most strongly (aside from SETL) by PL/I, APL, and Algol '68. However, it is somewhat closer to conventional mathematics than those languages, in both writing style and facilities provided.

ASL is a substantially higher level language than PL/I, APL, and Algol. As such, it makes tractable the coding of algorithms that might not even be attempted in the lower level languages. It is less machine dependent and hopefully will be less implementation dependent than the others.

For the most part, this paper presumes that the reader is an experienced programmer. No knowledge of SETL, however, is assumed.

1.1 Applications

ASL is applicable to roughly the same class of problems as PL/I without its business oriented and multitasking features. Its field of application is limited chiefly by its lack of efficiency: production programs will probably not be coded in ASL until the ratio of hardware to software costs decreases by an order of magnitude or so. For programs involving simple operations on arrays and strings, the ASL program will probably

execute from ten to a hundred times slower and will require about five times more storage.

For programs involving complicated data structures, it is possible that the ASL program will execute faster than a corresponding PL/I program that one is likely to take the trouble to write. This is because of techniques such as hashing that are built into ASL.

Some programming areas for which ASL is well suited are:

1. Compiler algorithms
2. Combinatorial algorithms
3. Operating systems
4. Formula manipulation
5. Numerical analysis
6. Linear programming
7. Problems involving graphs (e.g., directed graphs)
8. Miscellaneous computer-oriented algorithms (bit-parallel operations, pointer operations, etc.)

1.2 Principles of Design

ASL is constructed from as few basic ideas as is practical. This makes the language easier to learn, and minimizes the necessity of referring to the reference manual when coding. In keeping with this, there are only five basic data types, one data aggregate, and a total of nineteen statement types and statement "headers".

The full ASL character set is rather large, and there are a large number of operators. It is hoped that this will improve the clarity of algorithms while maintaining conciseness. Many of the operators and statements are defined in terms of more basic ones, but no attempt is made to rigorously isolate a "basic" subset of ASL.

The unifying concept is readability --- by people, rather than machines. The readability is achieved not by wordiness but by borrowing the modern mathematical style that has been evolving for hundreds of years. In particular, the notations of set theory are extensively used. As an example, if S is a set of numbers then the set of positive numbers obtained from S is written

$$\{x, \forall x \in S \mid x > 0\}.$$

One reason for stressing readability is that ASL is an algorithm specification language. That is, the main purpose of coding an algorithm in ASL is to communicate the essential features of the algorithm, in a rigorous and machine-verified manner, to people. The algorithm is written to be read by computer science researchers, by system analysts,

by system implementors, and by students. The fact that the algorithm can be executed by a machine is usually of secondary importance. It is executed mainly to verify its correctness by trying it on a large number of test cases with the unerring accuracy of which only a machine is capable.

The principle of readability has many effects on the language. Perhaps greatest is the implication that side effects are to be minimized. For example, in ASL a function cannot alter its arguments, in a normal call. This rule is imposed because an algorithm is significantly easier to understand if one knows a priori that a reference to f(x) cannot alter x. This restriction also keeps ASL closer to the mathematical notion of function, it avoids such sticky issues as the meaning of "a + f(a)" if f can change a, and it helps in the optimization of programs by a compiler.

Another common source of unpleasant side effects has been eliminated because in ASL, variables are not declared to be of any particular type. Wheras in PL/I the statement "X = 1;" would cause the value of X to be four blanks if it had been declared to be a character string of length four, in ASL relatively few such surprises occur. The meaning of "X = 1" is simply to make X a numeric quantity with a value of 1, regardless of the context. Nearly all data conversions must be explicitly called for, which is usually an advantage from the point of view of the reader.

The principle of readability also leads to keeping ASL procedures as self-contained as is practical. That is, the reader should be able to understand an ASL procedure in detail when he only has a vague understanding of the procedures which call it and the procedures which it calls.

Some of the implications of the stress on readability are that the flow of control via the "GO TO" is limited to be within the procedure, facilities are provided to encourage a highly structured or "go-to free" coding style, and assignments are easily visible. Pointer variables are treated in a way that prevents their changing the value of an ordinary variable. Assignments always involve a copy operation, at least conceptually. That is, even if A and B are aggregates, the statement "A = B" makes the value of A a copy of B; if either A or B subsequently changes in value, it does not affect the other.

Much of the above can be summarized in the single word "simplicity".

The language features desirable for the sake of readability coincide very nearly with the features desirable to enhance compiler optimization possibilities.

It will be found that coding for readability will make algorithms somewhat harder to write and less efficient than they might otherwise be. But these drawbacks are not serious. It is still far easier to code in ASL than in PL/I or Algol. The optimization possibilities will help to compensate for the inefficiencies brought about by such features (or lack thereof) as the restricted "go to".

Although an attempt has been made to minimize side effects, they have not at all been eliminated. For example, an ASL function might return the next prime number each time it is called. Invoking such a function then has the side effect of incrementing a kind of counter internal to the procedure. Functions can also have side effects through input/output activity.

Perhaps the greatest source of side effects in ASL is due to the fact that a procedure can share variables in a manner similar to FORTRAN COMMON. However, such sharing is made explicit in both the calling and the called procedures, and thus the reader is less apt to be caught unawares than he is with many programming languages.

1.3 Summary of ASL

1.3.1 Data Type Handling

Variables in ASL are not normally declared to be of any particular data type. A variable acquires a data type by assignment to a self-defining value, and the data item's type and other characteristics are passed on unaltered to other variables by assignment.

There is, however, a DECLARE statement in ASL. It provides information about a variable that cannot be supplied in any other way, such as the external and the static (with respect to recursion) attributes. The DECLARE statement may also be used in a purely informative way. For example, DECLARE K NUMERIC says that the values taken on by K will always be numeric. It does not cause any conversions of data types to force K to be numeric. Such a declaration has no effect on the operation of a program. It might be included to help the human reader's understanding of a program, and to help the optimizer to produce efficient object code.

The data types in ASL are the number, the single character, the truth or Boolean value, the "pointer" or reference value, the

procedure value, and the set. The first five are the "atomic" data types and the last, the set, is an unordered collection of items of any data type. From the set the familiar data types such as character strings and arrays may be constructed. Because of the generality of the set, one is not bound by the usual restrictions when composing data aggregates. For example, arrays may be sparse and they need not be homogeneous (that is, their elements need not be of the same data type).

1.3.2 Storage Allocation

The set and its special cases such as arrays and strings are assumed to be of varying size, and there is no limit to the size except the availability of main storage. This and the possibility of recursion imply that the ASL implementation must be based on a dynamic storage allocator.

There are two types of storage requests. These might be handled with different strategies for reasons of efficiency. To support recursion, a simple stack-like allocation and freeing mechanism is needed, for the stacking of the values of local non-static variables. For the storage of data structures, a capability to support random allocation and freeing requests is needed.

Both types of requests can be handled by a general capability such as the OS/360 GETMAIN/FREEMAIN. Another reasonable implementation would be to partition the dynamic storage area into two

areas: a "stack" and a "heap". Allocation and freeing from the stack area simply involves moving a pointer up and down (and checking for stack overflow). The "heap" area would be used for storing data structures, and it could be managed in any of a variety of ways, such as the GETMAIN/FREEMAIN strategy, the "buddy system", or a scheme based on a garbage collector, possibly with a compaction capability to avoid fragmentation. There is nothing inherent about ASL, however, that requires or even particularly suggests that a garbage collector be used.

1.3.3 Input/Output

The ASL I/O capabilities are modeled after PL/I, but are simplified. That is, the PL/I basic modes of stream I/O are included in ASL, but various efficiency related issues such as explicit control over the buffering strategy and the physical block size are not included.

The file being read or written consists of a (generally very large) character string. Conversion routines are automatically invoked to convert portions of the character string (or "I/O stream") to and from internal ASL format. An I/O file is normally strictly input or strictly output. An output file is generally intended to be printed or otherwise displayed for reading by humans, but it may be used as a "scratch" file, for computer-to-computer transmission, or as an intermediate file to be read by programs coded in another language. Changing from a

READ to a WRITE or vice versa causes the file to be closed and "rewound" if possible.

There are three varieties of I/O: simple, name, and format directed. In simple mode, only the item's value is written out or read in, and it appears on the external medium in exactly the same form as an ASL self-defining value (for example, a character string is written out with surrounding quotes).

In name mode, data on the external medium resembles an ASL assignment statement, with the data item's name (as known in the procedure containing the READ or WRITE statement), an assignment symbol (=), the value as it would be written as an ASL self-defining value, and a statement separator character (;).

In both simple and name modes, the number of characters used in the output medium depends upon the values being written. The format directed mode of I/O allows the programmer to explicitly control the exact number of characters to be written. In addition, control can be exercised over when a new line or page is begun, and certain details of the manner of converting items between internal format and character strings.

1.3.4 Compile Time Facility

The ASL compile time facility, or macro preprocessor, is modeled after that of PL/I. Many ASL statements, such as assignment, IF, GO TO, READ, WRITE, etc., can be prefixed by a percent sign (%), which causes them to be executed at compile time. A simple assignment such as %A = '1' can be used to substitute a string such as '1' for all occurrences of the name A that follow. The %IF and %GO TO permit conditional compilation, i.e., the deletion or repetition of blocks of code. The %READ allows blocks of code to be brought in from a file (as the PL/I %INCLUDE). The %WRITE allows the placement of code on a file for compilation at a later time, and the writing of diagnostics about the preprocessing phase of compilation.

In addition, facilities are provided for the coding of preprocessor procedures, which may be recursive. A symbol generation capability is provided to permit the introduction of "compiler temporaries" and generated labels. With these facilities, the programmer can, for example, define his own loop specification and have the preprocessor translate it into ASL.

Like the PL/I preprocessor, the ASL preprocessor is largely ignorant about the syntax of the text it is processing. It can recognize an ASL token and can distinguish which are identifiers (those that begin

with a letter), but that is about all. Thus the preprocessor can be used to some extent to process text that is not an ASL program.

1.3.5 Normal and Elaborated ASL

A working ASL program may be elaborated upon by adding statements that describe certain features of the program. These elaborations serve two purposes: to aid the human reader in comprehending a program, and to aid the optimization possibilities. The elaborations never change the results of a program, and thus they may be ignored if desired.

Most of the elaborations consist of DECLARE statements. One can specify, for example, that the values taken on by a variable are always numbers, and furthermore that they always fall within a specified range. If a variable's values are always character strings of fixed length or of some predictable maximum length, then this may be specified. One may specify that a procedure is free of side effects (thus avoiding a bottom-up compilation order to have this determined by the compiler), or that at any rate if it does have side effects, they are inconsequential and may be ignored for program optimization purposes (a procedure that contains statistic counters for self-analysis would probably fall into this category).

These elaborations can also be used as a debugging aid. By giving a variable the VERIFIED attribute, the compiler will generate code at each assignment to the variable to verify that all information given about the variable is satisfied, and to terminate processing if

it is not.

The general nature of the elaborations is to merely add information about a program. The elaborations do not explicitly state how to implement something. This limits the possibilities of the elaboration features to some extent, but it keeps them machine independent, at a high level, and of positive value to the human reader. As a simple example of this point, we do not specify "store x in a half word". Instead, we specify that the values of x range from 0 to 1000, or whatever.

The elaborations may all be grouped together and inserted at one point in a procedure. Except for this one insertion, there is no need to modify any text in the program.

1.3.6 Features Not Included

There are many important features found in other programming languages that are not included in ASL. Most of these features are directly aimed at efficiency questions, some are aimed at making programs easier to write at the expense of readability, and some are oriented toward applications which are either too specialized or to which ASL is not well suited anyway. Lastly some features are omitted simply because to include them would be to give ASL too ambitious a beginning.

Some of these features which are absent from ASL are:

1. Parallel or interleaved processing, e.g., multitasking, reentrancy, semaphors, etc.
2. Discrete event simulation language features.
3. Conversational programming features.
4. Teleprocessing capabilities.
5. Program fetching or overlays.
6. Interruption processing.
7. Complex numbers.
8. Explicit control over the precision of real numbers.
9. Explicit control over implementation details.
8. Extreme conciseness (as in APL, and to some extent Algol 68).
9. Intricateness, such as achieving brevity of expression through the exploitation of side effects.

1.3.7 Program Compilation

An ASL program is a collection of procedures of the function and operator types. Procedures may contain other procedures in a properly nested manner. Each outermost procedure may be separately compiled. The translation of a source program into executable form involves two intermediate languages, IL1 and IL2. Translation to IL1 is done by a "preprocessor", which is analogous to an assembly language macro expander. The programmer may control this preprocessing stage by means of statements written in the preprocessor's language. This language is similar to ASL proper, and the preprocessing control statements are identified by their beginning with a percent sign (%). A listing of the resulting program coded in IL1 may be obtained. The program at this point looks very much like the original source program. In fact, an entire program could be coded directly in IL1, although this would not normally be done.

Besides acting on programmer-supplied preprocessor control statements, the preprocessor expands many ASL statements into more primitive ones. For example, an iteration specification is converted into IF statements, GO TO statements, etc.

The program is next translated into the other intermediate language, IL2. In this form, the program is essentially compiled. It exists in a highly coded form that is only slightly above the assembly language level, but it is still machine independent. Arithmetic expressions

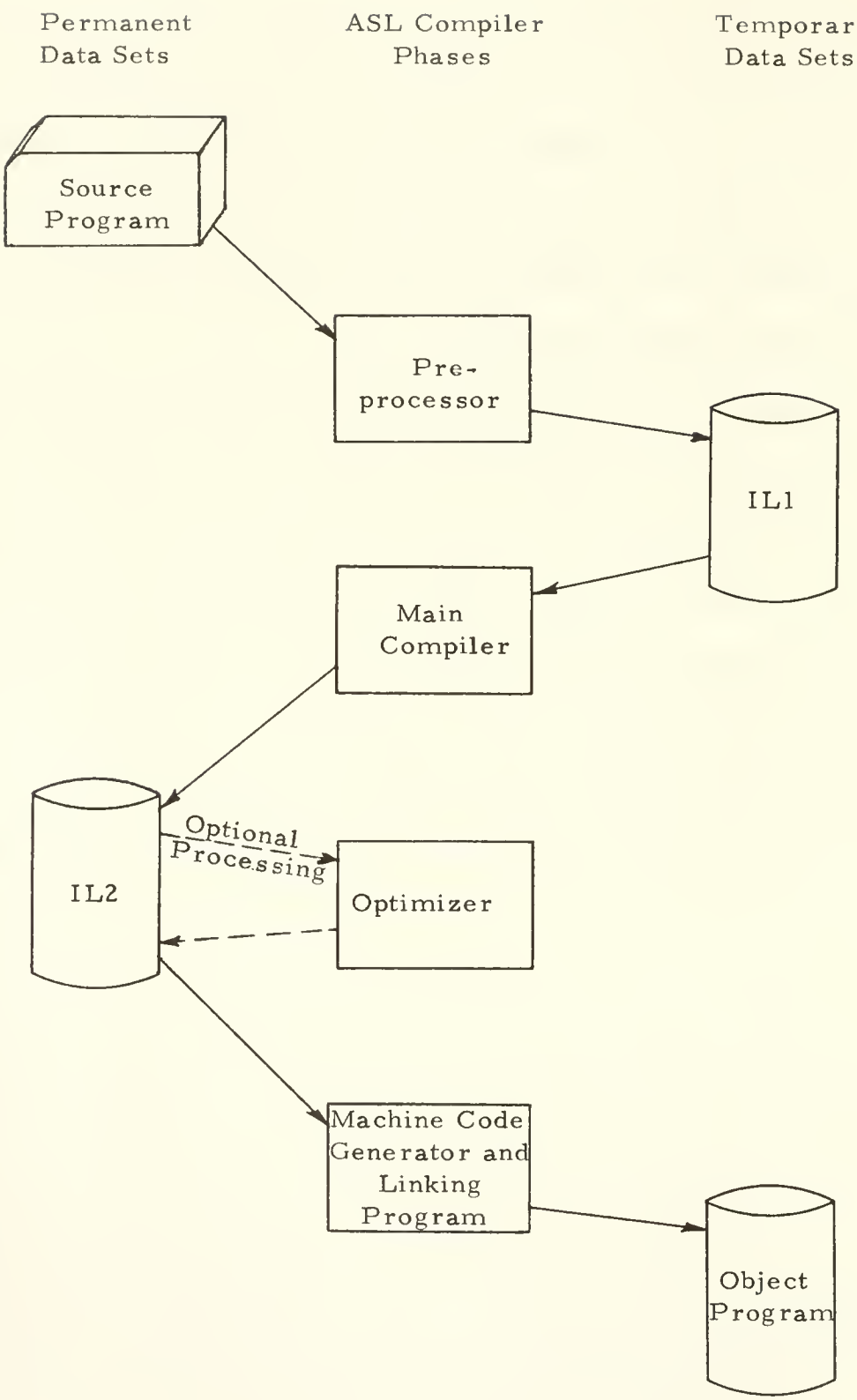
have been parsed, and they appear as a series of adds, multiplies, etc., with explicit references to "compiler temporaries". Statements such as READ and WRITE appear as library routine calls, as do many operations, such as the set membership test. If a procedure is to be saved for later linking with other compiled procedures, it is generally saved in IL2 form. A listing of the procedure in IL2 may be obtained.

At this point, the procedure may be combined with others and translated into machine code for execution. Alternatively, a global optimizer may be brought in. This optimizer has IL2 for both its input and its output; it is thus capable only of machine independent optimizations. The optimizer eliminates common subexpressions, factors code out of loops, etc., and produces the transformed IL2 procedure augmented with certain information about it. This information includes a list of external variables and whether they are set or merely used, a list of procedures called by the procedure, etc. This information may be printed out as documentation on the program. It is also used by the optimizer when processing higher level procedures. That is, the greatest optimization can be achieved by compiling all procedures in a program from the "bottom up", each time informing the optimizer (by means of job control language statements) where the lower level procedures reside. The optimizer cannot move code from one procedure to another, but it can make use of knowledge of a called procedure's side effects. If recursion is involved, it is necessary to optimize

some procedures twice to achieve greatest optimization. For example, if A calls B and B also calls A, then one might first compile and optimize A. Then when B is compiled and optimized, the optimizer can use the information about A that has been saved. Then recompiling A, allowing the optimizer to use information that is now available about B, might yield a more efficient procedure A.

When one procedure is contained within another, all procedures are compiled together. The optimizer then has greater opportunities, as it may move code from one procedure to another. This is possible because in such a case the optimizer knows all the places a procedure is called from (an internal procedure cannot be called from outside the containing procedure, unless that possibility is made explicit by means of the SHARED attribute in a declaration statement).

The ASL translation process is illustrated on the following page.



ASL Program Compilation Process

2. Program Elements

This section discusses the structure of an ASL program, with the emphasis on syntax rather than semantics. First the ASL character set will be described, and a way will be given to approximate the ASL character set with two widely used computer codes, EBCDIC and ASCII. The manner of building characters into "tokens" will be described, as will the terminology of the various types of tokens. Next it will be shown how to put tokens together to form statements, how to put statements together to form statement groups and procedures, and how to put procedures together to form a complete program.

2.1 Character Set

2.1.1 Language Character Set

As with all computer languages, the ASL character set is a compromise between the ideal world of the mathematician and the limitations of such hardware as typewriters and keypunch machines. We prefer the ideal world assumed by the mathematician, as a rich character set helps to achieve clarity through conciseness. Certainly $A_1 + B$ is clearer than A SUB 1 PLUS B; in ASL we have the usual compromise A(1) + B.

For the benefit of published algorithms, the ASL character set is larger than both EBCDIC and ASCII. The additions consist of simply more characters, however, there being 120 in all. The "full" ASL does not employ underlining, bold face type, italics, superscripts, etc. Hence an ASL program written in the full character set:

(1) can be typed on a standard typewriter with a minimum of hand work, backspacing, etc.,

(2) can be encoded in a direct way (for storing in a computer or transmitting in digital form) provided at least seven bits per character are available, and

(3) could be printed on a computer's printer if special printing elements were made up.

The full character set cannot be accomodated by making minor alterations to a conventional typewriter, unfortunately, because typewriters are generally limited to about 44 printing keys and two shift positions, which gives a maximum of 89 characters (counting the blank).

The table at the end of this section shows the full character set and indicates briefly how each character is used. An indication of the alternative for EBCDIC and for ASCII is also given.

An ASL character is either an alphanumeric character or a special character. The alphanumerics are the digits and letters, and the letters exist in both upper case and lower case form. The special characters are sometimes grouped according to most common usage, such as "arithmetic operator" and "grouping symbol". The terminology and hierarchy are given at the end of this section.

There are some symbols that might normally be considered "letters", such as ∅, ∇, and possibly ∀, but these are defined to be special characters in ASL. The significance of this is that these characters cannot be used to make up names.

The new-line character has no graphic. It indicates the beginning of a new line, and it cannot appear in any other place in source text (not even in comments and character string constants). It is accessible only through the predefined variable NL and input/output operations.

The new-line character indicates that a character string, as represented on an external medium, has begun a new line. This applies to the compiler's reading and printing of text, and to the ASL program's reading and printing of text. If the medium for source text or data is a typewriter-like device, NL is generated by depressing the carrier return. If the input medium is punched cards, an NL character is inserted between card images by the input/output control program. On printing, an NL character in the text causes a new line to be begun (unless the device is already at the beginning of a new line because of a hardware limitation on line size; see section 9, Input/Output).

The characters ∅ and ♭ are in the somewhat unusual category of "special constant". These characters merely provide an alternate way to denote the null set (rather than { }) and the blank character (rather than ' '.

The EBCDIC and ASCII substitutions for ASL characters must always be written without intervening blanks (for example, in .(, <=, and .V.). If the character being substituted for is a binary operator, then the EBCDIC and ASCII representations usually resemble user-defined operator names (e.g. ∈ becomes .IN.). The numeric comparison operators are an exception, however. The period-delimited notation is also used for the quantification

symbols $\forall$ and $\exists$, and the "such that" symbol $\mid$. If the character being substituted for is a unary operator, then the EBCDIC or ASCII representation is a predefined function name (e.g., $\exists S$ becomes ARB(S) or ARB S). The special constants ($\emptyset$ and $\flat$) are represented by predefined variable names (NULL and BLANK).

Variations that might be considered merely stylistic are acceptable. For example, $\&$ may be written & and $\in$ may be written ϵ.

There is no character collating sequence defined by ASL. Characters cannot enter into numeric comparison operations, and a program that requires a collating sequence must provide its own mapping between characters and numbers.

ASL LANGUAGE CHARACTER SET

Name	Graphic	Use	EBCDIC	ASCII
Digits	0, 1, . . . 9	Identifiers, numbers	0, 1, . . . , 9	0, 1, . . . , 9
Upper case letters	A, B, . . . Z	Identifiers, numbers	A, B, . . . Z	A, B, . . . Z
Lower case letters	a, b, . . . , z	Names	a, b, . . . , z	a, b, . . . , z
Break character	_	Names	_	Not avail.
Blank	(none)	Delimiter		
Plus	+	Numerical addition	+	+
Minus	-	Numerical and set difference	-	-
Asterisk	*	Multiplication, cross sections, omitted specifications, exponentiation	*	*
Slash	/	Division	/	/
Parentheses	( )	Grouping	( )	( )
Brackets	[ ]	Grouping	Not avail.	[ ]
Braces	{ }	Set former	.( ).	.( ).
Comma	,	Vectors, sets, DCL, iteration	,	,
Period	.	Numbers, user-defined operators, ellipsis, continuation symbol	.	.
Semicolon	;	Statement separator	;	;
New-line	(none)	Statement separator		
Colon	:	Statement labels, subarrays, predicates	:	:

ASL LANGUAGE CHARACTER SET (Con't.)

Name	Graphic	Use	EBCDIC	ASCII
Apostrophe	'	Character and Boolean strings	'	'
Quotation mark	"	Comments	"	"
Equal sign	=	Comparison, assignment	=	=
Not equal sign	≠	Comparison	¬=	-=
Less than symbol	<	Numerical comparison	<	<
Less than or equal	⩽	Numerical comparison	<=	<=
Greater than symbol	>	Numerical comparison	>	>
Greater than or equal	⩾	Numerical comparison	>=	>=
Negation symbol	¬	Not	¬	-
Ampersand	&	And	&	&
Crossed ampersand	⅋	Nand	¬&	-&
Wedge	∨	Or	.OR.	.OR.
Crossed wedge	⊻	Nor	.NOR.	.NOR.
Implication symbol	⟹	Material implication	==>	==>
Crossed implic. sym.	⇏	Negation of material implication	¬==>	-==>
Equivalence symbol	≡	Boolean equivalence	.EQ.	.EQ.
Exclusive or symbol	≢	Exclusive or	.XOR.	.XOR.
Stroke	|	Absolute value, such that	|	ABS .ST.
Percent sign	%	Compile time stmts.	%	%

ASL LANGUAGE CHARACTER SET (Con'd.)

Name	Graphic	Use	EBCDIC	ASCII
Cents sign	¢	Concatenation	¢	.C.
At sign	@	Origin of vectors	@	@
Exclamation point	!	Factorial	!	!
For all symbol	∀	Predicates, iteration	.V.	.V.
There exists symbol	∃	Existence test, existential predicate	?	?
Script D	𝒟	Domain	DOMAIN	DOMAIN
Script R	ℛ	Range	RANGE	RANGE
Script I	ℐ	Inverse	INVERSE	INVERSE
Number sign	#	Number of members	#	#
Phi	∅	Null set	NULL	NULL
Epsilon	∈	Membership test, range restrictors	.IN.	.IN.
Crossed epsilon	∉	Negated memb. test	.NOTIN.	.NOTIN.
Reversed epsilon	∍	Arbitrary member	ARB	ARB
Union symbol	∪	Set union	.UNION.	.UNION.
Intersection symbol	∩	Set intersection	.INTERSECT.	
Proper subset symbol	⊂	Proper subset test	.PSUBSET.	
Subset symbol	⊆	Subset test	.SUBSET.	
Proper superset sym.	⊃	Proper superset test	.PSUPERSET.	
Superset symbol	⊇	Superset test	.SUPERSET.	
Up arrow	↑	Reference	PTR	↑
Down arrow	↓	De-reference	OBJ	OBJ
Crossed b	♭	Blank	BLANK	BLANK

TYPES OF CHARACTERS USED IN ASL

Characters (120 in all)

- Alphanumerics (62 in all)
  - Digits 0 1 2 3 4 5 6 7 8 9
  - Letters
    - Upper Case A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
    - Lower Case a b c d e f g h i j k l m n o p q r s t u v w x y z
- Special Characters (58 in all)
  - Grouping Symbols ( ) [ ] { }
  - Separators , ; : . blank new-line
  - Special Constants ∅ ♭
  - Built-in Operators
    - Arithmetic + - * / !
    - Logical ¬ & ⊼ ∨ ⊽ ⇒ ⇏ ≡ ≢
    - Comparison
      - General = ≠
      - Numeric < ≦ > ≧
      - Set ⊂ ⊆ ⊃ ⊇
    - Pointer ↑ ↓
    - Set # ∈ ∉ ʒ ∪ ∩
    - Relation ℘ ℜ ℒ
    - Vector ¢ @
  - Quantification ∀ ∃
  - Miscellaneous % ' " _

2.1.2 Data Character Set

The data character set includes all the language characters plus any that an installation might care to define. The additional characters may be used in comments, character string constants, and input/output of character strings.

There is one EBCDIC graphic, $, that is not in the EBCDIC dialect of the language, but it would probably be in the data character set of an EBCDIC-oriented installation. ASCII graphics in the same category are $, ←, and \.

2.2 Tokens

This section describes how characters are put together to form tokens, which are the basic syntactic entities that are recognized by the ASL compiler's lexical scanner. A token may be informally defined as the simplest meaningful entity in a program: one whose meaning would be changed or lost if it were further subdivided.

The terminology of the various token types is sometimes confusing. There are eight basic token types distinguished in this document: identifiers, self-defining atomic values, grouping tokens, etc. Identifiers are further broken down into keywords and names. These are broken down further. The table at the end of this section summarizes the hierarchy and terminology of most of the terms used, and shows some examples. Those categories marked by an * are lexical types. That is, they can be recognized by their appearance alone. The other categories require contextural information

to be distinguished; they are recognized by the ASL compiler's parser.

Tokens are delimited in the PL/I style. That is, special characters, including the blank and new-line character, serve as delimiters. The comment also serves as a delimiter. An arbitrary number of blanks and comments may always be inserted between tokens. Blanks (or comments) are frequently necessary to achieve proper delimiting. The character string and the comment are the only entities that may contain blanks.

Examples:

log x is not equivalent to logx.

log''comment''x is (essentially) equivalent to log x.

1.0E-6 is not equivalent to 1.0E -6.

RETURN X + Y is not equivalent to RETURNX+Y.

TYPES OF TOKENS IN ASL

```
*  Tokens
*       Identifiers
              Names: x, MACK_THE_KNIFE, etc.
                    Variables
                    Named Constants
                          Statement Labels
                          Procedure Names
                          Declared Constants
                    Keywords
                          Statement Identifiers: IF, READ, etc.
                          Bracketing Keywords: BEGIN, DO, END
                          Separating Keywords: THEN, ELSE
                          Attribute Keywords: NUMERIC, STRING, etc.
                          Noise Keyword: STARTING
                          Predefined Variables: TRUE, FALSE, SQRT, etc.
*       Self-defining Tokens
*             Numbers: 1.0, 2E100, etc.
*             Character Strings: 'a', 'XY?', etc.
*             Boolean Strings: '1011'B, etc.
*       Operators
*             Built-in Operators: +, -, #, etc.
*             User-defined Operators: .MY+., .SYMDIF., etc.
*       Grouping Symbols: ( ) [ ] { }
*       Separators: colon, ellipsis, comma, stroke, equal sign, semicolon,
              new-line, continuation symbol
*       Special Constants: ∅, ♭
*       Quantification Symbols: ∀, ∃
*       Compile Time Statement Symbol: %
```

2.2.1 Identifiers

There are two types of identifiers: names and keywords. Names are made up by the programmer to refer symbolically to data and procedures. Keywords are identifiers that have a special meaning built into the language.

2.2.1.1 Names

A name, and in fact an identifier in general, is a string of alphanumeric and break characters such that the initial character is a letter.
The maximum number of characters permitted in a name is 100. The input medium may further limit this maximum because of the new-line character that the input reader inserts at the end of each line of source text (thus the maximum is 72 or possibly 80 for programs prepared on standard 80-column punched cards).

The reason for the limit of 100 is that the compiler may be more efficient if it is known that an identifier's length can be packed into a small number of bits (such as one byte). Although ASL usually does not yield so easily to minor compiler problems, in this case there seems to be very little value in allowing extremely long identifiers. The input medium will usually be the limitation, and even if it were not, it's hard to imagine a situation in which an identifier exceeding 100 characters in length improves readability.

Names are used for variables, statement labels, procedure names, and declared constants.

Examples:

Valid names	Invalid names
x	_x
X1__	2x
x_prime	#table
Set	x-1

In this document, the term "name" is not used in the sense synonymous with "address", as is frequently done.

2.2.1.2 Keywords

A keyword is an identifier that has a special meaning built into the language. Some keywords are reserved words, as indicated below. There are six types of keywords in ASL. Following is a partial list.

Statement and statement header identifying keywords (reserved): DECLARE, DCL, READ, WRITE, PAGE, FORMAT, RETURN, QUIT, CONTINUE, STOP, EXECUTE, EXEC, ENTRY, GO, TO, and IF.

Bracketing keywords (reserved): DO, BEGIN, CASE, FUNCTION, OPERATOR, and END.

Separating keywords (reserved): THEN, ELSE, WHILE, DOING.

Attribute keywords (not reserved): STATIC, DYNAMIC, INTERNAL, SHARED, IN, NUMERIC, NUM, CHARACTER, CHAR, BOOLEAN, BOOL, PROCEDURE, PROC, INTEGER, INT, POINTER, PTR, SET, RELATION, MAP, ARRAY, MATRIX, VECTOR, STRING, PAIR, DENSE, SPARSE, REGULAR, DIMENSION, DIM, SIZE, LI, HI, LENGTH, EXACT,

APPROXIMATE, APPROX, ASSIGNS, PRECEDENCE, PREC, and VERIFIED.

Noise keyword (reserved): STARTING.

Predefined variables (not reserved): TRUE, FALSE, SQRT, etc.

All keywords are written in upper case. This makes them stand out, which helps to reveal the structure of programs, and makes key-punched ASL closer to real ASL. The break character is not used in keywords.

2.2.2 Self-defining Tokens

A self-defining token is one that denotes its own value, i.e., its representation can be regarded as both its name and its value. It is sometimes called a "constant", although the term "constant" is used loosely and sometimes denotes other entities.

ASL includes numeric, character, and Boolean self-defining tokens.

2.2.2.1 Numeric Self-defining Tokens (Numbers)

Numbers are written as either (1) a string of decimal digits, (2) a period (decimal point) surrounded on both sides by a string of digits of nonzero length, or (3) either of the above followed by an exponent part. An exponent part is the letter E, optionally followed by a sign (+ or -), followed by a digit string. There may be no embedded blanks. A prefix sign is considered to be an operator.

Examples:

Valid Numbers	Invalid Numbers
0	2E 100
123	6.E-3
2E100	1.
6E-3	.5
1.0	3.0e-5
0.5	2 E100
3.0E-5	2E3.4

A self-defining numeric token has the attribute EXACT if it is written without a decimal point, and APPROXIMATE if it has one. An exact number is stored as a ratio of two arbitrarily large integers, and an approximate number is stored in some implementation-defined format and precision (such as decimal or binary floating point). Thus 6E-3 is stored as 6/1000, or equivalently as 3/500.

ASL does not include a facility for expressing numeric data in a base other than decimal.

2.2.2.2 Character Self-defining Tokens

Character self-defining tokens are written with a single character surrounded by apostrophes, or "single quotes", e.g., 'A'. The special symbol ⁄b is an abbreviation for ' '.

More than one character may be written between apostrophes, in

which case it is taken to represent a string of characters, e.g., 'abc' is an abbreviated way to write ('a', 'b', 'c') (this anomalistic treatment follows Algol 68). Also, two adjacent apostrophes, '', is taken to denote the null string; it is a way to write the null set ∅ (this will be made clear in section 3, Data Elements).

The apostrophe may be indicated within a character string by doubling up on it. Thus 'isn''t' is a representation of a string of length five, and the single apostrophe is written ''''.

The new-line character may not appear in a character string self-defining token. It may only be put into a string by referencing the predefined variable NL, and using some operation such as concatenation.

The maximum length character string is generally limited by the medium on which the source program is prepared.

A character string of length one may be written, for example, as ('a',), or ONEVECTOR 'a'.

2.2.2.3 Boolean Self-defining Tokens

There are two Boolean self-defining tokens, which are written '1'B and '0'B. Alternatively, one can use the predefined variables TRUE and FALSE, which are initialized to these values, respectively.

As in the case of character data, a sequence of ones and zeros may be written between the apostrophes, in which case it is taken to denote a Boolean string. For example, '101'B is an abbreviated way to write ('1'B, '0'B, '1'B).

Also, ''B is allowed; it is another way to write the null set ∅.

A Boolean string of length one may be written, for example, as ('1'B,), or ONEVECTOR '1'B.

Boolean strings are sometimes called bit strings, but in this document the term Boolean string is used to stress the fact that they are vectors of truth values, and not numbers ("bit" is a contraction of "binary digit"). Nevertheless, 0's and 1's are used for the self-defining tokens (rather than 'F'B, for example), because the algebra of truth values is so well expressed in the former notation.

A Boolean string may not contain a new-line character, and thus the maximum length is generally limited by the source program's medium.

2.2.3 Operators

An operator, like a function, transforms one or more operands into a single result. ASL includes a large number of built-in operators and allows the programmer to define his own.

2.2.3.1 Built-in Operators

The ASL built-in operators are discussed in detail in section 4, Expressions. Here we merely list them and discuss their syntactic qualities.

There are operators of both binary and unary types, and in the unary class there are prefix, suffix, and one "parenthesis-type" (absolute value).

The table at the end of this section shows all operators and their precedences. These precedences may be obtained by means of the PRECE-

DENCE predefined function, e.g., PRECEDENCE('#') is 19. A few symbols which are not really operators are also shown in the table. These symbols might be assigned precedences (in the order indicated by the table) to aid in parsing, but the precedence on non-operators is not given, as it is not available by means of the PRECEDENCE predefined function.

The precedence of the prefix + and prefix - is obtained by referencing PRECEDENCE('P+') (or 'P-'). The binary forms are obtained by using '+', '-', 'B+', or 'B-'. The absolute value function has no precedence; its meaning in an expression is always unambiguous.

The placement of "juxtaposition" in the table is intended to indicate that an expression such as $\alpha' F\ x$ is parsed as $(\alpha' F)(x)$.

Binary operators with equal precedences are grouped on the left, with the exception of exponentiation. Thus a - b + c is (a - b) + c, and a**b**c is a**(b**c). (Exponentiation is an exception because a**b**c is assumed to represent the mathematical a^{b^c}, and because (a**b)**c would usually be coded as a**(b*c)).

Prefix operators and juxtaposed names are grouped on the right. Thus ↑-log sin x is taken to mean ↑(-(log(sin x))).

ASL BUILT-IN OPERATORS

Name	Graphic	Precedence	Type
Range, Domain Inverse	$\mathcal{R}, \mathcal{D}, \mathcal{l}$	21	Prefix
Juxtaposition		20	
Number	#	19	Prefix
Arbitrary element	ɜ	19	Prefix
Dereference	↓	19	Prefix
Exponentiation	**	18	Binary (right)
Factorial	!	17	Suffix
Existence test	$\exists$	17	Prefix
Prefix +, -	+, -	16	Prefix
Multiplication, Division	*, /	15	Binary
Binary +, -	+, -	14	Binary
At	@	13	Binary
Concatenation	¢	12	Binary
Intersection	$\cap$	11	Binary
Union	$\cup$	10	Binary
Subset test	$\subset, \subseteq$	9	Binary
Superset test	$\supset, \supseteq$	9	Binary
Membership test	$\epsilon, \notin$	8	Binary
Equality test	=, $\neq$	7	Binary
Numerical comparison	$<, \leqslant, >, \geqslant$	7	Binary
Not	¬	6	Prefix
And, Nand	$\&, \mathcal{\&}$	5	Binary
Or, Nor	$\vee, \not\vee$	4	Binary
Equivalence, Exclusive or	$\equiv, \not\equiv$	3	Binary
Implies	$\Rightarrow, \not\Rightarrow$	2	Binary
Reference	↑	1	Prefix
Conditionals	IF, THEN, ELSE		Prefix
Range	:		Binary
Ellipsis	...		Trinary
Comma	,		Binary
Such that	|		Binary
Assignment	=		Binary
Label	:		Binary
Statement terminator	;		Suffix
Absolute value	| |		Parenthesis

2.2.3.2 User-defined Operators

ASL includes a limited capability for user-defined operators. User-defined operators are always clearly recognizable as such, from their syntax alone, and no facility is provided for changing the meaning of a built-in operator. Thus when one sees a "+" in a program, its meaning is known immediately, and one need not search for a possible redefinition of it.

A user-defined operator is a character string that begins and ends with a period, and between the periods there can be no blank, period, or new-line character.

Examples:

Valid operators	Invalid operators
.+.	..
.PARENT.	. .
.MY_*.	...
.cross.	.DOT PRODUCT.

User-defined operators can only be binary operators. There is no facility for user-defined prefix, suffix, parenthesis, or any other type of operator. There is little need for prefix type, as ASL functions of single variables can be written without parentheses.

A precedence may be assigned to a user-defined operator by means of the PRECEDENCE attribute in a DECLARE statement. These precedences are numeric values that are compared to the precedences of the built-in operators when parsing an ASL program. The precedence of a user-defined operator need not be an integer and it need not be positive.

2.2.4 Grouping Symbols

There are three types of grouping symbols:

parentheses ( )

brackets [ ]

braces { }

Parentheses and brackets are used in expressions and in enclosing lists. Braces are used in forming sets, either by listing the members (e.g. {1, 2, 3}) or by forming a set from certain members of another set (e.g. {SQRT(x), x ε S | x ⩾ 0}).

Parentheses and brackets may be used interchangeably. However, a left parenthesis must be matched by a right parenthesis, and not by a right bracket, and vice versa. Brackets are generally used to improve readability in an expression, for example (f(x))(y) may be expressed a little more clearly as [f(x)](y). Because of their interchangeability, either brackets or parentheses may be used to denote subscripting, e.g., A(i) and A[i] are equivalent. Since nearly any syntactic entity in ASL may be enclosed in parentheses or brackets without changing the meaning, the fairly common convention of enclosing labels in brackets may be used, e.g. [LOOP]: IF i ⩽ n THEN....

Following is a complete list of the uses of parentheses and brackets:

1. Grouping of expressions, e.g. a*(b+c).
2. Vector former, e.g. y = (1, 2, 3); f(x, y).
3. Grouping of statements, e.g. (x=y; y=y+1).
4. Factoring of attributes, e.g. DECLARE (x, y NUMERIC) STATIC.

Redundant parentheses and brackets may always be inserted. For example, if x is a name, then [(x)], [x], and x are always equivalent. As a specific example, log x and log(x) are equivalent. If x is a list, however, then the innermost parentheses cannot be removed without altering the meaning. For example, $\{1, 2, 3\}$ is a set of three integers, whereas $\{(1, 2, 3)\}$ is a set containing one object: a vector. Also, $\forall i = 6, 7, 8$ specifies an iteration with i = 6, then i = 7, and then i = 8. But $\forall i = (6, 7, 8)$ specifies an iteration to be done once, with i a vector.

2.2.5 Separators and Miscellaneous Tokens

The "separator" tokens are the colon, ellepsis, comma, stroke, equal sign (assignment symbol), semicolon, new-line character, and continuation symbol.

The colon is used to separate two numeric expressions in a range specification (e.g., STRING(3:5)), in the quantifier predicates, and to separate a statement label from the statement. The ellipsis (three consecutive periods) is used to separate the first and second expressions from the last expression in the elided list notation, e.g., $\forall i = 1, 2, \ldots, n$; $\{1, 3, \ldots, 15\}$, etc. The comma is used to separate items in a list. The stroke is used in its separator role in iteration specifications, e.g., $\forall x \in S \mid x \neq 0$. The stroke is also used in the notation for absolute value.

The equal sign is used as a separator in the assignment statement (it is also used as a relational operator). The semicolon is used to separate statements, as is the new-line character. The continuation

symbol (four consecutive periods) is used to cancel the effect of the next new-line character (see section 2.5, Statement Format).

The remaining tokens in ASL are the special constants ∅ (null set) and ♭ (blank character), the quantification symbols ∀ (universal) and ∃ (existential), and the compile time symbol % (percent sign).

As was mentioned in section 2.1.1, the character ∅ is merely an abbreviation for { }, and ♭ is merely an abbreviation for ' ' (one space between single quotes).

The ∀ symbol is used in iteration "for-specifications" (e.g., ∀x ∈ S, 1 ⩽ ∀i ⩽ n, etc.), and in a predicate expression such as IF ∀x : C(x) THEN.... The ∃ symbol is used only in predicates, as IF ∃x : C(x) THEN....

The compile time symbol is used to denote those statements that are to be executed at compile time, as in %IF DEBUG = 'ON' THEN WRITE x, y, z. (The use of % follows the use of % in PL/I; in the preceding example, the scope of the % ends with the word THEN).

2.3 Comments

Comments are written between quotation marks (or "double quotes"). This notation was chosen because it is concise and because quotation marks are frequently used when expressing one type of language in another. ASL uses the quotation mark to escape to English, and the apostrophe to escape to a lower level language (a character string to be taken literally); hence the apostrophe is frequently called a "single quote mark". It is

convenient to have a comment delimited by a single character (rather than the PL/I /* ... */ or the Algol comment ... ;), so that the comment may be written and read as easily as possible.

A comment may not contain a (double) quotation mark, as this would be taken as closing the comment. It may contain semicolons and new-line characters.

The fact that the comment opener and closer are the same symbol introduces the problem that inadvertently omitting one will cause an unsophisticated compiler to interchange what it attempts to compile and what it skips over. However, this is not a serious problem for a language that is primarily meant to be read. It is preferable to have comment delimiters concise and unobtrusive.

A comment may be inserted in a program wherever a blank can, with the exception of a character string self-defining token. Here the "comment", with its quotation marks, would be taken as characters in the string.

Examples of comments are given in section 2.8, Sample Procedures.

2.4 Statements

Section 5 describes in detail how each type of statement is formed from tokens. Here we describe certain properties common to all statement types, or to a class of statement types.

2.4.1 Simple Statements

There are fourteen types of simple statements in ASL: DECLARE,

assignment, READ, WRITE, PAGE, FORMAT, RETURN, QUIT, CONTINUE, STOP, EXECUTE, ENTRY, GO TO, and null. Most (twelve) of these start with an identifying reserved word (DECLARE, READ, etc.). The assignment statement does not have a reserved word associated with it; it is characterized by having one or more assignment symbols (a rigorous definition would be somewhat involved and is not given here). The null statement is simply nothing at all except possibly blanks and comments, written between statement separator characters.

The statement separator characters are the semicolon and the new-line character.

Examples:

```
DECLARE (x, y) NUMERIC
READ x, y; WRITE z;
z = x + y  "Add x and y."
```

There are five statements in the above. First is a declaration, which is terminated by a new-line character. Next is a READ, which is terminated by a semicolon. This is followed by a WRITE, which is also terminated by a semicolon. There is a null statement between the semicolon and the next new-line character. This is followed by an assignment statement, which has a comment embedded in it. The statement terminator is considered to be a part of the statement.

Null statements have no effect on an ASL program. They are not even counted when the compiler generates "statement numbers".

2.4.2 Statement Groups

Statements may be grouped for control purposes by enclosing them in parentheses, brackets, the words DO and END, or the words BEGIN and END. The word END may be followed by other tokens as explained in the next section. Statement groups must be properly nested.

Examples:

```
(x = y; READ y)

DO IF S ≠ ∅ THEN x = 3S

    t = x(3:5) END
```

In any context where a single statement normally appears, a statement group may instead be written. Enclosing a single statement in parentheses has no effect.

The ASL statement group is similar to the PL/I simple DO group. Unlike PL/I, however, the ASL group is not used to specify iteration. ASL also does not include anything similar to the PL/I BEGIN ... END, with its name-scoping effects.

The grouping keywords (DO, BEGIN, and END) do not require statement separator symbols, although supplying them simply introduces a null statement. For example, the following two lines of code are equivalent (null statements are not counted in the assignment of statement numbers to labels).

```
IF a ≠ b THEN DO x = 0; y = 0 END

IF a ≠ b THEN DO; x = 0; y = 0; END;
```

Similarly, the placement of statement separators with respect to parentheses and brackets is immaterial. For example, the following are equivalent:

```
(x = 0;  y = 0;)

(x = 0;  y = 0);
```

2.4.3 Compound Statements

A compound statement is a statement or statement group preceded by a statement header. There are six types of statement headers: IF, CASE, iteration header, INITIALLY, FUNCTION, and OPERATOR. The IF and CASE are collectively referred to as the conditional headers, and the FUNCTION and OPERATOR are collectively referred to as the procedure headers.

The IF, iteration header, and INITIALLY header apply to a single statement or statement group. This applies to both the THEN and the ELSE clauses of the IF. (The "decision table IF", however, is a more complicated construction; see section 6.3).

Some examples:

```
IF x <0 THEN x = 0;  ELSE x = 1

IF c ≠ '.' THEN (c = '.';  ERRORSW = TRUE)

(∀x ∈ S) y = y + x

(∀x ∈ S)(sum = sum + x;  sumsq = sumsq + x**2)

INITIALLY x = 0

INITIALLY DO x = 0;  S = ∅  END
```

If the statement group is delimited by DO ... END, then following the word END may be written a series of tokens obtained from the header as follows:

IF: The word IF or ELSE, for the THEN or ELSE clause, respectively, followed by the first $n \geqslant 0$ tokens obtained from the conditional expression in the header.

Iteration header: A series of the first $n \geqslant 0$ tokens obtained from the header.

INITIALLY: The word INITIALLY.

Examples:

```
IF x⩾ 0 THEN DO
    y = SQRT(x)
    z = 0
    END IF x⩾ 0
ELSE DO
    y = SQRT(-x)
    z = 1
    END ELSE x⩾ 0
∀x ε S DO
    sum = sum + x
    sumsq = sumsq + x**2
    END ∀x
INITIALLY DO x = 0; S = ∅ END INITIALLY
```

The CASE, FUNCTION, and OPERATOR headers are "openers" in themselves (act like a left parenthesis) and must be closed with an END. The END may be followed by a series of tokens from the header, starting optionally with the word CASE, FUNCTION, or OPERATOR. In the case of FUNCTION and OPERATOR, either all formal parameters must be displayed or none at all.

Some examples:

```
CASE i
    1: x = a + b
    2: x = 2*a + b
    3: x = a + 2*b
    END CASE i

FUNCTION REVERSE(X)
1 ≤∀i ≤#X/2 DO j = #X + 1 - i; [X(i), X(j)] = [X(j), X(i)] END
RETURN X
END FUNCTION REVERSE(X)
```

2.4.4 Statement Labels

Most statements can be labelled. A statement label is an expression that precedes the statement it labels and is separated from it (or from other labels) by a colon. Like expressions in general, labels may be enclosed in parentheses or brackets.

Statements in the range of a CASE must be labelled and the label

must be a value producing expression. In other cases, a label is optional and it is a value receiving expression.

Some examples:

```
L: x = y

L(3): LABEL(n,'a'): EXECUTE LBL

CASE c(i)
   '(': '[': n = n + 1
   ')': ']': n = n - 1
   '|':      (n = n + stroke_switch; stroke_switch = -stroke_switch)
             END CASE c(i)
```

Labels are used for the CASE header and as a target of the EXECUTE and GO TO statements. A label is mandatory on all statements in a CASE group, but cannot appear on a FUNCTION or OPERATOR header. Except for these restrictions, labels are optional. They may, for example, be used as follows:

```
IF x <0 THEN L: x = -x

(WHILE x(i) > 0) L: i = i + 1

D: DECLARE X STRING(CHARACTER) LENGTH ⩽10
```

Labels such as the L's in the above examples cannot be referenced by a GO TO from outside the THEN or WHILE group, but they can be referenced by an EXECUTE (provided they label the whole group) and by debugging and elaboration features of ASL. Labels such as these are also sometimes useful in connection with documenting a program.

When a non-executable statement (DECLARE or FORMAT) is referenced

by an EXECUTE or GO TO, it acts as a null statement (no-operation).

A label applies to an entire statement group, in general, and thus has a range associated with it. When a label is referenced in an EXECUTE statement, the entire labeled group is executed. For example, in:

```
L1: x = 0; y = 0

L2: (x = 0; y = 0)

(L3: x = 0; y = 0)
```

the statement EXECUTE L1 sets x = 0 (only). EXECUTE L2 sets x and y to zero. EXECUTE L3 is invalid (from outside the group containing L3).

2.5 Statement Format

The statement format of ASL is a cross between the FORTRAN record-oriented approach and the PL/I and Algol stream-oriented approach. It is recognized that the stream approach offers greater flexibility and is easier to deal with in a precompiler, but in view of the fact that programs are nearly always seen on some medium that has lines, such as the printed page, it seems rather extreme to completely ignor the natural boundaries that the lines provide.

Statements are delimited by either of two characters: the semicolon or the new-line (NL) character. These characters are not equivalent, however. The NL character may be effectively canceled by preceding it with a statement continuation symbol, which is four consecutive periods. Between the continuation symbol and the NL character, only blanks and comments may appear (these may be inserted between any two tokens).

The continuation symbol is not effective in a comment or character string self-defining value. The continuation symbol may not split any token, such as a character string, or a variable name, number, etc.

The NL character may split a comment, but it may not split any other token, including a character string and Boolean string.

2.6 Procedures

2.6.1 Defining Procedures

ASL statements and statement groups are put together to form procedures, of which there are two varieties: functions and operators. Procedures begin with a heading statement and end with an END statement. The heading begins with one of the keywords FUNCTION or OPERATOR, followed by the variable names to be used for the formal parameters. The END statement may optionally include the procedure name with or without its formal parameter display. Following are two complete procedures.

```
FUNCTION SYMDIF(R, S)
RETURN R-S ∪ S-R
END SYMDIF(R, S)

OPERATOR R .-. S
RETURN R-S ∪ S-R
END R .-. S
```

An ASL procedure is in many ways a self-contained unit. It is invalid to refer to a statement in a procedure by a GO TO or EXECUTE from outside the procedure. A procedure can only be entered at its heading and can only exit by a RETURN or STOP (it may, however, call other procedures).

Every variable is considered to be "owned" by a particular procedure. A variable is normally known only within its owning procedure, but this can be modified by the SHARED and external attributes. Unlike PL/I and Algol, variables are not automatically known to contained procedures.

Procedures may be separately compiled, provided they are outermost (see section 2.7).

2.6.2 Invoking Procedures

Function-procedures are invoked, or "called", by writing their name followed by a vector containing the procedure's arguments, e.g. F(x, y, z). If a procedure has only one argument, the parentheses need not be written, for example, if x is a variable name, then log(x) and log x are equivalent. This follows from the fact that x and (x) are equivalent. Occasionally one defines procedures that take no arguments, for example the system routine TIME (which returns the current time in seconds since midnight GMT January 1, 1900). This may be invoked as TIME( ) (undefined argument).

Juxtaposition of names and self-defining tokens denotes functional application. In an expression such as TIME + x, TIME is a variable (or a declared constant) whose current value must be numeric.

User-defined operators are invoked by writing the operator name, including its periods, between the two operands. For example:

x .OP. y

User-defined operators are always binary, and both operands must be present.

A user-defined operator may not appear in any context other than as an operator and in a DECLARE statement. For example, the expressions y = .OP., {.OP.}, and .OP. ε S are invalid. On the other hand, if P is a function name, then the expressions y = P, {P}, and P ε S are valid; their meaning is discussed in section 3.1.4.

Procedures of both types may be invoked recursively; that is, a procedure may call itself either directly or as a result of calling another

procedure. However, such concepts as multitasking and reentrancy, and parallel processing in general, are not included in ASL.

Procedures of both types may also be invoked in a "sinister call", that is, they may be invoked in a value receiving context such as the left side of an assignment statement. With suitable definitions of F and .OP., expressions such as:

```
x .OP. y = z;
```

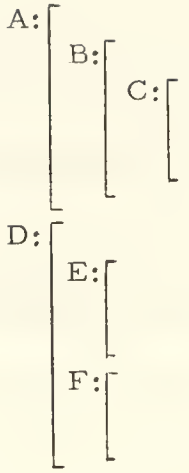
and $\forall F(x) = 1, 2, \ldots, n$

may be valid. The meaning of sinister calls is discussed in section 7.

2.7 Programs

A program is a set of procedures that is a complete executable unit. A program includes steps to read input data (if any), perform calculations, and print or otherwise make available the results.

ASL procedures can be nested, that is, a procedure can be contained in another. If a procedure B is contained in a procedure A, then B is said to be "owned" by A. The purpose of nesting procedures is simply to limit the scope of the procedure's name (nesting procedures may also enhance the compiler's optimization possibilities). A procedure is known to its immediate parent (its owner), and to all descendents of the parent. The outermost procedures are assumed to be contained in a single routine that is supplied by the ASL compiler; thus these outermost procedure names are known everywhere. As an example, consider a set of procedures structured as follows:



Here A and D are known everywhere. B is known to A, B, and C. C is known to B and C. E and F are known to themselves, each other, and to D.

A procedure may call any procedure whose name is known to it. That is, it may call its immediate descendents and the immediate descendents of any of its ancestors. In the above situation, A may call A, B, and D. B may call A, B, C, and D. C may also call A, B, C, and D. D may call A, D, E, and F. E and F may call A, D, themselves, and each other. Note the lack of symmetry: B may call D but D may not (normally) call B. Either E or F could have the same name as B (this is part of the purpose of nesting procedures), and any call can be resolved without ambiguity.

The above situation is the normal state of affairs; it can be modified by means of the external and shared attributes. For example, D could call B if (1) D declared B to be the B that is known within A (e.g. DECLARE B IN A), and (2) A declared B to be shared with D (e.g. DECLARE B SHARED(D)). See sections 5.1 and 8 for further details.

Predefined procedures, such as SQRT, are outermost and hence are known everywhere.

An internal procedure cannot be placed directly in the line of program flow. This is taken to be equivalent to an attempt to enter a procedure by means of a GO TO, which is invalid.

2.8 Sample Procedures

```
FUNCTION TOPSORT(P, S)

"This function returns a regular vector containing the members of
set S sorted according to the partial ordering P.  P is a set of pairs
of members of S.  The procedure repeatedly selects a member of S
that has no predecessor, appends it to a vector V, and deletes the
member from S.  Pairs beginning with the member are also deleted
from P.  This continues until S is null. "
V = ∅
WHILE S ≠ ∅ DO
   y = x ε S : ¬(∃p ε P : p(2) = x)
   V(#V+1) = y
   S = S - {y}
   P = P - {p ε P | p(1) = y}
   END WHILE
RETURN V
END TOPSORT
```

Since parameters are passed by value with no return of parameters, TOPSORT does not modify the caller's arguments. The statement V(#V+1) = y could be coded V = V ¢ (y,), or V = V ¢ ONEVECTOR(y), if desired.

```
FUNCTION PERMUTE(SEQ)

" This function maps a given sequence of numbers into the
next permutation of the sequence, in lexicographic order. For
each permutation, the last possible element is increased by the
least possible amount. When used with a statement such as
'seq = PERMUTE(seq)', successive permutations are generated
on each call. For example, if the first value of SEQ is (1, 2, 3, 4),
then successive values are (1, 2, 4, 3), (1, 3, 2, 4), (1, 3, 4, 2),
(1, 4, 2, 3), . . . , (4, 3, 2, 1), (1, 2, 3, 4), etc. SEQ must be a string
(regular vector) of numbers.
     The globally shared variable FIRSTPERM is set to TRUE
when PERMUTE returns with the first sequence (i.e., a sequence
in monotonic ascending order)."

DECLARE FIRSTPERM SHARED

FIRSTPERM = TRUE          "Initialize."
n = HI(SEQ)               "Highest index of input sequence."
n > ∀j ≥ 1 DO
   "Find last (rightmost) point of increase."
   IF SEQ(j) ≥ SEQ(j+1) THEN CONTINUE ∀j
   " Found a point of increase, SEQ(j) < SEQ(j+1). Find last
   (rightmost) SEQ(k) that exceeds SEQ(j), and swap."
   n ≥ ∀k > j IF SEQ(k) > SEQ(j) THEN QUIT ∀k
   [SEQ(j), SEQ(k)] = [SEQ(k), SEQ(j)]
   FIRSTPERM = FALSE
   QUIT ∀j
   END ∀j

IF FIRSTPERM THEN "No point of increase was found (SEQ is
   in reverse order). Force rearrangement below to work for
   this special case."
   j = 0

" Now rearrange elements after SEQ(j) into ascending order
(they are now in monotonic descending order)."
j = j + 1
WHILE j < n DO
   [SEQ(j), SEQ(n)] = [SEQ(n), SEQ(j)]
   j = j + 1; n = n - 1 END

RETURN SEQ
END PERMUTE(SEQ)
```

3. Data Elements

ASL includes five atomic data types and one data aggregate, the set. The atomic data types (those that cannot be subdivided) are:

Numeric (real numbers)

Character (a single character)

Boolean (a single truth-value)

Pointer (or "reference")

Procedure

All data structures consist of sets, which contain atoms, sets of atoms, etc. Although structures of arbitrary complexity can be built up, several simple structures occur so frequently in practice that they are given names that are part of the ASL language, and several built-in operators exist that can only be applied to these special structures. For example, atoms can be formed into vectors and arrays, and vectors can be combined by the concatenation operator ¢ .

3.1 Atomic Data

3.1.1 Numeric Data

ASL includes only one kind of numeric data, and it represents an attempt to implement a reasonable approximation to the mathematical idea of a real number. There is no facility for complex numbers.

The usual distinctions between types of numeric data, such as floating point versus fixed point, decimal versus binary, etc., are not used.

Numbers are stored internally in two forms: exact and approximate. An exact number is stored as a ratio of two arbitrarily large integers. An approximate number is handled as in APL\360: numbers are stored in some approximate form convenient for the machine, such as binary floating point. For comparison tests and when an integer is required by context, a "tolerance" is applied. For machines such as the IBM System/360 and the CDC 6600, the APL tolerance value of about 10^{-13} is reasonable. On the System/360, the long form floating point format (64 bits total) is suggested.

The mode of a number (exact or approximate) is determined by how it is written, if it is a self-defining value, and by the operations and operands that created it, if it is the current value of a variable. A self-defining numeric value is exact if written without a decimal point, and approximate if it has one. Thus 1 is exact, and 1.0 is approximate.

The operations of addition, subtraction, multiplication, and division produce an exact result if both operands are exact, and an approximate result if either operand is approximate. Exponentiation produces an exact result only if both operands are exact and the exponent is an integer. Thus (2/3)**-3 is 27/8; 1**(1/3) is 1.0. Factorial always produces an exact result or an undefined result. Its argument must be a non-negative integer. If the argument is approximate, it is converted to an exact integer (if possible) by applying the tolerance, as is explained below. The absolute value function and unary minus produce results in the same mode as their argument. The # function (number of members

of a set) always produces an exact result.

The transcendental and irrational predefined functions, such as LOG, SIN, SQRT, etc., always produce an approximate result. Functions that always have integer values, such as FLOOR and CEILING, produce exact results. Functions that select numbers, such as MAX and MIN, have their result in the same mode as the number selected.

Numbers may be converted from exact to approximate by using the predefined function APPROXIMATE (abbreviated APPROX). There is no predefined function for converting in the other direction, except for FLOOR and CEILING. For example, x = FLOOR(100*x + 0.5)/100 converts x from approximate to exact, with a resolution of 1/100.

Approximate numbers are not expected to exceed some large but implementation dependent maximum (on the System/360 it would be about 7.2×10^{75}). If this limit is exceeded, an "overflow" condition occurs, and the result is undefined (but execution continues). The limit could be exceeded by:

1. An arithmetic operation or predefined function reference.
2. Conversion from exact to approximate mode.
3. Conversion from external to internal form (e.g., attempting to read in the number "3.4E100" in a System/360 implementation; having such a number in the program text is invalid).

Exactly which arithmetic operations can cause overflow is implementation dependent. For example, the absolute value operation might in some implementations cause overflow, but never in others. Division by

zero, including the 0/0 case, is considered to be an overflow condition, and the result is undefined.

If an approximate mode result is extremely small or zero (underflow or significance exceptions on the System/360), then no exceptional condition occurs: the result is made zero.

3.1.1.1 Numeric Tolerance

The tolerance is used when an approximate number appears in one of the following contexts:

1. Equality test.
2. Explicit conversion to an integer (FLOOR and CEILING).
3. Implied conversion to an integer (n!).
4. The limits in a counting iteration (the m and n in $m \leqslant \forall i \leqslant n$). These limits need not be integers, but if they are close to an integer, they are treated as such.
5. As a subscript (e.g., A(i)).

The numerical equality test works as follows. If both numbers are exact, the result is TRUE if (and only if) both numbers are precisely the same. If one number is exact and the other is approximate, the exact number is converted to approximate mode, and they are considered to be equal if $|x - y| \leqslant T$, where T is the tolerance. If both numbers are approximate, they are considered to be equal if $|x - y| \leqslant 2T$. Neither number is actually altered by the equality test. Thus two numbers may compare equal, but their difference may be nonzero.

FLOOR(x) is defined to be the greatest integer less than or equal to x + T, if x is approximate. Similarly, CEILING(x) is the least integer greater than or equal to x - T.

For n! and $m \leqslant \forall i \leqslant n$, m and n are first tested to see if there is an integer k such that $m-T \leqslant k \leqslant m+T$, and similarly for n. If there is, the integer is used for the factorial or for the limits of the counting iteration. An iteration such as $m < \forall i < n$ is treated similarly. Thus if m = 0.999..., then i starts out at 2 (not 1).

The handling of subscripts is to some extent a special case of the equality test. To evaluate A(i), the set A is searched for a pair beginning with i, so the equality test is used there. This is the formal definition of the meaning of A(i). The way it is actually implemented might depart from this slightly. The subscript i might be converted to an integer k such that $i-T \leqslant k \leqslant i+T$, if possible, and then an indexing operation might be done, using k.

A set may at times be tested to see if it is a vector. To be a vector, the set must be a map and its domain must consist of integers. The tolerance is used in determining if its domain consists of integers. If the set is a vector, then its domain may be altered so that it consists of integers in exact mode. This is done so that the vector can be stored the way vectors normally are stored. However, it introduces a side effect to set building. Suppose V is null and i = (1.0/3)*6 = 1.999.... Then after V(i) = object (any ASL object), (i,object) = ⍹V, i may be 2 and may be exact, or it may be 1.999.... Which value is obtained may depend on the implementation, the type of optimization selected, and the declaration, if any, of V. These remarks apply to arrays in general.

This treatment of numbers is quite informal, but it allows the programmer to deal with them as easily as we do in everyday life. The success of APL attests to its utility.

One could make a case for having the tolerance be multiplicative rather than additive. That is, perhaps two numbers a and b should be considered equal if $1-\epsilon \leqslant |a/b| \leqslant 1+\epsilon$, where ϵ is small (compared to 1). This more closely expresses the fact that approximate numbers are represented to a fixed total number of digits, rather than a fixed number of digits after the radix point, and it would work better for very large or very small numbers, in some cases.

However, the usual case of concern is when one is dealing with

numbers that are not terribly large, and after subtraction of two such numbers, the result should be zero but it turns out to be some small number such as 10^{-15}. In such a case the multiplicative tolerance will not treat the 10^{-15} as equivalent to zero, but the additive tolerance will.

Perhaps best for ASL would be something more sophisticated, such as interval arithmetic (maintaining upper and lower bounds on all numbers). But it seems reasonable at this point to employ the tolerance in a way that is simple and of proven utility.

3.1.2 Character Data

Following APL and Algol 68, the atomic character datum is the single character. A character self-defining value is written between apostrophes, e.g., 'a'. Zero or two or more characters between apostrophes denotes a character string, as discussed in section 2.2.2.2. An item of character data must be in the data character set (see section 2.1.2).

Character data may not enter into numeric operations, including the inequality tests $<$, $\leqslant$, $>$, and $\geqslant$. The only operations that may be applied to character data are those operations that can be applied to any objects: assignment, the equality test, referencing ($\uparrow$), and the set operations.

There is no character collating sequence built into ASL.

3.1.3 Boolean Data

The atomic Boolean datum is the single truth value. They may be obtained by referencing the predefined variables TRUE and FALSE. Boolean string self-defining data may be written, for example, as '101'B (see section 2.2.2.3), similarly to character items and strings.

Boolean data may not enter into numeric operations, including the numeric comparison operations $<$, $\leqslant$, $>$, and $\geqslant$. Atomic Boolean data and Boolean vectors may be operated on with the not ($\neg$), and ($\&$), or ($\vee$), and other Boolean operators. These are explained in section 4.1.5.

The atomic Boolean datum is used as the result of a conditional expression, as occurs in an IF statement, the set former, etc.

3.1.4 Pointer Data

A pointer datum is an indirect reference to another object. It is generated by applying the up arrow prefix operator to a value, e.g. $\uparrow x$ or $\uparrow 2$. Pointers may be assigned to variables, put into sets, etc., but besides these operations (which apply to all objects), the only operator that may be applied to a pointer datum is the down arrow, or dereference

operator. The value of ↓p is the object that p points to.

Pointer data may enter into the equality test, and pointers are considered to be equal only if they reference the exact same copy of an object.

3.1.5 Procedure Data

A procedure of the FUNCTION type can be treated as data and assigned to variables. If F is a function name, then after y = F, F can be invoked by the expression y(x). The name F is considered to be a constant, and it cannot appear in a value receiving context.

Assignment of procedure names is different from other assignments in that a new copy of the value (the procedure) is not created. After y = F, if F should somehow be altered, then the alteration is reflected in references to F via y. In this way, procedure variables are similar to pointer variables.

Procedure data cannot enter into any operations other than those that apply to all data types.

3.2 Set Data

A set is an unordered collection of objects, which in ASL are always finite in number. The same value may not appear twice in a set. Two sets are considered equal if they have the same number of members and each member of one is a member of the other.

If it is attempted to add an object to a set, and the object is already in the set, then the new object is not added. The equality test is used to determine whether or not the object is already in the set, and hence the numeric tolerance is involved. This subject is discussed in section 3.2.3.

Some rudimentary set operations will be described here. A more complete discussion of set operations will be found in section 4.

If a, b, and c are expressions, then {a, b, c} denotes the set containing the current values of a, b, and c. If any of a, b, and c happen to have equal values, then the set will have fewer than three members. The set containing a single object a is denoted {a}, and it is distinct from a itself. There is no limit to the number of members that may be formed into a set in this way. It may be zero: { } denotes the null set. The special character ∅ also denotes this value.

If S is a set, c is a conditional expression, and e is any expression, then {e(x), ∀x∈S | c(x)} denotes the set formed by selecting members (x) from S, testing to see if the condition c(x) is satisfied, and if so, adding e(x) to the set being formed. For example, if S = {-1, 0, 1, 2, 3}, and R = {x**2, ∀x∈S | x≤2}, then R is {1, 0, 4}. For the purposes of this section,

we may assume that there are no side effects in evaluating c and e.

The following abbreviations are permitted:

$\{e(x),\ \forall x \varepsilon S\}$ denotes $\{e(x),\ \forall x \varepsilon S \mid TRUE\}$

$\{\forall x \varepsilon S \mid c(x)\}$ denotes $\{x,\ \forall x \varepsilon S \mid c(x)\}$

Thus $\{\forall x \varepsilon S\}$ is simply S itself.

The "membership test" is written $x \varepsilon S$, where x is an expression and S is a set expression. The value of $x \varepsilon S$ is TRUE if the current value of x is a member of the current value of S, and FALSE otherwise.

Sets may be formed into the usual programming structures such as arrays and strings. Below is given a definition of the structures that are formally recognized in ASL. These structures are discussed in more detail in the subsequent sections. By "formally recognized in ASL" we mean:

(1) there may be certain built-in operators and predefined functions that can be applied to the structure, but not to a set in general,

(2) there is a keyword corresponding to the structure, that may be used in a DECLARE statement, and

(3) the structure may be implemented in a way particularly efficient to it, although not suitable for sets in general.

A <u>regular vector</u> is a set (possibly null) all of whose members (if any) are regular vectors of two components with the first component a unique integer, such that all integers from 1 to n are used, where n is the number of members in the set. A regular vector is denoted by

(a, b, c, . . .), where there must be at least one comma present.

This definition is circular, and hence in a sense we never "really" know what a regular vector is. This definition is the key to much that is smooth in the ASL treatment of data structures, but it is also the source of a few rough edges. For example, an object that is thought of as a set will print as a vector in an unformatted write operation, if the value happens to be such that it looks like a vector.

Thus we have

$$(a, b) = \{(1, a),\ (2, b)\} = \{\{(1, 1),\ (2, a)\},\ \{(1, 2),\ (2, b)\}\} = \ldots$$

provided a and b are defined.

The notation for a vector involves one or more commas within parentheses or brackets. Parentheses (or brackets) without commas denote only grouping; (a) is a itself. Some other degenerate cases:

( ) denotes no value: the state of being undefined.

(a,) denotes the one-vector containing a, which may also be written ONEVECTOR(a).

(,) denotes the null vector, or null set, as does (a, b) if a and b are undefined.

A variable may be made undefined by the assignment x = ( ); as well as x = ;. The preference is a matter of taste.

A pair is a regular vector of two components.

A string is a regular vector; the two terms may be used interchangeably.

We not define several more general structures in terms of the regular vector, or string. We start with the most general, the relation, and end with the vector, which is a slight generalization of the regular vector.

A relation is a set of regular vectors, all of the same number of components, the number being two or more. A binary relation is a set of pairs. There is no special notation for relations, or for the other structures which are defined below. A self-defining relation is written as a set of regular vectors. For example, a finite subset of the relation "a divides b" may be written:

$$\{(2, 2),\ (2, 4),\ (2, 6),\ (3, 6)\}.$$

Of course in ASL we can only deal with a finite subset of the mathematical relation between integers.

An itemized map is a binary relation in which the first component of each pair is unique. A procedure map is a procedure that is free of side effects. It may be of either the function or operator type. A map is an itemized map or a procedure map. In this document the term "map" is sometimes used to denote one of the two types of maps; the context should make it clear when the word "itemized" or "procedure" has been dropped. An itemized map is sometimes called a set map.

We use the term "function" loosely; at times it means a map in general, and at times it means a procedure of the function type, which may have side effects, and may even give different values for f(x) each time it is invoked (with the same value of x).

We use the terms domain, range, and inverse in a slightly more general sense than the usual mathematical meaning: these terms may be applied to binary relations, which form a superset of the itemized maps.

If R is a binary relation, then the domain, range, and inverse are defined as follows:

$$\mathcal{D}(R) = \{x,\ \forall(x, y)\varepsilon R\}$$

$$\mathcal{R}(R) = \{y,\ \forall(x, y)\varepsilon R\}$$

$$\mathcal{I}(R) = \{(y, x),\ \forall(x, y)\varepsilon R\}$$

An array is an itemized map whose domain consists of strings of integers, all of the same length. The length is the dimension of the array. A matrix is a two-dimensional array.

A vector is an itemized map whose domain consists of integers.

Our definitions allow arrays, matrices, and vectors to be sparse and not necessarily one-origined. However, the elements in the domain of an array (or matrix) must be dense and one-origined.

A one-dimensional array is not a vector. The elements in the domain of a one-dimensional array are one-vectors of integers, whereas those in the domain of a vector are simply integers.

3.2.1 Relations and Maps

ASL concentrates on the set theoretic idea of a map, or a transformation of one object into another according to a specified rule. The image of an object x under a map f is denoted by the familiar f(x), or simply f x. The same notation is used whether f is an itemized map or a procedure map. In fact, if this is the only way a map is used, then the procedure that uses f is independent of whether f is a set or a procedure.

As an example of an itemized map, let

$$f = \{('a', 1), ('b', 2), ('c', 3)\}.$$

Then f('a') is 1, and so forth.

A binary relation is a map with the uniqueness condition dropped: it can be thought of as a "multiple valued" map. However, if R is merely a relation (binary or otherwise), then the notation R(a) is not permitted. Instead, if one wishes to form the set of all images of a one must write $\{y, \forall(x, y) \in R \mid x = a\}$. Alternatively, a binary relation may be converted into a map by grouping its multiple values into sets. For example, the relation "a divides b", mentioned in the previous section, may be structured as:

$$D = \{(2, \{2, 4, 6\}), (3, \{6\})\}.$$

Then D(2) is $\{2, 4, 6\}$, etc.

The relation is just barely a recognized structure in ASL. The only built-in operators that apply to relations but not to sets in general are $\mathcal{R}$, $\mathcal{D}$, and $\mathcal{I}$ (range, domain, and inverse), and these are restricted to binary relations. A variable may be declared to be a relation, which helps readability. Finally, relations may be implemented in a particularly efficient way. This might amount to merely storing the length of a relation's elements only once, along with the "dope vector" of the set itself. Alternatively, if it is known (through the elaboration language) that a relation is of high density, then the relation can be compactly stored as (1) a map of the objects into small integers, and (2) a bit array of dimensionality equal to the "order" of the relation. For example,

the relation {(a, b), (a, c), (a, d), (b, b), (c, a), (d, a), (d, c)} could be stored as {(a, 1), (b, 2), (c, 3), (d, 4)}, together with the array:

	1	2	3	4
1	0	1	1	1
2	0	1	0	0
3	1	0	0	0
4	1	0	1	0

The operations of putting an element (e.g., pair) into a relation, removing one, and testing for the presence of one are fairly fast, as they involve two functional evaluations (presumably based on hashing) and a double indexing. If a pair with a new component is added, the array must be re-allocated, or extra space may be provided initially.

3.2.2 Arrays, Matrices, Vectors, and Strings

Vectors play an important role in ASL, and they will be discussed first.

Vectors, as well as strings, are written by enclosing the components in parentheses. If the vector is sparse, the missing components are not written. For example,

$$V = ('a',\ 2,\ ,\ \{1, 2, 3\})$$

is a vector in which the first component is the character 'a', the second component is the number 2, the third component is absent, and the fourth component is a set of numbers. This can be viewed as a map from 1 into 'a', 2 into 2, and 4 into $\{1, 2, 3\}$. In fact, the notation V(i) suggests this view, as the same notation is used to obtain the image of i under V,

whether V is a procedure, itemized map, vector, or string.

A vector may be "shifted" by means of the @ operator, and this operator is used to write self-defining vectors that are other than one-origined. For example, the expression

$$(a, b, c)@-1$$

is the set $\{(-1, a), (0, b), (1, c)\}$. A vector that is not one-origined is "irregular". That is, it is not a string. Sparse vectors are also irregular. Some other examples of irregular vectors:

$$(a, , b) = \{(1, a), (3, b)\}$$

$$(, a) = \{(2, a)\}$$

The vector (a,) is the set $\{(1, a)\}$, which is ONEVECTOR a, and is regular. That is, elements may be dropped from the right end of a string, and it remains a string.

With the @ operator, any vector can be written as a "constant", or self-defining value. In fact, this is a general property of ASL: any structure that can be created at execution time can be expressed as a self-defining value, either in a procedure or in an I/O stream.

There are three predefined procedures that operate on vectors. The lowest index of a vector, LI(v), is the least integer in the domain of v, if v is a non-null vector. The highest index of a non-null vector, HI(v), is the greatest such integer. We somewhat arbitrarily define $LI(\emptyset) = 1$ and $HI(\emptyset) = 0$, as these definitions are convenient when working with variable length one-origined vectors. The length of a vector, LENGTH(v), is HI(v) - LI(v) + 1; thus $LENGTH(\emptyset) = 0$.

The size operator, #v, gives the number of members of the set v, which is the number of defined components of the vector v.

Here are some examples of various operations on vectors.

	(a, b, , c)	∅
LI	1	1
HI	4	0
LENGTH	4	0
#	3	0
$\mathcal{D}$	{1, 2, 4}	∅
$\mathcal{R}$	{a, b, c}	∅
$\mathcal{I}$	{(a, 1), (b, 2), (c, 4)}	∅

The various operations just discussed may of course be applied to strings. If s is a string, then LI(s) = 1, HI(s) = LENGTH(s) = #s, $\mathcal{D}$s = {1, 2, 3, . . . , LENGTH(s)}, etc. It should be pointed out that strings need not be homogeneous. That is, (1, 'a', ∅) is a string of length three.

As has been said, an array is a map from strings of integers to arbitrary objects, with the strings being all of the same length. There is no special notation for arrays, and to write very sizeable ones as self-defining values is rather tedious. For example, a simple 2 x 2 array may be written:

$$A = \{((1, 1), a), ((1, 2), b), ((2, 1), c), ((2, 2), d)\}.$$

Arrays may be sparse and not necessarily (1, 1, . . .)-origined. If the array is dense and (1, 1, . . .)-origined, it is <u>regular</u>, otherwise <u>irregular</u>.

For the above array, A(1, 1) is a, A(1, 2) is b, etc. A(1) is undefined; it would be defined if A included a pair such as (1, e). But then A would not be an array, it would just be a map. Cross sections of arrays may

be obtained. For example, A(1, *) is $\{(1, a), (2, b)\}$ = (a, b). Cross sections are discussed in section 4, as are subarrays, subvectors, and substrings.

The vector operations discussed above are generalized to arrays. The @ operator and the LI, HI, and LENGTH predefined functions operate on arrays by using strings to represent the array's extents. For the A above, A@(2, 3) is:

$$\{((2, 3), a), ((2, 4), b), ((3, 3), c), ((3, 4), d)\}.$$

For the original A, LI(A) is (1, 1), HI(A) is (2, 2), LENGTH(A) is (2, 2), and #A is 4. The number of dimensions of an array is LENGTH(LENGTH(A)), or #LENGTH(A), if preferred.

A matrix is a two-dimensional array. There are several operations unique to matrices: these are discussed in section 4.

In a language that includes vectors, but does not include arrays, it is common practice to represent a matrix as a vector of its rows (which are vectors), and similarly for higher order arrays. One may of course have vectors of vectors in ASL, but they are not at all the same as arrays. To emphasize this, let M be the 2 x 2 matrix mentioned above, and let V be its vector counterpart, i.e., V = ((a, b), (c, d)). Then:

M(1, 1) = a	V(1, 1) is undefined ([V(1)](1) = a)
M(1) is undefined	V(1) = (a, b)
LI(M) = (1, 1)	LI(V) = 1
$\mathcal{D}$(M) = {(1, 1), (1, 2), (2, 1), (2, 2)}	$\mathcal{D}$(V) = {1, 2}
$\mathcal{R}$(M) = {a, b, c, d}	$\mathcal{R}$(V) = {(a, b), (c, d)}
$\mathcal{L}$(M)(a) = (1, 1)	$\mathcal{L}$(V)(a) is undefined

Some structures that are similar and are sometimes treated as indistinguishable in mathematics, but which are distinct in ASL, are:

V = (a, b, c) = {(1, a), (2, b), (3, c)}

S1 = {((1,), a), ((2,), b), ((3,), c)}

S2 = {((, 1), a), ((, 2), b), ((, 3), c)}

M1 = {((1, 1), a), ((1, 2), b), ((1, 3), c)}

M2 = {((1, 1), a), ((2, 1), b), ((3, 1), c)}

Structures such as M1 and M2 will be called "row matrices" and "column matrices", respectively.

3.2.3 Numeric Tolerance of Set Members

There are a few side effects associated with forming sets (and vectors) of numbers. These will not usually be troublesome, as they involve numbers that are very close together (within the tolerance of about 10^{-13}), and the fact that two such numbers are considered to be equal in ASL.

First, consider a simple set of numbers. If x is any approximate number, suppose the set S = {x} is formed, and then the numbers x-2T and x+2T (where T is the tolerance) are put into the set S. Then the equality test will judge that x-2T and x+2T are already in S, and S will not be altered; #S is one. But if first the set S = {x-2T} were formed, and then x and x+2T were added to it, then the final S would be {x-2T, x+2T}; #S is two. The result of building a set may depend upon the order in which it is built.

Suppose x, y, and z are clearly distinct approximate numbers, and suppose the map M = {(x, a), (y, b), (z, c)} is formed. Then M maps all exact numbers between x-T and x+T into a, and it maps all approximate numbers from x-2T to x+2T into a. If an attempt were made to add a new pair (r, a) to the map, with r = x according to the equality test, then the new pair would not be added.

At any point in time, the map M may be inspected to see if x, y, and z are approximate and within the tolerance of being integers. If they are, then they may be changed to integers in exact mode. This alters the

mapping done by M: it now maps an exact number precisely equal to x into a, and an approximate number from x-T to x+T into a.

The reason for the change is that if M were built up precisely as specified, then it might be something like the set

$$\{(0.9999999999999, a),\ (2.0000000000001, b),\ (2.9999999999999, c)\}.$$

If the domain of such a map consists of approximate numbers, then the implementation is free to assume that round off errors have occurred, and that what was intended was the vector (a, b, c). The map may then be changed to (a, b, c), thus acquiring obvious benefits of compact representation and efficiency of accessing via indexing.

A reasonable algorithm for an implementation to use is to inspect an object when it is put into a set, and if it is of the form ((a, b, c, ...), d), and if a, b, c, ... are all integers (with the tolerance being used for those numbers that are approximate), then the object could be changed to ((i, j, k, ...), a), with i, j, k, ... exact integers.

Consider the comparison of the two sets $S_1 = \{x\}$ and $S_2 = \{x-2T,\ x+2T\}$, where x is an approximate number. If the set equality test merely required that each member of S_1 be in S_2, and vice versa, for the sets to be equal, then S_1 and S_2 would be judged equal. That is why we stated in the beginning of section 3.2 that two sets are equal if they have the same number of members, and each member of one is a member of the other.

4. Expressions

The forming and evaluation of expressions is largely conventional. Expressions may be arbitrarily long and arbitrarily deeply nested. The order of evaluation is determined by parentheses and operator precedences, which are given in section 2.2.3.1. When necessary to impose an arbitrary ordering, it is usually taken to be left-to-right, but it may be unspecified. For example, A - B + C is taken to be (A - B) + C, but the optimizer is free to evaluate A + B + C in any order it wishes (provided no side effects are involved).

These aspects of ASL expressions are not entirely conventional:

1. Suffix, as well as prefix, monadic operators are included.
2. There is one "parentheses type" operator, absolute value.
3. The user may define his own operators.
4. There is one triadic operator, the ellipsis.
5. Compiler temporaries are freely used to minimize side effects, or rather to treat them in a way that is "natural".
6. An expression such as A $<$B $<$C is allowed, as an abbreviation for A $<$B & B $<$C.

We shall first discuss the manner of writing and the meaning of all valid expressions, with little or no regard for side effects. The meaning in the presence of side effects will then be clarified in section 4.10, Expression Evaluation.

An expression is any of the following:

1. A self-defining value.
2. A name (variable, label, procedure name, or declared constant).
3. Two juxtaposed expressions (a functional application).
4. An expression enclosed in parentheses or brackets.
5. An expression preceded by a prefix operator or followed by a suffix operator.
6. Two expressions connected by an infix binary operator.
7. Three expressions connected by the ellipsis.
8. A vector former or set former expression (see section 4.3).
9. A search expression or quantifier predicate (see section 4.4).

There are no implicit data conversions done during expression evaluation.

All operations fall into three classes as determined by their outcome:

1. Normal
2. Undefined
3. Invalid

The normal operations are those that produce a "useable" result. By "undefined" operations we mean that no result is produced, but execution continues. The "invalid" operations are those that cause program termination.

An operation is undefined, i.e., it produces an undefined result, if it is valid but the result cannot be expressed for one reason or another. Generally speaking, an undefined result signifies that the operands were of reasonable and compatible forms, but their values were such that the

operation could not be done. For example, an undefined result is produced when an arithmetic overflow occurs (signifying that an approximate number is out of range for the machine) and when it is attempted to invert a singular matrix, or a matrix that is sufficiently close to singular so that the ASL built-in routines cannot invert it. Division by zero also produces an undefined result, even if the divisor is an exact zero.

Arithmetic overflow applies only to approximate numbers. An attempt to produce extremely large exact numbers is a different situation, and it results in termination due to insufficient storage. For example, the operation 100!! might be considered valid, but it would no doubt cause termination due to insufficient main storage to contain the result.

4.1 Operators

4.1.1 Equality Test

The equality test is written "a = b", where a and b are expressions. Its negation, $\neg(a = b)$, may be written $a \neq b$. The operands a and b may have any value whatever, but they must have some value (they must be defined). The result of the equality test "a = b" is TRUE ('1'B) if the current value of a equals that of b, and FALSE ('0'B) otherwise.

Generally speaking, for two expressions to be equal their values must be the same in every respect. Numbers are an exception, however. Two numbers x and y are considered to be equal if $|x - y| \leqslant \delta$, where $\delta = 0$ if x and y are both exact, $\delta = T$ (the numerical tolerance of about 10^{-13})

if one is exact and the other is approximate, and $\delta = 2T$ if both numbers are approximate.

Two atoms are equal if they have the same type and the same value. Thus the number 1, the character '1', and the Boolean value '1'B are all distinct. Two sets are equal if they have the same number of members and each member of one is contained in the other, as determined by the equality test (thus equality testing is a recursive process). From this it follows that vectors and arrays are equal only if they have the same number of components, the same origin, and the same values of corresponding components. There is no extension of the shorter operand in string comparisons (as is done in PL/I).

The notation "a = b = c" is permitted and, in the absense of side effects, it is equivalent to a = b & b = c. This may be extended to any number of operands, and the ≠ sign may also be used. For example, a = b ≠ c = d is (in the absense of side effects) equivalent to a = b & b ≠ c & c = d.

The evaluation of $e_1\ R_1\ e_2\ R_2 \ldots R_{n-1}\ e_n$, where the e_i are expressions and the R_i are either = or ≠ operators, is equivalent to:

```
r = FALSE
t2 = e2
IF e1 R1 t2 THEN DO
   t3 = e3
   IF t2 R2 t3 THEN DO
      ...
      IF tn-1 Rn-1 en THEN r = TRUE
      ...
      END IF t2
   END IF e1
```

Here r is set to the value of the expression, and the t_i are compiler temporaries. The main point is that when the shorter notation is used, the expressions e_i are evaluated only once (at most).

4.1.2 Existence Test

The existence test is written "∃e", where e is an expression. The value of ∃e is TRUE if e has a value (is defined), and FALSE otherwise.

The state of having no value (being undefined) originates in these ways:

1. It is the initial state of a procedure's local variables before the procedure has assigned them a value.
2. It is the result of an operation in which the operands are of valid types and forms (in the case of sets), but the values are such that no result exists.

Referencing a map with the argument not in the domain of the map, as for example a vector with the subscript out of range, is a special case of (2) above. A procedure may signify that its argument is not in its domain by executing the statement RETURN; or RETURN ( ); or RETURN e; with e undefined. Unlike an itemized map, if the argument is not in a procedure's domain, the procedure may terminate by attempting to execute an illegal operation, it may normally terminate, or it may loop indefinitely.

An expression u which has no value may appear in these contexts only:

1. The existence test (∃u).
2. A function reference such as y = u(x) or u(x) = e (assignment).

3. The right-hand side of an assignment statement (x = u; f(x) = u; etc.).

4. A component of a vector (e.g., (x, u, y)).

5. In a RETURN statement (RETURN u).

In cases (3), (4), and (5), the undefined state may be explicitly indicated, e.g., x = ;, (x, , y), RETURN;. Case (4) applies to function arguments, as in f(x, u, y) or f(x, , y).

To emphasize, if u is undefined then the expressions $\{u\}$ and u = x (comparison) are invalid. An attempt to put u into a set or to compare it with another quantity leads to termination of execution.

Occasionally in mathematics the conditional "x = y" is used, when the possibility exists that one operand (but usually not both) may be undefined. It is assumed in such a case that "x = y" is defined and is "false". However, for the sake of clarity, in ASL we require the programmer to write, for example, "$\exists$x & x = y". Conditionals are evaluated in left-to-right order with evaluation terminating when the outcome is known; hence if x is undefined, then $\exists$x is false, and "x = y" is not evaluated.

Regarding the evaluation of expressions, the general use of the undefined state is to produce it when the operands are of valid type, but the result does not exist, or cannot be expressed, for one reason or another. The program may then test for this outcome and take appropriate corrective action. However, if a nonexistent result is used in another

operation, other than those mentioned above, then the programmer is assumed to have made an oversight, and execution terminates. For example, if an arithmetic overflow occurs, there is no result. The programmer may then test for overflow after performing the operation, which is easier than testing for it beforehand. Similarly, after a matrix inversion one may test to see if the result exists.

4.1.3 Arithmetic Operators

The arithmetic operators are:

Addition (x + y)
Subtraction (x - y)
Multiplication (x*y)
Division (x/y)
Exponentiation (x**y)
Absolute Value, or Norm (|x|)
Factorial (x!)

With the exception of the factorial, these elementary operations are defined for matrix and vector operations, as well as for simple numeric operations. For example, if A and B are matrices, then A*B is the conventional matrix product, |A| is a certain norm of A (described below), etc.

As discussed in section 4.1.5, the Boolean operations are also defined for matrices and vectors. But that is as far as ASL goes in this direction. The predefined functionsgenerally do not accept matrix or vector operands. For example, although one can define the square root and the inverse tangent of a matrix, the ASL SQRT(x) and ATAN(x) terminate if x is anything other than a number. Of course some predefined functions operate on vectors and matrices, e.g. HI(v) and DET(A) (determinant).

The elementary arithmetic operations are actually defined over a class of objects that is larger than matrices and vectors as they are defined in ASL. The enlargement consists of not requiring that the "indexes" of a matrix or vector be numeric. For the arithmetic (and Boolean) operations, we define two structures: a generalized matrix is

an itemized map whose domain consists entirely of pairs, and a generalized vector is any itemized map that is not a generalized matrix. Thus we have simply taken the entire class of itemized maps and separated it into two subclasses: those that are considered to be matrix-like and those that are not.

This manner of recognizing a generalized matrix leads to anomalies that may occasionally be troublesome. One may be dealing with objects that are thought of as "vectors", and their domains may consist of objects whose structure varies in a way that is hard to predict. One might multiply two such objects together, meaning to employ the vector product (ASL uses the dot product for vector multiplication). However, if the objects happen to qualify as "generalized matrices", then the matrix product will apply, and the result will not be what was intended or, more likely, execution will terminate. In the great majority of cases, however, the data structures are of a fixed format that is known when a program is being written, and no ambiguity arises.

With the exception of the factorial, the definitions of the arithmetic operations on data structures are recursive. Thus we can add or multiply matrices of matrices, vectors of matrices of matrices, etc.

For an arithmetic operation to be valid, the operands must be compatible. The first requirement is that both operands must be itemized maps (or simply numbers). The next requirement is that their domains must satisfy certain conditions, which are described below for each

operation. The arithmetic operations on the elements in the range of the map must be valid. For example, matrix multiplication is defined in terms of vector multiplication. For the matrix multiplication to be valid, each vector multiplication that it implies must be valid. All arithmetic operations ultimately reduce to operations on atoms. The last requirement for validity is that the atoms must be numeric.

If the evaluation of an arithmetic operation on structures produces an undefined intermediate result, then the entire result is made undefined. For example, if in adding two vectors, one component overflows, then the entire result is made undefined. This may seem to be an overly severe action to take, but there are several reasons why it is done. One is that it gives the programmer a simple way to check that an operation was normal. He need not search an entire structure to see if some part of it does not exist. Another reason is that we wish to have simple rules such as "the addition of two regular vectors is a regular vector, if the result exists." To give up such rules would make ASL programs less transparent and would negate many optimization possibilities. Lastly, our treatment allows us to be more consistent. We do not have to state that a nonexistent value is allowed to enter into an arithmetic operation provided the arithmetic operation is being invoked to calculate a "larger" arithmetic operation. Instead, it is understood that as soon as a result is undefined, the whole outermost operation terminates, and the result is made undefined. As a simple example, consider the dot product of two

numeric vectors. If a multiplication causes an overflow, we prefer to say that the evaluation terminates, rather than to say that it continues but that a special "add" routine is used that allows one operand to be nonexistent.

4.1.3.1 Addition and Subtraction

As has already been mentioned, addition and subtraction are defined for itemized maps. That is, if f and g are maps with identical domains, then

$$f + g = \{(x,\ f(x)+g(x)),\ \forall x \in \mathcal{D} f\},$$

and the result is defined if and only if f(x) + g(x) is defined for all x in the common domain. The sum f(x) + g(x) is defined if both f(x) and g(x) are numeric, or if they fit this definition. Subtraction is defined similarly.

Addition and subtraction have been defined recursively, which is illustrated by the following addition of two vectors:

(1, (2, 3), (4, (5, 6))) + (7, (8, 9), (10, (11, 12)))

= (8, (10, 12), (14, (16, 18)))

If addition or subtraction is attempted in any of the following situations, then the operation is invalid and execution terminates:

1. An operand is undefined.
2. The domains are not identical.
3. An operand is a non-numeric atom, or a set that contains non-pairs, or a set that contains pairs that have the same first components.

The last part of (3) will frequently be stated "... a set that is not a map".

The null set vacuously satisfies the conditions for a valid addition or subtraction operand; we have $\emptyset + \emptyset = \emptyset - \emptyset = \emptyset$.

As was mentioned in section 3.1.1, the addition of two exact numbers produces an exact result, but if either operand is approximate, the result is approximate. In the latter case, the addition or subtraction is done in approximate mode; this is pointed out because the conversion of an exact number to approximate mode may cause an overflow condition (see section 3.1.1) which may not have occurred if the operation were done in exact mode.

Addition and subtraction of exact numbers is carried out according to the formula

$$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm bc}{bd},$$

where a, b, c, and d are integers. The result is then reduced to lowest terms.

If x is numeric, then -x means 0 - x, and +x means 0 + x. The meaning of -x, +x, 0 - x, and 0 + x when x is not numeric is not specified. This is left open to allow the optimizer to change an expression such as --x to x. That is, the question of whether or not "--x" is valid when x is non-numeric is dependent upon how much optimization is done, which in turn depends upon the implementation and possibly the declaration, if any, of x. Unusual expressions such as --x and 0 + x occur most often in practice as a result of preprocessor activity.

4.1.3.2 Multiplication

Multiplication is defined in a way that includes ordinary scalar multiplication, the vector dot product, and matrix multiplication.

The product of two exact numbers is exact. If either operand is approximate, the result is approximate and the multiplication is done in approximate mode.

The product of a number and a map is the map obtained by multiplying all members of the range of the map by the number. That is, if a is a number and f is a map, then

$$a * f = f * a = \{[x,\ a*f(x)],\ \forall x \in \mathcal{D} f\}.$$

The operation is valid and defined if and only if a*f(x) is valid and defined for all x in the domain of f.

If f and g are generalized vectors, and they have the same domain, then

$$f * g = \sum_{x \in \mathcal{D} f} f(x) * g(x).$$

This is called the generalized dot product. Expressed in ASL, f * g is the value computed by the somewhat formidable procedure below, which defines f * g for arbitrary f and g in terms of f * g for numbers.

```
FUNCTION p(f, g)
IF NUMERIC f & NUMERIC g THEN RETURN f*g
IF ATOM f ∨ 𝔇f ≠ 𝔇g THEN STOP "Invalid operation."
∀x ε 𝔇f DO
   temp = p(f(x), g(x))
   IF ¬∃temp THEN RETURN; "Undefined."
   IF ITERATION(∀x) = 1 THEN p = temp
                        ELSE p = p + temp
   IF ¬∃p THEN RETURN;
   END ∀x
RETURN p
END FUNCTION p(f, g)
```

If A and B are generalized matrices, then

$$A * B = \{[(x, y),\ A(x, *) * B(*, y)],\ \forall x \varepsilon\ \mathfrak{D}\mathfrak{D}A,\ \forall y \varepsilon\ \mathfrak{R}\mathfrak{D}B\}.$$

Here we have used the "cross section" notation (see section 4.6). A(x, *) is row x of A, and B(*, y) is column y of B. A*B is a valid operation if (and only if) A(x, *) * B(*, y) is valid in all cases. If x and y are not pairs (so that A(x, *) and B(*, y) are not matrices) then the "generalized dot product" of maps applies, and we must have

$$\mathfrak{D}[A(x, *)] = \mathfrak{D}[B(*, y)].$$

Expressed in another way, for A*B to be valid, we must have $\mathfrak{R}\mathfrak{D}A = \mathfrak{D}\mathfrak{D}B$. This latter condition is not always the requirement, however. For example, if x and y are pairs of atoms, then the condition becomes $\mathfrak{R}\mathfrak{R}\mathfrak{D}A = \mathfrak{D}\mathfrak{D}\mathfrak{D}B$.

The above definition is called the <u>generalized matrix product</u>. For matrices of numbers, it reduces to the familiar

$$C_{ij} = \sum_k A_{ik}B_{kj}.$$

The matrices need not be regular, but we must have the rows of A defined over the same domain as the columns of B.

These definitions of multiplication allow certain "strange" structures as operands. The simplest is one whose elements are indexed as $A[(i,j),(k,l)]$. For such a structure, matrix multiplication applies, and since its rows and columns are also matrices, matrix multiplication applies at the first two levels. Such a structure is similar to a matrix of matrices, but not the same; the latter would have its elements indexed as $[A(i,j)](k,l)$.

This generalization of matrix multiplication is not idle generalization for its own sake. In many, and perhaps most, applications, the "indexes" of a matrix are most naturally not integers. For example, one might choose to represent a directed graph by a square matrix of 1's and 0's (numbers, not truth values), in which M(a, b) is 1 if node a is connected to node b, and 0 otherwise. The matrix is indexed by some sort of node identifier which need not be numeric. Then the n^{th} power of such a matrix, using ASL multiplication, expresses the number of ways one can get from node a to node b by traversing exactly n edges. There are countless other cases where matrix multiplication turns up with non-numeric indexes.

Generalized matrices and generalized vectors may be multiplied together as follows. If A is a generalized matrix, and f is a generalized vector, then

$$A*f = \{[x,\ A(x,*) * f],\ \forall x \in \mathcal{D}\uparrow\mathcal{D}A\}$$

$$\text{and } f*A = \{[y,\ f * A(*,y)],\ \forall y \in \mathcal{R}\mathcal{D}A\}.$$

This is similar to matrix multiplication with f treated as a column (in the first case) or row (in the second case) matrix. Except in special cases the result is a vector, however, and not a column or row matrix. The conditions under which the operation is valid and the result is defined are similar to those for matrix multiplication.

This product and the generalized matrix product are not communicative but the others (generalized dot product and multiplication by a scalar) are.

4.1.3.3 Division

Division is the inverse of multiplication, when the inverse exists, and is either undefined or invalid otherwise.

The quotient of two exact numbers is exact. If either operand is approximate, the result is approximate and the division is done in approximate mode. If the divisor is zero or if an overflow occurs, the result is undefined.

If a is a scalar and f is a map, then f/a is f*(1/a), except that if a is zero, then f/a is undefined (f*(1/a) would be invalid). The operation a/f is invalid.

If f and g are maps, then f/g is invalid unless f and g are generalized matrices, in which case f/g may be defined, as follows. It is the map M satisfying

$$g * M = f$$

if such a map exists. If the forms of f and g are such that such a map

could not possibly exist, then f/g is invalid. Otherwise, the operation is valid but the result may be undefined. As a first condition on validity, we must have $\mathfrak{D}\mathfrak{D}f = \mathfrak{D}\mathfrak{D}g$. In addition, we must have $\#\mathfrak{D}\mathfrak{D}g = \#\mathfrak{R}\mathfrak{D}g$. This expresses the fact that there must be as many equations as there are unknowns. If f and g map pairs of atoms into atoms, then these conditions are sufficient for the validity of f/g. However, the result will be undefined if the postulated map does not exist.

4.1.3.4 Exponentiation

Exponentiation, x^y, is written x**y. Unlike all other binary operators, exponentiation is grouped to the right: x**y**z means x**(y**z) (see section 2.2.3.1).

If x and y are numbers, then x^y is exact if and only if both x and y are exact, and y is an integer. The result is defined only if x and y are in the domain of exponentiation and no overflow occurs. For numerical x and y, the domain of exponentiation is:

$x > 0$,
$x = 0,\ y > 0$,
$x < 0,$ y an integer.

This is essentially the range of values for which x^y is real. There are exceptions, however. For example, if $x < 0$ then $x^{1/3}$ has a real value but the ASL x**(1/3) is undefined.

Exponentiation is also defined for the raising of generalized vectors to non-negative integral powers. If f is a generalized vector, then f^n

is f*f* ... *f (n times); f^n is a number for even n and a generalized vector for odd n. f^0 is 1.

If M is a generalized matrix, then M^n is similarly defined, but only if ⊵⊵M = ⍀⊵M (except possibly for the case n = 1). In such a case M^0 is defined; it is the generalized matrix I with the same form as M, with identity elements on the diagonal, and with zero elements elsewhere. The diagonal elements are the elements M(x, x). The "identity elements" are arrived at by inspecting each M(x, x), and if M(x, x) is a generalized matrix, then I(x, x) is its identity element if it has one (this is a recursive definition) and the entire result is undefined if it does not. Otherwise (M(x, x) is not a generalized matrix), I(x, x) is the number 1. The "zero elements" are defined similarly.

This somewhat involved definition of M^0 is used so that M^0 can be added to M, as in the series $M^0 + M^1 + M^2 + \ldots$. Such a series is valid for matrices of matrices of ... of numbers, but it is not valid for matrices of vectors.

The meaning of f**1 is f if f is a map, and is otherwise unspecified. Hence an optimizer may simplify f**1 to f without regard for what f is.

4.1.3.5 Absolute Value or Norm

If x is a number, then |x| is a nonnegative number with the same magnitude. The precision of |x| is the same as that of x. The result is undefined if changing the sign of x causes an overflow condition.

If f is either a generalized vector or a generalized matrix, that is, a map, then

$$|f| = (\sum_{y \varepsilon \mathcal{R} f} |y|^2)^{1/2},$$

provided this is defined.

This is a recursive definition, and hence it defines the norm of a vector of vectors, a matrix of vectors, etc. The only requirement is that the map ultimately reduce to numbers; the norm of a character, Boolean value, pointer, or procedure is an invalid operation.

If f is a map of maps (vector of vectors, etc.) then the above becomes:

$$|f| = (\sum_{y \varepsilon \mathcal{R} f} \sum_{z \varepsilon \mathcal{R} y} |z|^2)^{1/2}.$$

For example (using a non-ASL notation for matrices):

$$|(1, \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}, (6, 7, 8))| = |(1, 2, 3, 4, 5, 6, 7, 8)| = \sqrt{204}.$$

4.1.3.5.1 Norm, Theoretical Remarks

Much of this section comes from Computational Methods of Linear Algebra, by V. N. Faddeeva (Dover, 1959). She defines a norm of a vector x as an associated nonnegative number |x| (in ASL notation) satisfying:

1. $|x| > 0$ for $x \neq 0$, and $|0| = 0$,

2. $|cx| = |c| \cdot |x|$ for a numerical multiplier c, and

3. $|x + y| \leqslant |x| + |y|$ (the triangular inequality).

A norm of a matrix A satisfies the above with x replaced by A, and in addition:

4. $|AB| \leqslant |A| \cdot |B|$.

The ASL norm obviously satisfies (1) - (3). For vector products, (4) is the familiar Cauchy-Schwarz inequality, which is equivalent to the triangular inequality. Inequality (4) may be verified for matrix products by repeated application of the Cauchy-Schwarz inequality, as follows, where A_i denotes row i of A and B^i denotes column i of B:

$$
\begin{aligned}
|A|^2|B|^2 &= (|A_1|^2 + |A_2|^2 + \ldots + |A_n|^2)\,(|B^1|^2 + |B^2|^2 + \ldots + |B^m|^2) \\
&= |A_1|^2|B^1|^2 + |A_1|^2|B^2|^2 + \ldots + |A_n|^2|B^m|^2 \\
&\geqslant |A_1B^1|^2 + |A_1B^2|^2 + \ldots + |A_nB^m|^2 \\
&\geqslant |(A_1B^1,\ A_1B^2,\ \ldots,\ A_nB^m|^2 \\
&\geqslant |AB|^2
\end{aligned}
$$

Hence the ASL $|A|$ is a norm.

Faddeeva defines three vector norms, which are the "p-norms" for p = 1, 2, and infinity:

$$|X|_p = (\sum |X_i|^p)^{1/p}.$$

For p = 1, this is the sum of the absolute values of the components of X. For p = 2 it is the familiar length, or magnitude, of X. For $p = \infty$ it is the absolute value of the largest component of X. The 2-norm

has by far the widest application, and hence it is chosen for the ASL vector norm. The advantage of the 1- and ∞-norms is that they are easy to compute, particularly for hand calculations, and they lead to easily computable matrix norms, as follows.

Faddeeva defines a matrix norm as compatible with a given vector norm if for any matrix A and any vector $X \neq 0$,

$$|A| \geqslant \frac{|AX|}{|X|}.$$

The ASL matrix norm is compatible with the vector norm, because for any matrix A and vector X:

$$\begin{aligned} |A|^2|X|^2 &= (|A_1|^2 + |A_2|^2 + \ldots + |A_n|^2)\,|X|^2 \\ &= |A_1|^2|X|^2 + |A_2|^2|X|^2 + \ldots + |A_n|^2|X|^2 \\ &\geqslant |A_1X|^2 + |A_2X|^2 + \ldots + |A_nX|^2 \\ &\geqslant |AX|^2. \end{aligned}$$

Faddeeva calls a matrix norm subordinate to a given vector norm if the compatibility condition is just barely met, i.e., if

$$|A| = \max_X \frac{|AX|}{|X|}.$$

On this basis, she proves that the matrix norms subordinate to the three vector norms being considered are:

1-norm: $\max_j \sum_i |a_{ij}|$

2-norm: λ_1, where λ_1^2 is the largest eigenvalue of A^TA (A^T = transpose of A).

∞-norm: $\max_i \sum_j |a_{ij}|$

Consideration was given to having the above matrix 2-norm the ASL matrix norm, as it seems attractive on the theoretical grounds just outlined. The main trouble with it is that it's too hard to compute.

The ASL matrix norm, $(\sum_i \sum_j |a_{ij}|^2)^{1/2}$, is easy to compute, as it involves only n^2 multiplications and additions (for a square matrix) and there are no significant accuracy problems. Furthermore, it is applicable to a wider class of structures than the subordinate matrix 2-norm, for example to non-square matrices, to sparse matrices, to matrices whose elements are mixed numbers, vectors, and matrices, etc. The fact that the same norm applies to both matrices and vectors (and in fact to maps in general) is a simplification to ASL.

The ASL matrix norm is equal to the square root of the sum of the eigenvalues of A^TA. This follows from the facts that the sum of the eigenvalues of any matrix equals the trace of the matrix (sum of its diagonal elements), and the trace of A^TA is the sum of the squares of the elements of A:

$$\lambda_1^2 + \lambda_2^2 + \ldots + \lambda_n^2 = \mathrm{tr}(A^TA) = \sum_i \sum_j a_{ij}^2 = |A|^2.$$

The smallness of a matrix norm implies various convergence properties of the matrix. Since the ASL norm is larger than the subordinate 2-norm, and since if either norm approaches zero then the other does, these theorems generally hold with a margin of safety. For example:

1. $A^n - A \longrightarrow 0$ if and only if $|A^n| - |A| \longrightarrow 0$, and
2. if $|A| < 1$ then $A^n \longrightarrow 0$.

Occasionally one wants an upper bound on the eigenvalues of a matrix. |A| is obviously such an upper bound.

One frequently encounters the expression $(X*X)^{1/2}$ for the vector norm. We have avoided this notation in favor of one that applies to a larger class of objects, for example to a vector of matrices.

4.1.3.6 Factorial

The factorial, n!, is defined only for n a nonnegative integer. 0! = 1 and n! = n(n-1)! for $n \geqslant 1$. If n is approximate, it is converted to exact mode, and the result is always in exact mode. If n is numeric but not a nonnegative integer, or if an overflow occurs, then n! is undefined. If n is not numeric, the operation is invalid and execution terminates.

4.1.4 Numerical Comparison Operators

The numerical comparison operators are $<$, $\leqslant$, $>$, and $\geqslant$, with their usual mathematical meaning. If e is an expression whose value is exact, a is approximate, and T is the numerical tolerance, then the meanings are as follows:

$e_1 < e_2$ means $e_1 < e_2$

$e < a$ means $e < a-T$

$e \leqslant a$ means $e \leqslant a+T$

$a_1 < a_2$ means $a_1 < a_2-2T$

$a_1 \leqslant a_2$ means $a_1 \leqslant a_2+2T$

The remaining comparison operators are defined similarly ($a < e$ means $a+T < e$, $e > a \equiv \neg(e \leqslant a)$, etc.).

If both operands are exact, then the comparison is done in exact mode: $a/b < c/d$ is equivalent to $ad < bc$, as the denominators are always positive. If either operand is approximate, then the comparison is done in approximate mode. Hence the above relationships involving T are themselves approximate, being limited to the accuracy of the machine.

If an overflow occurs in a conditional expression due to an operation such as the multiplication in "a*b < c", then execution terminates because an operand of a comparison is undefined. However, if overflow occurs in evaluating the comparison operation itself (as might happen if comparison is implemented by subtraction), then the result is undefined. If the

conditional expression is in an IF header or set former, for example, then execution terminates. If its value is merely assigned to a value receiving expression, on the other hand, then execution continues.

As in the case of the equality test, comparison operations such as $a < b \leqslant c$, $a < b > c \leqslant d$, etc., are allowed. These are taken to be abbreviations for $a < b$ & $b \leqslant c$ and $a < b$ & $b > c$ & $c \leqslant d$, respectively, except that the intermediate expressions are evaluated only once when the abbreviated notation is used. The exact meaning of the shorter notation is similar to that given in section 4.1.1, Equality Test.

Note that in ASL the expression $0 \leqslant 2 \leqslant 1$ is FALSE, whereas it is (unfortunately) TRUE in both PL/I and APL.

4.1.5 Boolean Operators

The Boolean operators, which operate on Boolean values to produce Boolean results, are:

Not ($\neg a$)
And, Nand (a & b, $a \not\& b$)
Or, Nor ($a \vee b$, $a \not\vee b$)
Implies and its negation ($a \Longrightarrow b$, $a \not\Longrightarrow b$)
Equivalence, Exclusive or ($a \equiv b$, $a \not\equiv b$)

These suffice to provide all twelve nontrivial Boolean functions of two variables:

a	b	a&b	$a \not\Longrightarrow b$	$b \not\Longrightarrow a$	$a \not\equiv b$	$a \vee b$	$a \not\vee b$	$a \equiv b$	$\neg b$	$b \Longrightarrow a$	$\neg a$	$a \Longrightarrow b$	$a \not\& b$
F	F	F	F	F	F	F	T	T	T	T	T	T	T
F	T	F	F	T	T	T	F	F	F	F	T	T	T
T	F	F	T	F	T	T	F	F	T	T	F	F	T
T	T	T	F	F	F	T	F	T	F	T	F	T	F

The Boolean operators may also be applied to maps of Boolean values, maps of maps of Boolean values, etc., in a manner similar to the arithmetic operators. However, for Boolean operations there is nothing analogous to arithmetical matrix multiplication. Boolean matrices are treated the same as any Boolean maps.

If "OP" is a Boolean operator, and f and g are maps with the same domains, then

$$f\ OP\ g = \{(x,\ f(x)\ OP\ g(x)),\ \forall x \in \mathcal{D}f\}.$$

For a Boolean operation on maps to be valid, the domains of the two maps must be identical. In particular, if the two maps are Boolean strings, they must be of the same length. To get the PL/I effect of extension of the shorter operand to the right with zeros, one must be explicit; for example (where a is the shorter operand):

$$[a \vee b(1:\#a)] \not\subset b(\#a+1:\#b).$$

Every Boolean operation on null maps produces the null map as the result, i.e., $\neg\emptyset = \emptyset$, $\emptyset\ \&\ \emptyset = \emptyset$, etc.

For Boolean atoms, the equality test ($=$, $\neq$) could be used in place of the Boolean equivalence operators ($\equiv$, $\not\equiv$). However, for Boolean maps the equality test and the equivalence operator are entirely different operations. It should be stressed that the equivalence operator may only be used for Boolean map operands.

The result of a Boolean operation always exists or the operation is invalid; it is never undefined.

4.1.6 Pointer Operators

The pointer operators are the up arrow and the down arrow. The value of "↑expr" is a "pointer", or "reference", to the current value of "expr". The pointer is a unique data type. If p is a pointer, then ↓p is the quantity pointed to by p.

Why have references? They are somewhat out of character for ASL, which is very "value oriented". Furthermore they are not entirely necessary for the description of algorithms, and they lead to obscure programs that are hard to follow, often necessitating the drawing of diagrams.

Three reasons for having pointer variables in ASL are:

1. for the "control block effect",
2. for the accurate description of computer-oriented algorithms, and
3. for efficiency.

Control blocks have the property that when something in them changes, then the change is reflected by the action of algorithms that have references to the control block. To get the control block effect without pointer variables, one would probably introduce a mapping of integers (or any convenient objects) into the values of the control block. For example, consider the following two code sequences.

Without Pointers	With Pointers
TAG = TAG + 1	
p = TAG	
C(p) = (TRUE, TRUE, TRUE)	p = ↑(TRUE, TRUE, TRUE)
Store copies of p	Store copies of p
...	...
Fetch any copy of p	Fetch any copy of p
(C(p))(2) = FALSE	(↓p)(2) = FALSE

In the first code sequence, a global variable TAG is incremented, and the new value is used as a pointer. It is saved in variable "p" because other processes may alter TAG. A map C is defined or augmented that maps p into a control block consisting of three flags, all of which are initially TRUE. The reference to the control block, p, is then stored away in various places such as members of sets. Subsequently, a reference to the control block is obtained and the second flag is changed to FALSE. If, after this, another copy of "p" is obtained, then a reference to (C(p))(2) will reflect the new value FALSE.

The second code sequence accomplishes the same purpose, but uses a pointer variable, rather than an integer, for p. These steps seem more direct and to the point, mainly because it is not necessary to introduce the map C.

A second reason for having pointer variables in ASL is to facilitate the accurate specification of computer-oriented algorithms that employ "address variables". For example, one might wish to use ASL as an aid in developing algebraic simplification algorithms that feature the sharing of data (as in ALTRAN). Pointers also permit ASL to rather

accurately imitate list processing languages such as LISP.

Finally, the introduction of pointers in ASL may at times allow one to write faster-running or more compact programs. This motivation for pointers is somewhat out of character for ASL, as it will usually lead to more obscure algorithms, but there will no doubt be times when it is justified.

When the term "↑expr" is encountered, memory space is allocated. Then "expr" is evaluated and the result is stored in the memory space. This occurs (or at least ASL acts as if it occurs) even if "expr" is simply a stand-alone constant or variable. It also occurs if "expr" is the name of a set. Thus "↑expr" always causes a copy operation. On the other hand, "↓p" does not in itself cause copying of the thing referenced by p.

Following are some examples of the use of pointers.

```
v = (a, b, c)
pv = ↑v                 "Same as pv = ↑(a, b, c)."
qv = ↑v                 "pv and qv do not compare equal.  They
                         reference different copies of v."
x = (↓pv)(1)            "x is a copy of a."
(↓pv)(1) = 0            "x and v are unchanged."
y = ↓pv                 "The value of y is (0, b, c)."
p1 = ↑1; ↓p1 = 0        "The program's constant '1' is not changed."
```

Note that the notation for pointers is concise, but explicit. It is more concise than PL/I, in which one writes P = ADDR(X) for approximately the effect of p = ↑x (the PL/I P = ADDR((X)) more accurately reflects ASL). Also, in PL/I one writes P->X for the ASL ↓p. The ASL notation is (intentionally) more explicit than Algol 68, in which "y := x" might mean the ASL "y = ↑↓x", depending upon the data types

of y and x. A concise notation for pointers is important because when they are used, they are apt to be heavily used.

The fact that the statement "p = ↑x" makes p point to a copy of x deserves special comment. One reason for this is for simple consistency with statements like "p = ↑(x+1)" and "p = ↑1", in which a copy is desirable. Another reason has to do with optimization.

If "p = ↑x" caused p to point to x itself, then every reference of the form ↓p would be a potential reference to all variables in the compilation that have occurred in the context ↑x. This would cause a degradation in optimization possibilities that is best avoided. For example, in the statements

```
p = ↑y
...
y = ↓q + 1
z = ↓q
```

the expression ↓q could not be factored out because there would be a possibility that ↓q is y.

There is no way to point directly to the value of a program variable. The statement x = ↓p does not cause x to be the object pointed to by p, because assignment causes a copy operation.

As another example contrasting the conventional pointer treatment with ASL's, consider the PL/I:

```
A = P -> D;
X = 0;
B = P -> D;
```

Here the common expression P->D cannot be factored out unless it is

known from a more global analysis that P does not point to X.

In ASL, the above would be coded as one of the following:

case 1	case 2
a = ↓p	a = ↓p
x = 0	↓p = 0
b = ↓p	b = ↓p

In case 1, the common expression ↓p may be factored out, as p cannot possibly point to x's value. In case 2, it is obvious from local analysis that ↓p may not be factored out of the first and last statements, because it is changed by the second statement (however, the optimization of replacing the last statement with "b = 0" may be done).

The ASL code segments above are not only better for optimization than their PL/I counterparts, but they are also easier to understand. In fact, there seems to be a general principle in language design that what is good for optimization is good for readability.

The expression ↑x is valid and defined if x has any value whatever, including a procedure value, but is invalid if x is undefined. The expression ↓p is only valid if p is a pointer. Its value may be undefined (as it is after ↓p = ;). For the null pointer, one would normally use the undefined state for the pointer itself, rather than for the object pointed to, so that the PL/I predicate "P = NULL" becomes "∃p".

4.1.7 Set Operators

The set operators are those that may be applied to arbitrary unordered sets of objects. They are:

Number of members (#S)
Membership test and its negation ($x \in S$, $x \notin S$)
Arbitrary member ($\exists S$)
Union ($S1 \cup S2$)
Intersection ($S1 \cap S2$)
Set difference (S1 - S2)
Proper subset test ($S1 \subset S2$)
Subset test ($S1 \subseteq S2$)
Proper superset test ($S1 \supset S2$)
Superset test ($S1 \supseteq S2$)

In the above, S, S1, and S2 are set expressions, and x is an arbitrary expression. These operations are invalid if either operand is undefined.

The value of #S is an integer giving the number of members in S. We have $\#\emptyset = 0$. The # operator is frequently used for operations that are not usually thought of as set operations, such as the length of a string (for a string S, LENGTH(S) = #S; however, for a sparse vector or array, LENGTH and # produce different results).

It is recommended that an ASL implementation maintain the size of a set along with the set itself, so that #S is a fast operation.

The value of $x \in S$ is TRUE if x is a member of S, and FALSE

otherwise. The expression $x \notin S$ is equivalent to $\neg(x \in S)$.

The value of $\exists S$ is (a copy of) an arbitrary element of S, and is undefined if S is null.

The implementation has a good deal of freedom in implementing $\exists S$, as two successive occurrences of $\exists S$ are not guaranteed to produce the same value, or different values, or to have any particular statistical properties.

The remaining set operations, union, intersection, set difference, subset test, etc., have their conventional mathematical meaning. The union operation is used to add a member to a set, e.g.

$$S = S \cup \{x\}.$$

Of course if x is already present in S, the above statement does not alter S.

The difference $S_1 - S_2$ is the set of members which belong to S_1 but not to S_2, i.e.,

$$\{x, \forall x \in S_1 \mid x \notin S_2\}.$$

The difference is used to delete a member from a set, e.g.,

$$S = S - \{x\}.$$

The subset and superset tests may be combined similarly to the numerical comparison operations. For example,

$$S_1 \subseteq S_2 \subset S_3$$

is equivalent to:

$$S_1 \subseteq S_2 \ \& \ S_2 \subset S_3,$$

except that in the former expression S_2 is evaluated only once.

These basic set operations exhaust those commonly found in mathematics with the exception of the complement, power set, and Cartesian product.

There is no way to form the "true" complement of a set, i.e., the set of members that do not belong to the given set. With respect to a given (finite) universe U, one may write U - S for the complement of S. The extension of ASL to handle infinite complement sets is an interesting area for research, but it is not taken very seriously at this time due to lack of motivation: in the algorithms studied so far there does not seem to be much use for such a concept, and it would be a considerable complication to ASL. The complement with respect to a finite universe turns up from time to time, and for this the expression U - S seems to be adequate.

The power set operation (set of all subsets of a given set) is available in ASL by means of the POW predefined function.

The Cartesian product is not available either as a built-in operator or as a predefined function. One must explicitly form the product set. Of course one could supply a procedure such as:

```
OPERATOR A .X. B
RETURN {(x, y), ∀x ∈ A, ∀y ∈ B}
END
```

In using cross products, one must beware that whereas in mathematics the sets A X (B X C), (A X B) X C, and A X B X C are generally considered

to be equal (it is understood in such contexts that "equal" means "isomorphic"), in ASL the products A X (B X C) and (A X B) X C are distinct. If a cross product operator were introduced into ASL, it would probably be best to have A X B X C denote a set of triples, making it an exception to the left association rule (which is acceptable inasmuch as there already are similar exceptions, such as $a < b < c$).

4.1.8 Relation Operators

The relation operators are:

Range ($\mathcal{R}$)
Domain ($\mathcal{D}$)
Inverse ($\mathcal{L}$)

These are called "relation operators" because they can be applied to binary relations, which are sets of ordered pairs that one may at times think of as "multiple valued functions". The relation operators may not, however, be applied to higher order relations (sets of triples, etc.).

The range of a binary relation is the set formed by taking as members the second component of each pair. The domain is the set of first components, and the inverse is the set with each pair reversed. In ASL:

$$\mathcal{R}\, R = \{y,\ \forall(x, y) \in R\}$$
$$\mathcal{D}\, R = \{x,\ \forall(x, y) \in R\}$$
$$\mathcal{L}\, R = \{(y, x),\ \forall(x, y) \in R\}.$$

It is invalid to apply a relation operator to a set that is not a binary relation. If an occurrence of a relation operator is valid, then the result is always defined. We have $\mathcal{R}\emptyset = \mathcal{D}\emptyset = \mathcal{L}\emptyset = \emptyset$.

$\mathcal{R}$, $\mathcal{D}$, and $\mathcal{L}$ are frequently used, and an ASL implementation should take care to make them reasonably fast operations. For example, one might provide a special implementation for the predicate $x \in \mathcal{D} S$ and the iterator $\forall x \in \mathcal{D} S$ so that $\mathcal{D} S$ need not be explicitly formed. The inverse of a binary relation might conceivably be implemented as a single pointer switching operation.

4.1.9 Array and Vector Operators

There are two operators that may only be applied to arrays and vectors:

Origin setting (A @ b), and
Concatenation (a ¢ b).

The @ operator was discussed in section 3.2.2. To review, if V is a vector expression and k is an integer expression, then V @ k is the vector V with its lowest index biased so that it is k, and with its other indexes biased by the same amount. For example, if

$$V = (a,\ b,\ ,\ c)\ @\ k$$

then V(k) = a, V(k+1) = b, and V(k+3) = c.

The @ operator also applies to arrays. If A is an n-dimensional array, then

$$A\ @\ (k_1,\ k_2,\ \ldots\ ,\ k_n)$$

is the same array with its indexes biased so that the array is $(k_1,\ k_2,\ \ldots\ ,\ k_n)$-origin. Any of the components $k_1,\ k_2,\ \ldots\ ,\ k_n$ may be absent, indicating that the corresponding dimension is not to be changed.

For A @ b to be valid, A must be a vector or array, and b must be an integer vector such that A has a dimension corresponding to each defined component of b. In other words, we must have

$$LI(b) \geqslant 1 \text{ and } HI(b) \leqslant \#LENGTH(A)$$

(see section 3.2.2 for a discussion of LI, HI, and LENGTH). If A @ b is valid then the result is always defined. Regarding the null set, we have $A\ @\ \emptyset = A$ and $\emptyset\ @\ (k_1,\ k_2,\ \ldots\ ,\ k_n) = \emptyset$ for all $n \geqslant 0$.

The @ operator could have been defined so that an expression such as A @ ((1, 2), 3) would be meaningful. This was not done because such an operation would probably have little use.

The concatenation operator ¢ is defined only for vectors. They may be sparse and other than one-origined. If a and b are vectors, then concatenation is defined as:

$$a \text{ ¢ } b = a \cup b@(\text{HI } a + 1).$$

Note that the left operand, a, determines the origin of the result. If a is null, the result is b@1, which is not necessarily equal to b.

4.2 Function Referencing

This section discusses the meaning of expressions involving function references. The related subjects of recursion, sinister calls, dynamic name scopes, etc., are discussed in section 7, Procedures.

The most common type of function reference is the function call, which is denoted by juxtaposition of expressions, e.g., LOG x, LOG (x), (LOG) x, etc. Association is to the right, for example LOG SIN x means LOG (SIN x).

A function may be implemented as either an itemized map or as a procedure. In addition, a function may be invoked in either a value producing (right-hand side) or a value receiving (left-hand side, or sinister) context. We first discuss the referencing of itemized maps, as that is the simplest.

4.2.1 Dexter Itemized Map Referencing

As discussed in section 3.2.1, an itemized map is a set of pairs such that each pair has a distinct first component. If f is an itemized map, then a right-hand side (dexter) occurrence of f(a) signifies a search of f for the pair whose first component is "a" (there can be at most one). The result is the second component of the pair, provided the pair exists, and otherwise the result is undefined. In ASL, provided f is a map:

$$f(a) = (p \in f : p(1) = a)(2).$$

If u is undefined, then f(u) is invalid. However, f(u,) is valid: it is equivalent to f(∅). u(x) is also valid: it is equivalent to ∅(x) (the result is undefined).

4.2.2 Sinister Itemized Map Referencing

A left-hand side (sinister) occurrence of f a, e.g., f(a) = b, signifies a search of f for a pair whose first component is a, the deletion of that pair from f, if it exists, and the addition of the pair (a, b) to f. However, if b is undefined, then the pair beginning with a is deleted but nothing is added to f. In ASL, f(a) = b is equivalent to:

$$f = f - \{p,\ \forall p \in f \mid p(1) = a\} \cup \{(a, b)\}$$

if f is a function and b is defined. If a is undefined then f(a) = b is invalid; however f(a,) = b is the above with a replaced by (a,) or $\emptyset$. If f is undefined then f(a) = b is valid and it sets $f = \{(a, b)\}$.

By "left-hand side" we mean any value receiving context, such as READ f(x), $\forall$f(x) = 1, 2, 3, etc.

4.2.3 Itemized Map Referencing (General)

As an example of itemized map referencing, let

$$f = \{(a, b),\ ((a, b), c),\ ((, b), d),\ ((a, b, c), e)\}.$$

Then:

f(a) is b,

f(a, b) is c,

f(, b) is d,

f(a, b, c) is e,

f(a, b, c, d) is undefined, and

f(, a) is undefined, as is f(a,).

After the assignments:

f(a) = x;

f(a, b) = ;

f(∅) = y;

f is the set:

$\{(a, x),\ ((, b), d),\ ((a, b, c), e),\ (\emptyset, y)\}$.

Functional application for itemized maps could be defined in a somewhat more general way than has been done, by dropping the restriction that an itemized map must consist entirely of pairs, and that the mapping be unique, so that any set is a potential map. This is done in SETL.

This approach is not taken in ASL because it is believed that making the meaning of various constructions as self-evident as possible leads to improved readability. For example, suppose the set:

$$S = \{a,\ (b, c),\ (d, e, f),\ ((d, e), g)\}$$

is to be regarded as a potential map. Then what is the domain of S? It's range? Does S have an inverse? Is S(d, e) equal to f, or g, or $\{f, g\}$, or is it undefined or invalid? When are two functions considered to be equal?

In ASL, an attempt is made to reject generalities if the meaning is not rather clear, especially if the concept has no precedent in conventional mathematics. If the meaning of a basic ASL operation on certain objects is not clear, then chances are that the operation is invalid or the result is intentionally undefined. For the above set S, $\mathcal{R}(S)$, $\mathcal{D}(S)$, $\mathcal{I}(S)$, and S(d, e) are all invalid, because S is not in the domain of these operations. One might consider two functions f and g to be equal if they have the same domain

D and if, for every $x \in D$, $f(x) = g(x)$. With the ASL definition of function, this sense of equality is equivalent to normal set equality, and thus one is not led astray.

It is possible that the restricted definition of "map" will lead to improved efficiency in function evaluation and in storing maps.

Regarding the evaluation of itemized map references, one can make the following general observations:

1. A function reference always involves exactly one argument. However, the argument may be a vector, in its full generality, and thus we can easily create the appearance of references involving a variable number of arguments, or in fact with any arguments omitted: f ∅, f(a), f(a, b), f(, b), and even f[(a, b, c)@-5] all have meaning (for the same function f).
2. An itemized map reference never causes a new value to be assigned to an argument.
3. If v = (a, b), then f(v) and f(a, b) are equivalent.
4. There is generally no relation between f(a), f(a, b), f(a, b, c), etc., unless the function f happens to be defined so that some relation exists.

As will be seen in the next section, with a modification to (2), these remarks continue to hold if f is a procedure. However, in spite of the attempt to unify the concept of function in ASL, a procedure reference does differ from an itemized map reference in several important respects. For example, a procedure may have side effects, and it can alter its arguments in a left-hand side call.

4.2.4 Dexter Procedure Referencing

Parameters are passed by value in a way that is based on vector assignment. Although vector assignment is fully discussed in a subsequent section (5.2.2), this section can probably be followed without first reading that. We first discuss right-hand side procedure calls.

When a function f is referenced by an expression such as f(a, b, c), and the function header statement is something like FUNCTION f(x, y, z), then the compiler in effect inserts the assignment

$$(x, y, z) = (a, b, c)$$

at the point of the call. No similar assignment is inserted at the point of the return, and thus procedure f cannot alter the arguments a, b, and c, even though f may contain assignments to its formal parameters (we will speak of the arguments a, b, and c, even though more formally there is only one: the vector (a, b, c)). Of course f could alter the arguments a, b, and c via external/shared linkage.

The meaning of the vector assignment (x, y, z) = expr, where x, y, and z are variables, is:

```
t = expr
z = t(3)
y = t(2)
x = t(1)
```

where t is a compiler temporary. The reason for introducing the temporary is so that expr will be evaluated only once. The ASL use of temporaries is discussed in sections 4.10.2 and 5.2.3; here we will largely ignore them.

Thus after the assignment (x, y, z) = (a, b, c), we have x = a, y = b, and z = c. Now consider invoking the above procedure f with some arguments omitted.

Call	Parameter assignment	Effect
f(a, b)	(x, y, z) = (a, b)	x = a y = b z = ;
f(, , c)	(x, y, z) = (, , c)	x = ; y = ; z = c

In the first case above we have (x, y, z) = (a, b), which is evaluated as:

```
t = (a, b)
z = t(3)      "z is undefined."
y = t(2)      "y = b"
x = t(1)      "x = a"
```

The parameters are matched as one would probably expect them to be, and the procedure may test for the presence of the third parameter with the existence test:

IF ∃z THEN . . . .

The second case above is similar.

Parameter lists may be constructed with the usual vector operations, remote from the call if desired, and then passed to the routine (although this generally doesn't do much for readability). For example, using the function f above,

```
v = (a, b) ¢ (c, )
. . . f(v) . . .
```

is the same reference to f as f(a, b, c); it causes the assignments x = a, y = b, and z = c (and <u>not</u> x = (a, b, c), y = z = undefined). As has already

been pointed out, if f is an itemized map then the same relationship holds: f(v) and f(a, b, c) produce the same result.

There are two anomalies in parameter matching. One is that to pass a single parameter to a procedure that expects a number of parameters, one must include a comma, e.g., f(a,). This is because (a) = a, but (a,) is a one-vector containing a. The other anomaly has to do with omitting all parameters. One may code this as f(Ø) or f Ø; f( ) is invalid because ( ) is taken to be undefined. The problem is that if the procedure has only one formal parameter, there is no way to omit it; f Ø assigns the null vector to the formal parameter. This behavior can be deduced from the way vector assignments work (see section 5.2.1). Of course in many cases it is possible to code the procedure to interpret Ø as meaning that all parameters are omitted.

A function may be coded to accept an arbitrary number of arguments, with any of them omitted (except for the anomalies above) by having only a single formal parameter:

FUNCTION g(x).

For a reference such as g(a, , c), the assignment x = (a, , c) is done, and the procedure g may then iterate over the components of x to process them all. An itemized map has the same property: it may have values specified for a variable number of parameters with any of them omitted.

The "formal parameter display" portion of a procedure header may be any valid value receiving expression (see section 4.8), as its only function is to become the left side of the assignment statement that the compiler inserts as a part of procedure linkage. This permits some strange displays, e.g.,

FUNCTION f(x(3), , y, y). For this function, the call f(a, b, c, d) causes the assignment (x(3), , y, y) = (a, b, c, d), which is equivalent to x(3) = a; y = c; in the absense of any complicating factors.

Unusual parameter displays such as this would practically never be used. It is pointed out merely to get across how ASL works.

4.2.5 Sinister Procedure Referencing

Procedures may be invoked in a left-hand side, or value receiving, context. Such a procedure call is referred to as a "sinister call". The subject of coding a procedure that may be invoked in a sinister manner is fully discussed in section 7.6; here we discuss the basic meaning of a sinister call, and the mechanics of the linkages, emphasizing the handling of parameters.

FORTRAN and many other languages provide one case of a sinister call in their array assignments, e.g.

A(1) = e

is understood to designate an altering of array A by changing the value of its first component to the value of the expression e, leaving all other components unchanged.

PL/I includes another type of sinister call, for example:

SUBSTR(S, 3, 2) = 'AB';

REAL(C) = 1.0; etc.

Eleven built-in functions have been singled out as permissible to use in a

left-hand side context. These are called "pseudo-variables".

ASL extends this idea so that the statement

f(x) = e

may be meaningful if f is a user-provided procedure, as well as an itemized map or one of a number of built-in functions. Only a small class of procedures may be used in a sinister call, however. For example, a routine "plus(x, y)", which adds x and y, probably could not be used in a sinister context. Precisely which procedures are valid in which mode depends upon how the procedure is coded. Many procedures (in fact, most) are only valid in dexter mode. If a procedure is valid in sinister mode, then it is also valid in dexter mode, but it may not be meaningful in dexter mode. If it is meaningful in both modes it is called "ambidextrous".

Usually the ambidextrous procedures are those that perform, in dexter mode, what one would intuitively think of as a "retrieval" operation, as for example the PL/I SUBSTR. Vaguely speaking, a retrieval operation is one that searches a data structure in some way and retrieves some part of the structure. The most natural meaning of a sinister invocation of the procedure is then to search the structure in the same way and replace what is found (if anything) with the value of the right-hand side expression. There are, however, reasonable ambidextrous procedures that are not retrieval operations (see procedure .LE. in section 7.6).

To fully understand sinister procedure calls, it is necessary to examine procedure linkage in more detail than was used in the previous section.

Procedure linkage may be thought of as involving four global variables:

F, the procedure,

ARG, the argument,

SINISTER, the "sinister flag", and

RESULT, the result in a dexter call, and an input quantity in a sinister call.

Two of these, SINISTER and RESULT, are key words in ASL and they may be directly employed by the problem programmer; their use is discussed in section 7.6.

Function calls expand as illustrated below, ignoring the possibility of side effects (for a more precise analysis, see section 5.2.3).

Dexter call, e = f(a, b, c)

```
F = f
ARG = (a, b, c)
SINISTER = FALSE
IF SET(F) THEN RESULT = (p ε f : p(1) = ARG)(2)
          ELSE call F   "Sets RESULT."
e = RESULT
```

Sinister call, f(a, b, c) = e

```
IF ∃f THEN F = f ELSE F = ∅
ARG = (a, b, c)
SINISTER = TRUE
RESULT = e
IF SET(F) THEN f = F - {p, p ε F | p(1) = ARG} ∪ {(ARG, RESULT)}
          ELSE DO call F; (a, b, c) = ARG END
```

The statement "call F" denotes a "branch and link", or "return jump"; "call" is not an ASL statement.

The sinister procedure call differs from its dexter counterpart in that

(1) RESULT is an input quantity to the called procedure, and (2) the sinister call includes the step (a, b, c) = ARG, which is executed after return. Hence if f is a procedure that alters its formal parameter, then the change is communicated to the caller in a manner similar to a conventional call by value with deferred argument return.

Note that without knowledge of what f is (set or procedure), one cannot tell whether the effect of f(x) = e is to alter f or x (or both). For example, the array assignment A(1) = e alters A, but SUBSTR(S, 1, 2) = 'AB' alters S. In ASL, the determination of the type of f must in general be done at run time (if f is a variable, then it may at times have sets and at other times have procedures for values).

The expansions given show the type test as in-line code. For the dexter call, this may trivially be moved to a system "apply" routine, so that the statement e = f(a, b, c) would become e = APPLYDEX(f, (a, b, c)). For the sinister call, the expansion of f(a, b, c) = e would be t = APPLYSIN(f, (a, b, c), e); IF SET(f) THEN f = t ELSE (a, b, c) = t. In some cases the function is altered and in some cases the argument is altered. We shall assume for simplicity that the type test is done in-line (at the point of the call).

For the called procedure, the only difference between dexter and sinister calls is in its handling of the RETURN statement. For a procedure to be capable of sinister execution, its return expression must be a valid value receiving expression, e.g. x, x(i), (x, y, z), or it must be omitted. The linkage on the called procedure side is as follows. The procedure:

```
FUNCTION f(x, y, z)
  ...
RETURN r
END f
```

compiles as follows:

```
f:  (x, y, z) = ARG
      ...
    IF SINISTER THEN DO
       r = RESULT
       ARG = (x, y, z)
       END
    ELSE RESULT = r
    RETURN
    END f
```

If the return expression is present but is not a value receiving expression, an error exit is invoked if it is attempted to execute the return in a sinister call.

The following example should help to tie this together. Procedure "nextnbc" retrieves or replaces the next non-blank character in a character string, based on an internal counter n.

```
FUNCTION nextnbc(s)
DECLARE n SHARED
(n < ∀i ≤ #s) IF  s(i) ≠ ♭ THEN [n = i; RETURN s(i)]
RETURN "Undefined result."
END nextnbc
```

It is fairly obvious what this routine does for a dexter call such as

```
C = nextnbc(S).
```

For a sinister call, such as

```
nextnbc(S) = 'a',
```

the expansion is given below, where for simplicity the tests of the type of the function and the sinister flag have been omitted, as have a few other irrelevant details.

```
                ARG = S
                RESULT = 'a'
                call nextnbc
nextnbc:        s = ARG
                (n <∀i≤#s) IF s(i) ≠ b THEN [
                    n = i
                    s(i) = RESULT
                    ARG = s
                    RETURN]
                ARG = s
                RETURN
                S = ARG
```

It may be seen that this does what was intended: it scans S for a non-blank character and replaces it with 'a'. If no such character is found, no action is taken.

4.2.6 Sinister Composition of Functions

The meaning of dexter composition of functions is obvious: by e = f(g(x)) we mean t = g(x); e = f(t); where t is a compiler temporary. Furthermore, this trivial expansion can be derived from the dexter call expansion given in the previous section (4.2.5). Expanding from the outside in, we have:

```
F = f
ARG = g(x)
SINISTER = FALSE
IF SET(F) THEN RESULT = (p ε f : p(1) = ARG)(2)
          ELSE call F
e = RESULT.
```

It is easily seen that t = g(x); e = f(t); gives an identical expansion, by expanding the call to f: we have t = g(x); F = f; ARG = t; SINISTER = FALSE; IF SET(F) THEN...ELSE call F; e = RESULT. Eliminating the temporary (t) reduces this to the above.

There is more to the sinister composition of functions, however, and the remainder of this section is devoted to it.

There are basically two types of expressions to consider:

$$f(g(x)) = e, \text{ and}$$

$$(f(g))(x) = e.$$

Each of these has four subcases, corresponding to whether f and g are procedures or itemized maps. We shall arrive at expansions that clearly show the meaning of the above assignments by a process of derivation from the basic sinister call expansion that has been given (which ignores side effects). Section 5.2.3 includes an alternate and more precise derivation of the results that follow.

First consider what is perhaps the most interesting case: f(g(x)) = e, with f and g both procedures. Expanding this from the outside in, we have

```
F = f
ARG = g(x)
SINISTER = TRUE
RESULT = e
call F
g(x) = ARG
```

Note that the inner function, g, must be ambidextrous.

The next level of expansion, for the sinister call g(x) = ARG, requires a temporary for ARG, but that is straightforward and will not be shown.

The above expansion of f(g(x)) = e is exactly equivalent to the following, where t is a temporary:

```
t = g(x)
f(t) = e
g(x) = t.
```

To see this, simply expand the sinister call f(t) = e to: F = f; ARG = t; SINISTER = TRUE; RESULT = e; call F; t = ARG;. With this expansion replacing f(t) = e in the above, it is obvious that the assignments involving t may be combined so that t is eliminated, which makes the code sequences identical.

The above three-assignment expansion of f(g(x)) = e has a symmetry that makes it easy to remember (the last line is the reverse of the first), but it should be stressed that it may not be correct if side effects are involved. This is because the basic expansions of function references given at the beginning of the previous section are not quite right (the situation will be clarified in section 5.2.3).

The last step in the expansion of f(g(x)) = e (i.e., the last step in t = g(x); f(t) = e; g(x) = t) is superfluous if procedure f does not modify t. If this step were always omitted, one would have the expansion that one would expect FORTRAN or PL/I to use, if these languages permitted left-hand side procedure calls (it is presumed that these languages would treat it similarly to the way they treat the array assignment A(B(I)) = E). The following simple example shows the utility of the last step, and also illustrates a simple but common use of sinister calls.

Suppose it is desired to name the components of vectors, for mnemonic purposes, in a manner imitative of the PL/I structure declaration. In such an application one might have a group of procedures such as the following.

```
FUNCTION name(v); RETURN v(1); END;
   FUNCTION first(v); RETURN v(1); END;
   FUNCTION last(v); RETURN v(HI v); END;
FUNCTION street(v); RETURN v(2); END;
FUNCTION city(v); RETURN v(3); END;
```

If initially:

```
V = (('John', 'Doe'), '251 Mercer Street', 'New York'),
```

then after:

```
last name V = 'Smith'
```

we have:

```
V = (('John', 'Smith'), '251 Mercer Street', 'New York').
```

The (abbreviated) execution is as follows:

```
t = name V          Sets t = ('John', 'Doe').
last t = 'Smith'    Sets t = ('John', 'Smith').
name V = t          Sets V = (('John', 'Smith'), '251 Mercer Street',
                              'New York').
```

The conventional method of expanding "last name V = 'Smith' ", if it were permitted at all, would amount to nothing since the last assignment would be omitted.

Thus far we have considered the expansion of f(g(x)) = e only if f and g are both procedures. To consider a different case, suppose we have A(B(i)) = e, with A and B itemized maps (possibly vectors). Then, after expanding this as above, we have:

F = A
ARG = B(i)
SINISTER = TRUE
RESULT = e
A = F - {p, p ε F | p(1) = ARG} $\cup$ {(ARG, RESULT)}

This is easily seen to be equivalent to:

t = B(i)
A(t) = e

The above expansion in this case is exact, as no side effects are involved in itemized map referencing. The cases of mixed itemized maps and procedures work out similarly.

If a function g is free of side effects and has the property that the sequence t = g(x), g(x) = t is a no-operation, then g will be called a pure retrieval function. Itemized maps are pure retrieval functions. The term "retrieval function" will be used loosely; for example the procedure "nextnbc" above might be called a retrieval function, but it is not a pure retrieval function because it has the side effect of updating an internal counter.

Functions with side effects should be used in sinister composition only with great care. Consider f(nextnbc(s)) = e, where f is a procedure. It doesn't work as would probably be intended:

t = nextnbc(s)
f(t) = e
nextnbc(s) = t

because the internal counter would be stepped twice.

Now consider composition grouped on the left, e.g.

$$[f(x)](y) = e.$$

For the moment assume that f and f(x) may be either procedures or itemized maps (we treat all four cases simultaneously). Mechanically expanding the above, we have (in the absense of side effects):

```
F = f(x)
ARG = y
SINISTER = TRUE
RESULT = e
IF SET(F) THEN f(x) = F - {p ε F | p(1) = ARG}....
                                  ∪{(ARG, RESULT)}
ELSE DO call F; y = ARG END
```

Here f must be ambidextrous if f(x) is an itemized map. In this case, the above is exactly equivalent to:

```
t = f(x)
t(y) = e
f(x) = t,
```

which may be seen by expanding the second assignment, t(y) = e. This is similar to the "three-assignment" rule previously given for f(g(x)) = e.

To illustrate with a concrete example, suppose V is a vector of vectors. Then the assignment:

$$[V(i)](j) = e$$

is equivalent to:

```
t = V(i)
t(j) = e
V(i) = t.
```

This obviously has the effect of changing component j in vector i of V; the third assignment is essential to accomplish this.

For the composition [f(x)](y) = e, if f(x) is a procedure, then the third assignment of the expansion is omitted. However, it could be included (as a no-operation) if f is a pure retrieval function and f(x) does not alter y.

4.2.6.1 Sinister Composition, General Remarks

It should be clear how multiple arguments are handled. Recall that every procedure reference really involves exactly one argument, although the argument may be a vector. Hence the meaning of:

f[g(x), h(y)] = e

if f is a procedure, is:

```
t = [g(x), h(y)]
f(t) = e
[g(x), h(y)] = t
```

in the absense of side effects. This may alternatively be written:

```
t1 = g(x)
t2 = h(y)
f(t1, t2) = e
h(y) = t2
g(x) = t1.
```

The meaning of more deeply nested composition of functions should also be clear. For example, the expansion of:

f[[A(i)](j)] = e,

if f is a procedure and A(i) is an itemized map, is:

$$t_1 = A(i)$$
$$t_2 = t_1(j)$$
$$f(t_2) = e$$
$$t_1(j) = t_2$$
$$A(i) = t_1.$$

This is easily derived by noting that it is of the form $f(g(j)) = e$, with $g = A(i)$. Hence we have:

$$t_2 = [A(i)](j)$$
$$f(t_2) = e$$
$$[A(i)](j) = t_2.$$

Expanding the last line, the above becomes:

$$t_2 = [A(i)](j)$$
$$f(t_2) = e$$
$$t_1 = A(i)$$
$$t_1(j) = t_2$$
$$A(i) = t_1.$$

Provided no side effects are involved (so that neither i nor A(i) is changed at line 2), the line $t_1 = A(i)$ may be moved to the top, and the next line may be changed to $t_2 = t_1(j)$. The expansion is then identical to the one originally given.

The compiler would probably generate code closer to the last sequence than the first. It would then be an optimizing task to factor out the dexter evaluation of A(i). The only reason for giving the original expansion is that it has a symmetry that makes it easy to write down immediately, and as a practical matter it is usually correct.

Sinister calls apply to some operators as well as to functions. That is, if .OP. is a suitably defined operator, one can write:

$$x \text{ .OP. } y = e.$$

The rules given for sinister call expansion frequently result in syntactically invalid assignments. When this occurs, the invalid assignments are omitted. For example, the expansion of f(1) = e is:

```
F = f
ARG = 1
SINISTER = TRUE
RESULT = e
IF SET(F) THEN ...
          ELSE DO call F; [1 = ARG] END
```

where the assignment in brackets is omitted.

So far we have glossed over the fact that f is allowed to be undefined in the assignment f(x) = e, in which case it is initialized to the null set. To see why this is so, consider again the composition of itemized map assignments:

$$[A(i)](j) = e.$$

This has the expansion:

```
t = A(i)
t(j) = e
A(i) = t.
```

Suppose the program is building up structure A by repeatedly executing [A(i)](j) = e for various values of i and j. Then presumably A would be initialized to the null set. This makes t undefined after t = A(i), and for the second assignment to work we want to treat t as the null set in this case.

We could require the programmer to initialize A to a map of the proper domain onto $\emptyset$, or to supply an initialization statement such as "IF $\neg\exists$A(i) THEN A(i) = $\emptyset$" before the assignment [A(i)](j) = e, but this is a burden

that's best left to the compiler.

This means that a program may build up a map component by component, by executing statements of the form M(i) = e, without initializing M to the null set (assuming M is a local variable and hence is initially undefined).

Carrying this analysis one step further reveals that it is necessary to allow the expression f(x) with f undefined in dexter context also. For example, the expansion of:

[[M(i)](j)](k) = e

is:

```
t1 = M(i)
t2 = t1(j)
t2(k) = e
t1(j) = t2
M(i) = t1.
```

If M is initially null, then the first assignment leaves t1 undefined. The second assignment is nevertheless permitted, and it leaves t2 undefined. The third assignment sets t2 = {(k, e)}, the fourth sets t1 = {(j, {(k, e)})}, and the last sets M = {(i, {(j, {(k, e)})})}, which is the desired result.

Hence in either dexter or sinister mode, f in the expression f(x) may be undefined, and it is then treated as the null set.

4.2.7 Expression-statements

An expression written as a statement, e.g.,

```
e;
```

is an abbreviation for the assignment:

```
e = TRUE;.
```

For the statement "e" to be valid, e must be a value receiving expression and any procedures that it causes to be invoked must be capable of execution in the appropriate mode (dexter or sinister). Normally, e would be an expression that would have a Boolean value if executed in dexter mode.

This minor syntactic device extends ASL in one significant way and in a few ways that are useful, although not of great significance. For an example of the latter, if x is a variable, then simply writing "x;" sets it to TRUE. If we wish to set a switch to indicate that some process is completed, we simply write:

```
done;
```

rather than "done = TRUE;".

More significantly, if "sub" is a function procedure capable of sinister execution, then we may write

```
sub(x);
```

to cause sub to be invoked in a way that allows x to be both an input and an output quantity. Hence for all practical purposes ASL has subroutines, although strictly speaking there are only function- and operator-type procedures. If sub is intended to be a "subroutine", that is, a function that

is only intended to be invoked in sinister mode, then it may of course ignore the value TRUE that is passed to it via RESULT. It would probably return using simply RETURN; without a return expression.

Similarly, if .OP. is a user-defined operator-procedure capable of sinister execution, then we may write

x .OP. y;

for the effect of x .OP. y = TRUE.

A few built-in operators may be used as expression-statements. These are:

not (¬)
and (&)
nor (↓)
implies (⟹)
membership test and its negation (∈, ∉)

Although the Boolean operators normally apply to maps (vectors, etc.), when used in sinister mode they apply only to atomic Boolean operands. In addition, and, nor, and implies may only be used when the right-hand side is TRUE.

If e is an expression then the statement ¬e; is equivalent to e = FALSE. To reset a switch we may write ¬done; rather than done = FALSE. The way this is (or at least could be) implemented is that the "not" routine is capable of sinister execution, and it sets its argument to ¬RESULT. Hence when we write ¬e;, the compiler translates this to ¬e = TRUE, which becomes (in essence):

```
ARG = e
SINISTER = TRUE
RESULT = TRUE
call "not routine"     "Sets ARG = ¬RESULT = FALSE."
e = ARG
```

All of the above sinister operators could be implemented similarly.

The "and" routine accepts only a TRUE value for RESULT. That is, we can write e_1 & e_2; or e_1 & e_2 = TRUE; but not e_1 & e_2 = FALSE. The routine sets both of its parameters TRUE.

Nor is similar to and, except that it sets both parameters FALSE.

The statement $e_1 \Rightarrow e_2$; causes the "imply" routine to test its first parameter (e_1). If it is TRUE, the second parameter is set to TRUE. If e_1 is FALSE, no action is taken (however, e_2 is evaluated in any event, since it is an argument of a procedure call). Hence $e_1 \Rightarrow e_2$ is equivalent to the statement IF e_1 THEN e_2, except for the evaluation of e_2.

The statement $x \in S$; (or $x \in S$ = TRUE;) causes x to be made a member of S. It is equivalent to $S = S \cup \{x\}$, but may be more efficient in an unsophisticated implementation (which would build the set $\{x\}$ and perform a set union operation for the latter statement).

Similarly, $x \notin S$, or $x \notin S$ = TRUE, or $x \in S$ = FALSE, causes x to be removed from S; they are equivalent to $S = S - \{x\}$.

4.2.8 Functions as Data

As discussed in section 3.1.5, a function procedure name may be treated as data and assigned to variables, put into sets, etc., much as an itemized map may be treated. There are important differences, however, between procedure maps and itemized maps.

One difference is that a procedure name is considered to be a constant; it cannot appear in a value receiving context.

Another difference is that if M is an itemized map, then the assignment x = M causes M to be copied, so that x is the current mapping of M. On the other hand, if P is a procedure name, the assignment x = P does not involve generating a copy of P (together with all objects that might affect its operation via shared/external linkages). Instead, future changes to P will be reflected as changes done by the mapping x.

4.3 Set and Vector Formers

There are non-iterative and iterative set former expressions. The non-iterative set former is written:

$$\{e_1, e_2, \ldots, e_n\}$$

where the e_i are expressions. The value of the set former expression is the set containing the current values of the e_i. The set will have fewer than n members if some of the expressions have the same values. The values must all be defined, or the set expression is invalid.

The general form of the iterative set former is:

$$\{e, \text{iteration}_1, \text{iteration}_2, \ldots, \text{iteration}_n\}.$$

Here e is an expression generally involving n iteration variables, and the iteration_i are iteration terms. An iteration term is explained in its full generality in section 6.5 (the same forms are used for iterating statement blocks and in set formers). In this section we will use only simple iteration terms such as $\forall x \in S \mid C(x)$ (iteration over a set S) and $m \leqslant \forall i \leqslant n \mid C(i)$ (iteration for i = m, m+1, ..., n).

The above iterative set former forms the same set as S in the following program section:

```
S = ∅
iteration_1
   ...
      iteration_n
         e ε S
         end_n
      ...
   end_1
```

As examples, here is a set of three integers:

$$\{1,\ 2,\ 3\},$$

and here is the set of primes from 2 to 100:

$$\{p,\ 2 \leqslant \forall p \leqslant 100 \mid \neg(2 \leqslant \exists m < p : \mathrm{REM}(p, m) = 0)\},$$

and here is a certain subset of the cross product of two sets R and S, as determined by C:

$$\{(x,\ y),\ \forall x \in R,\ \forall y \in S \mid C(x, y)\}.$$

The conditional expression and its associated stroke symbol may be omitted if it is simply TRUE; for example:

$$\{x**2,\ \forall x \in S\}$$

is a set of squares obtained from S. The expression e may be omitted if it is a vector obtained from the value receiving portions of each iteration term, taken in order. For example:

$$\{\forall x \in S \mid C(x)\} = \{x,\ \forall x \in S \mid C(x)\}$$

$$\{1 \leqslant \forall i \leqslant n,\ \forall x \in S\} = \{(i, x),\ 1 \leqslant \forall i \leqslant n,\ \forall x \in S\}$$

$$\{\forall f(x) \in S\} = \{f(x),\ \forall f(x) \in S\}$$

The last example illustrates that the iterand may be any value receiving expression.

The vector former is the same as the set former except that instead of braces, either parentheses or brackets are used. The non-iterative vector former has been sufficiently discussed in section 3.2.2. As an example of an iterative vector former, we have:

$$[p,\ 2 \leqslant \forall p \leqslant 100 \mid \neg(2 \leqslant \exists m < p \mid \mathrm{REM}(p,\ m) = 0)]$$

which is the vector (2, 3, 5, 7, 11, ..., 97). That is, the vector is built from left to right as the iteration proceeds.

A vector formed from the iterative vector former may have undefined (missing) components, for example $(f(x), \forall x \in S)$ if $f(x)$ is undefined for some $x \in S$.

4.4 Search Expressions and Quantifier Predicates

A search expression causes a search of a list of items, a range of integers, or a set until some condition is met. The value of the expression is the first item found for which the condition is met, or is undefined if the search is exhausted. Usually the condition expresses some property the item is to have. For example, suppose S is a set of vectors. Then the value of the expression

$$v \in S : v(2) = 0$$

is the first encountered vector in S whose second component is zero, or is undefined if S does not contain such a vector. Similarly, the value of the expression

$$x > 0 : P(x) = 0$$

is the first positive integral root of P, if there is one. Evaluation of this expression does not terminate if there is no root.

A quantifier predicate is either a universal predicate or an existential predicate. A universal predicate is an assertion that all items in some range (a list, range of integers, or set) satisfy some condition. For example,

the value of

$$\forall x \in S : x > 0$$

is TRUE if all members of S are positive (numbers) or if S is null, and is FALSE otherwise.

Similarly, an existential predicate is the assertion that at least one item in some range satisfies some condition, e.g.

$$\exists x \in S : x > 0.$$

Search expressions and quantifier predicates all involve iteration, and the mechanics of the iteration are the same as in the case of statement iteration, which is discussed in section 6.5. The reader is referred to that section for much of the detail, but this section can probably be followed without first reading that.

4.4.1 Search Expressions

The general form of a search expression is:

STARTING $stmt_1$ range-specification WHILE C_1 DOING $stmt_2$: C_2

The range specification and the ": C_2" are required, but the STARTING, WHILE, and DOING clauses are optional. Refer to section 6.5 for a discussion of the STARTING and DOING clauses.

The range specification is a list of range-items, each of which is in one of the following three forms:

1. (Itemized) x = list | C
2. (Counting) $m \leqslant i \leqslant n \mid C$, or $i > m \mid C$, etc.
3. (Set) $x \in S$ BY next | C

The syntax and meaning of the range-items are very similar to those of the for-items discussed in section 6.5.2, and the reader is referred to that section for details. Here we merely summarize. The x and i in the above are the iterands, and it is the final value of the iterand (if any) that is the value of the whole search expression. The iterand may be any value receiving expression.

The "list" above is a sequence of expressions separated by commas. It specifies the range of the iteration by explicitly listing each value the iterand is to be assigned.

C denotes a predicate expression. Only those values in the range that satisfy C will be used in the search expression. For example, the range specification

$$x = 2,\ 1,\ 3,\ 4 \mid \mathrm{MOD}(x, 2) = 1$$

specifies an iteration with x = 1, then 3 (2 and 4 being skipped). The stroke symbol and C may be omitted, in which case |TRUE is assumed.

The counting form specifies that i is to take on integer values from m to n, for the form $m \leqslant i \leqslant n$. The form $i > m$ signifies t = FLOOR(m) + 1, i = t, t+1, ... There are twelve varieties of such expressions (six of them count backwards, e.g. $m > i \geqslant n$).

S is a set expression, and "next" is a procedure or itemized map that determines the order of iterating over S. The "BY next" phrase may be (and usually is) omitted, in which case the predefined procedure variable NEXT is used. This variable is initialized, at procedure entry, to the

"standard" set iterator. For general sets, the standard order is arbitrary (not specified and not necessarily the same from one case to another for the same set). If the set is an array, the order of iteration is lexicographic (last subscript varying most rapidly).

The WHILE C clause in a search expression causes searching to terminate if C becomes false during the search. That is, in the search expression

$$x \varepsilon S \mid C_0 \text{ WHILE } C_1 : C_2$$

C_1 may cause termination of the search before all members of S have been examined, whereas C_0 merely causes certain members to be skipped over. The expression has no value if evaluation is terminated early.

The evaluation of the expression

STARTING $stmt_1$ $x \varepsilon$ S BY next $\mid C_0$ WHILE C_1 DOING $stmt_2 : C_2$

is equivalent to the following, where r is the result:

```
t = variables
r = ;
(STARTING stmt1 ∀x ε S BY next | C0 WHILE C1 DOING stmt2) DO
   IF C2 THEN (r = x; QUIT) END
variables = t
```

Here t is a temporary and "variables" is a vector containing the iterand x, and the variables that are referenced in $stmt_1$ and $stmt_2$.

Expanding this statement iteration as described in section 6.5 gives:

```
      t1 = variables
      r = ;
      stmt1
      t2 = ;
back: t2 = next(t, S)
      IF ∃t2 THEN DO
         x = t2
         IF C0 THEN
            IF C1 THEN IF C2 THEN (r = x; GO out)
                               stmt2
                       ELSE GO out
         t2 = x
         GO back
         END
out:  variables = t
```

Search expressions using the other types of range specifications have similar expansions. For example,

$$m \leqslant i \leqslant n : C$$

is equivalent to

```
t = i
r = ;
(m ⩽∀i ⩽n) IF C THEN (r = i; QUIT)
i = t
```

and this may be further expanded with reference to section 6.5.

In the absense of side effects, the search expression $x \varepsilon S : C$ is equivalent to $\exists\{x, \forall x \varepsilon S \mid C\}$, but the former is of course more to the point and faster running (for an unsophisticated compiler).

The iterand of a search expression is <u>not</u> a dummy, or bound, variable. However, it is saved and restored before and after evaluating the search expression, so that evaluation does not (normally) have the side effect of changing a variable. The reason for this is so that the iterand may be any value receiving expression. For example, suppose we have a vector V of

numbers and we wish to find a number in a set S which, when used in place of V(1), causes the magnitude of V to exceed 5. Then we may write

$$y = V(1) \varepsilon S : |V| > 5.$$

The variable y is assigned the appropriate number from S, and V is not altered.

There are other temporaries introduced for the evaluation of search expressions, coincident with the technique discussed in section 4.10.2, but we point out here the one for the iterand to show how ASL performs an assignment locally but without (normally) its having a global effect.

The temporaries also prevent the statements in the STARTING and DOING clauses from having any global effect (in the absense of side effects). They are "bound statements".

As a practical matter the STARTING, WHILE, and DOING clauses, and the conditional expression in the range specification, are seldom used. The latter is because the expression

$$x \varepsilon S \mid C_0 : C_1$$

is equivalent to

$$x \varepsilon S : C_0 \ \& \ C_1,$$

as can be readily seen from the expansion given. Similar remarks apply to the universal and existential predicates.

As an example of the use of the STARTING and DOING clauses in a search expression, here is a calculation of the 100^{th} prime:

```
y = STARTING k = 1; p≥ 2| [2 ≤∀i <p : REM(p, i) ≠ 0] DOING k = k + 1....
                                                    : k = 99
```

Search expressions and quantifier predicates involving the STARTING and DOING clauses are usually so intricate (as the above example illustrates) that they will seldom be used. The main reason they are permitted is because they are of value in statement iteration, and we wish to have a unified treatment of iteration in ASL to keep the language simple in the sense of having a simple grammar, being easy to remember various constructions, etc.

Here is an example of the value of this type of simplicity. In section 6.5, the pseudo-function ITERATION is discussed in connection with statement iteration. The question arises: can ITERATION be used in a search expression, universal predicate, existential predicate, set former, or vector former? The answer is obviously "yes" to all of these.

4.4.2 Universal Predicates

The general form of the universal predicate expression is:

STARTING $stmt_1$ for-specification WHILE C_1 DOING $stmt_2$: C_2

The for-specification is the same as a range specification except that a ∀ symbol precedes the iterand. The for-specification of the universal predicate is the same as the for-specification of a statement iteration header (section 6.5).

The details of the meaning of the universal predicate parallel those of the search expression. Here we summarize by giving the expansion of:

$$\forall x \in S \text{ BY next} \mid C_0 \text{ WHILE } C_1 : C_2,$$

which is shown below, where r is the result.

```
t = x
r = TRUE
(∀x ε S BY next | C0 WHILE C1) DO
   IF ¬C2 THEN (r = FALSE; QUIT) END
x = t
```

Note that the expression:

$$\forall x \in S \mid C_0 : C_2$$

is equivalent to:

$$\forall x \in S : \neg C_0 \vee C_2.$$

4.4.3 Existential Predicates

The general form of the existential predicate expression is:

STARTING $stmt_1$ exists-specification WHILE C_1 DOING $stmt_2 : C_2$

The exists-specification and the "$: C_2$" are required, but the other clauses are optional.

The exists-specification is the same as a range specification except that an $\exists$ symbol precedes the iterand.

The details of the meaning of the existential predicate parallel those of the search expression and universal predicate. Here we summarize by giving the expansion of:

$$\exists x \in S \text{ BY next} \mid C_0 \text{ WHILE } C_1 : C_2$$

which is shown below, where r is the result.

```
t = x
r = FALSE
(∀x ε S | C0 BY next WHILE C1) DO
   IF C2 THEN (r = TRUE; QUIT) END
x = t
```

The expression:

$$\exists x \varepsilon\ S \mid C_0 : C_2$$

is equivalent to:

$$\exists x \varepsilon\ S : C_0\ \&\ C_2$$

and to:

$$\neg(\forall x \varepsilon\ S \mid C_0 : \neg C_2)$$

and to:

$$\neg(\forall x \varepsilon\ S : \neg C_0 \vee \neg C_2).$$

Note that if S is null, then $\forall x \varepsilon\ S : C$ is TRUE, and $\exists x \varepsilon\ S : C$ is FALSE. This preserves the identities (in the absense of side effects) $\exists x \varepsilon\ S : C = \neg(\forall x \varepsilon\ S : \neg C)$ and $\forall x \varepsilon\ S : C = \neg(\exists x \varepsilon\ S : \neg C)$.

Expressions that violate the normal mathematician's rules regarding free and bound variables, such as $\exists x \varepsilon\ A : (\forall x \varepsilon\ B : C(x))$, and $\{e(x, y), \forall x \varepsilon\ A(x, y), \forall y \varepsilon\ B(x, y)\}$ are allowed and are given the meaning of the appropriate expansion. This is because it would be difficult to disallow them in view of the fact that the iterands may be arbitrary value receiving expressions. For example, if v is a vector, $\{v(i)+v(j), \forall v(i) \varepsilon\ S1, \forall v(j) \varepsilon\ S2\}$ makes sense if $i \neq j$, but not if $i = j$. In ASL we accept it in either case.

4.5 Ellipses

The ellipsis is used to specify simple sequences. There are two general forms:

e1, e2, . . . , en

e1, e2, . . .

where e1, e2, and en are expressions. The three periods are a single token and may not contain blanks. There may, however, be blanks between the ellipsis and the following comma in the first form.

The compiler parses e1, e2, and en, and then determines the lowest single node for which e1 and e2 differ (syntactically), and for which e1 and e2 are the same everywhere except at this node and its descendants. If we denote by u and v the subexpressions in e1 and e2 that correspond to this node, then the sequence is of the form e(u), e(v), . . . , en (or e(u), e(v), . . .). The expressions u and v must be numeric, and the compiler then generates the assignment d = v - u, where d is a compiler temporary. In general, d can only be evaluated at run time, but in practice it often can be determined at compilation time.

The compiler then generates code to fill in missing terms by changing the sequence to

e(u), e(u+d), e(u+2d), . . . , e(u+nd).

The ending point is arrived at as follows. First expression en is examined. If it is of the form e(w), then n is the largest integer such that u+nd $\leqslant$ w if

$d \geqslant 0$, and n is the largest integer such that $u+nd \geqslant w$ if $d < 0$. If en is not of the form e(w), then n is the largest integer such that $e(u+nd) \leqslant en$ if the sequence is increasing (i.e., if $e(v) \geqslant e(u)$), and n is the largest integer such that $e(u+nd) \geqslant en$ if the sequence is decreasing. There are other differences in these two forms, regarding what happens if the iterand alters the value of the stepping variable. This it can do if en is not of the form e(w), and hence e is numeric. A detailed expansion is given in section 6.5.2.1 Itemized Iteration.

As suggested above, the expressions e1, e2, and en need not be numeric if either en is absent, or if the three are of the form e(u), e(v), and e(w).

The ellipsis can be used in the set former, vector former, search expression, universal predicate, existential predicate, and an iteration in any context. Below are some examples.

1. {1, 2, . . . , n}
2. (-8, -6, . . . , 8)
3. i = a, b, . . . , #s : s(i) = 'a'
4. ∀y = x**1, x**2, . . . , x**n : f(y) > 0
5. ∃S = {n, $1 \leqslant \forall n \leqslant 1$}, {n, $1 \leqslant \forall n \leqslant 2$}, . . . , {n, $1 \leqslant \forall n \leqslant m$} : c(n)
6. (∀x = a, a+dx, . . . , b) S = S + f(x)*dx
7. ∀y = sin(x), sin(x+1.0E-3*PI), . . . , sin(x+2*PI) DO . . . END

The first example forms the set of integers from 1 to n. The same set can be written {i, ∀i = 1, 2, . . . , n}, which employs the ellipsis in a set former in a different sense (an iteration).

In the second example, the difference d would for most ASL implementations probably be 6 - 8 = -2, because the minus signs would be treated as operators. However, an implementation could compile the minus signs in as part of the numbers, in which case d = (-6) - (-8) = +2. It makes no difference, as in the former case the general term is -(8 + (-2)d) and in the latter (-8 + 2d), which is equivalent. Hence either type of compilation is acceptable.

The third example is a search expression (not an assignment statement). The expression searches certain characters of s for an 'a', and the value of the expression is the index of the first 'a' encountered, or undefined if there is none. The characters tested are those of index a, b, 2b-a, 3b-2a, etc.

The remaining examples should be clear.

In matching subexpressions, the compiler is really trying to "guess" what is intended by the ellipsis expression. There is room for interpretation. For example, the expression m, m+1, . . . should, in a sense, be written m+0, m+1, ... This latter form clearly shows the difference d to be 1-0. In the former expression the difference d is calculated as (m+1)-m. Fortunately it makes no difference in this frequently occurring case. But suppose example 4 above had been coded x, x**2, . . . , x**n. Then the difference $d = x^2-x$, and the sequence is x, x^2, $2x^2-x$, . . . , which is not what was

intended.

The exact manner of comparing e1, e2, and en in the ellipsis expression is not specified in ASL. One implementation may ignore redundant parentheses, the difference between parentheses and brackets, prefix plus signs, and the order of terms in a commutative operation, and it may evaluate subexpressions involving self-defining values at compile time, whereas another implementation may not. This is a weak point in ASL, as the implementations will differ in their execution of valid programs.

As an example of a pitfall, consider the sequence 1/2, (1/3), ... A compiler that ignores the redundant parentheses will treat this as 1/2, 1/3, 1/4, 1/5, ..., which is probably what is intended. However, a compiler that is sensitive to the parentheses will treat it as 1/2, 1/3, 1/2 + 2(1/3-1/2), 1/2 + 3(1/3-1/2), ... As another example, consider the sequence (x+1)**1, (1+x)**2, ... Here a sophisticated compiler, which recognizes x+1 and 1+x as the same, would supply the third term as (x+1)**3. An unsophisticated compiler would not.

What we have here is a problem in pattern matching. A "good" ASL compiler would not only eliminate redundant parentheses and prefix plus signs, but it would also be based on an unordered tree comparison, and would evaluate subexpressions at compile time when possible. Furthermore, possibly it could supply redundant subexpressions such as **1, *1, +0, etc., in order to lower the node at which different subexpressions are recognized, which is probably a desirable principle.

It is probably premature to specify elaborate pattern matching rules for ASL. Instead, we accept the fact that implementations will differ in this respect, taking comfort in the fact that the ambiguous cases probably won't arise very often, and they probably won't arise at all with "good" programmers. That is, a "good" programmer would not insert redundant parentheses in one term of an ellipsis expression and not in another, or write parts of the terms in different orders, because such constructions confuse the human reader as well as the compiler. The only disturbing thing is that he must, to be safe, supply such things as the **1 of example 4. Of course he shouldn't have to.

If the expressions involve only addition, subtraction, and multiplication, then it does not matter where on the tree they are considered to be different; they could always be regarded as differing at the top node. Consider, for example, a*(b+c-d*e), a*(b+c-(d+1)*e), ... The general term is a*(b+c-(d+n)*e) if considered to be different at nodes d and (d+1), and the general term is a*(b+c-d*e) + n*(-a*e) if considered to be different at the top node, and the two are equivalent.

Note that the capabilities for a "good" interpretation of ellipsis expressions are also desirable to do a good job of common expression elimination. For example, for both purposes the expressions x+2 and 1+x+(--1) should be recognized as equivalent.

4.6 Cross Sections

ASL includes a construction similar to the array cross sections of PL/I and Algol 68, but generalized to apply to any itemized map.

A cross section of a map f is denoted by

$$f(x1, \ldots, *, \ldots, xn)$$

with at least one asterisk present.

For a three-dimensional array A, the cross section A(*, *, z) denotes the "slice" at the given value of z and parallel to the x, y plane. A(*, y, *) and A(x, *, *) similarly denote slices parallel to the x, z and y, z planes. A(*, y, z) denotes the vector at the given values of y and z, and parallel to the x axis. A(*, *, *) is equal to A, if A is a three-dimensional map.

Cross sections of itemized maps are defined so that relations such as the following hold, and these relations are regarded as the fundamental properties of cross sections:

$$[f(x, *)](y) = [f(*, y)](x) = f(x, y)$$

$$[[f(x, *, *)](y, *)](z) = f(x, y, z), \text{ etc.}$$

(However, these relations are not sufficient to define cross sections).

We define f(*) to be simply f itself. In other cases, the expression f(x1, ..., *, ..., xn), with at least one asterisk present, is evaluated in dexter mode as follows. For each pair p in f, p(1) is examined. If p(1) is not a vector, then p is excluded from further consideration. If p(1) is a vector, then all components that correspond to asterisks are deleted (made undefined). Similarly treating the asterisk positions of the argument

(x1, ..., *, ..., xn) to be undefined, a "match" is said to occur if the modified p(1) and argument are equal. From the matched pair p, a new pair (q, p(2)) is constructed, where q is the vector of components of p(1) that correspond to asterisk positions, with q(i) corresponding to the i^{th} asterisk. However, if q is of length one (i.e., if there is only one asterisk in the argument), then q is replaced by its component.

Some specific cases, expressed in ASL, are:

$f(a, *, b) = \{[x(2), y],\ \forall(x, y) \varepsilon f \mid \text{VECTOR } x \ \&\ x(:1) \not\subset x(3:) = (a, b)\}$

$f(a, *, *) = \{[x(2:3), y],\ \forall(x, y) \varepsilon f \mid \text{VECTOR } x \ \&\ x(:1) \not\subset x(4:) = (a,)\}$

(This use of the colon is explained in the following section).

As an example, let

$$f = \{(a, b),\ (b, (c, d)),\ ((a, b), d),\ ((a, b, c), e)\}$$

and $g = \{((, a), b)\}$

where a and b are atoms. Then:

$f(a, *) = \{(b, d)\}$
$f(*, b) = \{(a, d)\}$
$f(*, *, c) = \{((a, b), e)\}$
$f(, *, c) = \emptyset$
$f(b, *) = \emptyset$
$f(*) = f$
$f(*, *) = \{((a, b), d)\}$
$g(*) = g$
$g(, *) = \{(a, b)\}$

A cross section of a map is always a map (uniqueness is preserved).

Note that the value of f(*, *) is a subset of f whose domain consists of vectors v such that $LI(v) \geqslant 1$ and $HI(v) \leqslant 2$, i.e., pairs, if undefined quantities are not involved. Similarly,

$f(*,) = \{[x(1), y],\ \forall(x, y) \varepsilon f \mid \text{VECTOR } x \ \&\ x(:0) \not\subset x(2:) = (,)\}$,

which is obtained from f by considering only pairs whose first component is a one-tuple, and by changing such a pair ((x,), y) to (x, y).

Cross sections may not be nested. That is, a construction such as f((a, *), b) is invalid. Furthermore, an argument with an asterisk may not appear in any context other than functional application; (a, *, b) is <u>not</u> a vector. Hence cross section parameter lists may not be built out of line, and a construction such as f((a, *, b)@2) is invalid. The reason for these restrictions is that we do not wish to open up a Pandora's box of definitions of how * behaves in various contexts, which would amount to adding a new data type, or a "state" analogous to the undefined state.

Cross sections may be used in sinister mode. For example, the assignment f(a, *, b) = e, where e is an itemized map, modifies f as follows:

$$f = f - \{\forall (x, y) \in f \mid \text{VECTOR } x \;\&\; x(:1) \not\subset x(3:) = (a, b)\}$$
$$\bigcup \{((a, x, b), e(x)),\ \forall x \in \mathfrak{D} f\}.$$

Similarly, the assignment f(a, *, *) = e, where e is an itemized map whose domain consists entirely of pairs, modifies f as follows:

$$f = f - \{\forall (x, y) \in f \mid \text{VECTOR } x \;\&\; x(:1) \not\subset x(4:) = (a,)\}$$
$$\bigcup \{((a, x, y), e(x, y)),\ \forall (x, y) \in \mathfrak{D} f\}.$$

4.7 Subarrays

The notation f(m:n) has meaning if f is a vector and m and n are integer expressions. It is then the vector (f(m), f(m+1), . . . , f(n)) if $n \geqslant m$, and $\emptyset$ if $n < m$. The expression f(m:n) is called a "subvector" or "substring".

The following forms are also permitted:

f(m:) means (f(m), f(m+1), . . .).

f(:n) means (f(LI f), f(LI f + 1), . . . , f(n)) if $n \geqslant$ LI f, and $\emptyset$ otherwise.

f(:) means (f(LI f), f(LI f + 1), . . .)

Note that the result is "shifted" so that the result is a one-origin vector (or $\emptyset$). Below are some examples of the subvector operation, where v = (, a, , b, c).

```
v(1:2) = ( , a)
v(3:)  = ( , b, c)
v(:2)  = ONEVECTOR a
v(:)   = (a, , b, c)
v(6:)  = ∅
```

The result is always a vector, and the indexes are permitted to be out of range.

The subvector operation could have been defined to have meaning for maps in general, by ignoring non-numeric elements in the domain. However, this would be of little utility, and its inclusion would make programs more obscure and would make the implementation more complicated and somewhat slower in execution.

The subvector notation extends to arrays in an obvious way. For example, if A is the array

	1	2	3
1	a	b	c
2		d	e
3			f

(i.e., the set {((1, 1), a), ((1, 2), b), ((1, 3), c), ((2, 2), d), ((2, 3), e), ((3, 3), f)}) then A(2:3, :) is the array

	1	2
1	d	e
2		f

The colon may not be nested. That is, a construction such as A((1:2, 3), 4) is not permitted. This is because such a construction implies that A is referenced by forms of the type A((i, j), k), which implies that A is not an array. Similarly, a construction such as A((1, 2), 3:4) is not valid.

A cross section is similar to a subarray in which neither limit is given. That is, if A is a regular array then A(i, *) = A(i, :). However, in general these concepts are different. The cross section is defined for maps in general, for one thing, whereas the subarray is only defined for arrays. Even if f is a vector, then f(*) and f(:) may not be the same. For example, if f = (, a, b) then f(*) = (, a, b) but f(:) = (a, b). For one-origin vectors, however, f(*) = f(:).

Subarrays may be used in left-hand side contexts. Intuitively, the meaning of s(m:n) = e, where e must be a vector expression, is to replace components m through n of s with all the components of e, sliding the right-hand components of s to the left or right as necessary to make room.

More precisely, it is equivalent to:

s = [s(:m-1) ¢ e ¢ s(n+1:)] @ MIN(LI s, m).

The domain of e need not match the indexes m and n. For example, we may write s(3:6) = (a, , b) and s(1:2) = ∅. Note that this is not a pure retrieval function; i.e., after s(m:n) = e, it is not necessarily the case that the value of s(m:n) is e. However, if m ⩾ LI s and LENGTH(e) = n-m+1, then it is a pure retrieval function.

The meaning of s(m:) = e is the same as s(m:HI s) = e, that is,

s = [s(:m-1) ¢ e] @ MIN(LI s, m).

Similarly, the meaning of s(:n) = e is the same as s(LI s:n) = e, that is,

s = [e ¢ s(n+1:)] @ LI s.

The assignment s(:) = e is equivalent to s(LI s:HI s) = e, that is, s = e @ LI s.

The assignment to a higher dimensional subarray is an obvious extension of this. Assignments such as A(i:j, k:, 2) = M are permitted.

4.8 IF Expressions

ASL includes the IF expression, similar to that of Algol. As an example, $y = |x|$ could be coded:

$$y = \text{IF } x \geqslant 0 \text{ THEN } x \text{ ELSE } -x.$$

These may of course be nested. The IF expression may also be used in a left hand side context (as in Algol). However, see section 5.2.1 Map Assignments for a similar but more general left hand side expression.

In ASL there is nothing analogous to the decision table for IF expressions.

The words IF, THEN, and ELSE must all be present. However, in a dexter context the expression following either THEN or ELSE may be absent or ( ), indicating the undefined state.

Parsing depends on the fact that IF, THEN, and ELSE are reserved words. They act like prefix operators of precedence lower than that of any built-in operator. For example, the parentheses are required to code $|x| + 1$ as (IF $x \geqslant 0$ THEN x ELSE -x) + 1.

4.9 Value Receiving Expressions

A value receiving expression is one that is syntactically valid in a left-hand side, or sinister, context. These contexts in ASL are:

1. The left-hand side of an assignment statement.
2. An item in a READ I/O list.
3. The stepping expression in a search expression or quantifier predicate.
4. The formal parameters of a procedure heading or ENTRY statement.
5. A RETURN statement when the procedure is invoked in sinister mode.
6. An expression-statement (this is really a case of (1) above; see section 4.2.7).

A value receiving expression is any of the following:

1. A variable name.
2. A function reference, f(x) or f x.
3. A map expression of the form {(x1, e1), (x2, e2), . . . , (xn, en)}, where the ei are value receiving expressions.
4. A vector expression of the form (e1, e2, . . . , en), where the ei are value receiving expressions.
5. A cross section expression, f(x, *,*), etc.
6. A subarray expression, A(i:j), A(i, j:), etc.

7. An IF expression, IF c THEN x ELSE y, where x and y are value receiving expressions.

8. The expression (e), where e is a value receiving expression.

The definition allows an assignment such as (1, 2, 3)(2) = 4, although the effect of such an assignment is nil (a compiler temporary holding the vector (1, 2, 3) is modified to (1, 4, 3)).

4.10 Expression Evaluation

4.10.1 Order of Evaluation

This section discusses the order in which operators are executed and the order in which their operands are evaluated. ASL is very left-to-right oriented in both respects.

Binary operators are evaluated as follows, where we consider juxtaposition of expressions to be a binary operation in which the operator is not explicitly shown. The higher precedence operations are performed first (see the table in section 2.2.3.1). Among operators of equal precedence, with two exceptions they are grouped on the left. That is, a - b + c is evaluated as (a - b) + c. The two exceptions are juxtaposition and exponentiation; log sin x is log(sin x) and x**y**z is x**(y**z).

The order of evaluation of operands is of less importance in ASL than in many languages, because the values of the variables in an expression are captured before evaluation begins, and the captured values cannot change during evaluation of the expression (see the following section). Nevertheless the order of evaluation is significant if the terms involve procedures that have internal counters, share global variables, perform input/output, etc.

The left operand of a binary operator is evaluated first, then the right operand (if at all). The components of set and vector formers, such as (a, b, c), are evaluated in left to right order. For the expression IF c THEN x ELSE y, c is evaluated first. If it is TRUE, then x is evaluated, and y

is not evaluated at all. If it is FALSE, then y is evaluated, and x is not evaluated at all.

For the binary Boolean operators, the left operand is first evaluated. If the result is then independent of the value of the right operand, then the right operand is not evaluated. This applies to &, $\not\&$, $\Longrightarrow$, and $\not\Longrightarrow$ if the first operand is FALSE, and to $\vee$ and $\not\vee$ if the first operand is TRUE. It also applies to an expression such as $a < b < c$; if after evaluating and comparing a and b it is found that $a < b$ is FALSE, then c is not evaluated.

Map and vector assignments are done in right to left order; (a, b, c) = e is executed as c = e(3), b = e(2), a = e(1), ignoring temporaries. Hence in an assignment such as (a, a) = e, it is the leftmost assignment to a that takes effect. The multiple assignment e1 = e2 = ... = en = expr is also done in right to left order.

For a left-hand side function reference such as f(x) = e, first f is evaluated, then x, then e, and finally the assignment itself is done, which may alter either f or x. For left-hand side composition of functions, and more complicated expressions such as the ellipsis, set former with an iteration, and quantifier predicate, see the expansions given in the appropriate section.

4.10.2 Use of Temporaries

This section discusses the use of compiler-generated temporaries in value producing (right-hand side) expression evaluation. Their use in value receiving (left-hand side) expressions is discussed in section 5.2.1.

The intuitive notion of the evaluation of a simple (non-iterative) expression is that all variables in the expression initially have values that are to be used in the evaluation, and the various distinct parts of the expression can be evaluated simultaneously. Unfortunately most programming languages break down on both of these intuitive concepts. ASL only breaks down on the latter concept (simultaniety), and it breaks down less easily than most languages.

For example, in many languages the expressions a + f(x) and f(x) + a have different values if procedure f has the side effect of changing a. This is because the initial value of a might not be used when it enters into the add operation.

This non-intuitive behavior is avoided in ASL by the rule that before any expression is evaluated, all variables in it are assigned to temporaries, and then the evaluation is done in terms of the temporaries. Of course a reasonably sophisticated compiler will omit most of the unnecessary assignments, or will introduce them and later delete them by an optimization process, but nevertheless the program's behavior is as if all the assignments had been done.

For example, if f is a procedure name (a type of constant), the evaluation of:

```
a + f(x)
```

is:

```
t1 = a
t2 = x
t3 = f(t2)
t4 = t1 + t3
```

where t4 gets the result. If f had been a variable (procedure variable or an itemized map), a temporary would have been used for it also (Alternatively, the compiler could simply always assign a temporary for f, and even for all constants.

It is easy to see that no matter what procedure f does, barring its result being a function of time, the expressions a + f(x) and f(x) + a have the same value. Hence there are some cases where addition in ASL commutes because of the use of temporaries, and this can be used to aid optimization efforts (however, addition in ASL does not <u>always</u> commute).

If the same variable occurs more than once in an expression, then for evaluation purposes it makes no difference whether or not it is assigned to distinct temporaries.

It was remarked in section 4.4 that the iteration variable in a search expression, set former, etc., was not a bound variable, although it behaves similarly to a bound variable in that the value of a variable with the same name as the iteration variable is preserved across the expression. For example, the expression:

$$\{v(1),\ \forall v(1) \in S \mid |v| > 5\}$$

is evaluated in terms of a temporary assigned to v. It is the temporary that receives assignments in the ∀ loop. On the other hand, v is not a bound variable because it is the initial value of v that is used to supply components other than v(1) in the conditional $|v| > 5$.

5.1 DECLARE Statement, Attribute Summary

This section briefly describes attributes in general, and shows how to write them in a DECLARE statement.

ASL classifies attributes as "essential" and "inessential". The table on the next page shows all the essential attributes that can be declared, and a few of the inessential ones. A reference where further details may be found is given.

Essential attributes are those that could be required for logically correct operation of a program. Inessential attributes are those that are never required, but are supplied only for commentary, efficiency enhancement, and as debugging aids. For example, the length of a string is an essential attribute, but the maximum length of a string variable is an inessential attribute.

The DECLARE statement may be used to specify any attribute. It is the only way to specify the attributes of variables (as opposed to the attributes of values), i.e., STATIC, DYNAMIC, external, SHARED, INTERNAL, and value. Hence the DECLARE statement is an essential part of ASL, although it is frequently used merely for commentary and efficiency enhancement.

Another way to classify attributes, which is sometimes useful, is as attributes of variables and attributes of values. For example, the storage class attributes STATIC and DYNAMIC are attributes of variables, and not of the values taken on by variables, or of constants. On the other

Attribute	Applies to	Reference Section
Essential		
STATIC, DYNAMIC	variable names	7.5.1
external	variable and procedure names	8.
SHARED, INTERNAL	variable, procedure, and declared constant names	8.
value	variable and declared constant names	5.1
type	all values	5.1
structure	set values	5.1
EXACT, APPROXIMATE	numerical values	3.1.1
ASSIGNS	procedure and variable names	7.6.1
PRECEDENCE	operator names	5.1
Inessential		
range of values	numerical variable names	11.
list of values	variable names	11.
maximum size	set variables	11.
maximum length	array variables	11.
internal representation	set variables and constants	11.
VERIFIED	variable names	11.
etc.		

hand, EXACT and APPROXIMATE are attributes of (numerical) values. Generally speaking, an attribute of a variable can only be specified by means of a DECLARE statement, but an attribute of a value originates by the way a constant is written, and it is passed on by assignment (and hence is passed on as a part of the parameters of procedure linkage).

A variable attribute (e.g. STATIC) is never considered to be a property of a value. However, a value attribute (e.g. type) can be specified (in a DECLARE statement) as a property of a variable. This is taken to be a comment to the effect that all values assigned to the variable will have the specified value attribute. Data conversions are not invoked, although internal representations might change by assignment (such a change does not affect the results of a valid program, except possibly when the numeric tolerance is involved.

As an example, consider the statements:

```
x = 1
y = x + 1.0
```

The constant "1" is EXACT, and hence the value assigned to x is EXACT. The program would be invalid if x had been declared as APPROXIMATE. The constant "1.0" is APPROXIMATE. The addition rules dictate that the sum of an EXACT number and an APPROXIMATE number is APPROXIMATE, and hence the value assigned to y is APPROXIMATE. Again, the program would be invalid if y had been declared EXACT.

An ASL variable cannot have a value attribute unless it also has a value. For example, if a program contains the declaration DECLARE K NUMERIC,

the type of K is undefined when K is undefined, and the assignment "K = ;" is valid (this is pointed out mainly because it differs from Algol 68).

The distinction between attributes of variables and attributes of values cannot conveniently be rigorously maintained. For example, the external attribute can be an attribute of a variable name or of a procedure name, which is a type of constant.

A declaration such as:

DECLARE $1 \leqslant i \leqslant 1000$ INTEGER

can be viewed as mere commentary, since it does not affect the results of a valid program. However, it is a type of comment that can be interpreted and used by the compiler. For example, the above declaration (which means that the values assigned to i are always integers from one to a thousand) might cause i to be stored as a half word on the IBM System/360. Hence there is a fair chance that a program will not work properly if a comment "lies". The VERIFIED attribute may be used to invoke a run-time check after each assignment to i to verify that i is, in fact, an integer from one to a thousand.

The essential attributes will now be briefly described. For further details refer to the section given in the preceding table.

STATIC and DYNAMIC are the storage class attributes. A STATIC variable retains its value when a procedure is invoked recursively. A DYNAMIC variable is pushed down, and a new (initially undefined) value is popped up.

External, SHARED, and INTERNAL are the name scoping attributes. A variable is INTERNAL unless declared otherwise, and an INTERNAL variable is known only within its "owning" procedure. A SHARED variable is known in other procedures, which may be all procedures in a program or an explicitly identified subset. The external attribute declares a variable to be identified with a variable (possibly of a different name) owned by another procedure.

An internal procedure's name is SHARED with all descendents of its parent, unless declared otherwise. An outermost procedure's name is SHARED globally. A "declared constant" (as the "x" in "DECLARE x 1.0") may be INTERNAL (the default) or SHARED, but it may not be external.

The value attribute is the value itself. It is given to a variable by an assignment statement, READ statement, etc., and to a declared constant by a declaration such as the one in the preceding paragraph and "DECLARE ARTICLES {'a', 'an', 'the'}". A declared constant cannot enter into an assignment (and hence is considered to be a constant), but it has some properties normally associated with variables, such as the fact that it can be SHARED.

The type attribute classifies a value as being one of the six ASL data types NUMERIC, CHARACTER, BOOLEAN, PROCEDURE, POINTER, and SET.

The structure attribute further classifies a SET as being a RELATION, MAP, ARRAY, MATRIX, VECTOR, STRING, or PAIR. In addition, the

structure may be further described by such phrases as STRING(CHARACTER), SPARSE ARRAY DIMENSION = 3, etc.

EXACT and APPROXIMATE are numerical precision attributes. An EXACT number is represented internally as a ratio of integers, either of which may be arbitrarily large. An APPROXIMATE number is represented in a more convenient (for the machine) form, generally in the machine's floating point format. Examples of EXACT constants are 1, 1E100, and 1/3 (the last is actually an expression whose value is EXACT). Examples of APPROXIMATE constants are 1.0, 1.0E100, and 0.333.

The ASSIGNS attribute identifies which arguments of a function or operator are assigned new values when the function or operator is invoked in a sinister (value receiving) context. For example,

DECLARE f(x, y, z) ASSIGNS(x, z)

specifies that in an assignment such as f(a, b, c) = . . . , a and c are assigned new values (provided they are valid value-receiving expressions), but b is not. The ASSIGNS attribute need not be given in a DECLARE statement. It is a value attribute, i.e., it is inferred from the procedure itself and passed on by assignments to procedure variables.

The PRECEDENCE attribute specifies the precedence of user-defined operators. The precedence is a numerical value (not necessarily an integer). An example of a precedence declaration is:

DECLARE .MYPLUS. PRECEDENCE = 3

The keyword PRECEDENCE is also a predefined function

that returns the precedence of any operator. Thus we may write (using the allowed abbreviation of PRECEDENCE):

```
DECLARE .MYPLUS. PREC = (PREC(+) + PREC(*))/2.
```

The precedence attribute is an attribute of an operator name, and is not inferred from the definition of the operator. Thus a statement can be parsed without having compiled the definition of user-defined operators. A user-defined operator could be given different precedences in different procedures, but (like variable name attributes) cannot be changed within a procedure.

The default precedence is that of the binary +.

The table on the following page shows which attributes are incompatible within the same procedure. For example, a variable could not be both SHARED and external in the same procedure, but it might be SHARED in one and external in another.

Many attributes are implied by others. For example, declaring a variable to be APPROXIMATE implies that its type is NUMERIC. Redundant attributes are always permitted, provided they don't conflict. For example, the following are valid:

```
DECLARE x NUMERIC APPROXIMATE
DECLARE S (1, 2, 3) SET VECTOR STRING.
```

	APPROXIMATE	ASSIGNS	DYNAMIC	EXACT	external	INTERNAL	structure	PRECEDENCE	SHARED	STATIC	type	value	Abbrev.
APPROXIMATE	-	X		X			X	X					APPROX
ASSIGNS	X	-		X									
DYNAMIC			-		X			X		X		X	
EXACT	X	X		-			X	X					
external			X		-	X		X	X	X		X	
INTERNAL					X	-		X	X				
structure	X			X			-	X					
PRECEDENCE	X		X	X	X	X	X	-	X	X	X	X	PREC
SHARED					X	X		X	-				
STATIC			X		X			X		-		X	
type								X			-		
value			X		X			X		X		-	

Attribute Conflict Matrix

(X = conflict)

5.1.1 Placement of DECLARE Statements

A declaration for an identifier must precede the first occurrence of the identifier in the procedure, not counting the occurrence of identifiers in a procedure heading.

The essential attributes of an identifier that can only be specified in a declaration must all be given in the same declaration. Thus it is invalid to write DECLARE x STATIC; DECLARE x SHARED; or even DECLARE x STATIC, x SHARED. It must be DECLARE x STATIC SHARED. The attributes in this category are STATIC, DYNAMIC, external, SHARED, INTERNAL, value (the effect of a value declaration cannot be specified by assignment), and PRECEDENCE.

The inessential attributes and the essential attributes that are specified for commentary reasons may appear in different declarations. They must, however, precede the first non-procedure-heading occurrence of the identifier.

These rules are included for the benefit of the reader (although they may also help the compiler). When the reader of a program wants to know the attributes of an identifier, he need only search in one direction: upwards. When he finds an attribute that can affect the operation of the program (such as SHARED), he knows that he need look no further for other such attributes. The rules are relaxed with regard to the other attributes to facilitate adding efficiency enhancing or debugging declarations without altering any existing code.

5.1.2 Writing the DECLARE Statement

The DECLARE statement is written in the PL/I style, with attributes listed with each item being declared, rather than the other way around, as in FORTRAN. Factoring is permitted. The general form without factoring is:

DECLARE item attribute [attribute] . . . [, item attribute [attribute]. . .]. . . ;

Here "item" is the item being declared. It must be an identifier (other than a keyword), or a user-defined operator name (which is a period-delimited character string with no embedded blanks or periods).

At least one attribute is required (unlike PL/I). The word DECLARE may be abbreviated DCL. A comma is used to separate items appearing in the same statement, but attributes are separated only by blanks. The attributes may be written in any order, but the item being declared must precede all of its attributes.

5.1.2.1 Storage Class Attributes

The storage class attributes are written STATIC and DYNAMIC, e.g.,

DECLARE x STATIC, y DYNAMIC;

5.1.2.2 Name Scoping Attributes

The internal attribute is written INTERNAL and the shared attribute is written SHARED, optionally followed by a list of procedure names enclosed in parentheses. DECLARE x SHARED declares x to be accessible to any procedure that has an appropriate external declaration. DECLARE x SHARED (P, .OP.)

declares x to be accessible from procedures P and .OP., provided they have the appropriate external statement, but x is not accessible anywhere else. The external attribute is written in either of the forms:

IN proc

= name IN proc.

For example, DECLARE x IN P1, y = z IN P2 declares x to be identified with the variable (or procedure) of the same name that is owned by procedure P1, and y to be identified with the variable (or procedure) z owned by procedure P2. The x in P1 and the z in P2 must be SHARED. This attribute may be factored, as illustrated below.

DECLARE (x, y, z) IN proc

DECLARE (x, y = a, z = b) IN proc

DECLARE (x, y) = a IN proc

5.1.2.3 Value Attribute

The value attribute is written as a self-defining value or an expression involving only self-defining values. For example:

DECLARE epsilon 0.001, (x, y, z) (1, 2, 3+4).

The second declaration declares x, y, and z to all be names for the vector constant (1, 2, 7) (it does not mean something closer to the assignment (x, y, z) = (1, 2, 7)).

The manner of writing a constant value implies all of its essential

attributes. It is sometimes useful, however, to supply inessential attributes along with the value attribute, for example:

```
DECLARE KEYWORDS {'IF', 'THEN', 'ELSE'} HASHED.
```

5.1.2.4 Type Attribute

The six type attributes are written NUMERIC, CHARACTER, BOOLEAN, PROCEDURE, POINTER, and SET. The first five may be abbreviated NUM, CHAR, BOOL, PROC, and PTR. Some examples:

```
DECLARE (x, y, z) NUMERIC
DECLARE SW BOOL, SUB PROC.
```

The attribute INTEGER (abbreviated INT) is frequently used. This is an inessential attribute that implies NUMERIC and has the obvious additional meaning.

5.1.2.5 Structure Attributes

In many cases a variable takes on values that are always sets of a certain fixed structure. The structure attributes serve to identify this structure. The "set display" and the "vector display" are catch-alls that can represent any structure more or less pictorially. The other structure attributes in some cases merely abbreviate commonly occurring structures, and in other cases they supply additional information, such as the maximum length of a string.

The structure attributes are:

set display

vector display

RELATION

MAP

ARRAY, MATRIX, VECTOR, STRING, PAIR

DENSE, SPARSE, REGULAR

DIMENSION, SIZE

LI, HI, LENGTH

5.1.2.5.1 Set Display Structure Attribute

The set display is written {attributes}. For example:

DECLARE S1 {NUMERIC}, S2 {{CHARACTER}};

declares that the variable S1 takes on values that are always sets of numbers (or the null set), and S2 takes on values that are sets of sets of characters. These declarations mean that the set is homogeneous.

There is no way to declare, for example, that a variable's values are sets of numbers and characters, but with no other data type.

The syntax of the set display is very close to the value attribute, the only difference being that in the case of the set display, the material within the braces cannot be reduced to a self-defining value (a constant). A similar appearing value attribute might be written DECLARE S1 {'NUMERIC'}.

The attributes STATIC, DYNAMIC, external, SHARED, INTERNAL, PRECEDENCE, and some of the inessential attributes are not permitted in a set display.

5.1.2.5.2 Vector Display Structure Attribute

The vector display is written in the following forms:

(attribute list)

(attribute list) @ expression

(attribute list, . . .)

(attribute list, . . .) @ expression

Here "attribute list" is a list of one or more attributes or an asterisk, separated by commas, and "expression" is an expression that can be evaluated at compile time. Some examples of the vector display:

```
DECLARE V1 (NUM, CHAR, NUM, PTR)
DECLARE V2 (NUM, *, PROC) @ (3*(4-1))
DECLARE V3 (NUM, , PROC)
DECLARE V4 (NUM, CHAR, . . . )
```

Here V1 is specified to have values that are vectors normally of length

four, whose first component (if present) is numeric, and whose second component (if present) is a single character, etc. The possibility of components not being present is allowed so that V1 can be built up component-by-component, i.e., "V1(2) = 'a'; V1(1) = 1;" etc.

V2's values, when all components are present, are nine-origined vectors, with V2(9) numeric, V2(10) of unspecified attributes, and V2(11) a procedure. V3 is similar to V2, except that V3 is one-origined and its second component is never present.

V4's values are (normally) one-origined vectors of length one or more, whose first component is numeric, and whose remaining components, if any, are all single characters.

There are some vector structures that cannot be declared in ASL. One is a vector whose left end "floats". That is, (..., NUM, CHAR), for example, is not allowed. There is no way to specify a list of alternative attributes for a component (analogously to the Algol 68 union). There is no direct way to specify that a component is always present, although some specifications to this effect may be accomplished by means of LI, HI, and LENGTH, which are discussed below.

The * and omitted specification serve as place markers. Thus in (NUM, *, CHAR), component number one is numeric, and component number three is a character, and this holds when the second component is undefined, as well as when the first or third components are undefined.

The value attribute may be used as a component of a vector, in

which case the value is assigned and may not be altered (the vector is a "partial self-defining value" or "partial constant"). Thus after DECLARE V (2, NUM, CHAR), assignments such as V(1) = . . . or V = . . . are invalid. The validity of V(i) = . . . can, in general, only be determined at run time. The value attribute may not precede the ellipsis (e.g., DECLARE X (1, . . .) is invalid). This anomaly is a consequence of the anomaly that the value attribute causes an assignation.

As with the set display, the vector display cannot include the attributes STATIC, DYNAMIC, external, SHARED, INTERNAL, PRECEDENCE, and some of the inessential attributes, within the display itself.

5.1.2.5.3 RELATION Structure Attribute

The RELATION attribute specifies that the values taken on by the declared variable are sets of regular vectors all of the same length, with the length greater than or equal to two. The attributes of the vectors may be shown in parentheses immediately following the word RELATION, similarly to a vector display. Some examples:

```
DECLARE R1 RELATION
DECLARE R2 RELATION(NUM, CHAR)
DECLARE R3 {(NUM, CHAR)} RELATION
DECLARE R4 RELATION((CHAR, ...), NUM, *)
```

Here R1 is merely declared to be a relation, but it is not known whether it is binary or trinary, etc. R2 and R3 are binary relations between numbers and single characters. For R3, the set display is used.

The word RELATION informs the reader (and the compiler) that the elements of R2 and R3 are always pairs. Without the word RELATION, set R3 could contain null vectors and vectors of length one. The declarations of R? and R3 are equivalent.

Another (possibly less suggestive) way to specify R2 is:

DECLARE R2 {(NUM, CHAR) LENGTH = 2}

R4 is declared to be a trinary relation between character strings, numbers, and a third quantity of unspecified attributes. In the context of the RELATION attribute, the * is taken to mean a component that is always present (defined). (This remark applies to (CHAR, . . .) and NUM also).

5.1.2.3.4 MAP Attribute

The MAP attribute declares that the values taken on by a variable are either itemized maps or procedure maps. That is, the values are either sets or ordered pairs whose first component is unique, or are procedures (of either function or operator type) that are free of side effects. The two possibilities can be distinguished by giving the type, i.e., SET MAP or PROCEDURE MAP.

The attributes of the domain and range of the map may be written in parentheses immediately following the word MAP. For example:

DECLARE M1 MAP(INT, CHAR)

DECLARE M2 MAP((CHAR, . . .), *)

declares M1 to be a map from integers to characters, and M2 to be a map

from character strings to objects of unspecified attributes.

A set map differs from a binary relation in that the pairs that make up a set map must have unique first components.

5.1.2.5.5 ARRAY, MATRIX, VECTOR, STRING, and PAIR Structure Attributes

These are special cases of SET MAP, and they imply those attributes. An array is a set map whose domain consists of regular vectors of integers, all of the same length, this length being the dimension of the array. A matrix is an array of two dimensions, and a vector is a set map whose domain consists of integers. A string is a regular vector, i.e., a vector whose domain contains all integers from one up to some maximum, or is null. In practice, strings are usually homogeneous (the members of the range have some attributes in common), but they need not be. A pair is a string of length two.

If an array, matrix, vector, string, or pair is homogeneous in some sense, the common attributes of the range may be written in parentheses immediately following the word ARRAY, MATRIX, VECTOR, STRING, or PAIR.

Some examples:

```
DECLARE A1 ARRAY
DECLARE A2 ARRAY(NUMERIC)
DECLARE V1 VECTOR(CHAR)
DECLARE V2 VECTOR(CHAR,...)
DECLARE S1 STRING(CHAR)
DECLARE P1 PAIR(CHAR)
```

Here A1 is merely declared to be an array. The number of dimensions

is unknown, as is the range of its indexes. A2 is an array of numbers, but its dimensionality is also unknown. V1 is a vector of single characters, and V2 is a vector of character strings. S1 is a conventional character string. An equivalent declaration of S1 is DECLARE S1 (CHAR, ...); the preference is a matter of taste. P1 is a pair of characters. PAIR is equivalent to STRING LENGTH = 2, and hence an equivalent declaration for P1 is DECLARE P1 STRING(CHAR) LENGTH = 2. The preference would depend upon which is more suggestive of how P1 is used in the program.

The vector display cannot be written immediately following the word RELATION, MAP, ARRAY, MATRIX, VECTOR, STRING, or PAIR. Thus DECLARE V1 VECTOR(CHAR, ...) declares V1 to be a vector of character strings, whereas DECLARE V2 (CHAR, ...) VECTOR declares V2 to be a character string, and the word VECTOR is redundant. This restriction also applies to the SHARED attribute; it and the fact that the item being declared must appear before its attributes are the only restrictions on the order of items in a declaration.

5.1.2.5.6 DENSE, SPARSE, and REGULAR Structure Attributes

These attributes apply to arrays and vectors. A dense array is one whose domain ranges from some (i_{min}, j_{min}, ...) to (i_{max}, j_{max}, ...) with no "holes". That is, if $i_{min} \leqslant i \leqslant i_{max}$ and $j_{min} \leqslant j \leqslant j_{max}$, etc., then A(i, j, ...) is present (defined) if A is dense. A dense vector is similarly defined. Note that a dense array cannot be "ragged". SPARSE means "not necessarily dense". If DENSE is not specified or implied by some

other attributes, then SPARSE is assumed, and hence the attribute SPARSE is of value mainly to the human reader (The compiler may use it to assure that it doesn't conflict with something else, for example SPARSE STRING is invalid).

REGULAR means (1, 1, . . .)-origined and dense. Although REGULAR may be applied to strings and pairs, it is redundant. In fact, STRING is merely an abbreviation for REGULAR VECTOR.

An example:

```
DECLARE A ARRAY(NUMERIC) DENSE;
```

5.1.?.5.7 DIMENSION and SIZE Structure Attributes

The DIMENSION attribute applies to arrays, and the SIZE attribute applies to sets in general. They are written in the form:

identifier relation expression

(for example "DIMENSION = 3"), where "identifier" is either DIMENSION or SIZE, "relation" is one of the six relational operators $= \neq < \leqslant > \geqslant$, and "expression" is an expression that can be evaluated at compile time, and whose value is such that the dimension and size are non-negative.

SIZE implies SET and DIMENSION implies ARRAY, which implies SET. For clarity, one might choose to be redundant and write, for example, SET SIZE <10.

DIMENSION may be abbreviated DIM.

Some examples:

```
DECLARE S SIZE ⩽100
```

```
DECLARE S SET SIZE ≤100
DECLARE A DIM = 2
DECLARE A ARRAY DIMENSION = 2
DECLARE F SET SIZE ≠ 0
```

Here the two declarations for S and for A are equivalent.

When DIMENSION and SIZE are written with one of the relational operators $<\leqslant>\geqslant$, the specification is, strictly speaking, an inessential attribute. However, it is convenient to discuss it in this section. The same remark applies to LI, HI, and LENGTH in the following section.

5.1.2.3.8 LI, HI, and LENGTH Structure Attributes

These attributes apply to arrays, vectors, and strings. They are written in either of the forms:

identifier relation expression

$\text{expression}_1\ \text{relation}_1\ \text{identifier}\ \text{relation}_2\ \text{expression}_2$

(for example $3 \leqslant \text{LENGTH} < 10$). Here "identifier" is either LI, HI, or LENGTH, optionally subscripted with an expression that can be evaluated at compile time and whose value is a positive integer. Each "relation" is one of the six relational operators $= \neq <\leqslant>\geqslant$. Each "expression" is an expression that can be evaluated at compile time and whose value is numeric. LENGTH is further restricted to non-negative values (although $\text{LENGTH} > -1$ is valid). If the relation is "=", the expression is further restricted to have an integral value. In the second form, the relational

operators are restricted to $\lessdot\leqslant>\geqslant$, and they must both point in the same direction.

The optional subscript on LI, HI, and LENGTH is used to refer to the different dimensions of the array. That is, to specify that a regular array is 10 x 20, one may write either "HI(1) = 10 HI(2) = 20" or "HI = (10, 20)". To specify that a regular array is of varying size up to 10 x 20, one can write "HI(1) $\leqslant$ 10 HI(2) $\leqslant$ 20", but the specification "HI $\leqslant$ (10, 20)" is not valid.

Some examples:

```
DECLARE STR STRING(CHAR) LENGTH ⩽20

DECLARE A1 ARRAY(NUMERIC) LI = (1, 1)

DECLARE A2 ARRAY(NUM) REGULAR HI = (10, 20)

DECLARE A3 ARRAY(CHAR, ...) LI = (1, 1) HI(1) = 2 ....
       HI(2)⩽100

DECLARE V STRING(ARRAY((CHAR, ...) LI ⩽10) LI = (1, *)) ....
       10 ⩽LENGTH ⩽ 20
```

Here STR is a character string of maximum length 20. Being a string, it is dense and LI(STR) is 1. The variable A1 is a two-dimensional array (as implied by LI = (1, 1)) of numbers. There is no specified limit on HI(A1), and A1 may be sparse. A2 is similar to A1 except that it is dense and of fixed size 10 x 20. A variable such as A2 is of limited value in ASL, as it may not be built up component-by-component. It would be more usual to replace "HI = (10, 20)" with "HI(1) $\leqslant$ 10 HI(2) $\leqslant$ 20 DIM = 2", and possibly also DENSE, if it is built up in an appropriate order. Declarations

of fixed size arrays would occur more frequently in declarations of nested structures.

A3 is a two-dimensional array of character strings.

V is a string whose length varies from 10 to 20, and whose elements are arrays. The arrays are two-dimensional with LI(1) = 1 and LI(2) unspecified (an * here is taken to mean "present but not specified"). Each element of the array is a character string of maximum length 10.

As a final example of the manner of writing structure attributes, the declaration below is for a simplified symbol table, such as might be found in a compiler. The symbol table is a map from character strings of varying length (the "symbols", or identifiers, that have appeared in the text being analyzed) into vectors giving the various properties of the symbols. The vectors might contain an integer giving the storage location that has been assigned to the symbol, a short character string identifying the symbol's type ("REAL", "INTEGER", etc.), and various flags. The vectors in the range of the symbol table are (in general) sparse, as the properties of each symbol are accumulated gradually. Such a simplified symbol table might be declared as follows:

```
DECLARE SYMTAB MAP[STRING(CHAR) LENGTH ⩾ 1, ....
                   (INT, STRING(CHAR) HI ⩽ 10, BOOL, BOOL)]
```

5.1.2.6 Precision Attributes

The precision attributes are written EXACT and APPROXIMATE, for example:

```
DECLARE x EXACT,  y APPROXIMATE;
```

5.1.2.7 ASSIGNS Attribute

The ASSIGNS attribute is written as shown by the following example:

```
DECLARE f(x, y, z) ASSIGNS(x, z)
```

The arguments (x, y, z in the above) are strictly local to the declaration. For example, in:

```
DECLARE f(x, y) ASSIGNS(x), x NUMERIC;
```

there is no connection between the x in "x NUMERIC" and the other x's.

In ASL, there is no way to specify the attributes of the parameters of a function, unless the function happens to be free of side effects. Then one can write, for example:

```
DECLARE f(x, y) PROCEDURE MAP((NUM, NUM), CHAR) ....
        ASSIGNS x;
```

5.1.2.8 PRECEDENCE Attribute

The PRECEDENCE attribute is written:

```
PRECEDENCE = expression
```

where "expression" is an expression that can be evaluated at compile time, and whose value is numeric. PRECEDENCE may be abbreviated PREC. For example:

```
DECLARE .SPECIALOP. PRECEDENCE = 0;
DECLARE .+. PREC = PREC(+) + 0.5;
```

5.2 Assignment Statement

ASL includes four varieties of assignment statements:

simple assignment,

function assignment,

map assignment, and

multiple assignment.

A simple assignment has a single variable on the left-hand side, possibly enclosed in parentheses. A function assignment is of the form f(x1, x2, ..., xn) = expr, where f is a function (itemized map or procedure). It may also be of the form x .OP. y = expr, where .OP. is an operator-procedure. A map assignment is of the form {(x1, e1), (x2, e2), ..., (xn, en)} = expr, where the ei are value receiving expressions (see section 4.9). A multiple assignment is of the form e1 = e2 = ... = en = expr, where the ei are value receiving expressions.

In a simple assignment, the right-hand side expression is evaluated, and a copy of the result is assigned to the variable on the left. Hence after

```
A = {1, 2, 3}
B = A
A = A - {1}
```

B is still the value assigned to it, namely {1, 2, 3} (this is pointed out only because some LISP-oriented languages do not work this way). An ASL implementation may, for efficiency, not actually perform a copy operation for the assignment B = A, but the program logically behaves as if a copy

had been made.

If the right-hand side expression is undefined, then the left-hand side is made undefined also. Furthermore, one may write "x = ;", or "x = ( );", to explicitly make x undefined.

ASL does not take the view that an assignment is an expression that has a value and hence can be used anywhere an expression can be used. The main reason for this is for the sake of readability. We do not wish to have deeply hidden assignments. Another reason is to preserve the dual use of the "=" sign. That is, the statement a(b=c) = d; means to compare b and c and set either a(TRUE) or a(FALSE) equal to d. It does not mean to set b = c and then set a(b) = d.

Function assignments are discussed in section 4.2, Function Referencing. Briefly, the meaning of f(x) = y depends on the data type of f. If f is an itemized map, it means

$$f = f - \{p,\ \forall p \in f \mid p(1) = x\} \cup \{(x, y)\}$$

except for minor differences involving side effects if f or x are expressions involving procedure references. It is permissible for f to be undefined, in which case it is treated as the null set. If f is a procedure, f(x) = y means to invoke f in a sinister call, which generally results in a change to x. If f is any other data type, or is a set that is not a map, then the assignment is invalid.

5.2.1 Map Assignments

The map assignment:

$$\{(x1, e1),\ (x2, e2),\ \ldots,\ (xn, en)\} = expr$$

is essentially equivalent to:

```
en = expr(xn)
...
e2 = expr(x2)
e1 = expr(x1).
```

By "essentially equivalent" we mean that we are ignoring side effects and peculiar cases involving interdependencies between the ei. The exact meaning will be clarified in the subsequent section on the use of temporaries in assignments.

The xi in the above must all be defined. The right-hand side, expr, may be undefined, in which case the ei are all made undefined.

As a practical matter the map assignment is usually used in the special case of a vector assignment. An assignment of the form:

$$\{(1, e1),\ (2, e2),\ \ldots,\ (n, en)\} = expr$$

may be abbreviated:

$$(e1,\ e2,\ \ldots,\ en) = expr.$$

Some of the ei may be absent, e.g.,

$$(x,\ ,\ y) = expr$$

is essentially equivalent to:

```
y = expr(3)
x = expr(1)
```

Map assignments may be nested. For example,

```
((x, y), z) = expr
```

is essentially equivalent to:

```
z = expr(2)
(x, y) = expr(1)
```

which in turn is essentially equivalent to:

```
z = expr(2)
y = [expr(1)](2)
x = [expr(1)](1).
```

Although permitted, assignments involving nested maps that are not vectors would seldom be used, because they are rather intricate. For example, the above nested vector assignment could be written:

```
{(1, {(1, x), (2, y)}), (2, z)} = expr.
```

The most frequent use of map assignments is the use of vector assignments to group related quantities, analogously to the PL/I structure declaration. Of course in ASL the grouping is done at execution time, whereas in PL/I it is static.

As another example of vector assignments, after

```
(, a, b, (c, d), e) = (1, 2, 3, (4, ), (5, 6), 7)
```

we have a = 2, b = 3, c = 4, d is undefined, and e = (5, 6).

Function assignments and map assignments may be combined, as illustrated by:

```
(x, y, z)(i) = expr.
```

Here either x, y, or z receives a new value, depending on whether i is 1,

2, or 3, respectively. It might seem that this would cause the vector (x, y, z) to be constructed and stored in a temporary, modified, and the result left in the temporary. However, this does not happen. As explained in the subsequent section on use of temporaries in assignments, the above is essentially equivalent to:

```
t = (x, y, z)
t(i) = expr
(x, y, z) = t.
```

Hence the assignment IF c THEN x ELSE y = z; may be coded:

```
(x, y)(IF c THEN 1 ELSE 2) = z;
```

or:

```
{(TRUE, x), (FALSE, y)}(c) = z.
```

The latter expands as follows:

```
t = MAKEMAP{(TRUE, x), (FALSE, y)}
t(c) = z
{(TRUE, x), (FALSE, y)} = t.
```

The last line further expands into x = t(TRUE); y = t(FALSE). The MAKEMAP predefined function converts its argument, which must be a set, to a map, by deleting all non-pairs. It is necessary to introduce this because either x or y may be initially undefined, and we must assure that t is a map for the assignment t(c) = z to be valid. The MAKEMAP function is not necessary in the case of vector assignments because the (right-hand) evaluation of, for example, (x, y, z) with y undefined, is {(1, x), (3, z)}, and not {(1, x), (2,), (3, z)}.

The MAKEMAP function is sometimes used in ASL source code for the same purpose.

For the assignment f(x) = expr, ASL does not introduce the MAKEMAP function. This is because in most cases it would be a waste of time, and some program errors would be missed.

Although one could give meaning to assignments such as:

$$(a, b) \not\subset (c, d) = expr$$
$$\{(a, b)\} \cup \{(c, d)\} = expr,$$

they are not permitted in ASL. This is mainly because they do not seem to be of significant value.

5.2.2 Multiple Assignments

The multiple assignment is written

$$e1 = e2 = \ldots = en = expr,$$

where the ei are value receiving expressions. This is essentially equivalent to:

```
t = expr
en = t
...
e2 = t
e1 = t.
```

However, expressions within the ei are evaluated before the right-hand side (in left to right order).

Parsing of assignments is complicated by the fact that the = sign has two precedences. The situation is similar to that existing in PL/I, but worse

because of the existence of multiple assignments. Some examples follow.

ASL Statement	Meaning
a = b = c = d	a←b←c←d
(a = b = c = d)	a←b←c←d
a = (b = c = d)	a←(b = c = d)
(a = b = c) = d	invalid (like "1 = d")
a = (b = c) = d	invalid
a = b = (c = d)	a←b←(c = d)

All the left-hand sides of a multiple assignment are done "simultaneously" when possible. For example, an assignment such as

a(i) = i = 2

sets a(i) = 2 using the old value of i, and sets i = 2. The assignment i = a(i) = 2 does the same thing. The order of assignment is only important in the case of sinister calls with side effects. For example, in

f(x) = g(y) = expr,

procedure g is invoked first, then f.

5.2.3 Use of Temporaries in Assignments

The use of temporaries in assignments and other value receiving contexts is similar to that discussed in section 4.10.2 for value producing expressions. However, a final step is necessary so that an assignment will in fact be done. An assignment is evaluated as follows:

1. Set temporaries for all variables appearing on both sides of the assignment.

2. Perform the assignment in terms of the temporaries. As for expressions in general, the order of tree traversal is left-right-top, and the assignment operation is of course at the top.

3. Append reversed assignments, in the reverse order, for all temporaries used in the value receiving (left-hand side) portion of the expression, omitting those that are obviously invalid (e.g., t1 + t2 = t3, 2 = t1, etc.).

For example, consider the simple assignment

```
x = y
```

where x and y are variables. This becomes:

```
t1 = x
t2 = y
t1 = t2
x = t1
```

This can be simplified by recognizing that the first assignment is an assignment to a dead variable and hence can be omitted. The variable y can be propagated to the third assignment and from there to the fourth. This makes the second and third assignments assignments to dead variables, and hence expendable. The final result is simply "x = y", and hence in this case the introduction of temporaries was superfluous.

Now consider:

```
f(x) = y.
```

This is defined to mean:

```
t1 = f
t2 = x
t3 = y
t1(t2) = t3
IF SET(t1) THEN f = t1 ELSE x = t2
```

(see section 4.2.5 for the meaning of t1(t2) = t3). If f is a map (array, etc.), then the variables x and y may be propagated forward, giving:

```
t1 = f
t1(x) = y
f = t1.
```

This sequence is of the form t1 = f; modify t1; f = t1; and hence may be simplified to f(x) = y. Again, the temporaries have no net effect.

If f is a procedure, then it presumably changes its argument when invoked in a sinister call, but it may also change f (assuming f is a variable), x, and y through external linkage. In this case the three temporaries have some effect. The use of t1 saves f for the subsequent test. The use of t2 clearly identifies which value is finally assigned to x if it is altered both by its appearance as an argument and by external linkage (its appearance as an argument takes precedence). The use of t3 clearly indicates what value of the right-hand side is used by f even if f changes y through external linkage (in some language f might be invoked and allowed to run until it needs y, and then y would be evaluated. This would be a "sinister call by name").

Similar remarks apply to the dexter procedure call y = f(x) if f can change y, f, and x through external linkage.

Now consider the assignment:

f(g(x)) = y.

This expands as follows:

```
t1 = f                  t1(t5) = t4
t2 = g                  IF SET(t1) THEN f = t1 ELSE DO
t3 = x                    t2(t3) = t5
t4 = y                    IF SET(t2) THEN g = t2 ELSE x = t3
t5 = t2(t3)               END
```

If f and g are maps (itemized), then the expansion simplifies to t5 = g(x); f(t5) = y; which is conventional.

In the general case, where f and g may be procedures with side effects, the above expansion cannot be simplified significantly. This is because the first reference to t2(t3) (i.e., to g(x)) may alter f, g, x, and y, and hence all four temporaries may be needed. However, if we assume that f and g are side effect free, or rather that they can't change certain of the four variables, then the above simplifies to:

```
t5 = g(x)
f(t5) = y
g(x) = t5.
```

This is because the full expansion does not alter t1 or t2 in this case, and the assignment g(x) = t5 alters x in the same way that the sequence t3 = x; g(t3) = t5; x = t3; does. This simplified expansion is precisely that given in section 4.2.6, Sinister Composition of Functions, and the utility of the last step (g(x) = t5) is discussed there. In section 4.2.6 this result is

derived in a somewhat different (but actually very similar) way, but we now know how sinister composition of functions works in the presence of side effects (This is defined by the full expansion given above).

Here is a more substantial example of the use of temporaries. The assignment:

```
f(g(x), h(y+z)) = x + y
```

becomes:

```
t1 = f          t7 = t2(t3)        IF SET(t1) THEN f = t1 ELSE DO
t2 = g          t8 = t5 + t6         (t7, t9) = t10
t3 = x          t9 = t4(t8)          t4(t8) = t9
t4 = h          t10 = (t7, t9)       IF SET(t4) THEN h = t4 ELSE;
t5 = y          t11 = t3 + t5        t2(t3) = t7
t6 = z          t1(t10) = t11        IF SET(t2) THEN g = t2 ELSE x = t3
                                     END
```

The vector assignment in the above may be further expanded into t9 = t10(2); t7 = t10(1).

The use of temporaries makes vector assignments work in a more intuitively satisfactory way. One thinks of the assignment (e1, e2) = e3 as assigning to e1 and e2 simultaneously. Consider;

```
(i, a(i)) = (2, 3).
```

This becomes:

```
t1 = i          t1 = t4(1)
t2 = a          t2(t3) = t4(2)
t3 = i          IF SET(t2) THEN a = t2 ELSE i = t3
t4 = (2, 3)     i = t1
```

If a is a map, then a(i) is set to 3, using the original value of i (saved in t3). If a is a procedure, which presumably alters i in the sinister call, then the final value of i is 2 (the leftmost assignment to i), and not that given by a(i) = 3.

Note that distinct temporaries must be used for the two occurrences of i on the left-hand side. However, identical variables occurring on the right-hand side need not have distinct temporaries.

The notion of simultaneous assignment breaks down if the same variable receives different values in the same assignment, for example in (i, i) = (2, 3). As pointed out above, in ASL the leftmost assignment takes effect. In spite of this rather unavoidable breakdown, the use of temporaries makes ASL closer to the intuitive idea of simultaneous assignment than it would otherwise be.

In the expansion of f(x) = y, one might possibly expect an assignment that pre-evaluates the left-hand side, as follows:

```
t1 = f          *t4 = t1(t2)
t2 = x            t1(t2) = t3
t3 = y            IF SET(t1) THEN f = t1 ELSE x = t2
```

However, the assignment t4 = t1(t2) (indicated by *) is not included, as it is usually superfluous and would, counter to intuition, invoke f in a dexter call for this case.

The general rule is that in the assignment $e_L = e_R$, if e_L is of the form e1(e2), we do not assign a temporary to this outermost operation e1(e2). However, expressions within e1 and e2 are pre-evaluated and stored in temporaries. If e_L is of the form (e1, e2, ..., en), we go further and do not assign a temporary to (e1, e2, ..., en), and then (recursively) consider e1, e2, ..., en to be outermost operations. Map assignments in general are handled similarly to vector assignments.

As an example of this rule, the assignment

```
(x, f(x), (y, g(h(z)))) = v
```

expands as:

```
t1 = x                t1 = t8(1)
t2 = f                t2(t3) = t8(2)
t3 = x                (t4, t5(t9)) = t8(3)
t4 = y                IF SET(t5) THEN g = t5 ELSE DO
t5 = g                  t6(t7) = t9
t6 = h                  IF SET(t6) THEN h = t6 ELSE z = t7
t7 = z                  END
t8 = v                y = t4
t9 = t6(t7)           IF SET(t2) THEN f = t2 ELSE x = t3
                      x = t1
```

The vector assignment above must be further expanded in the usual way.

Multiple assignments are handled similarly to vector assignments. For example,

```
a(i) = i = 2
```

expands as:

```
t1 = a                t3 = 2
t2 = i                t1(t2) = 2
t3 = i                i = t3
                      IF SET(t1) THEN a = t1 ELSE i = t2
```

In summary, the use of temporaries elevates the level of ASL by bringing it closer to one's intuition with respect to:

1. Parallel evaluation of expressions (section 4.10.2)
2. Pseudo bound variables (section 4.10.2)
3. Sinister composition of functions
4. Map and multiple assignments

In addition, a few optimization possibilities are introduced. One is that expressions are more likely to qualify for parallel computations on a machine of suitable architecture (such as the System/360 Model 195). Another related one is that expressions are more likely to be reorderable, as was pointed out in section 4.9.2. Consider also the evaluation of

$$\{x, \forall x \in S \mid f(x) = a + 1\}$$

where f is a procedure and a has the SHARED attribute. Because of the ASL use of temporaries, an optimizer can factor out the expression a + 1 from the loop, even without any knowledge of f. Without temporaries, f could possibly change a on each iteration, preventing the optimization.

5.13 GO TO Statement

The GO TO statement is written either GO TO s, or GO s, or TO s, where s is a positive integer expression. Control then transfers to statement number s, which is called the "target" of the GO TO.

It is recognized now that this statement introduces some of the worst problems in a programming language. If it is attempted to minimize the restrictions on the target of a GO TO, the language designer must make some rather arbitrary decisions regarding the meaning of a GO TO in a recursive environment. The language implementor may have to provide for abruptly altering the program's "environment" in a way that can only be determined at run time. The optimizer must assume worst case upon encountering a transfer in which little or nothing is known about the target. Perhaps worst off of all is the program reader, who must be aware that the side effects of a GO TO may cause practically all of the variables to change in value (due to a change in recursion level). More commonly, he is given a program whose flow of control is so intricate he can hardly flowchart it, let alone understand it.

Consideration was given to not having a GO TO statement, forcing ASL procedures to be highly structured in blocks that are always entered at the top and exit at the bottom. Until that programming style is more widely accepted, however, ASL employs the compromise of having a

restricted, and relatively simple, GO TO.

The GO TO is restricted in two ways:

(1) the target must be in the procedure containing the GO TO, and

(2) a statement group or a loop can only be entered at the top.

The first restriction helps in the top-down study of a program. When one encounters a reference to a procedure, one knows immediately that the procedure is certain to return to the point of invocation (or terminate). Thus one can follow the flow of a program precisely without having a detailed understanding of the procedures it calls. This restriction also helps readability by keeping a procedure more self-contained.

Because ASL does not have the "begin ... end" type of environment stacking, the first restriction implies that the GO TO is only a simple transfer of control without environment altering side effects. Thus it is natural to define the GO TO in a recursive environment as meaning nothing more complicated than it means in a non-recursive environment.

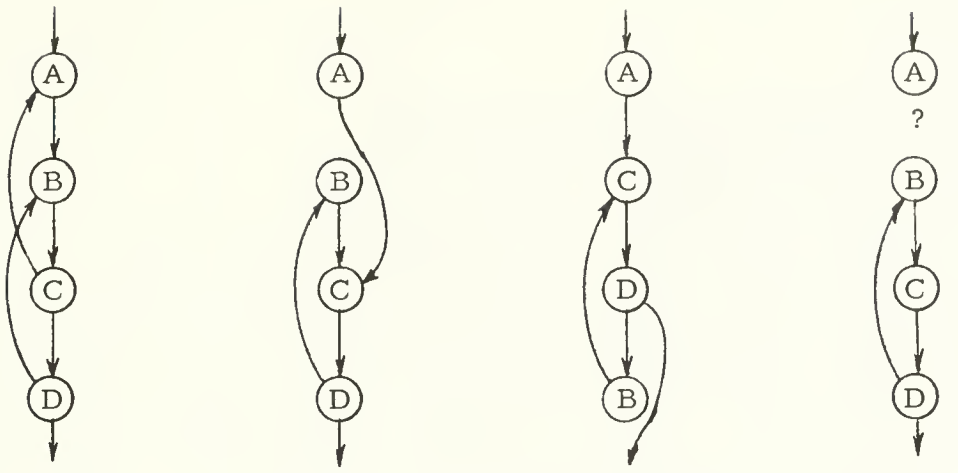
The second restriction, that statement groups and loops can be entered at the top only, is primarily aimed at reducing the occurrences of the spaghetti bowl structures that one encounters from time to time. The main point is to eliminate the branch into the middle of a loop or a THEN clause, for example, from the outside. This type of branch is fairly easy to give up, and giving it up will go a long way toward encouraging the coding of programs with a more fathomable structure.

The second restriction applies even if the loop is "hand coded"

using IF's and GO TO's. Thus the equivalence between the ASL iteration and its unraveling (which is given in section 6.5) is strictly maintained. This restriction is enforceable only with a fairly sophisticated compiler.

The second restriction is also intended to simplify and enhance the possibilities of global optimization of ASL procedures. For example, a well-formed ASL procedure is completely reducible without node splitting, in the sense of optimizers that use the concepts of the interval and the derived graph. In fact, the ASL restriction is stronger than necessary to prevent the necessity for node splitting, as illustrated by the graph (a) below. This structure is outlawed, although it is reducible without node splitting.

It might be reasonable to relax restriction (2) slightly by requiring only that every loop or group have exactly one entry. This is sometimes convenient, as illustrated by case (b) below, which is topologically equivalent to (c), and readability is probably only slightly impaired. However, an optimizer would then have trouble with case (d). Here node A ends with a "GO TO v", and the optimizer has no idea what values v may have. The optimizer would then have to assume that the loop could be entered at any point, nullifying many opportunities for optimizing the loop. Hence, for the sake of optimization (and partly for readability), ASL requires that loops be entered at a readily identifiable point.



The ASL GO TO still introduces optimization problems when the possible targets are not obvious, even though the GO TO restrictions make these problems much less than they would otherwise be. The problems may be made still less severe by some feature of the elaborations, such as listing all the possible targets of a variable GO TO. It does seem worthwhile to allow a general integer expression for the "s" in "GO TO s", however, rather than to adopt the FORTRAN approach of having two particular forms of variable GO TO's (the "assigned" and "computed" GO TO's).

Some sample GO TO statements:

```
GO BACK

GO TO (L1, L2, L3)(i)

GO TO {('a', L1), ('1', L2), ('+', L3)}(char)
```

6.3 IF Header

The IF statement has the general form:

IF condl, matchl; ...; condn, matchn;

THEN (stmtl actl; ...; stmtm actm;) ELSE else_stmt

Here "condi" is a conditional expression (which must evaluate to a TRUE or FALSE result), "matchi" is a self-defining Boolean vector or a self-defining character string containing only the letters T, F, blank, and period, "stmti" is a statement (or statement group), and "acti" is a Boolean string or a character string containing only the letters X and blank.

This format is chosen to allow a form that closely resembles decision tables, such as:

```
IF      x ε R,              'FTT'
        f(x) ε T,           ' TF'
        ∀y ε S : y > 0,     ' T '
THEN (x = 0;                'X  '
        x = y;              ' X '
        R = R - {x};        '  X'
        GO TO L1;           ' XX')
ELSE  GO TO L2;
```

The bracketing of the THEN unit may be done with any of the statement grouping symbols, [...], DO ... END, or BEGIN ... END.

The IF is executed as follows (ignoring the usual use of temporaries, which of course applies). First all the vectors "matchi" are evaluated, in order of increasing i. Any character strings are converted to corresponding Boolean vectors, by replacing 'T' with TRUE, 'F' with FALSE, and leaving blank and period positions undefined.

The resulting set of vectors can be regarded as a matrix. The columns of the matrix, which are called "rules", are examined in left-to-right order, starting with column 1. Each column is matched with the conditional expressions, taking the components in top-down order. If a component in a column is undefined, then the corresponding conditional expression is not examined. If the component is defined, then the conditional expression is evaluated, if necessary, and compared to the match component. If they are equal, the next component in the rule is tested. If they are not equal, then the current rule does not apply, and the next rule is tested.

The unraveling of the IF statement is shown below.

```
tmatch1 = MAKEBOOL(match1)
...
tmatchn = MAKEBOOL(matchn)
tcond1 = tcond2 = ... = tcondn = ;
1 ⩽∀j ⩽MAX(HI tmatch1, ..., HI tmatchn) DO
   IF ∃tmatch1(j) THEN (IF ¬ ∃tcond1 THEN tcond1 = cond1
                        IF tcond1 ≠ tmatch1(j) THEN CONTINUE ∀j)
   ...
   IF ∃tmatchn(j) THEN (IF ¬ ∃tcondn THEN tcondn = condn
                        IF tcondn ≠ tmatchn(j) THEN CONTINUE ∀j)
   "Got a match on column j."
   IF act1(j) = 'X' THEN stmt1
   ...
   IF actm(j) = 'X' THEN stmtm
   GO out
   END ∀j
   else_stmt
out:;
```

The match matrix (or "condition entry") must uniquely identify the rule that applies. This restriction is imposed to improve the readability of programs. It also simplifies the application of optimizations that depend

upon permuting the columns of the decision table. That is, two columns may be interchanged provided the order of evaluating the conditions is not altered, but it is not necessary to determine whether or not the two columns might in some cases both apply.

Whether or not this rule is observed cannot be determined from the match vectors alone. For example, the following is valid:

```
IF      x ⩾ 0,  'T '
        x < 0,  ' T'
THEN ...
```

Hence the compiler cannot always tell whether or not the unique rule restriction is observed. For this reason and for the sake of run-time efficiency, the action of the program is left unspecified if the unique rule restriction is violated.

The reasons for specifying that the conditional expressions cannot in general be pre-evaluated, and must instead be evaluated as needed, are the same as the reasons for the left-to-right evaluation of Boolean expressions: (1) to allow the programmer to gain efficiency by placing lengthy but frequently not required expressions near the end of the list, and (2) (more importantly) to allow decision tables such as the following:

```
IF      ∃x,         'FTT'
        f(x) > 0,   ' FT'
THEN ...            'X  '
```

A match vector and its preceding comma may be omitted, in which case a vector of all T's is assumed. Similarly, if there is only one "action" statement, then its action vector may be omitted, in which case a vector

of all X's is assumed. The ELSE clause is also optional. Thus the following are valid IF statements:

```
(1)   IF     ∃x; x > 0; c1(x),  'TTFF'
             c2(x),             'TFTF'
      THEN (y = x;              'X  X'
             z = x;             ' XX ')

(2)   IF c THEN s1 ELSE s2

(3)   IF c THEN s

(4)   IF c THEN; ELSE s
```

In case (4), a semicolon is required after the THEN, to denote a null statement.

The ELSE clause is always matched with the innermost unmatched IF (as in PL/I). Thus the statement:

```
IF c1 THEN IF c2 THEN s1 ELSE s2
```

means:

```
IF     c1,   'TT'
       c2,   'TF'
THEN (s1;    'X '
       s2;   ' X')
```

and not:

```
IF     c1,   'TF'
       c2,   'T '
THEN (s1;    'X '
       s2;   ' X').
```

The latter interpretation results from:

```
IF c1 THEN IF c2 THEN s1 ELSE; ELSE s2.
```

Like simple IF statements, decision table IF statements may also be nested. However, the resulting constructions are even more grotesque

than nested simple IF statements. Below is an example.

```
IF      c1,                'TTF'
        c2,                'FTF'
THEN (IF c3,    'TF'
         c4,    'F '
   THEN (s1;    'X '
         s2;    ' X');     'X X'
        s3;                'XX ')
```

Such nesting may always be removed. For example, the decision table above is logically equivalent to the one below, but in the absense of an optimizing compiler, the one below would probably be slower in execution.

```
IF      c1, 'TTTFF'
        c2, 'FFTFF'
        c3, 'TF TF'
        c4, 'F   F '
THEN (s1; 'X   X '
        s2; ' X   X'
        s3; 'XXX   ')
```

6.5 Iteration Header

ASL provides only one iterator, but it is of a general form that combines the three SETL iterators and adds a few embellishments. Although this section is oriented toward the iteration of statements, the same iterators are used for statement repetition, the set former, and the vector former. The overall syntax of a statement iteration is an iteration header followed by either a single statement or a statement group. Parentheses around the iteration header itself are optional, although there are cases where parentheses are necessary around either the header or a single-statement iterand, to correctly specify what is intended.

The general form of an iteration is:

STARTING $stmt_1$ for-specification WHILE cond DOING $stmt_2$ $stmt_3$

This may be regarded as a sequence of five clauses. The first four form the iteration header, and are independently optional. The last clause ($stmt_3$) is the iterand, and it is required to be present (although it may consist of a null statement).

Any of the stmt's above may be either a single statement or a statement group.

The overall approach taken in the ASL iterators is first of all that the stepping variables are free variables: they are of significance outside the iteration. This is often of use in search loops, and it seems to be necessary in view of the fact that a stepping "variable" may actually be any value receiving expression.

On exit from an iteration, whether by a go to, quit, or fall-through, the stepping expression is set at the last used value. If the loop is never executed, the stepping expression is not changed from the value it had before the iteration.

An attempt is made to give a reasonable interpretation to changing the stepping variable and the range of an iteration, in the iterand. The PL/I approach of first evaluating as much as possible outside the loop is not adopted. That is regarded as a slight compromise in the expressivity of a language for the sake of efficiency, and in ASL we opt for the expressivity. Many of the cases where various expressions (such as the upper limit of an incrementing variable) can be hoisted out of the loop will be recognized by known optimization techniques. (It is an oversimplification to write off the problem in this way, and the main reason ASL tends to opt for expressivity rather than efficiency is simply that the intended uses of ASL are different from those of PL/I).

6.5.1 STARTING Clause

Loop initialization statements are generally placed here. The word STARTING is in most contexts merely a noise word that serves only to aid the human reader.

6.5.2 For-specification

The for-specification consists of a list of "for-items" separated by commas. Each for-item is distinguished syntactically by the presence of the ∀ character, and has one of the following three forms:

1. Itemized iteration: ∀x = list | C(x).
2. Counting iteration: Any of the twelve varieties of expressions such as m ⩽∀x ⩽n | C(x), m > ∀x⩾ n | C(x), ∀x⩾ n | C(x), etc.
3. Set iteration: ∀x ε S BY next | C(x).

In the above, x denotes any value receiving expression, m and n are integer expressions, S is a set expression, C is a conditional expression, and next is a function expression. The conditional expression and its stroke symbol may be omitted if it is simply TRUE.

We will first discuss these three for-specifications without regard for the conditional expressions.

6.5.2.1 Itemized Iteration

In the itemized iteration, the "list" is a sequence of expressions separated by commas. The ellipsis may be used. Some valid itemized iterations:

```
∀i = 1, 2, 4, 8
∀i = 1, 3, ..., 99, 100, 98, ..., 2
∀x = 1, N+1, 'a', {1, 2, 3}
∀(i, j, k) = (1, 1, 2), (1, 2, 1), (2, 1, 1)
∀i = 1, 2, ...
```

The iteration "∀i = e1, e2, ..., en stmt", where the ellipsis is not actually used, is essentially equivalent to:

```
i = e1; stmt; i = e2; stmt; ...; i = en; stmt.
```

However, the generated code will actually correspond to:

```
        k = 1;          "k is a compiler temporary."
        i = e1; TO b;
a(2):   i = e2; TO b;
        ...
a(n):   i = en;
b:      stmt;
        k = k + 1;
        TO a(k);
a(n+1):
```

When the ellipsis is used, as in ∀x = e(u), e(v), ..., en stmt, where u and v are numeric expressions, the iteration is equivalent to the rather formidable expansion given below (we assume here that en is not of the form e(w), and hence e and en must be numeric expressions. See section 4.5.)

```
        k = u
        tv = v
        d = tv - k
        t1 = e(k)
        up = e(tv) - t1 ⩾ 0

back:   t2 = en
        IF up & t1 ⩽ t2 ∨ ¬up & t1 ⩾ t2 THEN DO(
          x = t1
          stmt
next:     IF up & x ⩾ t1 ∨ ¬up & x ⩽ t1 THEN DO "advance."
            k = k + d
            t1 = e(k)
            IF up & x < t1 ∨ ¬up & x > t1 THEN GO back
            ELSE TO next "(x was changed in stmt)."
            END
          ELSE DO "regress (x was backed up by stmt)."
            k = k - d
            t1 = e(k)
            TO next
            END
          )END
```

If en is omitted, then so is the assignment t2 = en and the IF ... THEN header at the statement labeled "back".

When en is of the form e(w) (with w a numeric expression), the expansion is simpler, mainly because changes to x within "stmt" are ignored in this case (it would be hard to do otherwise, as e(u) is not necessarily numeric). The expansion of ∀x = e(u), e(v), ..., e(w) is:

```
        k = u
        tv = v
        d = tv - k
back:   tw = w
        IF d ⩾ 0 & k ⩽ tw ∨ d < 0 & k ⩾ tw THEN DO
          x = e(k)
          stmt
          k = k + d
          GO back
          END
```

These "unravelings" are of course intended to point out such details as the value of the stepping variable on exit from the loop, and the effect of changing the stepping variable or the limit while iterating. The fact that the stepping variable is not set if the loop is not executed is intentional, as it is consistent with the straightforward treatment of nested iterations (to be discussed), and with the action of the set iteration (also to be discussed) when the set is empty.

Care should be taken if the itemized iteration with the ellipsis is used, and the sequence is not monotonic. For example, the iterand is never executed in:

∀i = (-1)**2, (0)**2, ..., 25.

However, it would be if the 25 were written (5)**2.

6.5.2.2 Counting Iteration

The counting iteration $m \leqslant \forall x \leqslant n$, where m and n must be numeric expressions, is equivalent to:

t = CEIL(m); ∀x = t, t+1, . . . , n.

The form ∀x> n is equivalent to:

t = FLOOR(n)+1; ∀x = t, t+1, . . .

and similarly for the other forms.

The counting iterator is provided because when it can be used, it is more concise and often easier to read than the corresponding itemized iteration. This is particularly true when the starting value is a lengthy expression. Compare, for example, $2*(m-n) \leqslant \forall i \leqslant e$ with ∀i = 2*(m-n)+0, 2*(m-n)+1, . . . e.

Because of the temporary t introduced, counting iteration has a different meaning than the corresponding iteration of the form $\forall i = e_1(m), e_1(n), \ldots, e_2$ if the variables defining e_1 change while iteration is in progress.

6.5.2.3 Set Iteration

The set iteration ∀x ε S BY next stmt is equivalent to the following:

```
      t = ;
back: t = next(t, S)
      IF ∃t THEN (x = t
                  stmt
                  t = x
                  GO back)
```

Here "next" is a map on S which, when given a set S and a member $t \varepsilon S$, produces the "next" member of the set. If t is not defined, "next" produces the "first" member. If t is the "last" member, next has no value. The map "next" may be either an itemized map or a procedure.

The "BY next" clause may be omitted. In this case, the predefined procedure variable NEXT is used. This variable is initialized, at procedure entry, to the "standard" set iterator. The ASL user may redefine NEXT as an alternate means of specifying his own ordering over sets. However, he is limited to the basic unraveling that has been given; it can only be avoided by hand-coding the type of loop control desired. Changing the value of NEXT would affect all set iterations in the procedure for which no other user-supplied routine is given, including those in set formers, etc.

For general sets, the standard order is arbitrary (not specified). If the set is an array, the order of iteration is lexicographic (last subscript varying most rapidly) on the first component of each pair in the set. This is referred to as the "natural" order. Vectors are handled similarly.

For a map, such as an array, vector, or string, the stepping variable is a pair. Thus an expression such as "$\forall$(i, c)ε string" is common, as one usually wants the components i and c separated. Some other forms of iteration over itemized maps are:

$\forall(i, *) \varepsilon$ v	Iterates over the domain of v.
$\forall(*, r) \varepsilon$ v	Iterates over the range of v (with repetitions).
$\forall((i, j, k), a) \varepsilon$ A	One way to iterate over a three-dimensional array.

Set iteration is defined in such a way that both the stepping variable and the set being iterated over maybe changed during the iteration. However, the action of the standard routine NEXT is in these cases not specified, except for certain special cases to be discussed below. This is due in part to implementation difficulties and in part to the fact that there does not seem to be a natural meaning of iteration in these cases.

Consider the case of iterating over a general set (that is, a non-array). Iteration is then defined to be a pass over the members in an arbitrary order, which might not even be the same from one case to the next over the same set. Iteration normally continues until all members have been used exactly once.

An idea of the difficulties encountered can be obtained by considering the following questions: (1) If a member has been used as the stepping variable, and it is then deleted from and added to the set, should it be used again? (2) Similarly, if a member has been used and a copy of it is added to the set (which doesn't actually change the set), should it be used again? (3) What is the meaning of changing the stepping variable within the iterand? Does it mean to skip over or repeat a range of members? Or should just the new value be skipped over, assuming it is in the set? Or should the change be ignored?

In addition to the above implied difficulties, some representations of sets can undergo drastic changes when a member is added or deleted.

This complicates the design of an efficient "next" algorithm that can tolerate such changes.

If the set is an array, the situation is different. In this case, NEXT searches the new value of the set for the next member in lexicographic order, starting with the new value of the array's indexes. As an example, the following iteration over a matrix has the effect of "backing up" to rescan the current row from the beginning (assuming it is one-origin):

$$\forall((i, j), m) \in M[\ldots\ j = 0;\ \ldots].$$

This has the unraveling:

```
        t = ;
back:   t = NEXT(t, M)
        IF ∃t THEN [((i, j), m) = t
                    ... j = 0; ...
                    t = ((i, j), m)
                    GO back]
```

To make a transformation on each member of a set, one cannot in general write (for example) "$\forall x \in S\ (x = x + 1)$", or "$\forall x \in S\ (x \notin S;\ (x+1) \in S)$". These are valid expressions, but their meaning is not defined when the standard "next" routine is used. One can accomplish the transformation only by the more awkward "STARTING $T = \emptyset$ $\forall x \in S\ ((x+1) \in T);\ S = T;$".

6.5.2.4 For-specifications: Conditional Modifier

Any of the three forms of the for-specification can have a "conditional modifier", or "such that" phrase appended on the right. This is separated from the left part by the stroke symbol. For example:

```
∀x = a, b, c | (x-1)**2 > 0
1 ⩽ ∀i ⩽ N | f(i) = 0
∀(i, c) ε string | c ≠ ♭
```

The meaning of a specification such as "∀x ε S BY next | cond stmt" is:

```
      t = ;
back: t = next(t, S)
      IF ∃t THEN (x = t
                  IF cond THEN stmt
                  t = x
                  GO back)
```

On exit from the iteration, the iteration variable x is set to the last value obtained from S, which may not have actually been used in the iterand. If S is null, then x is not changed; however if the iterand is not executed because the condition is always false, then x is set to the last value obtained from S.

It might seem better to move the assignments x = t and t = x into the THEN clause of the IF cond. Then x would not be set unless it were actually used in the iterand, and on exit it would indicate the last value actually used. However, it is debatable which operation is preferable, and the one given seems to be necessary in view of the fact that "cond"

may involve x or, worse yet, "parts" of x. This is illustrated by the iteration over a matrix M which skips the main diagonal:

```
(∀((i, j), x) ε M | i ≠ j) stmt;
```

which expands as:

```
        t = ;
back:   t = NEXT(t, M)
        IF ∃t THEN [((i, j), x) = t
                    IF i ≠ j THEN stmt
                    t = ((i, j), x)
                    GO back]
```

6.5.3 WHILE Clause

The WHILE clause may be used to specify when an iteration is to end. The statement "WHILE cond stmt" expands as:

```
back: IF cond THEN [stmt; GO back] ELSE GO out; out:
```

The reason for phrasing it in this way (with an apparently purposeless ELSE clause) will be apparent in section 6.5.5 below.

6.5.4 DOING Clause

The DOING clause may be used to specify a block to be inserted after the iterand. It provides a convenient way to make iteration controlling steps clearly visible, by placing them in the iteration header.

The statement "DOING $stmt_1$ $stmt_2$" expands as simply "$stmt_2$ $stmt_1$". It is useful in iterations such as "STARTING x = 0 WHILE ∃x DOING x = f(x) statement".

6.5.5 The Full Single Iteration

For the case of set iteration, the expansion of:

STARTING $stmt_1$ $\forall x \in S$ BY next | C_1 WHILE C_2 DOING $stmt_2$ $stmt_3$

is shown below.

```
        stmt1
        t = ;
back:   t = next(t, S)
        IF ∃t THEN DO
          x = t
          IF C1 THEN[
            IF C2 THEN (stmt3 stmt2)
                  ELSE GO out]
          t = x
          GO back
          END
out:    ;
```

Note the difference between C_1, which causes skipping of members, and C_2, which causes loop termination.

The full expansions involving itemized iteration and counting iteration are similar.

6.5.6 Multiple Iterations

An arbitrary number of single iterations can appear, separated by commas, in the same iteration header. This creates a nested iteration. The general form of a two-level nested iteration is:

STARTING $stmt_1$ for-$spec_1$ WHILE C_1 DOING $stmt_2$, ...

STARTING $stmt_3$ for-$spec_2$ WHILE C_2 DOING $stmt_4$ $stmt_5$;

The expansion, using simple set iterators $\forall x_1 \varepsilon S_1$ and $\forall x_2 \varepsilon S_2$, is:

```
          stmt1
          t1 = ;
back1:    t1 = NEXT(t1, S1)
          IF ∃t1 THEN[
             x1 = t1
             IF C1 THEN[

                stmt3
                t2 = ;
back2:          t2 = NEXT(t2, S2)
                IF ∃t2 THEN[
                   x2 = t2
                   IF C2 THEN (stmt5 stmt4)
                         ELSE GO out2
                   t2 = x2
                   GO back2]

out2:              stmt2]
             ELSE GO out1
             t1 = x1
             GO back1]
out1:     ;
```

If the iterand is never executed because, for example, one of the set expressions S_i evaluates to the null set, then x_1 through x_{i-1} are arbitrary members of S_1 through S_{i-1}, and the remaining x_i are not changed by the iteration.

An iteration such as "$\forall x \varepsilon R \mid C_1$, $\forall y \varepsilon S \mid C_2$" is in general different from "$\forall x \varepsilon R$, $\forall y \varepsilon S \mid C_1 \& C_2$", because C_1 might be a function of y.

Only the itemized form of a for-specification allows a "multiple iteration" that is not nested. That is, to step a single variable over two ranges or over two sets, one cannot write "$1 \leqslant \forall i \leqslant 10$, $20 \leqslant \forall i \leqslant 30$", or

"$\forall x \varepsilon S_1$, $\forall x \varepsilon S_2$". One must instead write something like "$\forall i = 1, 2, \ldots,$ 10, 20, 21, ..., 30" or $\forall x \varepsilon S_1 \cup S_2$ (if this is intended). The first expressions are valid, but they won't do what is presumed to be intended. (Note that in "$\forall x \varepsilon S_1$, $\forall x \varepsilon S_2$", NEXT($S_1$, x) will be invoked with $x \varepsilon S_2$ rather than S_1. This is a valid construction, but the action of NEXT is unspecified if $x \notin S_1$).

6.5.7 ITERATION Pseudo-function

It is sometimes desirable to have available a count of the number of times an iteration has been done, even though the iteration may not be of the counting type (see, for example, function p in section 4.1.3.2). This may be obtained by using the ITERATION pseudo-function, which avoids the petty details of initializing a counter, incrementing it, thinking up a name for it, and commenting it.

The "argument" of the iteration pseudo-function is obtained from either the for-specification or the WHILE specification of the iteration. Any number of tokens, starting with the $\forall$ or the word WHILE, may be used. If the argument does not uniquely identify the iteration, the ITERATION applies to the innermost one that matches. The argument may be omitted, in which case it applies to the containing iteration. If the argument includes parentheses or character string self defining values, they must be closed.

The ITERATION pseudo-function is only valid within an iteration (for example, one cannot branch out of a loop and refer to ITERATION

to see how many times the loop was executed). The for-specification conditional, WHILE conditional, DOING statement, and the initial expressions in a set former or vector former are all considered to be within the loop. For example, the first 20 members of a set may be converted to a vector by arbitrarily ordering the members by:

$$\{(\text{ITERATION}, x),\ \forall x \varepsilon S \text{ WHILE ITERATION} \leqslant 20\}$$

(the simpler [x, $\forall x \varepsilon$ S WHILE ITERATION $\leqslant$20] accomplishes the same thing).

6.5.8 Iteration Summary and Examples

General Form

STARTING $stmt_1$ for-specification WHILE cond DOING $stmt_2$ $stmt_3$

For-specifications

1. (Itemized) $\forall x = list \mid C(x)$
2. (Counting) $m \leqslant \forall i \leqslant n \mid C(i)$, etc.
3. (Set) $\forall x \in S$ BY next $\mid C(x)$

Multiple Iterations

$\forall x \in S$, $\forall y \in P$
$\forall x \in S \mid C_1$, $m < \forall i \leqslant n \mid C_2$, $\forall j = 3, 5, \ldots, 19$
etc.

Uses of Iterators

1. Statement repetition: iterator statement
2. Set former: {expr, iterator}
3. Vector former: (expr, iterator)
4. Universal predicate: iterator:conditional

Examples

The set of the first 100 primes:

$\{p,$ STARTING $k = 1; \forall p \geqslant 2 \mid [2 \leqslant \forall i < p : REM(p, i) \neq 0] \ldots .$
WHILE $k \leqslant 100$ DOING $k = k + 1\}$

The first Fibonacci numbers $\leqslant 100$:

$\{n,$ STARTING $m = n = 1;$ WHILE $n \leqslant 100$ DOING $(n, m) = (m+n, n)\}$

9. Input/Output

Input/output in ASL is modeled after that of PL/I stream I/O. The PL/I record I/O is not included because that is essentially the same as reading or writing with A (alphanumeric) format, the main difference being one of efficiency. No random access I/O capability (such as the now-defunct PL/I SAVE and RESTORE statements, or a declaration to the effect that a variable's values are to be disk resident) is provided, because the highest level way to incorporate such a capability is via paging or something similar. This is the highest level possible because it requires absolutely no dictions to be coded into a program to obtain its benefits, and of course this fact is largely the reason for its success.

What is needed in ASL is a way to communicate with the external world, and for the present we concentrate exclusively on a narrow aspect of that. We concentrate on the reading in of data that has been prepared in advance of execution time, and the writing out of data so that it can be printed or otherwise displayed for human consumption.

Our input activity is always computer-initiated; that is, we do not yet address the problem of how to write programs that can be alerted by "attention" signals, or converse with several terminals simultaneously, or read instruments. These capabilities are very much needed in ASL (to code prototype real time systems), but are best planned together with multitasking capabilities, and that whole issue is being deferred.

9.1 Files

Files are referenced by a character string expression. The value of the expression is the same as the name used for the file in the job control language.

We think of the file as corresponding to an I/O device. The I/O device is basically typewriter-like, although it is capable of handling a few simple page-oriented functions such as starting a new page (on output). The typewriter-to-computer connection consists, in principle, only of a data path, which may be one character wide.

We defer the ASL design for dealing with control characters such as backspace, line feed, delete, etc., and assume that the typewriter has only one control character: new-line (we consider "space" to be a print character). However, the underlying I/O routines in some installations may employ such control characters. For example, they might use horizontal tab for high-speed spacing, and overprinting to improve the appearance of output (e.g., printing the null set ∅ as zero, backspace, slash). The point is that these control characters are not (at present) explicitly available to the ASL programmer without going out of the language.

The new-line character is used as follows. If the input device is a typewriter (an uncommon case since reading is always computer-initiated), depressing carrier return (which is presumed to also cause a line feed) inserts a new-line character into the data stream. This

signals the computer that the line has been completely composed and processing of it may begin. When cards are being read, the underlying system routine inserts a new-line character into the input stream between each two cards. There is no particular "record size" associated with input or output; in ASL we prefer the term "line", and lines are of varying and arbitrary length.

On output, the new-line character is normally inserted into the output stream by explicit instructions in the ASL program. However, if the program attempts to write a line that is too long for the device, then a new line will be started at the appropriate point. Exactly how this is accomplished depends on the I/O device, but in any case it is not a burden on the ASL programmer.

There are several built-in procedures that may be used to reference various file attributes and states. These are:

PAGESIZE(file)	COLUMNNUM(file)
LINESIZE(file)	ENDINFO(file)
PAGENUM(file)	ERROR(file)
LINENUM(file)	

PAGESIZE and LINESIZE return the maximum number of lines and the maximum number of characters per line for the specified file. These functions may be used in sinister mode to set the parameters. PAGESIZE may only be used for output or unopened files (if not yet opened, PAGESIZE causes it to be opened for output). If the file has margins (m1, m2) (see below), then LINESIZE returns m2 - m1 + 1.

PAGENUM, LINENUM, and COLUMNNUM return the current "cursor" position for the named file. The cursor position is the position to be used for the next data character, except in the case that the last character was printed in the last column, in which case COLUMNNUM (file) is LINESIZE(file)+1. The reason for this has to do with the meaning of the CR (carrier return) control format item, which is used for over-printing.

ENDINFO(file) may only be used for input files. It returns TRUE if it was attempted to read past the end of information point, and FALSE otherwise. If it is attempted to read a file that has reached the end of information point, the data items are given the undefined state.

There is an ambiguity in the meaning of end of information that may occasionally be troublesome. ENDINFO means the file is definitely at the end; all characters have been read (or skipped over), and furthermore an attempt was made to read more data. If a file is unopened, or positioned so that all that remains are blanks and comments, then ENDINFO(file) is FALSE. This is because the following characters may be read with A (alphanumeric) format. However, if the subsequent READ operation uses simple (formatless) I/O, then the data items will be made undefined, as for this type of READ there is no more information. This operation will not normally be troublesome, as one normally tests for end of information after a READ and before it is attempted to use any of the data items.

The ERROR function returns a very crude indication that some sort of error occurred in a READ or WRITE operation. Its value is the undefined state for a file that has not been opened (and it may be used as a test for this condition), and otherwise is:

0: last READ or WRITE was normal

1: software error occurred

2: hardware error occurred

A "software error" results if the data did not agree with the format, for either input or output. A "hardware error" is a parity error, timeout, not-ready state, etc., which the I/O system is unable to correct. The precise state of the I/O device and software is not yet defined for either of these errors (e.g., where is the cursor after a software error?).

The ERROR routine may be invoked in sinister mode, provided the right-hand side is 0, 1, or 2. Also, if an I/O operation is attempted while the file's error state is nonzero, the run is terminated. The normal use of these properties is to allow checking for an error, presumably doing something about it, resetting the error status to zero, and resuming I/O operations. On the other hand, a program that does not check for errors will be terminated if one occurs, which is probably desirable. One might set the error code to 1 or 2 for debugging reasons.

An alternative design in common use is to have the error routine reset its status to zero when invoked (in a right-hand context). This

design was not chosen for ASL because of a general distaste for side effects. In ASL a program with the following structure will work:

```
IF ERROR(f) = 1 THEN DO ... END
ELSE IF ERROR(f) = 2 THEN DO ... END
```

This would not work with the side effect approach, because the first call to ERROR would reset it to zero. Also, in ASL an optimizer may factor calls to ERROR in cases such as the above.

There are four I/O statements: READ, WRITE, PAGE, and FORMAT. The PAGE statement sets certain page formatting data associated with the file. The other I/O statements have roughly the PL/I - FORTRAN meaning.

9.2 PAGE Statement

The general form of the PAGE statement is:

```
PAGE FILE f MARGIN(m1, m2) AT n1 stmt1 ...
                           AT nn stmtnn
```

All clauses are optional. The file f is a character string expression. Its default is the standard print unit, which is installation defined (e.g., 'SYSPRINT', or 'OUTPUT', etc.).

The margins m1 and m2 are positive integer expressions, with $m1 \leqslant m2$, that give the left and right margins, respectively, for reading or writing. On reading, data before character m1 or after character m2 on each line are ignored. On writing, m1 - 1 blanks are inserted at the beginning of each line, and a new line is begun if it is attempted to write into column m2 + 1. The defaults for m1 and m2 are installation and device dependent. For example, they might be (1, 72) for the standard input unit and (1, 130) for the standard output unit. (Column one is not used for format control in ASL. Hence, for some operating systems, such as SCOPE, our "column 1" will actually be column 2 on the printer but column 1 on the card punch. There is no way in ASL to directly exploit this column 1 convention.)

The AT clauses are used to specify actions to be taken when the specified line number is reached. For example, a page heading on the standard output unit can be provided with the statement:

```
PAGE AT 1 WRITE 'HEADING' FORMAT(A).
```

The ni must be distinct integers such that 1 $\leqslant$ni $\leqslant$PAGESIZE(f), where PAGESIZE(f) is the page size of the file for which the PAGE statement applies. The AT statement, which may be a statement group, is executed just before the character that causes the internal line counter to become equal to n is acted on. A page-eject causes the AT-statements for all passed-over lines to be executed, because page ejection is (conceptually) executed as a series of new-line and blank characters.

The line number expressions ni are evaluated at the time of execution of the PAGE statement.

The AT-statement, or statement group, may not contain a GO TO out of the AT-group. It is executed as a parameterless subroutine in which all variables are SHARED with the containing routine. The AT-groups are enabled when the PAGE statement is executed, and are disabled when the subroutine containing the PAGE statement returns (at the recursion level in which the PAGE statement was executed).

After executing a PAGE statement, another procedure may be invoked that performs some I/O operations. The AT-statements defined back in the calling procedure then are executed when the appropriate line numbers are reached. The AT-statements may alter various variables including some that are shared with the procedure that initiated the I/O operation. This is a situation similar to PL/I interrupts, with its attendant optimization problems: an I/O statement may cause access (set or use) of variables that are not explicitly given in the I/O statement. However, the situation is not as bad as that in PL/I, because the flow of

control is simpler: the AT-group can only return or terminate execution.

An AT-group may, however, be entered recursively, and it may contain a PAGE statement. To prevent every AT-group that initiates I/O operations on the file being processed from becoming an infinite recursion, the effect of the AT is suppressed until one data character is transmitted, and it is then reinstated.

The information in a PAGE statement is stacked if a new PAGE statement is executed for the same file after a procedure call or during the processing of an AT-group. Upon return, the old PAGE information becomes active. However, if two PAGE statements are executed by the same procedure (at the same level of control), then the new information completely erases and overrides the first. This is similar to the handling of PL/I ON-units. After a procedure call, one can be sure that the PAGE information has not been altered (except possibly by shared/external linkage to variables mentioned in AT-statements).

Input data is assumed to be on one page (which might be very long). Hence the AT-statements are of little utility here. However, they can be used on input. For example, one can guard against an excessive amount of input data without explicitly counting lines by means of:

```
PAGE FILE INPUT AT 10000 STOP.
```

The statement "PAGE" causes defaults to apply, i.e., the file is the standard print unit, the margins are the implementation-defined defaults for it, and no AT-groups are effective.

9.3 READ and WRITE Statements

ASL has three modes of I/O, which are called simple, name, and format directed (these correspond to the PL/I LIST, DATA, and EDIT directed). The general forms of the READ and WRITE statements are:

READ [data list][FILE f / STRING c] [COPY g / COPY STRING d]

[POSITION(control format list)] [NAMES / FORMAT(format list)];

WRITE [data list] [FILE f / STRING c] [POSITION(control format list)]

[NAMES / FORMAT(format list)];

The data list is a sequence of expressions separated by commas. For the READ statement, they must be value receiving expressions. The data list is optional. If absent then presumably the POSITION specification would be given, otherwise the READ or WRITE is a no-operation.

The data list must be written immediately after the word READ or WRITE. It should not be in parentheses unless the intent is to assign to, or write out, a vector.

Within the context of a READ or WRITE statement, the words FILE, STRING, COPY, POSITION, NAMES, and FORMAT are reserved words.

The brackets in the above general forms denote optional material. For items written one above the other, at most one may be selected.

The file to be read or written is given by f, which is a character string expression. This expression (and all others in the READ or WRITE statement) is evaluated each time the statement is executed. The FILE specification may be omitted; the default is the standard input unit for READ and the standard output unit for WRITE.

STRING c may be specified, rather than FILE f. If STRING is specified, no input/output operation takes place. Instead, character string c takes the place of the file. For READ...STRING c, c must be a character string expression. The I/O routines then treat c as if it were text from the input medium. Characters from c are extracted and converted, according to the format if one is given. The resulting data items are assigned to the value receiving expressions in the data list, as in a READ...FILE operation. String c may contain new-line characters, which are treated in the normal way (skipped over).

For WRITE...STRING c, c must be a value receiving expression. The data list is converted to a character string as it would be if FILE were specified, but the resulting character string is assigned to c. If the format specifies that more than one line is to be written, then c will contain embedded new-line characters. If no new line is explicitly specified in a format list (or POSITION specification), then c will not contain any new-line characters, no matter how long the string is (unless

of course a data item is a source of a new-line character).

COPY provides a convenient way to obtain a record of what was read. It may only be used for READ operations. All characters from the source file or string are transmitted to the named file or string, a line at a time, including those skipped over with control format specifications such as X, LINE, etc. Data that is reread (by using the carrier return control format item) is only copied once. If COPY is not specified, no copying is done. If COPY is specified but the file g is omitted, then the copying is done to the standard print unit. The statement READ FILE(f) COPY STRING d assigns to d a string containing the input data up to the next NL character. It is very similar to the PL/I READ FILE(f) INTO(d); (record I/O).

The POSITION specification is used to specify control format items, which are discussed below, to be acted on before data is read or written. For example, POSITION(NL, X 10) causes an advance to the next line (if not already at the beginning of a new line), and an indentation of ten spaces. On input, characters skipped over are ignored. On output, an NL character and ten blanks are supplied (the NL character has no effect if the device is already positioned at the beginning of a line). If there is only one control format item, it need not be enclosed in parentheses, e.g., POSITION PAGE.

Data is read or written in the order given by the data list until either the data list or the format list is exhausted, whichever occurs

first. If neither NAMES nor FORMAT is given in a WRITE statement (simple I/O), the current values of the expressions in the data list are written out, separated by a single space. If NAMES is specified, then each value is preceded by the expression that corresponds to it and " = ", and followed by a semicolon and a space. For example, if x is a certain character string and y is a certain integer, then WRITE x, y, y+1 and WRITE x, y, y+1 NAMES give, respectively:

```
'abc' 12 13

x = 'abc'; y = 12; y+1 = 13;.
```

FORMAT directed I/O is discussed below under the FORMAT statement.

On input, the format for simple and name I/O data is relaxed somewhat. There can be any number of spaces, and one optional comma, between items. Comments may also appear in the input stream, and they are ignored (treated as a space).

For NAMES input, the data names in both the READ data list and the input stream must be simple unsubscripted names. A sparse matrix may be read in by representing it as a set. The READ continues until all items in the data list have been assigned.

9.4 FORMAT Statement

The general form of a FORMAT statement is:

FORMAT(format list);

Normally this statement would have one or more labels, but it need not have any. The FORMAT statement is non-executable. If control passes to it, either by normal sequential flow or by a GO TO, it acts as a no-operation. It may be located anywhere in a procedure.

The "format list" is a sequence of any of the following, separated by commas:

item

n item

n(format list)

Each "item" is a format item, as described below. The letter "n" designates an arithmetic expression that is an iteration factor. It must be separated from the format item by a delimiter, generally a blank or a parenthesis. The iteration factor specifies that the associated format item or list is to be used n successive times (as long as data remains to be transmitted). The value of n must be an integer, and a zero or negative factor specifies that the associated iteration factor is to be skipped (the data list item will be associated with the next format item). The iteration factor is evaluated once for each set of iterations.

There are three types of format items: control format items,

data format items, and the remote format item. Control format items specify the page, line, column, and spacing operations. The control format items may be used in the list of a POSITION specification, as well as in a format list. Data format items specify the external forms that the data are to take. The remote format item allows format items to be specified in a separate FORMAT statement, located elsewhere in the procedure.

Control Format Items

The control format items are:

PAGE	Start new page
LINE n	Advance to line n
NL	New line
CR	Carrier return
LF	Line feed
C n	Column n
X n	Space n

The PAGE and LINE format items apply only to output files. The NL, CR, LF, C, and X format items apply to both input and output files. The only control format items valid in the STRING form of a READ or WRITE statement are the NL and X items.

In a format list, the control format items are executed as they are encountered, just prior to data transmission. When the data list is

exhausted, no further control format items are executed. The control format items in a POSITION specification, however, are executed before any data is transmitted, even if there is no data. Hence WRITE FORMAT (PAGE) is a no-operation, but WRITE POSITION(PAGE) causes the standard print unit to be positioned to the start of a page.

PAGE Format Item

This format item specifies that a new page is to be established. It is valid only on output files. The first character after starting a new page is printed at line 1 column 1. If the output unit is already positioned at line 1 column 1, then PAGE has no effect (two PAGE's in a row do not create a blank page).

LINE Format Item

The specification LINE n specifies the line number on an output file on which the next data item is to be printed. After executing LINE n, the file is positioned at line n column 1.

The expression n must be enclosed in parentheses if it is anything other than a self-defining value or a subscripted variable (LINE x+1 would be parsed as (LINE x) + 1, which is invalid). For the LINE specification to be valid, we must have n an integer such that $1 \leqslant n \leqslant$ PAGESIZE(f), where f is the applicable file.

If the file is already positioned at line n column 1, then LINE n

has no effect. If the file is already positioned after line n column 1, then it is repositioned to line n column 1 on the following page.

NL, CR, and LF Format Items

The NL format item causes the file to be positioned at column 1 of the following line; however, if it is already at column 1 then no action takes place.

The CR format item causes the file to be positioned at the start of the current line. For an output operation, subsequent characters will overprint those already on the line, if the device is capable of doing this (some displays, for example, are not). For an input operation, a line will be reread. If the last character transmitted went to the rightmost position on the line, then CR still does cause overprinting or rereading.

The LF format item causes the file to be positioned at the following line, at the current column number. On input, characters skipped over are ignored. On output, blanks are supplied. LF is equivalent to X (LINESIZE(f)), where f is the applicable file.

C and X Format Items

The specification C n causes the file position to be advanced to column n, unless it is already at that position. If n is less than the current column number, then the position advances to the following line.

The specification X n causes the file position to be advanced by n character positions. This may cause the position to be advanced by one or more lines.

For both C and X, blanks are supplied on output and characters passed over on input are ignored. The new-line character is not counted on either input or output.

The expression n must be enclosed in parentheses if it is anything other than a self-defining value or a subscripted variable. The value of n must be an integer. For X(n), we must have $n \geqslant 0$, and for C(n), we must have $1 \leqslant n \leqslant$ LINESIZE(f), where f is the applicable file. For the X specification n may be omitted, in which case 1 is assumed.

The X and NL format items are the only control format items that may be used in the STRING form of a READ or WRITE statement.

Data Format Items

A data format item describes the external (character string) form of a single data item. The data format items are:

N w	Numeric, exact
N (w, d)	Numeric, approximate, fixed point form
N (w, d, e)	Numeric, approximate, exponential form
A1 w	Single symbol
B1 w	Truth value
Q w	Pointer (output only)
P w	Procedure (output only)
S w	Set
V w	Vector

A w	Character string
B w	Boolean string
G w	General (data directed)

The data format items all have an optional field width w. This is a nonnegative integer expression that specifies the number of character positions on the I/O device (or STRING, for STRING I/O) to use for a data item. No additional spaces or other punctuation is inserted between items, in format directed I/O.

If the width w is zero on output, the format item and the associated data item are skipped (this action differs from that of a zero repetition factor, for which only the format item is skipped). On input, w may be zero only for the S, V, A, and B format items, in which case the data item is set equal to Ø.

In all cases except A format on input, w may be omitted, in which case a value appropriate for the data item is used. If w is omitted in an input operation, the string is scanned for a non-blank, non-double-quote, character, and this is taken to be the first character of the next data item. If a double quote is encountered, the scan continues, skipping characters until after a matching double quote. In other words, the input stream may contain comments in format I/O when in scan mode.

The field width w and the d and e expressions are evaluated each time the format item is used.

N Format Item

There are three forms of the numeric data format item. On output, an appropriate form must be used depending on whether the data item is exact or approximate. Some examples of output follow, where "∧" denotes a blank.

N(5):	∧∧∧∧0	∧-123	∧-3/5	12/49
N(5, 2):	∧0.00	∧0.12	-1.23	
N(9, 3, 2):	∧0.000E00	∧0.123E-1	-1.234E12	

The numbers are always converted to or from decimal. On input, the number may be anywhere in the field of width w. It may not contain embedded blanks or commas. If the entire field is blank, it is treated as an exact zero. If w is omitted, the readroutine scans for a number. The d and e fields are ignored on input; if the number is approximate it must have a decimal point. Numbers to be read in are written in the same form as a self-defining number in a program; see section 2.2.2.1. In addition, an exact rational may be entered as a ratio of integers, e.g. 3/2, -3E-3/27, etc.

On output, if the d and e parameters are omitted, then the data item must be exact. Alternatively, it must be approximate. However, a number that is approximate but is within the tolerance of an integer may be written with the N(w) format.

The parameter d specifies the number of digits to print after the decimal point, and e specifies the number of digits to print for the exponent field. The parameters must satisfy $d \geqslant 1$, $e \geqslant 1$ (if given), $w \geqslant d+2$ for N(w, d), and $w \geqslant d+e+3$ for N(w, d, e). This assures room for a decimal point, a leading digit, and the letter E in the case of N(w, d, e).

Parameter e is interpreted as a minimum. If the exponent field, with its sign if negative, does not fit within the e field, then e is effectively incremented by one, with w and d remaining the same. This is repeated until the exponent fits or the inequality $w \geqslant d+e+3$ is violated. For example, appropriate values printed with N(7, 1, 2) might print as ^1.0E99 or 1.0E100.

For a nonzero data item, the value of the exponent is always adjusted so there is exactly one significant digit to the left of the decimal point. Hence there are always d+1 significant digits printed.

The converted number is right-adjusted in the field of width w. If w is too small, truncation occurs on the left, with an * placed in the leftmost position to indicate that truncation occurred (provided $w > 0$).

A1 and B1 Format Items

The A1 and B1 format items are used for input/output of single symbols (characters) and truth values, respectively. Examples of output:

```
A1(3):      ?^^
B1(3):      T^^  F^^
B1(5):      TRUE^ FALSE
```

On input, we must have w $\geqslant$ 1. For the A1 format, the rightmost character in the field is used, and the others are ignored. For the B1 format, the field must contain a single T or F, or the words TRUE or FALSE, located anywhere in the field. If w is omitted, the read routine scans for a non-blank symbol. The data item is set equal to the symbol for A1 format. For B1, the symbol must be T or F, and the read routine skips over the next characters if they form TRUE or FALSE.

On output, if w is omitted then W=1 is assumed. For the A1 format, the single symbol is left-adjusted in the field. For the B1 format, if w $<$ 5 a T or F is left-adjusted in the field. If w $\geqslant$ 5 then TRUE or FALSE is left-adjusted in the field.

Q Format Item

The Q format item is used for printing pointer values. It may only be used for output or WRITE STRING. It would normally be used only for debugging. Examples:

```
Q(7):  ↑03677   ↑3F78E
```

The value is right-adjusted in the field of width w. It consists of an up arrow symbol followed by an installation-dependent representation of the pointer value. This would typically be a machine address printed in octal or hexadecimal.

If w is omitted, the value assumed for it is installation dependent. If w is too small but nonzero, truncation occurs on the left with an * placed in the leftmost position to indicate that truncation occurred.

P Format Item

The P format item is used for printing procedure values. It may only be used for output or WRITE STRING. It would normally be used for debugging. Examples:

P(6): MYPROC SORT^^

The value is the procedure name, left adjusted in the field of width w. Note that the procedure name does not uniquely identify the procedure, because the ASL name scoping rules allow multiple procedures with the same name.

If w is omitted, the value assumed for it is the number of characters in the procedure name. If w is too small but nonzero, truncation occurs on the right with an * placed in the rightmost position to indicate that truncation has occurred.

S and V Format Items

The S and V format items are used for the input/output of sets and vectors, respectively. Examples of output:

```
S(10):    Ø^^^^^^^^^^  {'ABCD'}^^    {1, 2, 3}^
V(10):    Ø^^^^^^^^^^  ('ABCD', )^   (1, 2, 3)^      (1, , 2)@3
```

The members of the set, or components of the vector, are in D (data-defined) format, which is discussed below.

On input, if w=0 no characters are read, and the data item is set equal to Ø. Otherwise, if w is given, then exactly w characters are read. This field must contain a set or vector written in the same manner as the non-iterative set or vector self-defining value is written in a program. The vector may be written with square brackets. The null set may be written as either Ø or { }, and the null vector as Ø, (), or [ ]. However, '' and ''B cannot be used with S or V format. If the width w is omitted, then the input stream is scanned for the first non-blank character, which must be the beginning of a valid set or vector.

On output, the set or vector is left-adjusted in the field. If w is too small, it is truncated on the right and the rightmost position is made an *.

The S format may be used for vectors, character strings, etc., and it causes them to be transmitted as sets. For example, after

```
x = 'ABC'
WRITE x FORMAT S
```

the item written is {(1,'A'), (2,'B'), (3,'C')} . Similarly, the V format may be used for character strings and Boolean strings, and it causes them to be transmitted as vectors. This applies to input, output, and the STRING option.

A and B Format Items

The A and B format items are used for input/output of character strings and Boolean strings, respectively. Some examples of output of strings of lengths 0, 2, and 3:

A(3):	^^^	A^^	ABC
B(3):	^^^	1^^	101

On input, if w=0 no characters are read, and the data item is set equal to ∅. For the A format, w must be given. Exactly w characters are read, which may be anything in the data character set, and the quote-doubling convention is not used. NL characters in the input stream are skipped over, as always, and do not form part of the string. The resulting string is of length w.

For input under B(w) format, the w characters must consist of a dense string of 0's and 1's, which may be located anywhere in the field. The resulting string length is the number of 0's and 1's found, which may be zero. For B format, w may be omitted, in which case the input stream is scanned for the first non-blank character. This must be a 0 or 1, and scanning continues until a character other than 0 or 1 is encountered.

On output, for B format the Boolean string is first converted to a character string of 0's and 1's of the same length as the Boolean string. Then, for either A or B format, the string is left-adjusted in

the field and transmitted to the output stream. No indication of truncation is given.

G Format Item

The G format item causes an item to be transmitted in the same manner as in simple I/O except that a field width w may be given, and no space or other punctuation is inserted between items. Examples of output:

```
G(5):   ^^123    ^1.23    1.4E9    '?'^^   TRUE^
        *7700    SUB^^    {1}^^    (1,)^   'ABC'  '11'B
```

On input, the string is scanned for an item, skipping over blanks and comments. If the item is a single character or character string, it must be delimited by single quote marks, with each internal quote mark indicated by two single quote marks. Boolean strings must be written with surrounding quotes and a suffix B, as in the example above. Numeric data, sets, and vectors are written as they are in a program. Pointer and procedure values may not be read in. The width w may be omitted, but it is an error to have $w \leqslant 0$ or to have $w > 0$ with an entirely blank field.

On output, the data item is converted to a character string, as in simple I/O. For approximate numeric data, either fixed point or exponential format is chosen, whichever shows more significant digits.

There is a maximum number of significant digits that ever results, which is implementation dependent. The maximum is used if w is omitted.

The left- or right-adjustment and truncation indication is provided, as in the case of the other format items.

Remote (R) Format Item

The remote format item is written R n, or R(n), where n is an integer expression for a line number of a FORMAT statement in the procedure containing the R format item. The effect is the same as if the referenced format list were inserted at the point of the R format item. The expression n is evaluated each time the R format item is encountered.

As a simple example of its use, suppose a data line or card image contains a 0 (or blank) or a 1 in column 1 to indicate which of two formats to use for the following data item. Then the card could be read as follows:

```
    READ I, X FORMAT (NL, N 1, R(I+1))
L:  FORMAT (G)
    FORMAT (A 71)
```

Here column 1 is read in, assigned to I, and used to select either format L or L+1 for the data item X. (This could also be accomplished by reading column 1, testing it with an IF statement, and then reading the rest of the line with either of two READ statements.

This is possible because a READ does not necessarily advance the file position by one line).

Matching Data Items to Format Items

Data items are matched with format items in the same way as in PL/I. The first item is evaluated (if it is an output operation). The format list is then scanned from left to right, executing any control format items encountered. If a repetition factor is encountered, it is evaluated and scanning continues as if the item or list replicated were written repeatedly the specified number of times (i.e., 3(NL, A) acts like NL A NL A NL A). When a data format item is encountered, the data item is transmitted from the input stream, converted, and assigned to the data item (READ) or converted and transmitted to the output stream (WRITE). This continues until either the data list or the format list is exhausted, whichever occurs first.

If the data list has two or more items and is enclosed in parentheses, it becomes a single item (a vector).

Data lists may not have repetition terms such the DO specifications of PL/I.

Examples

Let v = ('a', 13, TRUE). Then

WRITE v POSITION(NL)	prints	('a', 13, TRUE)
WRITE v FORMAT(NL, G)	prints	('a', 13, TRUE)
WRITE v(1), v(2), v(3) FORMAT[NL, A1, N(3), B1(6)]	prints	a 13 TRUE
WRITE v(1), v(2), v(3) FORMAT[NL, 3 G]	prints	'a'13TRUE
WRITE v(1), v(2), v(3) FORMAT[NL, 3(X, G)]	prints	'a' 13 TRUE
WRITE v(1), v(2), v(3) FORMAT[3(NL, G)]	prints	'a' 13 TRUE

The PL/I:

PUT EDIT (X(I), Y(I) DO I = 1 TO 10) (10(COLUMN(1), F, X, F))

may be coded in ASL:

(1 $\leqslant \forall$I $\leqslant$10) WRITE X(I), Y(I) FORMAT(NL, N, X, N)

The statement READ I, A(I) will read in a value for I, and then use that value for indexing A (if A is a map or is undefined) or for a sinister call to A (if A is a procedure).

13. Using the Compiler

13.1 Input and Listing Control Commands

There are a few instructions that affect how the compiler reads source text and prints the program listing. These are summarized below.

MARGIN(i, j); For each line read from this point on, pass only columns i through j to the compiler (with a new-line character appended after column j), but print the whole line. The parameters i and j are unsigned integer constants.

LISTING(ON); (or OFF) Enables or suppresses the listing of the source text.

EJECT; Eject to a new page, if not already at the top of a page.

SKIP(n); Skip n lines (or to a new page if that comes first). The parameter n is an unsigned integer constant.

INDENT(ON); (or OFF) When printing, adjust columns i through j so that each statement starts on a new line, and indenting is used to reflect the structure of blocks, IF's, explicit loops, etc.

OVERPRINT(ON); (or OFF) Use overprinting to more closely approximate the ASL characters. For example, print ¬= as ≠, NULL as ∅ (using the EBCDIC character set), etc.

LOWERCASE(ON); (or OFF) Change all non-keywords to lower case before printing and before compiling (columns i through j only).

NEATER(ON); (or OFF) Same as INDENT + OVERPRINT.

15. Differences Between ASL and SETL

15.1 Summary

Considering the whole spectrum of computer languages, or even limiting the view to the procedural algebraic languages, ASL is very similar to SETL. The goals and intended users (professional programmers, including mathematicians and analysts) are certainly the same.

The differences stem mainly from an increased emphasis on readability, and a closer alignment to standard mathematical practice. By "standard mathematical practice" I place more emphasis on the kind of mathematics used in science and engineering, and less on that found in a subject such as the algebraic theory of languages or the foundations of mathematics, where deviations from conventional notations are more apt to occur.

ASL pays more attention to numerical work than does SETL, although this does not detract from its expressivity in non-numerical work. Although non-numerical work is on the increase, relatively, I believe that computers will be finding sines and cosines and inverting matrices for a long time to come, and SETL ought to support numerical work at least to the extent that PL/I does. This is necessary for prototyping many large programs, such as command and control systems.

Another theme that leads to differences between ASL and SETL is that an attempt has been made in ASL to keep the language simple and

free of paradoxes. These will be mentioned below as they arise.

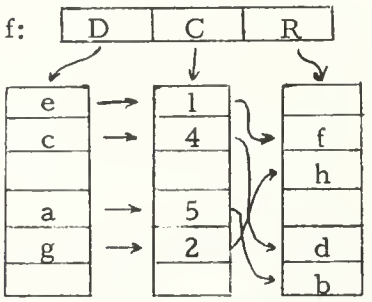
15.2 Syntax

The main change to the syntax is that ASL employs many more symbols, there being 58 special characters in ASL and 26 in SETL. This is an attempt to bring the notation closer to standard mathematics and to improve readability. SETL suffers on both counts by using the same symbol for different things much too often. For example, + in SETL denotes addition of numbers, string and tuple concatenation, and set union. ASL uses +, ¢, and $\cup$ for these three quite distinct operations.

Employing a larger alphabet not only improves readability but also enhances the possibilities for the optimization of minimizing run time type checking by inferring data types. For example, after A = B + C, one knows little about the data type of A in SETL. But in ASL it would be known that A is numeric or a map (although it may be integral, rational, or floating point). Furthermore, the amount of type checking required is less in ASL, because each operation is valid for fewer data types.

Besides these matters of readability and optimizability (which so often seem to go hand in hand), the enriched syntax may affect the way one uses the language. For example, ASL employs the symbols $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{I}$ for domain, range, and inverse. SETL has no such symbols; instead of $\mathcal{D}S$ one would write $\{x(1),\ x \in S\}$. Originally $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{I}$ were suggested

simply because these are very common operations in mathematics, particularly $\mathcal{D}$. However, in my limited experience with ASL so far, I find I use $\mathcal{D}$ and $\mathcal{R}$ even more often than I had anticipated. It seems that just because they are there and are so easy to write, one thinks in terms of them and uses them often. They probably should be given special attention in the implementation. For example, in contexts where copying is not required, $\mathcal{D}$ and $\mathcal{R}$ could be almost as fast as #S (which, it is assumed, merely retrieves a precalculated number) with a representation of functions as follows. A function is a triple (D, C, R) where D is the domain, R is the range, and C is a correspondence between members of D and members of R. D and R could be stored as standard sets, for example hash tables. C would then be a vector of integers that serve as pointers into R's hash table. To illustrate, the function f = {(a, b), (c, d), (e, f), (g, h)} might be stored as:



This scheme would not work for the "multiply defined functions" of SETL, but ASL does not permit functions to be multiply defined (this is discussed below). ASL does, however, allow $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{L}$ to be applied to binary relations, and for them another scheme would be required.

ASL employs a full set of precedence rules, similar to that of PL/I. SETL has essentially only three precedence classes.

The use of special characters in ASL is given in section 2.1.1. We mention here a few of the main differences.

The SETL eq, lt, etc. is replaced by =, $<$, etc. In ASL these can be strung out, for example $A < B \leqslant C$ means $A < B$ & $B \leqslant C$. The SETL and, or, not is replaced by &, $\vee$, $\neg$. ASL includes symbols for all the nontrivial Boolean functions of two variables, e.g. $\not\&$ for nand. The SETL nl, nult, nulc, and nulb is replaced by $\emptyset$, as these are all the same in ASL. There is no symbol in ASL for the SETL Ω. To test for being undefined, one writes $\exists x$ instead of x neΩ. In ASL vector concatenation is written v1 ¢ v2, rather than v1 + v2 (in both languages the same symbol is used for vectors and strings, but in ASL there's a compelling reason: strings are vectors). In ASL the origin of a vector may be shifted by the expression v @ i. This has no counterpart in SETL. Both SETL and ASL denote substrings by s(i:j). However, in ASL j denotes the upper limit, whereas in SETL it denotes the length. The ASL way is consistent with the way we code DO-loops and the mathematician's $\sum_{i=m}^{n}$, and I think it is preferable independently of this.

ASL writes set intersection as S1 $\cap$ S2, rather than the SETL S1*S2. ASL includes the four set relations S1 $\subset$ S2, S1 $\subseteq$ S2, etc., whereas SETL uses the symbols $<$, $\leqslant$, etc. for these (there is for some reason also incs in SETL).

The SETL iterators $\forall x \in S$ and $m \leqslant \forall i \leqslant n$ are present in ASL, and in ASL there is one that has no counterpart in SETL: the ellipsis. One

writes, for example, ∀i = 2, 4, 6, ... In ASL the ∀ symbol is written in the iterator of set and vector former expressions. SETL drops it in these contexts and has no vector former of the iterative type. The SETL vector former <x, y, z> is written (x, y, z) in ASL. Square brackets in ASL are equivalent to parentheses; hence the vector may also be written [x, y, z]. In SETL the brackets have special significance, and are used only in the contexts f[S] and [op: x ε S]e(x). The SETL notations f{x} and f[S] are not present in ASL. The notation [op: x ε S]e(x) also has no counterpart in ASL, but this situation should be rectified. Perhaps the APL-like notation op/v, where v is a vector, would be reasonable for ASL. By combining this with the vector former, one could write, for example, +/[e(x), ∀x ε S | C(x)].

ASL has the property that any value that can be calculated can also be written as a self-defining value (or "constant"). SETL misses this by the arbitrary exclusion of sparse tuples, i.e., <x,Λ, z> is invalid in SETL, but would be written (x, , z) in ASL. This kind of completeness is convenient at times in ordinary coding, and is important when writing programs that write other programs. It also allows the unification of the syntax of self-defining values for programs and for data to be read in by a program (which would presumably allow sparse vectors).

In ASL keywords are written in capitals, whereas in SETL they are in lower case and many of them are underlined. Capitals were chosen to make keypunched ASL close to published ASL. An algorithm reproduced

from a computer printout can be published and it will look almost as good as one that was typeset, but will be a little more believeable. This opinion may be shortsighted, because small letters are gradually being used to a greater extent in computing, and possibly before long virtually all data entry devices and printers will handle them. But the underlining in SETL strikes me as very unrealistic, like the bold face type of Algol.

In ASL statements are written without a terminating semicolon, unless there is more than one on the same line. The end of a line is marked by a character that is treated like a semicolon (it may be effectively cancelled by preceding it with four periods).

Comments are written in ASL between quotation marks, e.g., "comment", whereas in SETL it is /*comment*/. Comments should be delimited by as few characters as possible, to encourage their use and to improve appearance. ASL requires two characters for delimiting, whereas SETL requires four, or six if you like a space separating the delimiters from the comment. ASL has the disadvantage that by forgetting a quotation mark the rest of the program is inverted with respect to what is a comment and what is to be compiled, but I don't think this is a significant objection. That's the kind of annoyance that happens once in a while, and you correct it very quickly and easily. The same problem exists with character string self-defining values in both languages.

The procedure headings in ASL are FUNCTION and OPERATOR; in SETL they are "define" and "definef". The SETL "define" is used for

subroutines and "definef" for functions and operators. ASL allows only binary user-defined operators, and allows the user the option of specifying the precedence (which must be the same left and right). SETL allows unary (prefix) and binary user-defined operators, and does not allow the user to specify precedence. In SETL user-defined operators are denoted by underlining, or a trailing period in any presently possible implementation. The same notation is used for some built-in operators, e.g. abs, min. In ASL user-defined operators are delimited by periods, e.g. .OP., and .+., in both the "official" and (presumably) any implementation of the language. No built-in operator is written this way (except in implementations lacking a sufficient character set), and hence a reader knows immediately whether an operator is built-in or user-defined. The reason ASL does not allow user-defined prefix operators is that it is often hard to remember whether something that is spelled out with letters was defined as a prefix operator or as a function --- is it .OP. x or OP(x)? In SETL confusion exists even with the built-in operators, as there is log x and sin(x). Also, in ASL it is not necessary to put parentheses around a single argument; sin(x) and sin x are equivalent (as is sin((x)), (sin) x, etc.).

This brings us to the ASL use of parentheses. They denote grouping only. Grouping expressions separated by commas is understood to denote a vector. The notation (x) is the same as x as long as x is a single "expression". It is assumed that the parser will use the parentheses to force the structure of the parse tree, but will not put any indication of the

parentheses in the tree. This is of course done before statements are recognized. Hence one could put parentheses around the word IF of an IF statement, but one cannot write (IF x = y) THEN . . . , as this alters the structure of the IF statement. SETL has a more conventional use of parentheses, in which they must be put around the arguments of a function reference, around a "while" header, and a few other places in the language.

The ASL IF statement follows PL/I in that the THEN and ELSE clauses are single statements or groups. The groups may be delimited by DO. . . END, BEGIN. . . END, or by parentheses. The SETL "if" has the "then" clause terminated by the word "else" if present, and the whole "if" is terminated by an extra semicolon, or by "end;", or "end if;". This in itself is not too bad, but I do find it a little unnatural to have an extra semicolon when there's only one statement in the "then" or "else" clause. Moreover it precludes the possibility of a null statement in the language, which is sometimes convenient to have, particularly when a preprocessor is involved (ASL allows the null statement). But the worst thing about the SETL "if" is that it follows the Algol style, approximately (unlike Algol, SETL allows "then if"). That is, if an "else if" appears, then only one extra semicolon terminates the whole "if". For example, the statement "if C1 then a = b; else if C2 then c = d;;" is valid SETL. Now consider how to write the PL/I

```
IF C1 THEN A = B;
ELSE DO; IF C2 THEN C = D;
         E = F;
         END;
```

This cannot be expressed in SETL without resorting to parentheses around statements. The attempt "if C1 then a = b; else if C2 then c = d; e = f;;" fails because the "e = f" is under both C1 and C2 rather than only C1, and the attempt "if C1 then a = b; else if C2 then c = d;; e = f;" fails because the "e = f" is under neither C1 nor C2. This is a very confusing situation to encounter in practice. Although IF statements nested more deeply than about two deep are confusing no matter how you write them, it is my feeling that the block-oriented structure of PL/I and ASL is much more easily fathomed than the strung-out structure of Algol and SETL.

Statement iterators are terminated by an extra semicolon in SETL, but in ASL the iteration is understood to apply to a single statement or statement group. A more significant syntactic difference is that SETL has basically two iterators: the "while" and the "for all" loops. ASL combines these into a single one (following PL/I) in which the "while" and "for all" parts may both be present, e.g. $1 \leqslant \forall i \leqslant \#S$ WHILE $s(i) \neq b$ DO ... END scans a string s until either the end is reached or a blank character is encountered. There are semantic differences between the SETL and ASL iterators, which are discussed below.

In both ASL and SETL something has been done to raise the expressivity of the language for complicated decision processes. SETL has the unique "flow" statement. ASL suggests instead the use of decision tables, and also includes a CASE statement. The flow statement is quite popular with most programmers at NYU who have used it, but I find it not very

appealing. To explain why, I should first point out that there are two ways to use the flow statement. One is to place the decisions and actions directly in the flow tree, and the other is to refer to them indirectly using labels. The trouble with the former is that one soon runs out of space on a line, and one is limited to decision processes with only three or four actions. For these the if-then-else is not too bad, and the flow statement hardly seems worth having. The other style is to use labels. Usually they are heavily used, and the flow statement was designed primarily with this in mind. One can then code decision processes with typically eight or ten decisions and five or six leaves on the tree, before running out of space. The trouble with this is the indirectness itself: the coder is forced to think up a lot of trite label names, and the reader has trouble following through the tree and at the same time scanning the text below it to find the decisions and actions.

Another problem with the flow statement, besides the indirectness and the lack of room, is that many decision processes are simply not tree-like. The flowchart joins together after spreading apart, for instance. Or there may be a small number of "actions", but various combinations of the actions are to be done in various cases. The flow statement does not help at all in this situation. An attempt is made in SETL to make up for the logical lack of expressivity by adding various gimmicks to the basic flow statement. Thus one can place actions in amongst the nodes, including a branch to another node in the tree, detail decisions with the subflow

option, etc. But all this is insufficient: if used with restraint it is inadequate, and if used freely one risks creating an unfathomable maze.

The decision table, on the other hand, is at its best when various actions are to be executed in combinations according to the outcome of the decisions. For sufficiently complicated situations, the decision table is superior to a carefully drawn flowchart, complete with boxes and arrows. Hence in these situations it is far superior to the flow statement, which emulates the flow chart but is essentially restricted to a tree-like structure. Good examples of decision tables may be found in Appendix I of the SCOPE operating system manual. Here there are several tables describing the action of certain low level I/O routines for operations on magnetic tape. The decisions are recording format (standard binary, standard coded, X binary, etc.). The actions are to exit if insufficient room in the buffer for the maximum size record, to read one record into PP memory, to process end-of-file if applicable, etc. A few moments contemplation of these decision tables reveals how superior they are to flowcharts. By comparing columns one can readily grasp the similarities and differences between the handling of the different tape formats. Reading across a row reveals precisely for which formats a certain action is executed. No flowchart does this.

With decision tables it is less often necessary to resort to indirectness, as most of the line (and subsequent lines if necessary) is available for coding each decision and action. It is also never necessary to code a decision or an action more than once, unless order of execution is significant.

The main trouble with decision tables is that they don't begin to pay off until the logic gets very complicated. It is too much trouble to study columns and rows when there are only three or four decisions and actions. I think this is the main reason why decision tables have never really caught on: in most fields of programming we seldom encounter situations sufficiently complicated to require them. This includes numerical work, compiler writing, combinatorics, etc. One field that does benefit greatly from decision tables is I/O programming, both at the interrupt processing level and at the higher level of buffer management, device dependent actions, and possibly something like format-directed conversion routines. I understand that another area is "business programming", although I have no experience here. In fact, it is only in business programming that decision tables have received significant attention.

On reflection it seems possible that decision tables are only useful in "real world" problems, and not so much in programming with a more mathematical flavor. I include I/O in the real world category, particularly when it is device dependent. If it is true that decision tables are not useful in problems with a "mathematical flavor", then they will never catch on in a university. But it would be incorrect to conclude that they have no place in SETL just because SETL has such a mathematical style. This is because SETL is a general purpose language, and building a prototype of an I/O interruption processor or an inventory management system is within its intended range of application.

Getting back to the flow statement, it is possible that there is a gap between what is reasonable for the if-then-else and what is reasonable for decision tables, that can be filled by the flow statement. If this is the case, then I would suggest toning down the flow statement by omitting the gimmicks referred to above.

Another syntactic difference between SETL and ASL is that in SETL declarations follow the FORTRAN and Algol style, whereas in ASL they follow the PL/I style. Declarations play a small role in both SETL and ASL; they are found mostly in the "elaboration language". But the difference I am getting at is that FORTRAN, Algol, and SETL encourage the grouping of all variables with a given attribute, whereas PL/I and ASL encourage the grouping of all the attributes associated with a given variable. Contrast, for example,

```
INTEGER A, B                DECLARE A(10) FIXED EXTERNAL;
DIMENSION A(10), B(20)      DECLARE B(20) FIXED EXTERNAL;
COMMON A, B
```

The PL/I style is, in my opinion, definitely superior, because it does not force the reader to search through a large body of declarative material to find all the declared attributes of a given variable. This is something you want to know more often than, for example, all the integer variables in a program. The PL/I style requires more writing, generally, but this is not a serious objection, and it is minimized by factoring of attributes (which ASL also allows), for example DECLARE (A(10), B(20)) FIXED EXTERNAL.

Some of the built-in operators and functions of SETL have been omitted in ASL in the interest of simplicity. These are listed below, with an indication of the ASL substitute.

SETL	ASL
S with x	$S \cup \{x\}$
S less x	$S - \{x\}$
x in S;	$x \in S$;
x out S;	$x \notin S$;
x from S;	$x = \exists S$; $x \notin S$;
hd x	x(1)
tl x	x(2:)
dec, oct, bitr	use "string" form of I/O
S lesf x	$\{\forall t \in S \mid t(1) \neq x\}$
f{x}	$\{p(2), \forall p \in f \mid p(1) = x\}$
f[S]	$\{p(2), \forall p \in f \mid p(1) \in S\}$
is	no counterpart

There are a few features of SETL that have not been mentioned for ASL but which would be available in the form of library routines. This includes pow, npow, and compile. It is also suggested that ASL be provided with a library of math routines similar to that of PL/I, so that one has available ATAN(x), SINH(x), etc.

15.3 Data Types

A fundamental change is that ASL attempts to simplify the structure of SETL by minimizing the number of distinct concepts in the language. For example, the number of data types has been reduced, and the same class of expressions may be used in any value receiving context.

The data types of both languages are:

SETL	ASL
integer	number
real (floating point)	character
character string	Boolean value
Boolean string	pointer
label	procedure
subroutine	set
function	
blank atom	
tuple	
set	

Both languages have floating point quantities (of unspecified precision) and arbitrarily large integers. ASL also has rationals. ASL takes the view that these are all merely numbers, but some values are known precisely and some only approximately. ASL allows mixed mode arithmetic, whereas SETL does not. The ASL treatment of numbers is closer to mathematics, and gets away from the artificial and frequently

annoying distinction between floating point and integers. On the other hand one can, without difficulty, express approximate numerical calculations in ASL and thereby avoid huge rationals by having the implementation employ (presumably) floating point.

In place of SETL's character strings and Boolean strings, ASL has the single character and the single Boolean value. One forms strings of these by forming dense, one-origin vectors of characters or bits. This approach, which follows Algol 68, is done largely for aesthetic reasons, but I think it also elevates the level of the language somewhat. On the aesthetic side, it just makes sense that if one can have a string of objects one ought to be able to have the objects themselves. SETL is inconsistent in that the notation s(i) denotes the i'th component of s if s is a tuple, but it denotes a substring (of length one) if s is a string. Put another way, s(i) = s(i:1) for strings but not for tuples. On the more practical side, consider the question "Does SETL allow sparse strings?" That is, after s = 'abc', is s(2) = Λ a valid assignment? If one searches the manual long enough and is lucky, he will find that the answer is "no". This is an arbitrary restriction with no basis in fundamentals; it is probably based on efficiency considerations. In ASL, however, one would immediately know that the answer is "yes", because strings are vectors, which are maps, and a map may have any domain.

In SETL one can create an n-tuple of character strings of length one. Such an object behaves just like a character string for the purposes of indexing and substring referencing, except that it may be sparse. It occupies more space, but indexing it would be a faster operation on some machines because of the lack of unpacking. Someone is sure to discover this fact and write a program that deals with n-tuples of characters, and find that his program won't mesh with a part written by someone else, or it is cluttered with conversions from one type of object to the other. In ASL this type of difficulty is less likely to occur.

ASL does not have the label data type. Instead, labels are "declared constants" whose values are line numbers (integers). This is done merely to save a data type, and to avoid defining how labels print out, whether or not they may be put in sets, what happens on recursion (does their value somehow change, as in PL/I?), etc.

In place of SETL's subroutine and function data types, ASL has only the procedure. ASL does not distinguish between subroutines and functions by data type, and the operator-procedure may not be a data value.

In place of SETL's blank atom ASL has the pointer data type. This permits the straightforward description of algorithms involving "control blocks", which is awkward to do in SETL. The pointer may also be used as a blank atom: the SETL x = newat; may be written x = ↑0; (any expression whose value is defined could be used). It is possible that ASL should have the blank atom as well as the pointer, merely to

avoid this unnatural use of pointers.

An important difference between SETL and ASL is that in ASL the tuple is not a distinct data type. A tuple (which is called a vector in ASL) is regarded as a map from integers to other objects (the components), and it is represented as a map, i.e., a set of ordered pairs. This change is suggested for a number of reasons. One is that it is simply aesthetically appealing to be able to reduce all data structures to a single data type. Of course one should give it up if it leads too far away from what one would expect. This was the case in an early version of SETL, in which the pair $\langle x, y \rangle$ was defined to be $\{\{x\}, \{x, y\}\}$, as is sometimes done in mathematics (the n-tuple was defined as right-nested pairs, i.e., $\langle x, y, z \rangle = \langle x, \langle y, z \rangle \rangle$, etc.). However, this definition of a pair was later rejected because it is not useful in helping one to think about his data structures, it does not simplify the language beyond the fact that it eliminates a data type, and it would be a nuisance to implement the capability of being able to switch from one representation to another, when this capability would practically never be used.

Furthermore, the above trick, found in studies of the foundations of mathematics, does not reflect what a vector really is. What it is, in my opinion, is a map from integers to arbitrary objects. If this is in fact what it is, then it ought to be formally treated as such in a very high level programming language.

The present SETL definition of a tuple as a distinct data type is not too bad, but the ASL definition, a set of ordered pairs each of whose

first component is a distinct integer, has the advantage that the set operators automatically apply to vectors in an obvious way. The most important is of course the selection of the image of x under f, denoted by f(x). As another example, #v gives the number of defined components in vector v. In SETL the # operator also applies to tuples as well as sets, but through a separate definition. One must decide how it is to work with sparse tuples, and in SETL #v is arbitrarily defined to be the index of the highest defined component of v. ASL uses the unary function HI(v) (highest index of v) for this, and also includes LI(v) and LENGTH(v), all of which are absent in SETL.

An operation such as set union on vectors is not very useful in coding, and in fact it might be regarded as strange and hence undesirable that ASL allows it (SETL does not). However, it can be useful in matters of definition. For example, vector concatenation is defined as v1 ¢ v2 = v1 $\cup$ (v2 @ HI(v1)+1). The @ operator causes "shifting" of the vector so that the first component of v@i is numbered i. The above definition concisely shows exactly how concatenation works with vectors that are other than one-origin.

This brings us to another difference between SETL and ASL. In SETL tuples may only be one-origin. However, since they can be sparse, it might be said to allow i-origin for i $\geq$ 1. But it was not really intended to have tuples of other than one-origin, and the others are not supported to any extent. For example, LI(v) is missing, concatenation with an

i-origin vector on the right works in a way that implies that it is really one-origin (exactly i-1 undefined slots are left between the two parts of the result), and an i-origin vector is printed with i-1 "undefined" indications at the beginning. In ASL a vector may be i-origin for any integer i (quite possibly subject to an implementation restriction, such as $|i| < 2^{32}$). This capability is found in PL/I and Algol, and I think it is used often enough to justify its existence, particularly zero-origin. In ASL one may have arbitrary-origin and sparse character strings and Boolean strings (although strictly speaking they are not called "strings"), features that are absent from PL/I and Algol.

Since in ASL the character string and Boolean string are vectors of single characters and single bits, set operations such as f(x) and #x automatically apply to these objects in an obvious way. In SETL they require a separate definition.

In SETL one may of course have a set of pairs whose first components are distinct integers. Such an object is called a "sequence", and it behaves very similarly to a tuple. So similarly, in fact, that it may be a disadvantage to have two such similar objects, because it presents an opportunity for programs to fail to operate together.

ASL formally recognizes the "array", whereas SETL does not. An array is a map whose domain consists of equal-length regular vectors of integers. An array is not a distinct data type in ASL; it is merely recognized in that there is a declaration to that effect in the elaboration

language, they are iterated over in a certain order, a few built-in operators apply to arrays but not to maps in general, and they would be stored in the usual way in which the indexes are not explicitly stored, in a substantial implementation. In ASL the matrix, vector, and string are in the same category as the array, in that they are all merely formally recognized sets of a certain structure, rather than distinct data types.

15.4 Expressions

15.4.1 Built-in Operators

In ASL several of the built-in operators are extended to apply to maps of certain types. Standard mathematical practice is usually followed, although it is generalized slightly. For example, it is usual practice to define the sum of two vectors or matrices to be the sum of their corresponding components, provided the extents are the same. In ASL we generalize this slightly by permitting maps in general to be added, provided their domains are the same. Addition is then applied recursively to the corresponding members in the range of the maps; hence one can add arrays of vectors, etc., provided the domains match up correctly.

Multiplication of a map by a scalar follows the usual mathematical practice. For vector and matrix multiplication ASL uses the inner product and standard matrix product. Division of matrices is allowed, based on calculating inverses, but division by vectors or other maps is not allowed. Exponentiation also follows the usual practice (however,

ASL does not allow matrices to be raised to fractional powers). The absolute value notation is also used for the "norm" of a map, which in ASL is the Euclidian length of a vector. (Some have taken issue with ASL's confusing these two concepts. However, in my opinion they are very similar, as a norm is anything that satisfies $\|x\| > 0$ for $x \neq 0$, $\|0\| = 0$, $\|cx\| = |c| \cdot \|x\|$ for numeric c, and $\|x + y\| \leqslant \|x\| + \|y\|$, and absolute value satisfies these relations).

ASL includes the factorial, whereas SETL does not.

The comparison operators $<$, $\leqslant$, etc., are restricted in ASL to apply to numbers (as in mathematics, usually). In SETL they have been given peculiar meanings for Boolean strings, and are used for subset, etc., for sets.

In SETL the Boolean operators and, or, etc. apply to Boolean strings on a bit parallel basis, with the shorter operand being left-extended with zeros. In ASL they apply to maps of bits, or maps of maps of bits, etc., but the domains must match. I believe that zero-extending is of marginal utility, half the time it will be done on the wrong end, and it is best avoided anyway because it leads to paradoxes such as DeMorgan's law failing.

Unlike SETL, in ASL the Boolean expressions are evaluated from left to right, and evaluation stops as soon as the result is apparent. For example, "$m \neq 0$ & $a/m < 5$" is valid in ASL.

ASL has two pointer operators, $\uparrow x$ and $\downarrow x$, which SETL of course

does not have.

The set operators (union, membership test, etc.) work the same in ASL and SETL; only the syntax has been changed.

As was already discussed, ASL has the $\mathfrak{D}$, $\mathfrak{R}$, and $\mathfrak{l}$ operators, which are absent (and in fact unprogrammable) in SETL. As was also mentioned, ASL has the @ operator for changing the origin of a vector or array, whereas this is absent in SETL. Vector concatenation works slightly differently in ASL (when the vector on the right has missing initial components).

15.4.2 Function Referencing

The use of a set as a map is a key concept in SETL. The SETL approach is that any set is a potential map. The notation f(x) means to search f for an n-tuple, $n \geqslant 2$, whose first component is x. The result is its second component, if n = 2, and the vector of its remaining components, if $n > 2$. The result is undefined if zero or two or more n-tuples begin with x.

The ASL approach is much more restrictive. For a set to be used as a map it must consist entirely of pairs and no two pairs may have the same first component. This is the approach usually used in mathematics, and I think it is better because it is more paradox free, it leads to more transparent algorithms, and it permits a simpler and more efficient implementation of functional application.

One paradox of SETL is that it is impossible to identify the domain, range, and inverse of a function. This is because if the function includes the triple $\langle x, y, z \rangle$, it is not known whether it maps x to $\langle y, z \rangle$ or x, y to z, since it can be used either way. This has the practical consequence that one cannot write algorithms that deal with functions in general. As a simple example, besides domain, range, and inverse, one cannot code a routine that calculates the "product" of two functions. In ASL this can be done, and in fact is so simple it hardly deserves to be a procedure: $\{(x, f(g(x))), \forall x \in \mathcal{D} f\}$.

Another paradox of SETL is that two functions for which a domain is defined in some way may have the same domain, and for every member of the common domain its image may be the same, and yet the functions do not compare equal. This is because the functions may contain non-tuples.

In ASL all function references involve a single argument. In the notation f(x, y), (x, y) is a vector. This is true whether f is an itemized map or a procedure. SETL employs the more conventional approach in which the argument list is not a single object. The ASL approach is a language simplification in that (x, y) denotes a vector no matter what the context, and linkage conventions will probably be simpler. It also leads to a natural way to code procedures that apparently can be called with missing arguments, or with an arbitrarily large number of them.

ASL includes the "sinister" (left-hand side) calls of SETL. The semantics are essentially the same, but in ASL the meaning of left-hand composition of functions is derived rather than postulated. This provides a specification of how they work in the presence of side effects, which SETL does not pin down.

ASL allows the sinister call f(x) = y with f undefined, whereas SETL does not. This is important because it, together with the fact that in ASL there is no distinction between the null set, null tuple, null character string, and null Boolean string, avoids what is essentially a bug in SETL.

Consider building up a map by assignments such as A(1) = y. In SETL A would first have to be initialized to either nl, nult, nulc, or nulb, because otherwise there is no way to tell which type of object to create for A (the right-hand side can be used to eliminate either the character string or the Boolean string as a possibility, but that is all. After A(1) = 'a' we could have A = 'a', A = <'a'>, or A = {<1, 'a'>} if A were not initialized). In ASL if A is undefined we treat A(1) = y the same as if A were initialized to ∅. This works because the null set, null vector, etc., are all the same. So far this is merely a convenience.

Now suppose we are building up a more complicated structure and the first assignment is (A(1))(1) = y. The meaning of this is:

```
t = A(1)
t(1) = y
A(1) = t.
```

In SETL, initializing A to a null object is inadequate because t would then be undefined and the second assignment would not be allowed. A must be initialized to <nult> or {<1, nulb>}, or some such thing.

But the problems do not end here. Suppose the initial assignment is (A(m))(1) = y. Then A would have to be initialized to <nl, nl, ..., nl>, or some such thing, which can be most inconvenient if there is no reasonable bound on m. This, I think, is a bug in the present SETL definition. It is hard to fix because one cannot simply say that in f(x) = y, if f is undefined it is treated as null, because you don't know which null object to pick. In my opinion this situation offers concrete evidence that the ASL treatment of vectors and strings is the "right" one --- or at least more right than SETL's.

In ASL right-hand side function and operator references never change their arguments. In addition there is, formally, no such thing as a subroutine-procedure. There are only function- and operator-procedures. However, the effect of a subroutine call (something that may change arguments) is achieved by the minor syntactic trick of treating the statement "expr;" as equivalent to "expr = TRUE;", where expr is any value receiving expression. Hence sub(x); is equivalent to sub(x) = TRUE;, which is a sinister function reference and hence may cause a change to x. If "sub" is only used as a subroutine, then it would presumably ignor the right-hand side.

With this gimmick one can also set a switch to TRUE by simply

writing "x;". A few built-in operators can be called in sinister mode, for example writing x ε S; is equivalent to x ε S = TRUE; which causes the membership test routine to put x in set S.

15.4.3 Other Expressions

ASL includes a set former that is essentially the same as SETL's, but a larger variety of iterator clauses may be used. For example, in ASL we can write {x, ∀x = 2, 4, . . . , n} and {e(x), ∀x ε S WHILE C1(x) | C2(x)}. The use of a WHILE clause in a set former is not of very much utility, but it is permitted merely to have the same iteration expression valid in all contexts.

ASL also includes a vector former of the iterative type, which SETL does not. For example, if S is a character string then (c, ∀(i, c) ε S | c ≠ b) is the same string with blanks deleted and characters shifted to the left accordingly.

This example points out another difference: in ASL any value receiving expression may be used as the iteration variable. Also, iteration over strings is treated like set iteration, but it proceeds in left to right order.

Both ASL and SETL include existential and universal quantifier expressions, which are similar. However, the ASL existential expression does not have the side effect of assigning a value to the iteration variable if the result is "true". Admittedly this is often convenient in SETL,

but in ASL we prefer not to have hidden assignments.

A point in ASL which I am not sure is good or bad is that in the set former the iteration variable is not a bound variable. It is in some ways like a bound variable and in some ways like a free one. The interested reader is referred to section 4.10.2.

ASL includes a "search expression", which is written like an existential or universal predicate without the $\exists$ or $\forall$ symbol, and the value of the expression is the object found (rather than TRUE or FALSE). For example, "y = x ε S : C(x);" assigns to y a member of S that satisfies C(x), if there is one. In SETL this would be coded "y = $\exists\{x \varepsilon S \mid C(x)\}$;".

ASL includes cross section expressions that are written in the PL/I style, e.g., A(*, j) and A(i, *). SETL allows something similar to cross sections but the *'s are not written, and their implied positions must all be to the right. But the concepts are not the same, because they operate on structures of different types. For example, a matrix in ASL is a set whose members are of the form ((i, j), x). But in SETL, for "cross sections" to apply, they would have to be of the form (i, j, x) or (i, (j, x)), neither of which is a bona fide array, in my opinion.

In ASL the substring notation s(i:j) is generalized to apply to arrays (as in Algol 68), e.g., A(i:j, m:n). As was already mentioned, in ASL the j and n are indexes, whereas in SETL the variable after the colon is a length.

The class of expressions that may be used in a value receiving

context is approximately the same in ASL and SETL. One possible difference (I am not sure about SETL) is that in f(x) = y, f may be an explicitly displayed set (or vector). For example, the effect of (a, b)(i) = x is to assign x to a if i = 1, and to b if i = 2. This somewhat surprising result falls out of the sinister call definitions. One can also code assignments such as {('a', x), ('b', y)}(c) = z, which depends on ASL's map assignments, which is a generalization of vector assignments.

An interesting fundamental issue of language design that came out of ASL is its use of compiler generated temporaries. In an effort to live with side effects as comfortably as possible, before the evaluation of any expression, the variables are first assigned to temporaries, and the expression is evaluated in terms of the temporaries. For assignments, temporaries are introduced for both sides, and then after doing the assignment in terms of temporaries the target variables are assigned to the appropriate temporaries. This approach has many interesting effects. One is that the meaning of sinister composition of functions may be derived from it. The interested reader is referred to sections 4.10.2 and 5.2.3.

15.5 Statements

ASL combines the miscellaneous declarative statements of SETL into one: DECLARE. This is seldom used except with the elaboration language. However, it is used for a few things of more significance than efficiency, such as giving a variable the STATIC attribute and declaring the precedence of a user-defined operator.

ASL includes the FORMAT, STOP, EXECUTE, ENTRY, and null statements, and the remainder are essentially those of SETL. The EXECUTE statement is analogous to the execute instruction found on some computers. The target is a statement or statement group within the procedure containing the EXECUTE, which is executed, and then control returns. The statement group may not have a GO TO out of it. The ENTRY statement is similar to that of FORTRAN IV and PL/I.

As was mentioned, the meaning of the assignment statement has been expanded slightly. In ASL an assignment such as

$$\{(a, b),\ (c, d)\} = e$$

is allowed. It means (essentially) b = e(a) and d = e(c). As a special case we have (x, y) = e meaning x = e(1) and y = e(2) (in ASL the notation (x, y) means $\{(1, x),\ (2, y)\}$).

15.6 Statement Brackets and Headers

Similarly to Algol 68, statements in ASL may be grouped with parentheses, DO ... END, or BEGIN ... END, all of which have the same meaning. In SETL only parentheses may be used.

ASL takes the view that THEN, ELSE, iteration, and INITIALLY are all "headers" that take a single statement as an object, and do not in themselves require an "end" token. However, the "statement" may be a statement group. SETL takes the view that these "headers" act like left parentheses, and require a closing right parenthesis, for which an extra semicolon is used.

ASL includes a CASE statement, whereas SETL does not.

In SETL the iteration variable is bound, but in ASL it is free and may be used outside the loop. In fact, ASL goes to great lengths to give what is believed to be the most natural interpretation to the value of the variable on exit from the loop, and the effect of changing the upper limit while iterating, etc.

ASL includes an ITERATION pseudo-function, which allows one to conveniently obtain a count of the number of times a loop has been executed, even if the loop is not of the counting type. This can be used for a first-time switch, a reasonableness test, etc.

15.7 Procedures

As has already been mentioned, ASL has procedures of the function and operator type, but not subroutines (in a formal sense). Following Algol 68, every procedure is assumed to begin with a system-supplied prologue, which initializes many keyword variables, e.g. PI = 3.14159; SQRT = SYS.SQRT; etc. Most of these variables have library procedures as values. Although they have not all been defined by a long shot, it is anticipated that they may number about fifty to a hundred. The Algol 68 technique allows the coder to use them if he knows about them, and at the same time he won't get in trouble if he happens to inadvertently pick one for a variable or label name. It is probably reasonable for it to be standard practice for an implementation to add to the predefined variable list.

Parameter matching in ASL follows the same rules as vector assignment.

15.8 Name Scoping

At this point the name scoping rules in ASL are very simple. All variables are local to the procedure in which they are mentioned, unless they are declared "external" in one procedure and "shared" in the procedure to which they are local. Sharing variables by other than parameter linkage requires two-way cooperation. This approach is taken to enhance readability, but it may not be satisfactory for large

programs in which a good deal of sharing is involved.

SETL has a more complicated approach in which one declares "name scopes" in a nested manner, and one can declare that a certain variable name designates the same variable within a scope. Hence one can declare on page 5 of a program's listing that the "x" on page 20 is the same as the "x" in a different procedure on page 30. The author has acquired a great distaste for this, but it may be a necessary evil.

15.9 Input/Output

The I/O facilities of SETL are very primitive. One can write "print x, y, z", and the current values are printed, even if they are deeply nested structures. But one has no control over the format. In SETL one cannot even print a column of right adjusted integers, or a character string without delimiting quote marks.

The ASL I/O facilities are modeled after the PL/I LIST, DATA, and EDIT directed I/O. A very weak point of both languages is that at present there are no facilities to help one to write a program that converses with one or more users at terminals.

15.10 Macro Preprocessor

The macro, or preprocessor, for ASL has not yet been specified. SETL has a macro capability that consists of simple text replacement with parameters. That is, one can define a macro M that causes the replacement of a string M(x, y) with an arbitrary string involving x and y. It is believed that something with more flexibility is appropriate, and it is suggested to model a preprocessor after that of PL/I, but of course paralleling ASL. This would provide conditional assembly, which is a very significant capability for a preprocessor. One should also provide macro calls with a variable number of parameters (as in ASL itself), and a symbol generation capability. It is possible that ASL should go farther than this and expand macros based on some quite general pattern matching scheme. As a simple example, one might want to code a macro that would cause 2 + 3I to be translated into the vector (x, 2, 3), where x is a blank atom that flags the vector as designating a complex value. Of course other language extension facilities would be needed to allow the user to define his own data types, and no work in ASL has been done in this direction (a small amount of such work has been done in SETL).

15.11 Elaboration Facilities

The basic approach of the elaboration facilities is different in SETL and ASL. In SETL the elaborations are partly declaratory and partly executable. They are very direct, e.g., "store such and such a set as a hash table", and "from this point (in time) on, set S will not change very much".

The ASL approach is to try to get more mileage out of the elaborations by making them helpful to a human reader as well as to the compiler. They are at a higher level, and the emphasis is on telling the reader and compiler something about a program, rather than explicit instructions on how to compile something. For example, in SETL one might say "store I in ten bits and set S as a fixed block of five items, not hashed". In ASL one would instead say "the maximum magnitude of I is 500 and the maximum size of set S is five items". The compiler is then required to make its own decisions on how to best use the extra information. The decisions may depend on other factors, such as whether one is optimizing for speed or storage.

The SETL approach will probably lead to more efficient programs, and is certainly more easily implemented, but nevertheless I think it's a shame to pass up an opportunity to help the human reader and instead hinder him by littering the program with a lot of clutter.

16. Programming Examples

Integer Square Root Routine

SETL

```
definef ISQRT n;    /* Integer square root operator. */
if n le 1 then return n;;
x = n/2;
(while (x*x) gt n)
  x = (x + n/x)/2;;
return x;
end ISQRT;
```

ASL

```
FUNCTION ISQRT(n)    "Integer square root."
IF n ⩽ 1 THEN RETURN n
x = n/2
WHILE x**2 > n DO
  x = FLOOR[(x + n/x)/2]
  END
RETURN x
END ISQRT
```

Simple Insertion Sort

Reference: Knuth Volume 3, page 80.

PL/I

```
INSSORT:  PROCEDURE(SEQ);
          DECLARE SEQ(*) FIXED BINARY,
                  T FIXED BINARY;

          DO J = 2 TO DIM(SEQ, 1);
             T = SEQ(J);
             I = J - 1;
             DO WHILE (I >=1 & T < SEQ(I));
                SEQ(I+1) = SEQ(I);
                I = I - 1;
                END;
             SEQ(I+1) = T;
             END;
          END INSSORT;
```

ASL

```
          FUNCTION INSSORT(SEQ)
          2 ⩽ ∀j ⩽ #SEQ DO
             t = SEQ(j)
             i = j - 1
             WHILE (i ⩾ 1 & t < SEQ(i)) DOING i = i - 1
                SEQ(i+1) = SEQ(i)
             SEQ(i+1) = t
             END ∀j
          RETURN SEQ
          END INSSORT;
```

TOPOLOGICAL SORT

Problem

Suppose we are given a set S of arbitrary objects together with a partial ordering P on S. P is given as a set of pairs (a, b) with a, b ϵ S.

Arrange the members of S into a vector V such that if a = V(i) and b = V(j), and (a, b) ϵ P (meaning a $\lessdot$ b), then i $\leqslant$ j.

Solution

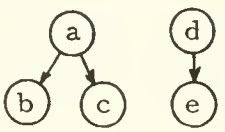
1. We select an arbitrary member x of S which has no predecessor, and append that to V (V is initially null).

2. Having successfully placed x in V, we delete x from S and also delete all pairs beginning with x from P (if any exist).

3. We continue this process until S is null.

Example

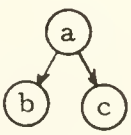
Suppose S = {a, b, c, d, e} and P = {(a, b), (a, c), (d, e)}.
Diagramatically, P is:



On the first iteration, either a or d might be selected. Suppose it is d. Then after the first iteration, the situation is:

V = (d,); S = {a, b, c, e}; P = {(a, b), (a, c)}.

Diagramatically, P is:



Next, either a or e could be selected. Suppose it is a. Then after the second iteration:

V = (d, a); S = {b, c, e}; P = $\emptyset$.

In the last three iterations, b, c, and e are selected in an arbitrary order. The final vector returned might be V = (d, a, c, e, b).

SETL

```
definef TOPSORT(P, S);
P1 = P;
S1 = S;
V = nult;
(while S1 ne nl)
   y = ³{x ε S1 | not (∃p ε P | p(2) eq x)};
   V(#V+1) = y;
   S1 = S1 less y;
   P1 = P1 - {p ε P | p(1) eq y};
   end while;
return V;
end TOPSORT;
```

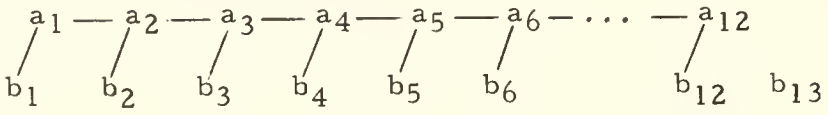
ASL

```
FUNCTION TOPSORT(P,S)
V = ∅
WHILE S ≠ ∅ DO
   y = x ε S : ¬(∃p ε P : p(2) = x)
   V(#V+1) = y
   S = S - {y}
   P = P - {p ε P | p(1) = y}
   END WHILE
RETURN V
END TOPSORT
```

References:

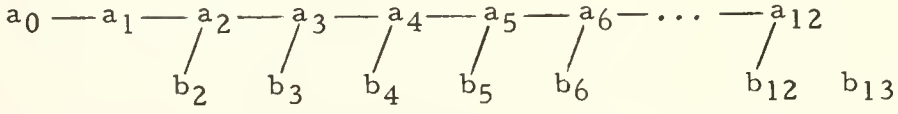
1. Knuth Volume 3, page 185.
2. Ford and Johnson, American Mathematical Monthly, Volume 66, 1959, page 387.

This algorithm sorts in the fewest number of comparisons of any algorithm known, in the minimax sense (not in the average sense). To illustrate how it works, assume we are given 25 items v1, v2, ..., v25. We start by comparing items in pairs v1:v2, v3:v4, ..., v23:v24. We place the larger of each pair in a vector A, and the smaller in a vector B. We then sort A, using this algorithm recursively, and rearrange B in the same way that A was permuted. The odd item v25 is then added to the right end of B, so the situation may be illustrated:



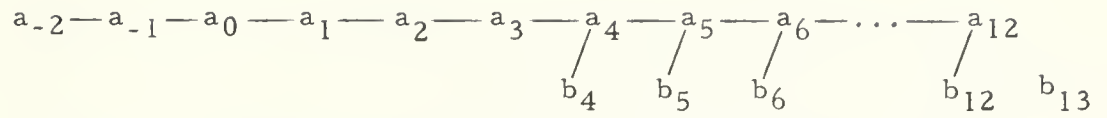
where the smaller items are to the left and below.

We now merge items from list B into list A. Since b_1 is smaller than any of the a's, it is immediately placed on the left end of A, giving:

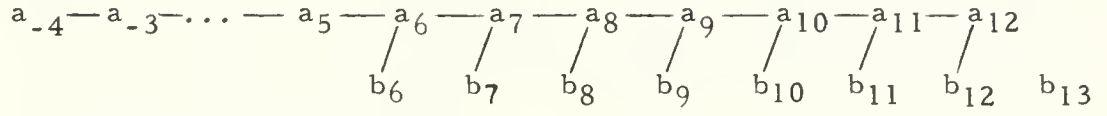


We continue merging items from B into A in what may be the most efficient way possible. We use binary searching, which is at its most efficient when the number of items being searched is of the form $2^n - 1$. The pattern to be described is always searching a list of this length, for worst-case data, except possibly on the last pass.

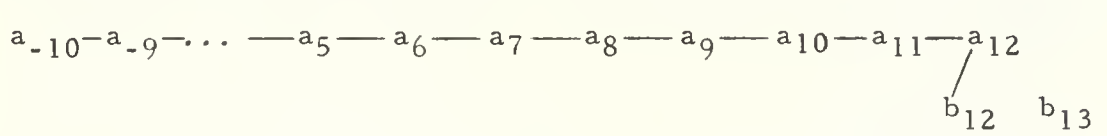
Observe that b_3 can be placed by considering a list of three items: a_0, a_1, and a_2. It is placed next, using two comparisons. After that, b_2 is placed. If b_3 went somewhere to the left of a_2 (worst case), then b_2 must also be inserted in a list of three items. If b_3 went to the right of a_2, then we are lucky, although in this case two comparisons may still be required to insert b_2. After inserting b_3 and b_2, the situation is:



It is now most efficient to insert into a list of length seven, namely a_{-2}, a_{-1}, ..., a_4. Item b_5 may be inserted, and then b_4 also is placed in a list of length seven (assuming b_5 goes to the left of a_4). The situation is then:



At this point the list length is 15, namely a_{-4}, a_{-3}, ..., a_{10}. Into this we insert b_{11} first, then b_{10}, b_9, ..., b_6. This leaves us with:



On the last pass, we insert b_{13}, followed by b_{12}. The number of comparisons required for the whole process is 86 (worst case). This consists of the initial 12, 30 for the recursive call, and 44 for the binary insertion steps outlined. The last two items were not inserted particularly efficiently, which leaves one with the feeling that there may be a better way to do it.

Ford-Johnson Tournament Sort

(Merge Insertion)

```
FUNCTION FORDJ(V)

"The parameter V is a vector of items to be sorted.  The value
returned is a vector containing the items of V in increasing order.
To compare two items, this procedure references an operator
.LE., which must be supplied by the user."

IF #V ⩽ 1 THEN RETURN V                "Trivial cases."

"Scan the components of V two at a time.  Put the larger of each
pair in a vector AU (A unsorted), and the smaller of each pair in
a vector BU."

STARTING AU = BU = ∅;  ∀i = 1, 3, ..., #V - 1 DO
   a = V(i)
   b = V(i+1)
   IF a .LE. b THEN (a, b) = (b, a)    "Make a the larger."
   AU(#AU+1) = a
   BU(#BU+1) = b
   END ∀i

oddone = V(i+2)                        "Only exists if #V is odd."

"Sort the half-length vector A, using this routine recursively."

A = FORDJ(AU)

"Now rearrange BU in the same way that AU was rearranged."

1 ⩽∀j ⩽#A DO
   n = 1 ⩽ n ⩽#A : AU(n) = A(j)        "n = position where A(j) came"
   AU(n) = ;                           "from.  Erase AU(n) so this"
   B(j) = BU(n)                        "n won't be used again."
   END ∀j
B(#B+1) = oddone
```

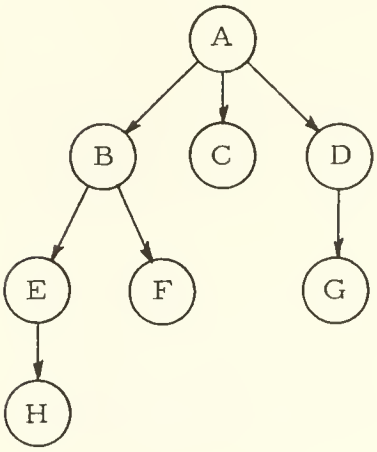
```
"Now merge the components of B into A using a binary search.
Vector A will grow on the left.  The components of B are picked
up in an order that maximizes the efficiency of the binary search
by always merging into a list of 2**n-1 elements (or less under
fortuitous circumstances), for some n.  The steps below place
B(1) on the left end of A with no comparisons done.  The order of
picking up B's is 1; 3, 2; 5, 4; 11, 10, ..., 6; 21, 20, ...12; ...."

STARTING jbot = 1                          "jbot = 1, 2, 4, 6, 12, ..."
         jtop = 1                          "jtop = 1, 3, 5, 11, 21, ..., #B"
         length = 1                        "length = 1, 3, 7, 15, 31, ..."
WHILE jbot ⩽ #B DO
   ∀j = jtop, jtop-1, ..., jbot DO

      "Merge B(j) into list A(LI A : j-1)."
      low = LI A - 1                       "One lower than leftmost item."
      high = j                             "One higher than rightmost."
      WHILE high-low > 1 DO
         mid = FLOOR((high+low)/2)
         IF B(j) .LE. A(mid) THEN high = mid; ELSE low = mid
         END

      "B(j) goes between low and high (even in the cases where it
      goes on an end)."
      (LI A ⩽ ∀i ⩽ low) A(i-1) = A(i)
      A(low) = B(j)
      END ∀j

   jbot = jtop + 1                         "Set indexes for next pass."
   length = 2*length + 1
   jtop = MIN(LI A + length, HI B)
   END WHILE jbot ⩽ #B

RETURN A@1
END FORDJ
```

An ordered tree is a descendent function desc(node, j) defined for j in some finite (possibly null) range.

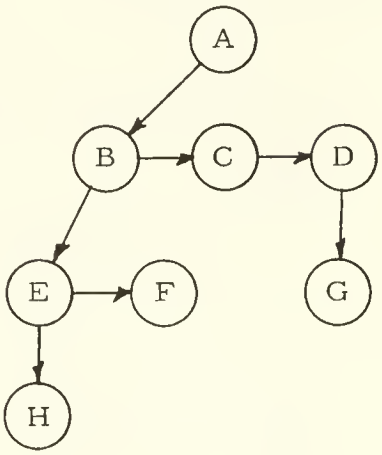
A binary tree is a pair of descendent functions L and R (left and right descendents).

The ordered and binary trees stand in an interesting 1-1 relationship that is illustrated below.

Ordered Tree



Binary Tree



Descendent Function

A	1	B
A	2	C
A	3	D
B	1	E
B	2	F
D	1	G
E	1	H

Descendent Functions

L		R	
A	B	B	C
B	E	C	D
D	G	E	F
E	H		

Ordered To Binary Tree Transformation

SETL

```
definef OTB(desc);
L = {<x(1), x(3)>, x ε desc | x(2) eq 1};
R = {<x(3), desc(x(1), x(2)+1)>, x ε desc | desc(x(1), x(2)+1) ne Ω};
return <L, R>;
end OTB;
```

ASL

```
FUNCTION OTB(desc)
L = desc(*, 1)
R = {[desc(x, i), desc(x, i+1)], ∀(x, i)ε D desc | ∃desc(x, i+1)}
RETURN (L, R)
END OTB
```

Ordered To Binary Tree Transformation

ASL

```
FUNCTION OTB(desc)
L = desc(*, 1)
R = {[desc(x, i), desc(x, i+1)], ∀(x, i)ε 𝒟 desc | ∃desc(x, i+1)}
RETURN (L, R)
END OTB
```

PL/I

```
OTB: PROCEDURE(NODE) RETURNS(POINTER)

     DECLARE 1 DESC BASED(NODE),
               2 NAME CHAR(50) VARYING,
               2 NSONS FIXED,
               2 SONS(NSONS) POINTER,
             1 NEW BASED(P),
               2 NAME CHAR(50) VARYING,
               2 LSON POINTER,
               2 RSIB POINTER,
             T POINTER;

     T = NULL;
     IF NODE ¬= NULL THEN DO I = NSONS TO 1 BY -1;
        ALLOCATE(NEW);
        NEW.NAME = SONS(I) -> NAME;
        LSON = OTB(SONS(I));
        RSIB = T;
        T = P;
        END;
     RETURN(T);
     END OTB;
```

Ordered To Binary Tree Transformation

Using List Structures

SETL

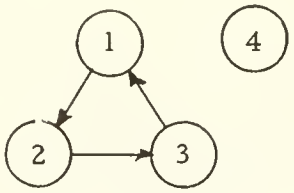
```
definef OTB(node);
t =Λ;  bin = nl;
if node ne Λ then
  (#sons(node) ⩾ ∀i ⩾ 1)
    p = newat;
    name2(p) = name1((sons(node))(i));
    lson(p) = OTB((sons(node))(i));
    rsib(p) = t;
    t = p;
    end ∀i;
  end if;
return t;
end OTB;

definef name1(p); return (ord(p))(1); end;
definef sons(p);  return (ord(p))(2); end;
definef name2(p); return (bin(p))(1); end;
definef lson(p);  return (bin(p))(2); end;
definef rsib(p);  return (bin(p))(3); end;
```

ASL

```
FUNCTION OTB(node)
t = ;
IF ∃node THEN DO
  [#sons(node) ⩾ ∀i ⩾ 1] DO
    p = ↑Ø
    name2(p) = name1[(sons(node))(i)]
    lson(p) = OTB[(sons(node))(i)]
    rsib(p) = t
    t = p
    END(∀i)
  END
RETURN t
END OTB

  FUNCTION name1(p); RETURN (↓p)(1); END;
  FUNCTION sons(p);  RETURN (↓p)(2); END;
  FUNCTION name2(p); RETURN (↓p)(1); END;
  FUNCTION lson(p);  RETURN (↓p)(2); END;
  FUNCTION rsib(p);  RETURN (↓p)(3); END;
```

Procedure to read in a graph, represented as follows:

```
4   3              n, m = sizes of node and arc sets
1   1   2          k, i, j = arc number, initial node, terminal node
2   2   3
3   3   1
```



GRAAL

```
procedure readone(G);
graph G;
begin integer n, m, k, i, j, ℓ; set x;
  read (n, m);
  for ℓ = 1 step 1 until n+m do x := create;
  for ℓ = 1 step 1 until m do
  begin read (k, i, j); assign (G, atom(i) - atom(j) to atom(n+k)) end
end
```

ASL

```
FUNCTION readone(G)
READ n, m
nodes = {ℓ, ∀ℓ = 1, 2, . . . , n}
STARTING arcs = ∅; ∀ℓ = 1, 2, . . . , m DO READ k, i, j
                                          (i, j) ε arcs
                                          END
G = (nodes, arcs)
RETURN
END readone(G)
```

If the node set were not important, and the graph were punched as:

$$\{(1, 2), (2, 3), (3, 1)\}$$

then the ASL "READ G" would suffice.

Subgraph of G having set N of nodes

GRAAL

```
procedure subgraph (G, N, SubG);
graph G, Sub G; set N;
begin set s, x, y, a;
while N ≠ empty do
   begin  x:= elt (1, N); s:=subset (a in star (G, x), inc (G, a)⊆N);
     N:=N ~ x;
     if s = empty then assign (Sub G, x) else for all a in s do
     begin y:= inc (G, a)~ x; if y = empty then y: = x;
      assign (Sub G, x - y to a)
     end
  end
end
```

ASL

```
FUNCTION subgraph (G, N)
RETURN {∀a ε G | a(1) ε N & a(2) ε N}
END
```

Line Graph of G

GRAAL

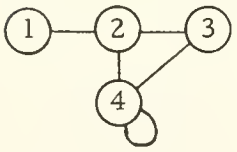
```
procedure linegraph(G, LineG);
graph G, LineG;
begin set S, R, x, a, b;
  for all x in nodes(G) do
  begin S := R := star(G, x);
    for all a in S do
    begin if x = inc(G,a) then assign(LineG, a-a to create);
      R:= R ~ a;
      for all b in R do assign(LineG, a-b to create)
    end
  end
end
```

ASL

FUNCTION linegraph(G)
RETURN $\{\{a, b\},\ \forall a \in G,\ \forall b \in G \mid \#(a \cap b) = 1\}$
END

Example of the ASL algorithm

Given Graph



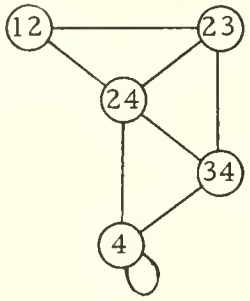
G

$\{\{1, 2\},\ \{2, 3\},\ \{2, 4\},\ \{3, 4\},\ \{4\}\}$

linegraph(G)

$\{\{\{1, 2\}, \{2, 3\}\},\ \{\{1, 2\}, \{2, 4\}\},\ \{\{2, 3\}, \{2, 4\}\},\ \{\{2, 3\}, \{3, 4\}\},\ \{\{2, 4\}, \{3, 4\}\},\ \{\{2, 4\}, \{4\}\},\ \{\{3, 4\}, \{4\}\},\ \{\{4\}\}\}$

New Graph



```
FUNCTION Pohls_shortest_path (G, start, end)
"G is a digraph containing the nodes 'start' and 'end'.  The value
of this function is a vector of nodes of G defining a shortest path
in G from 'start' to 'end', with each arc considered to be of unit
length.  If no path exists, the result is undefined."

fset = {start}
bset = {end}
fendset = fset
bendset = bset
fback = bback = ∅
WHILE fendset∩ bendset = ∅ DO
   IF #fset ≤ #bset THEN
      "Foward search"
      fnew = {y, ∀(x, y) ε G | x ε fendset & y ∉ fset}
      IF fnew = ∅ THEN RETURN; "Undefined, no path exists."
      (∀y ε fnew) fback(y) = x ε fset : (x, y) ε G
      fendset = fnew
      fset = fset ∪ fendset END
   ELSE DO
      "Backward search."
      bnew = {x, ∀ (x, y) ε G | y ε bendset & x ∉ bset}
      IF bnew = ∅ THEN RETURN;  "Undefined, no path exists."
      (∀x ε bnew) bback (x) = y ε bendset: (x, y) ε G
      bendset = bnew
      bset = bset ∪ bendset  END
   END WHILE

"A path has been found - construct it explicitly."
join = ɜ(fendset ∩ bendset)
path = ∅
x = join
[WHILE ∃x DOING x = bback (x)] path = (x, )¢path
x = bback (join)
[WHILE  ∃x DOING x = bback (x)] path = path¢(x, )
RETURN path
END Pohls__shortest__path
```

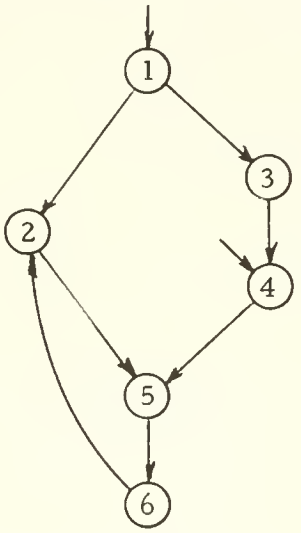
```
FUNCTION KTHONE(k, set)

"The value of this function is the k'th number, in ascending order, of
the given set 'set' of numbers.  If k is out of range, the result is undefined.
      This algorithm was discovered by Floyd, et al, in late 1971.  It
runs in a time directly proportional to the number of numbers in 'set'."

IF set = ∅ THEN RETURN;          "Undefined result."

WHILE #set ⩾ 3 DO
   "Build set 'midpoints', the set of middle values from 'set', taking
   the numbers three at a time."
   i = 2                         "Initialize."
   midpoints = ∅
   ∀x ∈ set DO
      i = MOD(i+1, 3)            "i is 0, 1, 2, 0, 1, 2, ..."
      CASE i
         0: u = x
         1: v = x
         2: DO "Put the median of u, v, and the current x into set
            midpoints.  Requires three comparisons (worst case)."
            IF x < v THEN m = 1; ELSE m = 0
            IF u < x THEN m = m+2
            IF v < u THEN m = 3 - m
            "Now m must be 1, 2, or 3."
            midpoints = midpoints ∪ {(u, v, x)(m)}
            END
         END CASE i
      END ∀x

   "As many as two members of 'set' have not been considered for
   placement in 'midpoints'.  But the error is not sufficient to prevent
   this algorithm from working in linear time.  Note that #midpoints ⩾ 1,
   because #set ⩾ 3.  Now find the median of 'midpoints', in linear time
   (this algorithm chooses on the low side if #midpoints is even)."

   median = KTHONE[FLOOR((#midpoints + 1)/2), midpoints]
```

```
   "Note that 'median' is somewhere in the middle third of 'set'.  Pre-
   cisely, the number of members of 'set' that are less than 'median' is
   at least (n/3 - 1)/2 + (n/3+1)/2, and the number of members that are
   greater is at least n/6 + (n/3+2)/2, where n = #set and '/' denotes
   'integer division'.
         Now divide 'set' into two piles; members of small_pile are ≤
   median, and members of big_pile are> median."

   small_pile = {x, ∀x ε set | x ≤median}
   big_pile = set - small_pile

   "Since #set≥ 3, and we have thrown the median into small_pile, we
   have #small_pile ≥ 2 and #big_pile ≥ 1.  Now iterate by finding the
   appropriate member of the appropriate pile."

   IF k ≤#small_pile THEN set = small_pile
   ELSE (set = big_pile; k = k - #small_pile)
   END WHILE #set≥ 3

"Now #set is 1 or 2 (it can't be zero).  k may be out of range if the original
call had k out of range."

(x, y) = (x, ∀x ε set)                    "x = one member, y = the other
                                           (if it exists)."

IF      #set = 1,              'YNN'
        k = 1,                 'YY '
        k = 2,                 '  Y'
THEN (RETURN x                 'X  '
        RETURN MIN(x, y)       ' X '
        RETURN MAX(x, y)       '  X')
ELSE  RETURN ;                          "Undefined result, k is out of range."

END KTHONE(k, set)
```

Backdominators

The concept of "backdominator" arises in the global optimization of computer programs. Given a node in a program graph, its backdominators are the nodes that must be passed through to reach it.

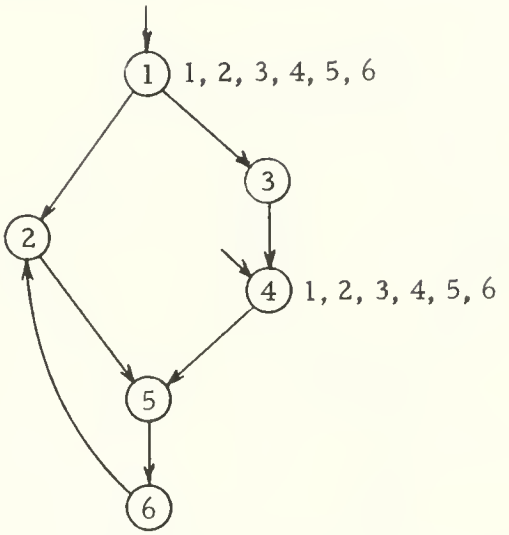
The algorithm to be described calculates all the backdominators of each node of a digraph. It operates in terms of the complement of the backdominators, which is the set of nodes not needed to reach a given node. This makes for an algorithm that is somewhat confusing, but it is actually very simple. We describe the algorithm using the example:



Nodes 1 and 4 are entry nodes. Any node may be an exit node; they do not enter into the picture.

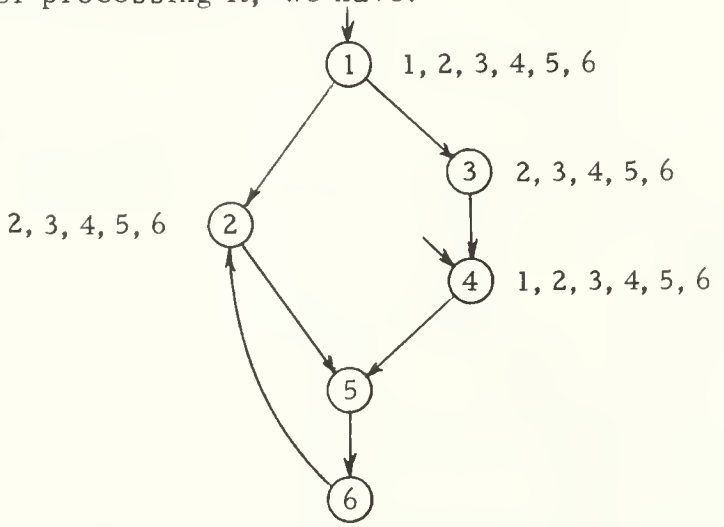
The algorithm works by first processing the entry nodes, then their descendants, then their descendants, etc. A set "todo" contains the current estimate of the nodes remaining to be processed. This is initially the set of entry nodes, and nodes are added to it and deleted from it in a way that is described below. The graph traversing terminates when "todo" is null.

No node is needed to reach the entry nodes, so the graph is marked accordingly:



todo = {1, 4}

On each pass, we choose from "todo" a node n to process. All the nodes that are not needed to reach node n are also not needed to reach its descendants, except possibly n itself, so this fact is recorded. In addition, if this process causes new nodes to be associated with a descendant, then that descendant must be reprocessed (or initially processed), so it is added to set "todo". Suppose node 1 is selected first. Then after processing it, we have:

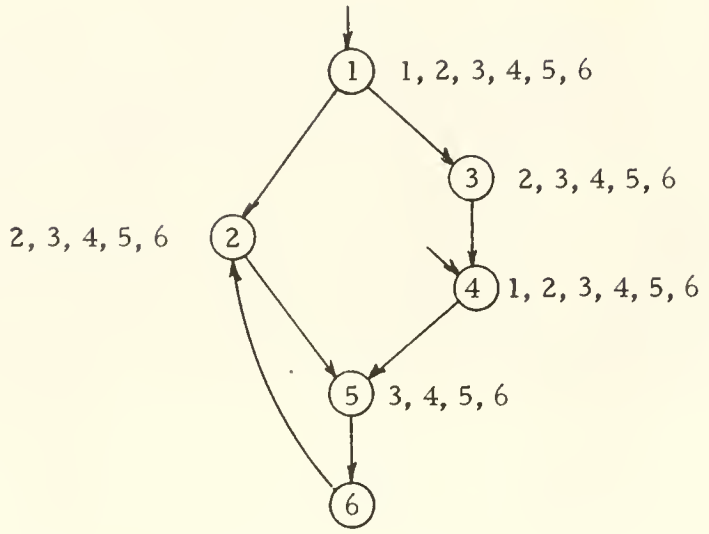


n = 1

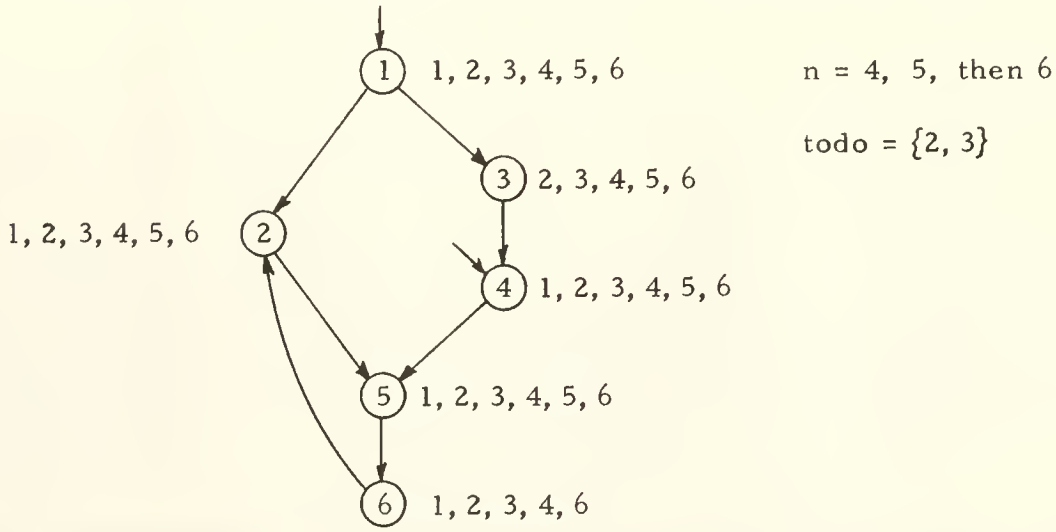
todo = {2, 3, 4}

Node 1 was removed from "todo", and nodes 2 and 3 were added, since they received new "not needed to reach" nodes.

For the second pass, assume node 2 is selected. Then we have:

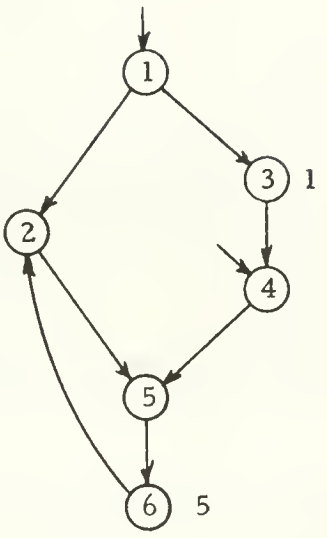


Assume that node 4 is selected next. This propagates nodes 1 and 2 to node 5, and sets todo = {3, 5}. If node 5 is selected next, then its nodes except for itself, namely 1, 2, 3, 4, 6, are propagated to node 6. The set todo at this point becomes {3, 6}. Next suppose node 6 is selected. Then its nodes except for itself, 1, 2, 3, 4, are propagated to node 2. This causes the addition of node 1 into the "not needed to reach" set for node 2, so node 2 is added into "todo". At this point the situation is:



On the last two passes, nodes 2 and 3 are processed. This does not change the above configuration, so "todo" becomes null.

The backdominators are given by the complements of the above sets:



For the procedure following, the graph of this example is represented by the set:

$$G = \{(1, \{2, 3\}),\ (2, \{5\}),\ (3, \{4\}),\ (4, \{5\}),\ (5, \{6\}),\ (6, \{2\})\}$$

together with the set:

$$\text{entries} = \{1, 4\}.$$

The result of the calculation is:

$$\{(1, \emptyset),\ (2, \emptyset),\ (3, \{1\}),\ (4, \emptyset),\ (5, \emptyset),\ (6, \{5\})\}.$$

```
FUNCTION BACKDOMS(G, entries)

"G is a digraph represented as a map from each node of G to the
set of all successors of the node.  This procedure calculates a
map that associates with each node of G the set of backdominators
of the node with respect to the given set of entry points."

nntr = {(n, ∅), ∀n ε ℘G}          "Initialize not-needed-to-reach map."
(∀n ε entries) nntr(n) = ℘G       "All nodes are not needed to reach
                                   an entry node."
STARTING todo = entries
WHILE todo ≠ ∅ DO
   n = ʒtodo                      "Get next node to process."
   n ∉ todo                       "Remove it from 'to do' work pile."

   "Process node n by passing on the set of nodes that are not
   needed to reach n to all the successors of n, except don't pass
   on n itself.  If any new nodes are actually added to the set of
   those not needed to reach a successor of n, then the successor
   must be reprocessed."

   ∀s ε G(n) DO                   "For all successors of n: "
      k = #nntr(s)                "Save current size of nntr(s)."
      nntr(s) = nntr(s) ∪ [nntr(n) - {n}]
      IF #nntr(s) > k THEN s ε todo
      END ∀s
   END WHILE

RETURN {(n, ℘G - S), ∀(n, S) ε nntr}  "Return the complement sets."

END BACKDOMS
```