Bruce Donald
Laboratory for Computer Graphics and Spatial Analysis
Graduate School of Design
Harvard University
40 Quincy Street
Cambridge MA 02138
U.S.A.
(617) 495-2526
(august, 1979)

ALTERNATIVE METHODS OF LANGUAGE CREATION

In order to facilitate the creation and structuring of a machine-transportable, table-driven language system, the Lab for Computer Graphics has developed the GLIB language maker and the LINGUIST language processing and control system. When supplied with a table of lexical, syntactic, and semantic data, GLIB then produces a language tree (the "Vault") for the parser containing syntactic transition states (Dougenik, 1976) and a "heap" which includes both storage space for variables and their type, structure, and length, as specified in the language definition. This information is packed into header words which preceed each element--arrays, of course, share a common header and are allocated sequentially.

In this paper we shall examine three alternative methods of manipulating the output from GLIB: the Classic (compilation) Method, the Co-Process (partial overlay) Method, and the Dynamic Loading method.

I.) Compilation

GLIB allows the user to define Language System varibles (Integers, Reals, Arrays, Switches, Sets, etc) which correspond to specific FORTRAN variables which he has presumably used in the Mother Program. Thus the programmer can, for example, write all his graphics and mapping routines and test them on a stand-alone basis, and add the control language later.

Until now, GLIB has generated FORTRAN Block Data which is then included in the user's program upon compilation. The Vault and Heap are allocated and initialized in COMMON with DATA statements. Next, a series of EQUIVALENCE statements are generated which force the user's variables to be coincident with the specific addresses in the Heap array that LINGUIST, the language management processor, sees. Thus, the language definition statement

```
20      (1 5, 0)        FILENAME        STRING: ILEN,NAMFIL(20)/0,20*4H    /
```

causes the FORTRAN variables ILEN and NAMFIL(i) to be equivalent to the heap addresses HEAP(n) and HEAP(n+i). Thus, when LINGUIST is asked to do an ASSIGN (place data in a variable) it looks up the language lexeme FILENAME (the external appearance of this string data structure) and references it via the Heap. The user would simply call ILEN (the length of the string) and NAMFIL (the string itself) by their normal names, and they would contain whatever LINGUIST had placed in them. Similarly, the language system can SHOW FILENAME, which is initialized to blanks.

Mechanically, this is accomplished through the statements

```
COMMON / PURITY/ LUMP(1000)
...
...
DIMENSION ... NAMFIL(20)
EQUIVALENCE  ... (LUMP(n),ILEN), (LUMP(n+1),NAMFIL(1)) ...
...
...
DATA ... LUMP(n-1)/<heap header>/,LUMP(n)/0/,NAMFIL/20*4H    /, ....
```

Here /PURITY/ is the virgin copy of the heap kept for reinitialization purposes, and ILEN is set equal to the length of NAMFIL by LINGUIST upon assignment. There are clearly three levels of representation here: the user sees only FILENAME, the programmer only ILEN and NAMFIL, and LINGUIST deals exclusively with the Heap (LUMP) entries. Only GLIB is aware of the larger picture, and of the interrelation of these three forms--this will be important later when we talk about dynamic variable dictionaries.

The Vault is similarly initialized and allocated at the same time. The user, however, need never access this array, since it is used exclusively to maintain the static and dynamic (user-defined) language trees. Thus there need be no interface between the Vault and the Mother program.

There are several unfortunate consequences with the straight Compilation method: first of all, any little change in the language requires re-compilation of the Mother program with the new BLOCK DATA from GLIB, especially if the Heap has changed in any way. For instance, adding or deleting any object structure referencing FORTRAN variable(s) will change the Heap so that the old versions will not run. In other words, the EQUIVALENCE pointers do not match any longer.

Ideally, one would like to be able to call a routine from the Mother Program which would dynamically load up the Vault and the Heap, while still allowing the user easy access to his variables.

Another problem arises when integration of a variety of system modules requires several different languages, each with different variables in the Heap, to be somehow accessable from one common main program. The first difficulty is the restriction on language size--while one could define co-resident dialects, each with a different initial key state, there is an upper limit on the number of lexemes and on the number of syntactic

transition states one can define. These limits are imposed by machine-dependent word lengths, since information is packed bitwise for efficiency—for instance, to allow binary searches. It is clear that several large languages would quickly overflow the limits of the language system were they to be integrated by concatenation.

A much more formidable problem arises when one considers that each of these languages has different variables, and different pointers into the Heap: thus they cannot be simply overlaid by reading in a "new" heap and Vault since, once again, the pointers into the heap will not match.

II.) The Co-Process or Partial Overlay Method

One possible solution is Partial Overlay of the Language System. Since the Vault itself requires no interface with the calling program, the language (aside from variable correspondencies) may be changed simply by reading in a new Vault for the Parser/Evaluator. When a new dialect is desired, a routine may be called which reads in the new Vault and Lump. If this is language 'A', then it is compatible only with, let us say, 'ASUB', which has the correct pointers into that specific heap, as supplied by GLIB for that paticular version of 'A'. Similarly, language 'B' may be read in by its Vault and Heap, and then 'BSUB' may be called, where its pointers are set correctly. Note that ASUB and BSUB are entirely incompatible, and if they were to be called at the wrong time (when their respective Language and Heap were not loaded) then their Equivalence statements would be pointing at garbage.

This is a more or less satisfactory method for solving the language overlay problem--(for more on the dynamic language loader, see SUBROUTINE MAKER, below)--but what happens when the language definition is changed? Unlike before, the Dynamic Vault/Lump File which the Language Loader reads in is all that must be changed for the parser—but if there is any change in the heap, if any new variables are added, or if old variables are deleted or changed, then the Equivalence pointers will be off once more, and the co-resident routines ASUB and BSUB must be recompiled with the new Lumpfile from GLIB.

However, for a developed language, ready for distribution, this is a somewhat elegant way of overlaying different languages. Once more, the problem is the recompilation necessary for maintenance and development. Even with Partial Overlay, the Mother Program is still hard-wired to that particular version of the language, until the next recompilation.

The Dynamic Vault/Lump file is created by executing the Dynamic File maker, DYNAM, with GLIB's usual output as Block Data. DYNAM includes a main conversion module along with the Laboratory Utility Routines for file handling. While the creation of a dynamic language file does require a compilation, it is certainly must faster than that of the entire overlaid

Mother program.

III.) Dynamic Loading

Truely dynamic loading of a language may be accomplished if variables are referenced not through equivalence statements, but through a call by the Mother Program to a "lookup" routine. Thus, when the language is loaded, a "variable" dictionary (produced by GLIB with the WRITE BINFILE option) may be read and packed into a search list. Then the user may reference his variables through calls to an interface routine, which looks up where the Mother variables are stored in the Heap.

With true dynamic loading, the necessity of recompiling the Mother program segments is eliminated--in other words, if only the language changes, then all we must do is run GLIB and DYNAM, and then specify the new filenames to MAKER, the dynamic language loader which pulls in the language and new heap. Variables are then gotten through the lookup subroutine.

Similarly, for overlaying different languages for various co-processes, the Master controler can call in a new language and variable dictionary. The co-processes can then copy data into their own common blocks with the aid of the Heap look up routines.

The procedure for producing the dynamic vault/lump file is the same for either partial or total overlay: DYNAM is executed along with the block data produced by the COPY BLOCKO option of GLIB, and a binary file is created. This can be done several times with different GLIB outputs to provide different runtime options.

Variable Dictionaries are created with the WRITE BINFILE <filename> command in GLIB. They contain the FORTRAN variable names, heap addresses, and dimensionality of all user variables for which GLIB allocates space, and are read dring the initialization of the lookup routines.

Languages may be changed in the run-time environment through calls to subroutine MAKER in LINGUIST, which takes the name of a DYNAM output file and (optionally) a GLIB variable dictionary.

A user may reference his variables in the heap by either calling a lookup function which returns an address into the LUMP array, or by calling a routine which will fill up an entire Mother program array. For example, to get the value of 'IVAL' the user would say

```
COMMON /HEAP/ LUMP(1)

DATA NAM/4HIVAL/
...
I=IREC(NAM)
IVAL=LUMP(I)
```

REALs and LOGICALs can be accessed by equivalencing arrays XUMP and QUMP to LUMP, and then saying

```
I=IREC('XVAL')
XVAL=XUMP(I)
```

or

```
QVAL=QUMP(I)     etc.
```

IREC takes three arguments: The name of the FORTRAN variable (stored in two A4 integer variables) and an optional index, which is used for referencing elements of one-dimensional arrays.

Other library lookup routines make call by reference slightly more palatable by filling one and two dimensional arrays (PUMP, PUMP2) and by placing the correct value in one element of multidimensional vectors (IPUT).

Unfortunately, the lookup routines require a certain amount of overhead. On an experimental basis, the COMMON space necessary to maintain these functions is on the order of 350 32-bit words. This can, of course, be reduced for smaller languages, since this much storage allows for lookup of nearly 100 different FORTRAN variables (arrays count only once).

Thus, there is also a function which returns the Heap address of any lexeme when supplied the lexeme number--the internal representation of the lexeme. LUMADR (Dougenik) differs from the lookup routines in that it requires no overhead or initialization. It does, however, force a more intimate knowledge of the language structure (to pack arrays or strings, for instance).

Let us return to our earlier example:

```
20      (1 5, 0)        FILENAME        STRING: ILEN,NAMFIL(20)/0,20*4H    /
```

To get at ILEN and NAMFIL with the lookup routines, one would create a variable dictionary with GLIB upon processing the language, and supply that name to MAKER when it was called to initialize the vault and heap. Then the code

```
COMMON /HEAP/ LUMP(1)
DIMENSION NAMFIL(20)
DATA NAMLEN, NAMF1, NAMF2/4HILEN,4HNAMF,4HIL  /
...
I=IREC(NAMLEN)
ILEN=LUMP(I)
CALL PUMP(NAMFIL, 20, NAMF1, NAMF2)
```

will fill up the counter and the string. (PUMP takes the name of the array to be filled and its length). While this may seem inelegant, it must be remembered that in the middle of the Mother Program's execution, MAKER can be called again and the LUMP can change dynamically. In fact, the same routine that called IREC here can be used again with the

new language in core, even if NAMFIL and ILEN refer to totally different variables (or are totally unrelated) in the new environment.

If in the new language different file parameters are needed, for example:

```
51      (22, 44)        NUMBEROFRECORDS         ILEN/1/
.
.
67      (2 51, 13)      OUTFILE                 NAMFIL(20)/4HOUT.,&
        4HMSG ,18*4H    /
```

then when IREC is reinitialized, the routines in the mother segment can still reference those variables.

To get the value of ILEN with LUMADR, one would write

```
COMMON /HEAP/ LUMP(1)
.
.
.
IADR=LUMADR(20)
ILEN=LUMP(IADR)
```

Note that 20 is the lexeme number for FILENAME. While the user must remember that for a STRING structure, the text (NAMFIL) is stored at Heap addresses IADR+1 to IADR+20, LUMADR has the advantage that it does not require the lookup routines or the GLIB dictionary file. However, unlike the dictionary lookup, the variables NAMFIL and ILEN are accessable by the same code in a newly loaded language only if the lexeme numbers are the same.

IV.) Evaluation

The key question concerning language creation must remain, "At what cost Dynamic Loading?" From what we have seen here, the greater the flexibility in language modfication--i.e., the greater the freedom from recompilation--the greater the sacrifice in elegance and ease of accessing variables in the Heap. While the intermediate steps are relatively efficient (to load a language and initialize the lookup routines with MAKER for a moderately large language requires on the close order of 50 disk reads and less than one second of CPU time on a PDP-10/K-A) true dynamic loading must require the user to go through interface routines to get at his variables. Most realistically, the following trade-offs must be taken into consideration: if the user does not mind being hard-wired into a particular language-base--i.e., can tolerate the necessity of recompilation--then he should remain with the old compilation method for small stand-alone languages, and use the partial-overlay technique for applications arising from the integration of various previously independent modules or requiring access to two or more different languages within the run-time environment.

While the dynamic processing method with variable lookup may also

be used for overlay purposes, it seems generally more useful for applications in which an entirely flexible language structure is desired. In debugging in a fully dynamic environment, only GLIB and DYNAM must be run to integrate the new language into the system. Translation into foreign languages may be similarly accomplished without a new compilation.

While the LUMADR routine requires less overhead than the dictionary lookup routines, it should be pointed out that conversion of previously written Mother Program segments may be more mnemonic with a call-by-name rather than a call-by-number system.

Once again, the size of the Mother Program and the cost of compilation may also be a factor in the decision. If frequent time-consuming recompilation is necessary to integrate a set of languages for, let us say, a partial overlay system, then perhaps conversion to dynamic loading is advisable. Perhaps each dynamic co-process should have its own "loader" which calls the dictionary routines and copies the language system variables from the dynamic heap to its own common.

Appendix I

Technical Notes

The IREC lookup routine is called my MAKER to initialize the variable dictionary. The characters of each FORTRAN variable name are packed in LCG internal (Q) format for machine transportability. The Binary dictionary file produced by GLIB is also in Q format, and is written in the routine DOEQIV, which generates the equivalence statements for the Block Data output file and the LUMPFILE.

A binary search for the lookup routines is envisioned, though not yet implemented.

Appendix II

Here we have an example using GLIB and DYNAM to create the necessary files for dynamic loading. First, the language definition file, DYNAM.LEX is TYPEd to get a listing. Next, GLIB is used to produce a Binary Dictionary File and a Block Data output file which contains the standard (constant) block data for the language system also (COPY BLOCK0).

Next TECO is used to add the FORTRAN statements BLOCK DATA and END to the GLIB output, so that it can be executed as a stand-alone file (as opposed to being included in the Mother Block Data).

Finally, the Dynamic Language File creator, DYNAM, is executed along with GLIB's output, and the LCG utility routines.

All we must give to MAKER to load the language are the names of the dynamic language file (TEST.DYN) and the GLIB dictionary file (TEST.BIN), and the language comes up.

MAKER asks for these names if they are not supplied by the calling program, but if they are given, language changing and loading is completely transparent.

```
.TYPE DYNAM.LEX
        OUT FILE: SNUG.DAT
        WRITE BINFILE: TEST.BIN
DEF LEX NUM: 10
FIN STAT: 33     (33,40)(3 34, 0)
VAL:             (40,0)
END
1       (1,3)   SHOW    SHOW:
2       (1,34)  DELETE  DEL:
10      (1,0)   DEFINE  DEF:
3       (1,3)   GIVE    PROG ONCE IMMED:
4       (1,3)   SET     ASS:

81      (3,40 0)        GAMMA   YY/9.1/
20      (3,40 0)        ALPH    A/5.5/
21      (3,40 0)        BET     B/-66./
22 (3,40 0) WORDS  ICHAR(5)/4HABCD,4HEFGH,4HIJKL,4HMNOP,4HQRST/
60      (3,40 0)
 REAL    VECTOR:  X(10)/1.1,2.2,3.3,4.4,5.,6.,7.,8.,-9.9,3.14/
61      (3,40 0)
BUNCH   INT:  ITWO(5,3)/1,2,3,4,5,6,7,8,9,10,11,12,23,14,15/
23      (3,40 0)        INT     II0/5/
34      (3,40 0)        ARRAY   IARR(7)/7*10/
35      (3,40 0)        LOGICALS        SET:QLOG(3)/3*.FALSE./
36      (3,40 0)        LOG1    LOGICALS[1]
37      (3,40 0)        LOG2    LOGICALS[2]
38      (3,40 0)        LOG3    LOGICALS[3]
40      (3,40 0)        SWIT    SWITCH: KOND1/1/
41      (3, 20 21)      FLOP    FORW:
```

```
42      (20,0)  ONE     NULL:
43      (21,0)  TWO     NULL:
END
50      FLOP    ONE     SWIT=2
51      FLOP    TWO     SWIT=3
END

.
.RUN GLIB [,13111]


WELCOME TO GLIB, Donald


!COPY BLOCK0
!IN FILE: DYNAM.LEX
  Transfering Control...


END OF LEXEME DEFINITIONS


TABLES REDUCED FROM  117 TO   116
       2 SWITCHES AND SETS        6 REFERENCES        6 SPACE FOR SCRATCH
HEAP ALLOCATED UP TO LOCATION     60

*** ERROR *** OUTPUT FILE ERROR OR FILE ALREADY   EXISTS


WANT TO OVERWRITE IT? YES
 MISSION ACCOMPLISHED


END OF EXECUTION
CPU TIME: 6.65  ELAPSED TIME: 51.58
EXIT

.

.TECO SNUG.DAT
HTECO.22.6F

[Lower Case Input]
*J      BLOCK DATA
$$
*AAAAAZJI       END
$$
[3K Core]
*EX$$

EXIT
```

```
.EXECUTE SNUG.DAT, DYNAM.NLA, MEFLAN.FLX, LCG10 [,13111]
FORTRAN: SNUG
.BLOCK
LINK:   Loading
[LNKXCT DYNAM Execution]


NAME THE DYNAMIC OUTPUT BINFILE TEST.DYN

I DID IT.


STOP

END OF EXECUTION
CPU TIME: 0.57  ELAPSED TIME: 7.23
EXIT

.RUN TEST


LANGUAGE LOADER...NAME THE DYNAMIC VAULTFILE *TEST.DYN
         NAME THE GLIB BIN FILE :
     ( TYPE "NOFILE" FOR NONE )  *TEST.BIN


?

?
***  SHOW
***  SET
***  DELETE
***  DEFINE
***  GIVE
?

^C^C
.
```

Appendix III

Here are two figures showing various options for language creation with different degrees of dynamic flexibility.

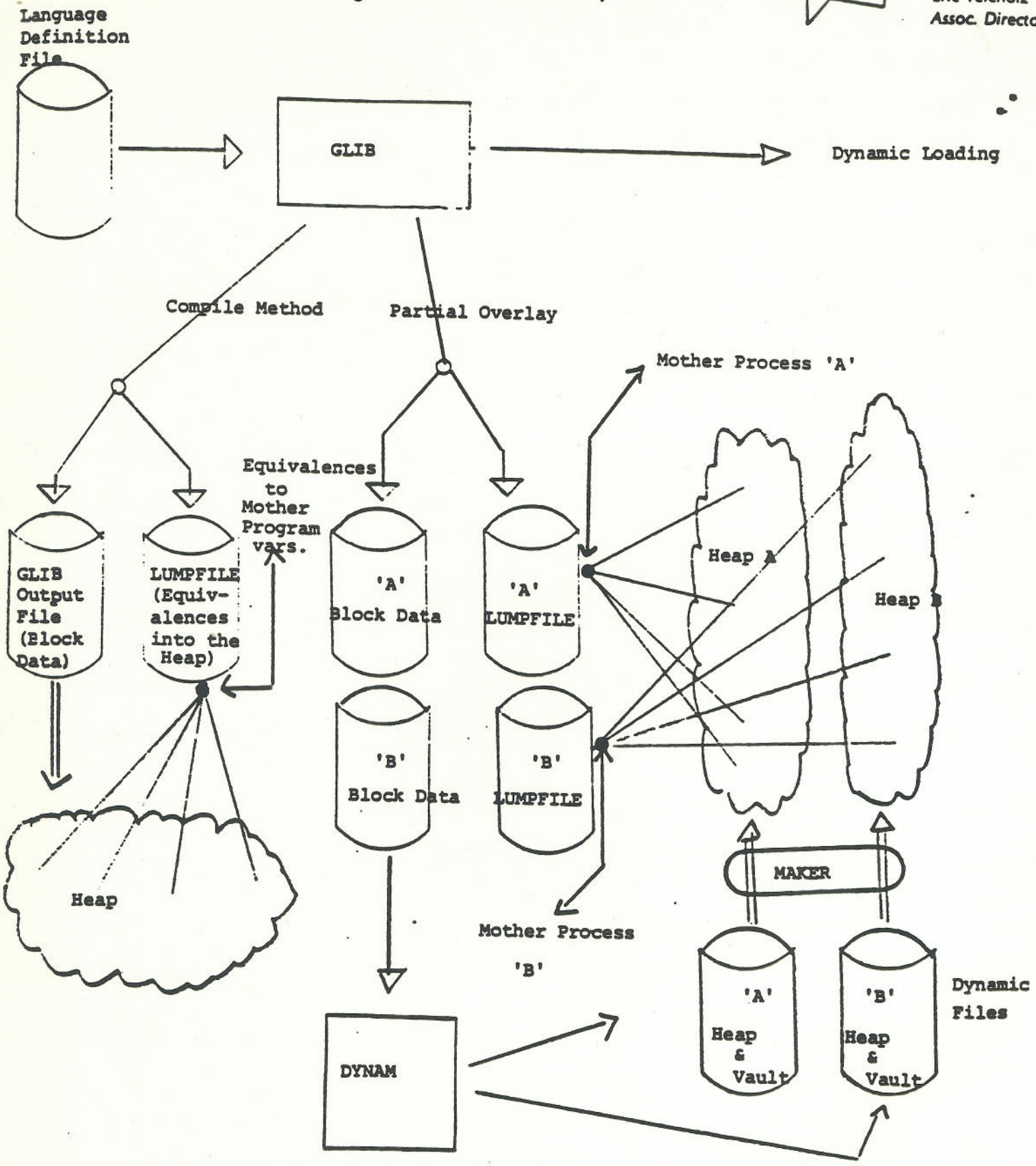
Laboratory for Computer Graphics and Spatial Analysis

Graduate School of Design ▫ Harvard University

Brian J. L. Berry
Director

Allan H. Schmi
Exec. Director

Eric Teicholz
Assoc. Director



520 Gund Hall 48 Quincy Street Cambridge, Mass. 02138 Tel: (617) 495-2526 Telex: 92-1

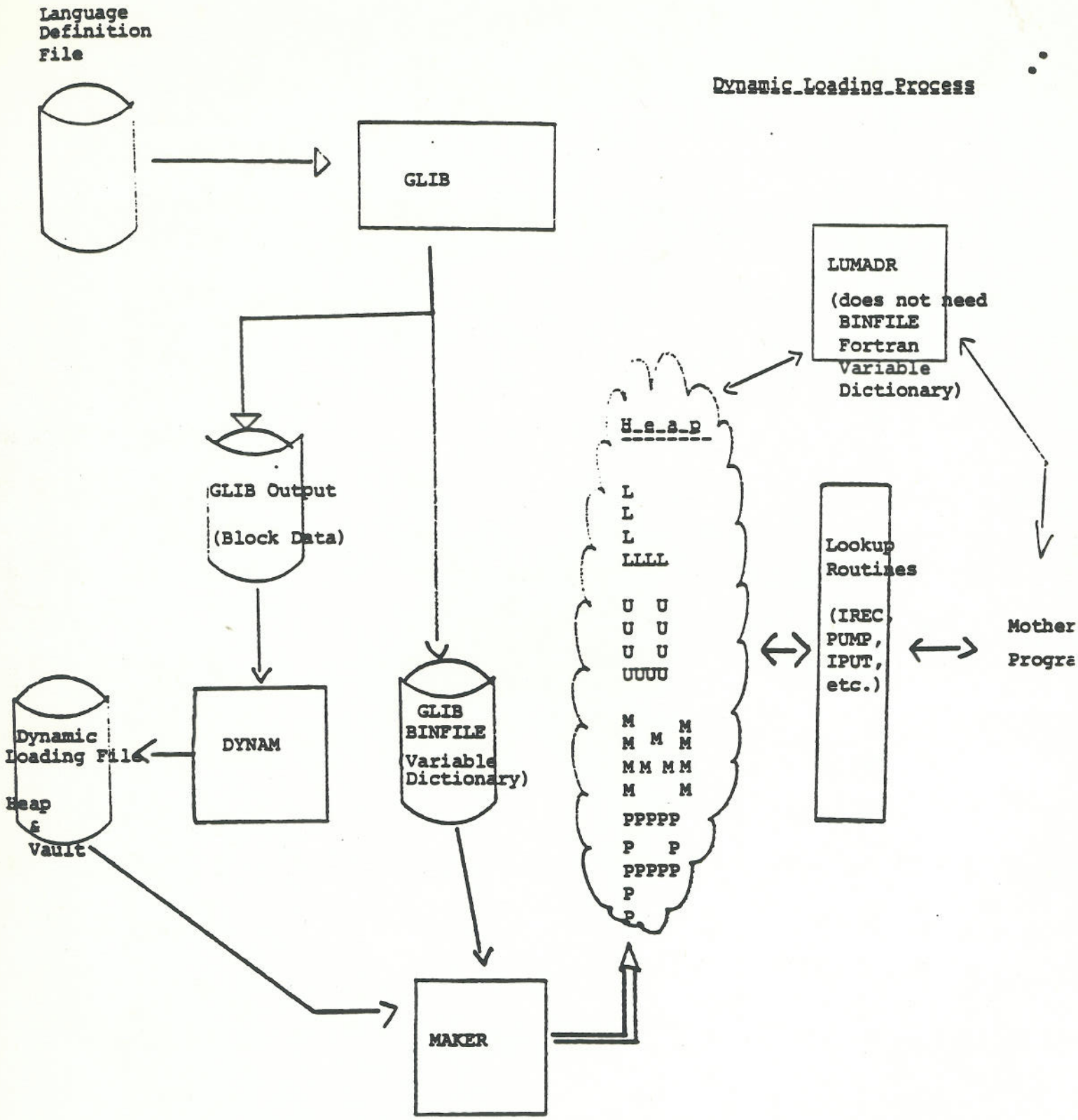
Laboratory for Computer Graphics and Spatial Analysis

Graduate School of Design ▫ Harvard University

Brian J. L. Berr
Director

Allan H. Schmi
Exec. Director

Eric Teicholz
Assoc. Director



520 Gund Hall 48 Quincy Street Cambridge, Mass. 02138 Tel: (617) 495-2526 Telex: 92-1

Appendix IV

Here is a listing of the chief dictionary lookup routine, IREC.

```
      FUNCTION IREC(INAME,JNAME,INDEX)
C
C BRUCE DONALD
C LAB FOR COMPUTER GRAPHICS
C HARVARD UNIVERSITY
C AUGUST, 1978
C
C GIVEN A FORTRAN NAME IN A4,
C SPECIFIED IN INAME AND JNAME
C AND AN INDEX (FOR ARRAYS OF ONE DIMENSION)
C
C THIS RETURNS A POINTER INTO^ THE LUMP FOR
C THAT VARIABLE.
C
C                FOR ARRAYS OF MANY DIMENSIONS,
C                (>1) IREC WILL BE SET EQUAL ONLY TO
C                THE ADDRESS OF THE FIRST ELEMENT OF THAT
C                ARRAY, SINCE TO PUMP
C                UP THE ARRAY, YOU MUST KNOW THE DIMENSIONS.
C
C RETURNS (-1) IF ILLEGAL OR NOT FOUND
C
C CALLING EXAMPLE:
C       I=IREC('ARRA','Y',34)
C       IF (I.GT.0) ARRAY(34)=XUMP(I)
C
C
C WHERE XUMP IS EQUIVALENCED TO LUMP FOR REALS,
C       QUMP IS EQUIVALENCED TO LUMP FOR LOGICALS, ETC.
C
C
C       ADD IS TRUE FOR ADDING NEW FORTRAN VAR NAMES,
C POINTERS INTO THE HEAP, (KH)
C AND # OF DIMENSIONS (ND)
C
C
      COMMON /UNITS/ IOCMND,IOLOG
      COMMON /HEAP/LUMP(1000)
      COMMON /FORVAR/IFOR(150),ISTAR(100),NUMFOR,NDIM(50),KHEP(50),
     + ADD,KH,ND,NPLACE,LBUF(8)
      COMMON/EXTINT/INCODE(127)
      LOGICAL ADD
      DIMENSION J(4)
      DATA NPACK/5/,MPACK/64/,LZZ/26/,LZERO,LNINE/48,57/,NW/1/
      NUM=0
      N=NPLACE+1
      JPT=1
```

```
      IF(ADD) GO TO 10
      CALL A4TOA1(INAME,J)
      CALL A1TOR1(4,J,J)
      DO 1 I=1,4
      JJ=J(I)
1     LBUF(I)=INCODE(JJ)
      CALL A4TOA1(JNAME,J)
      CALL A1TOR1(4,J,J)
      DO 2 I=5,8
      JJ=J(I-4)
2     LBUF(I)=INCODE(JJ)
C HAVE TRANSLATED INTO INTERNAL CODE...
C
C NOW PACK IT UP
C
10    IPACK=0
      DO 20 I=1,NPACK
              K=LBUF(JPT)
      IF (K.GT.0.AND.K.LE.LZZ) GO TO 22
      IF (K.LT.LZERO.OR.K.GT.LNINE) GO TO 25
22            IPACK=(IPACK*MPACK)+K
              JPT=JPT+1
20    CONTINUE
25    N=N+1
      IFOR(N)=IPACK
      IF ((K.GT.0.AND.K.LT.LZZ).OR.(K.GE.LZERO.AND.K.LE.LNINE))
     +        GO TO 10
      IF (ADD) GO TO 100
C WE HAVE RECOG TASK
C
C
C             *** HERE WE DO THE SEARCH ***
C
      KH=0
30    KH=KH+1
      IF (KH.GT.NUMFOR) GO TO 50
              I=ISTAR(KH)
              JP=NPLACE+1
35    IF (IFOR(I).NE.IFOR(JP))GO TO 30
              I=I+1
              JP=JP+1
      IF (I.LT.ISTAR(KH+1).AND.JP.LE.N) GO TO 35
      IF (I.NE.ISTAR(KH+1).OR.JP.LE.N) GO TO 30
C
C MATCH UP...
C SIMPLE CASE IS FOR SCALARS...
C
C
      IREC=KHEP(KH)
      NN=NDIM(KH)
C             NW IS THE NUMBER OF
C             WORDS PER VARIABLE
      IF (NN.EQ.1) IREC=IREC+((INDEX-1)*NW)
      RETURN
C
```

```
50      WRITE (IOLOG,1000) INAME,JNAME
1000    FORMAT(/' CANNOT GET ',2A4/)
        IREC=-1
        RETURN
C
C ADD IT TO THE LISTS AND TABLES
C
C
100     NUMFOR=NUMFOR+1
        ISTAR(NUMFOR+1)=N+1
        NPLACE=N
        NDIM(NUMFOR)=ND
        KHEP(NUMFOR)=KH
        IREC=KH
        IF(NUMFOR.EQ.1) ISTAR(1)=1
        RETURN
        END
```

REFERENCES

Dougenik, J., "Development of a Lexical and Syntactic Analyzer", Laboratory for Computer Graphics and Spatial Analysis, Internal Reports, 1976.

Donald, B., Language System Documentation, Laboratory for Computer Graphics and Spatial Analysis, 1978.

Ty @ Doc [311,523]

Bruce Donald
Harvard Laboratory for Computer Graphics and Spatial Analysis
40 Quincy Street
Graduate School of Design
Harvard University
Cambridge, Mass. 02138

Documentation of the GLIB Language Maker for the LINGUIST Language System

(LINGUIST was written by James Dougenik, Nick Chrisman, and Thomas Jaskelwicz)

(GLIB was written by Nick Chrisman, James Dougenik, and Bruce Donald)

I. INTRODUCTION

For a technical description of a language system (particularly one in which the format is variable) is is useful to turn to the Backus-Naur Form (BNF) of rewriting rules. This will offer us a degree of specificity that will come in handy later.

A brief summary of the symbol convention is given here for reference:

::= 'is rewritten as' (eg: <end statement>::= <CRLF> END

<.....> 'non-terminal delimiter.' The text enclosed is to be taken "metaphorically", i.e., not as a literal, but as a type--for instance, in the above

example, for instance, we obviously do not type the literal text 'CRLF'. Similarly, the statement <FORTRAN variable name> indicates not literal, but representational text

[....] indicates repetition of everything enclosed (a summed repitition) for 0-n times. a number n of iterations may preceed the expression. (eg.:
<variable definition>::=<variable> [,<variable>] <data>.
This allows for the defining of ! variable, or an optional number of additional variables, separated by commas.

<space> a space indicates concatenation. In the above example the elements <variable> and '[,<variable>]' are strung together to defing the variable.

\ Inclusive OR. (Eg.: '<input state>::=<integer> \ <state name>' defines ("rewrites") the input state as either an assigned integer ('7') or as a name, a variable in its own right ('START', 'QUIT', etc.)

It is important to note that literal text within a rewriting statement is the highest possible level of specificity In other words, the comma (above) which defines the <variable> mean "precisely comma" to the GLIB language processor.

II. DENINITIONS

The languages created by GLIB (from the user's input file) are table driven--since they all have certain salient similar characteristics, a language is created which we (more accurately) term as a "dialect."

(NOTE: First, we will need the following statements:

<CRLF>::= <new line> (we use CRLF as a terminator)
<end statement>::=<CRLF> END (definition of <end> in terms of literal text)
<integer>::=<digit>[<digit>] (definition of an integer)
<alphabetic>::=A\B\C\...\Z
<digit>::=0\1\2\3\4\...\9

In BNF, we rewrite (define) a dialect as follows:

<dialect>=<preface> <basics> <contexts>

as yet the preface section is basically unimplemented. In terms of existing software, PREFACE(IERR) in LINGUST corresponds to <preface>-generating subprogram, while COMBIN(IERR) creates the structure for context-sensitive lexemes(COMBIN is fond in GLIB). KHASER (in LINGUIST) chases down context-described lexemes in the run-time environment.

<basics> we may define as follows

<basics>::=[<basic lexeme>] <end statement>
(or, more exactly:
<basics>::=[<CRLF> <basic lexeme>]

similarly,

<contexts>::=[<CRLF> <context lexeme>] <end statement>

For GLIB, a basic lexeme must have the following form:

<basic lexeme>::=<lexeme #> <state transitions> <lexeme text> <semantics>

1 (1, 2) (3,7) SELECT ASSIGN PROGRESSING ONCE IMMEDIATE:

Here, '1' is the lexeme number (<lexeme #>::=<integer>), and 'SELECT' is the lexeme text--<lexeme text>::=<text>.

a) State transitions and the Finite State Automaton

The Finite State Automaton (FSA) is a mechanism for syntactic analysis which allows the linguistic decoder (LINGUIST) to determine when a given lexeme is valid in the course of a run-time job. The FSA maintains certain conditions, or states, and a table of lexemes--with each lexical item is associated a state, or choice of states, during which it is acceptable, and another state which is invoked upon exit from analysis. Thus, in the above example, if the automaton is in state '1', then SELECT is an acceptable command, and state '2' is called upon exit--or, if the automaton is in state '3', then state '7' is produced. States '1', and '3' are called input states, '2', and '7', output states. Thus a tree is created by which LINGUIST can keep track of what options are available to the user at any given branch. If the input states specified as valid in the GLIB file and the current state of the automaton do not match, then an UNRECOGNIZED SYNTAX error is generated. The advantage to this error is that it is non-fatal (recoverable) and prevents run-time errors generated by FORTRAN, which destroy the job.

For more on the Finite State Automaton, see the paper by James Dougenik available from the Harvard Laboratory for Computer Grapics.

The user may also specify several conditions for entry:

(2 4, 17) tells the system that either state 2 or 4 are acceptable environments for the given lexeme, and that state 17 is invoked upon exit. The entrance states are called Input States, 17 is the output state that is called when analysis of the current lexeme is over. Similarly,
(2 4, 17) (5, 56) Allows state 5 as an input state also, outputting 56 upon exit.

In BNF, we may define state transitions as follows:

<state transitions>::=[<state tuple>]
<state tuple>::= (<input state>[<input state] , <output state> [<output state>])

<input state>::= <integer>\<state name>

At present, the state names are not utilized--thus the initial state is still referred to as '1', rather that "BEgin", etc.

b) Semantics

In the semantics section of lexeme specification, the Language System is informed of what action it should take upon encountering a given lexical item. An important distinctin to make it that these GLIB primitives, by which lexemes are defined, refer only to the instruction of LINGUIST--thus, a powerful instruction such as DRAW [from ODYSSEY) is defined for GLIB as a NOOP (no opperation).

- Semantics may be of several kinds:

<semantics>::=<verb>\<token>\<object>\<set reference>\<switch reference>

VERBS: Verbs are composed of GLIB primitives and retreats. GLIB primitives require no intervention by the mother (calling) program), but instruct LINGUIST to modify its own internal tables (for instance, by inserting or deleting user-defined variables).Naturally, the new value assignments are accessible to the calling software.

Retreats allow the Mother Program to intervene for further syntactic, semantic, or computational analysis which LINGUIST cannot handle by itself.

<verb>::=<action>:\<action> <retreat>:

LINGUIST Actions:
-------- -------

ASSIGN--assign a value to a variable (call BINDER
and place data in a variable)

NOASSIGN--end assignation (exclude from assignation)
REASIGN--Restore the original (default) value to a variable.
CLEAR--Reset the system--INITIAL and PURGE

SHOW--show the elements and values of members of a set
HELP--type a help file, and a list of commands which are
available to the user at that point in the automaton tree.
NOOP--no operation

DEFINE--define a user variable
DELETE--delete a user-defined variable
PURGE--delete all user variables
(macro-delete)

INITIAL--resore the system variables to their virgin state
(macro reassign)

Retreats

RETREAT DOCUMENTATION

for the Language System

Havard Laboratory for Computer Graphics and Spatial Analysis

(Bruce Donald)

It is frequently desirable for a certain lexeme to invoke a return ("retreat") to the calling program during the course of execution. There are several types of retreats so that the programmer can control exactly when the retreat occurs and what state the syntactic and semantic processors are in.

As the language system parses off lexemes, is creates "sentences", or "executable units." Often there are several executable units to one syntactic unit, for instance, given

1	(1,2)	ASSIGN	ASS:
2	(2,3)	A	INT: A
3	(2,3)	BETA	INT: BETA
4	(2,3)	C	REAL: C
5	(2,3)		VALUE:

then ASSIGN A:5, BETA:7, C:17.0;

generates three executable units--the ASSIGN verb is syntactically distributed over the objects--and three assigns are done. Thus the syntactic processor is conservative, in that it retains the verb to distribute over objects, until it is told otherwise (given another verb, or a semicolon).

The programmer may specify whether, having found an executable segment, the system should retreat before or after executing it--the scan pointer remans in the same place. IMMEDIATE retreats occur before execution of the segment, DEFERRED retreats come after it is done. Note that upon return to LINGUIST after an IMMEDIATE retreat, the segment will be executed in any case, unless the user has removed the argument (let us say for an ASSIGN) from the stack. Thus, if a user tells the language system to rotate a plot 10E6 degrees, the mother program, sensing this in an IMMEDIATE retreat before the ASSIGN call occurs, can print an error message and substitute the old (and presumably acceptable) value for the outrageous one.

In general, EACH retreats occur after each executable segment is isolated--in the above example, 3 EACH retreats would occur, for all three ASSIGNS. ONCE retreats occur only upon encountering the verb--thus, only one ONCE retreat would occur above, that is, when ASSIGN was first parsed. In general, Executable statements require objects--an example would be ASSIGN A:5, or ASSIGN BETA:7--thus EACH retreats wait for the objects in these cases. However, lexemes that are executable without objects would cause retreats in the same place.

REGRESSING retreats wait until the verb (for ONCE) or until the object (for EACH) is no longer conserved by the parsing process. PROGRESSING retreats occur in the normal process of LINGO's scan of the stack for semantic analysis.

Types of retreats:
---

1) No retreat

2) PROGRESSING ONCE IMMEDIATE--retreat as soon as the verb is encountered, do not wait to analyze any more lexemes or to execute anything.

3) PROGRESSING ONCE DEFERRED--retreat after encountering the next verb, executing statements in between now and then. Does not execute the executable segment associeted with that verb until return from the Mother program.

(See examples.)

4) PROGRESSING EACH IMMEDIATE--find an executable statement, then retreat before doing it (i.e., prior to execution). To prevent execution of this statement, the calling program must modify the stack and throw away the illegal values, for instance. Otherwise the statement will be done upon return to LINGO.

5) PROGRESSING EACH DEFERRED--find an executable statement, do it, and then retreat. Scan pointer is in the same place as with Progressing Each Immediate.

it is no longer conserved by the system) and then retreat. For most intents and purposes, this is equivalent to waiting for a semicolon, i.e., for the end of the input string. With the semicolon, the stack is cleared, with a coma, it is popped one level.

7) REGRESSING EACH--as long as the object is conserved, do not retreat. In many cases this is the same as Progressing each deferred, except that Regressing Each is barricaded against semantic (but not syntactic!) distribution. (see examples).

Note: the following synonyms are also acceptable:

PROGRESSING=SEMANTIC

REGRESSING=SYNTACTIC

EACH=OBJECT

ONCE=VERB

DEFERRED=AFTER [EXECUTION]

IMMEDIATE=BEFORE [EXECUTION]

The NULLs EXECUTION and RETREAT are also permissable, but optional. These alternative terms are provided to give a practical, as well as stack-oriented view of retreats.

Examples: (optional lexemes are in square brackets)

PROG ONCE IMMED::=SEMANTIC VERB [RETREAT] BEFORE [EXECUTION]

REGRESSING ONCE::=SYNTACTIC VERB [RETREAT]

PROG EACH DEFER::=SEM OBJ AFTER or SEM OBJ RET AFT EXE

REGR EACH::= SYNT OBJ [RET]

PROG EACH IMMED::=SEM OBJ BEF or SEM OBJ BEF EXE, etc.]

Example:

Let us take a case where the lexeme DO is defined as follows:

DO::=ASSIGN <retreat> (eg.: ASS PROG EACH IMMED:)

(i.e., by

n (,) DO ASSIGN <retreat>:

where n=<lex #> and (,)= <state tuple>)

and the correct syntax for a DO statement is of the form,

DO <task> WITH <options>

```
DO TASK1 WITH OP1:VAL, OP2:VAL,  TASK2;
```

the language processor, by syntactic distribution, produces the following statements:

```
DO TASK1 WITH OP1:VAL
DO TASK1 WITH OP2:VAL

and

DO TASK2
```

The semantic distribution (with WITH as a NULL:) is as follows. Retreats are identified by their numbers above, in the place where the scan pointer is left upon exit:

```
ASSIGN          TASK1
      (3,2)          (4--before execution                 See Note 2
                      5--after execution)                 for explanation
                                                          of (3), PROG
                                                          ONCE DEFERRED

   "            OP1:VAL
                      (4--before execution
                       5--after  execution)
```

[Note: the verb (ASSIGN) is conserved and semantically distributed over the options. Since ASSIGN is not parsed off again (thus, does not invoke a Progressing Once retreat) its conservation is indicated here by a "ditto" mark. Below, ASSIGN is parsed off again, to be distributed over TASK1 (again) and the second set of options--this provokes PROG ONCE calls, as with the explicit case above.]

```
     ASSIGN          TASK1
(7)        (3,2)          (4,5)

   "                 OP2:VAL
                          (4,5)


     ASSIGN          TASK2
(7) (7)      (3,2)          (4,5,6,7)
[See note 1]
```

Note 1: Regressing Each is sensitive to syntactic, but not to semantic distribution. Thus, a retreat occurs when an object has been discarded--above, a Regressing Each retreat would occur when (1) OP1, (2) OP2, (3) TASK1, and (4) TASK2 were no longer conserved--i.e., when they were replaced or when the stack was cleared. Note that after "...OP2:VAL," two retreats occur (in the order above) since two objects are discarded consecutively.
(Since Regressing Each is sensitive to syntactic distribution, clearly commas are a good indication of where retreats will occur, if not of how many).

Note 2: A Progressing Once Deferred retreat occurs in the same place as the Progressing Once Immediate--except for one very important difference: A Once Deferred retreat is specified by the last verb the language system has encountered, whereas a once immediate retreat is specified by the current verb. In other words, were this input string part of the command:

SYNC, DO TASK1 WITH OP1:VAL, OP2:VAL, TASK2,

where SYNC::=NOOP PROGRESSING ONCE DEFERRED (for example)

then a retreat would occur in the places mentioned. However, a (3) retreat between DO and WITH OP1:VAL could not be specified by DO::=ASSIGN PROG ONCE DEFERRED--obviously, the return would wait for the next verb.

Let us take another example: if we define DO as follows:

```
<lex #> <states>        DO      ASSIGN PROGRESSING ONCE DEFERRED:
```

and also define WITH as a verb (instead of a NULL:) which reqires no retreat:

```
(lex #> <states>        WITH    NOASSIGN:
```

Then the retreat specified by DO will occur upon encountering WITH, the next verb:

```
--ASSIGN--                  --NOASSIGN--                        (:Semantics
   DO          TASK1            WITH            OP1:VAL,        (:Text
      (2)          (4,5)         (3)                  (4,5)     (:Retreats
                                 ---
                                 ---
```

Retreats and the position of JSTACK (the Scan Pointer)

```
--ASSIGN--                --0--                                 (:Semantics
   DO          TASK1      WITH     OP1:VAL,                      (:Text
     (3,2)        (4,5)                   (4,5)                  (:Retreats


--ASSIGN--                --0--                                 (:Semantics
   [DO         TASK1      WITH                 ]                (:Implicit Text
(7)   (3,2)        (4,5)                                        (:Retreats


                 --ASSIGN--                          (:Semantics
   OP2:VAL          [DO]       TASK2;                (:Explicit and Implicit Text
(7) (7)      (4,5) (7)    (3,2)     (4,5,6,7)        (:Retreats
```

More on Progressing Once Deferred: A Progressing Once Deferred retreat will take effect upon encountering the next verb, even if it is found in the following line -- for example,

DO TASK1;

Where DO is an ASSIGN PROG ONCE DEFER:

will cause a retreat when the next verb in the nextline is typed:

```
?
DO TASK1;
?
ASSIGN ALPHA:7.0;
```

(retreat occurs upon finding ASSIGN).

Commas can cause retreats to occur in dfferent places than spaces, since commas explicitly pop the stack.

in the sequence of commands

```
DO TASK1 WITH OP1, OP2, OP3;
?
ASSIGN ALPHA:8.9;
```

will cause a Progressing Once Deferred retreat after OP1, after OP2, and upon encountering ASSIGN in the next line. The commas force the semantic processor to distribute DO over al the options--thus, when a comma is found, a "new verb" is placed on the stack--DO.

Compare

DO TASK1 WITH OP1 OP2 OP3;

which only calls a retreat in the next line, at the first verb encountered there.

Comments: It is probably fair to say that regressing retreats are used more frequently for further syntactic analysis (since they are insensitive to semantic distribution) and that progressing returns are most useful for additional semantic work.

There are some special cases we should mention:

If DO is defined as ASS REGR EACH, a command such as

DO INPUT TRANSFORMATION, OUTPUT TRANSFORMATION;

will ordinarily cause two retreats to the mother program one at the comma (when the first object is discarded) and one at the semi-colon (when the stack is popped and the second object is no longer conserved).

Consider a sentence such as

DO NO INPUT TRANSFORMATION, NO OUTPUT TRANSFORMATION.

This translates (in LINGUIST primitives) to an ASSIGN REGRESSION

one HEGH EACH retreat will occur--at the end, at the semi-colon.

In other words, since multiple verbs are used (NO is a NOASSIGN) LINGUIST perceives no change in the object (and, indeed, NO does not change) the user may run into trouble if he expects two retreats out of such sentences.

Note that in a command such as

DO NO INPUT TRANSFORMATION, OUTPUT TRANSFORMATION WITH <options>

two Regressing Each retreats will occur, since the object is perceived as changing.

TOKENS: Tokens are used to define context-sensitive lexemes and nulls.

<token>::=NULL:\FORWARD:\BACKWARD:\ALIAS:

A context-described lexeme may be sensitive either FORWARD or BACKWARD, i.e., its meaning may depend either upon what preceeded of upon what followed it in the string. For example,

<lex #> <transition states> OR BACKWARD:
<lex #> <transition states> FILE BACKWARD:

both define OR and FILE as variable in meaning depending on what was typed before them.

<lex #> <transition states> FORM FORWARD:

declares FORM to be context-sensitive only in the forward direction. Thus in the line SELECT ELEVATION OR LANDUSE, "OR", being sensitive backwards, requires searching in a retrograde scan directin to determine its meaning, whereas in FORM DETAILED LANDUSE, FORM demands only a forward search, over the text "DETALED LANDUSE." (This is intuitively clear, since nothing can preceed FORM in valid syntax). See below for more on context-sensitive lexemes.

An ALIAS is used to inform the language maker that the given lexeme is a synonym for another elsewhere in the table. The same lexeme number is returned when the synonym os encountered. Thus, the semantic information neednot be spelled out twice.

OBJECTS: Objects in the language system declare and initialize variables and establish their type, structure, and correspondence with the FORTRAN variables accessible to the mother (calling) program.

<object>::= <sructure> <type>:<variable definition> \

<type>:<variable definition> \

<structure>:<variable definition> \

<variable definition>

i) <variable definition>::=<variable> <data>
ii) <variable>::=<FORTRAN variable name>\<name> (<integer>

iii) <data>::= /<text>/ \ /<text> [& <CRLF> <text>] /

iv) <type>::=INTEGER\REAL\LOGICAL\ etc.
(a <type> declaration declares the variable to be a FORTRAN typed variable. Implicit declarations are also valid).

v) <structure>::=VECTOR\SEQUENCE\STRING\SWITCH\SET

Examples:

EXCPP /4./ initializes the FORTRAN variable EXCPP to a value of the full statement were

<lex#> <state transitions> BUT EXCPP/4./

then the variable would be referenced by the user (in the external environment) as BUT, while the mother program would call it EXCPP.

WHERE MEM/100/ initializes the language system variable WHERE and the implicit integer FORTRAN variable MEM to 100.

VIEW VIEW(4)/4*0./ defines the language system array and the FORTRAN array VIEW under the same name, declares the array size to be 4, and initializes al values to zero.

LOGICAL:TICKS/.FALSE./ declares TICKS as a logical variable giving it a default value of .false.

Structures that may be specified are as follows:

A VECTOR is an array variable, and may be implictly declared (as with VIEW(4), above) through a dimension specification.

A SEQUENCE is like a vector, except that it has two (structural) components--the second is the actual body of the vector, and the first returns how many elements of the array are being used. For instance,
SEQUENCE INT: NPOL, IPOL(100)/0,100*0/ initializes IPOL to all zeros--NPOL will be set to the number of elements in IPOL that are filled up (referenced) every time the lexeme (in the langauge system) is called. Note that the name for the user may be very different:
<lex#> <syntactic position> POINTS SEQ INT:NPOL,IPOL(100)/0, 100*0/ will cause the sequence array to be referenced as POINTS by the user, by as IPOL, (with NPOL as a counter) by the internal tables.

<sequence def>::=SEQUENCE:<counter> <array> (<size>)/<initial values>/

A STRING is a text vector (Character variable) and may also be declared implicitly, as follows:
STRING: IOF,INFIL(8)/0,8*4H / defines a sequence and initializes the characters to 4 blank spaces. The string pointer (IOF) differs from a sequence pointer in that it specifies how many characters (not elements) are currently in the string. There are 4 characters to a word--but this calculation must be done by the mother program.

A SET definition defines (declaes and initializes) a user-accessable set:

```
LANDUSE          SET:QULAND (34)/34*.FALSE./
```

defines the lexeme LANDUSE in the Language as a logical array of length 34, named QULAND in FORTRAN, and initializes it to .false.

SETS are referenced through <set reference> statements (below).

SWITCH definitions: GLIB the langauge maker allows switches to be set upon encountering certain lexemes. This is particularly useful for cases where (for example) the user wishes to specify a device (a plotter, a line-printer) without knowing the device number within FORTRAN. SWITCHES are defined and given a defalt value as follows:

```
<lex#>  <syntactic pos>        DEVICE          SWITCH:KDEV/2/
```

Here, DEVICE (corresponding to the FORTRAN variable KDEV declared a switch and given a defalt value of 2. Switches are referenced in the run-time environment by <switch reference> statements (see below).

It is through the definition, declaration, and initialization of objects that GLIB creates a COMMON block for the FORTRAN mother program, from these statements is generated the mass of <type>, DIMENSION, EQUIVALENCE, and DATA statements that initialize the language system. Thus, from above, KDEV is declaed in a FORTRAN DATA statement as 2, IPOL is initialized and dimensioned, etc. The GLIB primitives

```
SEQ: NCORD, CORD(100)
```

generate a REAL array and a counter NCORD in the block common statements that GLIB produces for LINGUIST.

SET REFERENCES: Set references have the following form

```
<set reference>::=<set lexeme text> [ <integer> ]
```

and allow the user to access an individual element of a set defined in a SET definition statement (above). For example:

```
66      (35 2 100, 0)   AVF     LANDUSE[4]
```

tells the langauge system that whenever AVF is encounters in an input string, it is referencing the fourth element of the set LANDUSE. As always, Set references are only appropriate at certain syntactic states--thus the syntactic position (transition state) numbers.

SWITCH REFERENCES: When certain lexemes are encontered, the programmer may desire a switch to be set in a defines manner--this when the device CALCOMP is specified, the switch DEVICE (in the language system) and the switch KDEV (in the FORTRAN code) should be set to `1`, when TEK4010 is encountered, DEVICE/KDEV shold be set to `3`. (See above for SWITCH DEFINITIONS).

```
CALCOMP         DEVICE=1
TEK4010         DEVICE=3
```

default value, a switch reference sets the variable every time it it called. Switch references have the following form:

<switch reference>::=<switch lexeme text> = <integer>

Example: SYSSW=5. When the lexeme defined by `SYSSW=5` is encountered, SYSSW is set to 5.

Switches can be made inaccessable to the user by omitting the lexeme text and/or syntactic position information--thus the switch has no name or it is invalid in any position, in any state.

Example:

```
<lex #>          RANGE   SWITCH:IRANGE/3/
....
                         NEAR            RANGE=2
                         FAR             RANGE=3
```

Thus IRANGE becomes untouchale, can only be set to values 2 and 3, and cannot be seen by the user. Only the FORTRAN variable IRANGE exists; there is no corresponding LINGUIST name.

c) Contexts

While certain lexemes are unambiguous in any position, there are others whose meaning changes depending upon the context in which they are found--that is, depending on the other lexemes which srround them. These are known as "context-sensitive" or "context-defined" lexemes.

Context-defined lexemes can be sensitive either FORWARD or BACKWARD, that is, their meaning can depend either upon the lexemes immediately preceeding or immediately following them. This is defined in the <token> declaration (above).

Since context lexemes are modified by their environment, they are additionally defined in terms of the lexeme that determines their meaning in an instance--the "modifier." Thus, another name for context-described lexemes is "modifieds."

<context lexeme>::=<lex #> <modified> <modifier> <semantics>

<modified>::=<lexeme text of context lexeme> \ <lex # of same>
<modifier>::=<lexeme text of modifier>\<lex #>\ 0

When a context lexeme is encountered, a subroutine in LINGUIST called KHASER chases down the context in which it is found, to return the appropriate lexeme number for the language system. Thus, different contexts form different lexeme numbers--this is how context-sensitive lexemes are differentiated in different environments. In addition, semantic information may also be provided.

Examples:

Suppose that in the <basisc> section we define FILE and OR as context-sensitive lexemes, where OR is sensitive backwards, FILE forwards.

```
30      (2 4, 7)        OR      BACK:
46      (50 37, 19)     FILE    FORW:
```

but corresponding to the FORTRAN variable KOND1. We initialize it to KOND1=2:

```
55                    CONDONE SWITCH:KOND1/2/
```

Now, in the <contexts> section, we define the contexts for FILE and OR:

Lex#	Modified	Modifier	Semantics
57	OR	SELECT	CONDONE=1
59	OR	OR	CONDONE=2
59	OR	AND	ALIAS:
62	OR	PLOT	CONDONE=3
100	FILE	PRINTER	STRING:NPRNT,PRINT(100)
101	FILE	CROSSREFERENCE	SIRING:IDIR,NDIR(1000)

Thus, when OR is encountered following a SELECT (eg: SELECT ELEVATION OR LANDUSE) lexeme number 57 is returned and the switch CONDONE is set to `1`. When OR is preceeded by another OR, CONDONE is set to `2`. When OR is preceeded by an AND, (SHOW ELEVATION AND RAINFALL OR FLOODPLAIN) then this is to be treated the same as an OR...OR situation--this OR...AND is an "alias" for OR...OR, returning the same lexeme number (59).

Similarly, when OR is preceeded by a PLOT command, it is modified to have the meaning `62`, and CONDONE is set to `3`.

When FILE preceeds PRINTER (eg. WRITE FILE ON PRINTER) it references a string sequence composed of array PRINT and counter NPRNT, showing how many PRINT(i)s are being used at any given moment. But, when it is followed by the lexeme CROSSREFERENCE, it returns a different lexeme number (101), and the input line CREATE FILE CROSSREFERENCE, for instance, would 'modify' FILE so that it pointed (corresponded) to the sequence defined (in FORTRAN) by NDIR and IDIR.

It is important to realize that Modifieds have no meaning by themselves, but can be cmbined with other lexemes to yield what are, in effect, compound words.

III. DATA STRUCTURES

We should say a few words on the structure of the data tables employed by the Linguist system. "The Vault" contains all the information for lexical and syntactic analysis in a tree structure.

The Lexeme Table contains the semantic information and has pointers into the vault to allow for conversion between characters and the internal lexeme numbers (specified by the input file) by which lexical items are referenced within the language system.

The "Heap" contains all <type> and <structure> information along

names are EQUIVALENCEd in GLIB's output file.

In general, Lexical analysis involves the translation from a character string to internal form (lexeme numbers). Word recognition is acomplished by climbing a lexical tree. Short-form recongition is also allowed, if there is no ambiguity.

Syntactic analysis entails verifying the legality of a combination of lexical units, through the finite state automaton. After a lexeme is determined legal in the current state, the successor state is then obtained, and analysis proceeds.

Semantic analysis is concerned with interpretation and execution of lexical units, and with the interface wit the mother program (retreats).

The Semantic processor stores parameter values into a command common block, sets switches, allows display and initialization of parameters, etc.

IV. LCG Character Set

(by Nick Chrisman)

In order to facilitate the transportability of the language system, a standard character code must be chosen. Different machines use diferent character sets which are more or less standard. The most universal code is BCD, but this was defined in a number of different ways, each dependent on a particular machine. EBCDIC is the eight-bit code of IBM and nine-track tapes, but is very inefficient.

The most universal character code is ASCII, a seven-bit code associated conventionally with terminal communication. Roughly speaking, within ASCII there are four divisions of characters: control characters, digits and arithmetic operators, upper case alphabetic and lower case alphabetic. For the language system, operating on a broad spectrum of machines, we decided on a six bit subset of ASCII, defined by deleting the control characters and by converting all alphabetics to one case. The "Q-Character Set" is defined by the filowing operations on standard ASCII: (the values below are octal)

```
Map lower case into upper case:          if (code>137) Qcode_code-40
Delete control characters, move blank:   if (code<40)  Qcode_0
Swap alphabetics:                        if (code>101) Qcode_code-100
special case for '@':                    if (code=100) Qcode_40
Otherwise no change:                     if (code>40 &
                                            code(100) Qcode=code
```

The special case of the '@' moves it into the place of the blank, while blank is moved to zero along with control characters. The reasoning for this is that zero is a good code for blank and otherwise one special character would have a lower numeric value than the alphabet. Checks for blanks are the most common in the language system, and the value of zero is usually the easiest for machine to compare.

(User's reference manual for implementation of the Language System)

(Bruce Donald
Laboratory for Computer Graphics and Spatial Analysis)

I. Running GLIB

GLIB, the language maker, will take a table of lexeme definitions and create the <type>, Dimension, Equivalence, ane Data statements in FORTRAN code. This file is then placed in a command file along with the language system and the user's program.

Certain COMMON and EQUIVALENCE statements from this file must also placed before the user's code so that he can reference variables as the Language System assigns them values.

When GLIB is RUN, it asks

NAME THE INPUT FILE .

If all goes well, END OF LEXEME DEFINITIONS is typed, and after the user enters a carriage return, the specifics of the GLIB run are printed--errors, number of switch definitions, etc. Then GLIB asks

NAME THE OUTPUT FILE .

II. The Command File

The command file is typically a list of the routines in the language system, the user program, and a file of BLOCK DATA specification :

The sequence is compiled and loaded with the command

LOAD @ <filename>

Example of a command file:

TYPE COMFIL.CMD

```
BUG.FOR[3'1'1,523],BLOCKO.DAT,BANKER.LAN[3'1'1,1250],BURGLR.LAN[3'1'1,1250],CONVTR.LAN[3'1'1,1250],EXPRSR.LAN[3'1'1,1250],EVAL.LAN[
250],LINGER.LAN[3'1'1,1250],DECODE.LAN[3'1'1,1250],AUDITR.LAN[3'1'1,1250],LCG'10[,'13'1'1]
```

in this file, BUG is the user's program.

Of the Linguist routines we have

BANKER, BURGLR, CONVRTR, EXPRSR, EVAL, LINGER, DECODE, AUDITR, and LCG'10. These are FORTRAN source subroutines (with the exception of LCG'10 which is in MACRO) but if .REL files exist, LOAD will load them in, bypassing compilation.

BLOCKO is a BLOCK DATA and declaration sequence which is required by the language system. It must contain the statement

INCLUDE <Glib output file>

Alternatively, BLOCKO may be modified to contain the output file from GLIB.

Here is a listing of BLOCKO, which INCLUDES the file created by GLIB called BUGLA.DAT:

```
TY BLOCKO.DAT

      BLOCK DATA
C.-.BASIC BLOCK DATA FOR LINGUIST SYSTEM
      COMMON /OPATH/ IPATH,NPATH,OPATH(100)
      COMMON /LEXS/ NXTLEX,LBASIC,MAXLEX,LEXEME(255)
      COMMON /IFSTAK/ LEVIF,QSETUP,LIF,NIF,QIF(6),JIF(6)
      COMMON /STACK/ ISTACK,JSTACK,NSTACK,KEYST(20),KODEST(20),
     1  KPAST(20),KBUGST(20)
      COMMON /FILES/ LOCFRE,LENFRE,MINTED,MAXHEP
      COMMON/VAULT/JVAULT,LROOTD,LDYNAM,ICONTX,JCONTX,LROOT,LVALUE,
     A LTERM,LVAULT,NVAULT(1000)
      COMMON /GUARD/  KKGARD(20)
C             THE HEAP STORAGE APPEARS IN COMMON HEAP,BUT ITSPURE COPY
C.            IS KEPT IN THE BLOCK ENTERED IN THIS SECTION AS PURITY
      COMMON /PURITY/ LUMP(1000)
C
      LOGICAL QIF,QSETUP
      COMMON/EXTINT/INCODE(127)
C
      COMMON/INTEXT/INTOON(64)
      COMMON /UNITS/ IOIOIO(2)
      INCLUDE 'BUGLA.DAT[3111,523]'
      DATA INCODE/32*0,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
     A 48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,32,1,2,3,4,5,6,
     B 7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,
     C 29,30,31,32,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
     D 21,22,23,24,25,26,27,28,29,30,31/
      DATA INTOON/1H ,1HA,1HB,1HC,1HD,1HE,1HF,1HG,1HH,1HI,1HJ,1HK,1HL,
     A 1HM,1HN,1HO,1HP,1HQ,1HR,1HS,1HT,1HU,1HV,1HW,1HX,1HY,1HZ,1H[,1H\,
     B 1H],1H^,1H_,1H@,1H1,1H",1H#,1H$,1H%,1H&,1H',1H(,1H),1H*,1H+,1H,,
     C 1H-,1H.,1H/,1H0,1H1,1H2,1H3,1H4,1H5,1H6,1H7,1H8,1H9,1H:,1H;,1H<,
     D 1H=,1H>,1H?/
      DATA IOIOIO/5,5/
      DATA NPATH/100/
      DATA ISTACK,JSTACK,NSTACK /0,1,20/
      DATA LEVIF,QSETUP,LIF,NIF,QIF(1),JIF(1) /0,.FALSE.,1,6,.TRUE.,0/
      DATA MAXLEX,MAXHEP /255,1000/
      DATA LVAULT/1000/
      END
```

The input file for GLIB is well documented above. the output file (here, BUGLA.DAT) looks like this:

Example:

```
TY BUGLA.DAT

C.-.BLOCKDATA FOR LINGUIST GENERATED  5Jun78
      DIMENSION    IVQA(167),IVQB(30),IVQC(100)
```

```
     +ITELL),(LUMP(20),KSWIT)
       EQUIVALENCE  (IVQA(1),NVAULT(1)),(IVQB(1),NVAULT(168)),(IVQC(1),L
     +EXEME(1))
       DATA ALPH/7./,LUMP(1)/279570/,BET/14.0/,LUMP(3)/280594/,ITELL/1/,
     +TASK1/4.0/,LUMP(5)/272402/,TASK2/5.0/,LUMP(7)/273426/,IOP/2/,LUMP(
     +9)/275473/,IOP2/3/,LUMP(11)/276497/,IOP3/55/,LUMP(13)/277521/,KSWI
     +T/1/,LUMP(15)/808049/,LUMP(17)/90113/,LUMP(18)/94210/,LUMP(19)/820
     +337/,LUMP(21)/331777/,LUMP(22)/409610/,ICONTX/197/,JCONTX/197/,JVA
     +ULT/197/,LROOT/9/,LDYNAM/0/,LVALUE/3/,LTERM/1/,MINTED/22/,LBASIC/1
     +00/
       DATA KKGARD/32,64,16,32,64,4096,5,64,4,128,2,128,63,255,8192,512,
     +16,16,4,256/
       DATA IVQA/-1,15870,-98,84563243,15870,86021377,90242434,16130,996
     +1473,4194894,-16876577,130558,4,14777733,16133,7078476,6030352,-17
     +990689,10419837,513,10468862,262127358,1036,-23298081,10419837,526
     +4140,8651347,-2294817,17787,5018062,4130172,15868,594,-94568993,76
     +6,4019076,10583813,11534338,-19006497,10419837,21761,10468862,2621
     +27358,27000851,13110024,-20054561,1,983,253,21760773,3213,17829078
     +,3333,3410,3458,16256450,-3572769,17787,85521541,2163972,15748,352
     +1,-4588577,17787,1655122,2163972,15748,3279,4354,4426,20189570,-56
     +37153,17787,85521541,2163972,15748,4481,-6652961,17787,1655122,216
     +3972,15748,25692953,5326,5396,24122703,-7570465,17787,138,2163972,
     +15748,5462,-8651809,17787,21634,2163972,15748,2839,-34865217,10419
     +837,156250633,5,28049425,-21102625,1,86612,33030164,29629125,-2215
     +1713,19103357,780,27967018,6849,7251,7307,31726833,-10650625,10419
     +837,10468862,262127358,7410,-11699201,10419837,10468862,262127358,
     +34078734,-24313889,10419837,256922380,40108048,37232783,35922121,-
     +3245057,17787,2163972,15748,8388,-4293633,17787,2163972,15748,8325
     +,38806409,-5342209,17787,2163972,15748,9092,-6390785,17787,2163972
     +,15748,42991634,41690693,-7406593,17787,2163972,15748,9807,-845516
     +9,17787,2163972,15748,48758799,44574990,-85001217,10419837/
       DATA IVQB/8486277,260144773,10512,46148273,-13730817,114173,14614
     +011,16123,47458994,-14779393,114173,14614011,16123,10931,-15827969
     +,114173,14614011,16123,50331658,-87163937,10419837,86864,8486277,2
     +60144773,23,-12714529,766,38152,10600198,5462628/
       DATA IVQC/0,33574940,35672121,37769279,39866440,41963598,46157911
     +,44060765,0,20597,28794,33669313,37036,45233,53430,33669131,4114,1
     +2327,33566766,35655784,61549,61464,61567,0,0,0,0,0,0,0,0,0,77923,0
     +,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
     +,0,0,0,0,0,0,0,0,0,0,0,0,0,0,77990,0,33673403,0,0,0,0,0,0,33669154,0
     +,0,0,0,0,0,0,0,0,78021/
C.-.ENDBLOCK
```

III. Use of Glib Output and How to Use the Stack

The user must preceed his own code with the following elements:

A Common statement to reference the "heap",
A Common statement to reference the Stack,
An Equivalence statement setting the names of his own variables equal to those names they are assigned in the Heap.
Any declaration statements generated by GLIB (i.e., DIMENSION, <type>) which were specified in <object> definitions in the language table.

Example: (from above, the user will take)

```
       COMMON /HEAP/ LUMP (1000)
       EQUIVALENCE  (LUMP(2),ALPH),(LUMP(4),BET),(LUMP(6),TASK1),(LUMP(8
     +),TASK2),(LUMP(10),IOP),(LUMP(12),IOP2),(LUMP(14),IOP3),(LUMP(16),
     +ITELL),(LUMP(20),KSWIT)
```

A pure copy of the heap is kept in the common /PURITY/ for commands such as INITIAL which restore the system to its virgin state.

Initial values are assigned to user variables in BUGLA by the statement

```
      DATA ALPH/7./,LUMP(1)/279570/,BET/14.0/,LUMP(3)/280594/,ITELL/1/,
     +TASK1/4.0/,LUMP(5)/272402/,TASK2/5.0/,LUMP(7)/273426/,IOP/2/,LUMP(
     +9)/275473/,IOP2/3/,LUMP(11)/276497/,IOP3/55/,LUMP(13)/277521/,KSWI
     +T/1/,LUMP(15)/808049/,LUMP(17)/90113/,LUMP(18)/94210/,LUMP(19)/820
     +337/,LUMP(21)/331777/,LUMP(22)/409610/,ICONTX/197/,JCONTX/197/,JVA
     +ULT/197/,LROOT/9/,LDYNAM/0/,LVALUE/3/,LTERM/1/,MINTED/22/,LBASIC/1
     +00/
```

The user must be sure to reference /HEAP/ and not

```
COMMON /PURITY/ LUMP(1000)
```

Since then his variables will never change!

In the above example there are no arrays or explicit declarations.

But if the GLIB input file contained, for instance

```
34      (4,15)  TELL    SW: LOGICAL FTELL/.TRUE./
```

and

```
40      (2, 4)  WINDOW  WINDX (100) /100*1./
```

Then GLIB would generate (in the file BUGLA.DAT)

```
        DIMENSION WINDX(100)
        LOGICAL FTELL
```

and

```
        DATA FTELL/.TRUE./, (WINDX(I), I=1,100) /100*1./,...
```

Also, FTELL and WINDX would be EQUIVALENCED to elements of the HEAP (i.e., LUMP).

The COMMON reference to the stack is as follows:

```
        COMMON /STACK/ ISTACK,JSTACK,NSTACK,KEYST(20),KODEST(20),
     1    KPAST(20),KBUGST(20)
```

The user must look at the Stack, since upon retreat it is necessary to know what command string has been typed in. KODEST contains the lexeme numbers of the lexical items currently on the stack. Thus if the GLIB input file contains

```
3       (1,5)   OPEN    ASS REGR ONCE:
40      (5,55)  CROSSREFERENCE  FORW:
41      (55,2)  FILE    NULL:
50      (2,0)   =       VALUE STRING:
.
.
.
```

.
.
101 CROSSREFERENCE FILE NAMFIL /5HNONE /

Then the KODEST stack would contain

I	KODEST(I)
1	3
2	40
3	41
4	0
5	'01'

if the input string were OPEN CROSSREFERENCE FILE'<filename>';

KEYST is an array containing the transition states correspondng to the lexeme codes in KODEST. ISTACK gives the length of the stack, and JSTACK tells how far BINDER (which has the task of placing data in variables, that is, doing ASSIGNS) has gotten in the semantic processing.

KURBUG returns what type of retreat has been executed:

KURBUG	Type of Retreat
[1	No retreat. User will never see KURBUG=1]
2	Progressing Once Immediate [Semantic Verb Retreat Before Execution]
3	Progressing Once Deferred [semantic Verb Retreat After Execution]
4	Progressing Each Immediate [Semantic Objet Retreat Before Execution]
5	Progressing Each Deferred [Semantic Object Retreat After Execution]
6	Regressing Once [Syntactic Veb Retreat]
7	Regressing each [Syntactic Object Retreat]

Another array that may be referenced is KPATH, whih is available in the common

COMMON /OPATH/ IPATH, NPATH, KPATH(100)

KPATH is written in to and contains the lexical string currently being processed. Most users would not need to access KPATH, sonce the lexical items are availale in KODEST, and the data inputed for ASSIGNs (eg., the <filename>, above, or arithmetic expressins) are automatically placed in the corresponding FORTHAN variables. For instance, the filename typed in in the above example would be placed in the user variable NAMFIL, and accessible in the language environment as CROSSREFERENCE. But if, for instance, there was an illegal character in the filename and an error message which printed out a cursor pointing to the offending letter was desired, then KPATH would contain the text:

OPEN CROSSREFERENCE FILE:'MORFIL/DAT' (gotten through KPATH)

UPDATE

GLIB was modfied on 15 June, 1978 to more easily facilitate user access of variables. After asking for the OUTPUT NAME, GLIB will then type

DO YOU WANT BLOCKO COPIED WITH GLIB'S OUTPUT?

If the user responds 'YES', then BLOCKO (see above) will be copied as the output file, and in place of the INCLUDE will come the block data and equivalence statements generated by linguist.

GLIB then asks,

NAME THE LUMP (HEAP) FILE
(TYPE NOFILE FOR NONE)

If the user responds with a valid filename, the <type>, DIMENSION, and EQUIVALENCE statements generated by Linguist will be written into it. The user may then simply start his user subroutines with this file in order to access his FORTRAN variables.

By typing N[O] and NOFILE, GLIB is forced to work as before.

GLIB with a preface section is now available as GLIB.SAV[3111,523]. Old language files can be made to run with a minimun of modification, and there are some new features listed below.

Reactionaries can continue using the old GLIB, which is now called OGLIB.SAV[3111,523].

MAIL any problems or questions to DONALD@HARV-10.

--Bruce Donald

(USER'S REFERENCE MANUAL FOR GLIB, WITH THE PREFACE SECTION IMPLEMENTED)

1) The preface section of GLIB has now beeen implemented. It is terminated, as are the definition and context sections, with a <CRLF> END:

```
<preface>
END
<definitions>
END
```

ii) The Preface section starts up taking input from the user terminal--in this mode, value states, define lexeme states, the evaluable state, and file options and specification will be given. GLIB will not come back to the terminal for input, except in case of error.

iii) The language system is present in GLIB, and thus '?' will return a list of valid commands and options, as always. The category

*** <value> includes (a) integers (e.g., for unit numbers),
(b) strings (for filename specifications)
(c) tuples, for define and value state definition.

Due to the absence of the Expression evaluator and BANKER, there are some diferences: Quotes around filename specific are ILLEGAL. Specify the same way one would to the monitor.

Colons (before values, and after lexeme specification such as VALUE INTEGER[:]) and semi-colons are not enforced, but are acceptable. This is only true in the preface section.

Some character must separate the value from the lexeme--a colon or a space is usual.

The preface language system prompts with a '!' until control is transfered to the language file.

Entire language systems may be specified from the teletype. However, error recovery in all but the preface section is very poor.

File Specifications

The following are valid file specifications and options:

Command String	Example	Explanation
INPUT FILE	INP FILE: DHA.LAN	Specifies the input language file. Control is immediately transfered to that file. Note that all Preface commands Are valid in the file--thus, if the output parameters are constant, the user need only specify the INPUT FILE. Note that control is transferred while GLIB is still expecting Preface commands.
END	END	End preface definitions, and start the lexeme definition.
INPUT UNIT	INP UNIT: 1	Specifies the FORTRAN logical device number. Default is 1.
OUTPUT FILENAME	OUT FILE: BLK.LAN	Specifies the output file.
OUTPUT UNIT	OUTPUT UNIT: 5	[illegible]

COPY BLOCKO

```
COPY BLOCKO
```

Tells the GLIB processor to copy BLOCKO along with its newly generated data, so that it all may be INCLUDED in the mother BLOCK DATA segment.

WRITE LUMPFILE

```
WRITE LUMPFILE: LUMP.LAN[311,523]
```

Names the optional output file containing the <type>, DIMENSION, and Eqivalences to the heap.

WRITE BINFILE

```
WRITE BIN: TEST.BIN
```

The binary file option is for use with the dynamic linguist loader. It contains a binary dump of fortran variable names, their heap address, and number of dimensions.

This is read (for dynamic loading) and kept for use by the dictionary routines (call by lookup rather than equivalence).

For use with the standard method of compilation, the BINFILE option should not be used.

The fortran names are dumped in the Q character set.

The FILING option will list heap addresses of variables, switches, and sets as the input is echoed.

Help File Specification

HELP FILENAME
HELP UNIT

```
HELP UNIT:1
HELP FILENAME: HLPMSG.TXT
```

Soon the Linguist package will contain a help subroutine. The filename specified in the preface section will go into a special common block invoked when HELP is typed by the user.

Language Specification

The Preface section will allow value states, define lexemes, and the evaluable state to be given.

VALUE

```
VALUE <type>: <tuples>

VALUE REAL:   (41, 0) (30, 44 0)
VALUE STRING: (41 42, 0)
```

is line-transparant, so more
tuples may follow a carriage return.

EVALUABLE STATE

EVAL STATE: 106

Specifies what state is reserved for the evaluable state, i.e., where lexemes may be parsed when a <value> is expected.

Define lexeme specifications work as follows:

GLIB must be supplied with a define lexeme number, a Forced Final State (i.e., what state you would like to be in when the DEFINE is done) and a set of state transitions.

The Define Lexeme Number corresponds to the lexeme number of a define verb to be specified later in the second section. There may be several, or many define classes, each with a unique define lexeme number.

The Preface processor wil look for state tuples for define only after it has both a final state and a define lexeme number. The order of specification does not matter.

The tuples are then expected as the next thing the processor sees, although they may be on a separate line.

DEFINE LEXEME
or
DEFINE LEXEME NUMBER

DEF LEX: 20
DEF LEX NUM: 20

Specifies the define lexeme number.

FINAL STATE

FINAL STATE: 33

Specifies the final state for the last or next define lexeme. Final states and define numbers are taken in pairs by the preface section.

(NOTE: The old manner of generating values and defines will generate an error in the New Glib.)

See the example at the end of this manual for more on define and value specification.

FILING FILING

The FILING switch may be turned on from the Preface section, and causes every line to be echoed on the console as the file is read.
The FILING switch takes no arguements.

Example:

Here we have a PROTOCOL of a GLIB run. In it, the user has typed '?' to examine the language system, and then specified his output file, 'SNUG.DAT'.

Next, he orders GLIB to copy BLOCKO aong with his language info, and writes a file containing the Heap (LUMP) called SNUGL[.DAT]

A helpfile is given, which GLIB COMMONs and DATAs, and all input, output, and help I/O units are left at their default of '1'.

Next, the Eval state is specified to be 100.

When the input file DMA.LAN is given, control immediatly shifts to that file's preface section. Note that from DMA.LAN, control could once more be switched to yet another file, while still in the preface section.

The file DMA.LAN is listed at the bottom of the PROTOCOL. In it's preface section, one define lexeme, final state, and state transition set are given, and the VALUE states are defined.

After END, control passes over to the lexeme and context processors, which work as before...except:

ATTEMPTS TO DO VALUE OR DEFINE SPECIFICATION IN THE LEXEME SECTION WILL CAUSE ERRORS.

The Lump file produced is also listed.

Those who wish to specify no options, but still implement the preface section, may type

```
OUT FILE: <filename>
INP FILE: <file2>
```

and the output will be exactly the same as the original GLIB.

```
WSTART

WELCOME TO GLIB, Donald
```

CURRENT MODE- INPUT FROM TTY:

```
TO TRANSFER CONTROL TO YOUR LANG FILE--ENCLOSING  QUOTES
 FOR FILENAME SPECS ARE ILLEGAL IN THE PREFACE  SECTION.
 BE SURE TO SPECIFY OUTPUT FILE AND OPTIONS  EITHER HERE OR IN YOUR LANG FILE...

PREFACE, LEX DEF, AND CONTEXT MAY BE SPECIFIED FROM
TTY--CURRENT STATE :     PREFACE SECTION


!?
***  HELP
***  COPY
***  INPUT
***  DEFINE
***  WRITE
***  END
***  EVALUABLE
***  FINAL
***  OUTPUT
***  VALUE
!OUTPUT FILENAME: SNUG.DAT[3111,523]
!COPY BLOCKO
!WRITE ?
***  LUMPFILE
!LUMP: SNUGL
!HELP?
***  FILENAME
***  UNIT
!FILENAME? ,
***  <VALUE>
!HELPM.MSG[3111,641]
!EVAL STATE: 100
!INPUT FILE: DMA.LAN
  TRANSFERING CONTROL...


END OF LEXEME DEFINITIONS


TABLES REDUCED FROM  632 TO   692
      10 SWITCHES AND SETS       71 REFERENCES        33 SPACE FOR SCRATCH
HEAP ALLOCATED UP TO LOCATION    242

 MISSION ACCOMPLISHED


END OF EXECUTION
CPU TIME: 18.05 ELAPSED TIME: 2:37.65
EXIT

.
.DIR SNUG??.*

DSKB:
SNUG   DAT    24    30  05-Jul-78  10:59
SNUGL  DAT     2     5  05-Jul-78  10:59

   Total =    26    35


.TY SNUGL.DAT
      COMMON/HEAP/ LUMP(1000)
C.-.BLOCKDATA FOR LINGUIST GENERATED  5Jul78
      LOGICAL  QLAND,QELEV,QFLOW,QACR
```

```
      +ITL(25),QAGR(4),LEXOFF(3),LEXAGR(3),LEXOUT(3),UNITS(5),NMPRNT(8),N
      +MPHB(8),NMCDB(8)
        EQUIVALENCE  (LUMP(2),WINDOW(1)),(LUMP(7),VWPORT(1)),(LUMP(12),SC
      +ALE),(LUMP(14),ANGLE),(LUMP(16),NTITL),(LUMP(17),KTITL(1)),(LUMP(4
      +3),HEIGHT),(LUMP(45),IOMSG),(LUMP(47),TOLAR),(LUMP(49),LEXOFF(1)),
      +(LUMP(53),LEXAGR(1)),(LUMP(57),LEXOUT(1)),(LUMP(61),UNITS(1)),(LUM
      +P(67),NPRNTR),(LUMP(68),NMPRNT(1)),(LUMP(77),IOPRNT),(LUMP(79),LNM
      +PHB),(LUMP(80),NMPHB(1)),(LUMP(89),IOPHB),(LUMP(91),IOPHG),(LUMP(9
      +3),LNMCDB),(LUMP(94),NMCDB(1)),(LUMP(103),IOCDB),(LUMP(105),IOCDG)
      +,(LUMP(107),QLAND(1)),(LUMP(176),QFLOOD(1)),(LUMP(181),QELEV(1)),(
      +LUMP(204),KDEV),(LUMP(210),KUNIT),(LUMP(217),KOND1),(LUMP(221),KON
      +D2),(LUMP(225),QAGR(1)),(LUMP(234),KINPHB),(LUMP(239),KINCDB)
C.-.ENDBLOCK

.TE DMA.LAN
HTECO.22.6F
#SEND$-60T10T$$
DEFINE LEXEME NUMBER: 25
FINAL STATE: 33          (33,40) (100 34,0)
VALUE:              (40,0)
VALUE INT:          (41,0)
VALUE REAL:         (42,0)
VALUE STRING:       (43,0)
VALUE LOGICAL:      (44,0)
VALUE CLOSED:       (48,0) (8,9 0)(21,22 0) (25,26 0)
                    (17,18 0)(11,12 0)(27,28 0)
                    (14,15 0)(19,20 0)(23,24 0)
VALUE REF:          (47, 0)
END
66      (35 2 100,0)    ACC     LANDUSE[1]
67      (35 2 100,0)    ACP     LANDUSE[2]
68      (35 2 100,0)    AR      LANDUSE[3]
69      (35 2 100,0)    AVF     LANDUSE[4]
70      (35 2 100,0)    AVV     LANDUSE[5]
71      (35 2 100,0)    BBR     LANDUSE[6]
72      (35 2 100,0)    BEQ     LANDUSE[7]
73      (35 2 100,0)    BES     LANDUSE[8]
74      (35 2 100,0)    BT      LANDUSE[9]
#^C^C
.
```

Example:

Here we have another example of a language using the preface section. This language definition is, in effect, on 'autopilot', thus, the user must say only 'IN FILE: <filename><CR>' to start processing.

Output, input, and Lump file options are constant, and Value and define states are specified in the preface section.

(Note on DEFINEs: The define lexeme number specifies the define lexeme class--thus, multiple define classes with different states are allowed. The final state -- the 'state you would like to be in' after you say DEFINE--is a value state. (24).

The transition states for DEFINE in this language first allow the define to take place (a transition from 1 (DEFINE) to 33 (forced state) to 24 (VALUE) to 0 (final state)) and then allow DELETEs (34) and SHOWs (3) of the defined symbols).

```
COPY BLOCKO
OUTPUT FI:CYCGAB.BLK
WRITE LUMP:CYCGAB.COM
DEF LEX N:10
FIN ST:33          (33,24)(3 34,0)
VALUE INT:         (21,0)(43,41 0)
VALUE REAL:        (22,0)
VALUE STRING:      (23,0)
VALUE:             (24,0)
END
1        (1 51,0)(2,0)              RUN                        NOOP PROGRES:
2        (1 51,5 0)(2,0)            STEP                       ASSIGN REG ONCE:
3        (1 51,6)(2,0)(3,30)        PUBLISH                    BACK:
4        (1 51,0)(2,0)              QUIT                       NOOP PROGRES:
5        (51,7)(2,0)                MAKE                       ASSIGN:
6        (1 51,8)(2,0)(3,31)        OPEN                       BACK:
6        (1 51,18)(2,0)             CREATE                     ALIAS:
8        (51,10 11)(2,0)            DO                         ASSIGN:
9        (10,11)(2,0)(28,27)(38,37)
                                    NO                         NOASSIGN:
10       (1 51 2,0)                 DEFINE                     DEFINE:
11       (2,0)(1 51,34)             DELETE                     DELETE:
12       (51,7 11 17 27 37 42 49)RESTORE                       FOR:
13       (1 51,3)(2,0)              SHOW                       SHOW:
14       (1 51,0)(2,0)              CLEAR                      CLEAR:
15       (34 49,0)                  ALL                        NULL:
18       (7,21)(2 3,0)              IGNOREDPOLYGONS            SEQ INT:NIGNOR,IGNORE(20)/1,20*0/
19       (7,21)(2 3,0)              NULLPOLYGON                INTEGER:NULL/0/
20       (2 31,0)
         FILES    SET:POLYO,CHAINO,REPORT,SYMAP,CALFRM,INPUTN/6*.FALSE./
21       (2 3,0)
         ACTIONS SET:SELECT,RENUM/2*.FALSE./
22       (11,12 0)(3,32)(2,0)       SELECTION                  ACTIONS[1]
23       (11,46 0)(2,0)             RENUMBERING                ACTIONS[2]
24       (8 18,29)(6,16 41 0)(2 9,0)(3,30 45)(7,44)
         POLYGONS                   BACK:
25       (8 18,19)(2 9,0)(3,30)     CHAINS                     BACK:
26       (8 18,19 0)(2,0)(3,30)     REPORT                     FILES[3]
27       (8 18,39)(2,0)(3,30)       SYMAP                      FILES[4]
28                                  CALFORM                    FILES[5]
29       (8,19)(2,0)(3,30)          INPUT                      FILES[6]
30       (12,9)(2,0)                OF                         ASSIGN:
31       (41,40)(2,0)               ON                         NULL:
32       (16,42)(29,28 27)(39,38 37)(46,47)(2,0)
                                    WITH                       NULL:
33       (37 2 3,0)                 REFERENCE                  LOGICAL:REFER/.TRUE./
34       (27 2 3,0)                 MEASURES                   LOGICAL:MEASUR/.FALSE./
35       (27 2 3,0)                 POINTS                     LOGICAL:POINTO/.TRUE./
36       (6,15)(2,0)                CYCLING                    STATSW=1
37       (6,15)(2,0)                ALLOCATION                 STATSW=2
38       (6,15)(2,0)                MEMORY                     STATSW=3
39       (9 2,0)                    NODES                      BACK:
40       (5,21)(2 3,0)              THROUGH                    INTEGER:NSTEP/1/
41       (42,43)(2 3,0)
         IDENTIFIERS                SEQ INT:NPUBL,NAMEP(20)/0,20*0/
42       (17,21)(2 3,0)
         NAMES                      SEQ INT:NCHOSE,NAMES(50)/0,50*0/
43       (40 19 29 39,23)(2,0)(30,0)
                                    FILENAME                   BACK:
44       (40 19 29 39,21)(2,0)(30,0)
                                    UNITS                      BACK:
45       (15,41 0)(2,0)             STATISTICS                 PUBSW=1
46                                  PUBSW                      SW:NPUB/1/
47
```

```
49      (2 32,0)                    TYPE                        SW:INCLUD/1/
50      (12,17)(2,0)                INCLUDING                   TYPE=1
51      (12,17)(2,0)                EXCLUDING                   TYPE=2
52      (44,22)(2 45,0)             TABLESIZE                   BACK:
53      (7,44)(3,45)(2,0)           RINGSANDLINKS               NULL:
54      (47,21)(2 3,0)              FIRSTNUMBER                 INT:INITAL/1/
55      (47,21)(2 3,0)              INCREMENT                   INT:INCREM/1/
END
61      OPEN            0                       ASSIGN REG EACH:
62      OPEN            SHOW                    NULL:
63      PUBLISH         0                       ASSIGN REG EACH:
64      PUBLISH         SHOW                    NULL:
65      CHAINS          OPEN                    FILES[2]
66      CHAINS          OF                      OBJECT=1
67      NODES           OF                      OBJECT=2
68      POLYGONS        OF                      OBJECT=3
69      POLYGONS        OPEN                    FILES[1]
70      POLYGONS        PUBLISH                 PUBSW=2
71      FILENAME        POLYGONS                STR:ILPOLY,IDPOLY(8)
72      FILENAME        CHAINS                  STR:ILCHAI,IDCHAI(8)
73      FILENAME        REPORT                  STR:ILREPT,IDREPT(8)
74      FILENAME        SYMAP                   STR:ILSYMP,IDSYMP(8)
75      FILENAME        INPUT                   STR:ILINPT,IDINPT(8)
76      FILENAME        ON                      STR:ILPUBL,IDPUBL(8)
77      UNITS           POLYGONS                INT:IOPOLY(2)/3,13/
78      UNITS           CHAINS                  INT:IOCHAI(2)/2,12/
79      UNITS           REPORT                  INT:IOREPT/4/
80      UNITS           SYMAP                   INT:IOSYMP/8/
81      UNITS           INPUT                   INT:IOINPT(2)/1,11/
82      UNITS           ON                      INT:IOPUB/5/
83      TABLESIZE       POLYGONS                REAL:PTABLE/120./
84      TABLESIZE       RINGSANDLINKS           REAL:RTABLE/500./
END
```