PRØF. M. KLINE
25 WAVERLY PLACE

AEC Computing and Applied Mathematics Center

AEC RESEARCH AND DEVELOPMENT REPORT

TID-4500
35th Ed.

NYO-1480-9

THE NU-SPEAK SYSTEM

by

Judith Glasner, Stanley Ocken,
David Rosenberg, Jack Schwartz,
George Shapiro and Alan Silverman

November 1964

Courant Institute of Mathematical Sciences

NEW YORK UNIVERSITY

NEW YORK, NEW YORK

NYO-1480-9
uncat.

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, express or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

UNCLASSIFIED

AEC Computing and Applied Mathematics Center
Courant Institute of Mathematical Sciences
New York University

TID-4500
35th Ed.

NYO-1480-9

THE NU-SPEAK SYSTEM

by

Judith Glasner, Stanley Ocken,
David Rosenberg, Jack Schwartz,
George Shapiro and Alan Silverman

November 1964

Contract No. AT(30-1)-1480

UNCLASSIFIED

ABSTRACT

Nu-Speak is a list processing language embedded in FAP. This manual describes the use of the Nu-Speak system, the use of a corresponding Fortran version of the system, and some applications.

TABLE OF CONTENTS

A. INTRODUCTION

A1. Fortran Nu-Speak 4

A2. The Chaining Requirement 23

B. FAP NU-SPEAK

B1. Introduction 24

B2. Allocation of Core Storage in Nu-Speak 30

B3. Use of the Nu-Speak Macros of the First (Recursion) Group 31

B4. The Second (String-Manipulation) Set of Nu-Speak Macros 49

B5. The Nu-Speak Macros of the Third (Sublist Manipulating) Group 63

B6. Principles of Operation of the Automatic Erasing Mechanisms 67

B7. The Macros FREESP(ace) and FREEHD 75

B8. Miscellaneous Macros 77

B9. Forbidden Macro-Words and Entry Symbols 77

B10. Special Procedures for Dealing with Self-Reflexive List Structures 78

B11. The Form of a Nu-Speak Deck 80

C1. The Auxiliary Routines WRTLIS and INLSTR 82
a) Output of List Structures
b) Input of List Structures

C2. An Auxiliary Package of Subroutines for Letter Manipulation 89

C3. An Example of Nu-Speak Applications Programming: PØLPAC 95

INDEX 102

THE NU-SPEAK SYSTEM

by

Judith Glasner, Stanley Ocken, David Rosenberg,
Jack Schwartz, George Shapiro and Alan Silverman

A. Introduction.

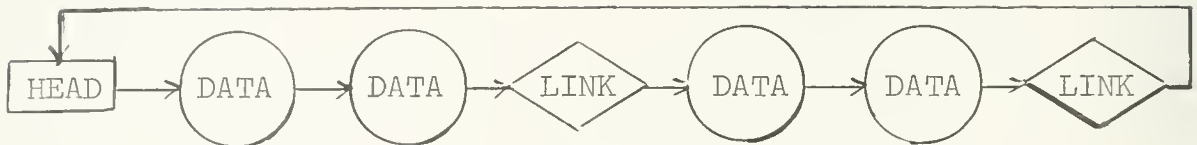
The Nu-Speak list processing system is available in two versions, a Fortran version and a FAP version. In the following report, the use and some of the underlying structure of both versions of Nu-Speak will be outlined. While many users will prefer to write Nu-Speak programs in Fortran, it should be noted that only in the FAP version can full advantage be taken of Nu-Speak flexibility.

A1. Fortran Nu-Speak.

To the two types of objects normally considered in Fortran — variables and arrays — Fortran Nu-Speak adds a third — *the list*. The logical structure of a Nu-Speak list is as follows:

Each list consists of a finite number of *items*. Each item is of one of three types: a *listhead*, a *link*, or a *data item*. Each list contains one and only one head, called the *head of the list*. The remaining items of the list follow in serial order, the last item being followed by

the head of the list. Thus, lists have a kind of 'looped' or 'circular' form. A list may contain no items other than its head in which case it is called a nullist. If squares designate listheads, rhomboids designate links, and circles designate data items, a typical list might be represented as follows



Each list may have one or several names; the name of a list is an ordinary Fortran variable. To the ordinary Fortran system, Nu-Speak adds a number of functions and subroutines which enable the manipulation of lists. These subroutines and functions are as follows: NEXT, PREV, READY, SUBLIS, ERASER, HANG, UNHANG, NAMLIS, CØPYTØ, JUMPTØ, REMØVE, CUPL, NCPLTØ, CLERNM, INSERT, TYPE, DSTRØY, CREATE, SAVER, GETTER, NØWW, and CURRNT. A final set of subroutines, CØMEIN and CØMØUT, enable recursive subroutines to be written within the Nu-Speak Fortran system.

One other concept, not present in the ordinary use of Fortran, enters the Nu-Speak Fortran system. At any given step of Nu-Speak list processing, we will say that the computer is looking at or examining a given item on a given list. The manner in which the various Nu-Speak list processes affect the list item at which the computer is looking will be

explained in what follows.

The initialising command

CALL READY

should be the first command of any Fortran/Nu-Speak main program. This prepares the computer for list processing and recursive function usage.

A list is created by the Fortran statement

CALL CREATE (LIST) ;

here LIST is a Fortran variable which becomes the name of the newly created list. The list, when created, is a nullist, The computer is left examining the head of the newly created nullist.

Suppose now that the computer is examining a given item on a given list. The Fortran statement

A = TYPE (DUMMY)

will make the variable A positive if the item currently being examined is a data item, zero if the item currently being examined is a link, and negative if the item currently being examined is a head. Here, DUMMY is any arbitrary Fortran variable, needed only for compatibility between Fortran and the underlying Nu-Speak programs. After executing this statement, the computer is left examining the same list element as before. The function TYPE can also be used in Fortran statements of IF-kind. Thus, the statement

IF (TYPE (DUMMY)) 1,2,3

will transfer control to statements 1, 2 or 3 of a program, depending on whether the list element under examination by the computer is a head, a link, or a data item.

Suppose again that the computer is examining a given item on a given list. The Fortran statement

```
CALL   NEXT (DUMMY)
```

will cause the computer to examine the next item on the given list. The Fortran statement CALL PREV(DUMMY) will cause the computer to examine the previous item on the given list. The Fortran statements

```
CURREN = PREV (DUMMY)
```

and

```
CURREN = NEXT (DUMMY)
```

will cause the computer to examine the previous and the next element on the given list respectively, and will set the variable CURREN equal to the previous or the next list item respectively. (If this previous (resp. next) item is a listhead or a link, the variable CURREN will be set equal to a mysterious octal integer.)

The Nu-Speak functions can be compounded in the manner usual in Fortran. Thus the statement

```
VAR = TYPE (PREV(NEXT(DUMMY)))
```

is legitimate, and has the same effect as

```
VAR = TYPE(DUMMY) .
```

The Fortran statement

```
CALL   INSERT(E)
```

will insert the value of the expression E on the list currently being examined, immediately after the item currently being examined. The computer will be left examining the newly inserted item. Thus, e.g., suppose that the variables A(1),...,A(6) are the data words NØW, IS, TIME, FØR, ALL, GØØD, read in from the input tape using an A6 format. Then the code

```
      CALL    CREATE(LIST1)
      DØ 7    I = 1,6
      CALL    INSERT(A(I))
    7 CALL    PREV(DUMMY)
```

will create a Nu-Speak list named LIST1, and containing the six data items GØØD, ALL, FØR, TIME, IS, NØW in that order. Note also that at the end of this particular do-loop, the computer will be left examining the head of LIST1. The function INSERT can also be used in the form

```
B = INSERT(A) ,
```

which has the same effect as

```
CALL  INSERT(A)
B = A
```

The inverse function to INSERT is performed by REMØVE, which may be used either in the form

```
CALL  REMØVE (DUMMY)
```

or

```
VAR = REMØVE (DUMMY)
```

Both of these lines of code remove from a list the single particular item at which the computer is looking, and cause the computer to look at the immediately following item. The second of the above lines of code also sets the variable VAR equal to the value of the immediately following data item (or to a mysterious octal number if the following item is a link or a head).

The function JUMPTØ is used in the form

```
CALL   JUMPTØ (LIST) ,
```

whose LIST is the name of a previously created list. This code will cause the computer to examine the head of the list named LIST. Thus, e.g., to count the number of items in a list named LIST, we may use the following code.

```
        CALL   JUMPTØ (LIST)
        N = 0
 11     IF  (TYPE(NEXT(DUMMY)))2,1,1
  1     N = N + 1
        GØ TØ 11
  2     ... [here follows the rest of program]
```

If it is desired to erase a list once created, the code

```
CALL   ERASER (LIST)
```

may be used. This will cause the list named LIST to be erased, and the storage cells which this list formerly occupied to be returned to a master 'junkpile' of space available for re-use. It is, of course, quite important

to erase lists when the data represented by them is no longer of use. After a call to ERASER of this sort, the computer will be left looking at the same data word of the same list at which it was looking at the time of the call. If, however, the list erased happens to contain the element at which the computer was looking, then, after the erasure, the computer is no longer looking at any item. To recommence list processing, a statement either of the type

```
CALL    JUMPTØ (LIST)
```

or

```
CALL    GETTER(VAR)  (cf. below)
```

or

```
CALL    CREATE (LIST)
```

must be executed. <u>A list should never be erased more than once, nor should a nonexistent list ever be erased</u>.

If the computer is looking at the head of a list, the statement

```
CALL    CØPYTØ (VAR)
```

will cause it to produce a copy of the list at whose head it is looking, and to make the variable VAR into a name of the copied list. After the command is executed, the computer will be left looking at the head of the newly produced copy.

Suppose that the computer is looking at a given item on a given list. Then the statement

```
CALL    NCPLTØ (VAR)
```

will cause the computer to break the list at which it is looking into two parts. The first part will consist of all items up to and including the item at which the computer was looking; the second part will consist of all remaining items. The first part of the list will have same name (or names) as it originally had; the second will receive the name VAR. Thus, e.g., if the Fortran variables A(1),...,A(6) are, as previously, the data words NØW, IS, TIME, FØR, ALL, GØØD, the code

```
      CALL    CREATE(LIST1)
      DØ 7    I = 1,6
      CALL    INSERT(A(I))
    7 CALL    PREV(DUMMY)
      CALL    CØPYTØ(LIST2)
      A  =  NEXT(NEXT(NEXT(DUMMY)))
      CALL    NCPLTØ(LIST3)
```

would result in the production of three lists: LIST1, consisting of a head and of the data items GØØD, ALL, FØR; LIST2, consisting of a head and of the data items GØØD, ALL, FØR, TIME, IS, NØW; and LIST3, consisting of a head and of the data items TIME, IS, NØW.

The inverse of NCPLTØ is CUPL, which may be used either in the form

```
      CALL    CUPL(LIST)
```

or

```
      VAR  =  CUPL(LIST)
```

The action of these commands is as follows: Suppose that the computer is looking at a given element on a given list, say, LIST1. Then either of the above commands will cause the whole body of the list named LIST (except its head) to be interpolated into LIST1 immediately following the element at which the computer was looking, and, immediately preceding what was originally the next following item. In this process, the LIST being CUPL-d loses its head and its separate identity; its head is automatically returned to a junkpile of free heads for reuse. The computer is left looking at the first inserted item. The second of the above form of the CUPL command sets the value of the variable VAR equal to the first inserted item in addition to performing the above functions.

Thus, if the Fortran variables A and B are the data words AXE and BOX, the code

```
      CALL   CREATE(LIST1)
      CALL   CREATE(LIST2)
      DØ 7   J = 1,3
      CALL   JUMPTØ(LIST1)
      CUR  = INSERT(A)
      CALL   JUMPTØ(LIST2)
    7 CUR  = INSERT(B)
      CUR  = NEXT(DUMMY)
      CUR  = CUPL(LIST1)
```

would leave LIST2 consisting of a head and of the data items BOX, BOX, AXE, AXE, AXE, BOX; LIST1 would no longer

exist, though it could be recreated as a nullist, if desired, by the subsequent command

```
CALL    CREATE(LIST1) .
```

We also note that in the above coding, the final value of the variable CUR is AXE, and that the computer is left looking at the first item AXE on LIST2.

The function of a link is to designate a list as a sublist of the list on which the link occurs. Each link 'points' in this sense to some sublist. To introduce such a link, the command

```
CALL    HANG(LIST)
```

is used. This command will interpolate a link pointing at the list named LIST immediately after the item at which the computer was looking; the computer is then left looking at the newly inserted link. To remove a link at which the computer is looking, the inverse command

```
CALL    UNHANG(DUMMY)
```

or

```
CUR  =  UNHANG(DUMMY)
```

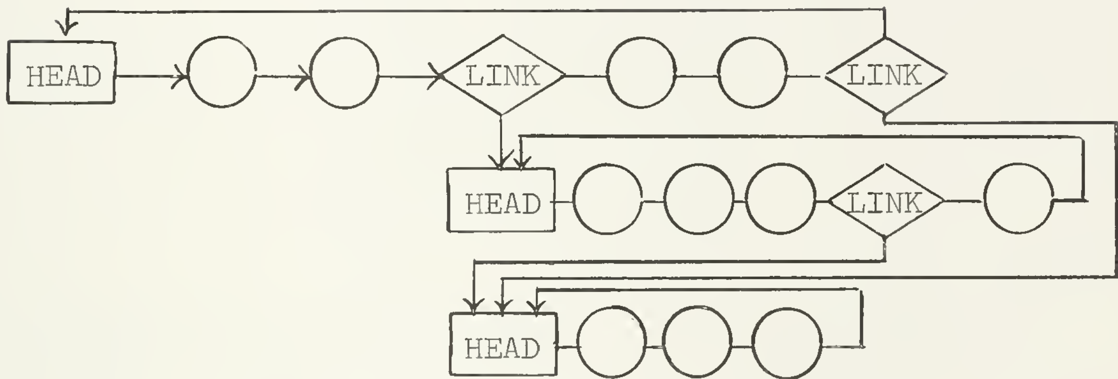
should be used. This will remove the link and leave the computer looking at the item which had followed the link. The second of the above forms of the UNHANG command will also set the variable CUR equal to the value of the next following item in addition to performing the above functions.

If the computer is looking at a link, the command

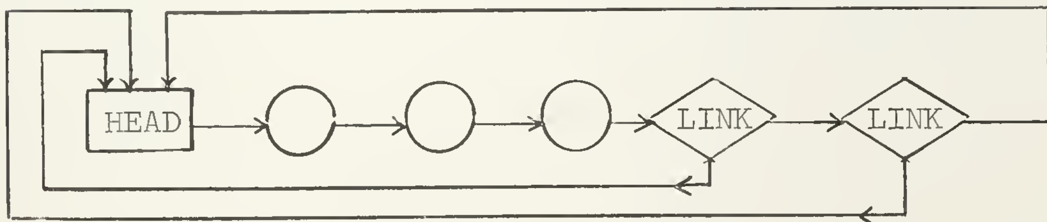
CALL SUBLIS(DUMMY)

will cause the computer to look at the head of the sublist at which the link points. In this way the computer can proceed from a list to one of its sublists.

In terms of the symbols for heads, links, and data introduced above, a typical 'list structure' of lists and sublists might be represented as follows:



It is even possible that a sublist, say LIST2, of a certain list, say LIST1, should itself have LIST1 as a sublist, or even that LIST1 should be a sublist of itself, a situation represented graphically by a diagram such as the following



<u>It is vital to the proper functioning of Nu-Speak that no sublist of a list ever be erased before the list</u>

itself is erased. If the sublist is to be erased, it should be 'unhung' first. Thus, the code

```
CALL   CREATE(LIST1)
CALL   CREATE(LIST2)
CALL   HANG(LIST1)
CALL   ERASER(LIST1)
```

is inadmissible, while

```
CALL   CREATE(LIST1)
CALL   CREATE(LIST2)
CALL   HANG(LIST1)
CALL   UNHANG(LIST1)
CALL   ERASER(LIST1)
```

is admissible.

The requirement that no sublist of a list be erased before the main list is erased naturally makes it difficult to deal with such 'reflexive' structures as the one pictured just above. For an account of the prcoedures to be employed in dealing with such structures, see the section, Special procedures for dealing with self-reflexive list structures, in the attached FAP-Nu-Speak manual.

The command NAMLIS has the form

```
CALL   NAMLIS(VAR)
```

where VAR is a Fortran variable. Its effect is to designate VAR as a name of the list at (an element of) which the computer is looking. The inverse operation to

NAMLIS is provided by the command CLERNM, which has the form

```
CALL    CLERNM(NAME) ,
```

and which frees the variable NAME from being a list name. Both commands CLERNM and NAMLIS leave the item at which the computer is looking unaltered. Thus the code

```
CALL    CREATE(LIST1)
CALL    CREATE(LIST2)
CALL    NAMLIS(TØM)
CALL    NAMLIS(DICK)
CALL    NAMLIS(HARRY)
CALL    JUMPTØ(LIST1)
CALL    NAMLIS(JØE)
CALL    NAMLIS(TØM)
CALL    CLERNM(DICK)
```

will leave the list named LIST1 with the two additional names JØE and TØM, and the LIST named LIST2 with the single additional name HARRY. The variable DICK will not be a list name.

The functions CURRNT and NØWW permit the user to obtain the current data item. In particular, performing arithmetic may have destroyed the MQ register which contained the current data item.(see "conventions involved in looking at a list element" in Sec. B4 of the Nu-Speak FAP Section of this manual). Either

```
X  =  CURRNT (DUMMY)
```

or

```
I  =  NØWW (DUMMY)
```

will restore the MQ and set the left hand side to the current data item. CURRENT and NØWW are identical except that the former should be used for floating, alphabetic or boolean data and the latter for integer data.

Similarly, all the function type subprograms in Nu-Speak have alternate but equivalent names in the opposite mode. Specifically

FIXED	FLØATING
NEXT	FNEXT
IPREV	PREV
NHANG	UNHANG
IREMØV	REMØVE
INSERT	FINSRT
ICUPL	CUPL
ITYPE	TYPE

In assignment or IF statements care must be taken to use the name with the appropriate mode. In CALL statements, of course, either name may be used.

It may now be mentioned that, as Nu-Speak list processing proceeds, a running count of the number of current references to each list is kept. This number is defined as the sum of the number of names of a list and the number of links pointing to the list. If this number reaches 0 (logically implying that there is no way of referring to the list), the list concerned is automatically erased. The following coding, with

comments, will illustrate the points involved.

```
CALL   CREATE(LIST1)        LIST1 HAS 1 REF.
CALL   CREATE(LIST2)        LIST2 HAS 1 REF.
CALL   NAMLIS(TØM)          LIST2 HAS 2 REFS
CALL   NAMLIS(DICK)         LIST2 HAS 3 REFS
CALL   NAMLIS(HARRY)        LIST2 HAS 4 REFS
CALL   CLERNM(DICK)         LIST2 HAS 3 REFS
CALL   JUMPTØ(LIST1)
CALL   HANG(TØM)            LIST2 HAS 4 REFS
CALL   CLERNM(LIST2)        'LIST2' HAS 3 REFS
CALL   CLERNM(TØM)          'LIST2' HAS 2 REFS
CALL   CLERNM(HARRY)        FØRMER 'LIST2' HAS 1 REF,
                            NØ NAME ANY LØNGER
THIS = UNHANG(DUMMY)        FØRMER 'LIST2' LØSES ALL
                            REFS, IS AUTØMATICALLY
                            ERASED.
```

The command DSTRØY has the form

CALL   DSTRØY(LIST).

Its effect is to erase the designated list, all its sublists, all their sublists, etc., thus erasing an entire list structure with a single command. It should never be used if the list structure contains lists which are sublists of more than one list, that is, if the list structure is not 'treelike.'

The instructions

```
CALL   SAVER(A)
```

and

```
CALL   GETTER(A)
```

are used to 'remember' the item at which the computer was looking at a given point in a list manipulation process, in order to be able to return to it later. The notation of the item is made with SAVER, and return may subsequently be made with GETTER. The variable A occurring in SAVER or GETTER should have a Fortran DIMENSIØN of 3.

The <u>user is warned</u> that SAVER and GETTER will work properly only if no lists are created either by CREATE or by CØPYTØ between a given use of SAVER and the corresponding subsequent use of GETTER. Thus, e.g., suppose that the computer is at a given moment examining a given item on a list named LIST. Then the code:

```
CALL   SAVER(A)
CALL   JUMPTØ(LIST)
CALL   CØPYTØ(LIST2)
CALL   GETTER(A)
```

<u>may fail</u> to bring the computer back to the list element originally under examination. To return to a list element even after an ensuing list creation, the following more round about coding can be used.

```
CALL    NCPLTØ(AUXLIS)       BREAK INTØ TWØ PARTS
CALL    JUMPTØ(LIST)
CALL    CØPYTØ(LIST2)        CØPIES FIRST PART
CALL    JUMPTØ(AUXLIS)
CALL    CØPYTØ(AUX2)         CØPIES SECØND PART
CALL    JUMPTØ(LIST2)
CALL    PREV(DUMMY)
CALL    CUPL(AUX2)           JØINS FIRST AND SECOND PARTS
CALL    JUMPTØ(LIST)
CALL    PREV(DUMMY)
CALL    CUPL(AUXLIS)         RESTØRES LIST TØ FIRST
                             CØNDITIØN
CALL    PREV                 GØES BACK TØ ØRIGINAL
                             ELEMENT
```

To enable the use of recursive processes within the Nu-Speak/Fortran system, two auxiliary subroutines, CØMIN and CØMØUT, are provided. These subroutines provide for the recursive bookkeeping of the non-input and non-output variables which are used within the recursive subroutine.

The definition of a Fortran subroutine which is to be used recursively and which uses internal variables should begin:

```
SUBRØUTINE   NAME(List of subroutine arguments)
CALL         CØMIN(LØC, list of internal variables of
                  the subroutine and their dimensions);
```

here LØC is to be a Fortran variable not used otherwise in the subroutine, except in a CALL CØMØUT statement (see

below). The remaining argument list of CØMEIN will have the form VAR1, dimension of VAR1, VAR2, dimension of VAR2, ..., etc. The internal variables of a subroutine are those variables whose values must be preserved around a recursive call to the subroutine.

When a recursive subroutine, say PRØCES, is to call itself, the statement:

```
            CALL      PRØCES (list of arguments)
```

may not be used. Instead, a call of the form:

```
            CALL      DUMMY (list of arguments)
```

should be used where DUMMY is a user coded FAP subroutine (see RECURSIVE SUBROUTINES in the FAP section of this manual) of the following form:

```
1      8      16                                          73
*      FAP
       ENTRY  DUMMMY  (Note: Nu-Speak subroutines should
                            have six letter names.)
       REWIND 16
       UPDATE 16
       REWIND 16                                           NS1ZZZZZ
       SUBR   DUMMMY,INPUTS, (list of input args),
       ETC    ØUTPTS, (list of output args)
       CALL   PRØCES, list of args.
       RETFRM DUMMMY
       END
```

All dummy recursion programs in a job may be included in a single FAP deck; each routine having an ENTRY card (before the first REWIND card) as well as SUBR, USE, RETFRM, and FINI cards.

Finally, any RETURN statement in the FØRTRAN subroutine must be preceded by the statement

```
CALL    CØMØUT(LØC)
```

where LØC is the variable which occurs as the first argument of the CALL CØMEIN statement of the subroutine.

A number of restrictions attach to the use of recursive subroutines in Fortran/Nu-Speak.

1. A recursive CALL statement should not occur within a do-loop. Thus

```
      DØ 7    J = 1,10
      ...
      CALL   PRØCES(ARG1,ARG2)
    7 CØNTINUE
```

is illegal. More generally, one should never CALL from within a do-loop to a routine which might directly or indirectly CALL back to the calling routine.

2. A Fortran/Nu-Speak program involving recursion is required to use the Fortran standard error procedure.

3. Error messages and termination of execution will result from attempts to execute various illegal operations (e.g. REMØVE a listhead, CØPYTØ when not looking at a listhead, etc.). These error messages

should in each case be self-explanatory.

The true power and efficiency of the Nu-Speak recursive subroutine feature can best be realized in the Nu-Speak FAP system. Programs, then, which center on recursiveness should, whenever possible, be coded in FAP.

Fortran/Nu-Speak list operations are executed by the routines described in additional detail in the following FAP/Nu-Speak section of the present manual. A user's Fortran program reaches these underlying programs after transit through a short interface program. Thus the user may consult the FAP/Nu-Speak section of this manual for additional details on the operation of the Nu-Speak system required from time to time. By reading a listing of the interface program it should be possible to clear up even the most subtle of difficulties.

A2. The Chaining Requirement.

The main link of a Fortran/Nu-Speak job must be designated as a chain job with a card * CHAIN(101,3) preceding the job. A two card binary program designated as * CHAIN (102,3) is provided; this calls in a library program containing final error messages as required. This second link program must follow the main link (* CHAIN(101,3)), and generally conform to all the standard Fortran rules for chain jobs.

B. FAP Nu-Speak.

B1. Introduction.

The FAP Nu-Speak System is embedded in FAP. Thus the user writes a program of FAP type, and may write any 7094 machine operation code, any FAP pseudo-operation, and may make use of the FAP macro-operation feature. To the basic FAP package, Nu-Speak adds a number of macros and associated programs. These macros and programs fall naturally into seven groups.

a. The first set of macros belongs to the basic subroutine package, providing for a completely recursive system of subroutine calls, with automatic freezing and unfreezing of stored data on a pushdown list as necessary. A debugging feature and a number of arithmetic macros, which make up for the absence of Fortran-type compiling, are also provided.

The macros of this group are SUBR (creates a recursively usable subroutine), INTARS (provides storage for the internal arguments of a subroutine), RETFRM (return from a subroutine), USE (use a subroutine), FINI (designates the end of a subprogram), DØBEG (begins a do-loop), DØEND (ends a do-loop), RDØBEG (begins a recursive do-loop), RDEND (ends a recursive do-loop), USESV4 (explained below), CALSV4 (explained below), DEBUG (assemble subroutine including debug feature to provide report on entry and leaving), BEGSTK (reserve a storage area for a pushdown stack), PUTIN (put contents of AC into

pushdown stack), TAKFRM (fill accumulator from top of pushdown stack), and CNTSTK (counts the number of entries in a stack). The arithmetic macros are ARITH, ARITHA, FLØAT, and FLØATA. Two additional macros REPT and STEP are provided as debugging aids. The use of most of these features will be made plain to the FAP-writer by the following program to calculate Ackerman's recursive function.

```
          CALL      NURAT,5,FØRMT1,A,1,B,1
          USE       ACKERM,(A,B),C
          CALL      NUWAT,6,FØRMT2,C,1
          CALL      EXIT
A         BSS       1
B         BSS       1
FØRMT1
            [Appropriate format should be included here.]
FØRMT2
          SUBR      ACKERM,INPUTS,(A,B),ØUTPTS,C
          CLA       A
          SUB       =1
          TMI       IFAISO
          STØ       AMINS1
          CLA       B
          SUB       =1
          TMI       IFBISO
          STØ       FMINS1
          USE       ACKERM;(A,BMINS1),VALUE
```

```
         USE       ACKERM,(AMINS1,VALUE),C
         RETFRM    ACKERM
IFAISO   CLA       B
         ADD       =1
         STØ       C
         RETFRM    ACKERM
IFBISO   USE       ACKERM,(AMINS1,=1),C
         RETFRM    ACKERM
         INTARS    (AMINS1,BMINS1,VALUE)
         FINI
```

b. The second set of macros constitutes half of the total list package and enables all operations on lists without sublists, that is, on strings. It includes NEXT (get next element of a list), PREV (get previous element), JUMPTØ (jump to the beginning of a list), CUPL (couple two lists into a single list), NCPLTØ (break a list into two parts), ERASER and ERASØR (for erasing lists), INSERT (inserts a data item on a list), REMØVE (deletes a data item from a list, TEHEAD (tests for a list head), and TEDATA (tests a data element). Other macros in this group are CREATE (creates a list), NAMLIS (designates a variable as the name of a list), CLERNM (frees a list name to name another list), RELEAS (releases a formerly reserved block of storage to freespace), and CØPYTØ (forms a copy of a list). This set of macros makes

use of the programs which maintain a list of free space and provide free space as needed for the other operations. These latter programs are entered by the macros FREESP (provides free space for new list elements), and FREEHD (provides free space for new list heads). A final macro, READY, initializes for list operations.

In advancing, via NEXT or PREV, from one list element to another, the data item of the new list element appears in the MQ register.

The following program section, which interchanges the first two elements of a list named LISTA, and then changes the name of the list to LISTB will illustrate the use of some of the above features.

```
          JUMPTØ    LISTA     JUMP  TØ HEAD ØF LISTA
          NEXT                NØW EXAMINING FIRST ENTRY
          STQ       STØRIJ    SAVE DATA ITEM
          REMØVE              DELETES FIRST DATA LEFT
*                             LØØKING AT SECØND ITEM
          INSERT    STØRIJ    INSERTS DATA ITEM IN
*                             STØRIJ AFTER SECØND ITEM
          CLERNM    LISTA     FREES NAME FØR ØTHER USE
          NAMLIS    LISTB     GIVES NEW NAME TØ PRESENT LIST
          ...
 STØRIJ   BSS       1
```

c. The third set of macros constitutes the remainder of the list package, and enables operations on lists with

sublists. This group of macros includes HANG ("hangs" a list as a sublist of another list), UNHANG (removes a sublist from a list), TELINK (tests a list element to see if it is a pointer to a sublist), SUBLIS (descends to a sublist through such a pointer), and DSTRØY (erases a list and all its sublists). The following program, to find and enter the first sublist of a list, if any such sublists exist, will illustrate the use of some of these macros.

```
        JUMPTØ   LISTA
 LØØP   NEXT              GØ TØ NEXT ELEMENT
        TELINK            HAVE WE FØUND A SUBLIST?
        TRA      SBFIND   IF SØ, TRANSFER ØUT ØF LØØP
        TEHEAD            ARE WE AT END ØF LIST?
        TRA      LØØP     IF NØT, CØNTINUE IN LØØP
 ...    ...
 SBFIND SUBLIS            IF SUBLIST PØINTER FØUND,
*                         DESCEND TØ SUBLIST.
```

d. The fourth set of macros constitutes a miscellaneous set, some of which are not particularly intended for ordinary use by the programmer using Nu-Speak, but which may be so used, if desired. This fourth set includes ERASYR (for deleting a link), PRNTMC (replaces the FAP pseudo-op PMC to assure printing of macro-cards in an assembly), HEADER (provides a heading character), and a few others described in more detail below.

e. The fifth set of macros pertains to the special Nu-Speak input/output system provided as an option, and will be explained later in this manual.

f. The sixth set of macros provides for the manipulation of BCD letters within data words in a manner similar to the manipulation of data words within blocks.

g. The seventh set of macros are intermediate macros, used by the system for the definition of the macros listed above, which the programmer will practically never use, and which he must know about only in order to avoid defining macros with the same name (which would, of course, destroy the whole house of cards). These macros have names of 6 characters starting with two periods.

The Nu-Speak user interested in saving core space will be able to use only part of the system, rather than the whole. The three options provided are:

(a) The small economy size. Only the recursive subroutine, list, and string macros.

(b) The macros of (a) and the arithmetic package.

(c) The whole works.

An additional core storage saving is provided by the WRITE macro of the Nu-Speak I/Ø system (described below) which uses chaining to restrict main-line I/Ø operations to binary operations using (IØB), thereby saving 2,000 words which otherwise would be occupied by various programs of the (IØH) complex. Note however that use of the DEBUG feature of Nu-Speak (which uses NUWAT) will call in

the (IØH) complex, thus requiring extra space.

Again as a space-saving measure, various terminal error messages and programs are relegated to a third chain link.

B2. Allocation of core storage in Nu-Speak.

Core storage in Nu-Speak will normally be allocated as follows:

(a) At bottom, the 100 reserved system locations.

(b) Then the program area, including blocks reserved for linear or other Fortran type arrays.

(c) From the end of the program area to the bottom of common, the listspace and pushdown-list area, divided as follows:

(c1) an area, growing upwards, in which occupied and free listheads are found;

(c2) an area above this, growing upwards, in which occupied and once used but presently unoccupied list cells are found;

(c3) an ever-shrinking area of the still unused core, bounded below by the listspace area and above by the

(c4) pushdown list area associated with the recursive use of subroutines, growing downward from upper core and shrinking back up as recursions proceed;

(d) The common area near the top of core;

(e) The usual desolate area of erasable common at the very top of core.

The boundaries of all these various areas of core are initialized automatically in the case of programs using only the minimal Nu-Speak package, and by the macro READY which must be the first instruction of any program using the Nu-Speak list macros.

B3. Use of the Nu-Speak Macros of the First (Recursion) Group.

Glendower: I can call spirits from the vasty deep.

Hotspur: Why so can I, or so can any man;
But will they come when you do call
for them?

-- Henry IV - Part I.

The form of the SUBR macro is

```
SUBR  NAME,INPUTS,(LIST1),ØUTPTS,(LIST2),(LIST3)
```

where

NAME is the subroutine name

LIST1 is the list of dummy inputs (with names normally but not necessarily less than 6 characters)

LIST2 is the list of dummy outputs (with names normally but not neces. less than 6 char.)

LIST3 is a list of registers used in the subroutine and to be restored on exit. Index register 4 is always saved. Index register N, $N \neq 4$, is specified by the symbol XN. Q specifies that

the MQ is to be saved; I the sense indicators. In addition, L specifies that all the list-involved registers are to be saved, i.e., L is equivalent to X6, X7, I and Q.

If a subroutine has no explicit input or output variables, the corresponding words INPUTS or ØUTPTS should be omitted. LIST3, too, may be omitted should the saving of the machine registers be unnecessary.

If a subroutine uses certain arguments internally and if these arguments do not remain constant during recursive use of the subroutine, a list of these arguments should appear in the INTARS macro

INTARS (List)

within the SUBR. At most one use of the INTARS macro may occur in any SUBR.

Arguments not logically needed in a recursion may be specified by any of the usual methods. (INTARS will work just as well for them, but will waste XEQ time and stack space.)

Storage will automatically be provided for any dummy variable explicitly designated as an input, output or internal argument by the INTARS macro, see below. None of these variables should be otherwise defined (in the ordinary FAP sense), or the assembly will fail owing to multiply defined symbols.

THE FØRTRAN USER IS WARNED that a subroutine entered

by the macro SUBR will bring in only those variables explicitly designated as INPUTS and will return only those variables explicitly designated as ØUTPTS. Thus, the lines

```
USE  PRØCES,(A,B),C
...
SUBR PRØCES,INPUTS,(A,B),ØUTPTS,(C)
CLA  A
ADD  =1
STØ  A
...
```

will change the values of A and B for use during the execution of PROCES, but will restore them to their original values upon executing a subsequent RETFRM. Similarly, the line

```
USE  PRØCES,(A,B),C
```

followed by

```
SUBR   PRØCES,INTPUTS,(A,B),ØUTPTS,(C)
CLA    C
...
```

will leave PRØCES operating on whatever value of C was left stored upon the last RETFRM PRØCES.

Subroutine names can be inputs to, and outputs from, other subroutines, just as in FORTRAN (and in this sense operate as transfer vectors). The following code will set C equal to the value obtained by applying a function called PRØCES 30 times iteratively to C.

```
          USE       ITERAT,(PRØCES,=30,C),C
          ...
          SUBR      ITERAT,INPUTS,(FCN,NØ,VAR),ØUTPTS,(VAL)
BEG1      DØBEG     END1,1,=1,NØ
          USE       FCN,VAR,VAL
END1      DØEND     BEG1,1
          FINI
          ...
```

the above lines also illustrate the use of the do-loop pair DØBEG and DØEND, which is described more carefully below.

The following code will calculate F(2), F being that one of the functions FUNC1 and FUNC2 for which F(1) assumes the larger value.

```
USE       SELECT,(FUNC1,FUNC2),WINNER
USE       WINNER,=2,RESULT
...
SUBR      SELECT,INPUTS,(PRØC1,PRØC2),ØUTPTS,BETTR
USE       PRØC1,=1,VAL1
USE       PRØC2,=1,VAL2
INTARS    (VAL1,VAL2)
CLA       VAL1
SUB       VAL2
TMI       2BETR
CLA       PRØC1
```

```
         STØ       BETTR
         RETFRM    SELECT
2BETR    CLA       PRØC2
         STØ       BETTR
         RETFRM    SELECT
         FINI
```

ETC cards can be used to extend the first line of SUBR.

THE FØRTRAN USER IS WARNED that, as distinct from the FØRTRAN word SUBRØUTINE, the appearance in a Nu-Speak program of the word SUBR will not begin a new program (in the sense of the FAP assembler). A Nu-Speak subprogram is terminated only by an END card, and begun only by a *FAP card. Thus all the FAP symbolic code belonging to the various SUBR's of a section of Nu-Speak code belongs to a single FAP program. To provide proper isolation from each other of variables belonging to different SUBR's, but having the same symbolic names, Nu-Speak uses a heading scheme, based on the HEAD pseudo-operation in FAP. This mechanism works as follows: each appearance of SUBR assembles the intermediate macro HEADER (see account below) which provides a heading character (the same) for every symbolic name in the body of code covered by the SUBR. Thirty-five heading characters are issued in the order Z,Y,X,...,A,9,8,...,0

to successive SUBR's. If there are more than 35 SUBR's in a program, the HEADER mechanism will begin again from Z. The line FINI, which should normally be the last line of each SUBR, assembles simply as HEAD 0, FAP symbols with head 0 being the same as unheaded symbols.

Since FAP symbols are at most 6 characters long, 6 letter symbols will not be headed, and will therefore be accessible to all SUBR's. The use of the same 6-character symbol as an explicit INPUT, ØUTPUT, or INTAR of two different SUBR's will therefore lead to multiply defined symbols. On the other hand, by using a six character symbol as a variable in a single SUBR, one allows other SUBR's to have access to it, and this possibility provides what is in effect a "named common" which can replace the "numbered" CØMMØN of the FØRTRAN-FAP system. However, if one SUBR accesses <u>and changes</u> an argument of another SUBR in this way, the recursiveness of the system may be spoiled. (For a detailed understanding of this point, see appendix below, or listing of the macros SUBR and SUBBDY.) Thus, the heading system makes it advisable, as a matter of ordinary practice, to name subroutines with six letter symbols (so that they may be USE-d within other SUBR's), and to name variables and other locations with symbols of five characters or less.

Programs written with more than 35 SUBR's may inadvertently produce multiply defined symbols, owing to

the re-use of heads. In the (hopefully few) cases in which this occurs, it can be cured by appropriately renaming the offending variables and locations.

The macro RETFRM assembles as a single line of machine code, e.g.

```
RETFRM   PRØCES
```

assembles as

```
TRA      PRØCES+5
```

It is used to return from a subroutine to the calling routine, and is analogous in function to the FØRTRAN word RETURN.

The macro USE assembles the same lines of code, and is used in much the same way, as the FAP pseudo-operation CALL, but with the exception that it makes no transfer-vector entry (naturally not!). Its form is

```
USE      NAME,(LIST1),(LIST2)
```

where

- NAME is the name of the subroutine to be used
- LIST1 is a list of the inputs provided to it or the word NØINPS
- LIST2 is a list of the outputs which it is to provide or the word NØØUPS.

The variables of these two lists must coincide in number and correspond in order with the INPUTS and ØUTPUTS declared in the first line of the SUBR named NAME.

If a Nu-Speak SUBR located in a given program is to be used in an external program, it should be called in the more conventional form

```
        CALL      NAME,LIST1,LIST2
```

where LIST1 is a list of input arguments, and LIST2 is a list of output arguments. In addition, the first card group of the program containing the SUBR in question must contain a card of the form

```
        ENTRY     NAME
```

for each SUBR which is to be used by an outside program.

Since the assembly of the macros CALL and USE generate a line of the form TSX SUBNAM,4 they involve an implicit use of index register XR4, and hence will change its contents. In order that the contents of XR4 may be restored on return from a SUBR if desired, secondary forms CALSV4 and USESV4 of these macros have been provided ("call saving XR4" and "use saving XR4"). These act in just the same way as CALL and USE, except that XR4 is stored before the CALL or USE and restored immediately after return from the SUBR. Normal Nu-Speak practice, however, will omit those saves and restorations, and consider the value of XR4 within any given SUBR to be indefinite. The proper "existing" value of XR4 will in all cases be saved on entry to a SUBR, and XR4 will be restored to its initial

value just prior to a return from the SUBR. (Cf. the discussion above of XR-save specifications in a SUBR head.)

The DEBUG feature of Nu-Speak is controlled by a card of the form

DEBUG blank, or ØN, or ØFF.

The appearance of a DEBUG ØN card will cause the assembly of all following SUBR's, up to the next DEBUG ØFF or DEBUG card, to be modified. As a consequence of this modification, each entry into the SUBR will produce a message of the form SUBRØUTINE NAMED (name of subroutine) HAS JUST BEEN ENTERED, together with a statement (in octal) of the condition of the accumulator, P+Q bits, MQ register, sense indicators, and XR4 (complemented), and input arguments while each subsequent return from the SUBR will produce a message of the form SUBRØUTINE NAMED (name of subroutine) IS BEING LEFT, together with the condition of the same principal registers as above (excepting XR4, however), and a statement (in octal) of the values of all output arguments. The DEBUG ØFF card terminates the debug feature; alternate appearances of the card DEBUG switch this feature off and on. While using the DEBUG feature, the Nu-Speak binary deck marked DEBUGPRØG must be included with the deck being loaded (see below, form of decks for Nu-Speak jobs). _This_

program calls in the 4,000-odd words of the (IØH) complex, and so may reduce the amount of core space available to the new program complex by a considerable amount.

When the debugging of a Nu-Speak program is complete, all the DEBUG cards may be removed from the source symbolic deck; the binary DEBUG PRØG deck is to be removed also. The source deck will then assemble into final form.

The macros ARITH and ARITHA provide a rudimentary set of floating point and double-precision floating point arithmetic macros. Their form is

ARITH   variable, word, (list), equals, variable 2

and

ARITHA word, (list), equals, variable 2.

Here

(a) "word" denotes one of the four single-precision control words PLUS, MINUS, TIMES, or ØVER, with obvious arithmetic operational significance, or one of the four double-precision control words DPLUS, DMINUS, DTIMES, or DØVER, of corresponding significance.

(b) "equals" denotes the control word EQUALS, which may however be absent, in which case the word "variable 2" may as well be absent also (cf. below).

(c) variable 2 is the symbolic name of a variable which is to be set equal to the result of the preceding arithmetic operations. (If this variable, together with the word EQUALS, are omitted, the result will be left in

the accumulator (or accumulator and MQ, in case of double-precision variables.)

(d) "list" is a list of the variables to be successively combined using the operation specified by "word."

(e) "variable" in the ARITH macro is the variable to be brought into the central processing unit at the start of the desired arithmetic operations; the macro ARITHA finds its first variable already in the accumulator.

The use of ARITH and ARITHA are illustrated in the following program, which has the same effect as the FØRTRAN statement A = (SQRTF(X*Y*Z*W-1.0))/X1.

```
ARITH     X, TIMES, (Y,Z,W)
ARITHA    MINUS, = 1.0
CALL      SQRT
ARITHA    ØVER, X1, EQUALS, A
...
```

The form of the macro FLØAT is

```
FLØAT     variable, equals, variable 2;
```

the form of the macro FLØATA is

```
FLØATA    equals, variable 2.
```

Here "equals," and "variable 2" have the same significance as above in the macros ARITH and ARITHA, and may be omitted with the same consequences as above. "Variable" is the symbolic name of the <u>full word</u>

(FØRTRAN USERS WARNED) integer to be converted to floating point. The macro FLØATA finds its variable in the accumulator.

The do-loop pair DØBEG (do-loop beginning) and DØEND (do-loop ending) have the following form:

```
location 1     DØBEG   location 2, numb, A, B
location 2     DØEND   location 1, numb
```

Here "location 1" and "location 2" must be valid FAP symbols designating the location of the beginning and end of the do-loop, "numb" is an integer from 1 to 7 designating the index register in which the count for the do-loop is to be kept, and A and B are locations ("variables") whose ADDRESS PØRTIØNS (FØRTRAN USERS WARNED) contain respectively the lower and upper count limits for the do-loop.

The alternate do-loop pair RDØBEG (recursive do-loop beginning) and RØEND (recursive do-loop ending) have exactly the same form and much the same use, but are more suitable in certain circumstances for use in recursive subroutines. The distinction is most easily perceived by examining the expansion of these macros. DØBEG expands as

```
          MACRØ
START     DØBEG    END, XR, K, L
          LXA      L,XR
          SXD      START,XR
          LXA      K,XR
START     TXH      END+1,XR,**
DØBEG     END                    ;
```

while the expansion of the DØEND is

```
DØEND     MACRØ    START,XR
          TXI      START,XR,1
DØEND     END
```

If a recursive subroutine calls itself (even indirectly) from within such a do-loop, the decrement in the instruction START will be changed, and the do-loop may fail to function correctly on return from the subroutine. To protect against such failure, the code

```
          CLA      START
          STØ      VAR
```

can be executed immediately before the recursive subroutine call, and the code

```
          CLA      VAR
          STØ      START
```

executed immediately after returning from the subroutine called recursively; here VAR should be one of the INTAR's of the calling subroutine.

To avoid this nuisance, the alternate do-loop pair may be employed. RDØBEG expands as

```
RDØBEG    MACRØ   END,XR,A,B
          LXA     A,XR
          SCD     *+5,XR
          SXD     *+3,XR
          LXA     B,XR
          TXI     *+1,XR,1
          TXL     END+1,XR,**
          TXI     *+1,XR,**
RDØBEG    END
```

and RDØEND expands as

```
RDØEND    MACRØ   START,XR
          TIX     START+7,XR,1
RDØEND    END
```

Since the cut off control for a do-loop of this type is carried in an index register which will be saved and restored during the recursion if its use is noted in the SUBR header, such a do-loop will function properly even in the above-described recursive situation.

To illustrate the use of these macros, we give the following lines of code, which will store 1,000 times the variable B in the variable C, and incidentally, leave index register 1 set at 1,001):

```
                CLA       =1000
                STØ       CØUNT
                FLØAT     =0, EQUALS, B
      HERE      DØBEG     THERE,1,=1,CØUNT
                ARITH     B,PLUS,C,EQUALS,C
      THERE     DØEND     HERE,1
                ...
```

Much the same code could be written more simply as

```
                FLØAT     =0,EQUALS,B
      WHENS     DØBEG     WITHR,1,=1,=1000
                ARITH     B,PLUS,C,EQUALS,C
      WITHR     DØEND     WHENS,1
```

If the upper limit of such a do-loop lies below its lower limit, it will not be executed at all.

Pairs of DØBEG's and DØEND's using different index registers may be nested within each other in obvious fashion.

The user is warned that the location symbol written in the location field of a DØBEG macro-operation <u>does not describe</u> the location of the first line of generated machine code. For a detailed understanding of the point, a listing of the DØBEG macro should be consulted.

The programmer wishing to obtain an automatic dump when freespace becomes exhausted in a program using only

the minimal Nu-Speak package may include the control card ALDUMP as the first instruction of his program. (WARNING: a dump produced in these circumstances will normally be approximately 200 pages long.) The four macros BEGSTK, PUTIN, TAKFRM, and CNTSTK enable the user to automatically construct and use any number of "LAST IN/FIRST ØUT" type push down stacks.

The macro BEGSTK has the form

```
BEGSTK        size,wd,loc
```

where size is a decimal integer equal to the number of core locations to be reserved for the operation of the stack.

This defines the size of the stack, and provides either for the production of an error message or for a specific transfer of control on stack overflow or underflow.

A stack of nominal size N will actually occupy N+4 core locations.

If the macro is used without the last 2 words, wd and size, as e.g. in

```
STACK1   BEGSTK   500
```

an error message and termination of execution will result on stack overflow or underflow. If the macrovariable is replaced by the control word RETURN, as in

```
STACK1   BEGSTK   500,RETURN,LØCL
```

return on overflow or underflow will be made to the program location LØCL.

If the programmer desires to return control after

the stack has been filled or emptied to some point designated by himself (by using the long form of the BEGSTK macro instruction), he will probably want one set of instructions to be executed after the stack has been filled, and another after it has been emptied.

In the case of stack underflow or overflow control will always be transferred to the location specified by the user in his BEGSTK statement. (If no return location is specified, stack underflow or overflow will produce an error message and program termination.) If, when the CPU arrives at the specified location, XR1 contains 0, a stack underflow condition exists. If XR1 contains -SIZE-2, a stack overflow exists. It is suggested that the instruction at location LØCL be:

```
LØCL      TXL      EMPTY,1,0
                      :
EMPTY
```

Since -SIZE-2 is stored in XR1 as a positive number (the 2's complement) control will then pass to one set of statements after the stack has been emptied (XR1 = 0) and to another set of statements after the stack has been filled (XR1 = -SIZE-2).

Note that XR1 may be used within the program since it is saved and restored by the stack macros.

BEGSTK must be placed in a portion of the program to which control can never pass (e.g. after a CALL EXIT state-

ment). Control will then only pass to it via the PUTIN or TAKFRM macros. Also there must be a symbol in the location field of the macro-instruction. This is the name of the stack and is defined as the location of the first machine instruction in the macro. It <u>must not be omitted</u>.

The form of the macro PUTIN is

```
                PUTIN           A
```

where A is the name of a stack created by the macro BEGSTK. It stored the contents of the accumulator in successive locations of the stack A. The inverse operation is performed by the macro TAKFRM which has the corresponding form.

```
                TAKFRM          A
                PUTIN           MYSTAK
                        .
                        .
                        .
                CALL            EXIT
      MYSTAK    BEGSTK          500
      A         BEGSTK          28
```

The first element PUTIN a stack is always the last element TAKen FRM the stack.

Both PUTIN and TAKFRM work must more rapidly (12 cycles) than the corresponding list macros INSERT and REMØVE, and should be used in preference to the list

macros where the additional flexibility and logical connections of the list macros are unnecessary.

The final macro of the Nu-Speak pushdown-stack set is CNTSTK, which has the form

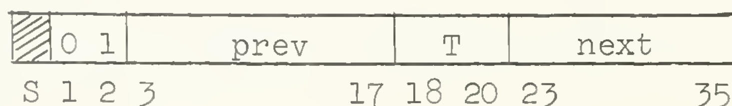
```
CNTSTK          NAM,SIZE,RMNING.
```

Here NAM is the name of a stack, SIZE is its size, and RMNING is a core location. Use of this macro will store the number of locations remaining in the stack in the address of the location RMNING.

B4. The Second (String-Manipulation) Set of Nu-Speak Macros.

To understand the use of the Nu-Speak list-processes, it is necessary to understand something of the Nu-Speak representation of lists within the 7094. Lists are constructed out of three types of elements: data elements, listheads, and links.

(a) Data elements. Data elements are stored in core in blocks, always containing an even number from 2-16 of successive core words, and always beginning at an even core location. The first word of a block is its identifier; the remaining words represent successive data items on a list. The structure of the identifier is as follows



where

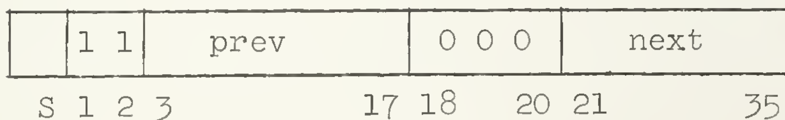
(a1) "next" denotes the address of the identifier of the next block in the list; the NEXT of the last identifier in a string points back to the head of the string (see below).

(a2) "prev" denotes the address of the identifier of the previous block in the list;

(a3) T is an octal integer between 0 and 7 representing the number of words in the block according to the convention no + 2(T+1);

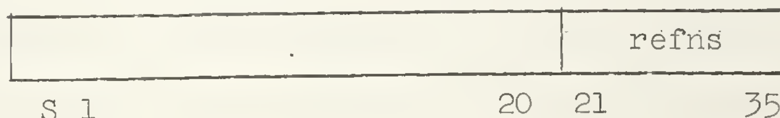
(a4) The prefix 01 (with indeterminate sign bit) indicates that the block which follows is data (rather than a head or a link).

(b) Head elements. A head element is stored in a block of two words consisting of an even location and the next successive odd location. The first word of this block is the head identifier, and has the form



Here, prev and next have the same significance as above, while the prefix 11 (with indeterminate sign bit) indicates that the blank which follows is a head. The prev of a head points to the last identifier in the string.

The second of the two words constituting a head has the form



here refns is the number of current references to the list (see below), while bits S-20 are unaffected by any of the built in Nu-Speak list macros (see, however, the account below of special procedures affecting self-referent lists) and are available for use by the programmer.

A list contains one and only one head.

(c) Link elements. Link elements are pointers to sublists; since strings are distinguished from more general lists by having no sublists, we postpone a discussion of the structure and function of these elements to the next section. Here, however, we shall note that a link, like a head, occupies a pair of locations, the first being even; and that the first word of the pair is the link identifier.

The computer is said to be "looking at" an element of a list (data item, head, or, for that matter, link) if

i. A copy of the identifier of the block in which the element lies is in the sense indicator register SI;

ii. The address of the identifier of the head of the list containing the element is contained in complemented form in the index register X6;

iii. An integer from 1 to 15, and denoting the position of the data word within its block is contained in the index register X7;

iv. A copy of the data word concerned (or, in the case of heads (or links, cf. next section), the odd-location word following the identifier) is contained in the MQ register.

With this understanding, we may proceed to explain the effect of the Nu-Speak list macros of the string-manipulation group.

The macros NEXT and PREV have the form

NEXT

and

PREV

respectively. If the computer is looking at an element of a list (whether head, data, or link) these macros cause it to look at the next and the previous elements respectively.

The macro CUPL has the form

CUPL listname,

where listname is the symbolic name of a certain list. If the computer is looking at an element on a list when this macro is encountered, it will couple the list named "listname" into the list containing the element being examined; that is, transform the list being examined by insertion of all data items and links of the list named "listname," the insertion beginning immediately after the data item originally being examined. The inserted items then precede the item originally following the item at which the computer was looking. The computer is left looking at the same element.

The list "listname" <u>being</u> coupled loses its separate identity in this process, and its head is erased and returned

to a pile of spare head locations. Thus, for example, if two lists, named LIST1 and LIST2 are contained in core, if LIST1 contains, besides its head, the 3 data ALPHA, BETA, and GAMMA, and if LIST2 contains, besides its head, the 3 data HEE, HAW, HØØ, then, if the computer encounters the macro

```
CUPL    LIST2
```

while looking at the datum ALPHA on LIST1, the result will be that LIST1 will come to have the six data ALPHA, HEE, HAW, HØØ, BETA, GAMMA, and LIST2 will have ceased to exist. The computer will be left looking at ALPHA. On the other hand, the lines

```
PREV
CUPL    LIST2
```

will produce a LIST1 containing the six data HEE, HAW, HØØ, ALPHA, BETA, GAMMA; LIST2 will again have ceased to exist. The computer will be left looking at the head of LIST1.

The macros INSERT and REMØVE have the form

```
INSERT      variable,
```

and

```
REMØVE
```

where "variable" is the symbolic name of a variable. If the computer is looking at a data item or link, REMØVE will cause this item to be properly deleted from the list,

and will leave the computer looking at the next element of the list. An attempt to delete a head will cause an error return.

If the computer is looking at a list item, "INSERT variable" will cause the contents of location named variable to be inserted in the list immediately following the said item, and will leave the computer looking at the inserted item.

The macros NAMLIS and CLERNM have the form

NAMLIS listname,

and

CLERNM listname

respectively, where "listname" is the symbolic name of a variable which, in the first case, is to become the name of a list, and, in the second, is to cease being the name of a list.

It should be said here that the name of a list is, in machine terms, a storage location whose address part contains the address of the identifier word of the head of the list.

The significance of the last two macros, and especially of CLERNM, may be further elucidated, as follows. As Nu-Speak list processing proceeds, a running account of the number of references to a list is automatically kept in 21-35 of the second word of the listhead. (If this number exceeds 32,767, <u>trouble</u>

will ensue). This number of references is determined as the total of all the names of the list, (plus all the links pointing to the listhead, that is the number of times the list occurs as a sublist, as explained in Section 5 below). Whenever the number of references to a list is reduced to zero, the list will automatically be erased. Care should therefore be taken not to disturb the count of references to a list inadvertently.

The macro "NAMLIS listname" increases the reference-count of the list, an element of which is currently being looked at, by 1, and delivers the address of its head to the location "listname." In this way, "listname" becomes a name of the said list.

The macro "CLERNM listname" reduces the number of references to the list named "listname" by 1. It leaves the MQ and SI registers, and the index registers of the central processor unaltered.

The macro JUMPTØ has the form

JUMPTØ   listname

where "listname" is the name of a list; and acts as follows: if the central processor is looking at an element on a certain list when it encounters the instruction JUMPTØ listname, then the central processor will be left looking at the head of the list named "listname".

The two test operations TEHEAD and TEDATA cause "skips" like the standard FAP "test-type" orders, and have

the following effect: if the central processor is looking at a list element, then TEDATA will cause the next machine instruction (WARNING: NØT MACRØ INSTRUCTIØN) to be skipped if this list element is data; if the list element being looked at is not data, the control processor will execute the next instruction. TEHEAD operates similarly, causing the central processor to skip the next machine instruction if the central processor is looking at a head, but to execute the next machine instruction if the central processor is not looking at a head. These instructions should be used only when it is known that the central processor is looking at some element on some list, as they may otherwise result in unaccountable skips.

The Nu-Speak macro CREATE has the form

CREATE listname

where "listname" is the name of a variable. This macro creates a list with no entries (and hence consisting exclusively of a head which is its own NEXT and PREV), and stores the address of the identifier of this head in the variable "listname," so that the variable becomes the name of the nullist. The central processor is left looking at the head of the newly created list. This macro can be used even at a point in a program where the central processor is not looking at any list element.

If the central processor is examining an item on a list, the macro NCPLTØ, which has the form

NCPLTØ variable

will break the list immediately after the item being examined removing from the list all items from this point on and up to but not including its head (which is both first and last item on every list). A new list will be formed, consisting of the items deleted, and having the name "variable." The central processor is left looking at the head of the newly formed list. Thus, e.g., if the central processor encounters the instruction

NCPLTØ LIST1

while looking at the item ALPHA on a list named LIST2 whose entries are ALPHA, BETA, GAMMA, LIST2 will be reduced to the single entry ALPHA, while LIST1 will have the entries BETA, GAMMA. In the same situation, the lines

```
PREV
NCPLTØ      LIST1
```

will leave LIST2 as a list without any entries (consisting only of a head) and LIST1 will have the entries ALPHA, BETA, GAMMA. Again in the same situation, on using the lines,

```
PREV
PREV
NCPLTØ      LIST1
```

the list LIST2 will retain the entries ALPHA, BETA, and GAMMA, and LIST1 will have no entries.

The macro CØPYTØ has the form

CØPYTØ listname

where "listname" is a FAP variable, that is a FAP location. If the central processor is not looking at a listhead when it encounters this instruction, an error message will result, and execution will be terminated. On the other hand, if "CØPYTØ listname" is encountered when the central processor is looking at the head of a list, a copy of the list, with all data items and links included will be produced, and the newly produced list will be named "listname." The central processor will be left looking at the head of the original list, with the second word of the original listhead in its MQ. (The arrangement of data words within blocks of core is not necessarily the same in the copy as in the original list, however.)

The pair of macros ERASER and ERASØR have the form

ERASER

and

ERASØR

respectively. Both of these instructions may be used when the address of a listhead identifier is contained in the address portion of the accumulator; either will then proceed to erase the list in question by attaching to it the bottom of a special "junkpile list" of

elements available for re-use. (See Section 5A for a more detailed account of the internal procedures involved.) ERASER tests the location concerned to verify that it contains a listhead and provides an error return if an error is detected; ERASØR omits this test. ERASER should therefore normally be preferred.

The macros RELEAS has the form

```
        RELEAS        A,B
```

the macro will form a headless string of B words beginning with location A and extending upward through consecutive core locations. This headless string will then be attached to the top of the freespace junkpile (see below). Thus, B core locations, beginning with location A, are in effect "released" for future reassignment by the FREESP macro. Since a data identifier will be inserted in A, A must be an even location. If odd, A will be increased by 1 and B will be decreased by 1. Care must be taken not to RELEAS the same block of core twice unless that block has completely been reassigned by FREESP.

The macro-instruction READY should be the first instruction in any Nu-Speak program using any of the (other) list macros. It initializes various counters and pointers in the basic Nu-Speak programs, thus preparing for the list processing operations which are to follow. This macro also provides the programmer with control over the Nu-Speak automatic dump procedures, which are as follows:

(a) If any one of a number of detectable errors (e.g., attempting to delete the head of a list, beginning to copy a list at a position other than its head, etc.) occurs in a Nu-Speak program, execution will be terminated, an appropriate error message will be produced, and a core dump in octal with mnemonics will be given automatically before EXIT is called.

(b) If the generation of lists or the growth of the pushdown list exhausts all available freespace, execution will be terminated, and message to this effect will be produced, but no dump will be given.

To suppress the dump that would otherwise be produced in case (a), the READY card should have the form

```
READY       NØDUMP
```

To give a dump in case (b), the ready card should have the form

```
READY       ALDUMP
```

(WARNING: a dump produced when freespace is exhausted will normally be 200 PAGES LONG.)

The program (FPT), which provides for automatic recovery from situations of floating point overflow or underflow, is automatically called in by an Nu-Speak program.

The auxiliary list macro TESTBL (test and break block if necessary) has the form

TESTBL

If this command is received while the processor is looking at a list element, it will check to see if the element is a data element embedded in a block of data elements of length longer than 1. If this is the case, the computer will proceed to break the block (normally into three segments) in such a way that the data element concerned becomes the end of the subblock containing it. After the block is thus broken, insertions, etc., can be made in normal fashion. The computer is left by TESTBL looking at the same list element as previously, (and with the data word of this element brought back into the MQ).

The following precautions are to be observed when using list processes and the SUBR and related macros together.

a. All the list processes make use of index registers XR6 and XR7, as well as the sense indicator

register SI (and naturally the accumulator and MQ). Thus a list-processing SUBR will effect these registers. The index register XR6 and XR7 will be restored to their entering condition if the symbols X6 and X7 or the letter L is included among the declared XR's in the normal SUBR heading. Similarly the sense indicators and MQ register will be saved if Q or L is specified in this heading.

All of the list operations <u>of the present group</u> will function properly even when the contents of the MQ (i.e., data word, or second word of a list head) have been destroyed by processing or testing. Thus

```
        NEXT
```

and

```
        LDQ     =0
        NEXT
```

have precisely the same result, etc.

All the Nu-Speak list-macros which have variable location addresses as sole arguments obtain these arguments from the first line of the macro-generated machine code. Thus, to obtain the effect of

```
        JUMPTØ      listname
```

but with a more easily variable "listname," lines like

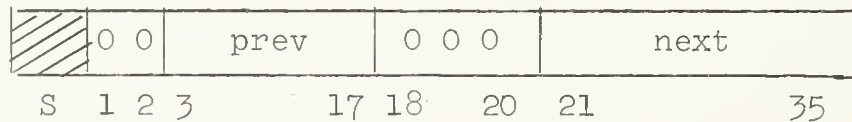
```
        CLA     listname
        STA     *+1
        JUMPTØ  **
```

are permissible.

The arguments, however, may not be tagged, nor may tags be stored into the macros in a manner similar to the above.

B5. The Nu-Speak Macros of the Third (Sublist Manipulating) Group.

Sublists are introduced into Nu-Speak lists by inserting into a main list "links" which point to a sublist. A link always occupies two successive words in core, the first in an even location. The first word of a link is its identifier, and has the following structure



where "prev" and "next" have the same significance as above (cf. the first paragraph of Section 4 above).

The second word of a link has the form

where "pointer" is the address of the identifier of the head of some other list. Bits S-20 are unaffected by any of the Nu-Speak processes, and are available to the programmer (as, e.g., for storing various bits descriptive of the sublist).

All the above-described macros of the string manipulation group, with the obvious exception of TEDATA, treat

links in exactly the same way as a data item of a list.

The macro TELINK operates as follows. If the central processor is looking at a list element when it encounters TELINK, it will skip the next <u>machine</u> instruction if the data item is not a link. The macro TELINK should not be used if the central processor is not looking at a list entry, as it may lead to unaccountable skips.

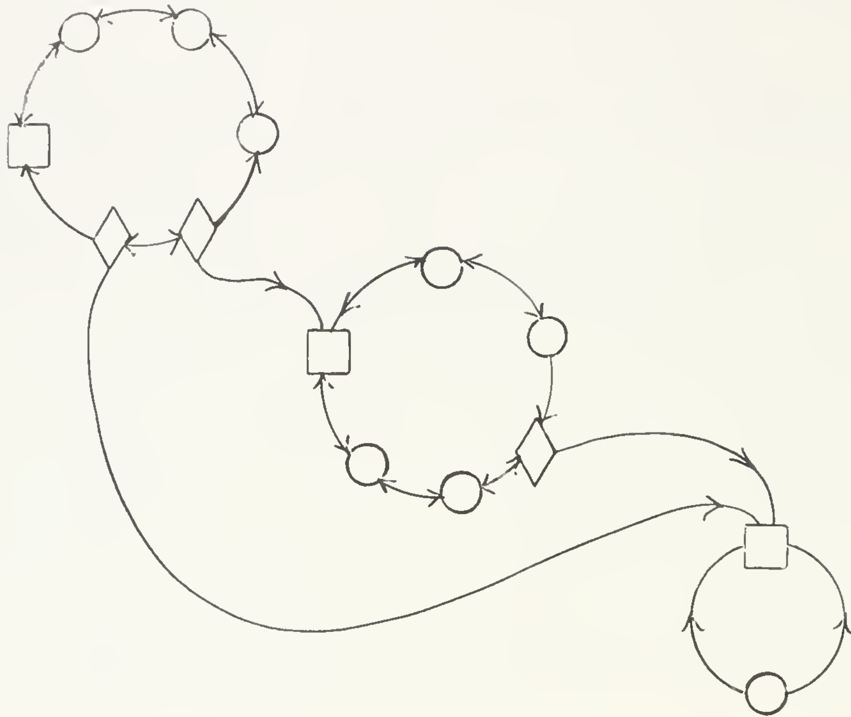
The macro HANG has the form

HANG listname

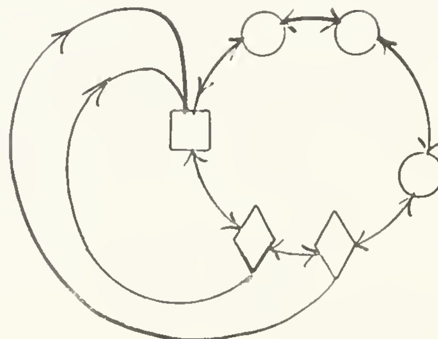
where "listname" is the name of a list. Its effect is as follows. If the central processor is looking at an item on a list when it encounters HANG listname, it will insert a link immediately after the item at which it is looking. The second word of this link will contain 21 leading zeros, followed by the address of the identifier of the list named "listname;" thus the link will "point" at the list named "listname." In this way, the list named "listname" becomes a "sublist" of the original list. The number of references to "listname" as enumerated in the head of the list named "listname," is augmented by 1. The central processor is left looking at the newly inserted link.

Thus, if we use circles to denote data items, squares to denote listheads, and rhomboids to denote links, the structure of a typical system of Nu-Speak lists might be graphically represented by a structure something like the

following:



It is even possible that a sublist, say LIST2, of a certain list, say LIST1, should itself have LIST1 as a sublist, or even that LIST1 should be a sublist of itself, a situation represented graphically by the following diagram



If a list structure is such that its lists may be arranged in an ordered hierarchy, in such a way that no list has either itself or any list higher in the hierarchy as a sublist, the structure is said to be non-reflexive; otherwise, it is said to be reflexive. The proper treatment of reflexive list structures, especially insofar as erasing and procedures which lead to the erasure of lists are concerned, is logically complicated and must involve the use of special subroutines. The programmer anticipating the development of such list structures should consult the special section below concerned with reflexive list structures.

The inverse operation to HANG is provided by the macro UNHANG, which has the form

UNHANG

If this instruction is encountered while the central processor is looking at a list item which is not a link, an error return will be produced, and execution terminated. If the instruction is encountered while the central processor is looking at a link, then

i) the link will be deleted;

ii) the number of references to the sublist to which the deleted link pointed will be diminished by 1;

iii) if no references to the aforesaid sublist remain, the sublist will AUTOMATICALLY BE ERASED;

iv) the central processor will be left looking at the next item on the list which had previously contained the said link.

Access from lists to sublists is provided by the macro SUBLIS, which has the form

SUBLIS

If this instruction is encountered while the central processor is looking at a list item which is not a link, an error message will be produced, and execution terminated. If the macro-instruction is encountered while the central processor is looking at a link, the instruction SUBLIS will leave the central processor looking at the head of the sublist to which the link points.

B6. Principles of Operation of the Automatic Erasing Mechanisms.

When a list is erased it is placed at the bottom of a list of released space ("junkpile list").

Whenever one of the Nu-Speak list macros (e.g. INSERT, CØPYTØ) requires a pair of free spaces for the formation of a new list element, an element is supplied, either from the topmost list on the junkpile, or from hitherto unused regions of core. Conversely, whenever space is released (e.g. by ERASER, RELEAS or REMØVE) it is automatically added to the junkpile. Heads are treated similarly, but have their own junkpile, cf. below, however.

As the progressive release of freespace proceeds, links pointing to sublists may be encountered in erased lists. In each such case, the number of references to

the indicated sublist is reduced by 1. If the number of references thereby falls to zero, the sublist is itself erased, and placed on the bottom of the junkpile list. When the freespace mechanism encounters the head of a list on the ordinary junkpile, it transfers it to the head junkpile.

The macro DSTRØY has the form

DSTRØY

Its effect is as follows. If the address portion of the accumulator contains the address of a listhead identifier, the prefix of the identifier will be changed from the normal [▨| 1 1] to the special [▨| 1 0], and the list erased by attaching it to the bottom of the junk-pile in the ordinary way. The marking of a list in this way has the consequence that when the mechanism for progressive release of freespace subsequently reaches a link in the marked list, pointing to one of its sublists, this sublist will automatically be erased, irrespective of the number of references to it. However, it also will inherit the prefixed "mark of Cain" [▨| 1 0], so that sub-sublists, etc., will be treated in the same way. Thus, by using DSTRØY, an entire list structure can be erased with a single command.

Note however that, as the above description of the operation of the underlying Nu-Speak mechanisms should make plain, it is always fatal to erase a sublist of a

list before the main list is erased. Indeed, in this case, an irrelevant link is left in the main list, and if, e.g., the main list is subsequently erased, the consequence will be that when this link is reached by the freespace providing mechanism, a change, quite unanticipated by the programmer, will be made at the location to which this link points, and at various related locations. One such error can easily suffice to disrupt everything.

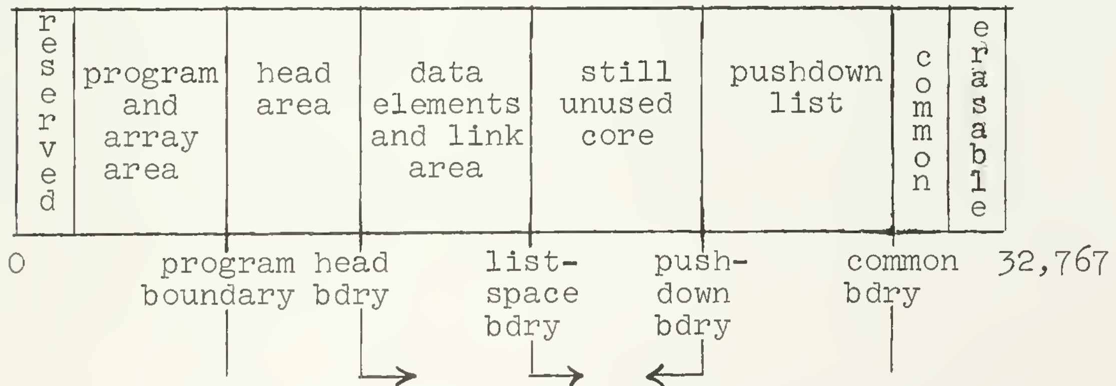
Thus the macro DSTRØY should be used cautiously, and only to erase the lists of a list structure which is known to have the property that each of its sub-lists, sub-sublists, etc., hangs from only one link, so that each of these sublists, sub-sublists, etc., which has a number of references larger than 1 has this number only because it has been given one or more names.

In such situations "DSTRØY" can be useful, in that it can spare the programmer the necessity of repeatedly using "CLERNM" in order to erase the whole structure with a single command.

A special routine, REGLAR (regularize lists) will be made available at a date in the near future. This routine will process and rearrange the core storage of all the data stored on any Nu-Speak list, compressing as much of it as possible into 15 word data blocks, etc.,

and in this way attempting to make more efficient use of available core storage. The desire to permit such a process accounts for a number of important pecularities of the Nu-Speak freespace-providing mechanisms.

In the regularization process described above, heads (to which program locations may refer) must remain unmoved. For this reason Nu-Speak makes an effort to keep all list-heads confined to as limited a region as is feasible; more precisely, to a belt in core extending from the top of the program area to the bottom of the area used for list elements other than heads. The following diagram, showing the utilization of core storage by Nu-Speak, will make this point clearer.



When a Nu-Speak macro requires a pair of locations to form a listhead (e.g. CØPYTØ, CREATE); the required locations are supplied from the head junkpile, if any such heads are available. If no such heads are available, the list item stored in the lowest pair of core positions above the head area is moved to another

location, this position used for the newly formed head, and the head area boundary advanced upward a corresponding number of locations.

From the programmer's point of view, this means that the actual storage location of a list item is quite indeterminate, and that list items (other than heads), once found and left cannot be found again by returning to a given core position. (There is, moreover, another process which will shift the location in core of a given data item: the automatic breaking of a block of data in a list when an additional item of data is INSERT-d in its middle, or REMØVED from its middle, etc.).

It follows from the above that any list operation but NEXT, PREV, TEDATA and the other tests, JUMPTØ, NAMLIS, and CLERNM, may change the actual core location of a word of data. To be able to return to a word of data, even when the central processor jumps out of the list containing the word, the following procedures may be used.

a. Keep a count of the location of the word relative to its listhead, and return to it by the sequence

```
                JUMPTØ      listhead
BEGIN           DØBEG       END,1,=1,N
                NEXT
END             DØEND       BEG,1
                ...
```

where N is the position number of the desired item in its list.

b. If the list is long, the above procedure may be unduly time-consuming. In this case, the following procedure may be used. Before leaving the list item in question, execute

```
NCPLTØ      auxiliary name
...
```

then return by

```
JUMPTØ      listhead
PREV
CUPL        auxiliary name
```

c. The two procedures described above may be used without any special preparation within recursive systems of SUBR's. In situations in which recursion is not used, the following faster procedure may be used. Before leaving the data item to which a subsequent return is to be made, execute the following lines of machine code, or some similar lines:

```
        SXA     AUXIL1      SAVEXR1
        TESTBL              BREAK BLØCK ØF DATA IF NECS.
        SXA     AUXIL1+1    SAVE XR7
        PIA                 GET IDENTIFIER WD
        STA     *+1         SUPPLY ADR. ØF NEXT WD
        LDC     **,1        XR1=-(ADR ØF CURRENT WD)
        STA     AUXIL       NEXT ØF CURRENT WD BECØMES
*                           NEXT ØF AUXIL
```

```
        SCD     AUXIL,1      CØRRECT WD BECØMES
*                            PREV ØF AUXIL
        PAC     ,7           X7=-(ADR ØF NEXT WD)
        CLA     AUXIL+2      PICK UP ADRESS ØF AUXIL
        STD     0.7          AUXIL BECØMES PREV
*                            ØF NEXT BLØCK
        STA     0,1          AUXIL BECØMES NEXT
*                            ØF CURRENT BLØCK
        TRA     AUXIL1       SKIP ØVER
*                            STØRAGE LØCATIØNS
        EVEN
 AUXIL  ØNE     0,0,0        LØCATIØNS ØF WØRD-PAIR
*                            TØ BE INSERTED
        PZE     0            SECØND WD ØF PAIR
        PZE   AUXIL,0,AUXIL  STØRED ADRS. ØF AUXIL
 AUXIL1 AXT     **,1         RESTØRE XR1
        AXT     **,7         RESTØRE XR7
        ...
```

The above routine will insert the auxiliary 'data item' contained in the pair of locations AUXIL and AUXIL+1 into the list; since these locations lie in the program area, and thus below the head area, none of the list processes can shift the physical position of this item. Return may be made with the sequence

```
        CLA     AUXIL
        PDC     ,7              XR7=-(CURRENT IDENT. ADR)
        PAC     ,6              XR6=-(NEXT-IDENT. ADR)
        STA     0,7             RESTØRE PRØPER NEXT ØF
*                               CURRENT
        STD     0,6             RESTØRE PRØPER PREV. ELT
*                               ØF NEXT
        DLD     0,7             RESTØRE MQ
        PAI                     RESTØRE SI
        LXA   AUXIL1+1,7        RESTØRE XR7
        LAC   listhead,6        RESTØRE XR6 TØ
*                               -(LISTHEAD IDENT ADR)
```

A suitably modified version of this procedure can be used in the context of recursive subroutines.

In the special case in which no list creation can logically intervene between leaving a given list element and returning to it subsequently, the following simple pair of macros is provided to enable the return:

```
        SAVE    MACRØ           A
                STQ             A
                STI             A+1
                SXA             A+2,7
                SXD             A+2,6
        SAVE    END
```

```
GET     MACRØ   A
        LDQ     A
        LDI     A+1
        LXA     A+2,7
        LXD     A+2,6
GET     END
```

The use of these macros should be plain.

B7. The Macros FREESP(ace) and FREEHD.

The macros FREESPACE and FREEHD have the form

FREESPACE

and

FREEHD

respectively. When the instruction FREESPACE is encountered, the address of the first (even location member) of a pair of locations suitable for forming a new list element will appear in the address portion of the otherwise cleared accumulator. FREEHD has much the same effect, but provides a pair of locations suitable for forming a new head. The macro FREEHD should be used with caution, since it has the following side effects:

i. The contents of the sense indicator are taken as the identifier word of a given list element. This word is stored, and when restored may have modified address, and/or decrement, and/or tag portion, if the rearrangement of the storage of link elements in core consequent on using

FREEHD has moved the "next" or "previous" element of this list element, or a word originally in the same block as this list element, etc.

ii. The process of rearranging list elements in core which may be consequent on using FREEHD assumes that all lists have their normal structure. Thus FREEHD should never be used unless all "next pointers" and "previous pointers" of all lists are appropriately set. (The Nu-Speak list operations will never disturb this condition, but, unless the programmer takes care, his own intermediate steps of additional FAP list processes might.)

In view of the touchiness of FREEHD, the programmer will normally prefer to use CREATE in order to obtain a new listhead.

The following remarks summarize the effect of the Nu-Speak list operations on the number of references to the lists they concern NEXT, PREV, CUPL, INSERT, REMØVE, JUMPTØ, SUBLIS, and the various tests (TEHEAD, etc.) have no effect. NCPLTØ does not affect the reference number of the list which it breaks up; the computer is left looking at the head of the newly broken off second part of the list, which hence has a reference number of 1. CREATE does not affect the number of references to any list, but leaves the computer looking at the head of the newly created list, which therefore has a reference number of 1. NAMLIS increases the number of references

to the newly named list by 1; CLERNM does the opposite. HANG increases the number of references to the sublist being hung by 1; and UNHANG decreases this same reference number by 1.

B8. Miscellaneous macros.

In a Nu-Speak assembly, the control PRNTMC replaces the FAP control PMC (print macro-generated cards). Use of PMC will fail in erratic ways.

The control card HEADER may be used to provide a heading character; these characters will be provided in sequence from the list Z, X, Y, ..., A, 9, ..., 1; other characters chosen from this list will head the variables occurring in SUBRS. The heading of a section of code can be terminated by the normal FAP card HEAD 0.

B9. Forbidden macro-words and entry symbols.

The following symbols are used as intermediate macro-names in Nu-Speak, and should never be used as macro-names by a programmer using Nu-Speak who has not pondered long over the profound changes in the system thereby introduced.

PØN..., PØF..., QØN..., PSW..., HØD..., HED..., HEADER, HAD..., BGS..., STR..., RST..., SB1..., SB2..., IRS..., MICRØ, RPR..., ØPA..., IFØ..., TIMESA, ØVERA, STØ..., CLA... , MUCRØ .

The following wierd symbols are used as control symbols or as entries to various Nu-Speak programs, and should not be used by the Nu-Speak user unless he knows what he is doing.

..0..., ..1..., ..2..., ..3..., ..5..., ..6..., ..7..., ..9..., ..A..., ..B..., ..C..., ..D..., ..E..., ..F..., ..G..., ..H..., ..J..., ..K..., ..N..., ..Ø..., ..Z...,)....(,).((.(,).(..(,).)(.(, ..IN..., ..NN..., ..ØT..., ..ØG..., ..ØW...,)()(.),)))))), ..HT.. ,

EXHAUS, CØNLØC, (ERR1) through (ER12) and (ER20), ..BN.., ..BB.., ..BØ.., ..BC.., ..LNK2, (.LG.), (.LB.), .R.PLT, .LET..., .L2.., ..LO.., .R....

HDCNTR, CARDSW, BUGZZQ, X1...., X2...., X3...., X5...., X6...., X7...., SI...., MQ...., 1....., 2....., XXSIXX, XXX4XX, XXMQXX, REPTSW, CØUNTR, CØUNTD, MULTSW, ..GG.., ..GH.., ..GI.. .

B10. Special Procedures for Dealing with Self-Reflexive List Structures.

The S-bit and first 20 bits of the second word of a listhead as formed by the Nu-Speak list-creating operations CREATE, NCPLTØ, CØPYTØ will always be blank; and the remaining Nu-Speak list processes will not affect these bits. (A similar remark can be made concerning links.) If the programmer takes the precaution never to tamper

with the S-bit of these listhead words, the following procedures may be used to deal with reflexive list structures.

i. A recursive routine MARKER, called by the sequence

CALL MARKER, listname

where "listname" is the name of a list in a self-reflexive structure, is available. This instruction will cause the 1-bit of the second word of the list head "listname", every one of its sublists, every sublist of a sublist, etc., to be set to 1.

ii. Recursive procedures may then be applied to the lists of a self-reflexive structure and their sublists provided that

a. A sublist is marked as "entered" by setting the 1-bit of the second word of its head equal to zero as soon as it is reached by the recursive procedure.

b. The recursive procedure avoids entering a list whose head shows that it has already been entered.

iii. In view of the catastrophe which will always ensue on erasing a sublist of a list either before unhanging the sublist or before erasing the main list, special care must be taken in the erasure of self-reflexive list structures or sublists thereof. These include:

a. To erase the whole structure, use a recursive procedure as above to UNHANG every sublist of every list, creating at the same time a (non-redundant) auxiliary list of the locations of all the listheads of the structure.

b. To erase one list of a structure, use a recursive procedure as above to UNHANG it from every list of the structure. Then erase the list in question.

B11. The Form of a Nu-Speak Deck.

In an attempt to save as much core space as possible from unessential uses, Nu-Speak utilizes chaining. Nu-Speak jobs are chained as follows:

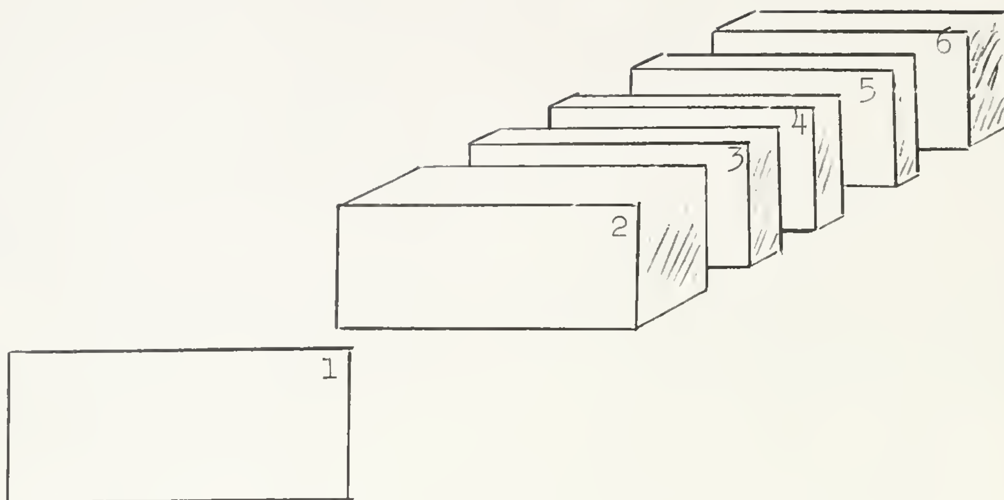
a. A first link, created by the chain control routine

```
*    CHAIN     (101,3)
```

This link contains the main program section. A second link contains the error and freespace-exhausted messages and procedures, and is created by a chain control card

```
*    CHAIN     (102,3)
*    FAP
     TRA       $..LNK2
     END
```

These conventions cause the Nu-Speak deck to have the following form



1 User's *ID card, either with or without an *XEQ execution order card

2 Nu-Speak control deck, always provided already containing all necessary *FAP and chain-creating control cards, and containing binary cards for first (preliminary) link programs. This deck will cause a single instruction, TSX $..W...,4 to assemble (which executes a preliminary initialization before returning to the user's program).

3 User's symbolic program, terminated by an END card

4 User's binary decks

5 Nu-Speak binary decks, either

i. minimum package,

ii. string manipulation package,

iii. total package, or

iv. any of the above, plus the "debug" binary package

6 Data cards, if any, preceded by the control *DATA

By installing some of the Nu-Speak programs on the library tape, and by installing the macros on an auxiliary update tape, various of these deck sections can be abbreviated or eliminated.

To provide a subtitle for his output, the user can either

a) put a *FAP immediately after his first *ID and *XEQ cards, followed by the title of his program, or

b) use an appropriate TTL card early in the symbolic deck.

C1. The Auxiliary Routines WRTLIS and INLSTR.

a) Output of List Structures.

For writing out lists or list structures, three routines are provided.

WRTSTR -- a routine for writing out any list structure

WRINCS -- a routine for writing out a non-reflexive list structure

WRTLIS -- a routine for writing out the elements of a single list.

These three list output routines write out a list or an entire structure, ten list elements per line, in both alphabetic and octal formats. (A list element is a head, link, or data element.) In addition, the word HEAD or LINK is written out next to a list element which is a head or link.

The routine WRTLIS writes out only the elements of a single list; it does not descend to sublists. It is called by a statement of the form

```
CALL      WRTLIS, LIST
```

where list is a listname.

Both WRINCS and WRTSTR recursively write out the entire structure of a list. WRTSTR calls MARKER and can write out any list structure, even one which is reflexive. WRINCS does not call MARKER and can successfully write out only a non-reflexive list structure. These routines are called by statements of the form

```
CALL      WRTSTR, LIST
```

and

```
CALL      WRINCS, LIST
```

These routines perform the recursive output of list structures as follows:

1) The argument of the routine becomes the "present" list, which is written out in its entirety. (The location of the listhead is written out prior to

the list itself.) The present list is marked as having been processed.

2) The first previously unprocessed sublist of the "present" list is searched for. If such a list exists, it becomes the argument of the recursive routine (step 1). Otherwise, the substructure of the "present" list has been completely written out; exit is made from this level of the recursive routine.

b) Input of List Structures

The subroutine INLSTR is provided for the input of data into list form. The statement

```
CALL      INLSTR, NAME
```

will cause the data which is next on line to be read from the monitor input tape to be read into core as a list with name NAME. The data to be read in by INLSTR should be in a regular monitor data deck which is headed by a *DATA card. The data to be put into list form should have the format described below.

Data Format

A list is begun on and continued on cards with a "1" in column 1, a series of 5 symbol redefinitions (described below) in columns 2 through 6, and the actual list elements and list control characters beginning in column 7 and not running beyond column 72. A list is ended by a blank card.

List Control Characters

The following characters have special control functions which communicate to INLSTR the form in which the list is to be stored in core. These characters cannot be used as letters of the actual list elements unless redefined in the symbol redefinitions field of the card throughout which the redefinitions are to hold.

These are the control characters:

, The comma is used to separate list elements.

A, BC, BETA, ...

(The left parenthesis is used to begin a sublist. Thus, the data

A, BC, BETA (SUB, LIST

will cause a sublist beginning with the words "SUB" and "LIST" to be hung on the list of which "A", "BC", and "BETA" are elements. The left parenthesis <u>must be the symbol in column seven of the first card of a list</u>.

) The right parenthesis is used to end a sublist. Thus, the data

A, B, (C, D), E

will cause the data word E to be an element of the same list as "A" and "B". The main list must be ended by a right parenthesis.

$ The dollar sign is used to signify the end of a card. Thus, the data

A, B$

will cause INLSTR to begin processing the next data card in the deck after processing "A" and "B".

/ The slash is used to indicate that the word following is to be recognized as a decimal number and is to be converted into a 36 bit binary fixed or floating point core word. A minus sign preceding the number will cause it to be stored in core with the S-bit on (negative number). The absence of a minus sign designates the number as positive. The plus sign should never be used. The appearance of a decimal point in the number will cause it to be converted to a floating point core word. The absence of a decimal point designates the number as an integer. The magnitude of an integer must not exceed $2^{35}-1$, and floating point quantities should have no more than 8 significant digits. The appearance of an Ø directly after the slash causes the word following to be taken as an octal integer. An octal integer may have from 0 to 12 octal digits.

A list may be continued from one card to another by simply ending the first card in the pair with a dollar sign and continuing where the list was cut off in column 7 of the next card. Columns 1-6 of the second card must contain the normal control information (see below). Single words cannot be continued from one card to another.

Redefinitions of Control Characters

Since it is sometimes necessary to treat the above special characters as actual letters, to be read into core as part of data words rather than as symbols which only perform a control function and which are not read into core, an optional redefinition feature is provided. To redefine a control character, place a valid BCD character in the appropriate column of the redefinition field of the card throughout which the redefinition is to hold. The inserted character will be taken to have the function of the control character it redefines. The old control character is then free for use as an actual letter. Redefinitions hold only over the card on which they appear. A blank in a redefinition column causes the corresponding control character to remain as shown on previous pages(control characters). The original control characters are reinstated at the beginning of the processing of each new card.

The redefinition columns are as follows:

Column	2	redefines	(
"	3	"	$
"	4	"	,
"	5	"	/
"	6	"	)

Thus, to enable the word "AL,PHA" to read into core as the single word "AL,PHA" rather than as the two words "AL" and "PHA," a convenient symbol, say "*", should be inserted in column 4 of the card on which "AL,PHA" is found. The symbol "*" should then be used throughout the data card instead of "," where word separation is necessary. In effect, "*" becomes "," and "," is freed for use as a normal letter to be read into core in regular data fashion. The following example should illustrate the use of INLSTR.

The statement

```
        CALL        INLSTR, LSTNAM
```

if used on the following data

	Cols.	1	2	3	4	5	6	7
CD1		1						(NØW,IS(THE(TIME$
CD2		1	*		=			F(ØR=*,ALL,=GØØD$
CD3		1		X	*		=	MEN*TØ,,,=C$ØME==TØ=X
CD4								

will cause these lists to be placed in core:

Name				List Elements	
LSTNAM	Head	NØW	IS	pointer to list1	TØ
list1	Head	THE		pointer to list2	
list2	Head	TIME	F(ØR	pointer to list3	C$OME
list3	Head	,ALL,	GØØD	MEN	TØ,,,

The computer will be left looking at the head of LSTNAM.

An error message and a subsequent call exit statement will be produced should a list contain a word of more than six BCD characters or an unbalanced parenthesis.

C2. An Auxiliary Package of Subroutines for Letter Manipulation.

The Nu-Speak letter macro package enables the programmer to manipulate letters within words in a manner similar to the manipulation of words within blocks. The letter macros will normally be used on data words in logical 6 letter BCD format.

Nu-Speak Letter Conventions

The computer is said to be looking at a letter when:

1) A copy of the ID of the word in which the letter is contained is in the SI.
2) The word in which the letter is contained is found in the MQ with the letter rotated left into positions 29-35 of the MQ.
3) The letter itself appears in positions 29-35 of the AC.

(Note: This convention is violated by certain of the letter macros, see below.)

4) XR7 contains the number of the word containing the letter in its block.

5) XR5 contains the number of the letter in its word counting from left to right as the word appears in storage.

In each program using letter processing, the code

```
AXT        0, 5
```

should be executed before the first of the letter macros is reached.

The BCD blank, octal 60, is treated as a normal character by the letter macros. The special Nu-Speak functional blank, octal 77, should be used when the absence of a normal BCD character is required.

The macros NEXT LETTER and PREVIOUS LETTER have the form:

```
NEXTL
```

and

```
PREVL
```

These macros cause the computer to look at the next and previous "non-functional blank" character in the word respectively. Nu-Speak functional blanks are skipped but will affect the count in XR5. Execution of a NEXTL while the computer is looking at the end of a word, (XR5 = 6), will cause the CPU to look at the first

non-functional blank character in the next word of the list. Should the next word in the list be non-data (i.e. head or link), the CPU will look at it in the normal Nu-Speak fashion, and XR5 will be set to 0. The analogous feature is included in PREVL. (Note: If PREVL should run into a non-data word, XR5 will be set to 7.) Execution of a NEXTL or PREVL while the computer is looking at a non-data word will cause the CPU to look at the first non-functional blank character in the NEXT and PREVIOUS words respectively. The new letter found by either NEXTL or PREVL will always be placed in positions 29-35 of the AC.

Please note the following examples of the use of NEXTL and PREVL:

Note: WD_n is taken to follow WD_{n-1} in its list.

XR5 should initially be set to 0 by the programmer.

WD_1 = head of LIST,

WD_2 = A(77)(77)(77)P(77)

WD_3 = (77)X(77)(77)(77)(77)

WD_4 = (77)(77)(77)(77)(77)T

WD_5 = link pointing to sublist 1

WD_6 = link pointing to sublist 2

MACRO	MQ	XR5
JUMPTØ LIST	WD_1	0
NEXTL	(77)(77)(77)P(77)A	1
NEXTL	(77)A(77)(77)(77)P	5
NEXTL	(77)(77)(77)(77)(77)X	2

MACRO (Cont'd.)	MQ	XR5
NEXTL	(77)(77)(77)(77)(77)T	6
NEXTL	WD_5	0
NEXTL	WD_6	0
PREVL	WD_5	7
PREVL	(77)(77)(77)(77)(77)T	6

Note that if the contents of the MQ are changed, for example at step (4), the results at step (5) would not be altered. All letter macros refer to the contents of XR5 and of core for data on which to base their operations — not to the MQ.

The macro REPLACE LETTER has the form:

REPLET A

The macro will replace the letter being looked at with the letter in positions 29-35 of location A. The change will appear in the AC, the MQ, and in storage. The CPU will be left looking at the new letter.

To delete a letter, two procedures may be used:

1) The code REPLET = Ø77 will replace the letter being looked at by (77). Thus, in effect, the letter is deleted since the macros NEXTL and PREVL skip functional blanks. The CPU will be left looking at the functional blank. The AC will be set to 77.
2) The macro DELETL may be used. DELETL will replace the letter being looked at by a functional blank and will cause the CPU to look at the next letter in the list (i.e. DELETL automatically calls NEXTL). Should the CPU be looking at a non-data word upon executing a DELTL, the

entire data word will be deleted (i.e. as with REMØVE). The CPU will then be left looking at the next word in the list but not at any particular letter in the word. XR5 will be set to zero.

The macro TEST AND BREAK WØRD has the form

TESTL

The macro will check to see whether the letter being looked at by the CPU is the last letter in its word (i.e. if XR5 = 6). If it is, the macro will perform no additional function. If it is not, the macro will split the word containing the letter being looked at into two words in a manner illustrated by the following example:

before TESTL, XR5 = 4, MQ = ØHMATZ, and WD_n = MATZØH

after TESTL, XR5 = 4, MQ = (77)(77)MATZ, WD_N= MATZ(77)(77),

and WD_{N+1} = (77)(77)(77)(77)ØH

WD_{N+1} will be inserted, in normal Nu-Speak fashion, after WD_n in the list. The CPU will be left looking at the same letter as it had before the execution of the TESTL.

The macro INSERT LETTER has the form

INSERL LØC

The macro will cause the letter, or letters, found in location LØC to be inserted directly after the letter being looked at. If the number of letters to be inserted is less than 6, the surplus digits in LØC must contain functional blanks. The insert will be made in the following manner:

1) A TESTL will be executed

2) An INSERT LØC will be executed

Please note the following example:

Before INSERL - XR5 = 4   MQ = ØHMATZ WD_n = MATZØH

LØC = (77)(77)XYQ(77)

After INSERL - XR5 = 3   MQ = YQ(77)(77)(77)X

WD_N = MATZ(77)(77)

WD_{N+1} = (77)(77)XYQ(77)

WD_{N+2} = (77)(77)(77)(77)ØH

Note that letters X, Y, and Q are in effect inserted between Z and Ø. This is due to the fact NEXTL and PREVL skip functional blanks.

The CPU will be left looking at the first non-functional blank letter in LØC.

The macro IF LETTER has the form

IFLET   A, B, C

If the letter contained in positions 29-35 of location A is identical with the letter being looked at by the CPU, control will be transferred to location B. If not, control will be transferred to location C.

The macro CØUPLE LETTER has the form

CUPLL   A

The macro will form a TESTL word break directly after the letter being looked at and will then execute a Nu-Speak

CUPL   A

XR5 will be set to 0. Thus, in effect, the letters of the list named A are inserted directly after the letter at which the CPU was looking, and directly before the letter which preceded it.

The macro UNCØUPLE LETTER has the form

NCPLTL   A

The macro will form a TESTL word break directly after the letter being looked at by the CPU and will then execute a Nu-Speak

NCPLTØ   A

XR5 will be set to 0. Thus, in effect, all the letters in the list which had followed the letter under examination are UNCØUPLED from the list.

C3. An Example of Nu-Speak Applications Programming: PØLPAC.

The PØLPAC package is coded in Nu-Speak and is designed for substituting , partially differentiating, and algebraically manipulating symbolic polynomials.

A PØLPAC polynomial is a Nu-Speak list structure of the following form:

The polynomial is a list which consists entirely of links. The sublist to which each link points is a term of the polynomial. The first element of each term is a floating point number and is the coefficient of the term. Succeeding elements of the term are variables raised to powers, each in the format

VARIABLE	EXPONENT
S,1 20	21 35

That is, the variable name occupies positions S, 1-20, and the exponent is a right-adjusted address integer.

The coefficient is always the first data element of a term. Within a given term, the variables are arranged according to increasing numeric order, with the S-bit treated in the sense of the "logical" machine instructions.

Example:

The polynomial $1.5 + 3.5\ x^2\ y^2$ would be generated as follows:

```
                CREATE      TERM1
                INSERT      1.5
                CREATE      TERM2
                INSERT      3.5
                INSERT      X2
                INSERT      Y3
                CREATE      PØLY
                HANG        TERM1
                HANG        TERM2
                  .
                  .
                  .
TERM1           PZE
TERM2           PZE
PØLY            PZE
1.5             DEC         1.5
3.5             DEC         3.5
X2              BCI         1,      X002
Y3              BCI         1,      Y003
```

At this point we will define a checklist, which is used to truncate the high-order terms of a polynomial. A checklist has the same form as a term of a polynomial except for the absence of a coefficient in the checklist. A term is said to exceed a checklist if and only if any one of the variables in the checklist appears in the term raised to a power higher than its exponent in the checklist.

The PØLPAC routines are as follows:

Subroutine CØMPAR - compares a given term to a checklist.

Calling sequence: CALL CØMPAR,TERM,CKLIS,YESNØ

TERM is the name of a term of a polynomial

CKLIS is the name of a checklist

YESNØ is a core location.

Upon return from CØMPAR, YESNØ is zero if and only if TERM does not exceed CKLIS. Note: The variables of CKLIS must be alphabetically ordered.

Subroutine ØRDERR - orders variables within a term.

Calling sequence: CALL ØRDERR,TERM1,TERM2

TERM1,TERM2 are both names of terms.

ØRDERR alphabetically orders the input term TERM1, which is erased. Upon return from ØRDERR, TERM2 is the alphabetization of TERM1.

Subroutine ISSAME - tests for identity of two terms.

Calling sequence: CALL ISSAME,TERM1;TERM2,YESNØ

TERM1, TERM2 are names of terms

YESNØ is a core location

Upon return from ISSAME, YESNØ is non-zero if and only if TERM1 and TERM2 are identical, except for coefficients. Thus the terms 1.5 and 2.7 are identical, as are $1.5\ x^2\ y^3$ and $3.7\ x^2\ y^3$, but $2.5\ x^2\ y^2\ z^2$ and $2.5\ x^4\ y^2$ are not.

Subroutine VARPØW - arranges the terms of a polynomial according to the exponent of a given variable.

Calling sequence: CALL VARPØW,PØLYN,VAR,ERASE,PØLYØ

PØLYN, PØLYØ are polynomial names

VAR, ERASE are core storage locations

VARPØW rearranges the terms of PØLYN into a new polynomial, PØLYØ, by ascending order of the exponent of the variable name in S, 1-20 of the word VAR. This change is effected by reordering the links of the list PØLYN into the new list PØLYØ. The decrement of the second word of each link of PØLYØ contains the exponent of VAR in the term to which the link points. The variable VAR will be deleted from the term of PØLYN if and only if ERASE is zero. With

the exception of the possible deletion of the variable VAR, the terms of PØLYN will not be changed and are hung from PØLYØ. The list PØLYN (that is, only the sequence of links) will be erased by VARPØW.

Subroutine <u>CØLECT</u> - collects terms of a polynomial.

Calling sequence: CALL CØLECT,PØLY

PØLY is the name of a polynomial

CØLECT condenses the polynomial PØLY by adding together the coefficients of matching terms. (Two terms, TERM1 and TERM2, are said to match if

CALL ISSAME,TERM1,TERM2,YESNØ

returns a zero to YESNØ.) Any terms which match and follow a given term of PØLY are unhung from PØLY and erased.

Subroutine <u>PØLYMP</u> - multiplies two polynomials while truncating according to a checklist.

Calling sequence: CALL PØLYMP,PØLYA,PØLYB,CKLIS,PØLYC

PØLYA,PØLYB,PØLYC are polynomial names

CKLIS is the name of a checklist

PØLYC contains the product of PØLYA and PØLYB. Only those terms of the product which do not exceed CKLIS appear in PØLYC. Those terms which do appear are alphabetically ordered. PØLYA and PØLYB are unchanged.

Subroutine PRTIAL - partially differentiates a polynomial with respect to a given variable.

Calling sequence: CALL PRTIAL,PØLYA,VAR,PØLYB

PØLYA, PØLYB are polynomial names

VAR is a core storage location

PRTIAL differentiates PØLYA with respect to the variable name in positions S, 1-20 of VAR. PØLYB is the partial derivative; PØLYA is left untouched.

Subroutine SUBSTI - substitutes a polynomial for a given variable of another polynomial while truncating according to a checklist.

Calling sequence: CALL SUBSTI,PØLYN,PØLYS,VAR,CKLIS,PØLYØ

PØLYN, PØLYS, PØLYØ are polynomial names

CKLIS is the name of a checklist

VAR is a core storage location

SUBSTI substitutes the polynomial PØLYS for each appearance in PØLYN of the variable whose name appears in positions S, 1-20 of location VAR. The expanded polynomial is named PØLYØ. PØLYN is destroyed. PØLYS is left unchanged. Only those terms which do not exceed the checklist will appear in PØLYØ.

Subroutine <u>PØLYRC</u> - finds the reciprocal of a polynomial to a given degree, while concurrently truncating according to a checklist.

Calling sequence: CALL PØLYRC,PØLYN,DEG,CKLIS,PØLYØ

PØLYN, PØLYØ are polynomial names.

CKLIS is the name of a checklist

DEG is an address integer

PØLYRC finds the reciprocal of a polynomial PØLYN according to the following algorithm:

Let

$$\text{PØLYN} = c(1 + \text{PØLYI})$$

Then

$$\text{PØLYØ} = \frac{1}{\text{PØLYN}} = \frac{1}{c}\,(1 - \text{PØLYI} + \text{PØLYI}^2 \ldots)$$

c is the unique non-zero constant term of PØLYN.

The address integer DEG specifies the number of terms which are to be computed in the above expansion. Only those terms of the expansion which do not exceed the checklist will appear in PØLYØ. PØLYN is left unchanged.

Subroutine <u>PØLΦUT</u> - polynomial output routine.

Calling sequence: CALL PØLØUT,PØLY

PØLY will be written out in easily understandable format.

INDEX

A ALDUMP 46, 60

ARITH 25, 40

ARITHA 25, 40

Arithmetic in Fap Nu-Speak 40-42

B BCD letters

looking at 89

manipulation of 89-95

C CALSV4 24, 38

CLERNM 5, 16, 26, 54-5

CNTSTK 25, 46, 48, 49

CØMIN 5, 20

CØMØUT 5, 20, 22

CØPYTØ 5, 10, 26, 38

Core allocation of 30, 70

automatic dump 46, 60

saving features 29

CREATE 5, 6, 8, 26, 56

CUPL 5, 11, 26, 52, 53

CUPLL 94

CURRNT 5, 16

D DEBUG 24, 39

DØBEG 24, 34, 42, 43, 45

DØEND 24, 34, 42, 43, 45

DSTRØY 5, 18, 28, 68-69

E ERASER 5, 9, 26, 88

ERASØR 26, 58

ERASYR 28

Error messages 80

F FINI 24

Fixed point mode in Fortran Nu-Speak 17

FLØAT 25, 41

FLØATA 25, 41

FREEHD 27, 75

FREESP 27, 75

G GET 75

GETTER 5, 19

H HANG 5, 13, 64

HEADER 28, 35, 77

I IFLET 94

INLSTR 84-89

INSERL 93

INSERT 5, 8, 26, 53

INTARS 24, 32

J JUMPTØ 5, 9, 26, 55

L List

Automatic erasure of 17, 18, 55, 67

Elements of

automatic shifting around core of 71,75-6

data 9, 49

head 4, 50

formation of 70

junkpile 70

link 4, 51, 63

List (Cont'd.)

link (Cont'd.)

procedures for keeping track of shifting of 71-75

Erasure of entire structure of 18,28,68-9

cautions involved in 14,15,68-9

Identifier 49

Input-Output of 82-89

Identifier 49

Reference number of 50,51,55,64

Reflexive 14,15,65

special procedures involved in dealing with 78-80

Structure of 4,14,65,66

Sublist 5, 63

M MARKER 79

N NAMLIS 5, 15, 26, 28, 54-55

NCPLTL 95

NCPLTØ 5, 10, 26, 56

NEXT 5, 7, 26, 52

NEXTL 90-92

NØDUMP 60

NØWW 5, 16

Nu-Speak System package options 29

P PMC failure of 77

substitute for 28, 77

PØLPAC CØLLECT 99

PØLPAC (Cont'd.)

CØMPAR 97

ISSAME 98

ØRDERR 97

PØLØUT 101

Polynomial

creation of 96

form of 95

PØLXMP 99

PØLYRC 101

PRTIAL 100

SUBSTI 100

VARPØW 98

PREV 5, 7, 26, 52

PREVL 90-92

PRNTMC 28, 77

Push down stack

as a substitute for INSERT and REMØVE 48

definition of 46

emptying 48

filling 48

overflow, underflow 47

speed of 48

PUTIN 24, 46, 48

R RDØBEG 24, 42, 44

RDØEND 24, 42, 44

READY 5, 6, 27, 60

Recursive Subroutine 20, 31-9

- as an argument of a subroutine 33-4
- automatic heading of variables in 35-6
- calling
  - by external programs 38
  - in FAP 37, 38
  - in Fortran 21
- example of use in FAP 25
- named common of 36
- naming 36
- restrictions on in Fortran 22

RELEAS 26, 59

REMØVE 5, 8, 26, 53

REPLET 92

REPT 25

RETFRM 29, 37

S SAVE 74

SAVER 5, 19

STEP 25

SUBLIS 5, 14, 28, 67

SUBR 24, 31

Summary of list macros 76-7

T TAKFRM 25, 46, 48

TEDATA 26, 55

TEHEAD 26,55

TELINK 28, 64

	TESTBL	67
	TESTL	93
	TYPE	5,6
U	UNHANG	5, 13, 28, 66
	USESV4	34, 38
W	WRINCS	82-83
	WRTLIS	82-83
	WRTSTR	82-83