Forward into the '90s

Donald E. Knuth
Grand Wizard of TeX

TeX Users Group Conference, College Station, Texas, June 18–20, 1990

The compiler "was beautifully designed, very fully documented, and patched at the machine level so we couldn't recompile it or reassemble it."

- Betty Holberton

Ann Alt Camp 20 (85) 18.
The compiler described herein was developed by Computer Sciences Corporation, Palos Verdes, California, under contract with Remington Rand Univac, Division of Sperry Rand Corporation.

Acknowledgment is given below to those members of the CSC staff whose efforts in design and implementation resulted in the system described herein.

Joel D. Erdwinn (Phases VI and VIII)
David E. Ferguson (Preliminary Design, Phase VII, and Sorting System)
Louis Gatt (Project Director)
Reginald E. Martin (Phase I and VII)
Jack P. Middlekauff (FILE Program and I/O Routines)
Hayden T. Richards (Phases IV, V, and VI)
David W. Roberts (Phases II, III, and IX)
INTRODUCTION

The Larc Scientific Compiler (LSC) is herein described by prose as well as by detailed flow charts. The major function of each phase is explained. Following the explanation, flow charts for that phase will immediately follow.

The peripheral programs associated with LSC are also described (See LSCN). In that section a description of the input and output system available to the object program is contained in the form of prose and flow charts.

During implementation, special attention was given to describing the coding of LSC with comments. These comments appear in the parallel code edits of the entire LSC system and should be helpful in going through the compiler.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Flow of Control in LSC</td>
<td>0.0</td>
</tr>
<tr>
<td>General Flow of Data in LSC</td>
<td>0.2</td>
</tr>
<tr>
<td>Input and Output of LSC</td>
<td>0.3</td>
</tr>
<tr>
<td>LSCN</td>
<td></td>
</tr>
<tr>
<td>I. Main Control</td>
<td>0.4</td>
</tr>
<tr>
<td>II. The Communication Region</td>
<td>0.4</td>
</tr>
<tr>
<td>III. The Sorting System</td>
<td>0.5</td>
</tr>
<tr>
<td>IV. LSCN Diagnostic System</td>
<td>0.5</td>
</tr>
<tr>
<td>Sorting System Flow Charts</td>
<td>0.7</td>
</tr>
<tr>
<td>File Program</td>
<td>0.16</td>
</tr>
<tr>
<td>File Program Flow Charts</td>
<td>0.24</td>
</tr>
<tr>
<td>Object Program Input-Output Routines</td>
<td>0.31</td>
</tr>
<tr>
<td>Input-Output Routines Flow Charts</td>
<td>0.34</td>
</tr>
</tbody>
</table>

**Phase I**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Description</td>
<td>1.0</td>
</tr>
<tr>
<td>Sequence Counters</td>
<td>1.4</td>
</tr>
<tr>
<td>Item Formats</td>
<td>1.8</td>
</tr>
<tr>
<td>Identification Digits Assignment</td>
<td>1.21</td>
</tr>
<tr>
<td>Mode Word</td>
<td>1.23</td>
</tr>
<tr>
<td>Flow Charts</td>
<td>1.28</td>
</tr>
</tbody>
</table>

**Phase II**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Description</td>
<td>2.0</td>
</tr>
<tr>
<td>Item Formats</td>
<td>2.3</td>
</tr>
<tr>
<td>Flow Charts</td>
<td>2.6</td>
</tr>
</tbody>
</table>
Phase III

General Description 3.0
Output to File 4 3.8
Output to File 91 3.17
Flow Charts 3.19

Phase IV

General Description 4.0
Input and Output 4.2
Treatment of:
FUNCTION and SUBROUTINE 4.3
LIBRARY CALLS 4.3
PARAMETERS 4.3
SIMPLE VARIABLES 4.4
DIMENSION 4.5
EQUIVALENCE and COMMON 4.5
CONSTANT 4.7
ARITHMETIC STATEMENT FUNCTIONS 4.7
EXECUTABLE STATEMENTS 4.7
Subroutines 4.14
Flow Charts 4.19

Phase V

General Description 5.0
Input 5.0
a – Rename Variables and Constant Subscripts 5.1
b – Number Subscript Terms 5.2
c – File Loops by Level 5.3
d – Unpack Terms 5.4
e – Form Non-constant Counts 5.4
f – Form Non-constant Initial and Incremental Multiples 5.4
g – Form Subscripts 5.5
Special Subroutines 5.7
Output to File 61 5.9
Format of Phase V Internal Files 5.12
Flow Charts 5.13

Phase VI
General Description 6.0
First Phase VI - Pass 6.1
Every Phase VI - Pass 6.2
Output 6.3
Flow Charts 6.4

Phase VII
General Description 7.0
Generator 7.1
Successor Item Generation 7.6
Generation of Arithmetic Expressions 7.7
Examples of Object Code 7.11
Library Routines 7.14
Flow Charts 7.15

Phase VIII
a - Backward Scan 8.0
b - Forward Scan 8.2
c - Build Flow Tables 8.10
d - Index Analysis 8.11
e - Produce F and S Commands 8.13
Item Formats 8.15
Flow Charts 8.27
Phase IX

General Description 9.0
a - Output Error Messages 9.1
b - Output of LSC 9.2
Name Generation 9.4
File 91 9.7
File 92 9.8
File 94 Items from Phase IV 9.10
File 90 and 94 Items from Phase VIII 9.13
Flow Charts 9.19
LOAD LSCN
1. Control and phase loader.
2. FILE.
3. SORT.
4. Diagnostic system.

Phase I:
1. Analyze source input
   a. Found on drum located by CRDRI.
2. Output items to:
   F2, F3, F4, F91, F93.
   (error items to F92 in all phases.)

SORT F2 on name (each group has same name).

Phase II: Input F2, generate more F3.
1. Associate statement names, N, with
   N in assign, to M.
2. Output items for built-in Function.

SORT F3 on name and sequence number.
All F3 items have a name, and all
name items are in F3.

Phase III: Input F3, output more F4.
Associate defining items with
reference to a name.
1. Generate mode word for item.
2. File names in F91.
3. Assign "Parameter" to references.
4. Set up Dimension table.
5. Detect errors of illegal use of a
   name.

F4 now has items for the entire
program (except FORMATS and
Hollerith arguments of calls.)
SORT F4 on the Phase I assigned
sequence number. (puts all expres-
sions in "POLISH" form.)

Phase IV:
1. Fill in Dimension table,
2. Solve Equivalence and COMMON
   relationship a. output F94.
3. Set up storage assignment for all
   variables.
4. Update definition point for variables.
5. Update modes of arithmetic
   operations.
6. Output to F50, F51 and F52
   information on subscripts, DO-loops
   and lists.
7. Converts subscripts to single
   expressions.
8. Outputs program to F60, F61.
Phase VII.
F7 is in sort.
F7 now contains the entire program where each item now contains all the information necessary to form code.
Output is to:
1. F80, the main body of code items in sort.
2. F85 code items out of sort.
The output also contains information items to Phase VIII. This will permit better optimization of A-registers.

Phase VIII.
The input F80, F85 is merged and analyzed with respect to optimum use of A-registers from the standpoint of indexing quantities and arithmetic registers.
Output is to:
1. F90, the main body of the program in sort.
2. F94, additional out of sort code-items.
Phase IX.
Edits the generated object code in a SAL accepted form.
1. Associates error messages with items in F92, output to F93.
2. SORT F93:
   a. The original source lines and error messages as SAL comments.
   b. FORMATS and Hollerith arguments as SAL - ALPH lines.
3. Merges F90, F94 and F93, forming SAL acceptable lines to F10. F10 is output according to CRDRØ.

Return to Operator Program.
DATA FLOW IN LSC. (continued)

** All Phases may contribute to F 92.

The following files are sorted before use: 2, 4, 50, 51, 52, 53, 55, 61, 82, 83, 84, 85, 93, 94.
INPUT AND OUTPUT OF LSC.

The input described in the Programmer's Reference Manual, is given to LSC either on tape or on a specified drum. The Communication Region (CR) contains a flag, CRT, which is zero or non-zero.

1. If CRT ≠ 0 input is on tape 11 in alphanumeric form, 2 lines per blockette.
2. If CRT = 0 input is on a drum. The starting sector, band, and drum numbers are specified in CRDRI.

During the operation of LSC, seven drums must be made available for temporary storage of generated files, and for the Sort-Merge program. The seven drums must have consecutive drum numbers, the first of which has drum number:

\[(CRDRF) + 1\]

The input drum, specified by CRDRI, may be one of the seven temporary drums if it is equivalent to: \((CRDRF) \div 1\), \(\equiv (CRDRF) + 2\), or \((CRDRF) + 6\).

The output of LSC is always output on a drum; however, it may also be output on tape or the on-line printer. The output drum is specified by CRDRO by specifying the starting sector and band of a drum. CRDRO must not be any of the temporary drums. The output itself is more fully described in the Phase IX section of this report.

If CRHSP is non zero, the output is also generated for the on-line high speed printer or a tape. This choice is specified by setting CROUT to either zero or the appropriate tape servo number. For example:

\[CRHSP \neq 0\]

and \[CROUT \neq 21\] output is sent to drum and on tape servo 21. When the output is generated for the high speed on-line printer or for tape, an additional line is written, which contains an internal sequence number as a SAL comment. The tape so generated may be used as input to SAL.
LSCN

Compilation of a source program begins by bringing into memory that portion of LSC labeled LSCN. This will consist of:

1. The LSC Main Control and drum loader (Main Control)
2. The Communication region
3. The sorting system
4. The diagnostic system
5. The FILE program

I. **Main Control**

When LSCN is read into memory, a word in the communication region, CRDLSC, contains a define-drum summary order. This summary order is issued so that the phases of LSC may be read in relative to the definition contained in CRDLSC. LSCN, itself, may be on any other drum, or it may be loaded from tape. When the phases are read by LSCN, they are assumed to be on drum according to the method of loading defined by the loading program LSCLDR (i.e., each segment of a phase occupies an integral number of consecutive sectors, with the \((n + 1)\)st segment beginning immediately after the \(n\)th segment).

When each phase is loaded into memory, control is transferred to it via:

```
TB    A1    Phase n
```

When each phase receives control, it displays in the 5-digit display register: ++n++ and sets up the diagnostic control in CRCRUD (see below - LSCN Diagnostic). Prior to the exit, the phase will display in the 5DD: --(n + 1) --. Upon returning from Phase IX, LSCN will return control to the master operating routine.

II. **The Communication Region**

The Communication Region in LSC has various functions:

1. Permits altering LSC parameters to produce a different object code.
2. Allows communication from one phase of LSC to another.
The Communication Region is located 762 words from the start of LSC. The SAL parallel code edit of LSCN contains a detailed description of every item in this region; hence, the entire list is not repeated here. Throughout this report, however, various quantities in the Communication Region are mentioned and described; each symbol in the Communication Region begins with "CR".

III. The Sorting System

Contained in LSC is a sorting system which is entered via:

\[
\begin{align*}
TB & \quad 1 & \quad \text{PRESRT} \\
SK & \quad K & \quad \text{Fn}
\end{align*}
\]

Where \( K \) represents the length of the sort key in number of 2-digit pairs and \( Fn \) is the file to be sorted. The flow chart of this section is contained herein; however, it is worth pointing out a general characteristic of the system. First, every file that is to be sorted has been previously "Closed" or "Released" via the File program. A test is made in PRESRT to see if the entire file is in memory (i.e., entirely contained in its double buffer). If it is, then only an internal sort of the file is made and left in the buffer. If the file is on drum, it may still be small enough for only a simple high-speed internal sort. If it is less than 12,500 words, then the entire file is read in, sorted, and written back on drum. If the file is greater than 12,500 words, then a more complicated and more time-consuming process takes place. This is described in the Sort Merge flow charts. However, in every case, the same internal sort program is used.

The internal sort is a left-to-right radix sort where the base is 100; i.e., two digits constitutes one character.

IV. LSCN Diagnostic System

Built into the control of LSC is a diagnostic system designed to simplify the obtaining of pertinent information should a difficulty arise while performing a compilation. This system includes A-register dumps, selective memory dumps, and dumps of sorted and unsorted files. LSC will produce the output on-line or on a tape for off-line printing according to the setting of CROUT: (See Communication Region).
There are two methods of taking action depending on whether:

1. LSC runs through to completion with incorrect results or
2. LSC fails to go to completion.

Case I
When LSC runs through completion with erroneous results, the files may be output to give all the necessary information. LSC will output all necessary files by setting:

\[ \text{CRFDMP} = 0 \]

CRFDMP is located at LSC + 768. If it is suspected that files produced by Phases M to N are in error, then only those files may be output by setting

\[ \text{CRFDMP} = 000 000 000 0 N M \]

Case II
If LSC fails to complete a compilation, the operator should cause the computer to transfer control to:

\[ \text{CRCRUD} = \text{LSC} + 794 \]

This will cause a selected dump of memory to be output according to the phase currently in operation as well as all important files. If LSC is in an infinite loop, the operator should first bring the computer to a stop and force an A-register dump before transferring control to LSC + 794.
LARC SCIENTIFIC COMPILER
SORT and PRINT FILES

PRESRT

Start
No registers are saved.

Save return, SEXIT

A:0

CRFDMPI phno (CRPH) ≥ CRFDMPI CRPH

Set up Presort call seq. for File, Fn.

Calling Sequence
a. T8 1 PRESRT
   x+1 op A Fn
   d+2 return

PRESRT1

CNMOVE

File is on drum.

op: SK

CRJUIT = 0

or tape no. is
use for off line printing.

SAVEMM

Save on CRDRE:
MIN(12500, NWFL + 2499) of memory.

CNMOVE

NWFL ≤ 12500

READFL

Read entire file in memory.

A:0

No sort, try print
one line.

CRFDMPI

Is file on drum?

Yes

CNMOVE

A:0

No

Sort file in buffer.
(no drum activity)

NOPRFL

CNPRFL

A+2

PRESRT2

CRFDMPI

NPFL < 12500

A:0 ≠

A:0

GPR

ENPFL

op: SK

d+2

Sort entire file in memory.

Write entire file on drum.

Restore memory.
LARC SCIENTIFIC COMPILER
SORT AND PRINT FILE.

PGSR2

File has more than 12500 words

A:0

Set up 4 temporary drum locations for sort merge. FLT8 27 indicates the proper drums to use.

MERGE

SORT MERGE routine.

Is sorted file on the original drum?

No

MOVE file to original drum.

Yes

GPR

PGSR3

CRFDMPCRPH

Read 12500 words of file for print

CRFDMPCRPH

CNPRFL

CNPRFL

CNPRFL

CRFDMPCRPH

Return

Return

Print file on line.

Print file on tape no. in CRBT

CRBT: 0

Print file on line.
LARC SCIENTIFIC COMPILER

INTERNAL SORT

Sort M items of length L on N digits starting at K.

1. \( L \rightarrow m \) (I1)
2. \( L \rightarrow n \) (BL)
3. \( K \rightarrow (B), (E) \)
4. \( O \rightarrow k \)
5. \( N-12 \rightarrow N \)
6. \( (B)+1 \rightarrow (B), (E) \)
7. \( (BL)-1 \rightarrow (BL) \)
8. \( \lfloor N/2 \rfloor \rightarrow i \)
9. \( L \rightarrow r \)
10. Set VO and V2 to select 1st half of \( (i+1) \)th digit.
11. \( i-1 \rightarrow i \)
12. \( j \rightarrow k \)
13. \( (T) \rightarrow h \)
14. \( (R^1h) + 1 \rightarrow (R^1h) \)
15. \( T+L \rightarrow T \)
16. \( j-1 \rightarrow j \)
17. \( k \rightarrow B1 \)
18. Set V1 to select 6th digit.
19. \( \rightarrow i \)
20. \( h+1 \rightarrow h \)
21. \( (R^1h) \rightarrow Y \)
22. \( \beta \rightarrow (R^1h) \)
23. \( O \rightarrow (R^1h) \)
24. \( \beta + Y \rightarrow beta \)
25. \( n-1 \rightarrow h \)
26. \( i \rightarrow j \)
27. \( A \)
28. \( V1 \rightarrow i \)
29. \( V1 \rightarrow j \)
Sort $m$ items of length $L$ on $N$ digits starting at $k$.
LARC SCIENTIFIC COMPILER.

FLOWCHART DEFINITIONS FOR PRESORT.

$L_0$ = Location of 1st. buffer.
$M_0$ = Location of 5th buffer = limit of string buffer.
k = Buffer length (in words).
f = Remaining length of file (initially length of file) (in words).
t = Item length (in words).
$L$ = Next location in 1st 4 buffers (for constructing string).
$M$ = Next location in 5th. buffer (for overflow items).
i = Output drum (3 or 4 in Presort).
$M_1$ = Limit of overflow buffer.

A REGISTER ASSIGNMENT.

<table>
<thead>
<tr>
<th>Presort</th>
<th>Merge</th>
<th>I/Ø</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>$f$</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>$k$</td>
<td>&quot;9...9&quot;</td>
</tr>
<tr>
<td>18</td>
<td>$M_1$</td>
<td>$K(A_i)$</td>
</tr>
<tr>
<td>19</td>
<td>$M$</td>
<td>$K'(A_i)$</td>
</tr>
<tr>
<td>20</td>
<td>$M_0$</td>
<td>$K'(B_j)$</td>
</tr>
<tr>
<td>21</td>
<td>$L$</td>
<td>$i$</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*note, 22 and 23 are saved by the I/Ø program.
MERGE

\[ k = 2k \text{ if item length} = 1 \]
\[ = 2.2k \text{ otherwise} \]

File length (words) \( \rightarrow f \)

\[ L_0 + 4k \rightarrow M_0 \]
\[ M_0 + k \rightarrow M_1 \]
\[ M_0 \rightarrow M \]

\( A \rightarrow i \) (output drum)

Initialize all other quantities depending on item size, drum number \& \( i \).
Reposition all heads (CREW)
LARC SCIENTIFIC COMPILER

MERGE

ML8K8

Yes

End of C block?

End of B block?

No

k+1 -> k

ML8KC

Yes

End of C block?

k+1 -> k

No

End of B block?

MLS

Yes

End of string?

Reset end data flag.

No

i+1 -> i

ML8KA

Yes

Get K(Ai)

Get K(Bj)

End of A block?

End of B block?

Get K(Ai)

Get K(Bj)

Get K(Ai)

Get K(Bj)

j+1 -> j

ML8KA

Yes

End of string?

Reset end data flag.

No

i+1 -> i

ML8KC

Yes

End of C block?

mLCS

Yes

Move Bj to Ck

j+1 -> j

No

End of A block?

Yes

Swap Input & Output drums.
Read 1st A block
Read 1st B block
Read 2nd A block
Read 2nd B block
Get K(Ai)
Get K(Bj)
MLBKA

Swap A buffers, 0 → i
Read next A block.

MLBKB

Swap B buffers 0 → j
Read next B block

MLBKC

Swap C buffers 0 → k
Write last C block

---

MLS

End of data string?

Yes

MLP END

No

R → k'
Swap output and alt. buffers.
Swap output drums.

---

1st string filed this pass?

Yes

No

O → MPF

---

MLP END

End string mark → Cr
R + 1 → k
Write last block (MLBKC)

---

End of MERGE?

Yes

CREW

NO

--

EXIT MERGE

Swap drums (MLS)
End string mark → Cr
k + 1 → k
Write last block on alternate (MLBKS)
Rewind drums.
Get end pass flag.
Reset end pass flag

Return.
LARC SCIENTIFIC COMPILER

I/O ROUTINES for SORT-MERGE

Read into (or write from) loc. F, S sectors.

CREAD

Set read flag.

SAVE 22, 23

CWRITE

Set write flag.

SAVE 26
Wait for drum d
(EPZ word)
Reset EPZ word
Td + P → Td
Transfer P sectors
starting at Td - P
into F. Restore 26.

CINOUT

Reposition all heads
0 → Ti
25 → Ri
(i = 0, ..., 5)

CREW

S: Rd

Rd: O

F + 100 Rd → F
S - Rd → S

Skipped and

25 → Rd

0 → Td

S → P

Rd → P

GINOUT

Rd = remaining sectors current band drum d
Td = next sectors this band drum d
Rd = A field of (MTS + d)
Td = φ P "..."
FILE

FILE is an LSC subroutine which stores items in files (writes) and retrieves items from files (reads). It is used by the main control program and by each phase of the compiler.

Different types of entrances to FILE specify opening, extending, marking, backing up, closing and releasing files, as well as reading and writing items singly or in groups. Description of these operations will be given later in this section.

For each file, the FILE subroutine keeps track of where in a buffer the next item is to be stored (if writing) or fetched from (if reading). In general, the buffers are intermediate storage areas which hold a block of items prior to and during writing on a drum or during and following reading from a drum. If short enough, however, files may be allowed to remain in their buffers without ever being written on a drum.

Each buffer is actually a double buffer. During writing, for example, items are collected in buffer 1, while the contents of buffer 2 are being written on a drum. When buffer 1 becomes full, the roles of buffers 1 and 2 are interchanged.

Information about each of the files is stored in a set of tables. A description of some of these tables follows.

Table 1 (origin, FLTB1) contains three consecutive words for each file:

```
OTIIIIDDBBPP
OOOOOOWWWWWW
OOOOOOZZZZZZZ or OMMMMMZZZZZZZ
```

T = relative table 2 origin for this file
I = number of words per item
D = logical drum number
B = band number of file origin
\[ P = \text{last permissible band number} \]
\[ W = \text{item counter} \]
\[ Z = \text{number of items in file} \]
\[ M = \text{location of file in memory (zero if file is on drum)} \]

The actual drum number is obtained by adding \( D \) to the contents of CRDRF (a cell in the communication region). Exceptions are files 1 and 10, whose actual drum numbers are found in CRDRI and CRDRO, respectively. The starting band and sector numbers of files 1 and 10 are likewise found in CRDRI and CRDRO. The format of each of these cells is:

\[
\text{BB Sector Band Drum}
\]

Word 3 of table 1 is used as an item counter during the writing of the file, and has the form indicated above only after the file has been closed or released.
Table 2 (origin of file F information: FLTB2+F) has the format

\[ \text{FORM 2235 PH R V V V M1} \]

\( \text{PH} = \text{phase number} \)

\( R = \text{relative locator for tables in phase PH} \)

\[ V_{11} = L1 = \text{capacity of buffer 1 in phase PH} \]

\[ V_{12} = V_{00} - V_{00} = L2 = \text{capacity of buffer 2 in phase PH} \]

\[ M1 = \text{origin of buffer 1 in phase PH} \]

\[ M1 + V_{11} = M2 = \text{origin of buffer 2 in phase PH} \]

The tables with which R is concerned contain origins of special sequences for handling writing or reading of items or groups of items, and locations of instructions in these sequences which must have their addresses preset. These tables are numbers 5, 6, 7, 9, 10, 12, and 14.

If I, the item length, is not a factor of 100, space for I-1 additional words must be provided in front of buffer 1 when reading, and space for I-1 additional words must be provided following buffer 2 when writing.

Table 4 is a storage area reserved for index words, one for each file. The index word format is given here to indicate the notation used:

\[ \text{BB NNN I Delta} \]

Table 8 consists of three words per file. These are initially zeros, and are replaced by drum summary orders when required. The location of the second of these three words is contained in the third word, which is an EPZ. Because of this, an absolute zero replaces the second word when the drum operation is completed.

Table 15 is a storage area reserved for the storage of buffer counters, one for each file. These have the format \( \text{BB 0 C 0} \), where \( C = 0, 2, 4, \text{etc}. \) C is increased by
2 each time a drum read or write operation for the associated file is initiated. The buffer counters are used only by the marking and backing up operations of phase 1. To form the mark word, the buffer counter is combined with the variable portions of the index word. The format of the mark word is:

\[ \text{BB NNN C Delta} \]

Another storage area, FLZ, contains transient information about a file. The contents of this area are as follows (cells not shown have general or temporary uses):

<table>
<thead>
<tr>
<th>Cell</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLZ + 0</td>
<td>file number</td>
</tr>
<tr>
<td>+ 1</td>
<td>3 X file number</td>
</tr>
<tr>
<td>+ 2</td>
<td>relative location of table 2 information</td>
</tr>
<tr>
<td>+ 3</td>
<td>I</td>
</tr>
<tr>
<td>+ 4</td>
<td>physical drum number</td>
</tr>
<tr>
<td>+ 5</td>
<td>starting band number</td>
</tr>
<tr>
<td>+ 6</td>
<td>P</td>
</tr>
<tr>
<td>+ 7</td>
<td>R</td>
</tr>
<tr>
<td>+ 8</td>
<td>M1</td>
</tr>
<tr>
<td>+ 9</td>
<td>M2</td>
</tr>
<tr>
<td>+ 10</td>
<td>L1</td>
</tr>
<tr>
<td>+ 11</td>
<td>L2</td>
</tr>
<tr>
<td>+ 12</td>
<td>L (from calling sequence)</td>
</tr>
<tr>
<td>+ 13</td>
<td>starting sector number</td>
</tr>
<tr>
<td>+ 17</td>
<td>current band number</td>
</tr>
<tr>
<td>+ 24</td>
<td>L2</td>
</tr>
<tr>
<td>+ 25</td>
<td>L1</td>
</tr>
<tr>
<td>+ 27</td>
<td>L1 + L2</td>
</tr>
<tr>
<td>+ 30</td>
<td>IX 100000</td>
</tr>
</tbody>
</table>
FILE has four calling sequences. Two of these (numbers 2 and 4) are high-speed entries to special instruction sequences within FILE which handle the two basic operations - reading or writing a single item.

Calling sequence number 1:

```
TB  1  FILE
XX  F  L
```

XX = SK: open file F for writing

XX = AX: mark the current positions in phase 1 output files 2, 3, 4, 91, 93, store mark words in L on

XX = A: store in file F the n items starting in L (n must be given in register 2)

XX = AM: back up to positions marked in phase 1 output files 2, 3, 4, 91, 93 (the mark words must be supplied in L on)

XX = AU: close output file F

XX = AAX: extend output file F

XX = AA: release output file F

XX = NX: open file F for reading forward

XX = N: open file F for reading backward, starting at the end of the file (items not reversed)

XX = NNX: mark current position input file 1, store mark word in L
XX = NN: back up to the item preceding the position marked in input file 1 (the mark word must be supplied in L)

XX = MXR: read the next line (10 words) from input file 1, store in L on.

If part of a file is on a drum and part in a buffer, closing or releasing the file causes the contents of the buffer to be written onto the drum. If the file is entirely in memory, closing the file leaves the file in memory, and releasing the file causes it to be written onto a drum. Extending a file allows more items to be added to a file after it has been closed or released.

Calling sequence number 2:

\[
\begin{array}{c}
TB \ 1 \ FILE + F \\
H \ O \ L
\end{array}
\]

This causes one item, whose first word is in L, to be stored in file F.

Calling sequence number 3:

\[
\begin{array}{c}
TB \ 1 \ RFILE \\
XX \ F \ L \\
\text{mask (or 0)}
\end{array}
\]

XX = MXE: read the next n items from file F, store in L on (n must be given in register 2)

XX = MR: read the group of items (defined by the mask) from the file F, store in L on, supply index word in register 2

XX = M: read the next item from file F without counting it, store in L on.

The mask is a word of 1's and 0's (e.g., 111100001111) and is applied in MR entries
to the first word of each item to determine a group. All consecutive items in a file which have their 1-corresponding first word digits respectively equal constitute a group (with respect to the mask).

Upon return from an MR entry, register 2 contains an index word:

```
BB  NNN  I  L
```

where NNN is the number of items in the group.

After an M entry has been used to read the nth item, the next reading of the same file will then again bring in the nth item, rather than the (n + 1)st item as in the normal case.

The normal return from the FILE subroutine following an entry via calling sequence number 3 or 4 is to the second instruction beyond the mask or zero word. The end-of-file return is to the first instruction following the mask or zero word. The end-of-file return is made following every entry after the end of file has been reached, unless an intervening NX or N entry has been made.

Upon a normal or end-of-file return following an MXE entry, the actual number of items read appears in register 2. An end-of-file return following an MR entry indicates that all the items in the file have been read previous to this entry.

All references to files are made symbolically. For example, F3 denotes file 3. The correspondences between symbol and absolute designations are given in the following table:

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>3</td>
</tr>
<tr>
<td>F2</td>
<td>4</td>
</tr>
<tr>
<td>F3</td>
<td>5</td>
</tr>
</tbody>
</table>
F4
F5 (or F50)
F51
F52
F53
F54
F55
F56
F6 (or F60)
F61
F7 (or F70)
F8 (or F80)
F81
F82
F83
F84
F85
F9 (or F90)
F91
F92
F93
F94
F10
FILE ROUTINE (details in Parallel Code Edit)

FILE

Calling Sequence Number 1.

XX = 00
XX = 00

Open File F for writing

SR routine.

Generate special information for File F in FLE from FLT131, FLT82
1. F, 3F
2. Drum location
3. Buffer size
4. Buffer location
(See storage for FLE)

Return

XX = 01

Mark the current positions in all Phase I output files and store information in FLE on. (This information will be used if these files are to be erased back to these marks by "Back up"—see AM)

XX = 02

File n items for File F.

(File must be opened for writing) Use the move sequence to file 1 item for File F n times.
See:
(TB 1 FILE + "F")

XX = 00

1. 0 ← FLT81 + 3*F+1
0 ← FLT81 + 3*F+2
2. Set up index word to load buffer
3. Set entry to file one item for File F.
FILE ROUTINE (continued)

**FLAM**

\[ XX = 03 \]

Back all Phase I output files to marks stored in \( L \) on. If the output on any file is on drum then, the drum is read back, index words and counts are adjusted as filing may take place from last mark on.

\[ d+2 \]

**FLAX**

\[ XX = 05 \]

Extend output File F for more writing. (The output file has been previously closed.)

**FLAY**

\[ XX = 09 \]

Close output File F.

**FLSR**

Is entire file in memory?

- Yes
  - Leave file in memory (No drum write)
    - \[ FLB1 + 3 + F + 2 = 0 \]
    - \[ L = \text{location of first item} \]

- No

**FLAX**

\[ XX = 05 \]

Extend output File F for more writing. (The output file has been previously closed.)

**FLSR**

Is file on the drum?

- No
  - Read last buffer in and adjust counts and index words for further filing.

- Yes
  - Set counts and index words for filing.

\[ d+2 \]
FILE ROUTINE (continued)

FLNX

XX = 11

Open File F for reading items in a forward direction.

FLSR

F: F1

No

Is file on drum?

Yes

No

Read in first 2 buffers (Do not delay)

Set index words and counts for reading. Set entry to read 1 item for file F in forward direction.

α+2

Reading File 1, the source program

CRT: O ≠

The source program is on the drum

Read source program from tape and drum on drum (drum number is in CRBRI)

FLN

XX = 12

Open File F is read in backward direction (this is used only by Phase VIII) Operations are same as for XX = 11

α+2

FLNX

XX = 15

Mark input file (F1) at current position for a possible back-up (see: FLNN) Store mark in L

α+2

FLNN

XX = 16

Adjust buffers and index words for F1 to back up the input file is the mark stored in L

α+2
FILE ROUTINE (continued)

FLMXR

XX = 20

Special entry to read 10 words (1 source line) from file, F1.
Store in L on.
There is no end of file exit here. Phase I will continue to read lines until the
"END statement is encountered.

α+2
FILE ROUTINE (continued)

Calling Sequence Number 2:
\( \text{D TO I FILE+F} \)
\( \text{H O L} \)

\( \text{FILE+F} \)

FLW L

The transfer from this point was set up by:
OPEN FILE (SK) or by
EXTEND FILE (AX)

Move \( n \) words (item) from L to buffer for File F

Is buffer full?

Yes

Initiate writing of full buffer on drum.
Set up index word to use other buffer.

No

\( \text{D+2} \)

\( \text{Delay} \)

Has other buffer been completely written?

Yes

\( \text{No} \)
FILE ROUTINE (continued)

RFILC

Calling Sequence
Number 3.
 freelTB 1 RFILC
x+1 Y Y F L
x+2 mask (or zero)
exit+3 possible End of File.
return

FLMXXE

YY = 21

Read the next n items from
File F, using the read 1
item sequence for File F
(see : RFILC+F)

YY = 21

Exit to x+3 if End of
File was encountered
during reading.

x+4

FLMRC

YY = 22

Read the next "group" of items
from File F.
A group of items are those
consecutive items that have
the same digits in the
positions defined by the
mask in x+2
(see : RFILC+F for buffering)

x+4

FLM

YY = 23

End of
File?

Yes

End of
File?

No

x+3

Read the next item from
File F into L on without
advancing buffer locations
or counts. (Allows peaking
at next item and reading
it later.

x+4
Calling Sequence
Number 4,
\(d\) TB I RFILE+ F
\(d+1\) H O L
\(d+2\) 0
\(d+3\) END OF FILE
\(d+4\) normal return

RF ILE + F

END OF FILE? Yes \(d+3\)

FLR i
The transfer to this point was set up by OPEN FILE for READ \((XX=N\ or\ NX)\)

Is input buffer empty? Yes

Move n words \((1\ item)\) from buffer to L on.

No

Set index words to read from next buffer.
Initiate read into emptied buffer.

Is reading complete for next buffer? Yes

Delay.

No
INPUT-OUTPUT ROUTINES

A typical calling sequence to an input-output routine is:

```
TB    # 0     9RIPT
SK     0     (0 . 2 (2) )
SK     0     10L  0  9
```

The second word contains the location of the tape number (the tape number is in floating point), and the third word contains the format location. If the format is assigned, the third word becomes

```
SK     0     (10L)  0  6
```

Immediately following the calling sequence are the list instructions. A typical list for an input routine is:

```
A , B , (C(I), I = 1,10)
```

which might be represented by the following instructions:

```
S    1     A     0     5
SS   2     B     0     5
F    #13   (BB   10    1    1)
S    1     C     #13   5
BIT   #13   HERE-1
T      9FIN
```

Here B is double precision, the other numbers single precision.

The same list for an output routine would have the S and SS instructions replaced by F and FF instructions.
Before referring to the list, the input-output output routine causes the tracing mode to be entered. This is done in the subroutine 9LEV.

Reference is made to the list simply by transferring to its first instruction (by way of the subroutine 9NTR, which transfers to 2600, in which 9LEV has stored a transfer to the list origin). When an instruction with a tracing mode selector of 5 is reached, control returns to the input-output routine by way of location 2601 (in which 9LEV has stored a transfer to the subroutine 9XCH). A transfer to the next list instruction is also automatically stored in 2600.

9XCH, after ascertaining that the tracing of an instruction did indeed just occur (rather than the occurrence of an error, which would also cause a transfer to 2601), forms an untraced version of the traced instruction, places it in the cell 9XCHX, and executes it.

For example, the instruction:

\[
\begin{array}{cccc}
S & 1 & A & 0 & 5
\end{array}
\]

would become

\[
\begin{array}{c}
S & \#1 & A
\end{array}
\]

Prior to the execution of this instruction (in fact, just before transferring to 9NTR), an input routine would place the current word from the input buffer in \#1 (or \#1 and \#2, if double-precision transmission is a possibility). An output routine would store the contents of \#1 (and perhaps \#2) in a buffer after the execution of a fetch in 9XCHX.

If a format is involved in the input-output operation, the scanning of the alphanumeric format is interpolated in the process described above. The subroutines WTG, GN, and GXB are used for this purpose.

Two 20-word buffers are used for alphanumeric tape input (READ; READ INPUT TAPE). While a tape block is being processed in one buffer, the next block is being read into the other buffer. However, if the program reads more than one input tape, the second one referred to must operate with a single buffer.
Two 20-word buffers are also used for alphanumeric tape output (PRINT; PUNCH;
WRITE OUTPUT TAPE). A block is prepared in the first buffer and then moved
to the second buffer for writing on tape, during which the preparation of the next
block in the first buffer can proceed.

The remaining input-output operations use single buffers (with the exception of
REWIND, REWIND INTERLOCK, and alphanumeric END FILE, which do not
require buffers). WRITE DRUM and READ DRUM share a single 100-word buffer.
WRITE TAPE, END FILE (decimal mode), and READ TAPE also share a single
buffer whose origin (9STB) and length (9TBL) are given in the main program.
(9TBL is a constant supplied from the communication region of LSC via CRTREC.)

More details on the above program are found in the attached flow charts and the
code listings of these programs.
WRITE DRUM

9 WDRM

Enter 9DRD subroutine.

Sector no. → read summary order. Issue summary orders (position head, read), delay until done (9DRS subroutine). Replace position head by write summary order.

Enter 9LEV subroutine. (See Read Input Tape.) Set final return from list in 9FIN+1

Sector no. → write summary order. Advance sector no.

Sector no. = 21?

Yes → 48 → buffer length

No

100 → buffer length.

0 → section number.

A

B

O → J

Return

Get initial location.

Find, store sector number.

Band number → summary orders.

LARC drum number → summary orders.

9 DRD
LARC SCIENTIFIC COMPILER

FINAL RETURN FROM LIST

9 WDR 9

Yes Location = 0 ?

No NZ → J

Replace read summary order by skip.

Enter QRTN subroutine (see Read Input Tape)

Return to Main Program

Sector no. → read summary order.

Enter list, fetch next word (QRTN subroutine). Store word in buffer. Advance location by precision number.

Beyond end of buffer ?

No Yes

Issue summary orders (QDRS)

J = 0 ?

No Yes

Decrease location by buffer length.

Store word following last word of buffer in first word location.

8 WDR 1
WRITE OUTPUT TAPE, PRINT, PUNCH

9 XCH

Tracing mode entered? [No.]

Reset tracing mode FF20

Tracing mode FF25 set?

No

Restore previous contents of 2601

To 2601

Precision no. to #0 Modify traced instruction store in 9xCHX, execute it.

Return (set up by 9mpt)

Yes

9 WOPT

Enter 9DCode subroutine.
Set final return from list in 9FIN+1
Fix WTC, XISR for output

Enter 9LEV subroutine (see Read Input Tape)

Fill buffer with blanks. (9FIL)
Set up buffer index words. Relinquish control to WTC.

Return from XSSL or XISR 9 WOPT4

FINAL RETURN FROM LIST

9 MPT

Delay until last write operation is done.
Move buffer contents output location (9mou)

Issue summary orders to write output buffer onto tape.
Set flag indicating buffer is empty.

Return.

9 WOPT5

Buffer empty?

Yes

Enter 9MPT subroutine.

Enter 9RTM subroutine (see Read Input Tape)

No

Return to Main Program.
LARC SCIENTIFIC COMPILER

READ INPUT TAPE, READ.

9RIPT
Enter 9dcopy subroutine.
Set final return from list in 9FIN+1.
Fix WTC, XISRP for input.

9RTN
Restore previous contents of 2601.
Restore previous FF25 status.

9LEV
Save contents of 2601.
Store (T 9XCH) in 2601.
Store (T list origin) in 2600.

9RIPT8
Enter 9RTN subroutine.

Return from 9ISSLP or 9ISRP
9RIPTA
First entry to 9RIPT?

Yes
Issue summary orders to read first tape blocks (RDFIL1) (RDFIL2).
Store tape no. in ISTAPE.

No
Delay until next block is in current buffer.

Yes
9STAP = NUTAPE?

No
Set up buffer index words.
Relinquish control to WTC.

Yes
Delay until next block is in read-ahead buffer.

Move contents of read-ahead buffer to current buffer (CMOV).

Issue summary orders to read next tape block into current buffer.
(RDFIL1)

Issue summary orders to read next tape block into read-ahead buffer.
(RDFIL2)
Flow chart for XISI is the same as for XISP, except:
1. No scaling is done.
2. The exponent is assumed to be > 50 for non-zero numbers.
LARC SCIENTIFIC COMPILER

Routines used by 9RIPT & 9WOPT (cont)

CNV 3

0 → CNTR, DECP, DGCNT, LDDG

ω = 0?
Yes
No
ω - 1 → ω

Get next digit or character (GXB)

Digit?
Yes
No
See chart

NZ → DGCNT

Form no. by shifting in digit from right.

Is partially formed no. 0?
Yes
No

Is no. > 999999999999?
Yes
No

Shift out last digit

Add 1 to CNTR

To C33

Consolidate no. (from CNNTN) and exponent with signs, form D.P. floating pt. no. (add LDDG & exponent; normalize)

Exp < 0?
Yes
No

Absolute zero → no.

Exp > 99?
Yes
No

.99...99 → no.

Return.
<table>
<thead>
<tr>
<th>CHAR</th>
<th>DECPT</th>
<th>O</th>
<th>37</th>
<th>&gt; 37</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank</td>
<td>If CNTR = 0, → C33</td>
<td></td>
<td>C39</td>
<td>Treat blank as a zero; → C33</td>
</tr>
<tr>
<td></td>
<td>If CNTR ≠ 0, → C39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Make exp. -; → C69</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>If DECNT = 0, make sign of no. -; → C33</td>
<td></td>
<td>C69</td>
<td>Store no. in CNVTN; make DECPT &gt; 37; → C33</td>
</tr>
<tr>
<td></td>
<td>If DECNT ≠ 0, make exp. -; → C66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>If DECNT = 0, → C33</td>
<td></td>
<td>C69</td>
<td>Store no. in CNVTN; make DECPT &gt; 37; → C33</td>
</tr>
<tr>
<td></td>
<td>If DECNT ≠ 0, → C66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>37 → DECPT; C(CNTR) → LDDG; → C33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>C66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CNTR - TRDG → LDDG (if neg., set = 0); → C69</td>
<td></td>
<td>C69</td>
<td>→ C33</td>
</tr>
</tbody>
</table>
LARC SCIENTIFIC COMPILER

WRITE TAPE

9 WRTP

Set final return from list in 9 FIN+1

Enter 9LEV subroutine (see Read Input Tape)

O → block no., loc., W

Delay until buffer available (9WRTS)
Tape address → summary orders (9WRTS)

Store mode (9 for numeric) in tape mode flag (9ST9)

F

Delay until buffer available (9WRTD)

Clear buffer to zeros

Word following buffer → 2nd word of buffer.

Add 1 to block no.

E

Enter list via 9NTR, or execute list instr. Taken from 9XCHX.

Store 2 words in buffer. Advance location by precision no.

Beyond end of buffer?

No

loc = 0?

Yes

Decr. location by 9TBL-1 (no. data words in buffer)

Yes

More in list? (Enter list via 9NTR) (if not, go to 9WRTPB via 9FIN)

loc = 0?

Yes

F

No

U = D?

Enter 9RTN (see Read Input Tape)

Before main program

Final return from list.

9 WRTPB

NZ → W

loc = 0?

Yes

No.

loc. → Q

9TBL-1 → Q

Q, block no. → first word.
PHASE I

The main function of Phase I is to analyze the initial source program and to output standard units of information (items), describing the input. These items are listed below with details of their construction. The source program is read from the input drum via the FILE routine. If the word in the Communication region indicates, the input line is converted to the LSC internal character code prior to analyzing the statement. SAL code is not analyzed except for recording the label of a SAL instruction.

The analysis of the source statement is sufficient for Phase I to output an item for every meaningful feature of the input and such that no other phase need refer to the source input again. The items output by Phase I are placed in different files according to their type and the original source lines (in the LSC internal character code) are filed for editing when compiling is complete. The items generated are filed by the FILE routine while the scan is in process. Hence, no restrictions are made regarding the number of lines (continuations) used to represent a single statement except that space on the drums must exist for the file being formed.

Phase I is broken down into many parts, some of which are described below; flowcharts of these sections are included. Some statements receive more thorough analysis in Phase I than others. In general, Phase I does not make an analysis of the program as a whole. In particular:

1. No distinction is made between subscripts and arguments.

2. The arithmetic mode implied by the name of a variable is ignored.

3. The appearance of an induction variable (explained later) in arithmetic expressions or subscripts is ignored.

4. Information contained in one expression or statement is not carried along when processing later statements, except in a DO statement (see 2. below).
The analysis of Phase I is best summarized by noting the list of items which it generates. The output of Phase I has been designed in such a manner as to facilitate the analysis of Phase II, III, and IV. On the other hand, the functions of Phases II, III, and IV make it unnecessary for Phase I to keep lengthy tables describing variables and searching for information about a variable before outputting items.

Some areas of Phase I deserve special attention:

1. Arithmetic Scan

During Phase I, an arithmetic expression is scanned from left to right and items are assigned a subsequence number in such a way as to cause the expression to appear in the Polish (Łukasiewicz) notation in later phases. The major sequence number (the lead five digits) is the same for all items in the expression. The included flowchart contains details on how the subsequence number (SS) is established.

2. Determination of a "DO range"

A DO statement is the only statement which causes Phase I to store interstatement information. Whenever a statement:

\[
\text{DO } \mu, \ I = b, c, d
\]

is encountered, the quantities: I, b, c, and d are output immediately. The statement sequence number, S, associated with these items is stored (hereafter referred to as DS) along with the referred to statement name, \( \mu \), and with I. The range is determined when a statement named \( \mu \) is encountered; at this time a second item for I is output (a "20" type item). Prior to the arrival of the statement \( \mu \) (or to another DO statement) DS, as described above, is associated with all items output by Phase I that have DS indicated in the item.
3. The occurrence of a "list" in the source program (input-output list or "list" = expression, etc.) causes Phase I, in some cases, to output items similar to those of a DO. For example:

\[
( ( ( A ( I, J, K), I = 1, 10), J = 1, 5), K = 1, 10) = expression
\]

will generate:

1. (Expression evaluation)
   DO β, K = 1, 10
   DO β, J = 1, 5
   DO β, I = 1, 10

4. Analyzing SAL statements:

   A. An "S" in the first column of a statement denotes SAL coding, all names interpreted by LSC must be legal LSC symbols.

   B. Output special continue item for all names of SAL code.

   C. Output a special continue item for the first statement in a block of SAL coding (ID = 11) to F4.

   \[
   \begin{align*}
   W1 & \quad \text{Seq.}, \quad O, \quad 11 \\
   W2 & \quad \quad \quad O \\
   W3 & \quad \text{DS}, \quad L, \quad P
   \end{align*}
   \]

   D. The induction variable in a DO loop is made available to SAL programs.
E. Any exits from a block of SAL coding to another block of SAL coding or to an LSC statement must be indicated in the following statement:

\[ \text{SAL EXITS (} N_1, N_2, \ldots, N_k \text{)} \]

where \( N_i \) is the symbol to which a transfer is made outside of the following block of SAL coding.

F. Phase I will recognize the LABEL line and a DIRECTORY; in general these items remain unprocessed and are assumed to be acceptable SAL items.
Sequence Counters

During Phase I, a set of sequence counters is maintained in order that every piece of information extracted from the source program (an item) can have a number associated with it. The sequence counters (ST, SD, SE, etc.) each have a range of numbers exclusive of the others. The number itself will be an integer less than $10^5$.

1. ST is the sequence counter for the following statements:

   INTEGER
   FLOATING
   DOUBLE PRECISION
   INTEGER FUNCTION
   FLOATING FUNCTION
   DOUBLE PRECISION FUNCTION
   PARAMETER

2. SC, SF, and SA are used for:

   CONSTANT
   FORMAT
   Arithmetic functions, respectively

3. SD is the sequence counter for DIMENSION statement items.

Example:  
DIMENSION A (I, B), B(7)  
DIMENSION C (M)
Assuming that the current value of the DIMENSION counter is SD, the first statement, as a whole, has SD as the sequence number. (This is for printing purposes). Items A, I and B have sequence number SD + 1. B and 7 have SD + 2. The second DIMENSION statement has sequence number SD + 3. Items C and M have sequence number SD + 4. The next DIMENSION statement encountered would have sequence number SD + 5, etc. Any error items get the same sequence number as the item which was in error.

4. SE is the sequence counter for EQUIVALENCE and COMMON statement items.

Example: EQUIVALENCE ( A (I), B(5) ), (C, D)
COMMON D, E, F

Assuming that the current value of the sequence counter is SE, the following assignments of sequence numbers will be made:

<table>
<thead>
<tr>
<th>Item</th>
<th>Sequence Number</th>
<th>Subsequence Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQUIVALENCE Statement</td>
<td>SE</td>
<td>0</td>
</tr>
<tr>
<td>A, I, B, 5</td>
<td>SE + 1</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>C, D</td>
<td>SE + 2</td>
<td>1, 2</td>
</tr>
<tr>
<td>COMMON Statement</td>
<td>SE + 3</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>SE + 4</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>SE + 5</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>SE + 6</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: Any error items get the same sequence number as the item which was in error.

5. S is the sequence counter for the items in the remainder of the statements.

In general, the entire sequencing number (10 digits) is made up of two parts, the Sequence number described above and a Subsequence number, which gives the order (or desired order) of the items within a group of related items. Variables in a list get numbered
sequentially. Variables in an expression all receive the same sequence number, but receive subsequence numbers in such a way as to transform the expression into a parenthesis free notation when the items are sorted on the sequencing number. Statements in the print file have sequence numbers which differ by at least 1. Error items always have a sequence number equal to the sequence number of the item which caused the error.

6. SB is used for the FUNCTION and SUBROUTINE statement.

Sequencing of Arithmetic Statements

S is the sequence counter used for arithmetic statements. Let n be the value of the sequence counter S associated with an arithmetic statement:

<table>
<thead>
<tr>
<th>S</th>
<th>Statement Name</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>JOE</td>
<td>X(l), Y = A + B</td>
</tr>
</tbody>
</table>

Sequence and subsequence numbers are assigned in the following manner:

<table>
<thead>
<tr>
<th>Item</th>
<th>Sequence S</th>
<th>Subsequence SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>line</td>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>JOE</td>
<td>n + 1</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>n + 2</td>
<td>0</td>
</tr>
<tr>
<td>* X</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>:</td>
<td>n + 3</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>* Y</td>
<td>n + 2</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>n + 4</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>A</td>
<td>n + 2</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

* Arithmetic MODE Item

Note that the sequencing numbers of the list items are greater than any of the expression items.
Sequencing of Arithmetic Functions

SA is the sequence counter used for arithmetic functions. Let n be the value of the sequence counter SA associated with the arithmetic function:

<table>
<thead>
<tr>
<th>SA</th>
<th>Statement Name</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>SAM</td>
<td>FIRSTF(A, B) = 2A + B**2</td>
</tr>
</tbody>
</table>

Sequence and subsequence numbers are assigned in the following manner:

<table>
<thead>
<tr>
<th>Item</th>
<th>Sequence</th>
<th>Subsequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>line</td>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>SAM</td>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>FIRST</td>
<td>n + 1</td>
<td>0</td>
</tr>
<tr>
<td>FIRST</td>
<td>n + 2</td>
<td>0 (expression mode item)</td>
</tr>
<tr>
<td>A</td>
<td>n + 1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>&quot;</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>n + 2</td>
<td>100</td>
</tr>
<tr>
<td>*</td>
<td>&quot;</td>
<td>99</td>
</tr>
<tr>
<td>A</td>
<td>&quot;</td>
<td>150</td>
</tr>
<tr>
<td>+</td>
<td>&quot;</td>
<td>98</td>
</tr>
<tr>
<td>B</td>
<td>&quot;</td>
<td>200</td>
</tr>
<tr>
<td>**</td>
<td>&quot;</td>
<td>199</td>
</tr>
<tr>
<td>2</td>
<td>&quot;</td>
<td>250</td>
</tr>
</tbody>
</table>

Note: The name "SAM" was used here to show that a continue item will be output for it with sequence number S (the executable statement counter). Normally a statement name is not given, since reference to the arithmetic function will be FIRST(x, y) or FIRSTF(x, y).
PHASE I  APPENDIX B

ITEM FORMATS

The following formats represent items which are output by Phase I, II, and III. Outputs from Phase II and III are included here for completeness. The File 2 items are on the left, File 3 in the center, and File 4 on the right side of each page.

PARAMETER \( V_1 = A_1 \), \( V_2 = A_2 \), \ldots, \( V_p = A_p \)

\[
\begin{array}{cc}
V_j & ST \\
ST & 0 \\
A_j & 01
\end{array}
\]

\( A_j \) is truncated to a floating point integer.

CONSTANT \( X = \text{numeric} \)

\[
\begin{array}{ccc}
SC & 1 & 45 \\
0b2 & 0000 & 0000
\end{array}
\]

high order part of numeric

\[
\begin{array}{ccc}
X & SC & 0 \\
SC & 38 & 0b1 \\
0000 & 0fff
\end{array}
\]

low order part of numeric

INTEGER \( V_1, \ldots, V_p \) \( (b = 1) \)

FLOATING \( V_1, \ldots, V_p \) \( (b = 2) \)

DOUBLE PRECISION \( V_1, \ldots, V_p \) \( (b = 3) \)

\[
\begin{array}{ccc}
V_j & ST \\
ST & 0 \\
0b1 & 0000
\end{array}
\]

00000
INTEGER FUNCTION $V_1, \ldots, V_p$ \hspace{1cm} (b = 1)

FLOATING FUNCTION $V_1, \ldots, V_p$ \hspace{1cm} (b = 2)

DOUBLE PRECISION FUNCTION $V_1, \ldots, V_p$ \hspace{1cm} (b = 3)

$V_j$

<table>
<thead>
<tr>
<th>ST</th>
<th>0</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b4</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

The mode digit "b" is assigned to all references to $V_j$ by Phase III when the reference items are sent to File 4.

SUBROUTINE $N(M_1, \ldots, M_p)$

<table>
<thead>
<tr>
<th>N</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>p</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SB</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>p</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>j</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>j</td>
<td>33</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SB</th>
<th>j</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION $N(M_1, \ldots, M_p)$

<table>
<thead>
<tr>
<th>N</th>
<th>0</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>p</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SB</th>
<th>0</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>p</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>0</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SB</th>
<th>0</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

9b c  0000  0fff
RETURN

\[
\begin{align*}
S & \quad 0 & 49 \\
6 & \quad 00 & 0014 & 00000 \\
0 & \underline{0} & \underline{0} & \underline{0}
\end{align*}
\]

ARITHMETIC STATEMENT FUNCTION \( N \ F ( \ M_1, \ldots, M_p ) = \ldots \)

\[
\begin{align*}
N & \quad 0 & 23 \\
SA & \quad 0 & 23 \\
0 & \underline{0} & \underline{p}
\end{align*}
\]

\[
\begin{align*}
SA & \quad 5 & 64 & 0000 & 0fff \\
0 & \underline{0} & \underline{p}
\end{align*}
\]

\( M_j \)

\[
\begin{align*}
SA & \quad j & 24 \\
0 & \underline{0}
\end{align*}
\]

(j) replaces the (uxffff) digits of references to \( (M_j) \) appearing on the right hand side of this statement. \( (SA +1.4) \) is the sequence of items in the arithmetic expression on the right.

CALL \( N ( \ M_1, \ldots, M_p ) \)

\[
\begin{align*}
N & \quad 1 & 30 \\
S & \quad 0 & b4 & 0000 & 0fff \\
DS & \quad L & p \\
\end{align*}
\]

\[
\begin{align*}
S & \quad 1 & 30 \\
DS & \quad L & p
\end{align*}
\]

\( (\text{LIB request item initiated by Phase III}). \)

\[
\begin{align*}
0 & \quad 5 & 44 \\
0 & \underline{0} & \underline{N}
\end{align*}
\]

If \( (M_j) \) is a single symbol:

\[
\begin{align*}
M_j & \quad ss & 31 \\
S & \quad abc & d & 000 & 0fff \\
DS & \quad L & p
\end{align*}
\]

\[
\begin{align*}
c=1 & \quad 0 & \underline{Dp} \\
c=2 & \quad aj \\
c=3 & \quad DS & L & p
\end{align*}
\]
If \((M_j)\) is a Hollerith argument:

\[
\begin{array}{c}
\text{w1} \quad \text{SF} \\
\text{w2} \quad \text{ALPH}
\end{array}
\]

characters in the Hollerith argument.

If \((M_j)\) is an arithmetic expression, output Polish string for arithmetic.

\[
\text{DIMENSION } V(A_1, \ldots, A_p), V^1(A_1^1, \ldots, A_p^1), \ldots
\]

\[
\begin{array}{c}
 V^j \\
\text{SD} \\
0
\end{array}
\]

\[
\begin{array}{c}
\text{SD} \\
0
\end{array}
\]

If a dimension is specified as a constant:

\[
\begin{array}{c}
\text{SC} \\
012
\end{array}
\]

\[
\begin{array}{c}
\text{A}_j
\end{array}
\]

If a dimension is specified as a parameter:

\[
\begin{array}{c}
\text{A}_j \\
\text{SD} \\
0
\end{array}
\]

\[
\begin{array}{c}
\text{SD} \\
0
\end{array}
\]

\[
\begin{array}{c}
\text{SE} \\
0
\end{array}
\]

\[
\begin{array}{c}
\text{SE} \\
0
\end{array}
\]

\[
\text{EQUIVALENCE } (V_1(A_1^1, \ldots, A_p^1), \ldots, V_n(A_n^1, \ldots, A_n^d)), \ldots
\]

\[
\begin{array}{c}
 V_j \\
\text{SE} \\
0
\end{array}
\]

\[
\begin{array}{c}
\text{SE} \\
i
\end{array}
\]

\[
\begin{array}{c}
\text{SE} \\
0
\end{array}
\]

\[
\begin{array}{c}
\text{SE} \\
0
\end{array}
\]

\[
\begin{array}{c}
\text{SE} \\
0
\end{array}
\]

\[a = 2 \text{ if not common} \]

\[a = 3 \text{ if common}\]
For each \( A_j^k \) specified as a constant:

\[
\begin{array}{crr}
\text{SE} & j & 45 \\
0 & 12 & 0000 & 00000 \\
A_j^k
\end{array}
\]

For each \( A_j^k \) specified as a parameter:

\[
\begin{array}{crr}
\text{SE} & j & 03 \\
0 & 12 & 0000 & 00000 \\
A_j^k
\end{array}
\]

COMMON \( V_1, \ldots, V_p \)

\[
\begin{array}{crr}
V_j & (SE+j) & 05 \\
0 & 0 & a b l & d e e & 0 f f f f \\
0 & 0 & 0 & 0 & 0
\end{array}
\]

\( a = 1 \) if not equivalent
\( a = 3 \) if equivalent

ARITHMETIC STATEMENTS

\( V \) in an expression

\[
\begin{array}{crr}
V & & \\
S & s s & 14 \\
0 & 0 & 0 & 0 & A A A & D P
\end{array}
\]

\( \text{AAA} = \) the subprogram argument number if \( V \)
\( \) is an argument of a subprogram that is
\( \) being compiled.

\begin{aligned}
\text{Store:} & \quad V = \\
\end{aligned}

\[
\begin{array}{crr}
V & & \\
S & 0 & 21 \\
0 & 0 & 0 & 0 & A A A & D P
\end{array}
\]

\[
\begin{array}{crr}
S & 0 & 21 \\
0 & 0 & 0 & 0 & A A A & D P
\end{array}
\]
Mode of expression item

\[
\begin{array}{ccc}
V & 0 & 39 \\
S & 0 & 39 \\
\end{array}
\begin{array}{ccc}
S & 0 & 39 \\
\text{abc} & \text{deee} & 0fff \\
0 & 0 & 0
\end{array}
\]

List Item

\[
\begin{array}{ccc}
S & 0 & 60 \\
0 & 0 & 26
\end{array}
\begin{array}{ccc}
S & 0 & 60 \\
\text{0000} & \text{AAA} & \text{DP}
\end{array}
\]

V in an input list

\[
\begin{array}{ccc}
V & 0 & 26 \\
S & 0 & 26 \\
\end{array}
\begin{array}{ccc}
S & 0 & 26 \\
\text{a b l} & \text{deee} & 0fff \\
0 & 0 & 0
\end{array}
\]

V in an output list

\[
\begin{array}{ccc}
V & 0 & 28 \\
S & 0 & 28 \\
\end{array}
\begin{array}{ccc}
S & 0 & 28 \\
\text{a b l} & \text{deee} & 0fff \\
0 & 0 & 0
\end{array}
\]

\[
\begin{array}{ccc}
S & 0 & 28 \\
\text{0000} & \text{AAA} & \text{DP}
\end{array}
\]

If V is a reference to the induction variable within the DO loop, Phase III changes the item to reflect this type of reference (e.g.)

\[
\begin{array}{ccc}
V & \text{ss} & 14 \\
S & \text{ss} & 14 \\
\end{array}
\begin{array}{ccc}
S & \text{ss} & 14 \\
\text{411} & \text{0000} & 0fff \\
0 & 0 & 0
\end{array}
\]

V (subscript) in an expression

\[
\begin{array}{ccc}
V & \text{ss} & 25 \\
S & \text{ss} & 25 \\
\end{array}
\begin{array}{ccc}
S & \text{ss} & 25 \\
\text{a b l} & \text{deee} & 0fff \\
0 & 0 & 0
\end{array}
\]

V (subscript) STORE: V ( ) =

\[
\begin{array}{ccc}
V & \text{02} & 22 \\
S & \text{02} & 22 \\
\end{array}
\begin{array}{ccc}
S & \text{02} & 22 \\
\text{a b l} & \text{deee} & 0fff \\
0 & 0 & 0
\end{array}
\]

\[
\begin{array}{ccc}
V & \text{02} & 22 \\
S & \text{02} & 22 \\
\end{array}
\begin{array}{ccc}
S & \text{02} & 22 \\
\text{0000} & \text{AAA} & \text{DP}
\end{array}
\]
Store expression item

\[ \begin{array}{ccc}
V & S & 0 \\
S & 0 & 39 \\
0 & 0 & 0
\end{array} \quad \begin{array}{ccc}
S & 0 & 39 \\
abl & deee & 0fff \\
0 & 0 & 0
\end{array} \]

V (subscript) in an input list

\[ \begin{array}{ccc}
V & S & 02 \\
S & 02 & 27 \\
0 & 0 & 0
\end{array} \quad \begin{array}{ccc}
S & 02 & 27 \\
abl & deee & 0fff \\
0 & 0 & DP
\end{array} \]

V (subscript) in an output list

\[ \begin{array}{ccc}
V & S & 02 \\
S & 02 & 29 \\
0 & 0 & 0
\end{array} \quad \begin{array}{ccc}
S & 02 & 29 \\
abl & deee & 0fff \\
0 & 0 & DP
\end{array} \]

\[ \text{DO L} \quad I = N_1 \text{, } N_2 \text{, } N_3 \\
( - - - \text{, } I = N_1^{1} \text{, } N_2^{1} \text{, } N_3^{1}) \]

(DO loop) (List loop)

implied name \( n \)

\[ \begin{array}{ccc}
I & S & 1 \\
S & 1 & 19 \\
DS & L & P \\
0 & S(e) & 0
\end{array} \quad \begin{array}{ccc}
S(e) & 0 & 20 \\
S(e) & 0 & 19 \\
DS & L & P \\
O & S(e) & 0
\end{array} \]

This item is output from Phase I when statement L is encountered. S(e) is greater than any value of S assigned to items in statement L.
If \( N_j \) is specified as a variable:

\[
\begin{array}{ccccccc}
N_j & \quad & S & j+1 & 46 & S & j+1 & 46 \\
S & j+1 & 46 & 411 & \quad & 0000 & \quad & 0ffff \\
DS & L & P & DS & L & P
\end{array}
\]

If \( N_j \) is specified as a constant:

\[
\begin{array}{ccccccc}
S & j+1 & 45 & 412 & \quad & 0000 & \quad & 0ffff \\
N_j
\end{array}
\]

CONTINUE – executable statement

\[
\begin{array}{ccccccc}
name & \quad & name & \quad & S & 0 & 08 \\
S & 0 & 08 & (S - 4) & S & 08 & 003 & 0000 & 0ffff \\
DS & L & P & DS & L & P & DS & L & P
\end{array}
\]

(if unreferenced do not send to File 4)

GO TO L

\[
\begin{array}{ccccccc}
L & \quad & S & 1 & 36 & S & 1 & 36 \\
S & 1 & 36 & 003 & d & 000 & \quad & 0ffff \\
DS & L & P & DS & L & P & S c o n t & 0
\end{array}
\]

(if not a drop-out)

\[
\begin{array}{ccccccc}
S & 1 & 36 & 003 & 0000 & 00000 & 0
\end{array}
\]

(if a drop-out)
GO TO \((N_1, \ldots, N_k)\), M \hspace{1cm} \text{(computed GO TO)}

\[
\begin{array}{cccc}
N_j & \quad S & \quad j+1 & \quad 16 \\
S & \quad j+1 & \quad 16 \\
DS & \quad L & \quad P \\
\end{array}
\]

\(S(c) = \text{sequence of statement name}\)

\[
\begin{array}{cccc}
M & \quad .1 & \quad 14 \\
S & \quad 003 & \quad 0000 & \quad 0fff \\
0000 & \quad AAA & \quad DP \\
\end{array}
\]

GO TO M, \((N_1, \ldots, N_k)\) \hspace{1cm} \text{(assigned GO TO)}

\[
\begin{array}{cccc}
M & \quad 1 & \quad 17 \\
S & \quad 003 & \quad 1000 & \quad 0fff \\
DS & \quad L & \quad P \\
\end{array}
\]

\[
\begin{array}{cccc}
N_j & \quad S & \quad j+1 & \quad 18 \\
S & \quad S(c) & \quad 18 \\
DS & \quad L & \quad P \\
\end{array}
\]

ASSIGN N, TO M

\[
\begin{array}{cccc}
N & \quad 1 & \quad 12 \\
S & \quad 003 & \quad 1000 & \quad 0fff \\
M & \quad M \\
\end{array}
\]

\[
\begin{array}{cccc}
M & \quad (S-2) & \quad S(c) & \quad 13 \\
DS & \quad L & \quad P \\
\end{array}
\]

\[
\begin{array}{cccc}
M & \quad 2 & \quad 13 \\
S & \quad abl & \quad 1000 & \quad 0fff \\
DS & \quad L & \quad P \\
\end{array}
\]
CONTINUE - (FORMAT statement)

name           name
S    0    10     (S - 3)    S    10
DS   L    P     DS   L    P

READ INPUT TAPE I, N, list

    READ INPUT TAPE I, N, list
    WRITE OUTPUT TAPE I, N, list
    READ PAPER TAPE, I, N, list
    PUNCH PAPER TAPE, I, N, list
    TYPEWRITE I, N, list
    READ TAPE I, list
    WRITE TAPE I, list
   REWIND I
   REWIND INTERLOCK I
BACKSPACE I
ENDFILE I
READ N, list
PRINT N, list
PUNCH N, list
READ DRUM I, J, list
WRITE DRUM I, J, list
DEFINE LARC DRUM I, J, K, L
POSITION HEAD I, J
ADVANCE HEAD I
BACK HEAD I
READ LARC DRUM I, J, list
WRITE LARC DRUM I, J, list
Reference to FORMAT "N"

\[
\begin{array}{ccc}
N & ss & 15 \\
S & ss & 15 \\
0 & & 0
\end{array}
\quad
\begin{array}{cccc}
S & ss & 15 \\
005 & d & 000 & 0xffff \\
0 & & 0
\end{array}
\]

For those values of \((I, J)\) specified symbolically:

\[
\begin{array}{ccc}
name & S & ss & 14 \\
S & ss & 14 \\
0 & & 0
\end{array}
\quad
\begin{array}{cccc}
S & ss & 14 \\
abl & 0 & 000 & 0xffff \\
0000 & AAA & DP
\end{array}
\]

CONTINUE - SAL label

\[
\begin{array}{ccc}
\text{name} & \text{name} \\
S & 0 & 11 \\
(S-4) & S & 11 \\
DS & L & P \\
DS & L & P
\end{array}
\]

Beginning of SAL block

\[
\begin{array}{ccc}
S & 0 & 11 \\
0 & & 0 \\
DS & L & P
\end{array}
\]

SAL EXITS \((V_1, \ldots, V_k)\)

\[
\begin{array}{ccc}
V_j & S & j & 35 \\
& DS & L & P
\end{array}
\]

IF \((-\ldots-\ldots-)\ N_1, N_2, N_3\)

If \((N_j)\) is missing

\[
\begin{array}{ccc}
S & j & 36 \\
003 & 0000 & 0000 \\
0 & & 0
\end{array}
\]
If \((N_j)\) is stated:

\[
\begin{array}{ccc}
N_j & S & j & 36 \\
& DS & L & P & S(c) \\
\end{array}
\]

(if not drop-out)

\[
\begin{array}{ccc}
S & j & 36 \\
& 003 & 0000 & 00000 \\
\end{array}
\]

Sense Light I

If (Sense Light I) \(N_1, N_2\)

If (Sense Switch I) \(N_1, N_2\)

If Divide Check \(N_1, N_2\)

If Accumulator Overflow \(N_1, N_2\)

If Quotient Overflow \(N_1, N_2\)

If \((I)\) is specified numerically:

\[
\begin{array}{ccc}
S & 1 & 45 \\
& 012 & 0000 & 00000 \\
I & 0 & 00000 \\
\end{array}
\]

If \((I)\) is specified symbolically:

\[
\begin{array}{ccc}
I & S & 1 & 03 \\
& 012 & 0000 & 00000 \\
0 & 00000 \\
\end{array}
\]

Parameter, \(I\)

If \((N_j)\) is missing:

\[
\begin{array}{ccc}
S & j & 36 \\
& 003 & 0000 & 00000 \\
\end{array}
\]

\[
\begin{array}{ccc}
\end{array}
\]

\[
\begin{array}{ccc}
0 & 00000 \\
\end{array}
\]
If \( N_j \) is stated:

\[
\begin{array}{ccc}
N_j & j & 36 \\
S & 0 & 0
\end{array}
\]

\[
\begin{array}{ccc}
S & j & 16 \\
003 & d & 000 \text{ 0fff} \\
S(c) & 0 & 0
\end{array}
\]

(if not drop-out)

\[
\begin{array}{ccc}
S & j & 36 \\
003 & 0000 & 0000 \\
0 & 0
\end{array}
\]

(if drop out)

SUCCESSOR item \((ID = 03)\)

(See Phase III)

Ignore item \((ID = 50)\)

(See Phase III)

Adjacent register flag item

(When one expression must be in a register adjacent to another, as in the relational - IF statements.)

TO and FROM items

ID = 40 \quad ID = 43

(See Phase III)
Summary of Identification Digits Assignment (Assigned by Phase I)

The preceding items are all identified by a two digit number, the ID.  File 2 and File 3 items have their ID in the right-most digits of the second word (W2);  File 4 has the ID in the first word (W1).

<table>
<thead>
<tr>
<th>ID</th>
<th>DEFINITION</th>
<th>ID</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>parameter</td>
<td>26</td>
<td>input listed variable</td>
</tr>
<tr>
<td>02</td>
<td>dimensioned variable</td>
<td>27</td>
<td>input listed variable(</td>
</tr>
<tr>
<td>03</td>
<td>name referencing a parameter</td>
<td>28</td>
<td>output listed variable(</td>
</tr>
<tr>
<td>04</td>
<td>equivalenced variable</td>
<td>29</td>
<td>output listed variable(</td>
</tr>
<tr>
<td>05</td>
<td>common</td>
<td>30</td>
<td>name of routine being &quot;CALLED&quot;.</td>
</tr>
<tr>
<td>06</td>
<td>mode specifier (variable)</td>
<td>31</td>
<td>single name unsubscripted CALL argument</td>
</tr>
<tr>
<td>07</td>
<td>mode specifier (function)</td>
<td>32</td>
<td>name of SUBROUTINE</td>
</tr>
<tr>
<td>08</td>
<td>executable continue</td>
<td>33</td>
<td>SUBROUTINE argument</td>
</tr>
<tr>
<td>09</td>
<td>non-executable continue</td>
<td>34</td>
<td>name of FUNCTION</td>
</tr>
<tr>
<td>10</td>
<td>format continue</td>
<td>35</td>
<td>SAL exit</td>
</tr>
<tr>
<td>11</td>
<td>SAL continue</td>
<td>36</td>
<td>drop out reference in &quot;IF&quot;</td>
</tr>
<tr>
<td>12</td>
<td>assignor (assign A,</td>
<td>37</td>
<td>Hollerith argument reference</td>
</tr>
<tr>
<td>13</td>
<td>assignee to B)</td>
<td></td>
<td>variable in CONSTANT</td>
</tr>
<tr>
<td>14</td>
<td>unparenthesized variable reference</td>
<td>38</td>
<td>mode of store item preceding exp. TO item</td>
</tr>
<tr>
<td>15</td>
<td>format reference</td>
<td>39</td>
<td>single name subscripted CALL argument</td>
</tr>
<tr>
<td>16</td>
<td>statement name reference</td>
<td>40</td>
<td>FUNCTION argument</td>
</tr>
<tr>
<td>17</td>
<td>must be assigned variable</td>
<td>41</td>
<td>FROM item</td>
</tr>
<tr>
<td>18</td>
<td>listed assignment</td>
<td></td>
<td>OPEN subroutine, library subroutine</td>
</tr>
<tr>
<td>19</td>
<td>induction variable - begin loop</td>
<td>42</td>
<td>literal appearance of numeric</td>
</tr>
<tr>
<td>20</td>
<td>induction variable - end loop</td>
<td>43</td>
<td>quantities and numeric assignments</td>
</tr>
<tr>
<td>21</td>
<td>stored variable</td>
<td>44</td>
<td>limit of DO</td>
</tr>
<tr>
<td>22</td>
<td>stored variable(</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>arithmetic function name</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>arithmetic function argument</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>variable( in arithmetic)</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>DEFINITION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>Function reference (&quot;F&quot; was dropped.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>adjacent register flag (precedes first expression)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>arithmetic and LSC statement operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>beginning and end of list item</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MODE WORD:

Information about an item is passed from phase to phase by a mode word which is part of the item. Phase I assigned mode words to literal appearances of numeric quantities and to operations. Phase III assigns mode words to items which represent variables and functions. Mode words are further up-dated in Phase IV, V, and VI; it is then used by Phase VII to generate code items. The mode word is a full 12-digit LARC word broken down as follows. This summary includes information supplied to the mode word by Phases I to VI.

\[
a, b, c, d, e, e, e, u, f, f, f, f
\]

<table>
<thead>
<tr>
<th>a (storage assignment)</th>
<th>b)arithmetic mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ordinary</td>
<td>Not a Variable, Function, or Constant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Common</td>
<td>Integer</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Equivalence</td>
<td>FLOATING</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Common and Equivalence</td>
<td>Double Precision</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Induction Variable</td>
<td>Integer in integer form</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Arithmetic Function</td>
<td>B-Box word</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Dummy Argument of an Arithmetic Function</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Built in Function</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Library Function</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Dummy Argument of FUNCTION or SUBROUTINE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>Fast Register</td>
</tr>
<tr>
<td>1</td>
<td>Variable</td>
</tr>
<tr>
<td>2</td>
<td>Constant</td>
</tr>
<tr>
<td>3</td>
<td>Statement name</td>
</tr>
<tr>
<td>4</td>
<td>Function or Subroutine</td>
</tr>
<tr>
<td>5</td>
<td>Format</td>
</tr>
<tr>
<td>6</td>
<td>Operation or Punctuation</td>
</tr>
<tr>
<td>7</td>
<td>Built in Function</td>
</tr>
<tr>
<td>8</td>
<td>Marth routine</td>
</tr>
</tbody>
</table>

**d(quantifier):**

If \( c = 0 \)

\[ d = \text{number of fast registers (1 or 2)} \]

If \( c = 1 \)

\[ d = \text{dimensionality (0, 1, 2, \ldots, 9)} \]

After Phase IV, \( d = 1 \) if reference is subscripted, otherwise 0.

If \( c = 2 \)

\[ d = 0 \]

If \( c = 3 \)

\[ d = 0 \] if the statement reference is direct
\[ d = 1 \] if the statement reference is assigned

If \( c = 4 \)

\[ d = 0 \]

If \( c = 5 \)

\[ d = 0 \] if the format reference is direct
\[ d = 1 \] if the format reference is assigned

If \( c = 6 \)

\[ d = \text{number of operands for the operator (see definition of eee for the value of } d \text{ in each case)} \]
If \( c = 1 \) and \( d \neq 0 \)

> _eee_ is the dimension table reference for the variable.

If \( a = 9 \)

> after Phase IV _eee_ is the argument number of the variable

If \( c = 4 \)

> _eee_ is the number of arguments of the function.

If \( c = 6 \)

> _eee_ represents the operation (the list below gives values for _eee_ and the corresponding value for _d_).

Specifier digits _eee_ when \( c = 6 \). (Operation or Punctuation)

<table>
<thead>
<tr>
<th><em>eee</em></th>
<th><em>d</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
</tr>
</tbody>
</table>
36 IF SENSE LIGHT 3
37 GO TO 1
(and ASSIGNED GO TO)

38 IF SS 2
39 IF > 0 1 Followed by GO TO . . .
40 IF = 0 1 (generated for IF POS. etc.)
41 IF < 0 1
42 IF > 1
43 IF = 1
45 COMPUTED GO TO 0 K, the number of GO TO operands, is put into the M-portion of 3rd word of the operator item.

49 WT 1
50 REW 1
51 REW INT 1
52 BACKSP 1

54 IF ( ) 3 (FORTRAN IF statement)
55 IF > 0 1
56 IF ≠ 0 1
57 IF < 1 Followed by a GO TO
58 IF > 1 (generated for IF NOT, etc.)
59 IF ≠ 1
60 IF < 1
61 RT 1

ufff:

If c = 1 and a ≠ 4

u = 0

fff = dictionary reference to name or dictionary reference to name and constant (u = 1 if a = 9 and constant is negative or sequence of generated statement name.

If c = 1 and a = 4

ufffff = new name of induction variable item

If c = 2 and b = 3

u = 0

ffff = dictionary reference to low order part of the constant

If c = 6

Phase VI may further use uffff for the mode words of operations as follows
\( u = 0: \) expression is not common

\( u = 1: \) expression is common and is not in the first major sequence of its occurrence.

\( u = 2: \) expression is common and is in the first major sequence of its occurrence.

If \( u = 1, 2, \) then \( \text{fff} = \) the integer which represents which subexpression.

**Note:** After Phase VII, the mode word described above no longer exists. The code items to File 8 incorporate the information in the mode word of the File 7 items. The Phase VIII report contains this information.
General Phase I flow with details on continue items and sequencing.
LARC SCIENTIFIC COMPILER Phase I
"GET $3$" "GET XI" "Get next character"

Flags and symbols:
BLANK : set if blank or % character is not to be ignored.
eos : end of statement flag.
i : current word in line
j : current character position
Note: i and j are set initially by main control program.

$1 \rightarrow T8 \ A1 \ GETXI$
$x+1 \ end \ of \ statement \ return \ (j:99)$
$x+2 \ normal \ return$

Get character in word i, position j of line into lo-order part of $3$.

Set $6$ to $x+2$.

Output line to print file. Advance print sequence for continuation "Get next line".

Refresh return index.

Set eos flag. 99 $\rightarrow$ $8$ (end symbol)
GET V

START

0 → V, V+1
0 → C
0 → N

eos
set
reset
d+1

0 → f
reset; exp, exs-in, exps, err, func, dec
70 → d.

G

Get next character ($) → 71

β → GETV3

* / (%) → β → GETV3

blank

G

+ → exps.
reset

set

β

- → expsin.
reset

18

17

exp.
reset

set

C

G

⊙ 
TB A1 GETV

α+1 end return
α+2 normal return
with variable in V and V+1

numeric

C: 0

set

N

reset

= 1

1(10) → N
fix → mode
0 → exponent

alph.
reset

G

GETV3

numeric part of f → f

E was encountered.

exponent : 10^0

Attach f to exponent, reset exps.

d+1 → d

= 22

f: 2a

≤

≠

f: 0

E
reset

exp

set

12

14

13

f + 1

f

f

92

C+1 → C.
d: T8 A1 ASCAN

\( d+1 \) error
\( d+2 \) -1 level ")" return
\( d+3 \) 0 level ")" return
\( d+4 \) end return

Note: All items output get sequence number SW
Note: \( d+2 \) exit prohibited if arith. is set and
      subscript is reset
\( d+3 \) exit prohibited if subscript is set.
The variable obtained is the first of an argument of a CALL.

Form name item from SF, appending H. Make dictionary entry for name thus formed (SFH). Output item for name with DREF in mode word to File 4. Get next V characters and form SAL item(s) with SFH as tag. Make last char. '99'. Send item(s) to F9, 3
SF+1 → SF  Asoft+1 → Asoft

Output type 31 "call argument" item to File 3. (Only one variable is the argument)
Asoft+1 → Asoft

Save V and G8 etc., Set arg.

Set next

Output previously saved argument with ID: 41 to File 3
Asoft+1 → Asoft

Set next
LARC SCIENTIFIC COMPILER  Phase I

LIST SCAN

LSCAN 1

reset funct 2, LS9
O -> P1
L -> L'

Get V

Q

25

none

set d+2

return

C:10

1

LSCAN 2

set

reset

N

26

set

25

reset

f

function

set

arith.

set

LS9

reset

A

LSCAN 3

10

set

37

V -> V'

Output subscripted variable sequence SW and subsequence O2.
Output colon item with subsequence O1.
Set subscript flag.

We have: V(?

21

set LS9

D:

= ?

57

Arithmetic scan program to scan subscript is -1
parenthesis level.
Arithmetic scan output subscript.

28

Reset subscript flag.

25

none

LSCAN 1

Get g

- g -> D

53

Output unsubscripted variable item.

23

SW+1 -> 3W

3

LSCAN 4

phase

1.40

D: = ?

29

P1:0

= ?

TS A1 LSCAN

d+1: Fatal error

d+2: End return

d+3: " = " return

Note: The GETX1 program obtains the next character of input (ignoring blanks)
(ARITHMETIC FUNCTION)
SA -> print sequence
SA+1 -> Sw!
set function 2 flag
SAVE Arithmetic function
definition name.
0 -> i

14

Get V

11

18

numeric
dummy arg.

17

i+1 -> i
Output dummy
variable item 5th
sequence SW and
subsequence b

Output type 23 item for V to
File 3 with SW, SS=0, and i
in 3rd word.
Output sp. expression mode item
for V(39) with & Sw' to File 3

missing argument

15

27

set

reset

12

13

Get 3

should be "="

α+3

3 :=

assume it
was "="
Enter after letters "GσT" encountered

1. **TO Gscan**

2. **d+1**
   - **return** if not a GσT

3. **d+2**
   - **if a GσT**
     - with appropriate GσT items output and D the delimiter

Note: This program outputs all items with sequence SW

Note: GSCAN set upon exit if this statement still might be arithmetic.
LARC SCIENTIFIC COMPILER  Phase I

Go to SCAN (GSCAN) (2)

GSCAN 2.

Output type 18
item for V, ss = 1
15 File 3
i+1 → i

Output type 16
item for V, ss = 1
15 File 3
i+1 → i
Subroutine GETSV
\( ax \)  TB A1 GETSV
\( ax+1 \)  EDS
\( ax+2 \)  return with signed
number in \( V, V+1 \)

\[ \text{S} \]

\[ \text{GET V} \]

\[ \text{none} \rightarrow R1 \]

\[ \text{C:O} \]

\[ \neq \rightarrow R2 \]

\[ D:+ \]

\[ = \rightarrow \text{set plus} \]

\[ D:- \]

\[ = \rightarrow \text{reset plus} \]

\[ \text{GET V} \]

\[ \text{none} \rightarrow R1 \]

\[ \text{plus} \]

\[ \text{set} \rightarrow R2 \]

\[ \text{reset} \rightarrow \]

\[ R2 \]

\[ \text{set} \rightarrow \]

\[ V \rightarrow V \rightarrow R2 \]

Subroutine VFUNCT
Use: \( ax \)  TB A1 VFUNCT
\( ax+1 \)  here if \( V \) ends in \( F \) and \( D = ( \)
\( ax+2 \)  here if not

\[ \text{S} \]

\[ \text{FUNCT1} \]

\[ \text{set} \rightarrow ax+1 \]

\[ \text{reset} \]

\[ C:O \]

\[ \neq \rightarrow ax+2 \]

\[ D:() \]

\[ \neq \rightarrow ax+2 \]

\[ C:3 \]

\[ \geq \rightarrow ax+2 \]

\[ \text{ax+1} \]

\[ \text{Does } V \text{ end in } "F"? \]

\[ \text{Yes} \rightarrow ax+1 \]

\[ \text{No} \rightarrow ax+2 \]
LARC SCIENTIFIC COMPILERS

Phase 1

ARITHMETIC STATEMENT SCAN

END of arithmetic expression. May not
be the end of a source statement

1

Function 2
reset

3

SA + 3 → SA

Finished Arithmetic function

If EoS is reset, then LSC is analyzing:

(expression)

\[
\begin{align*}
\text{IF POSITIVE} & > 0 \\
\text{IF NEGATIVE} & < 0 \\
\text{IF ZERO} & = 0 \\
\text{IF NOT POSITIVE} & \leq 0 \\
\text{IF NOT NEGATIVE} & \geq 0 \\
\text{IF NOT ZERO} & \neq 0
\end{align*}
\]

11

13

19

3

ARS 2

next chars, positive?

Get 3

none

ARS 2

next char, ≤ 0

Is it E?

ARS 2

next chars, positive?

next char, T?

Reverse not

ARS 2

ARS 2

ARS 2

ARS 2

25

ARS 1

ARS 1

ARS 1
LARC SCIENTIFIC COMPILER

Phase I

CONITM.

S

SAVE return index
S SS 45 \rightarrow VITM

N:1

≠

35

VEponent > 59?

No

Yes

50

1.0 \rightarrow V

[V] \rightarrow V

0 1 2 0 0 \ldots 0 \rightarrow VITM + 1 \text{ (mode word)}

V \rightarrow VITM + 2

Output 3 word item to File 4.
Refresh return index

α

T8 A1 CONITM

α + 1

Sequence (hi 5 digits)

α + 2

Subsequence (next 5 digits)

α + 3

return
LARC SCIENTIFIC COMPILER

READINPUTTAPE i,n,list
READOUTPUTTAPE i,n,list

Phases

Phase I
READPAPERTAPE i,n,list
WRITEPAPERTAPE i,n,list
TYPEWRITE i,n,list

Diagram:

[Diagram of flowchart with decision points and data flow]

Output:
Type: 49
Operator: item with C=6, D=3, EEE=OPID
in mode word.

Enter subroutine if output type 45
item is File 4

Output 5 File 3:
W1: V
W2: S 119
W3: —
LARC SCIENTIFIC COMPILER  Phase I

READ DRUM i, j, list
WRITE DRUM i, j, list
READ LARC DRUM i, j, list
WRITE LARC DRUM i, j, list
READ TAPE i, list
WRITE TAPE i, list

GET1

nextchar m ?

24 → opid  
2 → u

C1

26 → usid  
27 → sid

C2

Output operator item to File 4
1 → i

GETV

N set  
CONIRM

Output is File 3:
W1: V
W2: S, i, 14
W3: —
DEFINE LARC DRUM i, j, k, l.
POSITION HEAD i, j
ADVANCE HEAD i
BACK HEAD i

NEXT CHARS.

L? No R1

NEXT CHARS.

ARC DRUM? No 1

18 -> OPID
4 -> il

C2

NEXT CHAR.

E? No R1

NEXT CHAR.

HEAD? No 1

20 -> OPID

C1

NEXT CHAR.

D? No R1

NEXT CHAR.

A? No R1

NEXT CHAR.

NHEAD? No 1

19 -> OPID
2 -> il

C2
LARC SCIENTIFIC COMPILER  
Phase I

ASSIGN N (9) TO M

1

get v

C:0 = R1

D:5 = N

V: Fbbbbb

D:1 =

V ← first word of File 2 item

next chars, "to"?

No

4

48

F

R2

V ← word 3 of File 2 item (5 1 12) → word 3
Send 3 word item to File 2
Output to F4:
W1: Sw 0 49
W2: 62030 (5)
W3:

Yes

0

Set alph

GET v

31 =

C:0

R2

R1

1 → i

Sw + 1 → s

R2

D: end

2

R2

R2

Reset

Set

4

25

2
LARC SCIENTIFIC COMPILER Phase I
DIMENSION A(4,N), B(M,N)... (1)

DIMENS

next char. i? No R1

SD → print sequence
SD+1 → SW

next char. on? No 1

Yes

1

GETV none R2

5

set

N

reset

12

reset

13

D: (eos set
eq

14

C: 0

14+1

V → W

1 → j

2

GETV none

C: 0 ≠

16

set

Output is F3:
W1: V
W2: 8w j 03
W3: -
LARC SCIENTIFIC COMPILER  Phase I

DIMENSION A(q, N), B(M, N), ... (2)

DIMENS 2.
LARC SCIENTIFIC COMPILER

EQUVA

next char. L?

Yes

SE → SW
Print sequence

next char. No

E N C E ?

Yes

3

Start of an Equivalence class

Get \( i \)

\( \neq \)

3: (i)

\( \rightarrow 1 \)

SW + 1 → SW

1

EQUVA 2
LARC SCIENTIFIC COMPILER Phase I

EQUIVALENCE (A,B,(S,N,M),C), (D,E)...

EQUVA 2.

Output to File 3:
W1: V
W2: SW i 04
W3: —
i + 1 —— i

Output to File 4:
W1: V
W2: SW i 03
W3: —

EQUVA 2.
LARC SCIENTIFIC COMPILER
COMMON A, B, C

Phase I

SE -> print sequence
SE+1 -> SW
1 -> j, 0 -> W3

GETV

V: Fbbbbb

D: ( -> R1

10: ≠

15: ¬

2: ¬

R2

1: ¬

GETV

C: O

11: =

14: ¬

31: ¬

1: ¬

Output to File 3:
W1: V
W2: SW j 05
W3: j + 1 -> j

D: 9

13: ≠

1: ¬

D: end

2: ≠

14: ¬

31: ¬

1: ¬

R2

1: ¬

R1

1: ≠

2: ≠

SW+1 -> SE
9 -> CONTB

α + i
LARC SCIENTIFIC COMPILER  Phase I

GOTO n
GOTO n, (n1, n2, ...)
GOTO n, (n1, n2, ...), n

GSCAN

D: end = R2

Gspar

D := = R1

31

R2

SW + 1 → S
2 → i

loop

reset

This statement ends a DO range

1 → i

x+l
Here if really a 'DO'. If this statement terminates a 'DO', loop flag is set by main program, which is an error.

23

\[ \text{SW} + 1 \rightarrow \text{SW} \]
\[ \text{P} + 1 \rightarrow \text{P} (\text{P} \neq \text{DO}) \]
\[ \text{L} + 1 \rightarrow \text{L} \]
\[ \text{V} \rightarrow \text{DATA1}_L \]
\[ \text{S}, \text{L}, \text{P} \rightarrow \text{DATA2}_L \]

Output k File 3:
\[ \text{W1: Vj} \]
\[ \text{W2: SW 1 19} \]
\[ \text{W3: DS L P} \]

6

\( V_j: \text{numeric} \)

\( \neq \)

Output k File 3:
\[ \text{W1: Vj} \]
\[ \text{W2: S j+1 46} \]
\[ \text{W3: DS L P} \]

25

Output k File 4:
\[ \text{W1: S j+1 45} \]
\[ \text{W2: mode} \]
\[ \text{W3: Vj} \]

\[ j: 3 \neq \]

\[ j+1 \rightarrow j \]
\[ \text{DO4} \]
Phase I
IF STATEMENT ANALYSIS

LARC SCIENTIFIC COMPILER

Here for "OR"
Output:
F3: W1: JOE
W2: SW 1 16
W3: DS L P

Here for "AND"
Output:
F3: W1: SAM
W2: SW 1 16
W3: DS L P

Reset M

Set M
SW+1 → SW

Output:
F2: W1: SAM
W2: SW O B
W3: DS L P

Generate new SAM from SW.

20

Output:
F3: W1: SAM
W2: SW 1 16
W3: DS L P
SW+1 → SW
F2: W1: JOE
W2: SW O B
W3: DS L P
SW+1 → SW

Here for "GOT"

next char. (?)
Yes
No

next char. (?)
Yes
No

24 IF Q

23 IF 7

GSCAN error

86
IF 2
**LARC SCIENTIFIC COMPILER  Phase I**

**CONSTANT** \( A_1 = a_1, A_2 = a_2 \ldots A_k = a_k \)

---

**Diagram:**
- **Node 1:**
  - **Next char N?**
    - No: **R1**
    - Yes: **D =**
- **Node 2:**
  - **D =**
    - No: **1**
    - Yes: **R2**
- **Node 3:**
  - **Cos set**
    - **R2**
  - **Cos reset**
- **Node 4:**
  - **31**
- **Node 5:**
  - **1**
- **Node 6:**
  - **C = O**
    - No: **25**
    - Yes: **N**
- **Node 7:**
  - **V → X**
- **Node 8:**
  - **GETSV**
    - No: **R1**
    - Yes: **C = O**
- **Node 9:**
  - **C = O**
    - No: **24**
    - Yes: **N**
- **Node 10:**
  - **Output:**
    - F3: W1: X
      - W2: SC = 38
      - W3: Y lo order
      - W4: W1: SC = 95
      - W2: mode
      - W3: V hi order
      - SC + 1 → SC

---

**Notes:**
LARC SCIENTIFIC COMPILER  Phase I
CALL n(m1, m2, ..., mR)
IF ACCUMULATOR OVERFLOW $n_1, n_2$,
IF QUOTIENT OVERFLOW $n_1, n_2$
IF DIVIDE CHECK $n_1, n_2$

IF QUOT

next char
I?

No

Yes

R1

set alph.

GET V
none

eos

reset

D: != 31

12

set alph.

GET V
none

R2

2

Output F3:
W1: V
W2: SW 1 16
W3: DS LP

Output F4:
W1: SW 0 49
W2: 61037 (5)
W3: __

SW + 1 → SW
2 → i

R1

R2

1 → i
FORMAT

next char (?)

SF → print sequence
SF + 1 → SF, $W$
1 → L
0 → J
reset $a$, $b$, $c$, $h$
set continue ID = 10
SNAME → SALNAM
SF → SALSEQ
Start sal item generation

G
Form 2

R1

2 → i

44

1 → i

R2
LARC SCIENTIFIC COMPILER

STOP n
PAUSE n

Phase I

next char P?

16 → OPID

GET V

eos

reset

Output to F4:
W1: SW 0 49
W2: 6001D(5)
W3: V

next char S?

next char E?

15 → OPID

R2

2 → i
SW + 1 → S

R1

1 → i

dt0
LARC SCIENTIFIC COMPILER

Phase 1

END (i1, i2, i3, i4, i5)

END

22222 \rightarrow i1, i2, i3, i4, i5
Reset enderr.

GET XI \rightarrow \text{none} \rightarrow 1

i \rightarrow \text{R1}

j \rightarrow \text{\#}

10 \rightarrow 1 \rightarrow j

GET V \rightarrow \text{none} \rightarrow 19 \rightarrow 2 \rightarrow 1

C:O \rightarrow \# \rightarrow \text{Is \( V \) an integer?}

\text{Yes} \rightarrow \text{Is it 0, 1, or 2?}

\text{No} \rightarrow \text{set enderr}

\text{Is it 0, 1, or 2?}

\text{Yes} \rightarrow \text{set enderr}

\text{No} \rightarrow \text{Is \( V \) an integer?}

\text{Yes} \rightarrow V \rightarrow i1, j

\text{No} \rightarrow D:5 \rightarrow \# \rightarrow \text{set enderr}

j \rightarrow \text{Reset}

D:0 \rightarrow \# \rightarrow \text{eos set}

eos \rightarrow \text{Reset}

D:0 \rightarrow \# \rightarrow \text{eos set}

eos \rightarrow \text{Reset}

GET XI \rightarrow \text{none} \rightarrow R1
PHASE II

GENERAL DESCRIPTION

The functions of Phase II are:

1. To append information associated with a statement name, \( N_i \), (continue item) to items generated from ASSIGN \( N_i \) to \( M \) and \( \text{GO TO } M \), \( (\ldots N_i \ldots) \) items,

2. To detect, error mark and eliminate multiple uses of an alphanumeric word as the name of statements within an LSC source program,

3. To issue "collector" items for each built-in (open) and library (closed) subroutines, and

4. To attach pseudo-sequence numbers for most items written in File 3 by Phase II.

All items output by Phase II will be written in File 3, except error items which go to File 92.

A group of items to be analyzed in concert by Phase II consists of all items with the same name; there should be one continue item in each group. When all groups in File 2 have been processed, the function collector items are filed, library subroutine names are given dictionary references, and return is made to the control program.

STATEMENT NAME (CONTINUE) ITEMS

A group which contains more than one continue item will generate only one continue item for File 3 and one error item for each of the other continue items. The continue item sent to File 3 will be:
a. The first item generated from a named executable statement (ID = 08), or

b. If none of these, the first item generated from either a named non-executable statement (ID = 09), or from a named SAL instruction (ID = 11), or

c. The first item generated from a FORMAT statement (ID = 10).

The sequence number of (08, 09, and 11) items in File 3 will be (CRS-4); the sequence number of (10) items will be (CRS-3). The original sequence number of these items has become the subsequence. (CRS) is the initial value sequence counter used for executable statements.

ASSIGN STATEMENTS

Each assign item (ID = 12) in File 2 generates three items in File 3 and an error item if a continue item is not within the group. The first of the items in File 3 is merely the assign item carried forward. Two assigned items (ID=13) are generated for each assign item. The first item has sequence (CRS-2) and sub-sequence equal to the sequence of the continue item which was assigned; the second item has the same sequence as in the (12) item. Each of these (13) items will be in an assigned variable group in Phase III and will contain in the third word the DO sequence, level, and plateau numbers of the appropriate continue item.

ASSIGNED GO TO STATEMENTS

Executable statement names which appear in "assigned go to" lists and in assign statements generate items (ID=18) which appear in assigned variable groups in Phase III. These items contain the DO sequence, level and plateau numbers of the appropriate continue item in word 3. An error item is generated if the listed assignment does not have a corresponding assign statement or continue item.

LIBRARY AND BUILT-IN FUNCTIONS

One Item (ID=44) is sent to File 3 for each possible built-in and library function which
may have been referenced in an LSC source program. The sequence number for these items is (CRS-1); the third word contains an embryonic mode word for use by later phases. This mode word contains the (a, b, c) digits and for built-in functions an assigned operator for Phase VII in the (ffff) digits.
PHASE II ITEM FORMATS

From Phase I

1. CONTINUE - executable statement

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>NAME</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>S . O . 08</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>DS , L , P</td>
<td></td>
</tr>
</tbody>
</table>

ID = 09, 10, 11 are similar

2. ASSIGN NAME = TO M

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>NAME</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>S . 1 . 12</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

$S_c$ = sequence of continue item
DS , L , P are from the continue item for NAME

3. GO TO M, ( . . . N_j . . . )

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>N_j</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>S . j+1 . 18</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

To Phase III

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>NAME</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>(CRS-4) . S . 08</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>DS , L , P</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>(CRS-2) . S_c . 13</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>DS , L , P</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>A . S_c . 13</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>DS , L , P</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>S . S_c . 18</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>DS , L , P</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>W2</td>
<td>W3</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>ABS</td>
<td>(CRS-1)</td>
<td>0000144</td>
</tr>
<tr>
<td>DIM</td>
<td></td>
<td>0000244</td>
</tr>
<tr>
<td>FLOAT</td>
<td></td>
<td>0000344</td>
</tr>
<tr>
<td>INT</td>
<td></td>
<td>0000444</td>
</tr>
<tr>
<td>MAXO</td>
<td></td>
<td>0000544</td>
</tr>
<tr>
<td>MAX1</td>
<td></td>
<td>0000644</td>
</tr>
<tr>
<td>MINO</td>
<td></td>
<td>0000744</td>
</tr>
<tr>
<td>MIN1</td>
<td></td>
<td>0000844</td>
</tr>
<tr>
<td>MOD</td>
<td></td>
<td>0000944</td>
</tr>
<tr>
<td>SIGN</td>
<td></td>
<td>0001044</td>
</tr>
<tr>
<td>XABS</td>
<td></td>
<td>0001144</td>
</tr>
<tr>
<td>XDIM</td>
<td></td>
<td>0001244</td>
</tr>
<tr>
<td>XFIX</td>
<td></td>
<td>0001344</td>
</tr>
<tr>
<td>XINT</td>
<td></td>
<td>0001444</td>
</tr>
<tr>
<td>XMAXO</td>
<td></td>
<td>0001544</td>
</tr>
<tr>
<td>XMAX1</td>
<td></td>
<td>0001644</td>
</tr>
<tr>
<td>XMINO</td>
<td></td>
<td>0001744</td>
</tr>
<tr>
<td>XMIN1</td>
<td></td>
<td>0001844</td>
</tr>
<tr>
<td>XMOD</td>
<td></td>
<td>0001944</td>
</tr>
<tr>
<td>XSIGN</td>
<td></td>
<td>0002044</td>
</tr>
<tr>
<td>MAX</td>
<td></td>
<td>0002144</td>
</tr>
<tr>
<td>MIN</td>
<td></td>
<td>0002244</td>
</tr>
<tr>
<td>XMAX</td>
<td></td>
<td>0002344</td>
</tr>
<tr>
<td>XMIN</td>
<td>(CRS-1)</td>
<td>0022444</td>
</tr>
</tbody>
</table>

707000000012 707000000085 70700000009 707000000010 707000000086 707000000086 707000000087 707000000087 707000000083 707000000084 707000000011 707000000095 707000000010 707000000096 707000000096 707000000097 707000000097 707000000094 707000000086 707000000087 707000000096 707000000097
<table>
<thead>
<tr>
<th>W1</th>
<th>W2</th>
<th>W3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS</td>
<td>(CRS-1)</td>
<td>0000044</td>
</tr>
<tr>
<td>ASIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COSH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TANH</td>
<td>(CRS-1)</td>
<td>0000044</td>
</tr>
<tr>
<td></td>
<td></td>
<td>82400000FFFF</td>
</tr>
</tbody>
</table>
LARC SCIENTIFIC COMPILER  Phase II  2.7

ID08  EXECUTABLE CONTINUE

1  set  001  2NEXT1
    reset

2  set  002  reset
    reset

word 3  →  2DSLP
index  →  I of C
S  →  S cont

set  1

2NEXT1

ID09  NON-EXECUTABLE CONTINUE

1  set  002  2NEXT1
    reset

2  set  OO9  reset
    reset

word 3  →  2DSLP
index  →  I of C

set  2

2NEXT1

ID11  SAL ITEM NAME

2NEXT1

ID10  FORMAT

3  set  005  2NEXT1
    reset

set  3

index  →  I of F

2NEXT1

001  Multiple executable statements with same name.

002  Executable and non-executable statement have same name.

009  Multiple non-executable statements and/or SAL items have same name.

005  Multiple Format statements have same name.
PHASE III

GENERAL DESCRIPTION

The function of Phase III is to analyze the use of all names in an LSC source program. This analysis involves the generation of mode words for functions and variables, the association of statement names with references to statements, and the detailed cross checking of the use of names. The important function of error detection in the compiler is also accomplished in Phase III through this process of cross referencing.

Before Phase III begins operation, File 3, which has been generated by Phases I and II, is sorted on the name of the item and the assigned sequence number. This sort makes it possible to avoid forming and searching tables to obtain information about a name. File 3 items have three words of the following form:

Word 1: Name (8 alphanumeric characters)
Word 2: Digits 1-10: assigned sequence number
        Digits 11-12: item type identification (ID)
Word 3: Special information

Phase III begins its operation by requesting a group from the general file maintenance routine (FILE). A group of items in Phase III contains all items with the same name arranged on monotonically increasing order according to sequence numbers assigned by Phase I and Phase II. This ordering brings defining items to the top of a group enabling Phase III to complete mode definition by skimming off the top of a group rather than requiring a scan of the complete group, necessary if defining items had been homogenized with reference items.

The first scan analyzes items for determination of mode word information. Items are examined until the storage assignment, arithmetic type, and class of a group have been determined. These characteristics are translated into a, b, and c digits of the mode word; this word is completed; and the second scan, which produces items for File 4
is initiated. After all name-groups have been processed by Phase III, return is made to the control program.

DEFINITION HIERARCHY

The LSC set of sequence counters described in Phase I and Phase II is used to arbitrate errors arising from multiple definition of LSC names. Phase III will classify a group as belonging in one of the 16 possible categories as dictated by the first item in a group with ID digits listed below. Subsequent definitions and improper references will be considered offenses against the first definition.

<table>
<thead>
<tr>
<th>SEQ.</th>
<th>GROUP CLASSIFICATION</th>
<th>ID DIGITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>FUNCTION</td>
<td>34</td>
</tr>
<tr>
<td>SB</td>
<td>SUBROUTINE</td>
<td>32</td>
</tr>
<tr>
<td>ST</td>
<td>PARAMETER</td>
<td>01</td>
</tr>
<tr>
<td>SD</td>
<td>DIMENSIONED VARIABLE</td>
<td>02</td>
</tr>
<tr>
<td>SC</td>
<td>CONSTANT</td>
<td>38</td>
</tr>
<tr>
<td>SA</td>
<td>ARITHMETIC FUNCTION</td>
<td>23</td>
</tr>
<tr>
<td>SA</td>
<td>DUMMY ARGUMENT OF ARITHMETIC FUNCTION</td>
<td>24, note 2</td>
</tr>
<tr>
<td>S-4; S</td>
<td>STATEMENT NAME</td>
<td>08, 09, 11; 16, 35</td>
</tr>
<tr>
<td>S-3; S</td>
<td>FORMAT NAME</td>
<td>10, 15</td>
</tr>
<tr>
<td>S-2</td>
<td>ASSIGNED VARIABLE</td>
<td>13</td>
</tr>
<tr>
<td>S-1</td>
<td>BUILT-IN FUNCTION</td>
<td>44</td>
</tr>
<tr>
<td>S-1</td>
<td>LIBRARY FUNCTION</td>
<td>44</td>
</tr>
<tr>
<td>S</td>
<td>CALL</td>
<td>30</td>
</tr>
<tr>
<td>S</td>
<td>IMPLICIT FUNCTION</td>
<td>47, 25</td>
</tr>
<tr>
<td>S</td>
<td>GENERAL VARIABLE</td>
<td>note 1</td>
</tr>
<tr>
<td>S</td>
<td>INDUCTION VARIABLE</td>
<td>19, note 2</td>
</tr>
</tbody>
</table>
Notes:

1. A group is classified as a general variable group if its first item with sequence (S) has valid ID digits different from those in the preceding table.

2. These groups are sub-groups and may appear within groups with different classifications.

Arguments of a FUNCTION (42) and SUBROUTINE (33) are considered as reference items although they are assigned sequence (SB).

COMMON (05), EQUIVALENCE (04) and mode specification (06, 07) items are adjectival items; that is they restrict, but do not themselves classify a group into one of the 16 categories.

PRONOUN GROUPS

If one item in the group has the ID of a PARAMETER statement, then the value of the parameter is attached to all other items in the group. These items (except the PARAMETER item) are then filed in File 4 as "numeric" items. If the group has more than one PARAMETER item, an error is recorded and the first parameter given is used.

Any item in the group which would not have its value supplied by a parameter generates an error (e.g., GO TO N may not have a parameter given for N). If a group contains an item with the ID of a CONSTANT statement, the group is treated in a similar manner.

STATEMENT NAME GROUP

If the group contains a "continue" item (a statement name), then all other names in the group must be references to that name. (GO TO NAME, etc.) If a group contains only a continue item, nothing is output from that group, as a result only those continue items which are referenced will be treated as "entry points" when computation of common sub-expressions are eliminated in Phase VI.
Reference items (output to File 4) contain the sequence of the continue item in the third word.

In addition to reference items, special control items may be generated; these items are SUCCESSOR and TO and FROM items.

SUCCESSOR ITEMS

A successor item is generated whenever control leaves or enters a block. Phase III generates successor items when four types of items are encountered:

1. A listed assignment in an ASSIGNED GO TO statement,

2. SAL EXIT item,

3. A statement name used as an argument of a CALL,

4. A GO TO item which occurs within an assigned variable group, but the possible assignments are not listed. In this case, a successor is generated for each possible assignment.

The successor item contains the sequence numbers of the continue item (SCONT) and the reference item (SREF). These numbers are obtained in the following manner for the case listed above:

<table>
<thead>
<tr>
<th>SCONT</th>
<th>SREF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Subsequence of an 18 item</td>
<td>Sequence of the preceding 17 item</td>
</tr>
<tr>
<td>2. Sequence of an 08 item</td>
<td>Sequence of a 35 item</td>
</tr>
<tr>
<td>3. Sequence of an 08 item</td>
<td>Sequence of a 31 item</td>
</tr>
<tr>
<td>4. Subsequence of each 13 item within the group.</td>
<td>Sequence of a 16 item</td>
</tr>
</tbody>
</table>

The format of this successor item is contained in the item format part of the Phase III section. (A full description of successor items is in the Appendix).
TO AND FROM ITEMS

Possible transfers of control into, within, and from DO loops generate TO and FROM items. The generation of these items is subject to the following tests:

1. If \( (DS, P)_C = (DS, P)_R \), neither item is output.

2. If \( (DS, P)_C \neq (DS, P)_R \), and if \( (DS)_C \neq 0 \) a TO item is output.

3. If \( (DS, P)_C \neq (DS, P)_R \), and if \( (DS)_R \neq 0 \) a FROM item is output.

DS is the DO sequence number of the current loop
P is a count of the number of preceding DO statements
sub C denotes a continue item
sub R denotes a reference item

The DS and P numbers of continue and reference items are contained in the third word of these items. The occurrence of items for which TO/FROM tests are made is identical to the four occurrences for which successor items are generated; in addition all ordinary references to statement names \((ID = 16)\) which are not in an assigned variable group are tested for TO/FROM generation. These TO and FROM items are used by Phase IV for DO loop analysis.

The format of TO and FROM items is contained in the item format part of the Phase III section.

PSEUDO-CONTINUE ITEM

The pseudo-continue item \((ID = 13)\) generated in Phase II, will be processed according to the type of reference made to it as a statement name. However, in the same group, the name may be referred to as a variable; if it is, then the remainder of the items (not statement name references) are processed accordingly. In this case, the "ASSIGN A, TO B" statement is treated as a "definition point" for the variable B. (Definition point is described below.) A group which contains an assigned variable may, therefore, have both statement referencing items and variable referencing items without generat-
ing error messages. Note, however, that such a variable may not be subscripted; an error item will be recorded if such a variable is subscripted.

**VARIABLE GROUP**

If the group obtained does not contain any specified defining items, it is assumed to represent a variable. All occurrences of the variable in the source program will be in the group. The function of some of the items is to indicate information about the variable (e.g., it is floating point, double precision, 3-dimensional, is in an Equivalence or Common statement, etc.). When the mode is not specified, its first letter determines its arithmetic mode according to the LSC naming conventions. This information about the variable is collected to form a single mode word for the variable. This mode word is attached to every item in the group generated from an executable statement.

If the variable occurs in a dimension statement, Phase III does not know what the maximum dimensions are since these may be parameters which are assigned in a different group by a Parameter statement, however the number of dimensions is known. The number of dimensions is assigned to the mode word as is the location in the dimension table where the maximum dimensions for the variable are to be found. (The actual maximum dimensions will be stored there by Phase IV).

Later, in the compiler (Phase VI), an analysis is performed to avoid the recomputation of common subexpressions within a "block." In order to perform this analysis, it is necessary to know the last definition point (DP) of each occurrence of a variable. The last definition point shows where the last point (with the smallest sequence) in the program was that the variable was defined. A defining item may come from the following sources: (1) An Input Statement, (2) an Argument of a CALL, (3) the occurrence of a variable on the left hand side of an equation (a store item) or an ASSIGN statement. A defining item will have its sequence number (DP) assigned to every other item below it in the group until another defining item in the group appears, then the sequence number of the new defining item is used for DP, etc. (Recall that a group is sorted on the Phase I assigned sequence number in File 3).
Although the main function of Phase III, with respect to a variable, is generating a mode word and assigning definition points to each occurrence of the variable in an executable statement, some items require additional processing. These are:

1. Induction Variable (the controlling variable of a DO loop).

In the statement: \[ \text{DO } \mu, \quad I = L, M, N \]

I is the induction variable. Every occurrence of the quantity, I, in the range of the DO (all statements from the DO, down to, and including \( \mu \)) must be recognized and associated with the induction variable in the DO statement. It is possible, in a source program, that several other induction variables named "I" are controlled by different DO loops. The variable, I, may also appear outside the range of a DO. Any other item (not a defining item) in the group whose sequence number lies in this range is identified as being associated with that particular induction variable. This information is retained in the mode word for that item as well as a new name for the induction variable of a DO. In the case of induction variables, the DO statement is the definition point for the variable. Phase III will error any defining item for the induction variable in the range of a DO. (i.e., a defining item between a 19 and a 20 item).

2. Dummy Variables in an Arithmetic Function Definition

Phase I will have associated with each dummy variable \( X_i \) in:

\[ \text{ABCF} (X_1, X_2, \ldots, X_n) = \text{expression (}X_1, X_2, \ldots, X_n) \]

an identification that they are dummy variables. During Phase III any occurrence of \( X_i \) in the sequence range for the function definitions will be renamed, but a name \( X_i \) occurring elsewhere in the program will not be confused with a dummy variable. Furthermore, a name may be the dummy variable of several arithmetic functions, it will be unambiguously renamed for each function. The arithmetic mode of the variable, \( X_i \), will be defined in the same manner as other variables.
3. Variable storage items are output for non-dimensioned, non-common, non-equivalenced variables.

Further checks performed in Phase III.

1. If N appears in: FUNCTION N( . . . )
then N must also appear in the group as a non-subscripted store item or in an input statement list.

2. If N appears in: CALL N( . . . )
then every occurrence of N in the group must be from a CALL N( . . . ).

3. If N appears in: SUBROUTINE N( . . . )
then there must be only that one item in the N group (i.e., N may not occur anywhere else in the program).

4. A name will be indicated as a subprogram reference (in its mode word) if it is followed by a left parenthesis but does not occur in a DIMENSION statement.

5. If the first item of a group is a built-in or library function item which was generated in Phase II, the group is processed accordingly.
Library function names have been sent to File 91 by Phase I; built-in function names are dropped, since these functions are considered to be unique operators after Phase III.

ERROR PROCEDURES

The recognition of an error in the source language causes a two-word error item to be sent to File 92 and an appropriate item to File 4; the File 4 item generally contains the usual sequence information in word one, the mode word for the group being processed in word two, and the third word of the item in File 3 in the third word. If the error
detected would create anomalies in later phases. Phase III will output a necessary item to allow the compilation to be completed.

OUTPUT FROM PHASE III

Items output to File 4 have a three-word format:

Word 1: Digits 1-5  Sequence number
         Digits 6-10  Subsequence number
         Digits 11-12 Item identification

Word 2: Mode word

Word 3: Special information dependent upon item type.

1. References to variables

   ID = 14, 21, 22, 25, 26, 27, 28, 29, 46

   Word 3 contains the definition point (DP) right adjusted.

2. Referenced statement

   ID = 08, 11

   Word 3 contains (DS, L, P)

   If any non-executable statements (ID = 09) were references, their ID was changed to 08.

3. Statement references

   A. ID = 16

   Word 3 contains the sequence of the continue item right
adjusted. If this item occurs in an assigned group, or should be assigned the D digit of the mode word is 1.

B. \( \text{ID} = 12 \)

Word 3 contains the alphanumeric name of the assignee.

C. \( \text{ID} = 35 \)

This SAL EXITS item dropped (see SUCCESSOR).

4. FORMAT statement

ID = 10

This item is dropped.

5. Format references

ID = 15

Word 3 contains \((DS, L, P)\)

6. PARAMETER

ID , 01

Word 3 contains the value of the parameter in floating point notation. The B digit of the mode word is 1; the C digit, 2. Thus, a parameter must be an integer and need not appear in a mode specifier statement \((\text{ID} = 06)\).
7. Parameter references

ID = 45

Word 3 contains the value of the parameter in floating point notation. Valid references, ID = 03 or 14, are changed to 45. Missing parameter values are set equal to 1.0

8. CONSTANT statements

ID = 38

Word 3 contains the value of the low order part of the constant. It is not checked for being non-zero, nor is an improper mode word B digit changed. Constant references are treated like variable references.

9. Mode specifier

ID = 06, 07

These items are dropped.

10. DIMENSION

ID = 02

Word 3 contains the number of dimensions in fixed point, right adjusted. If more than one dimension declaration is made for a variable, all save the first have ID = 50.
11. EQUVALENCE and COMMON

ID = 04 and 05

Word 3 contains zeros

12. Induction variables

A. ID = 19

Word 1 DS, 1, 19
Word 2 mode word for non-induction variable
Word 3 O . (Se) . 00

Se = sequence of 20 item or, 0———0 for nested DO's on same induction variable.

13. References to induction variable (within DO loop)

A. ID = 07

Word 1 S . SS . 07
Word 2 411 0000 0FFFF
Word 3 DS, L, P

Valid references (ID = 14, 28) are changed to 07.
Arguments of a CALL, ID = 31, are unchanged.

14. CALL

ID = 30

Word 3 contains (DS , L , P) (see 21).
15. Arguments of CALL, unsubscripted

A. \( \text{ID} = 31 \)

These items are treated as references to variables (\( \text{ID} = 14 \)), statement names (\( \text{ID} = 16 \)), or formats (\( \text{ID} = 15 \)) as appropriate.

Arguments of CALL, subscripted

B. \( \text{ID} = 41 \)

These items are treated as references to dimensioned variables (\( \text{ID} = 25 \)).

16. SUBROUTINE

\( \text{ID} = 32 \)

Word 3 contains the number of arguments
The name becomes the SAL LABEL for the object program. (See "OUTPUT TO FILE 91" - below).

17. Argument of SUBROUTINE

\( \text{ID} = 33 \)

References to an argument will contain the argument's order number (within the list of arguments) in the mode word.

Word 3 contains the definition point.
18. FUNCTION

ID = 34

Word 3 contains the number of arguments.
The name becomes the SAL LABEL for the object program. (See "OUTPUT TO FILE 91" - below).

19. Arguments of a FUNCTION

ID = 42

References to an argument will contain the argument's order number (within the list of arguments) in the mode word.

Word 3 contains the definition point.

20. Store expression item

ID = 39

Word 3 contains zeros.
(The mode word of this item is required to properly complete the Polish notation analysis in Phase IV).

21. Library Request Function

ID = 44

This item is output to File 4 only for CALLeed and implicit functions. It provides proper mode information for referenced functions. Word 3 contains the alphanumeric name of function to be included in the object program.
22. **Name of Arithmetic Statement Function**

\[
\text{ID} = 23
\]

Word 3 contains the number of arguments. (See "OUTPUT TO FILE 91" - below.

23. **Argument of Arithmetic Statement Function**

This item is dropped; however, its subsequence number becomes a pseudo-name (UFFFF of mode word) of references to this variable which are in the function skeleton appearing on the right hand side of the function statement.

24. **Variable Storage Item**

For all non-dimensioned, non-equivalence, non-common variables, a storage item is generated with the following format:

Word 1 \((\text{SD}-5)\), 1, 0€
Word 2 mode word
Word 3 0———0

\((\text{SD}-5)\) is the DIMENSION sequence counter less 5;
(i) is a counter denoting the \(i^{th}\) such variable.

25. **SUCCESSOR Items (non-extensible. - see Phase VIII).**

Word 1 \((\text{SREF})\), 0, 03 sequence of reference
Word 2 0200000 (SCONT) sequence of continue
Word 3 0————01
26. TO Item

A. Word 1 (DSc) 00007 40
   Word 2 (DSr) 0———0
   Word 3 (Sc) 0———0

   DO sequence of continue.
   DO sequence of reference.

FROM Item

B. Word 1 (DSr) 00006 43
   Word 2 (DSc) 0———0
   Word 3 (Sr) 0———0

27. Function Reference

ID = 47

If a group contains only 47 and/or 25 items it will be treated as a reference to a compiled function.

Word 3 contains zero. (see 21).

28. Reference to assigned variable) from "ASSIGNED GO TO"

ID = 17

Word 3 contains (DS, L, P)
The mode word D digit is 1.
29. Listed Assignment

ID = 18

These items are dropped; they produce SUCCESSOR and may produce TO and FROM items.

30. Assignee

ID = 13

Word 3 contains the DP, which is the sequence of this item. The mode word D digit is 1.
OUTPUT TO FILE 91

A. File 91 is the dictionary file for LSC; it contains all the names (i.e., symbols) used in the source program and names generated by LSC. Phase III will file all names it encounters (one for each group) with two exceptions.

1. Names of unreferenced statements.

2. Names of library and of all built-in functions.

B. The name of an ARITHMETIC STATEMENT FUNCTION, FUNCTION, or SUBROUTINE when filed in File 91, will be immediately followed by (n) items, one for each argument of the subprogram. These list names are generated by converting the File 91 entry number (CRDREF) to alpha-numeric and appending AR.

   e.g., FUNCTION NAME F (A, B)

   FILE 91 66 NAME
   67 67AR
   68 68AR

C. LSC names which are incompatible with SAL are prefaced with a 9 by Phase III. These names are; HERE, SAME, and SEG ii. The name of a FUNCTION will be attached to the first executable statement of the subprogram; all references to this name within the subprogram will be prefaced with an 8 to avoid having a statement name and variable with the same name.
Error items (described in the Phase IX report) are output as two-word items. They will contain the alphanumeric name of the group in question and hence will facilitate understanding the source error (or probable error). A complete list of all conditions which will cause an error item to be output is found in the Phase IX listing (the "parallel codedit").
Phase III

3.19

LARC SCIENTIFIC COMPILER

**流程图说明**

1. **CONTROL**
   - **SORT F3 ON NAME AND SEQ.**

2. **3QNC**
   - 1. \((\text{CRSD} - 5) \Rightarrow \text{SQ} \Rightarrow \text{CRSD6} \Rightarrow \text{SS} \Rightarrow \text{CRSD6} \Rightarrow \text{DD} \Rightarrow \text{CRSD6}\)
   - 2. \(0 \Rightarrow \text{ONLY 1} \Rightarrow \text{FROM} + 2 \Rightarrow \text{TO} + 1\)
   - 3. \((\text{CRSA} - 100) \Rightarrow \text{CRSK}\)

3. **3NEXTG**

4. **Get next group.**
   - (all items with same NAME)
   - **None**
   - **3CLOSE**
   - Close F4, F91, F92
   - (End of 3Q3)
   - **CONTROL**

5. **Save index word for pass 2.**

6. **3RESET**
   - 1. **Reset**
   - 2. \(0 \Rightarrow A, B, C, D, C, M, D, P, XXX\)

7. **3XXX**
   - **3NOT**
   - **LSCER**
   - (Typewrite: LSCER 3iiiii)
   - \(iii iii = \text{loc. of error detection}\)
   - Transfer to loc. specified by XXX

8. **3AHEAD**
   - Seq of next item: CRSK
   - **3NEXTI**
   - **3D90**

9. **(Continue to look for mode word information)**
   - **(Construct mode word and begin pass 2)**
LARC SCIENTIFIC COMPILER

Phase III

3NAME

(Dictionary entries for arguments of subprograms)

O: no. of args.

≠

3ID 99

NAME → F91

set

1

no. of args → D

Simple variable store item

CRSD6 + 10000 → CRSD6 → L

O → N

L; M; N → F4

3CHEK1

≠ (dimensioned)

DP; O

= (common or equivalent)

A; O

≠ (not a variable)

C; 1

3CHECK

S; D → D10

D10

≠

3ID 99

e.g. If the name of a subprogram, which has three arguments, were filled as the 88th item in the dictionary; the 89th, 90th and 91st entries would be, respectively: 89AR, 90AR and 91AR.

060

No definition point for variable
Parameter is defined more than once.
**LARC SCIENTIFIC COMPILER**

**Phase III**

**ID02**

**Dimension**

**302X**

- **ID02.0**
  - $a' : o$
  - $\neq$
  - **D : o**
  - $\neq$
  - **024**
  - **50 -> ID**
  - **3AHEAD**

- **302X3**
  - **CRDIMT + D : 991**
  - $\leq$
  - **D : 0**
  - $\neq$
  - **096**
  - **1 -> D**

- **302X5**
  - **D + CRDIMT**
  - **EEE**
  - **and CRDIMT**
  - **3AHEAD**

**302V**

- **ID02.1**
  - **number of dimen -> N**
  - **3ITEM4**

**302C**

- **ID02.2**
  - **O26**
  - **number of dimen -> N**
  - **3ITEM4**

**302**

- **ID02.3**
  - **3NOT**

- **ID02.4**
  - **3XXX**

- **ID050**
  - **3ITEM4**

- **ID025**
  - **3NOT**

---

**024**

Name appears in more than one dimension statement.

**026**

Dimension variable and parameter have same name.

**025**

Dimension table exceeded.

**096**

0 - Dimensioned variable.
103 Parameter in equivalence list.
ID 05

A: 0, 1, 2, 3

A + 1 → A

305x

ID 05.0

= 0.2

= 1.3

305x1

AHEAD

027

3AHEAD

305v

ID 05.1

0 → N

3ITEM4

305c

ID 05.2

set

reset

2

104

104

3NST

3XXX

3NST

027 Name in more than one common statement.

104 Parameter in common statement.
VARIABLE MODE SPECIFIER

\[ ID_{06} \]

\[ B : 0 \]

\[ \not= \]

303A

030

\[ \text{Extract B digit of word 3} \]

\[ 3068 \]

\[ 00 \rightarrow ID \]

\[ \text{number of items in group 1} \]

\[ > \]

3AHEAD

FUNCTION MODE SPECIFIER

\[ ID_{07} \]

\[ B : 0 \]

\[ \not= \]

307A

030

\[ \text{Extract B digit of word 3} \]

\[ 3078 \]

\[ 00 \rightarrow ID \]

\[ \text{number of items in group 1} \]

\[ > \]

3AHEAD

030

Name appears in more than 1 mode specifier statement.
NAME OF EXECUTABLE TABLE

308x

308v

308c

308s

308q

308s

Statement and variable have same name

Statement and constant have same name

Statement and parameter have same name

No references to named statement.
3xxxvii (Check for "drop-out") references

0 → 3 ALL

Get next ref. item

3 ALL = 0

(All refs. were "drop-outs")

3id99 (Begin pass 100.)

non-zero → 3 ALL

≤ (sref + 2); (scont)

≠ ID of REF: 36

500 → ID of Ref.

003 0000 00000

S . SS . 36

F9

ID 36 Possible drop-out reference

16 → ID of Ref. (Change possible drop-out"
to st. name ref.

ID 16
ASSIGNED ST. NAME REFERENCES

312x

ID_{12.0} \rightarrow 31D99

312v

ID_{12.1} \rightarrow 041 \rightarrow 312s

312c

ID_{12.2} \rightarrow 2 \rightarrow 042 \rightarrow 312s

ID_{12.2} \rightarrow \text{reset} \rightarrow 043 \rightarrow 312s

312s

ID_{12.3} \rightarrow 3ITEMQ

ID_{12.4} \rightarrow 3xxx

312f

ID_{12.5} \rightarrow (DS, L, P) \rightarrow N \rightarrow 3ITEMQ

041 Variable was assigned as statement reference.

042 Parameter was assigned as statement reference.

043 Constant was assigned as statement reference.
UNSUBSCRIPTED VARIABLE REFERENCE

314K

ID_{14.0} -> SD99

314V

ID_{14.1} -> WORD 31G

VALUE -> N
45 -> ID

314C

ID_{14.2} -> 3ITEM4

314S

ID_{14.3} -> CDS

314F

ID_{14.5} -> C4G

Statement name referenced as variable

Format name referenced as variable
Unassigned variable referenced as statement name.

Parameter referenced as statement name.

Constant referenced as statement name.

Format name referenced as statement name.
\[ \text{ID}_{17} \quad \text{ASSIGNED VARIABLE REFERENCE} \]

\[ \text{ID}_{17,x} \quad \text{ID}_{17,0} \quad \text{ID}_{17,a} \quad \text{ID}_{17,y} \quad \text{ID}_{17,c} \]

\[ 317x \quad 317y \quad 317v1 \quad 317v2 \quad 317c \]

\[ \text{1} \rightarrow D \quad \text{word 3} \rightarrow (D,S,L,P)_R \]

\[ \text{SITEMQ} \]

\[ \text{108} \quad \text{No assignment for variable} \]

\[ \text{093} \quad \text{Assignment made to constant} \]

\[ \text{092} \quad \text{Assignment made to parameter} \]
**LARC Scientific Compiler**

**Phase III**

**Induction Variable**

- **319X**
  - **319V**
    - **319V3**
      - **319V4**
        - **319V5**
          - **319V6**

- **319V7**
  - **L → Sdf 19**
    - **411 → ABC of mode 319V2 → xxx**

- **319C**
  - **319V8**
    - **319V9**
      - **319V10**
        - **319V11**

**Notes:**
- **056** Induction variable not an integer.
- **055** Subscripted induction variable
- **057** Improper to induction variable within its loop
- **058** Constant or parameter used as I.V.
- **059** St. name used as I.V.
- **060** Format name used as I.V.
LARC SCIENTIFIC COMPILER

Phase III

ID_{21}  STORED UNSUBSCRIBED VARIABLE

321x
ID_{21,0} \rightarrow 3ID99

321v
ID_{21,1} \rightarrow \text{Seq.} \rightarrow DP \rightarrow N \rightarrow 3ITEM4

321c
ID_{21,2} \rightarrow 2 \rightarrow \text{reset} \rightarrow O61 \rightarrow 3ITEM4

ID_{21,2} \rightarrow \text{set} \rightarrow O62 \rightarrow 3ITEM4

321s
ID_{21,3} \rightarrow O63 \rightarrow 3ITEM4

ID_{21,4} \rightarrow 3XXX

321f
ID_{21,5} \rightarrow O64 \rightarrow 3ITEM4

\begin{align*}
\text{O61} & \quad \text{Parameter appears as store item} \\
\text{O63} & \quad \text{Statement name appears as store item} \\
\text{O62} & \quad \text{Constant appears as store item} \\
\text{O64} & \quad \text{Format name appears as store item}
\end{align*}
**LARC SCIENTIFIC COMPILER**

**Phase III**

**ID_{26}**

INPUT-LISTED UNSUBSCRIPTED VARIABLE

326x

ID_{26.0} → 3ID99

326v

ID_{26.1} → Seq → DP → N → 3ITEM4

326c

ID_{26.2} → reset

Set

076 → 3ITEM4

075 → 3ITEM4

326s

ID_{26.3} → O77 → 3ITEM4

326a

ID_{26.4} → 3XXX

326f

ID_{26.5} → O78 → 3ITEM4

**076** Parameter in input list

**077** St. name in I/φ list

**075** Constant in input list

**078** Format name in I/φ list
LARC SCIENTIFIC COMPILER

Phase III

328x

328y

328z

328s

328f

\( ID_{28} \)  OUTPUT LISTED VARIABLE UNSUBSCRIPTED

\( ID_{28.0} \)  \( \rightarrow \)  \( 3ID_{99} \)

\( ID_{28.1} \)  \( \rightarrow \)  \( DP \rightarrow N \)  \( \rightarrow \)  \( 3ITEMq \)

\( ID_{28.2} \)  \( \rightarrow \)  \( 45 \rightarrow ID \)  \( VALUE \rightarrow N \)  \( \rightarrow \)  \( 3ITEMq \)

\( ID_{28.3} \)  \( \rightarrow \)  \( 077 \)  \( \rightarrow \)  \( 3ITEMq \)

\( ID_{28.4} \)  \( \rightarrow \)  \( 3\bar{X} \bar{X} \)

\( ID_{28.5} \)  \( \rightarrow \)  \( 078 \)  \( \rightarrow \)  \( 3ITEMq \)

\( 077 \)  St. name in I/O list

\( 078 \)  Format name in I/O list
STORED SUBSCRIPTED VARIABLE

INPUT-LISTED SUBSCRIPTED VARIABLE

OUTPUT-LISTED SUBSCRIPTED VARIABLE

Subscripted reference to non-dimensional variable.
**LARC SCIENTIFIC COMPILER**

**Phase III**

**ID_{25}**

**SUBSCRIPTED VARIABLE IN ARITHMETIC**

325x

```
ID_{26.0} \rightarrow c'.o \neq 47 \rightarrow ID \rightarrow 347x \ (Treat \ as \ function \ reference.)
```

325v

```
ID_{26.1} \rightarrow D:0 \neq DP \rightarrow N \rightarrow 3ITEM4
```

325c

```
ID_{28.2} \rightarrow 072 \rightarrow 3ITEM4
```

325s

```
ID_{28.3} \rightarrow 073 \rightarrow 3ITEM4
```

325f

```
ID_{28.4} \rightarrow 3XXX
```

325f

```
ID_{28.5} \rightarrow 074 \rightarrow 3ITEM4
```

**072**

Subscripted reference to parameter

**074**

Subscripted reference to format name.

**073**

Subscripted reference to statement name.
LARC SCIENTIFIC COMPILER

Phase III

Page 1

1. \( ID_{23} \)

ARITHMETIC STATEMENT FUNCTION

\[ 323,6 \]

\[ ID_{23,0} \]

\[ 4 \rightarrow c', 5 \rightarrow A \]

\[ B : 0 \]

\[ \leq \]

\[ 1^\text{st. char.} \neq X \]

\[ 2 \rightarrow B \]

\[ 1 \rightarrow B \]

\[ 323 \times 6 \rightarrow \text{xxx} \]

\[ \# \text{of args.} \rightarrow D \]

\[ 323 \times 6 \rightarrow \text{ITEMA} \]

\[ \# \text{of args.} \rightarrow N \]

\[ 323 \times 2 \rightarrow \text{xxx} \]

\[ 323 \times 2 \rightarrow \text{ITEMA} \]

\[ 323 \times 2 \rightarrow \text{ITEMA} \]

\[ 323 \times 2 \rightarrow \text{ITEMA} \]

\[ \text{ID: 25, 47} \neq \]

\[ 323 \times 4 \rightarrow \text{ITEMA} \]

\[ 47 \rightarrow \text{ID} \]

\[ \text{Seq.} \rightarrow N \]

\[ \text{ID: 25, 47} \neq \]

\[ \text{Save index word.} \]

\[ D : 0 \neq \]

\[ \text{A: 0, 1, 2, 3} \]

\[ = 2 \]

\[ = 0 \]

\[ = 1, 3 \]

\[ = 3 \]

\[ = 1 \]

\[ \text{ID: 23, 1} \]

\[ \text{ID: 23, 2} \]

\[ \text{reset} \]

\[ \text{set} \]

\[ 069 \rightarrow 323 \times \]

\[ 070 \rightarrow 323 \times \]

\[ 323 \times \]

\[ 323 \times \]
ARITHMETIC STATEMENT FUNCTION

- ID_{23} \rightarrow 3\text{NOT}
- ID_{23,3} \rightarrow 3\text{XXX}
- ID_{23,5} \rightarrow 3\text{NOT}

- 065: Improper reference to arithmetic st. function.
- 066: Arithmetic function name in dimension st.
- 067: Arithmetic function name in common st.
- 068: Arithmetic function name in equivalence st.
Improper use of dummy argument of arithmetic function.
CALL

330X

330X1

ID:99

3ITEM4

3ITEM4

330V

330C

ID:30.1

O15

3ITEM4

3ITEM4

2

reset

set

O17

3ITEM4

3ITEM4

330S

ID:30.3

O18

3ITEM4

3ITEM4

ID:30.4

3XXX

3ITEM4

330F

ID:30.5

O19

3ITEM4

O14

Improper use of called subroutine

O16

Constant referenced as called subroutine

O18

St. name referenced as called subroutine

O15

Variable referenced as called subroutine

O17

Parameter referenced as called subroutine

O19

Format referenced as called subroutine
UNSUBSCRIPTED ARGUMENT OF CALL

331X
ID_{31.0} \xrightarrow{} 3NEXTI

331Y
ID_{31.1} \xrightarrow{} S \rightarrow DP
S \rightarrow N \xrightarrow{} 3ITEM4

331c
ID_{31.2} \xrightarrow{} 4S \rightarrow ID
VALUE \rightarrow N \xrightarrow{} 3ITEM4

331s
ID_{31.3} \xrightarrow{} (DS, L, P) \rightarrow N \xrightarrow{} SUCCESSOR \rightarrow F4 \rightarrow 316TR6

ID_{31.4} \xrightarrow{} 3XXX

ID_{31.5} \xrightarrow{} 315F
LARC SCIENTIFIC COMPILER  Phase III  3.49

**SUBROUTINE**

- **ID**32
  - Other 32 items? Yes → 001
  - No

- **any**34 items? Yes → 002
  - No
  - NAME → CRLABL (Name for object code)

3526

- 4 → c!
- 332 D → XXX
- no. args → D

3 NAME

332 D

- 332E → XXX
- no. args → N

3 ITEM4

332 E

- 003

3 ITEM4

**001**  Multiple subroutine statements.

**002**  Subroutine and function statement.

**003**  Name of subroutine used elsewhere in program.
ARGUMENT OF SUBROUTINE

333x

333y

333z

333c

333s

333a

333f

006 Argument of subroutine in common

008 Argument of subroutine is statement name

007 Argument of subroutine in equivalence

009 Argument of subroutine is format name
LARC SCIENTIFIC COMPILER

Phase III

FUNCTION

Any other 34 items? Yes → 004

Any 32 items? Yes → 002

NAME → CRLABL

A → c', 339Y → XXX no. of args. → D

3NAME

339Y

I → C

(CRDREF) → UFFFF of mode word,
mode word → CRFUNC
&NAME → NAME

RESULT 1

Multiple function statements.

Subroutine and function
LARC SCIENTIFIC COMPILER  Phase III

ID₃₅  SAL EXITS

335x

ID₃₅.0  →  3ID99

335v

ID₃₅.1  →  3NEXTI

3SUXX

335c

ID₃₅.2  →  3NEXTI

2   →  081

set  →  080

3NEXTI

ID₃₅.3  →  3SUXX

Output Successor item 16 F4

set  →  316TCP

ID₃₅.4  →  3XXX

335F

ID₃₅.5  →  082  →  3NEXTI

079  Unassigned variable in SAL exit list.

081  Constant in SAL exit list.

080  Parameter in SAL exit list.

082  Format in SAL exit list.
LARC SCIENTIFIC COMPILER Phase III

ID38  CONSTANT

335x
ID38.0  2 --> c' word 3 --> VALUE -> SF2  --3ID99

335y
ID38.1  083  --3ITEM4

335c
ID38.2
VALUE --> N
N:word3 = 3ITEM4

ID38.3  3SUXS

ID38.4  3XXX

ID38.5  3NOT

3SUXS  SUCCESSOR ITEM

S5555.99999, 03 --> L
020000 (seq. continue) --> M
0 --> N

083  Variable and constant have same name.

095  Multiple defining items for constant.
Parameter and expression have same name.

St. name and expression have same name.

Constant and expression have same name.

Format and expression have same name.
SUBSCRIPTED ARGUMENT OF CALL

341x

341V

341c

341s

341f

Non-subscripted reference to dimensioned variable.

Subscripted reference to statement name.

Subscripted reference to parameter or constant.

Subscripted reference to format name.
ARGUMENT OF FUNCTION

342x

342v

342c

342s

342f

\( ID_{42} \) Argument of function in common statement. 

\( ID_{42.0} \) Argument of function in dimension statement. 

\( ID_{42.1} \) Argument of function is statement name. 

\( ID_{42.2} \) Argument of function is format name. 

\( ID_{42.3} \) Argument of function is format name. 

\( ID_{42.4} \) Argument of function is format name. 

\( ID_{42.5} \) Argument of function is format name.
ID_{44} \text{ BUILT-IN OR LIBRARY FUNCTION}

ID_{44.0} \rightarrow \text{any refs?} \rightarrow \text{Yes} \rightarrow 00 \rightarrow \text{ID} \rightarrow 9 \rightarrow \text{C!} \rightarrow 347x1 \rightarrow XXX \rightarrow SF \rightarrow 3\text{NEXTI}

ID_{44} \rightarrow 349x2 \rightarrow \text{ID! 47,25} \rightarrow 47 \rightarrow \text{ID} \rightarrow \text{Seq} \rightarrow \text{DP} \rightarrow N \rightarrow 3\text{ITEM4}

ID_{44,1,23,4,5} \rightarrow 00 \rightarrow \text{ID} \rightarrow 3\text{NEXTI}

\text{Name used differently from function.}

084 \text{ Improper reference to library or built-in function.}
Limit of D∅

Limit of D∅

Word 3 → N

Value → N
O3 → ID

St. name used as limit of D∅

Format used as limit of D∅
Phase IV performs the following major tasks.

1. Determines relative locations of all variables in memory.

2. Solves the system of linear equations provided by EQUIVALENCE statements.

3. Determines the mode of arithmetic expressions and attaches it to operations and functions within the expression.

4. Updates the definition points of common and equivalent variables.

5. Determines and forwards to Phase V a great deal of information about DO-loops and lists.

6. Converts subscripts to single expressions.

7. Performs the numeric arithmetic in reasonably uncomplicated subscripts and splits or puts their terms into three parts.

8. Computes for Phase VII the number of elements in implicit arrays.

9. Creates all of Files 50, 51, 52 and 60, and part of Files 61 and 94. (Output to File 60 and File 61 is in F4 format. Appendices describe the other files).

10. Performs several error checks on the source-program-generated items.
File 4 contains nine classes of source language statements in the order below. (Within each class source language ordering is preserved, unless otherwise noted).

A. FUNCTION or SUBROUTINE statement if one is present.

B. LIBRARY calls

C. PARAMETER statements

D. One item for each variable not appearing in a DIMENSION, EQUIVALENCE, or COMMON statement, or as a dummy argument of a FUNCTION or SUBROUTINE statement; hereafter called "simple variable" items. (Alphabetic order).

E. DIMENSION statements.

F. EQUIVALENCE and COMMON statements.

G. CONSTANT statements.

H. Arithmetic statement functions.

I. Executable statements, including:

1. Arithmetic statements
2. Induction variable items
3. Control statements (IF's, GO TO's, input-output statements, etc.).
4. CALL statements.

The treatment of each class of statement by Phase IV is indicated below.
Phase IV operates on File 4, which is sorted on the first word of each item. The general form of the items in File 4 is:

Word 1:  
- Digits 1 - 5   Sequence number
- Digits 6 - 10  Subsequence number
- Digits 11 - 12 Item identification

Word 2:  Mode word for the item

Word 3:  Special information depending on the item type.
  a. A numeric value (for a constant item).
  or  b. A "definition point" (D. P.) (for a variable).
  or  c. The level (for an induction variable reference).
      etc.

File 4 was formed by Phase I and Phase III. Items in File 4 that represent punctuation, arithmetic operators, numeric quantities in the source program, and executable LSC statements are filed by Phase I, the rest by Phase III. Phase III creates several new types of items for File 4, as well as substituting constant items for all parameter references. The new items conform to the above three-word format. In most cases, the item identification is that assigned by Phase I.

Output from Phase IV goes to Phases V, VI, and IX. When F4 is thoroughly processed, Phase IV operation is complete.

Note: Phase IV has its own contingency routine.
A. FUNCTION or SUBROUTINE Statement (if any).

Subprograms differ from main programs in that the memory locations of parameters to the subprogram cannot be known at compile time. Later phases of LSC produce coding to generate these locations from the parameters given in the main program's calling sequence.

Phase IV assists in this generative process by sending to File 94 items defining each parameter as 0. (See Appendix III for formats of items sent to File 94).

Example:
Source Statement: FUNCTION F(X, Y)
Object Statements: DEF X: 0
DEF Y: 0

B. LIBRARY CALLS

A library call item is output to F94 for each item.

C. PARAMETER Statements.

Phase IV sends a raw-code item to File 94 for each parameter.

Example:
Source Statement: PARAMETER N=30
Object Statement: DEF N: 30

Since the value of a parameter is substituted for its symbolic name throughout the program, the definition of parameters is not necessary; it is added as a convenience to users who may want to refer to the parameter in SAL coding embedded in the LSC code.
D. SIMPLE VARIABLES

Phase IV makes storage assignments for all variables mentioned in the source program (except CONSTANT's, subprogram parameters, and COMMON variables in subprograms).

The storage is reserved by Phase IX. Phase IV sends out an item giving the total size of COMMON and of non-COMMON data. (The COMMON item is not output for subprograms).

Each variable is then defined by a DEF item to fall in COMMON or non-COMMON storage, whichever is appropriate.

1. DIMENSION statements produce DEF's for non-equivalent non-common arrays.

2. EQUIVALENCE statements produce DEF's for all equivalent variables.

3. COMMON statements produce DEF's for all non-equivalent common variables.

4. The remaining variables are defined by Phase IV on receipt of a group of "simple variable" items from Phase III. The "simple variables" are defined in alphabetic order and are in alphabetic order in memory when the program is run.

The leading statements of the object code thus provide the user with a complete storage map for variables in his program.
E. DIMENSION STATEMENTS

Phase III places in the mode word of each array item its relative dimension table reference. Phase IV creates the dimension table and files all dimensions in it for use in later Phase IV processing.

DIMENSION statements also have reservative force. DEF's for "ordinary" arrays go to FILE 94. Also, each array name is defined in terms of an LSC-created symbol which represents the actual location of the array in storage, again by F 94 items.

Note: The location of the array contains its first element (1, . . . , 1) while the name of the array refers to the non-existent element (0, . . . , 0); that is, the array X begins in the storage location (1, . . . , 1) beyond the location named X. This substitution of variable is done in Phase IV to speed up later processing and enhance the readability of the LSC output.

F. EQUIVALENCE and COMMON STATEMENTS

These two types of statements are intermixed. Since EQUIVALENCE statements cannot be processed until all the EQUIVALENCE relations are known, the subroutine which processed COMMON variables is called by the EQUIVALENCE processor.

1. Phase IV Treatment of EQUIVALENCE Statements

Each statement contains one or more chains of variables. A set of interrelated chains is here called a class. The processing as as follows:
a. Two internal files are created, one containing an item for each chain; the other, an item for each variable.

b. The terms of the subscript of each variable are multiplied by its dimensions to yield a single subscript. (If any term is zero, 1 is substituted and an error is output). The dimensions of the variable are multiplied to yield its total size. If the variable is COMMON, the chain is marked COMMON.

c. All previously processed variables are searched. If a match is found in a chain of the same class, the current item is either redundant or in error. The two subscripts are compared for error and the current item is dropped. If a match is found in a different class, the difference of the two subscripts is used to "relativize" all chains of the later class to the earlier class.

d. After all variables have been scanned, the subscripts within each class are relativized to a mutual point. The largest subscript and the largest inverse subscript (distance from the mutual point to the end of the array) are saved.

e. During the final scan, Phase IV computes the size of each class (largest subscript + largest inverse subscript) and the start of each class. It files in F94 an item for each variable giving
its category (COMMON or not) and its distance from the beginning of its category

2. **Phase IV Treatment of COMMON statements**

Variables which appear in EQUIVALENCE statements are ignored (they have been processed already). DEF items are filed in F94 for the others.

G. **CONSTANT Statements**

Both halves of the constant and its mode word are forwarded to F94 for output.

H. **Arithmetic Statement Functions**

Phase IV ignores the left sides of arithmetic statement functions. The right side begins with a mode-of-expression item and it's analyzed by the Expression Analysis Subroutine EXAN described in Appendix I.

While subscripts are permissible in ordinary arithmetic statements, they are not in arithmetic statement functions; if one appears, EXAN outputs an error.

I. **EXECUTABLE STATEMENTS**

When the first executable statement is encountered, the two data-size items (above) are output. To aid Phase VI and VIII, a continue item goes to File 61 which makes the first executable statement the beginning of a block of code.
Executable statements are of many types, and the types are intermixed unpredictably. Phase IV processes input from File 4 in groups, each group containing all items with identical sequence numbers as assigned by the previous phases. The identification of the least item of the group determines the path of processing.

1. Arithmetic Statements: Left Side

The EXAN subroutine processes subscripted variables on the left and in input-output lists, the entire right side of arithmetic statements and arithmetic statement functions, the arguments of CALL statements and the operands of control statements.

Each variable on the left appears in a group by itself. Preceding and following the list of variables on the left are "list" items. Phase IV attached the arithmetic mode of the expression to these items.

a. If the variable is equivalent, it is treated as a definition point for all equivalent variables. (Treatment of definition points in Phase IV is described in Appendix II, TEST 1 subroutine).

b. If the variable is unsubscripted, Phase IV asks whether it represents an undimensioned variable or an implicit array, if the latter, the number of elements in the array; if attached to the item.

c. If the variable is subscripted, EXAN analyzes the subscript.
d. If the variable is an array scanned by an explicit induction variable, it appears in File 4 as a DO loop group (q.v.) where the group contains only the variable within its range.

2. Arithmetic Statements: Ride Side

The leading item in a group containing an arithmetic expression determines the processing as follows:

a. \(+, -, *, /, \text{ or } **\). Go to EXAN.

b. Colon. Colons also precede CALL's and subscripted variables on the left side and input-output lists. Treatment of these cases is explained elsewhere. Unexceptional cases are handled by EXAN.

c. Mode of Expression Items. Phase III outputs one such item corresponding to each variable on the left. Phase IV examines them; if any is double precision, the mode of the expression is set to double precision; if not, it is set to single precision. EXAN then analyzes the expression, attaching the mode to operations and functions.

3. CALL Statements

Because the called subprogram may redefine any variable in COMMON storage, the CALL statement is considered a definition point for all COMMON variables (see TEST 1, Appendix II). At each CALL, Phase IV arranges for Phase V to compute and store all current common induction variables.
The arguments of the CALL are processed by EXAN. If statement names appear as the arguments of CALL's, Phase III will have generated successor items and attached them to the CALL group. These are sent unchanged to File 60.

4. Control Statements

DO statements are described separately below. The operands of all other control statements are processed by EXAN.

5. DO Statements

Note: Explicit arrays in input-output lists and on the left of arithmetic expressions are treated as DO loops after Phase I.

Phase I and III generate and Phase IV receives the following items:

a. At the beginning of each loop:

1. A 19 item, containing:

   A. The mode word of the induction variable.

   B. The sequence number of the loop's end.
2. The initial, final, and incrementing values of the induction variable (constant or variable). These values are forwarded immediately to F51, along with the dictionary reference of the induction variable.

3. TO and FROM items for possible transfers of control to or from the current loop, except transfers to the same loop which do not bypass inner loops.

b. At the end of each loop:

A 20 item marking the end of the loop and containing the mode word of the induction variable.

Throughout Phase IV, a table is kept in memory of current loops. An entry is made in the table at the beginning of each loop; the entry is output at the end of the loop.

Each entry contains:

a. The sequence number at which the loop begins.

b. The sequence number at which it ends.

c. Whether or not the induction variable is COMMON.

d. Whether or not the loop has the same range as the next outer loop.
e. Whether or not there are entrances to the loop, i.e., FROM items from outside the loop's range.

f. Whether or not there are exits from the loop, i.e., TO items to outside the loop's range.

g. Whether or not the loop is optimum as an outer loop, i.e., the only transfers to the loop are from nearby in the same loop.

h. Whether or not it is optimum as an inner loop, i.e., has no exits to outside its range.

i. Whether or not coding must be generated to compute and store the induction variable, i.e.,

1. It is used in arithmetic, or in a control statement.

2. It is COMMON and a CALL exists within its range.

3. SAL coding exists within its range.

4. Its loop contains an exit.

j. The new name of the induction variable.
   To facilitate Phase V processing, each induction variable is "renamed" according to
its nest number and level number, where these are defined as follows:

1. The nest number (0–999) begins at zero and increases by 1 each time one or more loops end. There are as many "nests" in this sense as there are innermost loops.

2. The level number (0–99) begins at zero, increases each time a loop begins and decreases each time one ends; it represents the number of current loops at any point. 100-level is the L part of the new name.

Items a through i are forwarded to F51 only when the loop is complete. At that time, also, an error is output if there exist one or more outside entrances to the loop but no outside exits. Also, the loop is called non-optimum if it is at Level 1 or if the next outer loop is non-optimum as an outer loop.

**Note on Optimum Loops**

When a subscript contains induction variables from two successive loops, Phase V will attempt to combine the increment of the outer variable with the initializing of the inner variable if the compiler can guarantee that the execution of one of these terms necessitates the execution of the other; if so, the inner loop is called "optimum".
The inner loop is optimum only if it is optimum as an inner loop and the next loop out is optimum as an outer loop.

The flow of control can be quite complex. Phase IV does not do a flow analysis; but where there is doubt it errs on the side of safety.

5. SAL Code

The presence of SAL code is signalled by one item in F4. When such an item is encountered, the SAL block item output by Phase III is changed to a continue item for Phases VI and VIII. All current induction variables are marked compute and store.

Phase IV -- Appendix I -- EXAN (Expression Analysis Subroutine)

A. Treatment of Expressions by EXAN

Expressions treated by EXAN arise in a number of different types of LSC statements, as described by the writeup. All expressions in File 4, however, consist of sequences of operators and operands in "Polish" notation, each operator requiring two operands.

B. Processing in Phase IV consists of:

1. Attaching definition points to variables as described in Appendix II.

2. Attaching the mode of the expression to operators and functions not within subscripts for use by Phases VI and VII.

3. Recording uses of induction variables (except in simple subscripts) for Phase V (see DO loops).
4. Deleting subscripts within subscripts and subscripts within arithmetic function statements (these are errors).

5. Breaking down subscripts into several parts, testing the parts and sending them to Files 50, 52, and 60, as described below.

C. Treatment of Subscripts by EXAN.

Each subscript consists of from 1 to 9 expressions or terms, terms being set off by commas in the source language. Except that terms may not contain subscripts, each term is itself a full arithmetic expression, as simple as a single constant or as complex as a nest of 400 functions.

By multiplying each term by the product of the dimension of the previous terms, Phase IV converts the source language subscript

\[ T_1, T_2, \ldots \text{ into the single arithmetic expression} \]
\[ T_1 + D_1 \cdot T_2 + D_1 \cdot D_2 \cdot T_3 + \ldots \]

(where \( D_1, D_2, \ldots \) are the dimensions of the subscripted variable).

If the wrong number of terms is present, Phase IV outputs an error item and processes only those for which dimensions have been given.

As it creates a single expression from the subscript, Phase IV breaks the expression into three parts.

1. A constant part, This, and the name and sequence number of the subscripted variable, go to File 50. The two are renamed to become the M-address of the output SAL code.
2. An induction variable part. Each induction variable, its numeric coefficient, and the sequence number go to F52. The induction variables are included in the B-register of the output code.

3. A remainder. The remainder contains ordinary variables, functions, and the part of the subscript, if any, which is too complicated for Phase V processing. The remainder goes to F60, and is included in the B-register of the output code.

**EXAMPLES:**

A has dimension (2, 3, 4)

a. File 1
   A (1, 2, 3)  
   1 + 2 (2) + 3 (6) = 23

File 2
   : A , , 1 2 3

File 50
   + 23 A

Files 60-61
   : A ' 0

Files 7
   A'

b. File 1

File 4
   : A , , I J K

File 52
   1I, 2J, 6K; n = 0

Files 60-61
   : AB

B is a B-register name.

c. File 1
   A (X, Y, Z)  no induction variables.

File 4
   : A , , X Y Z

Files 60-61
   : A : CX + X + *2 Y * 6Z

   CX = "convert to fixed" function.

d. File 1
   A (1, J, Z)  J and Z as above

File 4
   : A , , 1 J Z

File 50
   + 1 A

File 52
   2 J ; n = 1

Files 60-61
   : A fx+ * 6 Z B

fx+ = fixed point add operation
D. Treatment of a Single-Item Subscript-Term by EXAN

1. Numeric. The term is multiplied by p (Product of dimensions to that point) and added to total constant for the subscript. An error is written if it is larger than the corresponding dimension, or less than 1.

2. Induction variable. The product p is added to the coefficient for the appropriate induction variable. (Coefficients for all current induction variables are set to zero before processing of the subscript begins).

3. Ordinary variable, or function without arguments. The SAVE subroutine saves the item and its coefficient.

E. Treatment of Multiple-Item Subscript Terms by EXAN

The processing is in three passes, as follows:

1. Scan forward, creating an internal file, one item for each item of the term.

   A. Operation items include the type of operation and the addresses of the operands.

   B. Numeric items contain their value.

   C. Variable items contain coefficients of 1 and whether or not induction variable.

   D. Terms containing division, exponentiation, a function with arguments, or more than one non-induction variable are not broken down. Instead, the term goes to F60, preceded by +, *, and p items.
2. Scan backwards, examining operations only.

A. Addition. Perform addition of numerics.

B. Subtraction. Perform subtraction of numerics. If either operand is non-numeric, multiply the terms of the second operand by -1.

C. Multiplication. Perform multiplication of numerics. If only one is numeric, multiply the terms of the other by it. If both are non-numeric, send the complete term to F60 preceded by +, *, and p items.

3. Scan forward, examining operands only.

A. Numeric. Multiply by p and add to term's constant. If zero, ignore.

B. Induction variable. Multiply coefficient by p and add to induction variable coefficient table. If the coefficient is negative, note special situation.

C. Other variable or function without arguments. The SAVE subroutine saves the item.
Source Statements: $A = B(\ldots, (4*3)-(2+1), \ldots)$

Treatment of the term: $(4*3) - (2+1)$

F4 input: $- * 4 3 + 2 I$

1st Pass Table:

<table>
<thead>
<tr>
<th>Entry</th>
<th>0:</th>
<th>-</th>
<th>1, 4 (operand addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>*</td>
<td>2, 3</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>2:</td>
<td>const.</td>
<td>4, 0</td>
<td>(coefficient)</td>
</tr>
<tr>
<td>3:</td>
<td>const.</td>
<td>3, 0</td>
<td>&quot;</td>
</tr>
<tr>
<td>4:</td>
<td>+</td>
<td>5, 6</td>
<td>(operand addresses)</td>
</tr>
<tr>
<td>5:</td>
<td>c</td>
<td>2, 0</td>
<td>(coefficient)</td>
</tr>
<tr>
<td>6:</td>
<td>I</td>
<td>1</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

2nd Pass:

Scanning backward the first operation is + in (4). Its operands (5) and (6) are both single but not both numeric. No change. Next operation is in (1). Both operands are numeric; so the constant designator c goes to (1); 4*3 goes to coefficient (1); "ignore" mark goes to (2), (3). - in (0) has one non-numeric operand. Its terms (2 and I) go to a new table, the coefficients of which are multiplied by -1; so that -2 goes to coefficient (5); -1 to coefficient (6).

3rd Pass:

The table looks like this (ignoring operations):

| W0   | ignore |
| W1   | constant 12 |
| W2   | ignore |
| W3   | ignore |
| W4   | ignore |
The constants 12 and -2 are added to the total constant for the subscript.

If I is an induction variable, -1 is added to its coefficient so far. If not, the SAVE subroutine files it.

F. Post-Processing of Subscript Terms

After a decomposable term is processed, a note is made if it is potentially negative. This is necessary because the LARC represents numbers in signed magnitude form but does not consider the sign of the index register in indexing.

2. Induction Variable Part.

An item goes to F52 for each current induction variable with non-zero coefficient.

3. Non-Induction Variable Part.

The table set up by the SAVE routine is scanned and its entries output. If it is not empty, items causing fixing of its components are output.

4. Potentially Negative Subscripts.

If part of the subscript goes to F52, the note of negativity goes with it. If not, "+100000" goes to F60. Thus, the M-field of the generated B-register is biased so that it cannot go negative.

The command

\[
F \ #0 \ A+20 \ #1
\]

works identically for the two cases

\((#1) = 1 \text{ and } (#1) = -1.\)
LSC adds the constant portion of a subscript to the M-address of the object code: the rest of the subscript is contained in the referenced B-register. LSC must, therefore, insure that no B-register is negative when it is referenced.

LSC assumed that each term of the subscript is positive and, therefore, only those terms which Phase IV decomposes can possibly create negative B-registers. In fact, a negative B-register can result only if the term contains a constant greater than zero. It must also contain a negative induction variable or another factor or factors non-analyzable by Phase IV.

Phase IV also checks whether a decomposed term contains only induction variables and a positive constant greater than the corresponding dimension. This is an error.

G. Post-processing of Subscripts

1. Constant Part

   If the total constant is zero, the item for the subscripted variable goes to F 61. If not, the variable and its constant go to F50.

   If the subscript is all constant, a zero goes to F60 as the subscript.
   (Phase VI eliminates zero subscripts.)

Phase IV--Appendix II--Subroutines

A. TEST1

TEST1 processes an item in an arithmetic expression and returns to the calling sequence at one of 5 exits as follows:

1. Constant
2. Induction Variable
3. Unsubscripted Variable

The parameter number of the variable, if any, is moved to the mode word. The d-digit of the mode word is set to zero.
4. Subscripted Variable

After the above, d is set to 1.

5. Operation

Operation and functions are marked with the mode of computation of the expression. Functions are then treated as variables.

To eliminate unnecessary recomputation of arithmetic quantities by the object program, Phases III and IV attach to each non-induction variable a definition point, representing the last possible change in the variable's value.

TEST 1 modified the Phase III assigned definition points of common and equivalent variables only. The phase keeps a running record of the definition points for those special classes of variables.

The EQUIVALENCE point is initialized when the first executable statement is processed, and replaced by the current sequence number whenever an equivalent variable is encountered:

1. On the left of an arithmetic expression.
2. In an input list.
3. As an argument of a CALL.
4. As the induction variable of a loop.

The COMMON point is updated at each CALL statement.

B. TEST2

TEST2 tests all limits of loops. Its primary function is to update the definition points of common and equivalent variable limits.
C. SETZ

This subroutine, which is called by TEST2 (see below) and by some of the calling sequences to TEST1, simply records the necessity to compute and store an induction variable by setting a digit to 1 in the table for that variable.

D. SAVE

This routine sets up a table of non-induction variables in subscripts and their coefficients. (A maximum of 9 entries).

It insures that the source statements

\[
\text{DIMENSION} \quad A(4, 5, 6) \\
B = (A (X, X, X)
\]

produce this code

\[
\begin{align*}
F & \quad #0 \quad (.25 \ (02) ) \\
M & \quad #0 \quad X \\
CX & \quad #0 \quad 61 \\
F & \quad #0 \quad A \ #0 \\
S & \quad #0 \quad B
\end{align*}
\]

not this code

\[
\begin{align*}
F & \quad #0 \quad (.20 \ (02) ) \\
M & \quad #0 \quad X \\
F & \quad #1 \quad (.4 \ (01) ) \\
M & \quad #1 \quad X \\
A & \quad #0 \quad #1 \\
A & \quad #0 \quad X \\
CX & \quad #0 \quad 61 \\
\text{etc.}
\end{align*}
\]

E. ITEM

Given a coefficient, R, and item, K, and a sequence number, S, this routine outputs to File 60 on S the items

\[
* \quad R \quad K
\]

If this is not the first call to ITEM within a given subscript, a + item is put out on the sequence left from the last call.
Phase IV--Appendix III--Output to F50, F52, F52

---

**F50**

<table>
<thead>
<tr>
<th>4-word item</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
</tr>
<tr>
<td>W2, 4</td>
</tr>
</tbody>
</table>

- c : fixed point constant
- leading digit = 0 or 1 for + or -
- : dictionary reference of variable

---

**F51**

<table>
<thead>
<tr>
<th>4-word item, 4 items per loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 1</td>
</tr>
<tr>
<td>W1</td>
</tr>
<tr>
<td>W2</td>
</tr>
<tr>
<td>W3-4</td>
</tr>
<tr>
<td>Items 2-4</td>
</tr>
<tr>
<td>W2</td>
</tr>
<tr>
<td>W2-4</td>
</tr>
</tbody>
</table>

- I : new induction variable name
- D : sequence of start of loop
- E : sequence of end of loop
- t to z : indicators
- j : subsequence 1, 2, 3, of limits
- F : dictionary reference of I

---

**F52**

<table>
<thead>
<tr>
<th>2-word item</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
</tr>
<tr>
<td>W2</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>n</td>
</tr>
</tbody>
</table>
Phase IV--Appendix VI--Output to F94

F94 4-word item

W1 (Sequence)
W2 (I.D.)
W3 (In general, left of =)
W4 (In general, right of =)

Note: For ID = 11, second word contains abcdeuffffID

<table>
<thead>
<tr>
<th>ID</th>
<th>Implied by I.D.</th>
<th>Word 3</th>
<th>Implied by I.D.</th>
<th>Word 4</th>
<th>Source Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>DEF</td>
<td>F</td>
<td>: G -</td>
<td>K</td>
<td>DIMENSION</td>
</tr>
<tr>
<td>3</td>
<td>DEF</td>
<td>G</td>
<td>: 9 DAT+</td>
<td>K</td>
<td>DIM, EQU I</td>
</tr>
<tr>
<td>4</td>
<td>DEF</td>
<td>G</td>
<td>: 9 COM+</td>
<td>K</td>
<td>EQUIVALENCE</td>
</tr>
<tr>
<td>5</td>
<td>DEF</td>
<td>G</td>
<td>: 9 COM+</td>
<td>K</td>
<td>COMMON</td>
</tr>
<tr>
<td>6</td>
<td>DEF</td>
<td>F</td>
<td>: 9 DAT+</td>
<td>K</td>
<td>EQUI</td>
</tr>
<tr>
<td>7</td>
<td>DEF</td>
<td>F</td>
<td>: 9 COM+</td>
<td>K</td>
<td>COMM, EQUI</td>
</tr>
<tr>
<td>8</td>
<td>$STOR</td>
<td></td>
<td>: 9 DAT</td>
<td>K</td>
<td>PARAMETER</td>
</tr>
<tr>
<td>9</td>
<td>DEF</td>
<td>F</td>
<td></td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$STOR</td>
<td></td>
<td>: 9 COM</td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Reserve k-</td>
<td>1st half</td>
<td></td>
<td>2nd half</td>
<td>CONSTANT</td>
</tr>
<tr>
<td></td>
<td>storage for F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>LIB</td>
<td>F</td>
<td></td>
<td></td>
<td>LIBRARY</td>
</tr>
</tbody>
</table>

F = contents of dictionary reference f.
G = symbol generated by Phase IX from f.
K = fixed-point constant
CONTROL

Sort F4 on sequence, subsequence

Q4

open F4 (R)
open F60, F94 (W)
extend F92 (W)

O → NC
O → CC
1 → SU
CRS → EXC
trap location 2701
set FF15

AA
F4 group [S]
None

AB
Close F50, F51
F52, F60, F61

Reset loc. 2701
Reset FF15

CONTROL

(R) = file is to be read
(W) = file is to be written
CRS = sequence of first executable instruction.
EXC = statement name of first executable instruction.
NC = count of storage for not-common variables
CC = count of storage for common variables

location 2701 is trapped and FF15 is set so that Q4 can do its own analysis of contingencies.
\( QP = \) definition point of equivalent variables.

\( CP = \) definition point of common variables.

\( NL = \) nest and level numbers of current induction variable.

\( N = 000 - 999 \)

\( L = 00 - 99 \)

\( CRMEMS = \) memory available for data storage.

Too much storage required.
$b$ and $f$ are mode word fields. $n$ is the numeric value of the parameter.
LARC SCIENTIFIC COMPILER

Phase 4

DIMENSION

DA
1 → t (total size)
0 → u (offset)

≠
double b: precision

= t + t → t

DK
u + t → u
t + kn → t
n → T3e
e + i → e

<
n ! CREMENS

≥ 8

F4 'item
none

fixed t; available memory

≤
a : 9

≥ AA
4.19

≠

DD

≤ 9

1

Too much storage required

8

Subscript too large

9

Subscript too small.

T3 = dimension table
a ≠ 0 for common, equivalent or subprogram argument.
Subroutine to assign storage to common variable.

Subroutine to multiply the dimensions of a variable.
TS = table of variables.

**TS item**

<table>
<thead>
<tr>
<th>SE</th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>subscript 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total size t

W scans the table.
SE = major sequence.
d = # dimensions.
f = dictionary reference.

**T4 = table of chains.**

**T4 item.**

<table>
<thead>
<tr>
<th>c</th>
<th>0</th>
<th>0</th>
<th>H</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T scans the table.
c = 0 chain not common
- chain common

H = relative location of first chain in this class.
W = relative location of first variable in chain.
ms = largest subscript in class.
mt = largest t! (inverse subscript) in class.
r = relative size, i.e., distance in memory between chain and first chain in class.

To second pass

4.27
Equivalence (continued): Compute Subscript

\[ p = \text{product of dimension so far}, \]
\[ d' = \# \text{of terms given}, \]
\[ d = \# \text{dimensions stated}. \]

**Phase IV**

**LARC SCIENTIFIC COMPILER**

2: Wrong number of subscripts.

3: Subscript too small.

8: Subscript too large.
EQUIVALENCE (continued): Search chain subroutine.

This subroutine compares the current variable to each previous variable. If a match is found, the latter class is made part of the former.

Inconsistent uses of EQUIVALENT variable.
This pass marks the class items (first chain item for each class) common or not, relativizes every variable in the class to a single point, and computes the maximum subscript and inverse subscript for the class.
EQUIVALENCE (continued): Third pass.

This pass outputs a DEF item to F94 for each EQUIVALENT variable.

---

Diagram details and corresponding text are not directly transcribed due to the complexity and the nature of the diagram. It involves a flowchart with various nodes and edges, indicating the flow of data or actions in a process related to compiler equivalence and variable handling.
Subscript within arithmetic function statement.
EXAN (continued): Initialize Term Processing.

- PL = 0, no terms to F60
- p = product of dimensions.
- ac = # terms written
- d = # terms required
- i = # terms to process
- j = # terms to ignore
- c = total constant for subscript
- T6 = table of induction variable coefficients.
- T6o = 0 B register cannot be negative.

Wrong number of subscript terms.
EXAN (continued): Process Simple Term.

- Term too small
- Term too large.
EXAN (continued): Process Complex Term.

Subscript within subscript.
1. **First Pass.**

The first pass creates the \( W \) table, which contains one item for each item in the subscript term, as follows:

<table>
<thead>
<tr>
<th>operation</th>
<th>( T )</th>
<th>( T_w )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1, 2, 3 for ( +, -, \times )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( B )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( A )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>numeric</th>
<th>( T_z ) = floating pt. 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( C_z )</td>
</tr>
<tr>
<td></td>
<td>( X )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>variable</th>
<th>location of item</th>
<th>( T_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>numeric</td>
<td>( C_v )</td>
</tr>
<tr>
<td></td>
<td>( D )</td>
<td></td>
</tr>
</tbody>
</table>

\( D = 0 \Rightarrow \text{induction variable.} \)

2. **Second Pass.**

The second pass consists of an inspection from back to front of the operations in the table. Constant arithmetic is done; the coefficients of variables are computed.

3. **Third Pass.**

Nothing remains in the table but additive constants and additive variable terms with coefficients. These are output.
EXAN (continued): Term First Pass.

Diagram description...
EXAN (continued): Term Second Pass.
pc = constant part of term.
LARC SCIENTIFIC COMPILER

Phase IV

EXAN (continued): Post-Analysis of Subscript.

Diagram showing flow of operations for a scientific compiler phase IV.
This subroutine sends a variable or punctuation item to F60.
The form of A(X+2Y+I) in G6 is:  
\[ A^{FX} : CX + X \ast 2YI' \]
where \( ^{FX} \) represents fx. pt. addition, and \( I' \) is the A register containing I.

This subroutine scans a Polish string without operating on it.
Notice of Compliance

LARC SCIENTIFIC COMPILER

Phase IV

4.44

Dφ LOOP (continued): Limits.

I, j, F → lowest item
I item → T9L

F4 item

Test 2

→ FSI1: b item

F4 item

Test 2

→ FSI1: c item

F4 item

Test 2

→ FSI1: d item

H4

10-way branch on a

Test 2

0 → EXIT

1 → D → DP → EXIT

2

3

4 → set 2 → EXIT

5 → LSCER

6 → LSCER

7 → LSCER

8 → LSCER

9 → parameter # → d → EXIT

EXIT
DF LOOP (continued): TO & FROM ITEMS.

Illegal transfer from this loop.
END DO

L - 1
get L item

NS
1
0
NS
N+1

J A
L: 1
\ne

X_{L-1}: 0
\ne

J B
\# 0 \rightarrow Y_L

J C
V_L: 0
\ne

W_L: 0
\ne

11

JD
next group end DO?

JE
\# 0 \rightarrow u_L

L item \rightarrow FS1

AA 4.19

SAL BLOCK

\rightarrow F60: block header.

F4 item

Item \rightarrow F60

L \rightarrow K

K: 0
\ne

\# 0 \rightarrow Z_K

K \rightarrow 1

AA 4.19

INDUCTION VARIABLE

\# 0 \rightarrow Z_K

group \rightarrow F60

AA 4.19

\text{illegal transfer to this loop.}
PHASE V

PURPOSE:
The main function of Phase V is to optimize an object program with respect to manipulation of indexing quantities. This includes:

1. Renaming subscripted variables containing constants in their subscripts.

2. Creating code-generating items to form induction variables which must be made available to the program.

3. Creating code-generating items to form subscripts containing induction variables.

4. Creating code-generating items to count the number of times through each loop.

INPUT:
Prior to Phase V, Phase III has labeled references to induction variables in the range of the controlling loop as such, and Phase IV has collected a great deal of information about each induction variable, passing this information on in the form of Files 50, 51, and 52.

File 50 contains one item for each variable containing a constant in its subscript. F50 has nothing to do with induction variables; a program without any may still generate a large F50.

File 51 contains a group of items for each loop in the source program, including the old and new names of the induction variable, its range, its limits, exit-entrance information, etc.

File 52 contains an item for each reference to an induction variable in a subscript (unusually complicated subscripts excepted). The item contains the variable's new name, its constant coefficient, the sequence of the reference, and indication whether the subscript is potentially negative.
DATA FLOW:
F50 is read in and its item split to be sent to F61 and F91.

F52 is then input and used to form F53 and F54.

F53 is sorted and processed to form F55 and F56.

F55 is sorted. Then it and F56 are processed in alternating groups by F51 and F54.

When F51 is exhausted, Phase V is finished.

The forms of F50, F51, and F52 are described in an Appendix to the description of Phase IV. The internal files are described in Appendix A. The processing is described in detail below.

Note: Phase V has its own contingency routine.

PHASE Va: RENAME VARIABLES AND CONSTANT SUBSCRIPTS.

Example

Source statement: \[ A (5) = A (I+5) + B (2*J-1) \]

Object statements:

\[ \text{F50} \#0 \; \text{A}+5 \; \#1 \; (I) \]
\[ \text{A5} \#0 \; \text{B}-1 \; \#2 \; (2J) \]
\[ \text{S} \#0 \; \text{A}+5 \]

F50 contains the pairs A, +5; B, -1; A, +5; these become the M-fields of the object code.

Phase Va's function is to file each pair once in the dictionary (F91) and replace each F50 item for the variable (A, B) with an item for the variable-subscript combination (A+5, B-1).
A F50 group containing identical constants and variables is read in. The constant and variable are sent to F91. The dictionary reference is attached to the mode word of each F50 item, and the item is passed to F61. (The size of F50 and the number of dictionary words required is greatly reduced for most programs by the Phase IV device of starting subscripts from 0 instead of 1.) At the same time subprogram arguments containing negative constants are marked for Phase VII.

**PHASE Vb: NUMBER SUBSCRIPT TERMS**

**Example**

**Source statements:**

\[
\text{DIMENSION } A(3, 4, 5), B(3, 4, 5), C(3, 4, 5) \\
\vdots \\
A(I, J, K) = B(I+1, J+1, K+1) + C(I+2, J+2, K+2) \\
\text{where } I, J, K \text{ are induction variables.}
\]

**Object statements:**

\[
F \#0 B+16 \#1 \\
A \#0 C+32 \#1 \\
S \#0 A \#1
\]

Phase V insures that multiply-referenced subscripts (I, J, K in the example) are computed only once by naming "identical" subscripts alike. The names are sent to F61 to be merged into their proper places in the Polish string.

Each term of a subscript contains an induction variable and a coefficient (11 digits of information). The first step in eliminating duplicate subscripts is to condense the information given into four-digit numbers.

Input comes from F52 in groups by term. The term of the group is filed in F54; its number goes to F53 once for each reference to it.
F53 is sorted so that within each subscript, the terms appear "inside out."

Subscripts containing 1–3 terms go to F55; larger subscripts to F56.

F55 is sorted and now the major work of Phase V begins. All its files are formed; they must now be processed in interrelationship with each other. Output will be to F61 only in the form of both Polish strings and pseudo-code.

PHASE Vc: FILE LOOPS BY LEVEL

The loops shown have been numbered by Phase IV as follows:

\[
\begin{align*}
W & \quad 000\ 99 \\
X & \quad 000\ 98 \\
Y & \quad 001\ 98 \\
Z & \quad 001\ 97
\end{align*}
\]

Phase V will process loops W and X and the subscripts within them completely before beginning work on loops Y and Z. When loops Y and Z are processed, all information on loop W will still be available for use in forming subscripts containing terms from Y and W, Z and W, or all three.

Phase Vc reads F51 and files the 16 words for each loop with minor modifications in a table (where each remains until replaced by another loop of the same level or until Phase V is over.) At this point also 3 sequence counters are initialized for each loop:

- C (initialization outside the loop); D (incrementation inside the loop); and E (looping logic at the bottom of the loop).
PHASE Vd: UNPACK TERMS

The F54 items for the loop, if any, are input for use by the subscripts and unpacked into more convenient form by a scan.

PHASE Ve: FORM NON-CONSTANT COUNTS; MATERIALIZE INDUCTION VARIABLES

Each loop is now examined in turn. If its initial, final, and incremental values (b, c, and d, respectively in DØ S, I=b, c, d) are all constant, its count is constant. The fastest object code for counting and looping would load one or more A-registers with counts outside the loop, then loop with a BIT command. This is what Phase V does, as described later.

Unfortunately, this method is impractical for variable counts. Phase Ve produces instead code to initialize an A-register to (b-d) outside the loop and to add d and test against zero at the end of the loop.

If the induction variable must be made available to the program (the conditions are described in Phase IV), it is "materialized" by creating the instructions F #n b, S #n I, A #n d, at C, D, and E respectively. (See bottom of Vc.)

PHASE Vf: FORM NON-CONSTANT INITIAL AND INCREMENTAL MULTIPLES; CHECK ERRORS

The formation of initial \(a \cdot (b-d)\) and incremental \(a \cdot d\) multiples of subscript terms (where \(a\) is the coefficient of an induction variable in a subscript) is a space-and-time-consuming process (especially when, as is often the case, there is more than one \(a\) to be multiplied). Phase Vf effects the production of code to form these multiples as many levels before use as possible, and saves the "names" of the A-registers in which they are formed for later reference.

Phase Vf also evaluates the count if it is constant, checks that the limits of the loop are reasonable and saves optimization information.
PHASE Vg: FORM SUBSCRIPTS

The subscripts are now in F55 and F56 in slightly different forms. Associated with each subscript are its sequence and subsequence and the Phase Vb-assigned numbers of its terms. The indication whether or not it is potentially negative is still attached.

All F55 subscripts whose innermost terms contain the current induction variable are processed, then all F56 subscripts of this type. The processing for the files is similar:

1. The subscript is checked to see if it is contained in a current loop.
2. The terms are filed in a table.
3. The negative indicator is saved.
4. Item(s) are sent to F61 naming the subscript at each reference to it.
5. The terms are then processed by the subscript subroutine as described in Appendix I.

PHASE Vh: CONSTANT COUNTS

(A reading of Appendix I will clarify this section.)

Constant counts may not be formed before Phase Vh because:

1. In some cases two or more adjacent constant counts may be combined, but these cases can't be distinguished until after Phase Vg.

2. In some cases the BIT instruction which counts may also be incrementing a subscript provided by Phase Vg.

Phase Vh consists of a final scan of the loops input in Phase Vc, from the inside out, with attention to constant counts only.

Tests are made as described below to determine whether the loop can be collapsed with the next outside loop. If not, items to fetch and count down an A-register containing the count modulo 1000 and ad and ab fields, if any have been saved, are output. As many other A-registers as necessary are loaded and "BIT"-ed to complete the count. (The program takes account of the fact that setting up multiple BIT's is not straightforward when any of the partial counts is zero.)
If two loops can be collapsed, no items are output; the product of their counts becomes the count of the outer loop, and if the inner loop's BIT command was to have incremented a subscript, the outer's now will.

Two loops can be collapsed only if:

1. The inner loop is optimum.
2. Their counts are constant.
3. They include exactly the same executable statements.
4. All subscripts containing either induction variable contain both in such a proportion that one complete execution of the inner loop increments the subscript by the amount of one step in the outer loop.

After the last loop's count is inspected, the program returns to Phase Vc to process the next nest. When F51 is exhausted, Phase V is through.
Subscripts are formed in A-registers. Through most of Phase V a five-digit counter serves as a source of A-register names. The counter is initialized at the start of each level-one nest of loops. A-registers are named in such a way as to minimize use of index registers for arithmetic by Phase VIII.

Each subscript is compounded of terms of the form ad and a(b-d) (where b and d may be variable) and constants from four other sources:

1. 100000 - added at the first initialization of a potentially negative subscript.

2. The ad, ab parts of a counting A-register - added as that term's loop is executed.

3. The count of a counting A-register - fetched originally or added later.

4. The total increments of optimum loops with constant limits - added and subtracted as described below.

Like everything else in Phase V, subscripts are examined from the inside out, for only in that way can the optimacy of each loop in turn be checked.

The following examples show the operations produced for several subscripts, where the successive P numbers are successive A-register "names."

**Example 1: Single Term**

at C: \[ P1 = a1 (b1 - d1) \]

at D: \[ P1 = P1 + a1 d1 \]

**Example 2: Two Terms**

at C1: \[ P2 = a1 (b1 - d1) \]

at D1: \[ P2 = P2 + a1 d1 \]

at C2: \[ P1 = P2 + a2 (b2 - d2) \]

at D2: \[ P1 = P1 + a2 d2 \]
In the above examples, \( ad \) or \( a(b-d) \) may be variable; if so it is referred to by the name assigned its register in Phase Vf.

**Example 3: Two Terms, Optimizable**

"Optimizable" means:

1. Inner loop optimum.
2. Terms from 2 successive loops
3. \( d \) of outer loop constant and total increment \( t \) of inner loop calculable. \( t = dk \) where count \( k \) is constant.

<table>
<thead>
<tr>
<th>At C1</th>
<th>( P_1 = a_1 (b_1 - d_1) + a_2 (b_2 - d_2) + a_2 t_2 )</th>
<th>1 - 2 commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>At D1</td>
<td>nothing</td>
<td></td>
</tr>
<tr>
<td>At C2</td>
<td>( P_1 = P_1 + a_1 d_1 - a_2 t_2 )</td>
<td>1 command</td>
</tr>
<tr>
<td>At D2</td>
<td>( P_1 = P_1 + a_2 d_2 )</td>
<td>1 command</td>
</tr>
</tbody>
</table>

Formidable though the analysis seems, the commands generated are minimal, because all operands are A-registers or constants.

**Example 4: Completely General Subscript**

<table>
<thead>
<tr>
<th>At D, inner loop:</th>
<th>( P = P + ad )</th>
</tr>
</thead>
<tbody>
<tr>
<td>At C, optimizable loops:</td>
<td>( P = P + ad(outer) - at(inner) )</td>
</tr>
<tr>
<td>At C, non-optimizable loops:</td>
<td>( P = P' + (all \text{ waiting } a (b - d) s) + (BB, \text{ if any}) )</td>
</tr>
<tr>
<td>At C, outer loop:</td>
<td>( P' = (all \text{ waiting } a (b - d) s) + (all \text{ mt}'s added later) + (100000 \text{ if applicable}) )</td>
</tr>
</tbody>
</table>

The **Form Subscript Subroutine** forms code-generating items to do the work described in the examples. In doing so, it sends several new types of items to F61. (New items are described in Appendix B.)
A. Items in Phase IV form:

1. Fixed-point multiply operation:

   W1 - SSSSS  sssss 49  S = Sequence
   W2 - 00620  07000 00  s = subsequence
   W3 - 0———0

   Used to form ad, a (b-d) multiples.

2. Fixed-point add operation:

   W1 - SSSSS  sssss 49
   W2 - 00620  08000 00
   W3 - 0———0

3. Convert-to-fixed-point function:

   W1 - SSSSS  sssss 47  V = scale factor
   W2 - 70700  VV00013
   W3 - 0———0

   Used to fix d, b-d before forming multiples.

4. Integer in integer form:

   W1 - SSSSS  sssss 45
   W2 - 0420 000 0000
   W3 - integer

   Used for adding subscript terms to A-registers.
5.  BB literal

W1 - SSSSS sssss 45
W2 - 05200 00000 00
W3 - BB

Used for initializing counts of loops with constant counts.

B.  Pseudo-code items:

1.  Register name item

W1 - SSSSS sssss 58       A = A-register "name"
W2 - 0000001AAAAA
W3 - 0000000000

Used to name register in which following expression is to be calculated.

2.  Register Definition Item

W1 - SSSSS sssss 57
W2 - 0000001AAAAA
W3 - 000LL00TTTTT

Used as a signal to Phase VIII that an A-register has been defined.
LL = inverse level modulo 100.
T is zero except after BIT command; then it is the sequence to which the
transfer is made.

3.  Add -- same with ID = 52

Used to increment induction variable.

4.  Add fixed-point -- same with ID = 54

Used to increment subscript.
5. Negate fixed-point -- same with ID = 59.

6. Store -- same with ID = 53

Used to store induction variable in memory.

7. Transfer Less than Zero

W1 - SSSSS sssss 51  
W2 - 0000001 AAAAA  
W3 - CCCCC0000000

C = numeric part of statement  
name created by Phase IX

8. BIT -- same with ID = 56.

9. Store to fast register

W1 - SSSSS sssss 21  
W2 - 0  
W3 - 0002001AAAAA
FORMAT OF INTERNAL FILES

F53: SSSSS sssss LL  
     0000WWWW000n  
        L = level  
        n = negative indicator  
        W = name of term

F54: IIIII 0 aaaaaa  
        I = new name of variable  
        a = coefficient

F54 (unpacked):  
+ aaaaa 0 KKKKK  
        K = F51 table ref

F55: W1  W2  W3  
     SSSSS sssss 0n

F56: W1  W2  W3  
     -n SSSSS sssss

One or more a) items followed by one or more b) items, followed by a zero word.
LARC SCIENTIFIC COMPILER

Phase V

FS3 group [s.ss]

S.ss. n → item
item → F55

last
WWW
more

O → i

WWW: Ti

i + 1

WWW
last

WWW: Ti

i : Ti

W → Ti

i + 1 items → F56

n s.ss → F56

FS6 item

W₁ W₂ W₃

variable number

... 

-n s ss

variable number

0

end of "group"

FS5 item

W₁ W₂ W₃

s ss o n
**LARC SCIENTIFIC COMPILER**

**Phase V**

- **AG**
  - Close F55, F56 (W)
  - Sort F55 [W1, W2, W3]
  - Discard F56 item
  - Open F51, 54, 55, 56

- **Po → P (A-register name)**
  - 1 → W

- **J**
  - DJ
  - none

- **FS1 word**
  - level → J, L

- **FS1 group [I]**
  - → KJ

  - \( y_k \rightarrow K_0 \)
  - \( D_k \rightarrow K_1 (c) \)
  - \( D_k + 1.2 \rightarrow K_2 (d) \)
  - \( E_k \rightarrow K_3 (e) \)
  - \( J + 1 \rightarrow J \)

  - FS1 word

  - same nest?

  - yes

  - last group level 99?

    - yes

    - \( W \rightarrow \alpha \) (offset)

    - \( P_0 \rightarrow R \) (FS9 input)

    - AN

    - C

- **K Table Item**

<table>
<thead>
<tr>
<th>K Table Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

- **K8** : \( \alpha k \) for \( i = 0, y = 0 \)
  - otherwise 0
- **K12** : \( k \) for \( i = 0 \)
  - otherwise 0
- \( K_4, K_9, K_{10}, K_{11} \) used for BIT information.
FSQ item (unpacked)

± m m m m m o k address

LARC SCIENTIFIC COMPILER

Phase V

5.16

next 2 pages.

branch on i
Branch on i

BA

C-b:0

\[ \frac{c-b+d}{d} \rightarrow k \]

1

\[ k \rightarrow k_{12} \]

\[ 3 \rightarrow k_{10} \]

\( d_{K+1} \) numeric?

No

F

Yes

BRK

A * k \rightarrow k_{8}

V_{k} \rightarrow k_{0}

BVAR

R + # items \rightarrow R

K+1

k:j

\[ \frac{d}{k} \rightarrow k_{8} \]

\[ V_{k} \rightarrow k_{0} \]

\[ R + \# \ items \rightarrow R \]

\[ k+1 \]

5.16

\[ \frac{d}{k} \rightarrow k_{8} \]

\[ V_{k} \rightarrow k_{0} \]

\[ R + \# \ items \rightarrow R \]

K+1

k:j

\[ \frac{d}{k} \rightarrow k_{8} \]

\[ V_{k} \rightarrow k_{0} \]

\[ R + \# \ items \rightarrow R \]
Branch on i (continued)

LARC SCIENTIFIC COMPILER
Phase V

5.18

BE

4

BG

6

BF

7

BF

5

BE

m-1=m

BG

5

BF

5

3

5.17

DP_d:DP_b

m:99

DP_b:CM

m:99

DP_b:CM

m=99

m-1=m

DP_d:DP_b

m:99

DP_b→DP_d

3

5.17

5

3

5.17

Any F59 group?

No

Yes

F

5.17

F

5.17

8u

8v

≤

1

≤

Any F59 group?

No

Yes

b var.

b var.

b:d

= EXIT

EXIT

→ F61 on C_m!
W= a: cx-bd, P+1
for each W

EXIT
Subroutine

\[ \text{WWW} \]

combines up to 3 4-digit \( W \) fields into one word.

Subroutine

\[ \text{ZZZ} \]

unpacks up to 3 4-digit \( W \) fields and forms Table 3.

Table 3 Item

<table>
<thead>
<tr>
<th>Item</th>
<th>( W )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F54</td>
<td># ( W )</td>
</tr>
</tbody>
</table>

\( \oplus \) fixed point add operation

\([P+1]\) next A-register name
PHASE VI

PURPOSE

Phase VI reduces the space and time required by the object program by marking arithmetic computations which occur more than once as redundant. Phase VII can then generate code to compute just once, quantities which are used more than once. These quantities are called "common subexpressions" hereafter, and are of the form ABC where A is an arithmetic operation or : or , and B and C are operands (or other common subexpressions).

Phase VI also performs what arithmetic it can, preventing generation of unnecessary code and loss of time in the object program.

The marking of subexpressions takes place throughout each block of code (where a block of code is that object code between two successive points to which transfers can be made). Phase VI does no flow analysis of the program, as such an analysis would be costly in compiling time; hence if a subexpression is required in more than one block, it will be recomputed in each.

PROCESSING

A "block" extends from one block header to the next. Each block of the program is processed in turn. When there are no more blocks, Phase VI is through.

The following items are recognized as block headers:

1. Referenced statement names.
2. Arithmetic function statements.
3. Items indictating presence of SAL code.
4. Statement names generated by Phases IV and V before the first executable statement of the program and at the beginning of each loop.
The First Pass

The first pass of each block differs from the later passes in these two ways.

1. Items from Files 60 and 61 are input and merged during the pass. Thereafter the block is held in memory.

2. Subexpressions with known or computable values are replaced by constants representing those values. (The constants may then become the operands of other known or computable subexpressions, until all such expressions are eliminated).

A. \(+, -, *, /, **\) operations are performed on single-precision constant operands. (If a contingency occurs, an error item is filed in F92).

B. Known values are substituted as shown, where X represents a variable or function, and the constants are single-precision.

1. 0 for: \(X*0, 0*X, 0/X, 0**X\).

2. 1 for: \(X/X, 1**X, X**0\).

3. \(X\) for: \(X+0, 0+X, X*1, 1*X, X/1\).

4. \(X\) for: \(\backslash(0)\) where \(X\) is not a function.

5. \(X\) for: \(\text{FLOAT (X)}\) (The mode of \(X\) made correct).

6. 1 for: \(X/0\) (an error item is filed).
The mode word of the result is marked 1 (integer), 2 (floating), or 3 (double-precision) as the greatest mode word of the three terms.

Example: The expression

\[ X = 3 \ (N \cdot Y) + (1-N) \cdot Z + 3.14159265 \times 2 \]

reduces to \[ X = 3Y + 9.80960438 \] when PARAMETER \( N = 1 \) is given

\[ X = 3Z + 9.86960438 \] when PARAMETER \( N = 0 \) is given

Four instructions of code are generated.

Every Pass

During each pass the program sets up a table containing the relative memory locations of the operation and two operands of each subexpression in the block. The operations recognized are the arithmetic ones and colon and comma. The operands are constants, variables, functions, or (on the second and succeeding passes) marked subexpressions. The operands of commutative operations are arranged in ascending order (thus \( A \cdot B \) and \( B \cdot A \) are treated as identical).

The table is sorted and then scanned. During the scan the second and third words of each operation are modified as follows.

1. The u digit of the mode word.
   
   A. Marked 5 if the subexpression is not common. This mark prevents later passes from "processing the subexpression).  

   B. Marked 1 throughout first major sequence number in which common subexpression occurs.
Marked 2 thereafter.

2. The f digits of the mode word.

Marked with the "name" (0000-9999) of the subexpression.

3. The third word.

Contains the total number of occurrences in every case.

The next pass then begins. It will seek redundancies only among subexpressions containing smaller marked subexpressions. When none exist, the block is output to File 7.

Output Processing

On output, operations marked 5 are remarked 0.

Operations marked 1 or 2 are examined carefully. If one of these contains as an operation and a small subexpression which occurs the same number of times, the smaller operation is remarked 0.

The operations comma, colon (preceding a subscripted and variable preceding zero), are special cases. They are always re-marked 0, since nothing is to be gained in the object code by treating them as subexpressions. For example, in the statement

\[ X = ABCF (A, B+C) * ABCF (A, B + C) * DEFF (A, B + C) \]

the subexpression \( B + C \) is not re-marked even though it occurs as often as the larger \( A, B + C \); because it occurs more often than \( ABCF (A, B + C) \).

When the last block is output, Phase VI returns to the control program.
LARC SCIENTIFIC COMPILER  Phase VI

FORM TABLE OF COMMON SUB-EXPRESSIONS

FORMTAB 1

Diagram:

Start with $\omega_0$, which leads to

- $0 \rightarrow \lambda, \beta, \delta, \varepsilon, \phi$
- $0 \rightarrow \kappa$

Then:

- Extend F92
- Open F70

Next step:

- CRN45P: C

Followed by:

- M.G.

Final steps:

- Close F92
- Close F70

END

Graphical flowchart with nodes and arrows indicating the flow of execution and decision points.
S.P. = Single precision
D.P. = Double precision
LARC SCIENTIFIC COMPILER

Phase VI

EXECUTE OPERATIONS WITH DETERMINATE RESULTS.

AC

set contingency
trap to Sc

Branch on A

SE

SE

SE

SE

SE

reset Trap

SE

reset Trap

S.P. = Single precision
D.P. = Double precision.

CONTINGENCY

return to
point of error.

B+C

B-C

B*C

B/C

B**C

Brightarrow C

truncates if
both integers.

round &
truncates if
both integers.

CONTINGENCY

reset trap.

OP

99

99

99

99

99

 Scot

O \rightarrow 41

O \rightarrow 42

O \rightarrow 43

O \rightarrow 41

99 = 99

1 \rightarrow B

O \rightarrow B

1 \rightarrow B

1 \rightarrow B
GET NEXT ITEM

GI

ALF GETNX 2

ENDp1 EOF

MG

entry pt.? Yes GIS .S0

= L: LMAX

No

i tem -- S1

= .B1

0 -- L

UL TEST

GI 6

GI 1

GI 2

GI 3

GI 4

EOF

GI7

.y1

GIS

.B0

GI 10

L -- LL

K:0 = O

"SR sqrt"
COMPARISON SUBROUTINE

LARC SCIENTIFIC COMPILER

Phase VI

6.12

Compar

CM

K:0

≠

I' → I

I' + 1

K - 1

get u_I, u_I'

CM

K:0

≠

< Compare operations

=  

< Compare left operands.

=  

< Compare right operands.

1

2

3
"ELIMINATE" SUB-EXPRESSION

EL

LMAX -> I
reset LMAX

EL2

U_i, U_i
CMA

EL3

NAME + 1
2, NAME -> MARK
I -> FI
1 -> T

EL4

mark A_i

SEQU_i : SEQU_i

I, NAME -> MARK

T + 1

EL5

U_i, U_i
CMA

mark A_i

EL6
PHASE VII - LSC

General Description

Prior to Phase VII, file 7 has been ordered according to sequence number. The items contain complete mode words and other information necessary to the generation of machine code. Phase VII then processes the items in file 7 in one forward pass, generating "proto" instruction items for file 8. These instructions are complete with operation code, dictionary reference for the symbolic names, and A-Register assignments. These A-Register assignments are given assuming that an infinite number of A-Registers exist. It is necessary then, for Phase VIII to reassign the A-Register designations under the restriction of the number available. Other information is passed on to Phase VIII such as "successor" items and "end of list" items. The successor items give information about transfer points and the end of list items mark the end of Input-Output calling sequences. File 85 is also formed during Phase VII and contains all items not in sequence with file 8. Thus only file 85 need be sorted and merged with the larger file 8, which was produced in proper sequence. Examples of file 85 items include library call items, and instructions necessary for initializing input data addresses for subroutine and function subprogram compilations.
Phase VII Generator

Phase VII generation is facilitated by an interpretive system which offers a certain convenience when operating on Phase VII items.

A. Linkage to Generator

TR 7GEN This instruction transfers to the interpretive mode and is followed by SK instructions which give the parameters to be used. The first instruction to be interpreted is the first word following the last SK following the TR.

SK 0 P B This instruction's following a TR gives a parameter to the interpreter. The parameter is a standard three-word Phase VII item and the first word is in P + (B). If the location of the TR is L and the location of the SK is L + i, the parameter is transferred to in the interpretive mode as Pi (it is the i'th parameter).

T 7GEN This causes a transfer to the interpretive mode. The next word to be interpreted is the next word. (No parameters can be given and the subroutine level is not changed.)

B. Linkage from Interpreter

TB 0 LOC B Transfer back to machine language. This interpreter instruction causes the next instruction to be taken from LOC + (B) in machine language.

C. Subroutine Linkage Within Interpreter

TR SUBR Transfer and return. This interpreter instruction causes the next interpreter instruction to be taken from (SUBR). The subroutine level is increased by 1. Parameters to the subroutine is given by following SK instructions. Return is to the location following the last SK instruction following the TR.
SK 0 P

This gives a parameter to the interpretive subroutine called by the TR. If the SK is in the i'th word following the TR, the i'th parameter to the subroutine is the P'th parameter given to this program (or subroutine).

Exit from subroutine. Reduce level by one and transfer back to that level return.

D. Data Manipulation

Interpretive data manipulation instructions operate on fields of standard Phase VII items. Phase VII items have the following format:

Word 1

| S |

Word 2

| a b c d e e e u f f f f |

Word 3

| x y z T |

S is the sign of the quantity referred to and tells whether it is the quantity or the negative of the quantity.

a, b, c, d, e, u, and f are the fields of the standard mode word as described elsewhere.

x, y, z, and T are fields defined by Phase VII and describe data in index registers.

x tells whether the quantity is single or double precision (x=0 implies single, x=1 implies double).

y tells whether the quantity is in a left or right-hand register or a pair (single or double) of registers. y=0 implies left, y=1 implies right.

z tells whether the register currently contains the second or third operand and is a specialized field used only by Phase VII.
T is the class of register, T=0, 1, 2, 3, or 4 for absolute, arithmetic, index (see description of Phase VIII items).

Also word one, two or three can be referred to as an entire field by W1, W2, or W3.

In data manipulation, the field name is followed by the parameter number (as in A1, Y3, or W22). Parameter number 0 implies the result which is an index word being constructed. This parameter has only one word, W3, and so has fields W30, X0, Y0, Z0, and T0 only. Parameters number 1 through 5 refer to parameters as indicated above.

M \[ fP_1 \to fP_2 \]
Move \( fP_1 \) to \( fP_2 \).

S \[ k \to fP \]
Store \( k \) (a constant) in \( fP \).

C \[ fP_1 \to fP_2 \]
Compliment of \( fP_1 \) to \( fP_2 \). Compliment is defined here as: 0 replaces 1 and 1 replaces everything else.

E. Data Tests

TE \[ fP_1 \to LOC fP_2 \]
Transfer to LOC is \( fP_1 = fP_2 \). Either \( fP_1 \) or \( fP_2 \) can be a constant < 10.

TG \[ fP_1 \to LOC fP_2 \]
Transfer to LOC if \( fP_1 > fP_2 \). Either \( fP_1 \) or \( fP_2 \) can be a constant < 10.

TZ \[ fP \to LOC \]
Transfer to LOC if \( fP = 0 \).

F. Transfers

T \[ \to LOC \]
Transfers to LOC.

BIT \[ 0 \to LOC B \]
Transfers to LOC + (B)

G. Generate

TF \[ k \to LOC 0 T \]
Transfer to file. Transfer \( k \) items to file 8 starting at LOC with tracing digit \( T \). (If the tracing digit of items should be \( \backslash T \) is 7, tracing digit cannot be 7.) Description of items to be generated appears as follows:
H. Items to be Generated

Items to be generated have the following form:

\[
\text{OP} \quad A \quad M \quad B \quad T
\]

Items will have the tracing digit specified by the TF instructions.

OP is the operation mode of the item

A is the A field of the item. A is either the absolute A field or P0, \ldots, P5 for Parameter 0 through 5.

B is the B field of the item. B is either the absolute B field or P0, \ldots, P5 for Parameter 0 through 5 or F, S, D, or B which specifies that the M field is a literal and is fixed, single precision, double precision or B box respectively.

T describes the M field (unless B is F, S, D, or B in which case T is ignored) and no T implies M is a parameter number (P0, \ldots, P5).

T=R implies M is relative and T=A implies M is absolute.

T=ES or NES implies that an extensible or non-extensible successor is to be generated for M. (See successor item generation.)

M is the M field. M is P0, \ldots, P5 or if T is R or A, M the relative or absolute amount.

I. Example: Divide Parameter 2 by Parameter 3

Calling Sequence

\[
\begin{array}{llll}
\text{TR} & 7GD &  & \\
\text{SK} & 7EXP & 7J & \text{loc of P1} \\
\text{SK} & 7EXP & 7I & \text{loc of P2} \\
\text{SK} & 7EXP+7IL & 7I & \text{loc of P3} \\
\text{7GD} & \text{TG} & \text{B2} & 7GD1 \quad 1 \\
\text{TG} & \text{B3} & 7GD1 \quad 1 \\
\text{TR} & 7GFIF & & \text{P2 and P3 are integers} \\
\text{SK} & 2 & & \text{Fetch P2} \\
\text{TF} & 4 & 7GD2 & \\
\text{T} & 7GD8 & & \text{Generate 7GD2}
\end{array}
\]
<table>
<thead>
<tr>
<th>Source Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7GD2 ME P0 (l. 0) S</td>
<td>Make P0 D. P.</td>
</tr>
<tr>
<td>DSE P0 P3</td>
<td>Divide</td>
</tr>
<tr>
<td>CX P0 61 0 A</td>
<td>Truncate</td>
</tr>
<tr>
<td>C P0 61 0 A</td>
<td></td>
</tr>
<tr>
<td>7GD1 TG B1 7GD3 B2</td>
<td>Maximum of B1, B2, B3 to B1</td>
</tr>
<tr>
<td>M B2 B1</td>
<td></td>
</tr>
<tr>
<td>7GD3 TG B1 7GD4 B3</td>
<td></td>
</tr>
<tr>
<td>M B3 B1</td>
<td></td>
</tr>
<tr>
<td>7GD4 TG 3 7GD5 B1</td>
<td></td>
</tr>
<tr>
<td>S 1 X0</td>
<td>Make result D. P.</td>
</tr>
<tr>
<td>7GD5 TR TGFIF</td>
<td>Fetch P2</td>
</tr>
<tr>
<td>SK 0 2</td>
<td></td>
</tr>
<tr>
<td>TE B1 7GD6 3</td>
<td></td>
</tr>
<tr>
<td>TF 1 (DR P0 P3)</td>
<td>Generate S. P. Divide</td>
</tr>
<tr>
<td>T 7GD8</td>
<td></td>
</tr>
<tr>
<td>7GD6 TE B3 7GD7 3</td>
<td></td>
</tr>
<tr>
<td>TF 1 (DSE P0 P3)</td>
<td>P2, D. P., P3 S. P.</td>
</tr>
<tr>
<td>T 7GD8</td>
<td></td>
</tr>
<tr>
<td>7GD7 TF 1 (DD P0 P3)</td>
<td>P2 and P3 D. P.</td>
</tr>
<tr>
<td>7GD8 M W30 W31</td>
<td>Move result to P1 index word</td>
</tr>
<tr>
<td>TE S2 7GD9 S3</td>
<td></td>
</tr>
<tr>
<td>S 1 S1</td>
<td>1→ S1 if S2 = S3</td>
</tr>
<tr>
<td>TB 1 7Z1</td>
<td>Exit</td>
</tr>
<tr>
<td>7GD9 S 0 S1</td>
<td>0→ S1 if S2 = S3</td>
</tr>
<tr>
<td>TB 1 7Z1</td>
<td>Exit</td>
</tr>
</tbody>
</table>

This generator program uses subroutine 7GFIF which will fetch its 1st parameter and make the result double precision (1→ X0) if it is double precision and expand the result to double precision if X0=1 and the parameter is single precision.
PHASE VII - Successor Item Generation

1. Within a macro, the tracing mode designation on an instruction to be generated controls the successor item generation. If the tracing mode designation is:

   a. ES, then an extensible successor item for the M address parameter is generated and sent to file 8:

      | Word 1 | S(exit) - 1 | 99999 | 00 |
      | Word 2 | 02000       | 00 S (cont) |
      | Word 3 | 00000       | 00 000 00 |
      | Word 4 |             |__________________________|

      where S(exit) is the first five digits of the first word of the parameter item and S(cont) is the first five digits of the third word of the parameter item.

   b. NES, then a non-extensible successor item for the M address parameter is generated and sent to file 8:

      | Word 1 | S(exit) | 99999 | 00 |
      | Word 2 | 02000   | 00 S(cont) |
      | Word 3 | 00000   | 00000 01 |
      | Word 4 |         |__________________________|
Generation of Arithmetic Expressions

The portion of phase 7 that generates code for arithmetic expressions is divided into three sections, 7A, 7B and 7C. 7A converts the Lukasiewitz notation into a "tree notation," 7B determines the order of computation, and 7C generates the code.

I. 7A

Items are read in one at a time and sequence numbers are generated to sort into tree notation. The sequence numbers are divided into two parts: the high order part is a level number and the low order part is the ordinal number of the item within the expression. The level number of an item following an operation is the level number of the preceding item plus one; and, if the preceding number is an operand, the level number is equal to the highest unmatched level number (and causes that level number to be matched).

If a common-subexpression is encountered, which is not a first occurrence, it is skipped over in 7A.

When open subroutines are encountered, they are converted to special operations; e.g., the Lukasiewitz string

: MAX , X Y (from the source MAX (X, Y) is converted to

OP86 X Y where operation code 86 is the phase 7 code for MAX.

When closed subroutines are encountered, the ordinal parameter number is inserted into the corresponding parameter item and the number of parameters is inserted into the function's name item.

After all items have been read, they are sorted on the new sequence numbers. Following the sort, both operands of an operation appear together and must be related to the operand. The operation of two operands is always the closest preceding operation. At this time, if the operation is "-", it is changed to "+" and the second operand is made minus.
This section determines order of computation (except as regards where a common subexpression is computed within an expression). This is done by assigning "Q" numbers to all items as follows:

The "Q" of all simple operands (operands which are not subexpressions) is zero, and if operation \( j \) has operands \( i \) and \( i+1 \), then \( Q_j = \max (Q_i, Q_{i+1} + S(Q_i, Q_{i+1})) \) where \( S(Q_i, Q_{i+1}) \) is the Kronecker delta. The "Q" of the lowest level operation is usually the number of A registers required to compute the expression, if the entire expression is single precision and has no first occurrence of common subexpressions.

The computational order is now determined by starting at the lowest level operation. First, compute the operand with the higher "Q" and repeat the step for the lowest level operation in this subexpression, and repeat this step until an operation is found where both of its operands have a "Q" of zero. Now this operand may be computed. After this is computed, the "Q" of this subexpression may be replaced by zero and the process repeated for the other operand of the operation involving this operand.

The above description would indicate that this is a two-pass process. Actually, this is not the case, but it is easier to understand the process in this way. In the implementation of the above algorithm, an associative list is built up at the same time the "Q's" are generated, which give the order of computation. If the starting points for computing two subexpressions joined by an operation to form a new subexpression are known, then the starting point for computing the new subexpression is the same as the starting point for computing the operand with the higher "Q" (or the right-hand operand if they have equal Q's).

Symbolically, the actual algorithm employed is as follows:

If \( A \) is a subexpression, let

\[
S(A) = \text{the first operation in } A \text{ to be computed.}
\]

\[
L(A) = \text{the last operation in } A \text{ to be computed.}
\]

If \( X \) and \( Y \) are two operations to be computed, let

\[
Y = F(X) \text{ mean that } Y \text{ is to be computed immediately after } X.
\]
If operation "A" joins the operands B and C, then there are two cases:

Case 1  \( Q(B) \leq Q(C) \) in which case
\[
\begin{align*}
S(C) & \rightarrow S(A) \\
S(B) & \rightarrow F(L(C)) \\
A & \rightarrow F(L(B)) \\
A & \rightarrow L(A)
\end{align*}
\]

Case 2  \( Q(B) > Q(C) \) in which case B and C are interchanged in the above.

Finally, \( \max (Q(B), Q(C)) + S(Z(B), Q(C)) \rightarrow Q(A) \)

Consequently, the starting and ending points for two subexpressions are known, the starting and ending points for the operation joining them are determined, and as each operation (except the last or lowest level one) has a follower (F), the computation order is determined because \( F \) is determined for each operation.

To start the process is actually a separate case. If \( \max (Q(B), Q(C)) = 0 \), then
\[
\begin{align*}
A & \rightarrow S(A) \\
A & \rightarrow L(A)
\end{align*}
\]

When a common subexpression is encountered, its \( Q \) is set to zero.

III. 7C

Here the actual generation is done. If the lowest level operation is A and Z is the next operation to be generated, then \( S(A) \rightarrow Z \) and thereafter \( F(Z) \rightarrow Z \) unless one of the operands of an operation is a common subexpression in which case \( S(B) \rightarrow Z \) if B is the subexpression. If both operands are common subexpressions, \( S \) of the right-hand operand goes to \( Z \) first.

After a common subexpression is generated, it is left in an A register (which may require an F or FF to be generated) and the mode information, including the index number is entered in a table. When a subexpression is already computed, instead of \( S \) of that subexpression replacing \( Z \) in the above, the mode information is placed in the item from the table and \( Z \) modified in the usual way \( (F(Z) \rightarrow Z) \).
Once a particular operation is to be generated, that operation is placed in an index register, and a jump to the proper interpretive routine is made.

Sometimes the negative of an expression is computed instead of the expression desired; i.e., consider the expression A-B+C, where A, B, and C are single precision variables, the code generated will be:

\[
\begin{align*}
F & \leftarrow B \\
M & \leftarrow C \\
N & \leftarrow A
\end{align*}
\]

leaving the negative of the expression desired in A register \( \alpha \). Therefore, all items in 7C are signed and the sign of an operation (as well as the code generated) is a function of the operation and the signs of its two operands. Although no extra instruction can ever be generated in computing an expression in this way, the sign of the resulting expression may be negative.

If a negative expression results in generating an arithmetic statement, a SN or SSN will be used to store the result instead of a S or SS. However, if the expression results in generating an IF statement, an extra SN or SSN may be generated. Nevertheless, ending up with a negative expression means that at least one instruction was saved in generating the expression.
Phase VII Sample Codes

Generation of object code for arithmetic combinations of quantities with different modes.

I. Addition and Subtraction: \( a + b \)

A. If the signs of \( a \) and \( b \) are the same, \( v = A \) and the operation gets the same sign.

B. If only one of \( a \) and \( b \) are positive, \( v = N \), let \( a \) be the positive one, and \( b \) the negative one. The operation gets the opposite sign of the quantity in the \( M \) address of the last instruction.

<table>
<thead>
<tr>
<th></th>
<th>a in M store</th>
<th>a in M store</th>
<th>a in A store</th>
<th>a in A store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b in M store</td>
<td>b in A store</td>
<td>b in M store</td>
<td>b in A store</td>
</tr>
<tr>
<td>a single</td>
<td>F ( \delta ) a</td>
<td>V B a</td>
<td>V a b</td>
<td>V a b</td>
</tr>
<tr>
<td>b single</td>
<td>V ( \delta ) b</td>
<td>SM 0 ( b+1 )</td>
<td>SM 0 ( a+1 )</td>
<td>SM 0 ( a+1 )</td>
</tr>
<tr>
<td>(op normal)</td>
<td></td>
<td>F ( \delta ) a</td>
<td>F ( \beta ) b</td>
<td>SM 0 ( \beta+1 )</td>
</tr>
<tr>
<td>a single</td>
<td>F ( \delta ) a</td>
<td>SM ( \delta+1 )</td>
<td>SM 0 ( \beta+1 )</td>
<td>VV a b</td>
</tr>
<tr>
<td>b single</td>
<td>F ( \beta ) b</td>
<td>SM ( \delta+1 )</td>
<td>SM 0 ( \beta+1 )</td>
<td>VV a b</td>
</tr>
<tr>
<td>(op double)</td>
<td>SM 0 ( \delta+1 )</td>
<td>VV ( \delta ) b</td>
<td>VV ( \delta ) b</td>
<td>VV a b</td>
</tr>
<tr>
<td>a single</td>
<td>F ( \delta ) a</td>
<td>SM 0 ( a+1 )</td>
<td>VV a b</td>
<td>VV a b</td>
</tr>
<tr>
<td>b double</td>
<td>SM 0 ( d+1 )</td>
<td>SM 0 ( d+1 )</td>
<td>VV a b</td>
<td>VV a b</td>
</tr>
<tr>
<td>VV ( \delta ) b</td>
<td>VV ( \delta ) b</td>
<td>VV a b</td>
<td>VV a b</td>
<td></td>
</tr>
<tr>
<td>a single</td>
<td>F ( \beta ) b</td>
<td>SM 0 ( b+1 )</td>
<td>SM 0 ( \beta+1 )</td>
<td>VV a b</td>
</tr>
<tr>
<td>b single</td>
<td>SM 0 ( \beta+1 )</td>
<td>VV b a</td>
<td>VV a b</td>
<td>VV a b</td>
</tr>
<tr>
<td>VV b a</td>
<td>VV a b</td>
<td>VV a b</td>
<td>VV a b</td>
<td></td>
</tr>
<tr>
<td>a double</td>
<td>FF ( \delta ) a</td>
<td>VV b a</td>
<td>VV a b</td>
<td>VV a b</td>
</tr>
<tr>
<td>b double</td>
<td>VV ( \delta ) b</td>
<td>VV b a</td>
<td>VV a b</td>
<td>VV a b</td>
</tr>
</tbody>
</table>
II. Multiplication: a × b

A. If signs of a and b are the same, the operation is tagged positive.

B. If the signs of a and b differ, the operation is tagged negative.

<table>
<thead>
<tr>
<th></th>
<th>a in M store</th>
<th>a in M store</th>
<th>a in A store</th>
<th>a in A store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b in M store</td>
<td>b in A store</td>
<td>b in M store</td>
<td>a in A store</td>
</tr>
<tr>
<td>a single</td>
<td>F  δ  a</td>
<td>M  b  a</td>
<td>M  a  b</td>
<td>M  a  b</td>
</tr>
<tr>
<td>b single</td>
<td>M  δ  b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(op normal)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a single</td>
<td>F  δ  a</td>
<td>ME  b  a</td>
<td>ME  a  b</td>
<td>ME  a  b</td>
</tr>
<tr>
<td>b single</td>
<td>ME  δ  b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(op double)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a single</td>
<td>F  δ  a</td>
<td>S  δ  δ+1</td>
<td>S  δ  δ+1</td>
<td>S  δ  δ+1</td>
</tr>
<tr>
<td>b double</td>
<td></td>
<td>PR  δ+1  11</td>
<td>PR  δ+1  11</td>
<td>PR  δ+1  11</td>
</tr>
<tr>
<td></td>
<td>MM  δ  b</td>
<td>MM  b</td>
<td>MM  δ  b</td>
<td>MM  δ  b</td>
</tr>
<tr>
<td>a double</td>
<td>F  β  b</td>
<td>S  β  β+1</td>
<td>F  β  b</td>
<td>S  β  β+1</td>
</tr>
<tr>
<td>b single</td>
<td></td>
<td>PR  β+1  11</td>
<td></td>
<td>PR  β+1  11</td>
</tr>
<tr>
<td></td>
<td>S  β  β+1</td>
<td>MM  β  a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a double</td>
<td>FF  δ  a</td>
<td>MM  b  a</td>
<td>MM  a  b</td>
<td>MM  a  b</td>
</tr>
<tr>
<td>b double</td>
<td>MM  δ  b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
III. Division: \( a/b \)

A. If signs of \( a \) and \( b \) are the same, the operation is tagged positive.

B. If the signs of \( a \) and \( b \) differ, the operation is tagged negative.

<table>
<thead>
<tr>
<th></th>
<th>a in M store b in M store</th>
<th>a in M store b in A store</th>
<th>a in A store b in M store</th>
<th>a in A store b in A store</th>
</tr>
</thead>
<tbody>
<tr>
<td>a single</td>
<td>F ( \delta ) a</td>
<td>F ( \delta ) a</td>
<td>DR ( a ) ( b )</td>
<td>DR ( a ) ( b )</td>
</tr>
<tr>
<td>b single</td>
<td>DR ( \delta ) b</td>
<td>DR ( \delta ) b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(op normal)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a single</td>
<td>F ( \delta ) a</td>
<td>F ( \delta ) a</td>
<td>SM 0 ( a+1 ) ( b )</td>
<td>SM 0 ( a+1 ) ( b )</td>
</tr>
<tr>
<td>b single</td>
<td>SM 0 ( \delta )</td>
<td>SM 0 ( \delta+1 )</td>
<td>DSE ( a ) ( b )</td>
<td>DSE ( a ) ( b )</td>
</tr>
<tr>
<td>(op double)</td>
<td>DSE ( \delta ) b</td>
<td>DSE ( \delta ) b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a single</td>
<td>F ( \delta ) a</td>
<td>F ( \delta ) a</td>
<td>SM 0 ( a+1 ) ( b )</td>
<td>SM 0 ( a+1 ) ( b )</td>
</tr>
<tr>
<td>b double</td>
<td>SM 0 ( \delta+1 )</td>
<td>SM 0 ( \delta+1 )</td>
<td>DD ( a ) ( b )</td>
<td>DD ( a ) ( b )</td>
</tr>
<tr>
<td></td>
<td>DD ( \delta ) b</td>
<td>DD ( \delta ) b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a double</td>
<td>FF ( \delta ) a</td>
<td>FF ( \delta ) a</td>
<td>DSE ( a ) ( b )</td>
<td>DSE ( a ) ( b )</td>
</tr>
<tr>
<td>b single</td>
<td>DSE ( \delta ) b</td>
<td>DSE ( \delta ) b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a double</td>
<td>FF ( \delta ) a</td>
<td>FF ( \delta ) a</td>
<td>DD ( a ) ( b )</td>
<td>DD ( a ) ( b )</td>
</tr>
<tr>
<td>b double</td>
<td>DD ( \delta ) b</td>
<td>DD ( \delta ) b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For integer divide use corresponding macro for "a single, b single, op double" followed by:

\[
\begin{align*}
\text{CX} & \quad \delta & \quad 61 \\
\text{C} & \quad \delta & \quad 61
\end{align*}
\]
LIBRARY ROUTINES

1. Phase III detects occurrences of a reference to a sub-program (including library routines) and transmits them to Phase IV. The list of subroutines generated by Phase II allows the distinction of open subroutines.

2. Phase IV produces LIB items for F9.4 as requested.

3. Phase VII marks a table of library routines over which it has control.
   a. Introduced routines
   b. Convertable routines

   At the end of file 7, Phase VII sends LIB items for each marked entry in the table to F9.4. They will be of the same form as those produced by Phase IV.

   All references to subroutines will be given an address mode of 06.

4. Phase IX will produce LIB lines in SAL as specified by F9.4. The address mode of 06 will force editing with a marker.
LARC SCIENTIFIC COMPILER

Phase VII

MAIN FLOW

START

Display +7+ in 5 digit display.
Store crumbout transfer in loc. 10794
O → K, SEQ
Reset FTHRU, ADJR, LIST
Open F?0 for reading
Open F80, 85 for writing.

GET

8 word file 70 to item 1 ITEM:
WORD 1 sequence, Subsequence, ID
WORD 2 mode word
WORD 3

Items generated for File 8 are sequenced using "SEQ"

Is sequence the same as SEQ

Yes

Subsequence of ITEM
Subsequence of SEQ

No

0 → subsequence of SEQ

ID: 8, 9 or 11 ?

Yes

CNTN

No

ID ≤ 50

Yes

Set FTHRU
Use transfer table indexed by ID to determine item type. If ID = ii,
then symbol transferred to is named ID ii

No

err 22

FINIS

End of File 70
Reset ADJR
Preset CREG so that next expression is computed in the right half of the register of the last expression.
Input list item for variable "X".

Generate:
\[
\begin{align*}
S & \times 0.5 \\
S2 & \times 0.5
\end{align*}
\]
if \(X\) is double precision.

Here after calculating expression and ADJR reset.

Output list item for variable "X".

Generate:
\[
\begin{align*}
F & \times 0.5 \\
FF & \times 0.5
\end{align*}
\]
if \(X\) is double precision.

Get

Here after computing arithmetic function (in EXP)

Generate:
\[
F \ #1 \ EXP
\]

Function double prec.? Yes

No

EXP positive?

Yes

No

Generate:
\[
SN \ #1 \ #1
\]

Get

Expanding EXP is double precision.

EXP positive?

Yes

No

Generate:
\[
T \ #0
\]

Get

Fetch RSLT into register indicated by GREG.

Get

No

Yes

Generate:
\[
SSN \ #1 \ #1
\]

Get

Get

Get
LARC SCIENTIFIC COMPILER

Phase VII

ID03

Successor item from Phase III

Make 4th word of successor item zero and output 4 word item to File 8

GET

CNTNI

Continue item generation.

0 → e
(erasability flag)

ID: 08 =

ID: 23 =

ID: 23 =

ID: 32

Form F8 continue item:
W1 s ss o
W2 01 0000e (dict. ref.)
W3
W4 word 3 of incoming item
Output 3 word continue item to F8

A

Set FTTHU flag.

ID: 23 =

Is a subprogram being compiled?

No

GET

Yes

Is this the first executable continue? (is first word 1999900000008)

No

GET

Yes
Unsubscribed variable ref.

A1

A1 is the expression analysis subroutine, and is flow-charted separately. (The result is left in EXP)

GETA1

Unsubscribed store ref.

L1 checks for implicit arrays and is flow-charted separately.

ID22

Subscribed store ref.

A1

L2

L2 generates BIT instructions for implicit arrays if needed and is flow-charted separately.

GETA2

Unsubscribed input ref.

L1

ID27

Subscribed input ref.

A1

L2

GETA3
Unsubscripted output ref.

ID28

Unsubscripted output ref.

L1

ID29

Set arithmetic function flag (ARF). Advance assignment and generate forced assign items for fast registers #0 and #1. Define register #0

ID23

GET

ID30

"CALL" with no arguments.

Generate:
TB #0 sub.name, where sub.name is dict. ref. from input item

GET

A1

L2

GETAA
ID32

Here for beginning of subprogram.

Generate forced assign and definition item for fast register #0.

GET

ID45

Here for simple constant.

ID14

ID48

The following 2 expressions are to be placed in adjacent registers.

Set adjacent register flag. (ADJR)

GET
LARC SCIENTIFIC COMPILER

Phase VII

ID₄₉

Operator 'item'

ee: 9 ≤ ID₄₉

≥

ee: 61 ≤ err 20

O → i

ID₄₉₁

Read in D (from op. mode word) items, into ITEM+(i+1)*IL

End of File 7 → err 21

i+IL≥D i

Was last item read in an op. item?

Yes

No

Use transfer table indexed by ee (from first item mode word) to determine operator type.
If ee digits are xy, symbol transferred to is EEXY

ID₄₉₂

Is next item subs. store? (ID = 22)

Yes → ID₂₂

No

Is next item subs. input? (ID = 22)

Yes → ID₂₇

No

Is next item subs. output? (ID = 29)

Yes → ID₂₉

No

A 1

ARF reset GETAE1

Just did arith. function.

Reset ARF

GETAE6
F P1 P2 → INSTR.

TLz P1 P2 O NES → INSTR.

A P1 P2 → INSTR.

Generate INSTR. where P1, P2 are parameters from ITEM

S P1 P2 → INSTR.

Ax P1 P2 → INSTR.
**Phase VII**

**ID₅₅**

1. **F P₁ P₂** → **INSTR**
   - Read in next item from File 7 to ITEM + IL
   - Generate INSTR where P₁, P₂ are parameters from ITEM and ITEM + IL
   - GET

**ID₅₆**

1. **BIT P₁ P₂ O N E S** → **INSTR**
   - End of File 7 err 21

**ID₅₇**

1. **This item is met after a polish indexing string has been processed.**
   - Send out register definition item for the register just used for the polish indexing string.
   - O → GREG
   - Reset RICH flag
   - GET
This item precedes a polish indexing string, and gives information about which fast register to use.

Place register name (from 58 item) in GREG. (This causes the following expression to be computed there.) Set RICH flag.

ID 58

ID 59

NX P1 P2

← INSTR.

ID 551

GET
ID 60

Beginning or end of list item.

Is it beginning? (Is 3rd word non-zero)

Yes

This is the start of an arithmetic list which has a loop in it.

Set EOL flag. Generate item for F8 to cause storing of arithmetic register, if necessary.

No

ID 602

This is an end-of-list item. 2nd. word is 2 for I/O lists, 3 for arithmetic lists.

ID 602

Is this end of arithmetic list?

No

ID 601

Yes

EOL

reset

GET

set

Reset EOL. Output end of list item to F8

GET

Generate: T 9NTR 8 9FIN Item for F8

GET
Here for all input/output operations i.e.,
EE = 17-28, 31-35, 49-52 or 61
Generate the I/O calling sequence.

Generate:
TB #0 subroutine name.
Get D from operator mode word.

If $i < 15$
(formatted ref.)

= Is format name assigned?

Yes

Generate:
sk o p_{i+1} 0 6

$D_i = i$

No

Generate:
sk o p_{i+1} 0 9

$i+1 \rightarrow i$

$D-1 \rightarrow D$

$D \neq 0$

"PAUSE"

Generate:
T n
where n is the location specified in CEPAUS of control region.

"STOP"

Generate:
T n
where n is the location specified in CESTOP of control region.
SENSE LIGHT i

Note: Sense light is on if flip-flop i is reset.
IF (SENSE LIGHT i) n₁, n₂

**Phase VII**

- **EE36**
  - **0 → j**
  - **n₁ dropout?**
    - No: **j + 1 → j**
    - Yes: **n₂ dropout?**
      - No: **j + 2 → j**
      - Yes: *

- **j**
  - **j = 0**
    - **IF (SENSE LIGHT i)**
      - Generate: TF i
      - SF i
      - T n₁
      - Generate non-extensible successor (NES) for n₁
  - **j = 1**
    - **IF (SENSE LIGHT i) n₁**
      - Generate: TF i
      - HERE + 3
      - SF i
      - T n₁
      - Generate NES for n₁
  - **j = 2**
    - **IF (SENSE LIGHT i) n₂**
      - Generate: TF i
      - n₂
      - SF i
      - T n₁
      - Generate NES for n₂
  - **j = 3**
    - **IF (SENSE LIGHT i) n₁, n₂**
      - Generate: TF i
      - n₂
      - SF i
      - T n₁
      - Generate NES for n₂ and ES for n₁, unless n₁ is assigned (k=1)
      - Reset FRHEU flag

**PGET**
IF (SENSE SWITCH $j$) $n_1, n_2$

$0 \rightarrow i$

Is $n_1$ dropout? Yes

No

$i + 2 \rightarrow i$

Is $n_2$ dropout? Yes

No

$i + 1 \rightarrow i$

CRSSWT + $j$ → P1

(CRSSWT has loc. of first storage loc. for sense switches)
Generate: $F \times P1$
where $x$ is fast register

$i = 0$

$i = 1$

$i = 2$

$i = 3$

Generate: $TGE \times n_2$ (NES)

Generate: $T2 \times n_1$ (NES)

Generate: $TGE \times n_2$ (NES)

Reset FTHEU

Generate: $T2 \times n_1$ (NES)
EE45

\[ \text{GOTOF} (n_1, n_2, \ldots, n_k), i \]
(Computed \text{GOTOF})

\begin{align*}
\text{Generate:} & \\
F & \leftarrow i \\
ax & \leftarrow 61 \\
A[k] & \leftarrow (88000999999-9k) \\
BIT & \leftarrow \text{HERE} + 2 + k \\
H & \\
T & \leftarrow \text{HERE} + 1 + k \\
R & \rightarrow j \\
\end{align*}

Get next item from File 7 \((n_k-j+1 \text{ item})\)

Generate:
\[ T \leftarrow n_k-j+1 \text{ (NES)} \]
\[ k-1 \rightarrow k \]

If \(k = 0\), then \text{GET}

End of File 7

err 21
IF ( ) n₁, n₂, n₃

NES : Non extensible successor.
ES : Extensible successor.

j = 0  j = 1  j = 2  j = 3  j = 4  j = 5  j = 6  j = 7
all dropouts (error?)

IF ( ), n₂

IF ( ) n₁

IF ( ) n₁, n₂

IF ( ) n₁, n₂, n₃

G.
Check for Implicit array on arithmetic or I/O unsubscripted list item

Is item dimensioned?

This is an implicit array.

BB O I offset → Z1

Is list item double precision?

Let number of items in array be denoted by a1 a2 a3 a4 a5 a6 (6 digits)

a1 a2 a3 → C2

a4 a5 a6 → PC1

1 → bit count

Form B-Box (es) using counts and offset, and generate fetch instructions for them.

Code generated.

| F | ν | (1,0) |
| F | β | (BB 100 111) |
| S | α | X β |
| BIT | β | HERE-1 |
| F | β | (BB 0 111) |
| S | α | Y β |
| BIT | β | HERE-1 |
| F | β | (BB 10 2 22) |
| F | β+1 | (BB 3 0 0) |
| SM | 0 | α+1 |
| SS | α | Z β |
| BIT | β | HERE-1 |
| BIT | β+1 | HERE-2 |

Example:

DIMENSION X (10,10), Y (10,10), Z (10,201)

DOUBLE PRECISION Z

X, Y, Z = 1
Fetch Upper

Fetch Upper & Match

Fetch

Fetch CR

Expand
LARC SCIENTIFIC COMPILER

Phase VII A

16

A

i → k
Sort k 3 word items on S char (S,S,S)
Starting at X_0
k-2 → k
k-1 → j

18

j

19

j_0 - 49

21

j-1 → j

22

k → P_k, S_k

23

e_j - 2

24

1 → e_j
1 → A_{k+1}

25

k → P_{k+1}; S_{k+1}
j → C_{k+1}
k-2 → k

26

k

> 0

> 0

78.
GENERATOR

E

Is arg. indexed?

Yes → F

0 to tracing digit
Set literal flag.
Output F, AX

No → E → M (arg.#)
Make address
absolute.
Set indirect addressing.
Store index word for #0

A

Can there be a carry?

Yes → Output E, B.

No → Output S.

D+1
LARC SCIENTIFIC COMPILER  Phase VII

7GDVD

Move(B1, B2, B3) → B1
Fetch P2

op double?

No

op integer?

No

DR P2 P3

Yes

expand

P2 double?

No

DSE P2 P3

Yes

P3 double?

No

DD P2 P3

Yes

Signs equal?

Yes

+ → P1

No

P2 and P3 integer?

No

⇒

Save
7GADD X, Y
restore
[TGE P1 *+2]

DEM (X, Y)

[ F P1 0.0 ]

Pi d.p.

P1

No

Yes

[FF Pi 0.0]

Save
7GDVD P2, P3
fix P1
7GMPY P1, P3
-P2 → P2
7GADD P1, P2
-P1 → P1

MOD (X, Y)

CX P1 G1
C P1 G1
Make op. integer.
LARC SCIENTIFIC COMPILER

Phase VII

1. Store

2. GEN one of:
   - [S a T]
   - SN a T
   - S5 a T
   - SSN a T
   - T → 7ST
   - T+1 → T

3. GEN [Ax b lit]
   - [S b v]
   - Y → 7ST
   - lit.flag → 7ST
   - Y+1 → Y

4. mode word of var. to 7ST

5. 7ST+1 → 7ST

---

Gen Skc

6. (7ST) var?
   - Yes
   - No

7. lit.flag?
   - No
   - GEN cont.
   - Yes

8. set lit 2 item

GEN [Sk 0 7ST 0 9]
Phase VIII is concerned with the assignment of physical fast registers to the instructions generated by Phase VII. In addition, movements to and from fast registers and memory cells used for temporary storage are interpolated into the generated program. The phase is divided into five subphases called VIIIa through VIIIe. The various functions of these subphases are described below.

**PHASE VIIIa - Backward Scan**

The output of Phase VII is in two files: File 8.0 and File 8.5. File 8.0 is generated in order on sequence number and contains the bulk of the generated code. File 8.5 is generated out of sort and contains relatively few items (to wit: extensible successors produce by T commands, library call items, and code used to initialize certain references to parameters in functions and subroutines). File 8.5 is sorted by the control program prior to the inception of Phase VIII.

Phase VIIIa reads Files 8.0 and 8.5 backwards and merges them into descending sequence. Library call items from 8.5 are detected by the merge and written directly to File 9.4 so that they do not enter into the remainder of Phase VIII processing.

The items from the merged file are now identified as representing instructions, continues, successors, instruction literal flags, or end of list flags. Successors and the two flag items are converted to their File 8.1 form and written out. The remainder of the Phase VIIIa processing is done on instruction and continue items.

Continue items serve to identify block boundaries and as such initialize certain cells and tables which collect information about a block. Prior to this initialization, the information concerning the previous block is written in File 3.1. In particular, the tables (DR and DIS) containing the distances between usages of symbolic registers are emptied, the table (MAP) marking the preferential positions of indices (see VIIIb for a description of preferential positions) is cleared, and the cell containing the distance to the next TB (92) command is set to infinity (99999). In addition, the register SQ, which represents the position of each command in the block, is set to 99998.
Instructions have their symbolic registers processed. Each instruction contains a register reference word for its A field and perhaps (if the address mode is less than 10) also one for its B field. The register reference words have the form:

<table>
<thead>
<tr>
<th>O</th>
<th>O</th>
<th>P</th>
<th>O</th>
<th>T</th>
<th>O</th>
<th>O</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRECISION NUMBER</td>
<td>TYPE OF REGISTER</td>
<td>NAME OF REGISTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The precision number, P, identifies that portion of a symbolic register which is involved with this particular reference. (A symbolic register may comprise several physical fast registers.) The T field identifies the type of the register by:

T = 0 Absolute reference
1 Arithmetic
2 Index
3 Common subexpression
4 Preassigned reference
5 Parameter reference

The N field is a five-digit number which identifies within a given type the particular register involved. For T = 1, 2, or 3, N is a symbolic reference to be assigned by Phase VIII. For T = 0 or 4, N specifies directly which register is to be used. For T = 5, a memory cell is actually specified which is to be used to hold an expression that is a parameter to a function or subroutine.

Each instruction encountered is checked to see if it is one of the list TE, TG, TTE, or TTG. If so, certain anomalies in the value of P produced by Phase VII are corrected. Also, a check is made for the occurrence of a TB command so that steps may be taken to insure that registers #01 through #04 are free when the TB occurs.
For each register reference word, a subroutine (8AT) is entered which attaches to the instruction item the distances (recorded in the DIS table) to the next occurrence of the register and to the last occurrence. Also, the current value of SQ is attached and placed into the DIS table. Then SQ is decremented. With this information, Phase VIIIb can, during a forward scan, determine the relative undesirability of destroying the contents of a physical register by noting how soon it will be required in the computation.

**PHASE VIIIb - Forward Scan**

Phase VIIIb assigns a physical fast register for each symbolic fast register. The input is File 8.1, which, if read backwards, contains the executable part of the program in "execution order".

For each item in File 8.1, the suffix word is read and a branch is made on the identification to the proper routine to process that item. These individual routines read the remainder of the item.

**Instructions:**

The "B" reference is processed first if one is present. If the address mode is 05, then the "B" reference becomes the address of the instruction with no indexing. In this case, the "B" reference is tested to determine if it represents a single-precision common subexpression which has been previously put into temporary storage. If so, the address of the instruction is replaced by the corresponding temporary storage reference.

If the address mode is not 05 and is less than 10, then the proper assignment is made by the assign subroutine.

The instruction is checked to determine if it is an unindexed fetch command which immediately follows a store command which referred to the same symbolic register. If so, a flag (REGSYM$_{0}$) is set for use by the assign subroutine.
Some special instructions identified by operation codes 44, 49, and 54 are called forced-assign instructions. They refer to single, double, and quadruple precision registers respectively. Their use is to force VIIIb to make a particular fast register assignment for a given symbol rather than searching for an optimal one.

In this manner, for example, the result of the evaluation of an arithmetic statement function is forced to appear in register #01. If a forced assignment instruction is detected, the Pi switch is reset and the dictionary reference field of the instruction item is recorded.

The assign subroutine is entered with the parameters describing the "A" reference. Then the File 90 form of the instruction is constructed.

If the item immediately prior to this instruction was an instruction literal item, the implied continue item is written in File 90 and the instruction is written to File 94 once or twice with any non-zero sequence numbers specified by the instruction literal.

If the GOOD flag was set non-zero by the assign routine or the Pi switch is reset, or if for a F or FF instruction the "A" assignment equaled the "B" assignment, then the instruction is not output and control returns for the next File 8.1 item.

The instruction assigned is checked to determine if it is a S command; if not, it is written to File 9.0 and control returns. If so, and it is not indexed, its address is saved to compare with the next fetch command as described above. Also, checks are made to determine if the instruction represents an index definition item or a beginning of list item (address mode equal 40 and 50 respectively).

In the former case, an index definition item is formed and written in File 8.4 (see VIIIe description). The inverse level number is taken from the item and stored in LEVEL. Now if the dictionary reference field of the instruction item is not zero, it represents the sequence number of a continue to which the defining command transfers (either a BIT or an A-TLZ pair). This return point for a loop then also represents a point where the index is defined, so a second File 8.4 definition item is produced using the sequence number. In addition, it is
required that the index be available in the same register at the beginning of the loop as it was at the BIT so that, if the second definition is reconverted to a store command by VIIIe, then a meaningful quantity will be stored for the first entrance into the loop. Thus, a File 8.3 entry condition item (File 8.3 continue item) is required. Since the serial number (see discussion of treatment of continue items) of the reference continue is not known, a special item identification of 2 is required so that VIIIId can determine the position of that continue in the object code by means of the sequence number alone.

In the latter case, the "A" reference is saved in LOCK which inhibits the assign routine from using that particular register until released. This is necessitated for the List=Expression items which contain explicit loops since, in this case, continue items will appear within the code generated for the list, and it is normally assumed that arithmetic quantities (in this case the expression) are not preserved across a continue.

In either case, control returns immediately for a new item so that the instruction does not appear in File 9.0.

Continues:

The continue item serves to force the output of information collected since the last continue and also to initialize the areas in which information will be collected starting with this continue. The commands required to save and restore any fast registers beyond #04 which are used by an arithmetic statement function are generated.

In particular, the code first reads the remainder of the continue item and checks to see if it does not introduce a new block (the identification is negative); if so, the File 9.0 form of the continue is produced and written out. Control returns immediately for a new item.

If the Tau Switch is reset, the previous continue introduced an arithmetic statement function. Subroutine X1 is entered to save and restore registers. Tau is then set.

The continue flag is then examined. The possible values are zero, one, and two, which respectively indicate a normal (programmer referenced statement name), erasable (beginning of SAL block), and arithmetic function continue.
If an arithmetic function is detected, the AFRST table is cleared and Tau reset. AFRST will record registers used.

For arithmetic function and normal continues, the File 9.0 item is written and the inverse level (if non-zero) is saved in LEVEL.

Now for normal and erasable continues, the table of available arithmetic temporary storage, ATSAT, cells is reset and the Epsilon switch tested. If set, this is the first continue encountered. The ENCT, EXCT, and HIST tables are cleared and Epsilon reset.

If reset, the File 8.2 continue item formed by the last continue is written out, the NI counter advanced, and a File 8.3 entry condition item is written for each non-zero entry in ENCT. Again ENCT, EXCT, and HIST are cleared.

In any case, the next File 8.2 continue item is formed and saved, and, if this continue is erasable, register #00 through #04 are marked as having been used.

Control returns for a new item.

Successors:

Successor items serve to detail the flow of control from one block to another. They are processed as follows:

The successor item is read in, the NI counter advanced, and a File 8.2 successor item is formed and written out. Then for each non-zero entry in EXCT, a File 8.3 exit condition item is written. Control is returned for a new item.

Index Map:

The index map contains a mark for each physical register preferred for symbolic index quantities. These marks are used to initialize the register commitment table, RCT, which aids the assign subroutine to select the optimal assignment for a given symbolic register.
The index map is read in and its digits are stored one to the word in RCT. Also, the storage history table, SHT, is marked as empty, and the register usage table, RUT, is cleared.

**Instruction Literal:**

The instruction literal item signals that the next instruction is to be used as a literal. The item is read in and saved and the IL switch is set on.

**TB Distance:**

The TB distance item specifies the distance to the next TB command. This information is used to determine if, were a given symbolic register assigned in the range #00 through #04, it would have to be moved before its last usage due to an interviewing subroutine call.

The TB distance item is read in and its value saved.

**End of List:**

The end of list item resets the LOCK and LIST cells. The use of these quantities is explained in the paragraphs entitled "instructions".

**The Assign Subroutine:**

The assign subroutine determines which physical fast register to attach to a given symbolic register. The parameters given the subroutine are the register reference word, the position and number of the distances recorded by VIIIa and the current SQ for this reference. These parameters are first listed to determine if the reference is absolute, preassigned, or to a locked register. If so, the appropriate action is taken and the subroutine exited immediately. If not, the main body of the routine is entered.

The assign subroutine selects a physical register by examining each of the registers as a possible assignment. Several numbers are calculated for each possible assignment and are compared with the best assignment thus far found. If better, this assignment is substituted for the "best so far" and the process continues. The criteria used, in order of significance, are:
1. A penalty count representing roughly the number of additional instructions introduced into the code by this assignment.

2. The inverse sum of distances to next usages of any quantities already is the register.

3. The number of "index committed" registers used by the assignment.

4. The number of "arithmetic committed" registers used by the assignment.

5. The number of previously unused registers used by the assignment.

6. The size of the group of free registers in which the assignment appears.

7. The number of the physical register assigned.

Two of these criteria vectors are compared element by element until an inequality is found; the assignment with the smaller element is considered best. The effect of a possible assignment is kept in a set of tables called the action tables. Actually, two sets are maintained—one for the current assignment and one for the previous best assignment. These tables are called T0 through T9 and contain the following information:

T0: The criteria vector described above.
T1: Any fetches which must be inserted into the program.
T2: Any stores which must be inserted into the program.
T3: New histories which must be inserted into the register usage table (see below).
T4: Entries in the register usage table which are to be converted to secondary histories (see below).
T5: Entries in the register usage table which are to be deleted.
T6: Not used.
T7: Used to initialize the other tables.
The register usage table, RUT, is a matrix containing seven entries for each available fast register. Entries in the RUT are divided into two classes: primary and secondary. A primary entry is one which represents a quantity residing in that particular register. A secondary entry is one which represents a future usage of a register. For example, consider the code:

\[
\begin{align*}
F & \prec X \\
ME & \prec Y
\end{align*}
\]

When assigning \( \prec \) for the \( F \) command, an entry is made in two successive rows of the RUT. The first represents the register \( \prec \) containing an actual quantity and is primary. The second represents the register \( \prec + 1 \) containing the second half of the product \( x \times y \). In this manner, assignments may be made in the light of the commitment of a register to a certain quantity even though that quantity does not yet reside in the register.

Attached to the RUT are two other matrices called DIST and ULTD which contain the distance and ultimate distance respectively associated with the RUT entry. DIST is the SQ of the next references to the quantity and ULTD is the SQ of the last such reference. A fundamental rule of formation of the RUT is that it contains no conflicts. Two entries, \( E_1 \) and \( E_2 \), conflict under the following conditions:

1. Both \( E_1 \) and \( E_2 \) primary.
   Always a conflict.

2. \( E_1 \) primary and \( E_2 \) secondary.
   A conflict provided that ULTD \( (E_1) > \) DIST \( (E_2) \).

3. Both \( E_1 \) and \( E_2 \) secondary.
   A conflict provided that DIST \( (B_1) < \) DIST \( (E_2) < \) ULTDCE \( E_1 \) or
   
   \[
   \text{DIST} (E_2) < \text{DIST} (E_1) < \text{ULTD} (E_2).
   \]

In brief, a conflict exists when a register is simultaneously committed to two quantities, either now or in the future.
The heart of the assign routine consists of three nested loops: the I, J, and K loops. The variable I specifies which of the physical registers is under consideration as the assignment for the first segment of the symbolic register being assigned; the variable J specifies which segment of the symbolic register is being treated; and the variable K, which of the seven entries in the RUT is currently being tested for conflicts. Note that the J loop is usually trivial, since symbolic registers are usually single precision.

When an entry in the RUT is examined for conflict with the symbolic register being assigned, appropriate information is filed in the action tables under the assumption that this is the physical register to be assigned. For example, if the entry contains a common subexpression which has not been previously sent to temporary storage and the current register being assigned is a secondary arithmetic quantity, then an entry is made in T2 to specify that the common subexpression be stored and an entry in T6 to specify the primary RUT entry be made secondary. If considering the RUT entry as secondary still produces a conflict, an additional entry is made in T7 to specify that the RUT entry be erased completely. If the common rule expression were a double-precision register where other part was secondary, then that secondary usage is meaningless if not associated with a near-by primary.

This example should serve to depict the sort of processing that occurs within the loops of the assign routine.

After all possible assignments have been considered, the action tables representing the optimal assignment are processed and the actions represented there are carried out; i.e., the required instructions are introduced into the object code and the RUT is updated. In addition, any empty positions in the entry condition table are updated. The exit condition table is always updated.

It should be noted that if it is specified that an index quantity be fetched, and the corresponding ENCT position is blank, the fetch is not actually produced. Phase VIIIId will insure that the index quantity is present when the code is executed. If not blank, an index fetch request item is sent to File 8.4, and Phase VIIIle produces the actual fetch command.
PHASE VIIIc – Build Flow Tables

The duty of Phase VIIIc is to prepare a "flow-chart" of the problem being compiled. This "flow-chart" consists of two tables called UV and P. The P table consists of one-word entries of the following form:

```
S S S S S O E L L L L L
```

An entry is made in the P table for each entry point (continue item) and each transfer (successor item) in the object code. Two possible orderings of the continues and successesors are discernable: Entry Order and Exit Order. In Entry Order, each continue is followed by those successors which reference it. In Exit Order, each continue is followed by those successors which leave the block initiated by the continue.

The P table is formed in Entry Order. The Exit Order is implied by the L field. The L field contains the position, relative to the beginning of the P table, of the next entry in Exit Order. Thus, the P table may be processed in either order—stepping directly through or following the chain as specified by L. The S field is the major sequence number of the continue or successor item which produced the entry. The E field is the extensibility flag copied from the successor item. It is zero for continues.

The UV table is merely a list of those positions in the P table which contain continue entries. An additional entry is made in the UV table which references the first all in the P table not containing information.

An additional table called T is used to associate Exit Order with Entry Order. If an item is in position i in Exit Order and position j in Entry Order, then the M portion of T contains the value j. The T table also contains in position j the inverse level (see Phase V description) of the jth item in Entry Order.
The input to Phase VIIIC is File 8.2. This file is arranged in Entry Order by sorting on the first word and read into memory in its entirety. For each item read, the serial field representing Exit Order is stored into the T table, the item rearranged so that it may be sorted to Exit Order, and the position in the item in Entry Order is attached. After the end of File 8.2 is reached, it is sorted to Exit Order and each item receives the value of i from the one following it. This forms the L field described above. The file is then returned to Entry Order and the P and UV tables are filed. All is now ready for Phase VIIId.

**PHASE VIIId – Index Analysis**

Phase VIIId insures that index quantities needed at the beginning of a block are actually present. When required, index fetch request items are manufactured and sent to File 8.4 for subsequent processing by Phase VIIIE. Two sources of information are combined to determine the fetches required: the "flow-chart" constructed by Phase VIIIC and the Entry and Exit condition items developed by Phase VIIIB and recovered in File 8.3.

File 8.3 is sorted first on physical register and within physical register into Entry Order. Each physical is treated separately, and the fetches required for it throughout the entire program are determined before progressing to the next register. Information is read from File 8.3 in groups on the physical register. Groups not containing indexes are ignored. It is stored into a precleared table in the position corresponding to its entry order.

The serial number (NI counter) is used to reference the T table and thus determine the proper positions. Since the serial is not known for special continued items, their sequence number is looked up in that part of the P table which corresponds to entries in the UV table. If the position so located is empty, the item is simply stored there. If it contains an arithmetic quantity, the special continue supercedes it. If, however, another index is found, that index must necessarily be fetched. A file 8.4 item is produced before the special continue is stored.
After a group has been obtained, an iterative process is performed. The iteration is continued until a maximum number of passes have occurred (as specified by CRNOOP in the communication region) or until no further information can be obtained. Then a final pass occurs which produces the actual fetch requests on the basis of the information obtained.

The iteration process determines whether or not an index quantity can be guaranteed to exist in the register when a block is entered. Such a guarantee is possible if all successors referring to a given continue are identical. If so, that continue is marked as containing the index. If not, and all the successors which reference a given continue contain the same index insofar as they contain anything, then that continue is unknown and no updating occurs. If, however, two successors contain different quantities, then the continue is marked as ambiguous. Whenever information is gained by either discovering that an index does indeed exist at a block entry or that the register is ambiguous, then that information is passed on to these successors which are themselves unknown until a successor is found. This "passing on" of information is accomplished by processing the items in Exit Order. Thus, the next pass may well yield information about other continues.

One refinement to the above process has been built in. If the successors entering a block disagree only by virtue of some of them being unknown, and those which are unknown are exits from the same block that they enter; i.e., represent simple loops, then it may be assumed that the common known value is then available at the continue.

When the iteration terminates, each known continue is compared with the successors referring it. If they agree, no fetch is required. If none agree or if some do, and any of those which do not are non-extensible, then a File 8.4 fetch request is produced which will cause the desired index to be fetched at the continue. If all of those successors which disagree are extensible, then the File 8.4 fetch request is produced to fetch the value at the successor. This scheme effects an excellent compromise between time and space efficiency in the object code produced.
PHASE VIIIe - Produce F and S Commands

Phase VIIIe combines the index definition items produced by Phase VIIb and the index fetch items produced by both Phase VIIb and Phase VIIId to produce fetch and store commands. The input is from File 8.4 and the output is to File 9.4. File 8.4 is previously sorted on three characters (six digits) so that all references to a given symbolic index register are brought together with the fetch request items preceding the definition items.

Phase VIIIe reads a group on six digits into the FETCH area. If the data obtained represent definition items, they are discarded immediately, since no fetch requests are present for that particular symbolic index register. Whenever a group of fetch requests is found, the corresponding group of index definition items is read into the STORE area. The two words of each of the items in STORE are reversed so that the sequence number of the definition point is on the left and the entire group is sorted into ascending sequence on the sequence number.

Now for each item in FETCH, the following operations occur:

1. The STORE area is scanned from the beginning until an entry is found whose sequence number is greater than that of the fetch request. Then the previous definition is flagged (by setting the three high-order digits of the symbolic index name to zero).

2. The level number of the definition so marked is compared to that of the fetch request. If they are not equal, the remainder of the STORE area is scanned until that definition item is found with the greatest sequence number such that the level number is equal to that of the fetch request. If such an item exists, it also is flagged as above.

3. A F command is formed from the fetch request item and written to File 9.4.
After all the entries in FETCH have been processed, the STORE area is once again scanned. For each entry which has been flagged, an S command is generated and written to File 9.4. Control then reverts to the beginning and the next symbolic index register is processed.

The F and S commands generated are given the sequence numbers supplied with the File 8.4 items. The A register is likewise obtained. The M field is marked as temporary storage. The particular temporary storage is determined by a counter which is advanced by one for each group of fetch requests processed.

The algorithm described here guarantees that only those definitions are converted to S commands which are absolutely required for each F command to have a meaningful operand.
PHASE VIII FILE FORMATS

File 8.0 Items

Instruction item

W1

上

Minor Sequence

W2

0 0

Dictionary Reference, etc.

W3

0 0 0 0

Register Reference Word - A field

W4

If the address mode is less than 10, the W4 is the register reference word for the B field and has the same format as W3. If the address mode is 10 or greater, W4 contains the specific literal referenced and is formatted appropriately for that literal.
The Address Modes are
00 Absolute Address
01 Self relative forward (HERE +)
02 Self relative backward (HERE -)
03 Symbolic
04 Temporary storage reference
05 Fast register reference
06 Symbol with same symbol as marker
07 Symbol with special marker

{ Literals - See Phase IX description }

40 Register definition mode
50 Beginning of List mode

The Register Modes are
00 Absolute register reference
01 Index register
02 Arithmetic register
03 Common subexpression
04 Pre-assigned reference (with #)
05 Parameter storage reference
The precision number describes which of the segments of a symbolic fast register is active for this particular reference. The four possible segments are arranged as follows:

![Diagram of symbolic fast register segments]

- **Continue Item**
  - **W1**: Minor Sequence
  - **W2**: Dictionary Reference
  - **W3**: Unused
  - **W4**: Inverse Level number

The Continue flags are:
- 0 - Normal entry point
- 1 - Introduces Authentic Statement Function
- 2 - Introduces SAL block
Successor Item

The Extensibility flags are - 0 - Non-extensible
1 - Extensible

A successor is extensible implies that Phase VIII may insert instructions at the successor without disturbing the registers used in the code that immediately follows and without invalidating any self-relative references.

Instruction Literal Item
Dictionary reference of inserted continue

Sequence numbers of first usage as a literal (if any).

Sequence number of second usage as a literal (if any).

End of List Item

Minor Sequence

Major Sequence

Item Identification

Unused

Unused
File 8.5 Items

File 8.5 items have precisely the same format as file 8.0 items with the exception that library call items may occur in file 8.5. A library call item has "-0" for its item identification. As library call items are only passed through Phase VIII without processing, no detailed item format is given here.

File 8.1 Items

File 8.1 is essentially a variable length item file. The file routine is coerced into treating variable length items by describing the file as consisting of one-word items and using the 17-word-read and n-word-write options. Its contents are essentially those of the merged input from files 8.0 and 8.5 with information added as developed by Phase VIIIa. Each item produced by Phase VIIIa is suffixed (not prefixed as file 8.1 is read backwards) by a word containing the item identification and a count of the number of words (not including this one) in the item. This suffix allows Phase VIIIb to read the remainder of the item in one operation.

Instruction Item

![Diagram of Instruction Item]

Words W1 through W4 of File 8.0 instruction items

SQ First "A" Last "B" Suffix
Word Reference Reference

The SQ word is

![Diagram of SQ Word]

Current SQ for "B" reference

Current SQ for "A" reference

The "A" and "B" reference words are

![Diagram of Distance to Reference]

"-" implies that this reference is prior to the register definition item (C.S.E. only)
The total item may contain up to four "A" reference words and four "B" reference words.

The suffix is

Continue item

Successor item
Instruction literal item

Words W1 through W4 of File 8.0 instruction literal

The suffix is

Index map item

Each index map word records the usage of ten fast registers and is of the form

Preferred register flags
0 - not preferred
1 - preferred

The suffix is
TB distance item

Distance to next TB command

Suffix

End of List item

The End of List item is merely a flag and carries information only by its presence. Hence, only the suffix is required.

Suffix

File 8.2 Items

Continue item

Level Number
Serial Number
Item Identification
Major sequence of continue from File 8.0

Serial Number
Major sequence of routine from File 8.0
Successor item

1

Level Number
Serial Number
Major Sequence of continue referenced

0
Serial Number
Extensibility Flag
Sequence Number of successor from File 8.0

File 8.3 Items

Entry Condition item

0
Serial Number
Item Identification
Sequence Number of continue
Fast Register assigned

0 0 0 0 0 0
Symbolic Name of Register
Condition
1 - index quantity
2 - arithmetic or C. S. E.

Note: for condition 2 the symbolic name is not required and is always made zero.
Exit Condition item

Special Continue item

File 8.4 items

Fetch Request item
Sequence Numbers

Index Definition item

[Diagram showing sequence numbers with annotations for Fast Register assigned, Level, Item Identification, and Symbolic Name of Register]
Set diagnostic entry to 8Acru
Set 5DD to ++8+1

Open F80, F84 to read back
Extend F84.
Open F81 to write.

Read F80 to I80
EOF \rightarrow -1 \rightarrow I80

Read F85 to I85
EOF \rightarrow -1 \rightarrow I85

Is I85, negative?
Yes \rightarrow L18

99999 \rightarrow T8DIST

Write dummy continue to F81

OFF

CRNOAR:10 \rightarrow CRNOAR-7
NRM4 = CRNOAR-7 at 6
ON

8 \rightarrow
LARC SCIENTIFIC COMPILERS
Phase VIII A
BACKWARD SCAN

I: Successor Item.

Form F81 Successor (2 words + ID = 2)

Write to F81

R: End-of-List Item.

Write ID=6 to F81

A

H: Continue Item.

Form F81 Continue (2 words + ID = 1)

Write to F81

Is Item<0

Yes

D

No

i = 0
n = [KERNEL]/107
j = 0

ITEMj = 0
K = 0

MAPi : 0

= ≠

ITEMj + 10^K → ITEMj

i + 1 → i
K + 1 → K

k: 10

j + 1 → j

Note: When Item<0 the continue does not represent an entry point, but names an instruction to be modified.
LARC SCIENTIFIC COMPILER
BACKWARD SCAN

M

Form NOW word from RQSTo and SQ
Form ID word from TEMPI and J

Write ITEM to F81

D

J

Instruction Literal Item

Form F81 instruction literal item.

Write to F81 (4 words + ID = 3)

D
LARC SCIENTIFIC COMPILER

BACKWARD SCAN.

1. T

   Analyze register mode.

   0. Absolute
   1. Arithmetic
   2. Index
   3. C.S.E
   4. Preassigned
   5. Parameter

   0

   1

   0

   1

   Register name > 6000

   #0 → MAP

   i = (Register name) mod (CRNOR - 9)

   #0 → MAP i + 7

   I = 1
   J = 0

   Prepare table search argument for Ith segment of register.
LARC SCIENTIFIC COMPILER

FORWARD SCAN.

Phase VIII B

8.34

A

Set dump linkage and flash ++ 8 + 2

0 → NI
0 → ENTRY + 1
0 → LEVEL
996 → NAMER
499 → TEMENT
0 → IL
Set IF, P, T
Form B-Box words.

B2

Read IDENT from FB1

EOF → B3

Form B-Box word to read variable length item.

Branch on identification

IDENT

A | ID | N
---|----|---

A: Number of A segments for instruction item.
ID: Item identification.
N: Number of words in item.

B4

Instruction

B5

Continue

B6

Successor

B20

Instruction Literal

B7

Index Map

B30

T0 Distance.
Read instruction into IRITHA from F81

Initialize don't table O INSSER
Form NUMA NUMB PSTNA and PSTNB from IDENT

Is instruction a Fetch (43)?

Set α off

III1 → EFFECT
1 → ABM

Address mode ≠ 10

Address mode ≠ 5

(Reference is parameter)

B6A = Register mode : 5

ABM = 0 Assigning A
1 Assigning B
2 Assigning M

Address mode = 10 (Reference to literal)

B16 (Other)
Is the B register single precision and not altered?

Is the register arithmetic or a CSE?

Search SHT for B register

Form EFFECT for B-reference

Enter C with parameters for B-reference

Convert Address Mode to refer to Parameter Storage.

Is B-reference distance 00?

X3 (DREF)

Assignment → REFB

B17

No

Yes

No

NOT THERE

THERE

04 → Address mode Cell marked in SHT → DREF

B14

B16A

B16

B15
B5 Continue Item.

Read continue item into CNTNU from F81

CNTNU1:0 ≥ 0 → REGSYM1

Set T

X1 (AFRST, SQNE CNTNU0)

Reset T

Test continue flag.

0 → BSD
1 → BSF
2 → BSE

Normal
Erasable
Arithmetic Function

B5E

Clear AFRST
i = 1, ..., CNFNOAR

Reset T

Major sequence of CNTNU → SQNE

BSD

Form F90 continue item in AREA

Write AREA to F90

CNTNU1:0 ≥

A.CNTNU1:0 ≠

A.CNTNU1 → LEVEL

↑ Reset

BSD

Set

BSD

BSF
LARC SCIENTIFIC COMPILER
FORWARD SCAN

880  Instruction Literal Item.

Read instruction literal item from F80 into INSLIT

1 → IL

82

830  TB Distance Item.

Read TB dist. item into TBDIST

EOF → B10

82

840

99 → LOCK0, LOCK1
99... 99 → LIST

Enter Error Routine:
with option 1
The Assign Subroutine

Save parameters NUMR, PSTNR, RRW and Count

C

0 → Good

Register mode of RRW: 0

C2

> N

C1

RRW: LIST

≠ CIA

0 → BASE
100 → ALTER
999 → T01, T01
0 → T00, T00
∞ → TO2

Set W

Use LOCK for assignment

Obtain assignment

C8

EXIT

Note: Assignment is negative for absolute references and positive for "#" references

C3

Set up RRTi from RRW and (PSTNR + I)

I + 1 → I

≤ I: NUMR

Remark: TYPE =

> 0 C.S.E

= 0 Index

< 0 Arithmetic

LAST = CRNOAR + NUMR - 1

EFFECT = EFFECT X 10 - (ASSIGN p-1)

Reset

Set

≤ (M RRW: 59999)

RES = 0

= TYPE: 0

≠ (M RRW - 7) Mod (CRNOAR - 7) + 7

C4
Note: The operation "count" specifies the following open subroutine.

2x count specifies that the subroutine be included twice.
LARC SCIENTIFIC COMPILER

FORWARD SCAN

Note: Sum is kept in register Z2.

Phase VIII B

D17

PRIME = 0

ULTDJ,K > RRTNHQ

D7

D18

Contribute DISTJ,K to sum

Is RUJ,K single precision?

D9

Y

E

2X COUNT

E

2

D9A

Is RUJ,K an index?

Yes

No

Yes

No

COUNT

COUNT

F

E

D18

Insert (J,K) in T6

DIBA

COUNT

1

2

3

Y
LARC SCIENTIFIC COMPILER

Phase VIII B

FORWARD SCAN

I3F

= ENCFJ: 0

≠ ISM

INSEQ + 1 → INSEQ
Form F84 fetch request from INSEQ, TIb+i, and LEVEL

Write to F84

M, TI b+i → ENCFJ, and ENCFJ

Is quantity to be fetched an index?

Yes

No

INSEQ + 1 → INSEQ
Form Memory to Register Fetch from INSEQ, SHT and TI b+i

Write Fetch to F90

Is TIb+i flagged "-"?

Yes

X3

No

Delete corresponding SHT entry

i + 1 → i

i : T9b+i ≤ I3L

> I5

IS
LARC SCIENTIFIC COMPILER
FORWARD SCAN

I7E

Is RRTi primary?
Yes

K ← EXT TO B+i+7

ENCT TO B+i+7

ABM:0

≤

Set

EXIT

#O ← FIRST TO B+i+7

No

Reset

I7A

P

Reset

#O ← FIRST TO B+i+7

Set

I7A

ABM:0

= 

I7A

≠

Yes

I7A

Is i-th digit of EFFECT a 1?

No

Form DON'T Table entry from TO B+i+7

Flag entry "L"

Insert in DON'T DON'TH
DON'TN+1 → DON'TN

ABM:2

= 

I7A

≠

Reset

i+1 → i

i:NUMR

≥

TOB ← GOOD

Reset

Set

I7A

Reset

I7G

I7H

K = 20000
Subroutine to save and restore fast registers for arithmetic functions or subprograms.

Form F84 fetch request for #i with NAMER and specified sequence number.

Write to F84

Form corresponding definition point.

Write to F84

NAMER + 1 → NAMER

XIB

i + 1 → i

<

i : CRNOAR

≥

EXIT
Subroutine to obtain the name of a temporary storage location.

Subroutine to mark a temporary storage location as again available.
Set diagnostic entry and display
++ 8+3

NI : 1649

Open F82 to read.

I → I

B

I+1 → I

Re-arrange item, attach I, and store at SI

Start with SI = 1

Sort S to exit order.

LINK : S0 → START
0 → I

D

LINK : SI+1
I+1 → I

Sort S to entry order

0 → I
0 → K

Is SI a continue?

Yes

I → UV(K)
K+1 → K

K : 1000

<

Form P(I) from S(I) and store.
I+1 → I

<=

K → KMAX

To Phase VIII D

Write error mess.
Table size exceeded

8RTRN
LARC SCIENTIFIC COMPILER

INDEX ANALYSIS

Phase VIII D

T

Form FETCH Item (ID=0) for G(I) in RAST

Write RAST to F84

Set FF0

K

D

Reset FF2

Set FF2

E

D'

Close F83 and F84

Flash -- 8-5

Restore diagnostic entry.

SORT F84

To Phase VIII E
Set diagnostic entry.
Flash ++8+S
Open F84 is read.
\( \gamma = 0 \)

Read a group on
register name and ID from F84
into FETCH

\( \text{ID}. \text{FETCH} = 0 \)

Number of items \( \rightarrow N \)

Read a group on
register name and ID from F84
into STORE.
(m items)

\( i = 0 \)

Reverse order of
words in each item in STORE

\( i + 1 \rightarrow i \)

\( i : m \)

Compare sequences
\( \text{FETCH}(i) : \text{STORE}(j) \)

\( j + 1 \rightarrow j \)

\( j : m \)

Set FF0

Mark STORE \((j-1)\)

\( \text{Compare levels} \)
\( \text{FETCH}(i) : \text{STORE}(j-1) \)

\( k = -1 \)

Set FF0

Reset
**LARC SCIENTIFIC COMPILER Phase VIII**

**PRODUCE F & S COMMANDS**

---

**Diagram Explanation**

1. **H**: Compare levels
   - **FETCH(i)**: **STORE(j)**
   - If ≠, go to F.
   - If =,
     - **k = j**
     - **j = j + 1**

2. **I**: Is **STORE(j)** marked?
   - **Yes**:
     - Form S command from **STORE(j)** using **J** for name of temporary cell.
     - Write to F94
     - **J**
     - **j + 1 → j**
     - **j → m**
     - **v + 1 → v**
   - **No**

3. **F**: Form F command from **FETCH(i)** using **V** for name of temporary cell.
   - Write to F94
   - **i + 1 → i**
   - **i → Z**
   - **j = 0**

4. **C**: Return to control routine.

---

**Notes**

- **RTRN**
  - Restore diagnostic entry
  - Flash **9**
  - Reset all **F**'s.
PHASE IX

GENERAL DESCRIPTION

Phase IX is the editing phase of LSC; it is divided into two parts. Phase IXa generates error messages from probable source program errors detected by Phases I through XIII. These errors are filed as two word error items in File 92 and the messages are written in File 93. Phase IXb produces the SAL lines from the items in File 90 and File 94 and merges them with the original source statements and error messages, File 93, to create the object program, File 10. The dictionary, File 91, is available throughout Phase IX to provide the parameters for the error messages and the names to replace numeric dictionary references in instruction items.

When the object program has been completely generated, Phase IX writes LSCOMPILED on the typewriter, reinitializes the communication region, and returns to the control program. Refreshing the communication region permits successive compilations without an external restart of LSC.
Phase IXa writes an LSC error diagnostic message in File 93 for each probable source program error detected by Phase I through XIII. These detected errors were written in File 92 as two word items. The first word contains, the phase and error number, the sequence number to be assigned to the error message and an indication that the message does or does not contain a parameter. The second word will contain the parameter or the dictionary reference of the parameter. The error message skeletons which are assembled into Phase IX are located by phase and error number; a % character within the skeleton indicated the position for the parameter, if any. Phase IXa will delete all blanks following the parameter name since the skeleton has the proper number of blanks to obtain the desired spacing. The item written in File 93 is as follows:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>F - ER</th>
<th>Diagnostic message</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>W2</td>
<td>W3 W11</td>
</tr>
</tbody>
</table>

Detailed descriptions of file 91 and 92 are appended to this section of the manual.
GENERAL DESCRIPTION

After Phase IXa, File 93 and File 94 are sorted (independently) on sequence. File 93 contains the source program statements, ALPH lines generated by Phase I from Format statements and Hollerith arguments, and the error messages generated by Phase IXa. File 90 and File 94 contains pre-SAL items generated by Phases IV and XIII; these items are to be edited into SAL lines and merged with File 93 according to the assigned sequence numbers to form File 10, the object program.

OUTPUT OPTIONS

File 10 is written on drum in an area specified by the communication region. The output may also be printed under user option by presetting of two other words in the communication region as follows:

1. CRHSP = 0; no printing

2. CRHSP ≠ 0 and CROUT = 0; output on on-line high speed print number one

3. CRHSP ≠ 0 and CROUT ≠ 0; output on the uniservo specified by CROUT for off-line listing

DESCRIPTION OF FILE 10

The first line of the object program is a SAL LABEL line. The user may select any name which is acceptable to SAL for his program as long as the first character is alphabetic (to avoid possible conflict with LSC generated symbols). When compiling a SUBROUTINE or FUNCTION, the name of the subprogram will be the symbol on the LABEL line of File 10. For main programs, the symbol will be LSCODE unless
the first line of the source program is a SAL LABEL line; in this case, the name on the LABEL line will be the name of the object program. A directory may also be specified by the user for inclusion in LSC compiled programs, except for SUBROUTINE and FUNCTION. If a directory is to be specified it must appear immediately after the LABEL line, if any, in the source program and conform to SAL specifications, particularly (for recognition by LSC) beginning with DIRECTOry and terminating with ENDIRECTORY. The directory is merely copied by LSC without attempting to detect errors within or initiated by the included directory. In the absence of a directory, the standard SAL directory will be used by the assembler.

For main programs, the next line 10 will be a transfer to the first executable instruction compiled unless the source program contained SAL coding before the first LSC statement which generated executable code. In the later case, the transfer will be to the first SAL statement.

For SUBROUTINE and FUNCTION subprograms, the LABEL is immediately followed by the VARIABLES line (s). The "variables" will be 9COM, 9TBL, 9STB and all required I/O programs, math, library routines, all referenced external functions. These "program variables" should generate LIB instructions for a main program by the operation program. There is no analogous transfer instruction as in main programs, since entry to subprograms is accomplished with a TB instruction to the name. These leading lines are followed by the merged original source statements, comments, and LSC generated instructions and diagnostic messages. The origin of these lines is indicated in columns 1 to 6 of File 10.

<table>
<thead>
<tr>
<th>Columns</th>
<th>1 2 3 4 5 6</th>
<th>origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>source program LSC statement</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>source program comment</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>source program SAL line</td>
<td></td>
</tr>
<tr>
<td>F - ER</td>
<td>LSC error diagnostic message</td>
<td></td>
</tr>
<tr>
<td>(blank)</td>
<td>LSC generated line</td>
<td></td>
</tr>
</tbody>
</table>
Each printed line of the object program will contain a % sign followed by the line's twelve digit sequence number; this number is not included in File 10 on drum.

Following the END statement LSC writes an END OF TAPE line in File 10 to signal to SAL the end of the program to be assembled.

A description of and corresponding examples for File 94 and File 90 items appear below.

NAME GENERATION

LSC renames source program names only when they might be internally doubly defined or incompatible with SAL requirements. There are three categories which are renamed.

1. Numeric statement names are appended with L. (e.g., 17 becomes 17L),

2. Special SAL lexicon names (HERE, SAME, SEGii) are prefixed with 9, (e.g., SEG 48 becomes 9SEG48),

3. References to the name of a FUNCTION are treated as references to the name prefixed by 8. (e.g.) FUNCTION PIE (A, PP, LE) . . . could generate:

<table>
<thead>
<tr>
<th>PIE</th>
<th>F</th>
<th>#05</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>#05</td>
<td>PP</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>#05</td>
<td>LE</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>#05</td>
<td>8PIE</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION NAME
There are six categories of names that are generated by LSC. All of them have a numeric as the first character. These categories are:

1. COMMON storage is reserved via the symbol 9COM which is itself defined in one of two ways:

   A. If CRCOMM = 0 then: $STOR 9COM: N

   B. If CRCOMM = 0 then: DEF 9COM: (CRCOMM)

   C. If the program is a SUBPROGRAM, 9 COM is not defined.

2. Subscripted variables are located relative to 9COM or 9DAT by defining them equal to a symbol generated from their sequence number appended with A. e.g., DEF JEAN: 18A - 6 and DEF 18A: 9DAT + 147

3. Generated continues (statement names) are formed by appending X to the sequence of the item to which the name should be attached.

4. Temporary storage locations necessitated by the generated code are named according to type of quantity being stored and a sequence number (denoted as ijk): The types of storage are:

   A. Index $9Xijk
   B. Parameter $9Pijk
   C. Arithmetic $9Tijk
   D. Subprogram $9Mijk or $9Mijk; 2
5. Generated instruction literals are named 9IT\text{ijk}.

\begin{verbatim}
e.g., column: 12 17 18 19
F #13 9IT012
9IT012 K TG #21 JEAN
\end{verbatim}

6. A dictionary reference (i.e., name) is created for each argument of a SUBROUTINE, FUNCTION, or ARITHMETIC STATEMENT FUNCTION immediately after the name of the program. This name is generated from the dictionary entry number appended by AR. e.g., SUBROUTINE KSFO(AARON, DON, JIM)

<table>
<thead>
<tr>
<th>File 91 entry</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>KSFO</td>
</tr>
<tr>
<td>94</td>
<td>94AR</td>
</tr>
<tr>
<td>95</td>
<td>95AR</td>
</tr>
<tr>
<td>96</td>
<td>96AR</td>
</tr>
</tbody>
</table>
FILE 91 DICTIONARY FILE

File 91 consists of one word items containing one of the following:

1. 12 digits of alphanumeric information representing a given name (left adjusted).

2. The low order part of a double precision numeric quantity in floating point.

3. \[ Z \ A \ A \ A \ A \ O \ g \ g \ g \ g \]

\[ Z = 0 \] for positive \( A \ A \ A \ A \)
\[ Z = 1 \] for negative \( A \ A \ A \ A \)

\( A \ A \ A \ A \) represents a numeric quantity to be added or subtracted (according to \( Z \)) from the alphanumeric symbol located in File 91 by \( (gggg) \)

The (ffff) digits of the mode word of an item, which has a dictionary reference, will locate one of the above three words. If the sign is "-", then type (3.) above will apply.

FORMATION OF FILE 91:

At the start of each compilation DREF is set equal to 1, prior to filing a word in File 91. The program uses the contents of DREF for the dictionary reference digits (ffff) in the mode word. The FILE program then advances DREF when the word is filed.

Phase I files names of library programs, I/O subroutines, the low order part of any double precision number, and the generated name of the Hollerith arguments of a CALL statement in File 91. Phase III files a name (in the dictionary) for each
group hence all references to the same name will have the same value (f f f f) in their mode words. Phase V generates a word for subscripted references to variables (format 3, above) whenever there is a numeric quantity (\( \pm A A A A A \)) to be added to a symbol for instruction generation. In this case Phase V replaces the references to the name with reference to the word containing the increment and the original File 91 reference.

FILE 92 ERROR ITEMS FILE

Each LSC phase will write any detected error as a two word item in File 92. These items will generate error messages in File 93 during Phase IXa.

File 92 Item Format

<table>
<thead>
<tr>
<th>Word 1</th>
<th>Word 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>

P: Number of the phase that produces error item.
E: Error item number
S: Sequence number of the item that generates the error.
W: Parameter for error messages (if there is one).
A: Determines mode of W as follows:

<table>
<thead>
<tr>
<th>A</th>
<th>Mode of W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>alphanumeric (left adjusted)</td>
</tr>
<tr>
<td>1</td>
<td>numeric; use W as name dictionary reference to get error message parameter</td>
</tr>
</tbody>
</table>

B: Degrees of fatality of error.

B = 0, not so serious
B = 1, serious source program error
The error messages themselves are in sort on the phase and error number (P and E), they have the following form:

\[
\begin{array}{cccccc}
P & E & C & (\text{Message}) & \% & (\text{rest of Message}) \\
W1 & W2 & & & W10 \\
\end{array}
\]

The entire item has 10 words.

\[
\begin{align*}
C &= 0, & \text{there is no parameter} \\
C &= 1, & \text{there is a parameter} \\
\end{align*}
\]

The "\%" character is in the position in the message where the parameter is to be placed (if there is one).
STORAGE RESERVATION AND DEFINITION ITEMS.

The file 10 items generated by these file 94 items are best described by examples of the original items and the generated SAL lines.

Assume that a portion of the dictionary (file 91) is:

<table>
<thead>
<tr>
<th>Entry number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>JIM</td>
</tr>
<tr>
<td>90</td>
<td>WES</td>
</tr>
<tr>
<td>91</td>
<td>JACK</td>
</tr>
<tr>
<td>92</td>
<td>KEN</td>
</tr>
<tr>
<td>93</td>
<td>BOB</td>
</tr>
<tr>
<td>94</td>
<td>TED</td>
</tr>
<tr>
<td>95</td>
<td>ERV</td>
</tr>
<tr>
<td>96</td>
<td>035597932385</td>
</tr>
<tr>
<td>97</td>
<td>PI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>F94 Item</th>
<th>F10 Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td></td>
<td>DEF Jim : 89A - 6</td>
</tr>
<tr>
<td></td>
<td>W1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequence</td>
</tr>
<tr>
<td></td>
<td>W2</td>
<td>0-002</td>
</tr>
<tr>
<td></td>
<td>W3</td>
<td>0-089</td>
</tr>
<tr>
<td></td>
<td>W4</td>
<td>0-06</td>
</tr>
<tr>
<td>03</td>
<td></td>
<td>DEF 89A : 9DAT + 14</td>
</tr>
<tr>
<td></td>
<td>W1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequence</td>
</tr>
<tr>
<td></td>
<td>W2</td>
<td>0-003</td>
</tr>
<tr>
<td></td>
<td>W3</td>
<td>0-089</td>
</tr>
<tr>
<td></td>
<td>W4</td>
<td>0-014</td>
</tr>
<tr>
<td>04</td>
<td></td>
<td>DEF 90A : 9COM + 17</td>
</tr>
<tr>
<td></td>
<td>W1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequence</td>
</tr>
<tr>
<td></td>
<td>W2</td>
<td>0-004</td>
</tr>
<tr>
<td></td>
<td>W3</td>
<td>0-090</td>
</tr>
<tr>
<td></td>
<td>W4</td>
<td>0-017</td>
</tr>
</tbody>
</table>
### LARC Scientific Compiler

#### Phase IX

**Storage Reservation and Definition Items (continued)**

<table>
<thead>
<tr>
<th>ID</th>
<th>F94 Item</th>
<th>F10 Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>0 - 006</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>0 - 091</td>
<td></td>
</tr>
<tr>
<td>W4</td>
<td>0 - 019</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DEF</td>
<td>JACK : 9DAT + 19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>0 - 007</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>0 - 092</td>
<td></td>
</tr>
<tr>
<td>W4</td>
<td>0 - 023</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DEF</td>
<td>KEN : 9COM + 23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>0 - 008</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>0 - 0</td>
<td></td>
</tr>
<tr>
<td>W4</td>
<td>0 - 083</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$STOR</td>
<td>9DAT ; 83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>0 - 009</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>0 - 094</td>
<td></td>
</tr>
<tr>
<td>W4</td>
<td>0 - 017</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DEF</td>
<td>TED : 17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>0 - 010</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>0 - 0</td>
<td></td>
</tr>
<tr>
<td>W4</td>
<td>0 - 077</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i) if (CRCOMM # 0) DEF 9COM : 80000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(80000 for example)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ii) if (CRCOMM = 0) $STOR 9COM ; 77</td>
<td></td>
</tr>
</tbody>
</table>
### LARC Scientific Compiler - Phase IX

**Storage Reservation and Definition Items (continued)**

<table>
<thead>
<tr>
<th>ID</th>
<th>FG4 Item</th>
<th>F10 Item</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Sequence</strong></td>
<td><strong>Sequence</strong></td>
</tr>
<tr>
<td>11</td>
<td>W1: <code>a l c d e e e 00095</code></td>
<td>W1: <code>a l c d e e e 00090</code></td>
</tr>
<tr>
<td></td>
<td>W2: <code>0 0123</code></td>
<td>W2: <code>0 0123</code></td>
</tr>
<tr>
<td></td>
<td>W3: <code>0 0123</code></td>
<td>W3: <code>0 0123</code></td>
</tr>
<tr>
<td></td>
<td>W4: <code>0 0123</code></td>
<td>W4: <code>0 0123</code></td>
</tr>
</tbody>
</table>

**Additional Diagram**

- **981D12** (Library Request)
- **Generate:**
  - Symbol: `(symbol)`
  - Marker: `(marker)`
  - `LIB COS COS`
- **980M1** (Output Item to F10)
Instruction Items

Instruction items in File 90 and 94 are four-word items with the following general format:

1. **W1**: Sequence.

2. **W2**:
   - Instruction item
   - B register
   - A register
   - Operation code
   - Tracing digit
     - \( T = 0 \Rightarrow \) no tracing digit
     - \( T = 7 \Rightarrow \) indirect addressing
     - \( T \neq 0,7 \Rightarrow \) indicated digit

3. **W3**:
   - Dictionary references, etc., according to address mode:
     - \( 00 \Rightarrow \) B is absolute
     - \( 01 \Rightarrow \) B is relative (i.e., #BA)
     - \( 00 \Rightarrow \) A is absolute
     - \( 01 \Rightarrow \) A is relative (i.e., #AA)
   - Address mode 00 to 07, 16 to 21

4. **W4**: Literal, if any. (Second dictionary reference for address mode 07)

The following examples assume that a portion of the dictionary (File 91) is:

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>146</td>
<td>TRUCK</td>
</tr>
<tr>
<td>147</td>
<td>HI7</td>
</tr>
<tr>
<td>148</td>
<td>COS</td>
</tr>
<tr>
<td>149</td>
<td>E88</td>
</tr>
<tr>
<td>150</td>
<td>FLO</td>
</tr>
<tr>
<td>151</td>
<td>.000000000000</td>
</tr>
<tr>
<td>152</td>
<td>-10110000147</td>
</tr>
</tbody>
</table>
### Address Mode

#### F90 or F94 Item

**absolute**

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 93 00 00 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 00 00 00 23461</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

**relative forward**

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 70 06 00 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 01 01 00 00002</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

**relative backward**

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 80 12 00 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 02 01 00 00017</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

**Symbolic**

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>7 48 08 22 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 03 01 00 00146</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

**Symbolic**

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 23 03 03 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 03 00 01 00147</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

**Symbolic**

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 90 00 01 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 03 00 01 20014</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
<tr>
<td>Address mode</td>
<td>F90 or F99 item</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>temporary (parameter)</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
</tr>
<tr>
<td>W2</td>
<td>0 40 16 00 000001</td>
</tr>
<tr>
<td>W3</td>
<td>0 04 01 00 000000</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
<tr>
<td>temporary</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
</tr>
<tr>
<td>W2</td>
<td>0 43 10 06 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 04 00 00 10211</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
<tr>
<td>temporary (arithmetic)</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
</tr>
<tr>
<td>W2</td>
<td>0 40 04 00 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 04 00 00 30052</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
<tr>
<td>temporary (arithmetic)</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
</tr>
<tr>
<td>W2</td>
<td>0 48 04 00 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 04 01 00 40551</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
<tr>
<td>fast register</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
</tr>
<tr>
<td>W2</td>
<td>0 32 11 07 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 05 01 00 00000</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
<tr>
<td>fast register</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
</tr>
<tr>
<td>W2</td>
<td>0 01 09 07 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 05 01 01 00000</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>
### Address mode

#### F90 or F94 item

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 92 00 00 000001</td>
</tr>
<tr>
<td>W3</td>
<td>0 06 01 00 00149</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

#### F10 item

| TB   | #00 00           |

#### marker & symbol

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 91 00 05 000001</td>
</tr>
<tr>
<td>W3</td>
<td>0 07 00 01 00149</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

#### floating literal

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 02 06 00 000001</td>
</tr>
<tr>
<td>W3</td>
<td>0 17 01 00 000000</td>
</tr>
<tr>
<td>W4</td>
<td>05 13 14 159 000</td>
</tr>
</tbody>
</table>

#### A #06 (0.314159(01))

#### double precision literal

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 16 00 00 000001</td>
</tr>
<tr>
<td>W3</td>
<td>0 18 01 00 00151</td>
</tr>
<tr>
<td>W4</td>
<td>-42 12345 00000</td>
</tr>
</tbody>
</table>

#### NN #00 (-0.12345(-08))

#### fixed literal

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 01 14 00 000001</td>
</tr>
<tr>
<td>W3</td>
<td>0 19 01 00 000000</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

#### AX #14 (81)

#### B Box literal

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 43 09 00 000001</td>
</tr>
<tr>
<td>W3</td>
<td>0 20 00 00 000000</td>
</tr>
<tr>
<td>W4</td>
<td>003 0031 00 852</td>
</tr>
</tbody>
</table>

#### F 09 (88 003 0031 00882)
**LARC SCIENTIFIC COMPILER**

**Instruction Items (continued)**

**Address mode**

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 43 20 00 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 21 00 00 00000</td>
</tr>
<tr>
<td>W4</td>
<td>0 -------------- 0</td>
</tr>
</tbody>
</table>

(Next item will be given the name of this literal.)

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>5 00 09 08 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 03 00 00 00146</td>
</tr>
<tr>
<td>W4</td>
<td>0 -------------- 0</td>
</tr>
</tbody>
</table>

**Temporary (index)**

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0 43 19 00 00001</td>
</tr>
<tr>
<td>W3</td>
<td>0 04 01 00 20079</td>
</tr>
<tr>
<td>W4</td>
<td>0 -------------- 0</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F 10 item.</strong></td>
<td><strong>F 20 K9IT123</strong></td>
</tr>
<tr>
<td><strong>K9IT123 SK 09 TRUE 08 5</strong></td>
<td><strong>F #19 $9X79</strong></td>
</tr>
<tr>
<td>W1</td>
<td>Sequence</td>
</tr>
<tr>
<td>----</td>
<td>---------</td>
</tr>
<tr>
<td>W2</td>
<td>0</td>
</tr>
<tr>
<td>W3</td>
<td>0</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

Name will be the contents of the 123rd entry in the dictionary File 91.

<table>
<thead>
<tr>
<th>W1</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0</td>
</tr>
<tr>
<td>W3</td>
<td>0</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
</tr>
</tbody>
</table>

Name will be: ...23579 X
LARC SCIENTIFIC COMPILER

CONTROL

Open and read F91 (dictionary)

End of file? No set S 9003

1. Close F92 (error items).
2. Open F92 for reading.
3. Extend F93 for writing.

Number of F92 items #

"NO ERRORS"

→ F93

F9A

Read F92 item.

End of file? Yes

Does message contain a parameter?

1. Sequence → word 1
2. "F-ER" → word 2
3. Error message → word 3 thru 11

F93

Is parameter in 2nd word of F92 item?

Get parameter from dictionary.

Yes

No

Replace each % in error message with parameter word.

No
1. Close F93 for writing.
2. Sort F93
3. Open F90, F91, F93 for reading.
4. Open F10 for writing.

Set up end of file conditions F90, F94

Set up [T (CRS)] as next F93 item.

Transfer to first executable statement

Output label and directory lines to file 10

(no directory) =

Read first item from F90 and F94

(CRLABL) = LSCODE of the name of a subroutine or function, if one is being compiled.

Read first item from F90 and F94

CRDRT : 0 ≠

CRESUBP : 0 ≠

Resubroutine (FUNCTION) ≠

(Subroutine) ≠

9BEGET
98PID00 (continue item)

col.12 IS 17 of LINE ≠

DREF: 10000 ≥
L0E → col.19 of LINE

Get name from dictionary

name → col.12 IS 17 of LINE

980TH2 (To get next instruction item)
Example - Non-literals

23461
  absolute
  \( m_{00} \)
  \( m_{16} \) (floating)
  \( (0.2 \pm 0.01) \)

Here + 2
  self-relative +
  \( m_{01} \)
  \( m_{17} \) (floating)
  \( (0.314159(01)) \)

Here - 17
  self-relative -
  \( m_{02} \)
  \( m_{18} \) (double-precision)
  \( (-0.12345(-08)) \)

Truck
  symbolic
  \( m_{03} \)
  \( m_{19} \) (fixed)
  \( (81) \)

$9 \text{ PO}$
  temporary
  \( m_{04} \)
  \( m_{20} \) (6 box)
  \( (68 003 0031 00832) \)

$9 \text{ T211}$
  \( m_{05} \)
  \( m_{21} \) (instruction)
  9IT 1 2 3

$9 \text{ A79}$
  \( m_{06} \)

$9 \text{ M52}$
  \( m_{07} \)

$9 \text{ M551,2}$

#07
  fast register
  \( m_{08} \)

COS & COS
  marker & symbol
  \( m_{09} \)

E8 & FLO
  marker & symbol
  \( m_{10} \)

\( 901C \) (branch on M mode ID)
Purpose:
Print F10 on high speed printer or specified Uniservo

Calling sequence:
TB #0 FILE+F10
H #0 LOC
TB #0 9PTNTL
Return

9PTNTL

RETURN (no printing)

CRHSP ;0

Has last printing been completed?

No

Move 10 words to print buffer; converting any unprintable characters to ~

Yes

Convert sequence number to dN. Move it to 13th word of buffer.

CRBUT ;0

Set up tape summary orders for tape specified by CRBUT

Set up HSP summary orders.

Issue summary orders.

RETURN