(NASA-CR-120264) COMPILER WRITING SYSTEM DETAIL DESIGN SPECIFICATION. VOLUME 2: COMPONENT SPECIFICATION (McDonnell-Douglas Astronautics Co.)

N74-27660

CSCL 09B G3/08

Unclas 16937

Reproduced by
NATIONAL TECHNICAL INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151

PRICES SUBJECT TO CHANGE

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY

MCDONNELL DOUGLAS CORPORATION

MCDONNELL DOUGLAS

COMPILER WRITING SYSTEM DETAIL DESIGN SPECIFICATION

VOLUME II

Component Specification

APRIL 1974

MDC G5359

PREPARED BY:

W. J. ARTHUR
MCDONNELL DOUGLAS ASTRONAUTICS COMPANY
ADVANCE INFORMATION SYSTEMS

PREPARED UNDER THE DIRECTION OF:

MR. B. C. HODGES
MARSHALL SPACE FLIGHT CENTER
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
UNDER CONTRACT NAS8-27202

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY-WEST

5301 Bolsa Avenue, Huntington Beach, CA 92647

COMPILER WRITING SYSTEM

DETAIL DESIGN SPECIFICATION

VOLUME II - COMPONENT SPECIFICATION

MDAC CONTRACT NUMBER NAS8-27202

"TECHNIQUES IN THE GENERATION

OF SUPPORT SOFTWARE"

30 APRIL 1974

Preceding page blank

PREFACE

This report is Volume II of the design specification for the Compiler Writing System. It will introduce the reader to the organization and structure of the system components and their relationship to the overall compilation process.

Volume I should be referred to for a discussion of the compiler definition and support languages.

This report has been prepared in compliance with the requirements of contract NAS8-27202, covering work done between 1 June 1973 and 30 April 1974. If additional information is required, please contact any of the following McDonnell Douglas or NASA representatives:

- Mr. G. M. Jones, Contract Negotiator/Administrator
  Huntington Beach, California
  Telephone: (714) 896-2795

- Mr. B. C. Hodges, Project COR, S&E-COMP-C
  Marshall Space Flight Center, Alabama
  Telephone: (205) 453-1385

Preceding page blank

TABLE OF CONTENTS

Preceding page blank

FIGURES

INTRODUCTION

Volume II of the Compiler Writing System design specification is contained herein. It consists of five sections and an appendix. Each section describes a system component, either a program logic module or an external data file.

Section 1 describes the logic modules and data structures composing the Meta-Translator module. This module is responsible for the actual generation of the executable language compiler as a function of the input Meta-Language. Machine definitions are also processed and are placed as encoded data on the Compiler Library Data File.

Section 2 presents the Source Processor design. Logic modules and internal data structures are described as well as the collection of semantic support functions facilitating the generation of Function Language.

Section 3 defines the logic modules and internal data structures of the Function Processor. The algorithms and effects of local and global optimization, and the code generation methodology are presented.

Section 4 deals with defining the optionally invoked Operation Processor Pass II. The transformation of intermediate language in Target Language object text is described.

Preceding page blank

Section 5 describes the format of all external data files utilized by the system. The meaning of the elements contained in each file is described in Volume I.

Figure 1 describes the flow of the Compiler Writing System in terms of a block diagram.

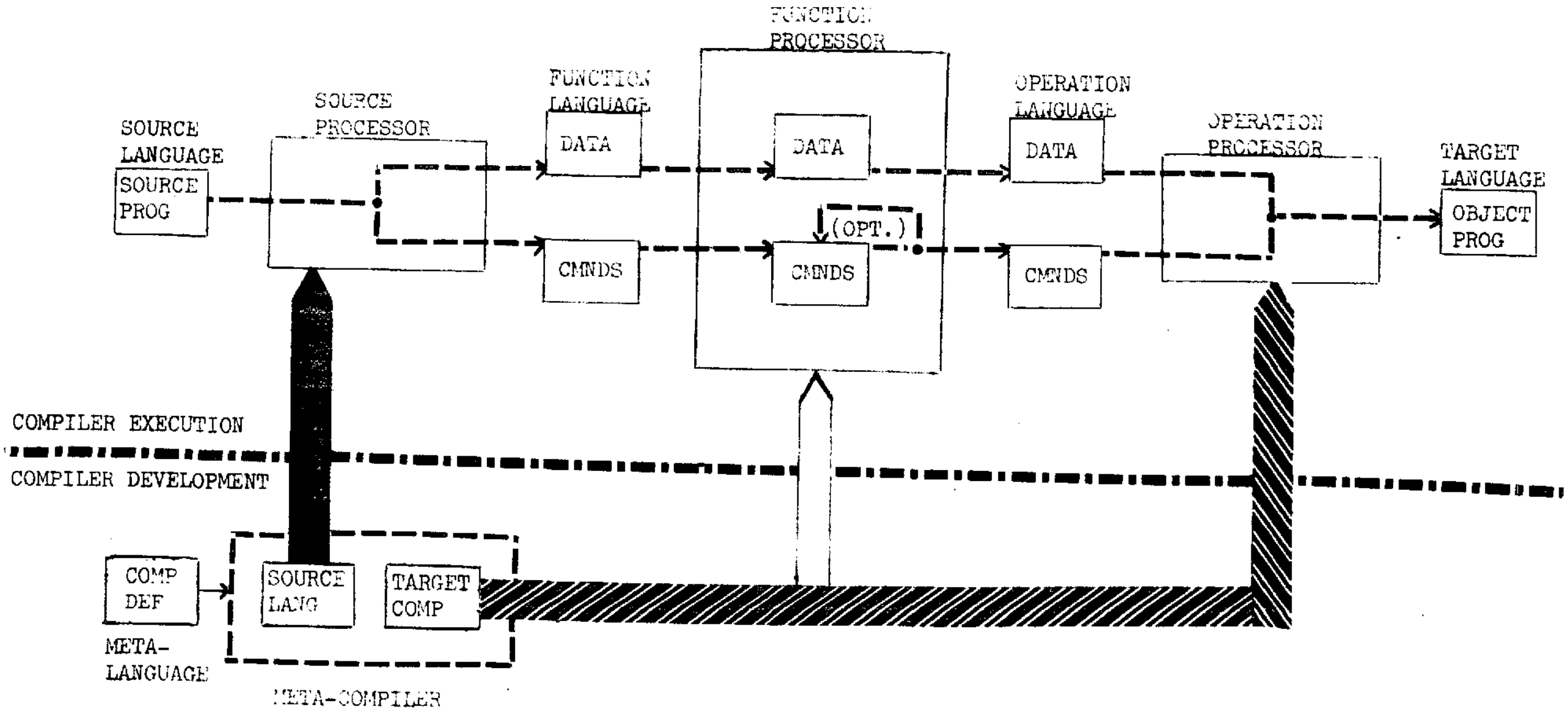


FIGURE 1. COMPILER WRITING SYSTEM BLOCK DIAGRAM

1. META-COMPILER

This section describes the major modules and submodules of the Meta-Compiler as well as the hash tables, stacks, and arrays necessary to process the Meta-Language input. A method for the bootstrap development of the Meta-Compiler using the existing Meta-Translator as a base is also presented.

1.1 Program Logic Modules

The Meta-Compiler is organized as a collection of distinct modules, each having a clearly defined logical function. Only the major modules and sub-modules are presented in this document as further division of tasks between subprograms composing each module is to be accomplished at implementation time. Figure 1.1, the Meta-Compiler block diagram, illustrates the inter-relationships between each module and the internal and external data paths.

The function and processes of each logic module is presented below.

MODULE: Main Control (MAIN)

FUNCTION: Coordinates and controls the execution of all other logic modules of MECOM.

PROCESS:

The main control module initiates execution of MECOM and retains control at all levels. It performs the phased execution of overlay modules, each of which provides processing of a particular section of the Meta-Language.

The Options module is initiated to read the first card image and extract the option parameter settings. A repetitive loop is then entered to process each encountered Meta-Language statement. The relevant statement flags (CURSOR, etc...) are initialized prior to recognizing each statement type. Each statement is recognized as being part of a particular section or level of the Meta-Language, and each distinct section has its own processing module in the form of a separate overlay. A given overlay is repeatedly executed until all statements in the section have been processed, after which the next overlay module is initiated. Subsequent to processing a given statement, control returns to the Main Control module, which then executes the Deferred Execution module for processing any deferred procedure calls that have been stacked, primarily TEXT code generation requests.

Upon encountering the END OF DEFINITION statement, the Wrap-Up module is executed to complete the Meta-Language listing. Control is then returned to the supporting operating system.

01769

META-COMPLIER BLOCK DIAGRAM

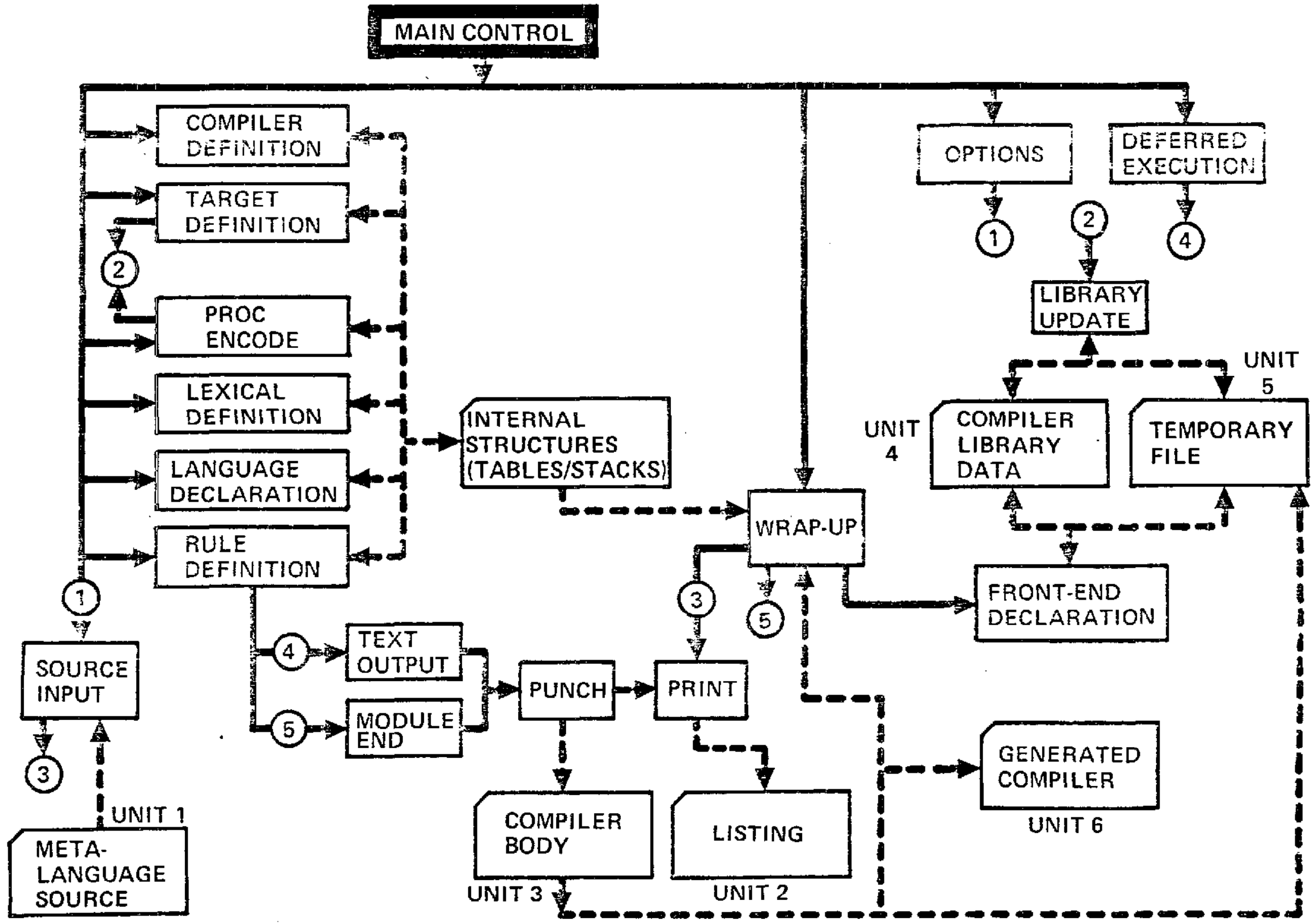


FIGURE 1.1

The processing modules include: Compiler Definition, Target Definition, PROC encode, Lexical Definition, Language Declaration, Rule Definition.

IMPLEMENTATION:

Generated by Meta-Translator bootstrap (1.3), phase 2.

MODULE: Options Parsing

FUNCTION: Parses the MECOM options card and sets the corresponding option flags.

PROCESS:

A single card image of the format specified in section 1.4, Volume I, is parsed with the following interpretation applied to each option:

NO -- No punched output is produced for the generated compiler;

NS -- No listing of the generated compiler statements is produced;

$$\left\{\begin{array}{l}1\\2\\3\\4\\5\\6\end{array}\right.$$
-- the corresponding debug options are activated (see Volume I, DEBUG declaration).

All other 'compiler' options, such as level of optimization, etc..., are ignored during MECOM execution since they have no relevance.

IMPLEMENTATION:

Generated by Meta-Translator bootstrap (1.3), phase 6.

MODULE: Compiler Definition

FUNCTION: Processes the DEFINE statement initiating a compiler definition, and the symbolic EQUATE.

PROCESS:

The compiler name is parsed and placed as the first entry in the NAMLST table. The host word size, card image size, and the table size specifiers (if present) are then parsed and the values are saved. These values will be preset into the correct position of the generated BLOCK DATA initializer.

The EQUATE statement handling involves simply placing each equate name in table TEQU and its equivalent integer value in the parallel attribute. All future references to the name will be replaced by their integer equivalent.

IMPLEMENTATION:

In Meta-Language via bootstrapping (1.3).

<u>MODULE</u>: Lexical Definition

<u>FUNCTION</u>: Processes the lexical preprocessor definition section of a compiler definition.

<u>PROCESS</u>:

This module cycles continuously until all statements defining the lexical preprocessing of the compiler input language have been processed.

The TERMINAL statement handling involves:

(1) Placing each detected terminal name in table TNAME and its corresponding LTERMD start position into the parallel attribute vector;

(2) Encoding each terminal operand into stack LTERMD so as to interpretively drive the compiler Terminal Detection module at compile time. Each operand results in a number of character intervals within LTERMD;

(3) The entries within LTERMD are sorted by interval at the end of processing a terminal definition. Each interval is disjoint.

A literal string terminal operand results in one or more intervals in stack LTERMD, depending on whether the literal characters have consecutive internal values or not. A terminal operand which refers to another terminal name causes all intervals defining the referenced terminal (which must have been previously defined) to be included.

A preceding NOT on a terminal operand causes the intervals for the operand to be 'complemented.' For example:

DIGIT results in the interval	i_o to i_o+9;
NOT DIGIT results in the intervals	1 to i_o-1
	i_o+10 to n

where

i_o -- internal representation of '0';

n -- the total number of meta-characters.

The FIELD statement processing involves placing an entry into stack LFIELD for each encountered field specification.

The CONTINUATION statement processing involves parsing the type of continuation, the beginning and ending append columns (colbegin and colend) and the column operands to selectively generate the FORTRAN function QCONTI, which supports Source Input module in the Source Processor:

```
      FUNCTION QCONTI (DUMMY)
      QCONTI= -1
      CALL QINPUT (IMAGE (CLEN+1))                          Free continuation rule
      CLEN=CLEN+80
      IF (LENGTH .NE. 0) RETURN
      CALL QINPUT (IMAGE)
      CLEN=80                                               No continuation rule
      IF (LENGTH .EQ. 0) GO TO 10                           Card 1 or 2 rule:
      IF (CARD(column).{NE|EQ} .character 1.{OR|AND} .      [normal column operand]
                 :                                          [NOT column operand   ]
    X CARD(column+1-1.{NE|EQ} .character n)RETURN
      [code for next column operand]
      [10 CALL QINPUT (CARD)]                               Card 1 rule
      CALL QMOVE(CARD,IMAGE,colbegin,CLEN+1,colend-colbegin+1)
      CLEN=CLEN + colend-colbegin + 1
      [10 CALL QINPUT(CARD)]                                Card 2 rule
      IF(CLEN.LT.colmax) QCONTI=1
      RETURN
      END
```

The PRESCAN statement processing involves encoding the prescan operands into stack NTABLE to interpretively drive the Source Input module at compile time in the Source Processor. If any direct execution prescan operands are detected, the following subprogram is generated prior to exiting back to the Main Driver module:

```
         FUNCTION  QDIRE(INDEX)
         I=INDEX
         QDIRE=1
         GO TO (800,801,...),I
   800   IF(...)RETURN          [Relational IF]
         GO TO 50
   800   . . .                  [Replacement Statement]
         RETURN
            :
    50   QDIRE= -1
         RETURN
         END
```

Any variable names referenced within direct execution prescan operands are placed in table AUXNAM (variable names). This will cause their later generation as regular FORTRAN variables by the Front-End declaration module.

IMPLEMENTATION:

In Meta-Language via bootstrapping (1.3).

MODULE: Language Declaration

FUNCTION: Processes the language declaration section of a compiler definition.

PROCESS:

This module cycles continuously until all language declaration statements have been processed.

Processing the STORAGE statement merely involves saving the parsed storage direction flag and bias values.

The MODES and OPERATORS statements causes the specified computational modes or allowed operators to be flagged as legal within arrays LMODES and LOPERS. Any operator hierarchy, commutivity, and associativity operand values are saved in array LOPERS.

The ILLEGAL MODES statement processing causes triplet entries to be made into stack IMODES.

The SYMBOL ATTRIBUTES, TABLE, and STACK statements cause the following entries to be made:

a. A table or stack name is placed in table TABLST.
b. For symbol or hash table attribute declarations, the TABLST attribute is set to the next position in stack IATR. The attribute information is then encoded into stack IATR. The number of lost words per entry is computed from the highest numbered word declaration.
c. For stack declarations, only TABLST entries are made.

The IATR information will be subsequently utilized by the Rule Definition module when processing the PUT, IF, PACK, and UNPACK statements. The initial eight entries of IATR and TABLST correspond to the eight segments of the generated compiler symbol table.

The ARRAY and DATA statements cause the generation of FORTRAN DIMENSION or DATA statements with the same information. Each array name is placed in table AUXNAM for later inclusion in an INTEGER and COMMON declaration (see Wrap-Up module). The generated DATA and ARRAY statements are copied onto the Punch file (unit 3) and the temporary file (unit 5), respectively, for later inclusion in the generated BLOCK DATA initializer (for DATA) and the front end declaration (ARRAY).

Processing of the ERROR MESSAGE directive causes each message to be placed in table CLITS, which contains all compressed literal strings.

The DEBUG directive parsing causes appropriate flags (0 or 1) to be saved for each debug option.

The STATEMENT level declaration causes the level for each statement processing mlv to be saved in stack LLEV along with the one-time flag (ONCE) and recognition criteria (mlv or literal pointer). Since an mlv is allowed to occur at more than one level, the various LLEV entries for a given mlv are threaded together through the mlv attribute.

<u>IMPLEMENTATION</u>:

In Meta-Language via bootstrapping (1.3).

MODULE: Source Input

FUNCTION: Reads each Meta-Language statement into the IMAGE processing buffer with external to internal conversion.

PROCESS:

Each card image is read from the standard input device (unit 1) into a working buffer using an '80A1' format. Each character is then converted to an internal number and placed into the consecutive positions in the IMAGE buffer. The original card image is printed on the standard print output device.

IMPLEMENTATION:

Exists in FORTRAN, and is equivalent to the Input module of the Source Processor.

MODULE: Print Output

FUNCTION: Prints a line image from the designated argument buffer with internal to external conversion.

PROCESS:

The argument array is converted from internal character numbers to external 'A1' format characters, which are then sent to the standard print output device (unit 2).

IMPLEMENTATION:

Exists in FORTRAN, and is equivalent to the Print module of the Source Processor.

MODULE: Punch Output

FUNCTION: Punches a card image from the designated argument buffer with internal to external conversion.

PROCESS:

The argument array is converted from internal character numbers to external 'Al' format characters, which are then sent to the standard punch output device (unit 3).

IMPLEMENTATION:

Exists in FORTRAN, and is equivalent to the Punch module of the Source Processor.

MODULE: Target Definition

FUNCTION: Processes the target declaration section of a compiler definition.

PROCESS:

This module cycles until the first PROC definition is encountered, processing all intermediate target declarations. The parsed information is saved in cells, arrays, and tables in a form suitable for encoding onto the Compiler Library Data file by the Library Update module.

The START TARGET processing causes the eight character target name to be saved in two cells. The PSET statement gives values for the code generation parameter settings.

The processing of the REGISTERS statement:

(1) Determines the number of available target registers;

(2) Saves the bit size of each register;

(3) Saves the classification of each register;

(4) Encodes the above information in the stack NREGS and the array RTYPE.

The parsing of the CHARACTERS statement causes the array INT to be set to the target machine internal value corresponding to each language character.

The ARITHMETIC statement is parsed and the floating point and integer parameter settings are saved.

The OPERATION statement parsing is as follows:

(1) An operation mnemonic is placed in table OPTAB;

(2) The next available slot in stack NOPER is placed in the OPTAB parallel attribute;

(3) The OPTAB name pointer, the operation code, and the IFORM name pointers are encoded sequentially into stack NOPER.

The IFORM statement processing causes the instruction format names to be saved in table IFNAME and the next available slot in the IFORM descriptor stack to be placed in the parallel attribute. The IFORM descriptor stack is encoded with the appropriate information.

IMPLEMENTATION:

In Meta-Language via bootstrapping (1.3).

MODULE: PROC Encode

FUNCTION: Encodes the PROC definition section of a target definition into a table to drive interpretive PROC expansion at compile time.

PROCESS:

This module cycles continuously until all PROC definitions have been processed. Each PROC name is placed in table TPROC and the parallel attribute is set to the position in the IPROC stack for the expansion. Each PROC argument is parsed and placed in table PARGS, which is cleared prior to processing each PROC definition.

Each PROC definition is processed by a single left-to-right scan of the definition line. The starting positions in IPROC of each active PROC string are kept in stack QSTACK to enable a pointer thread to be maintained through the string thread word (word 1) of each string. Thus, upon processing the end of a string, the control word of the string (pointed to by the top of stack QSTACK) is set to the next IPROC position unless the end of the PROC is detected, in which case it is set to zero, and QSTACK is popped.

Each element within a string is similarly threaded through a link word, with a pointer to the beginning of each active element saved in stack ACELM.

The various types of PROC elements are encoded into stack IPROC in the format described in section 1.2. One or more operands are parsed and placed in the table in the form of double-entry flags. Each operand which is a symbolic name is checked as follows:

(1) If a trailing '.SYM' modifier is found, the name is marked as a symbol pointer and is placed in table PVARS;

(2) If the name is found in table PARGS, it is marked as a PROC argument; otherwise,

(3) the name is considered a variable, analogous to a 'set' symbol in conventional assemblers, and is placed in table PVARS.

Literal string operands are identified by the position of the string in table CLITS (the compressed literals). Numeric value operands are computed and used directly as operands. Expression operands are treated as follows:

(1) A simple expression of the form 'variable $\pm$ constant' is treated as a single operand;

(2) Complex expressions are encoded into stack PEXP on a strict left-to-right basis. The operand value portion of the expression operand is set to the expression's PEXP position.

Upon detecting the END TARGET statement the PROC Encode module executes the Library Update module to cause the complete target definition to be placed on the Compiler Library Data file.

Prior to exiting back to the Main Driver the following stacks and tables are released to conserve storage:

TPARGS, PVARS, OPTAB, IFNAME, TPROC, NTREGS, NOPER,
IFORM, QSTACK, ACELM, PEXP, IPROC.

IMPLEMENTATION:

In Meta-Language via bootstrapping (1.3).

MODULE: Library Update

FUNCTION: Updates the Compiler Library Data file, creating a new updated file including the current target definition entry.

PROCESS:

Logical units (files) 4 and 5 are rewound and unit 4 (the Compiler Library Data) is copied onto unit 5 until the target definition entry just defined, identified by the saved target ID name, is found. If found the entry is skipped on unit 4. The target data identifying the current target definition is then copied onto unit 5, and the remainder of unit 4 is then copied also. Both units are then rewound and unit 5 is copied in total to unit 4.

The target definition is written as logical binary records in the format indicated for the Compiler Library Data File (see Volume I). The total number of definition words required and the relative starting positions of the IFORM information, PROC expressions, PROC variables, and compressed strings are computed. This information along with the target parameters defined in the declarative section of the target definition becomes the header information. The information contained in the following stacks and tables are then written sequentially following the constant-length header information:

```
Stack  NREGS
Stack  NOPER (the iform pointers are first replaced by the stack
             IFORM-relative start positions)
Stack  IFORM
Table  IPROC attributes (PROC transfer vector)
Stack  IPROC
Stack  PEXP
NPVP  zero cells (space for PROC variables
Table  OPTAB names (operating names)
Table  CLITS literals (compressed literals)
```

MODULE: Rule Definition

FUNCTION: Processes Meta-Language rule definition.

PROCESS:

This module cycles continuously until all language definition rules have been processed. Each rule causes a collection of FORTRAN statements to be generated as part of a module of the Source Processor.

Upon encountering the END OF DEFINITION statement the Wrap-Up module is executed to complete generation of the compiler.

The first rule encountered, the INITIALIZER, causes the following statement to be output:

```
          SUBROUTINE QCINIT
```

This is followed by the code for the initialization rule, generated as described for any rule below.

The statement recognition code for the statement processing implied by the STATEMENTS statement (if present) is then generated by artificially creating the following Meta-Language input via the STRING element:

```
     $compilername.=$QRECO,$QEXECO//QTEMP=PLEVEL,$QSKIP,
     (IF QTEMP GT PLEVEL,PLEVEL=QTEMP,$QEXECO//
     FATAL('STATEMENT SEQUENCE ERROR'))//FATAL('UNRECOGNIZABLE STATEMENT').
     $QSKIP.=PLEVEL=PLEVEL+1,(IF PLEVEL LE n //PLEVEL=1),
     $QRECo//IF PLEVEL LT QTEMP-1,$QSKIP.
     $QRECo.=$LABEL,CASE PLEVEL OF ($QREC1,...,$QRECn).
```

```
$QEXECo.=CASE PLEVEL OF ($QEXEC1,...,$QEXECn).
$QRECi.=SET CASE Q1=(r_i1,...,r_ik).
   :
$QEXEC_i.=CASE Q1 OF ($e_i1,$e_i2,...,$e_ik).
```

compilername	--	the name of the compiler;
n	--	the number of declared statement levels;
rij	--	the mlv name or literal recognizing the j'th statement at level i.
eij	--	the mlv executing the j'th statement at level i.

For each following mlv definition rule, the mlv name is placed in table NAMLST, a comment card with the mlv name is ejected, and the label '5000 + mlvnumber' is prepared for the first statement of the generated body code. Each element of each string of the definition is then processed as described below, with the indicated FORTRAN code generated for each element. The values of the true and false execution path labels for an element are maintained at all times, and are referred to below by 'true' and 'false.' The scheme for maintaining 'true' and 'false' is as follows:

(1) The cells ℓdeft (true path for a definition) and ℓdeff (false path for a definition) are set to 9001 (true exit from an mlv) and 9002 (false exit from an mlv), respectively;

(2) As each string is processed, 'true' and 'false' are set to ℓdeft and ℓdeff. A forward scan is then performed to see if an alternative string ('//') for the current string is present, and if so 'false' is reset to a unique label (2000+K) which is saved on stack PSTACK. At the end of the string processing, the or ('//') causes this label to be unstacked and the code:

```
2000+k   CALL   QORP
```

to be generated;

(3) When all strings of a definition are so processed, the code 'GO TO ℓdeft' is generated;

(4) Code is generated for each element of a string as defined below. A reoccurrence element is considered as containing a sub-definition within its parens; thus, ℓdeft and ℓdeff are saved on PSTACK and are reset to two unique end label exits (8000+K and 8001+K) identifying the code at the end of the reoccurrence. When the reoccurrence has been processed the old values of ℓdeft and ℓdeff are restored.

(5) The code for most elements (see below) is such that if an element is false a branch is made, so that a true element flows into its successor element directly. Thus, the value of 'true' is rarely used as branch label. However, three elements (mlv reference, literal reference, and the end of a reoccurrence) may generate a 'true' branch. In those cases the element is checked to see if it is followed by a comma (i.e., has a successor), and if so a unique label (7000+K) is used in place of 'true', the code is generated, and the unique label is set up as the label for the first statement of the next element.

Code Generation by Element

(1) MLV Stepdown Element

Code: QSAVE = knnn

GO TO 9003

6nnn IF (TRUTH) $\left\{\begin{array}{l}\text{false, mlvnum,true}\\ \text{true, mlvnum,false}\end{array}\right\}$ [NOT prefix]

True.... [Generated if this element is followed by a comma]

n -- This is the n'th stepdown in this module;

k -- The mlv number of the stepped-into mlv;

mlvnumm -- 50XX, where XX is the mlv number of the stepped into mlv if the mlv is in this module; or

-- 95XX, where XX is the label of a call statement in this SUBROUTINE to the module containing the called mlv.

(2) Literal Element

Code: IF (QCHAR.NE.charcode) GO TO $\left\{\begin{array}{l}\text{false}\\ \text{true}\end{array}\right\}$ [If NOT prefix]

IF (QLITSC(litpos, 1).RO.0) GO TO false [for multi-character literals]

True.... [Generated if this element is followed by a comma]

CURSOR=CURSOR+1
QCHAR=IMAGE(CURSOR) [For single-character literals and no NOT prefix]

charcode - internal value for the first literal character;

litpos -- literal position within QLVECT.

label -- $\left\{\begin{array}{l}\text{false}\\ \text{true}\end{array}\right\}$ [if NOT prefix]

RO -- $\left\{\begin{array}{l}\text{NE}\\ \text{EQ}\end{array}\right\}$ [if NOT prefix]

(3) Scan Element

Code: IF(QLITSC(litpos,3).LE.0) GO TO false

litpos -- literal position within QLVECT.

(4) IF Test Element

(a) IF literal

Code: IF(QCHAR.NE.charcode)GO TO false

IF(QLITSC(litpos,4).LE.0)GO TO false [multi-character literals]

charcode, litpos--see element (2).

IF terminal

code: IF(QTERM(tpos,1,1,1).LT.0)GO TO false

TPOS -- terminal position within LTERMD.

Note: The form 'NOT IF literal' = 'NOT literal',element type (2).

(b) IF iexp relop iexp

code: IF(iexp.relopc.iexp)GO TO false

relopc -- relational complement of relop, or is relop if NOT prefix.

(c) IF mlv IN hashname

Code: CALL QSTACK(tabnum, mlvnum, 2, ptr, ℓ)

IF (ptr .$\left\{\begin{matrix}\text{LE}\\ \text{GT}\end{matrix}\right\}$. 0) GO TO false [NOT prefix]

(d) IF mlv $\left\{\begin{matrix}\text{EQ}\\ \text{NE}\end{matrix}\right\}$ mlv

code: IF(QSTART(mlv1). $\left\{\begin{matrix}\text{NE}\\ \text{EQ}\end{matrix}\right\}$. QSTART(mlv2))GO TO false

IF(QSIZE(mlv1). $\left\{\begin{matrix}\text{NE}\\ \text{EQ}\end{matrix}\right\}$. QSTART(mlv2))GO TO false

Note: EQ $\rightarrow$ NE and NE $\rightarrow$ EQ if NOT prefix.

(e) IF iexp ON stackname

```
code:      IF(QSTKSR(iexp,stacknum). {LT} .0)GO TO false
                                     {GE}  [if NOT prefix]
```

(f) IF mlv TYPE iexp

```
code:      CALL QSTACK9iexp,mlvnum,2,SYMP,SYML)
           IF(SYMP.{LE}.0) GO TO false
                   {GT}  [if NOT prefix]
           mlvnum--the mlv number.
```

(g) IF NEW SYMBOL

```
code:      IF(QSTKCK. {EQ} .0)GO TO false
                      {NE}  [if NOT prefix]
```

(5) Reoccurrence Element

```
Code:           CALL QREOCB (min,max)
   8XXX         [Code for the
                 body of the
                 reoccurrence]

   8YYY         QLNAP= -1
   8nnn         IF(QREOCE(Q9999))false,8XXX,true
```

min --	minimum repeat value
max --	maximum repeat value
8yyy --	true exit point for a true iteration of the reoccurrence
8nnn --	false exit point for a false iteration of the n'th reoccurrence.

Note: Y=X+1, n=X+2.

(6) Replacement Element

Code: dvar = iexp [i.e., direct copy]

(7) CASE iexp OF (element [,element]$_n$)

Code: Q9999=iexp [if iexp is not a simple variable]

```
        GO TO ($3\ell_1\ell_1\ell_1$,$3\ell_2\ell_2\ell_2$,...),{Q999}
                                         {iexp}
$3\ell_1\ell_1\ell_1$ [code for element 1]
        GO TO true
$3\ell_2\ell_2\ell_2$ [code for element 2]
        GO TO true
          .
          .
          .
```

(8) Procedure Reference

[code for saving each argument value in QPARGS, whenever necessary]

```
{CALL QARG10 (chain,priority,argptr)} [deferred call]
{CALL QARG10 (  0,      0,    argptr)} [immediate call]
```

chain -- deferred call chain pointer

priority - deferred call priority

argptr -- QPARGS position pointer for the arguments

The following arguments require direct code replacement into the value cell[s] following the argument type cell in QPARGS. All arguments whose value can be determined at meta-compile time by MECOM are placed in QPARGS via the block data mechanism:

(a) iexp [not a constant]

```
QPARGS(argpos)=iexp
```

(b) hashname(iexp)

```
QPARGS(argpos+1)=iexp
```

(c) stackname(iexp)

```
QPARGS(argpos+1)=iexp
```

(d) ELEMENTS [iexp THRU iexp] OF stackname

```
QPARGS(argpos+1)=iexp1  [if not constant]
QPARGS(argpos+2)=iexp2  [if not constant]
```

(9) Put Element

Code: CALL QSTACK(tabnum,symbol,1, $\begin{Bmatrix}\text{SYMP,SYML}\\ \text{ptr,L}\end{Bmatrix}$)

tabnum --	hashname table number or symbol type;
symbol --	>0 = mlv number
	<0 = negative of compressed string pointer for literal;
ptr,L --	running pointer and length designators for the hash table.

If 'PUT literal[,literal]$_n$...', the above code is generated for each literal operand.

(10) Push Element

Code: Q9999=QSTKM(1,stacknum,iexp_1,iexp_2)

stacknum --	stack number
iexp_1 --	expression operand to push onto stack.
iexp_2 --	specified entry, or -1 if not present.

For 'PUSH mlv ON...', the following is generated:

```
Q9999=QSTKM(1,stacknum,QSTART(mlvnum),iexp2)
Q9999=QSTKM(1,stacknum,QSIZE(mlvnum),iexp2)
```

mlvnum -- the mlv number

(11) Pop Element

Code: Q9999=QSTKM(2,stacknum,0,0)

stacknum -- stack number.

The above code is generated for each stack operand in the pop element.

(12) Set Element

(a) SET mlv_1=mlv_2

Code: QSTART($mlvnum_1$)=QSTART($mlvnum_2$)

QSIZE($mlvnum_1$)=QSIZE($mlvnum_2$)

(b) SET mlv=STRING (arg$[,arg]_n$)

QSTART(mlvnum)=LENGTH

QSIZE(mlvnum)=IFPOS

$\left[\begin{array}{l}\text{Generate an immediate procedure} \\ \text{call to QSTRM with the} \\ \text{given arguments}\end{array}\right]$

(c) SET CASE svar=(element$[,element]_n$)

Code: svar=0

[element 1 code]

svar=1

GO TO true

[element 2 code]

svar=2

GO TO true

⋮

(13) Terminal Element

Code: IF(QTERM(tpos,min,max,$\left\{\begin{matrix}\text{0.LT.}\\ \text{1.GE.}\end{matrix}\right\}$0) GO TO false
[If NOT prefix]

tpos --	terminal LTERMD position
min --	minimum number to find
max --	maximum to look for

(14) Null Element

Code: None generated

(15) False Element

Code: GO TO false

(16) Chain Element

Code: CALL QCHSTK

svar=CHAIN [if the svar operand is present]

(17) Fortran Element

Code:

col 1	col 7
iexp	literal

(18) Test Element

Code: IF(QLITSC(litpos,2).LT.0) GO TO 9002

(19) Field Element

Code: IF(QFIELD($\left\{\begin{matrix}\text{0}\\ \text{iexp}\end{matrix}\right\}$)) GO TO false

iexp -- the iexp field designator, if present

(20) Attribute Element

Code: CALL QATTR(func, $\left\{\begin{matrix}\text{tabnum}\\ 0\end{matrix}\right\}$, $\left\{\begin{matrix}\text{iexp}\\ \text{ptr}\\ \text{symp}\end{matrix}\right\}$, $\left\{\begin{matrix}0\\ \text{attribnum}\end{matrix}\right\}$

func -- 0=pack, 1=unpack function

tabnum -- hash table number or symbol type, or zero for '[UN]PACK svar' element, form (c).

iexp -- entry index operand.

ptr -- hash table running pointer name if no iexp operand is present for element types (a) and (b).

attribnum -- the attribute number for the attribute name operand (svar), if present.

(21) Error Element

(a) Warning Error

```
Code: [Regular argument code]
      [for procedure args   ]
      CALL QARG10 (0,0,argptr)
```

argptr -- QPARGS position pointer for the arguments

(b) Regular Error

```
QERRSW=1
[Regular argument code]
[for procedure args   ]
CALL QERR(argptr)
GO TO false
```

(c) Fatal Error

```
QERRSW=2
[Regular argument code]
[for procedure args   ]
CALL QERR (argptr)
QCTI=0
GO TO 9002
```

argptr -- QPARGS argument pointer.

(22) Table Initializers

(a) Write Function

Code: CALL QTABD

(b) Read Function

Code: IF(QTABR(0).LT.0)GO TO false

(23) Debug Element

Code: IDEBUG(opnum)= $\begin{Bmatrix} 1 \\ 0 \end{Bmatrix}$ $\begin{matrix} \text{[ON]} \\ \text{[OFF]} \end{matrix}$

The above is generated for each detected option setting.

opnum -- an option number (1-6).

(24) Free Table Element

Code: CALL QFREE(tablenum)

⋮

tablenum -- a hash table number or symbol type.

The above is generated for each operand.

(25) Save/Restore Element

Code: SAVE $iexp_1$ $iexp_2$ ....$iexp_n$

[Save]Q9999=QSTKM(1,1,$iexp_1$,-1)

Q9999=QSTKM(1,1,$iexp_n$,-1)

Code: RESTORE $svar_1$ $svar_2$ ...$svar_n$

svar1=QSTKM (2,1,0,0)

⋮

svarn=QSTKM (2,1,0,0)

The following is a summary of the label usage in the generated code:

2000+N -- Label for the alternative string to a false string.

8000+N -- Reoccurrence control labels.

3000+N -- CASE OF alternative labels.

6000+N -- N'th stepdown call in a module.

5000+N -- Start of mlv N.

7000+N -- General work label.

Whenever an mlv definition rule is detected for an mlv belonging to a different module from the current one (i.e., resides at a different statement level), the Module End routine is called to generate the module epilog code. The following code is then generated:

```
SUBROUTINE QLEVi
QSYN=QCTI
Q9999=QMLV-j
GO TO (5001+j,5002+j,...,5000+m+j),Q9999
```

i -- the processing level of the new mlv (see STATEMENTS statement);

j -- the number of mlvs declared in the previous module

m -- the number of mlvs declared in this module.

MODULE: Module End

FUNCTION: Generates code to end the active FORTRAN subprogram module being generated.

PROCESS:

The following standard code is generated:

```
     9001    TRUTH=1
             IF(QCTI.LE.QSYN) RETURN
             GO TO 9004
     9003    TRUTH=0
     9004    CALL QSTEP
             GO TO (6001,6002,...,6mmm),Q9999
     9501    CALL QLEVi
             GO TO 1000
       .
       .
     950n    CALL QLEVn
     1000    IF (TRUTH) 9002,9003,9001
             END
```

m -- the number of step-downs in this module;

i,...,n -- the statement levels of mlvs called from this module which reside in other modules.

IMPLEMENTATION:

In Meta-Language via bootstrapping (1.3).

MODULE: Wrap-Up

FUNCTION: Complete the generation of the defined compiler.

PROCESS:

The Module End routine is executed to generate the epilog code ending the last generated module.

The following subroutine PEXEC is then generated, becoming the Deferred Execution module of the Source Processor:

```
          SUBROUTINE PEXEC
          IF(QERRSW.LE.0.AND.TRUTH.EQ.-1)RETURN
          IF(QERRSW.GT.0)GO TO 9999
     1    IF(QUNSTK(Q9999).LE.0)RETURN
          GO TO (100,101,102,...),Q9999
   100    CALL procedurename 1
          GO TO 1
   101    CALL procedurename 2
          GO TO 1
           .
           .
           .
  9999    CALL QERTXT
          RETURN
          END
```

procedurename--a semantic procedure name.

The procedurenames are those saved in table PRCLST.

The Front-End Declaration module is called to produce a group of declaration statements on unit 5. A BLOCK DATA subprogram is then formed on the Generated Compiler file (unit 6) by first writing the 'BLOCK DATA' statement and then copying unit 5 onto unit 6. Then, the relevant arrays and control cells of the Source Processor are initialized to the saved information derived from Meta-Language declarations and rules as follows:

```
        DATA QLVECT/[info in table LITLST and its attributes]/
        DATA QPARGS/[stack QPARG data]/
        DATA QTERMD/[stack LTERMD data]/
        DATA QPRES/[stack NTABLE data]/
        DATA QIMODC/[stack IMODES data]/
        DATA QFIELD/[stack LFIELD data]/
       {DATA QTABP/[compressed literal string data (table CLITS);/
       {DATA QTABS/ error message data (CLITS); declared table
       {DATA QTABLE/layouts and sizes (TABLST)]
        DATA QDEBUG/[the saved DEBUG declaration flags]/
        DATA QLMODE/[array LMODES]/
        DATA QLOPER/[array LOPERS data]/
        DATA QALPH/1H,,1H.,...[using array INT to output each valid
                     language character; missing characters go out
                     as a zero constant]/
        DATA QWDSZ,QOUTU,QINU,QPCHU,QHWD,QLIBU,QBIAS,QDIR
          /decsize,2,1,3,4,qhwd,5,qbias,qdir/
          decsize--declared host word size
          qhwd--2**((QWDSZ+1)/2)
          qbias,qdir--the bias and storage direction derived from the
                    STORAGE directive.
        END
```

The subroutine QATR is generated as a function of the hash table and symbol table declarative information in stack IATR for the purpose of packing and unpacking table structures. It becomes the Attribute Maintenance Utility submodule of the Source Processor:

```
          SUBROUTINE QATTR (IFUNC,ITAB,IPTR,ATTNUM)
          IP=IPTR
          IF (ITAB.NE.0)GO TO 1
          ITAB=IPTR/qhwd
          IP=IPTR-ITAB*qhwd
  1       J=QTABP(1,ITAB)+QTABP(3,ITAB)+(IP-1)*QTABP(5,ITAB)
          IF (IFUNC.GT.0)GO TO 5000
  C   PACK FUNCTION
          GO TO (100,200,300,....),ITAB
  C   PACK TABLE 1
 100      IF (ATTNUM.NE.0)GO TO(101,102,....),ATTNUM
  C   PACK ATTRIBUTE 1
 101      QTABS(J+wrdnum)=QOR(QAND(QTABS(J+wrdnum),imsk),QLSFT(QAND(
            attrname,2**len-1),qwdsz-bitbeg-ℓen))
            [normal case]
 102      QTABS(J+wrdnum)=attrname    [if fills whole word]
          IF(ATTNUM.NE.0)RETURN
           :
 200      IF(ATTNUM.NE.0)GO TO(201,202,...),ATTNUM
           :
          RETURN
  C   UNPACK FUNCTION
5000      CONTINUE
          GO TO (5100,5200,5300,...),ITAB
  C   UNPACK TABLE 1
5100      IF(ATTNUM.NE.0)GO TO(5101,5102,...),ATTNUM
  C   UNPACK ATTRIBUTE 1
5101      attrname=QAND(QRSFT(QTABS(J+wrdnum),qwdzs-bitbeg-len),2**len-1)
          [IF(attrname.GE.2**(len-1))attrname=QOR(attrname,isign)][signed value]
5102      attrname=QTABS(J+wrdnum)  [if fills whole word]
          IF(ATTNUM.NE.0)RETURN
           :
```

```
5200    IF(ATTNUM.NE.0)GO TO (5201,5202,...),ATTNUM
          :
        RETURN
        END
```

wrdnum--the declared host word position of an attribute.

imsk--a mask with zeros in the attribute bits, ones elsewhere.

len--bit size of an attribute.

bitbeg--bit start position of an attribute.

attrname--the attribute name.

isign--a mask having ones except for the rightmost len bits.

The final action performed is that of rewinding the temporary file (5) and punch image file (3) in preparation for a scan of the generated compiler to insert the declaration header generated by the Front-End Declaration module on unit 5 onto the front of each generated module on unit 3. The result is a completed compiler on unit 6, which is then rewound and printed and/or punched depending on options selected by the compiler writer (see Options module).

IMPLEMENTATION:

In Meta-Language.

MODULE: Front-End Declaration

FUNCTION: Produce a declarative section to be inserted in the front of each generated subprogram.

PROCESS:

The Meta-Language declarative information is used to generate a sequence of COMMON, DIMENSION, INTEGER, and EQUIVALENCE statements to be later inserted into all generated modules by the Wrap-Up module. The declarative data is stored in card image format on the temporary file (unit 5).

The declarations are produced in the following order:

```
     INTEGER TRUTH,CURSOR,...          [standard compiler cells]
     INTEGER QSTART,QSIZE,QLVECT,...   [standard compiler arrays]
     INTEGER QBCHBI,QREOCE,...         [standard compiler support functions]
     INTEGER A,X,...                   [mlv cells-from table AUXNAM]
     INTEGER FUNC,...                  [deferred semantic functions-table PRCLST]
     COMMON/QLVECT/QLVECT(size1)/QSTART/QSTART(mlvs)/QSIZE/QSIZE(mlvs)
     /QTERMD/QTERMD(size2)/QPARGS/QPARGS(size3)/QPRES/QPRES(size4)/
     QIMODC/QIMODC(size4)/QFIELD/QFIELD(size5)/QTABP/QTABP(5,numtab)/
     QTABS/QTABS(decsize)/QCTAB/QCTAB(decsize)/QTABLE/QTABLE(256+decsize)/
     IMAGE/IMAGE(decsize)/QFIXED/QDEBUG(8),QLMODE(13),QLOPER(3,23)
     EQUIVALENCE(entryname,QTABP(2,tabnum)),...   [for each table/stack]
     COMMON/QCELLS/TRUTH,CURSOR,...    [standard compiler cells]
     COMMON/QCELLS/A,X,SYMP,...   [user cells-all entries in table AUXNAM,
                                   including variables, declared arrays,
                                   attribute names]
     COMMON/QTARG/QTARG(desize)    [target definition area]
     EQUIVALENCE(QNTWRD,QTARG(1)),(QNREGS,QTARG(2)),...
                         [fixed length target header-control variables]
```

MODULE: Deferred Execution

FUNCTION: To execute any deferred procedure calls controlling the execution of MECOM.

PROCESS:

This module is executed by the Main Control routine in order to execute any stacked procedured calls saved through the true parsing of the current Meta-Language statement within MECOM. In addition, any warning errors, normal errors, or fatal errors are listed at this time. The Textual Output module is called to handle a stacked TEXT request (see Volume I, TEXT element).

IMPLEMENTATION:

Automatically generated as a result of MECOM being written in its own Meta-Language.

MODULE: Textual Output

FUNCTION: Produces the actual card images forming the generated compiler.

PROCESS:

This module handles the execution of any TEXT output request made by the execution of any modules within MECOM. The textual information is expanded and copied into the output buffer and control is passed to the Punch module to generate the punch image(s).

This module is never executed if the punch generation option to MECOM is de-selected.

1.2 Internal Table Structures

Internal to the MECOM processor are several sets of table, stack, and array structures containing symbolic and control information driving the compiler generation process. With the exception of those structures having predefined initialization values, the structures are prepared dynamically during the Meta-Compilation process. Additional internal cells and flags exist within METRAN but are not discussed in this document due to their readily apparent use, such as control values, title and line controls, etc...

The information encoded within a word of a table, stack, or array is shown in symbolic format, the exact bit positions of each item to be determined at implementation time.

1.2.1 Hash Tables

The following tables are defined within MECOM for the purpose of saving symbolic names and strings. Certain tables have a parallel set of attribute values which may be referenced using the running table pointer.

Those tables having predefined symbolic entries and attributes are initialized to their designated values at the time MECOM is initiated. This is accomplished using the READ TABLES Meta-Language element, implemented through the MECOM bootstrap algorithm (see section 1.3). The table entries will be constructed using PUT and PACK elements followed by a WRITE TABLES element as a separate run.

Terminal Names - TNAME

Pointer: TNP

Attributes: ℓtermpos

The table of terminal names. The attribute ℓtermpos gives the starting word position (in the generated LTERMD terminal driver array) for each terminal.

Literal Language Strings - LITLST

Pointer: LI

Attributes: ℓitpos

The table of literal (quoted) strings detected during the Meta-Language parse. The ℓitpos attribute gives the position of the literal string (one character (internal) per word with a leading character count word) in the generated array QLVECT.

MLV Names - NAMLST

Pointer: NI

Attributes: level thread
reference thread

level thread -- LLEV stack pointer defining the mlv processing level;

reference thread -- mlv reference thread within SREFS stack.

The table of unique mlv names.

Equate Names - TEQU

Pointer: TEQUP

Attributes: intval

The table of equate names. The intval attribute gives the integer value to which each equate name is equivalent.

Table/Stack Names - TABLST

The table of declared stack and table names.

Pointer: ITABS

Attributes:	entry ptr name
	entry len name
	length
	attribptr

entryptrname -- declared table/stack position pointer name in table AUXNAM;

entrylenname -- declared table entry length AUXNAM pointer for tables, or,
0 for stacks;

length -- declared table or stack length;

attribptr -- IATR table pointer for table attribute names, or,
zero for stacks.

Initialization:

The first eight table names which correspond to the compiler symbol table, and their associated attributes as defined in Appendix A, Volume I, are initialized to the indicated names and bit layouts. The two automatically generated compiler stack names, PSTACK and QSTACK are also predefined with the running pointer names PPOINT and QPOINT.

<u>Auxiliary Names - AUXNAM</u>

Table of stack and list control cells as well as any detected variable names.

Pointer: IAUX

Attributes: None

Initialization:

All system-defined variables, including control cells, symbol attributes, etc... as described in the Source Processor design, are predefined in table AUXNAM.

<u>Procedure Names - PRCLST</u>

Table of deferred or immediate procedure reference names.

Pointer: NPRO

Attributes: None

<u>Compressed Strings - CLITS</u>

Table of compressed literal strings.

Pointer: ICLP

Attributes: None

All literals detected within a procedure reference or error request are saved in this table in normal packed symbol format. The CLITS table within the generated compiler will be initialized to the contents of this table.

PROC Argument Names - TPARGS

Pointer: NPARG

Attributes: None

The table of argument names for the current PROC are being analyzed.

PROC Variable Names - PVARS

Pointer: NPVP

Attributes: None

The table of variable names referenced within all PROCs as operands.

Operation Names - OPTAB

Pointer: OPN

Attributes: startpos

startpos--starting position in stack NOPER of the operation descriptive information.

The table of target machine operation names declared in the OPERATIONS statements.

Instruction Format Names - IFNAME

Pointer: IFN

Attributes: iformpos

iformpos--starting position in stack IFORM of the instruction format descriptive information.

The table of instruction format names declared in the IFORM statements.

PROC Names - TPROC

Pointer: NPRC

Attributes: procpos

procpos--starting position of the PROC in the PROC skeleton stack IPROC. If negative, the PROC name is equivalent to operation '-procpos'.

The table of PROC expansion names.

Initialization:

All optional and required Operation Language term names (Volume I) are initially placed within TPROC, since each may correspond to a PROC expansion. The attributes are all set to zero (i.e., no expansion provided yet).

1.2.2 Stacks

The following stacks are defined within MECOM as dynamic last-in-first-out program stacks holding single-word control entries. Most MECOM stacks have an identical structure to an array within the generated Source Processor, and are used to initialize the array by the Wrap-Up module via the generated BLOCK DATA subprogram.

Terminal Driver Stack - LTERMD

Pointer: LTP

Format:

$$\left.\begin{array}{cc} b_{11} & e_{11} \\ \vdots & \vdots \\ b_{k1} & e_{k1} \\ 0 & 0 \end{array}\right\} \text{Terminal 1}$$

$$\begin{array}{cc} \vdots & \vdots \end{array}$$

$$\left.\begin{array}{cc} b_{1n} & e_{1n} \\ \vdots & \vdots \\ b_{in} & e_{in} \\ 0 & 0 \end{array}\right\} \text{Terminal n}$$

b, e -- internal language character codes

Each b_{jm}, e_{jm} pair represents an interval of language characters, from b_{jm} to e_{jm}, any one of which satisfies terminal m. A terminal will be detected at compile-time if the current language character is found to be within one of the terminal's intervals. This stack is used to initialize the compile-time stack LTERMD in the source processor.

Prescan Driver Stack - NTABLE

Pointer: NPNT

This stack is identical in structure to the NTABLE array driving the PRESCAN activities of the lexical preprocessing function in the Source Processor. The stack is used to initialize the array in the generated block data subprogram by the Wrap-Up module.

```
Format:   string control word 1 )
          operand 1 (3 words)   } PRESCAN
          operand 2 (3 words)   ) string
             :
          string control word 2               PRESCAN 1
             :
          string control word i
             :
          string control word 1          )
             :                           } PRESCAN 2
          string control word k          )
          ___________________________    :
          ___________________________    :
          0                              stack end
```

string control word:

+K -- pointer to next string in this PRESCAN;

-K -- pointer to first string of the next PRESCAN;

 0 -- this is the last string of the last PRESCAN.

string operand:

word 1: ifflag, not flag, c

ifflag -- on if the operand has an IF prefix;

notflag -- on if the operand has a NOT prefix;

c -- operand control type.

	Word 2	Word 3
C=1, Syntax Operand	-K -- ℓitptr +K -- max, terminalpos	columnnum
C=2, Value Request	0	0
C=3, Direct Execution	index	0
C=4, SET NEXT FIELD Request	0	0
C=5, COPY Request	-K -- ℓitptr +K -- max, terminalpos	count
C=6, SKIP Request	-K -- ℓitptr +K -- max, terminalpos	count

ℓitptr -- literal string position in generated QLVECT array;

max -- maximum number of terminal occurrences (1 if single);

terminalpos -- terminal starting position in LTERMD terminal driver stack;

columnnum -- 0 -- use current prescan CURSOR value,
             K -- use column K of current card image;

index -- the index of the directly executable relational IF or
         replacement code;

count -- -1 -- COPY or SKIP THRU request,
          0 -- no count value,
          1 -- word 2 is the actual number of characters to
               SKIP or COPY,
          2 -- SKIP or COPY the computed contents of VALUE.

Illegal Modes Stack - IMODES

Pointer: IMODE

This stack is identical in structure to the IMODES array in the Source Processor controlling expression mixed mode validity checking. This stack is used directly to initialize the array in the generated block data subprogram by the Wrap-Up module.

Format:

Composed of triple-word entries followed by a zero word.

Word 1:	0 -- end of data
	m1ℓ, m1h
Word 2:	O1ℓ, O1h
Word 3:	m2ℓ, m2h

m1ℓ	-- lower bound for mode one;
m1h	-- upper bound for mode one;
O1ℓ	-- lower bound for operator value;
O1h	-- upper bound for operator value;
m2ℓ	-- lower bound for mode two;
m2h	-- upper bound for mode two.

Symbol Attribute Stack - IATR

Pointer: IATRP

This stack contains symbol or hash table attribute information defining the cell name and host field position for each declared attribute value.

Format:

```
        nameptr,signflg                declared
        wordpos,bitpos,bit length      attribute
                                                       Table 1
        nameptr,signflg                                attributes
        wordpos,bitpos,bit length
            .
            .
        0   .
            .
            .
            .
            .
```

nameptr--symbol attribute name pointer (in table AUXNAM);

signflg--on if a signed attribute, off if not;

wordpos--host word position;

bitpos--host bit start position within wordpos;

bitlength--number of bits to represent the attribute value.

Initialization:

The first eight IATR entries, corresponding to the default attributes and bit layouts of the compiler symbol table, are initialized to the values indicated in Volume I, Appendix A.

Label Control Stack - PSTACK

Pointer: PPOINT

Format:

Composed of two word entries:

Word 1: ℓdeft

Word 2: ℓdeff

ℓdeft -- the value of the true exit label out of the current definition;

ℓdeff -- the value of the false exit label out of the current definition.

Processing Level Stack - LLEV

Pointer: LPLEV

Format:

Composed of double-word entries:

Word 1: levelnum, mlv thread

Word 2: onceflag, recogflag

levelnum -- a processing level for the mlv as declared in the STATEMENTS declaration;

mlv thread -- 0 -- no further levels for the mlv,
+N -- LLEV position for the next double-word entry;

onceflag -- indicates the statement processed by the mlv may appear but once;

recogflag -- +N -- literal position (in QLVECT) for a literal string acting as a statement recognition criteria;
-N -- statement recognized by mlv number N;
0 -- no criteria; simple include the mlv at level levelnum.

Each mlv name has an attribute pointing to the first LLEV entry for that mlv.

Mlv Reference Stack - SREFS

Pointer: MREF

Format:

Composed of single-word entries:

mlvnum, thread

mlvnum -- the number of an mlv referencing the mlv in question;

thread -- SREFS pointer to the next reference word.

Each mlv name has an attribute pointing to an MREF entry defining a series of references to the mlv.

Procedure Argument Stack - QPARG

Pointer: QPNA

This stack is identical to the QPARGS array in the Source Processor, which contains argument control words and values for all procedure references. The stack is used to initialize the array in the generated block data program by the wrap-up module.

Format:

```
argcount 1
  argtype 11 | Procedure call
  value 11   | argument             Procedure call 1
     :
  argtype 1n
  value 1n

argcount 2
  argtype 21                        Procedure call 2
     :
  argtype 2K
     :
```

argcount -- number of argument control and value words following, collectively forming one procedure call;

argtype -- -1000+K -- this is a procedure call to semantic routine number K (K'th PRCLST table name). This word will be followed by the first real argtype word.

-K -- argument type as defined below:

argtype	Value Word[s]	Meaning
-1	[None]	PUNCH operand (TEXT request)
-2	[None]	PRINT operand (TEXT request)
-3	columnnum (0 if variable)	COLUMN operand (TEXT request) columnnum -- column value
-4	[None]	EJECT operand (TEXT request)
-5	label (0 if variable)	EJECT label operand (TEXT request) label -- label value
-6	start (0 if variable) end (0 if variable) stacknum	ELEMENTS OF operand start -- start position end -- end position stacknum -- stack number
-7	0	Hash/Symbol Table operand
-9	intval (0 if variable)	Single-valued expression operand intval -- integer value
-10	litptr	Literal string operand litptr -- literal position in QLVECT
-11	mlvnum	MLV operand mlvnum -- the mlv number

Field Specification Stack - LFIELD

Pointer: LFP

This stack is identical in structure to the LFIELD array in the source processor controlling cursor movement within fields. The stack is used directly to initialize the array in the generated block data subprogram by the Wrap-Up module.

Format:

```
    Composed of double-word entries:
         word i:    startcol
         word iH:   endcol
    startcol--starting column for field i (0 if unknown);
    endcol--ending column for field i (0 if unknown).
```

Register Identification Stack - NREGS

Pointer: NTREG

Format:

```
    word i:  size
    size--the declared size for register i.
```

Operation Pointer Stack - NOPER

Pointer: NOPSIZ

```
Format:     nameptr
            K, opcode
            iform1, iform2,           An OPERATION definition
               :
            iformk-1, iformk
               :
```

nameptr--operation name pointer (table OPTAB);

K--number of optional instruction formats;

opcode--op code value;

iformi--instruction format name pointers (table IFNAME).

Each entry of this stack represents a declared OPERATION having several optional instruction format types. Prior to encoding this table into a target definition, the Library Update module replaces the iform pointers with the corresponding table IFORM starting positions.

<u>Instruction Format Descriptor Stack - IFORM</u>

Pointer: IFSIZ

```
Format:  bitlen            \
         type,start,end     |
           :                 >   An IFORM specification
         type,start,end     |
           :               /
```

bitlen--instruction bit length;

type--operand type;

start--operand start position in generated instruction;

end--operand end position in general instruction.

<u>Active Strings Stack - QSTACK</u>

Pointer: QPOINT

Format:

word i: startpos

startpos--active PROC string starting position within IPROC stack.

The i'th entry of this stack points to the start position of the active string at level i of the current PROC.

Active Elements Stack - ACELM

Pointer: NACE

Format:

```
        word i:  startpos
```

startpos--active PROC element starting position within IPROC stack.

The i'th entry of this stack points to the start position of the active element at level i of the current PROC.

PROC Expression Stack - PEXP

Pointer: NEXP

```
Format:     operand 1 (2 cells) }
                                     A PROC
            operator 1               Expression
            operand 2 (2 cells)
               .
               .
               0
               .
               .
               .
```

operand--PROC operand of the same form as described for operands in stack IPROC;

```
operator--1 = +
          2 = -
          3 = *
          4 = /
          5 = **
          0 = end of the expression
```

PROC Skeleton Stack - IPROC

Pointer: IPROCP

This stack contains the interpretive description of each defined PROC. It is used to initialize the IPROC array in the Function Processor driving the execution of the PROC Expansion module.

```
Format:  PROC i             next-string-ptr
         (Table TPROC(i)    element-type, next-element-ptr
          attribute)        element-operand
                                                       } element 1
                                 :
                            element-operand_k
                                                                       } string 1
                            element-type,next-element-ptr
                            element-operand
                                 :                     } element 2
                            element-operand_i
                            next-string-ptr
                                 :                                     } string 2
                                 :
                                                                          :
```

next-string-ptr--0 if this is the last PROC string; or,
points to the next string entry within IPROC.

next-element-ptr--0 if the last element of this string; or,
points to the next element entry within IPROC.

element-type--control code defining the type of element:

Element Type	Element-Operand[s]
1=Replacement	left variable operand-ptr right side operand-ptr
2=Condition test	neg-flag, cond-code operand-ptr operand-ptr
3=NULL element	[No operand]
4=FALSE element	[No operand]
5=PROC call	neg-flag, proc number (table TPROC index) operand-ptr [argument 1] [if any] : operand-ptr [argument n]

Element Type	Element-Operand[s]
6=Code request	operation code pointer (within table NOPER)
	operand-ptr [argument 1] [if any]
	⋮
	operand-ptr [argument n]
7=Support function call	function number (table PRCLST index)
	operand-ptr [argument 1] [if any]
	⋮
	operand-ptr [argument n]
8=Substring	[no operand]

neg-flag--on if element preceded by NOT prefix; OFF otherwise.

cond-code-- 1=LT, 2=GT, 3=EQ, 4=NE, 5=LE, 6=GE.

operand-ptr--A general PROC operand of the form:
operand-type, operand-value

Operand-Type	Operand-Value
0=variable	variable position within IPROC
1=integer	the integer value itself
2=literal string	string index within CLITS table
3=location counter ($ reference)	0
4=symbol pointer contained in a variable	variable position within IPROC
5=expression	starting position of the expression within PEXP stack
6=PROC argument	argument number
7=TYPE OF operand	argument number
-K=simple expression of the form: operand + K or K + operand	operand-ptr for an operand of types 0, 3, 4, or 6.

1.2.3 Arrays

The following arrays are declared within MECOM as fixed-length arrays having the described format.

Register Types - RTYPE

Size: 26

Format: One word per alphabetic letter:

bit i: On if register i is included in the register class, off if not.

Internal/External Conversion - INT

Size: [total number of Meta-characters, to be determined at implementation time]

Format:

word i: Target machine representation for Meta-character i.

Computational Modes - LMODES

Size: 13

Format:

word i: Zero if mode i is allowed in the language being defined.

Operator Flags - LOPERS

Size: 23

Format:

word i: hierarchy, commutivity, associativity for operator i.

1.3 Meta-Translator Bootstrap

The development of MECOM from the existing METRAN Meta-Translator will proceed as a series of discrete steps, each step resulting in a more powerful version of MECOM. Each new version will thus have additional Meta-Language capability which may be utilized to define the next version. The final result will be a Meta-Compiler (MECOM) capable of generating itself from a description of itself in Meta-Language as well as the required C.W.S. compilers. Thus, MECOM will be generated as a compiler, although only the Source Processor module and its supporting routines will be utilized.

The development phases and the capability included at each phase are as follows:

Phase 1: Modify existing stack and table structures

a. Hand-code or modify all table support modules (see 2.2);
b. Meta-code table attribute and symbol table parsers; hand code attribute pack/unpacking code generation;
c. Add READ[WRITE] TABLES capability via Meta-Language and QTABD, QTABR modules;

Result: Extended METRAN with dynamic symbol tables and stacks, along with parallel attributes and initialization capability.

Phase 2: Add lexical preprocessing capabilities

a. Recode QMORE, QTERM, QCONV modules;
b. Meta-code required parsers; generate main driver program;
c. Using all the above extensions, rewrite METRAN using more powerful Meta-language. Delete SCAN and NOSCAN logic.

Result: MECOM, version 1.

Phase 3: Element extensions

a. Meta-code extended element capability within rule definitions:

   1. Symbol table manipulators
   2. STRING element
   3. TEST element
   4. FIELD element
   5. FREE element
   6. SAVE/RESTORE elements

Result: Extended MECOM, version 1

Phase 4: Code generation refinements

a. Convert rule definition string and element parsing to new scheme (see Rule Definition module);

b. Improve generated code; provide linkages to semantic processing routines;

c. Utilize new false/true label management algorithm in code generation;

d. Remove all current 'M' support routines, or recode those required in Meta-Language;

e. Meta-compile MECOM.

Result: MECOM, version 2

Phase 5: Target definition capability

a. Add target declaration parsers;

b. Add PROC encode parsers;

c. Code Library Update module;
   incorporate Compiler Library Data file.

Result: MECOM, version 3

Phase 6: Language declarative extensions

a. Add parsers for language declaration statements;

b. Redesign parser front-end code generation for INITIALIZER and STATEMENTS processing; delete MODULE CONTAINS and BEGIN MODULE capability;

c. Add options parsing capability.

Result: Completed METCOM.

2. SOURCE PROCESSOR

The design of the Source Processor is described in this section. The major modules and submodules are identified, their functions are inter-related, and the supporting internal data structures are described. A collection of hand-coded semantic support functions are defined as to function and calling sequence.

The main body of the Source Processor, in the form of a collection of parsers, is generated by the Meta-Compiler as a function of the input Meta-Language.

2.1 Program Logic Modules

The Source Processor consists of a collection of distinct modules organized into two basic groups: syntax parsing and semantic processing.

The syntax parsing routines are further divided into two sections: the statement parsers and the syntax support modules. The statement parsers are generated in FORTRAN as a result of the mlv definitions of the compiler language provided by the compiler writer. The syntax support routines are hand coded in FORTRAN and are table driven to a certain extent by the arrays described in section 2.3, which are initialized by the Meta-Compiler as a result of declarations in Meta-Language.

The semantic processing routines are organized into two general groups: utility support and language semantics. The utility routines provide for general compilation support of all other modules. Language semantics routines respond to specific processing requirements of the Semantic Profile of the application programs.

Figure 2, the Source Processor block diagram, illustrates the interrelationships between each module and the internal and external data paths.

SOURCE PROCESSOR BLOCK DIAGRAM

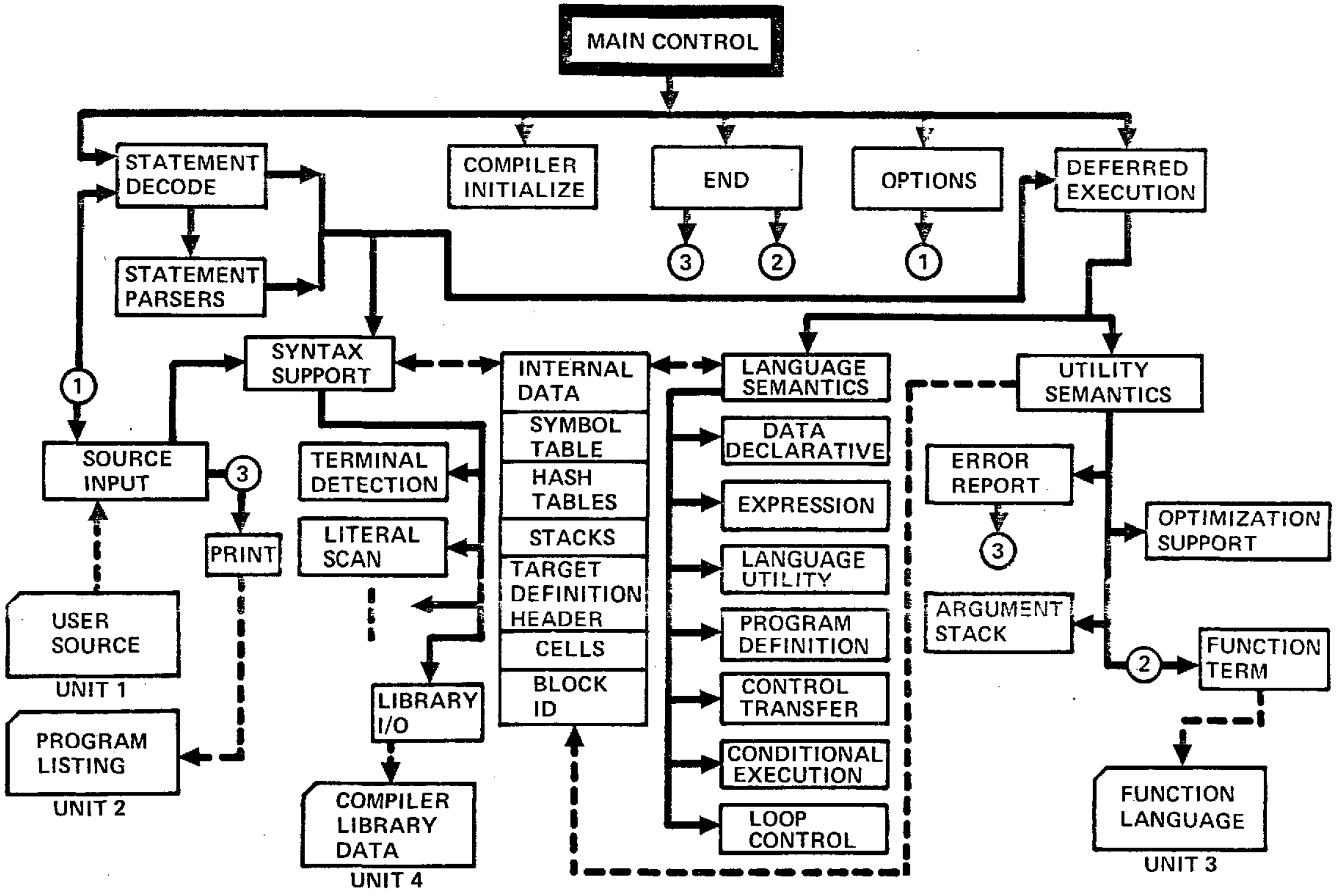


FIGURE 2.

MODULE: Main Control

FUNCTION: Coordinates and controls the execution of all other logic modules and sub-modules of the Source Processor.

PROCESS:

The Main Control module initiates execution of the Source Processor and retains control at all levels. It performs the phased execution of a collection of overlay modules, each of which is responsible for syntax and semantic processing of a section of the compiler language.

The Options module is executed to read the first user card image and extract the option parameter settings. The Compiler Initialize module is then called to perform initialization functions, such as reading any predefined symbol or hash tables and stacks, clearing pointers, etc...

The Statement Decode module is then repetitively executed to recognize each language statement and call the corresponding processor. Subsequent to processing a given statement, control returns to Main Control and the Deferred Execution module is executed to process any deferred procedure calls.

Upon encountering the ending statement of the language, MAIN calls the End module to generate the storage and linkage maps completing the compiler listing (if any). The Function Language file (unit 3) is then rewound and control is passed to the Function Processor.

IMPLEMENTATION:

Main Control is coded in FORTRAN as a standard driver program.

MODULE: Compiler Options

FUNCTION: Reads and parses the options card.

PROCESS:

A single card image of the format specified in section 1.4, volume I, is parsed and the compiler control parameters are saved.

IMPLEMENTATION:

Hand coded in FORTRAN.

MODULE: Compiler Initialize

FUNCTION: Initialize all structures controlling compiler execution.

PROCESS:

Any stacks or symbol and/or hash tables requiring presetting to initial values or strings are read into memory by calling the Library Input syntax support module. All other control cells, including stack and table pointers and lengths are automatically cleared. The BLOCK DATA initialization data generated by the Meta-Compiler is also included in this module to set parser control information (literals, etc...).

IMPLEMENTATION

This module is generated automatically as a function of the $INITIALZER mlv in the Meta-Language definition (see Volume I).

MODULE: Deferred Execution

FUNCTION: To execute any deferred semantic procedure calls.

PROCESS:

This module is executed by the Main Control routine in order to execute any stacked procedured calls saved through the true parsing of the current compiler language statement by the statement parsers. In addition, any warning errors, normal errors, or fatal errors are listed at this time.

IMPLEMENTATION:

Automatically generated through detection of semantic requests in the Meta-Language compiler description.

MODULE: Print Output

FUNCTION: Prints a line image from the designated argument buffer with internal to external conversion.

PROCESS:

The argument array is converted from internal character numbers to external 'A1' format characters, which are then sent to the standard print output device (unit 2).

IMPLEMENTATION:

Exists in FORTRAN, and is equivalent to the Print module of the current Meta-Translator.

MODULE: Statement Decode

FUNCTION: Identifies each compiler statement and invokes the corresponding processor.

PROCESS:

Each compiler input statement is typically associated with one or more processing levels. Each level has an associated processing module for handling all statements at that level. The Statement Decode module identifies each individual statement in a given level and maintains the level counter. Unrecognizable statements or statements appearing out of the declared order (i.e., a previous level) are flagged as such.

Upon decoding a particular language statement, its associated processor is executed and control returns to the Main Driver module. Each statement processor is represented by code generated as a result of mlv definitions in Meta-Language.

IMPLEMENTATION:

This module is generated from a predefined mlv by the Meta-Compiler. The STATEMENT LEVELS Meta-Language declaration defines the recognition criteria, processing level, and occurrence frequency for each processing and support mlv.

MODULE: Source Input

FUNCTION: Reads and formats compiler language statements.

PROCESS:

The Input module supports all parsing functions through maintenance of the input image buffer and the control cells CURSOR and LENGTH.

Whenever CURSOR exceeds LENGTH this module is invoked to perform the following tasks:

a. If a field has been terminated and another field is available, CURSOR and LENGTH are updated and control returns; otherwise,
b. A new card image is read in, the Print module is called to list it, and the declared continuation rules are applied to see if the card is part of the current statement. If not, a statement end character (-1) is placed in the input and control returns; otherwise,
c. The card image is pre-scanned as a function of the lexical PRESCAN directives. This results in comments, noise words, and possibly blanks to be skipped, and literals and other constructs to be copied.
d. The resulting modified card image is added to the image buffer as part of the current statement, and the LENGTH and CURSOR cells are updated.

The arrays QTERMD, QFIELD, and QPRES are used to interpretively drive the process.

IMPLEMENTATION:

This module and its supporting sub-modules are to be hand coded in FORTRAN. Certain submodules involve modification of existing support routines for the Meta-Compiler, developed via the bootstrap development of the Meta-Compiler as outlined in section 1.3. In fact, the implementation of source input mechanism in the fully bootstrapped Meta-Compiler will result in a 95% complete implementation of the compiler Source Input module.

MODULE: Syntax Support

FUNCTION: Provide utility functions manipulating characters in support of all parsers.

PROCESS:

This module actually consists of a collection of submodules designed to support the character manipulation functions required during parsing activity. The submodules are linked directly within the generated code constituting the statement parsers via parameterized subprogram calls. Thus, the mlv's written in Meta-Language by the compiler writer results in a series of syntax support function calls in most cases.

The key submodules and their overall tasks are described below:

- Character Conversion (QCONV)
  Provides for internal to external, external to internal character conversion for all language characters.
- Literal Scan (QLITSC)
  Searches for a match between a literal and the input image buffer in support of literal mlv operands.
- Reoccurrence Maintenance (QREOCB, QREOCE)
  Supports the repeated application of a group of mlv elements organized as a reoccurrence element, including cursor maintenance, control stack (QCTAB) maintenance, etc...
- Table Manipulation (QSTACK)
  Supports the entry of symbols or literals into symbol tables and hash tables. Maintains the arrays QTABP, QTABS, and QSTACK containing hashed strings and pointers.

- Step-Up/Down (QSTEP)

  Handles the step down into an mlv, with the associated saving of information on stack QCTAB, the control stack. This module also restores this saved status information when a step up from an mlv is performed.
- Terminal Detection (QTERM)

  Searches for a parameterized number of occurrences of a given terminal syntax element starting at the current position in the image input buffer.

All the above syntax support modules currently exist as support routines to the existing Meta-Translator. They require only minor modifications to provide full syntax support to the Source Processor. The support submodules listed below are new and will be hand coded:

- Stack Manipulation (QSTKSR, QSTKM)

  Adds and deletes entries from dynamic stacks and searches for values contained on stacks.
- String Manipulator (QSTRM)

  Supports the Meta-Language STRING element. This module will be coded in a similar manner to the existing TEXT support routine QTEXT, except that the textual strings will be formed and concatenated in the input buffer rather than the output buffer.
- Attribute Maintenance (QATTR)

  This support function will be generated as a function of the SYMBOL ATTRIBUTE and TABLE Meta-Language statements by the Meta-Compiler (see section 1). It will handle packing and unpacking of all table structures.

- Compiler Library Input

  The module inputs both a compiler initialization entry and a target definition header from the Compiler Library Data file (unit 4). The initialization data contains initial values for stacks and hash or symbol entries and their attribute fields. The data is associated with the particular compiler by name. The header, or fixed length, portion of the designated target machine is also input since certain semantic functions in the Source Processor utilize this information. The file is left positioned at this point so that the remainder of the target definition (PROCS, etc...) can be read in by the Function Processor during the next phase of compilation.

- Compiler Library Output

  This module handles the creation of a compiler initialization data entry on the Compiler Library Data file. The data is associated with the name of the compiler.

MODULE: Utility Semantic Support

FUNCTION: Provide utility functions to support all parsers and semantic functions.

PROCESS:

This module actually consists of a collection of submodules designed to perform various support tasks for the statement parsers and language semantic processors as indicated below:

- Argument Stacking (QARGIO)

  Handles the stacking of all variable length deferred and immediate calls from parsers to support functions, or between support functions themselves. The arguments are stacked onto control array QCTAB from the argument descriptor array (QPARGS) such that multi-level deferred calls can be made concurrently.

- Error Report (QERR)

  This module prints error messages associated with FATAL and ERROR Meta-Language elements onto the print file. Any control diagnostics, such as dynamic table/stack overflows or control stack overflows are also printed directly along with any selected debug monitoring messages resulting from compiler options and the DEBUG Meta-Language element.

The above utility support functions currently exist as support routines in the existing Meta-Translator system. They require only minor modifications for operation within C.W.S.

The following support function is required to facilitate generation of Function Language output:

- Function Term

This module accepts a variable number of arguments and creates a single parameterized Function Language term on unit 3 as output. The first argument provided is an integer value representing the desired F.L. term operator. The remaining arguments provided are the F.L. parameters and include:

(a) symbol pointers

(b) integer values

(c) literal string pointers

MODULE: Optimization Support

FUNCTION: Partition source program into basic blocks.

PROCESS:

The semantic functions in the Language Semantics group require support in the area of block partitioning of a user source program and maintenance of associated labels. This information will be subsequently utilized by the Function Processor code generation mechanics and optimization submodules.

The Optimization Support module consists of a collection of submodules to process block formation and interrelationships. The resulting information is encoded into the Block Identification and Symbol Tables, and special Function Language terms are generated. The overall logic is illustrated by Figure 2-2.

Each major submodule is described below as to function and effects. The call requests and interfaces with the Language Semantics routines are described in section 2.2 for each individual routine.

Block Identifier:

- Creates a new program block;
- If a label is passed as an argument, it is associated with the block; otherwise, an internal label is created for the association;
- The new block entry is encoded into the Block Identification Table;
- The label association with the block is marked in the Symbol Table, label segment.

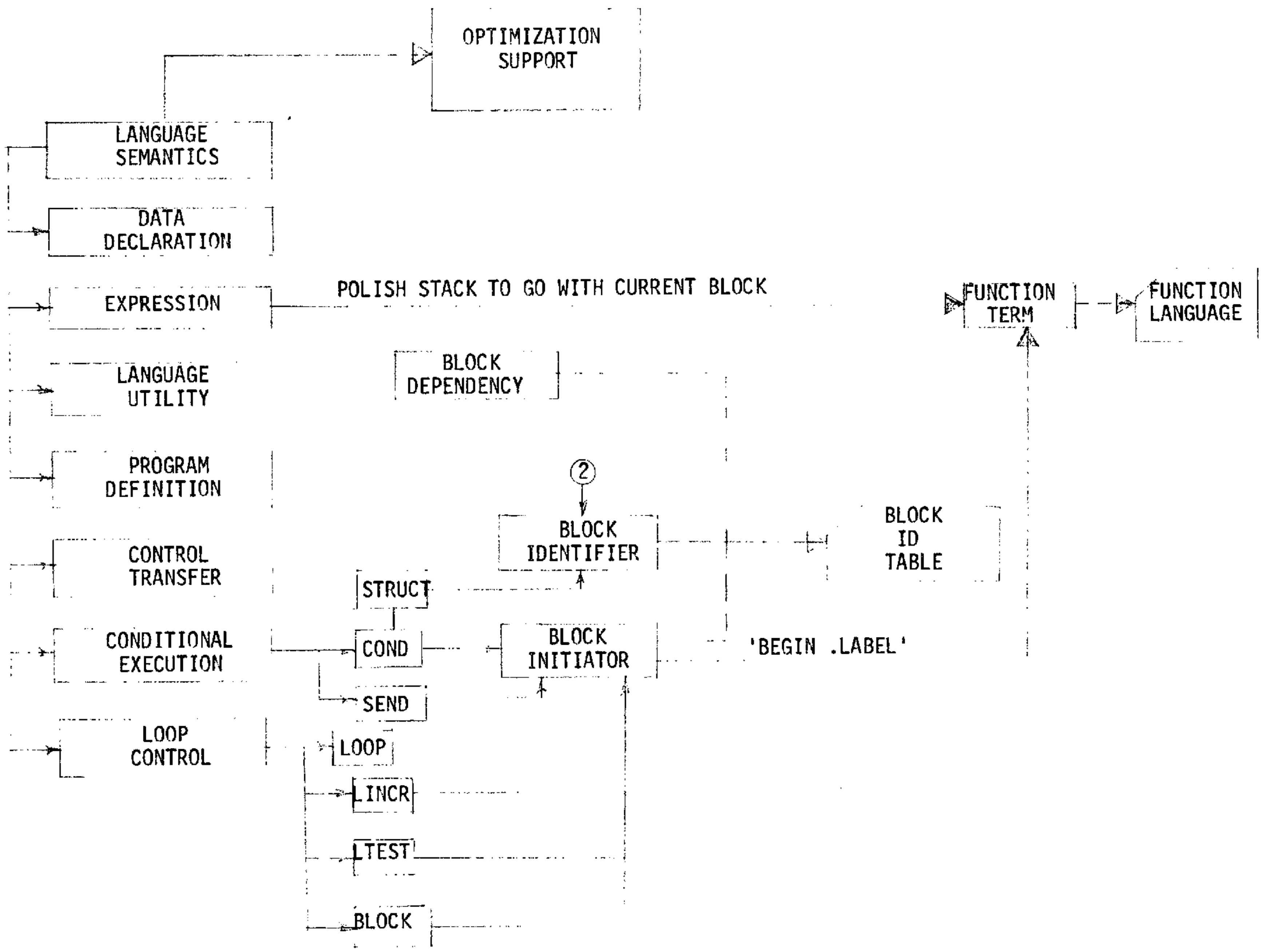


FIGURE 2.2 OPTIMIZATION SUPPORT LOGIC FLOW

Block Initiator:

- ° Outputs a 'Begin Block label' function term;
- ° The block associated with the argument label is activated. All code following up to the next call to this routine will be associated with this block;
- ° Multiple block initiation requests to the same block are allowed.

Block Dependency Handler:

- ° For a single label argument, marks a path from the currently active block to the argument block;
- ° For two label arguments, marks a path from the block associated with the first label to the second block;
- ° If either label is undefined (i.e., not yet associated with a block), the Block Identifier is called to associate the label[s] with block[s].

New Label Handler:

- ° Its argument label is passed to the Block Identifier (if undefined) to assign it to a block;
- ° The Block Initiator is called to start a new block.

New Statement Identifier:

- ° Calls the New Label Handler if the new statement has a label (or labels);
- ° Calls the Block Dependency Handler to reflect a flow path to the newly created block from the previous block if not in a transfer of control mode (see 2.2);
- ° Checks for the statement having a label if in a Transfer of Control mode;
- ° Resets any transfer of control mode.

MODULE: Language Semantics

FUNCTION: Provide for the semantic processing of parsed operands in the statement parsers.

PROCESS:

This module consists of a collection of submodules, each performing the required processing of a high-level, highly parameterized, semantic request. The semantic requests are found within the mlvs written by the compiler writer in the statement processors.

The Argument Stacking Utility Support routine handles the required linkage and passes the actual argument values to the specific semantic processor.

The following section (2.2) defines the function, calling sequence, arguments, and effects of each of the currently specified semantic functions. Each corresponds to a specific language requirement, and must be hand-coded in FORTRAN.

MODULE: End

FUNCTION: Terminates the Source Processor.

PROCESS:

The End module is initiated when the END statement of the compiler language is encountered and the END semantic request is invoked by the compiler writer in the processing mlv.

The END Function Language term is output to the Function Language file. The program print listing is completed with the addition of the variable storage map and STATISTICS (see 2.3).

Any unterminated loop or conditional blocks are flagged by calling the Error Report module.

2.2 Semantic Support Functions

The submodules defined in this section act as support routines to the Language Semantics module. The routine descriptions are divided into semantic categories corresponding to the semantic profile. The generated Function Language resulting from each semantic call is indicated where applicable. The Function Term support submodule is called to pass any F.L. terms to unit 3.

In the descriptions of the semantic procedure arguments a reference to a symbolic name of any type (variable, array, etc...) used as an argument has any of the following forms:

- A symbol pointer to the symbol table segment or hash table and the sequence number within;
- An mlv representing the symbolic character strings. The mlv will be automatically placed within the symbol table in the appropriate segment.

C-2

2.2.1 Data Declarative Functions

FUNCTION: ARRAY   Array Declaration

CALL: ARRAY (N, D1[,DI])

ARGUMENTS: N -- Array name.

DI -- Integer value of the i'th dimension of N, or a simple variable name for the i'th dimension.

EFFECTS:
1. Marks N as an array and links the dimension vector D1,....., DI to N through the symbol link.
2. The dimension vector is stored in QTABLE.

FUNCTION: EQUATE Equivalence of Symbols

CALL: EQUATE (-1,S[,DI][-1,S[,DI]])

ARGUMENTS:	S	-- Variable name symbol pointer
	EI	-- Index vector of integer constant pointers defining the logical offset to S [if any array or table item].

EFFECTS:

1. The arguments collectively define an equivalence group that is to be assigned the same memory address. For arrays, tables, and table items the group address is assigned to the indicated index element.
2. For a simple variable S, there are no DI.
3. For an array or table item S, the number of DI should match the dimensionality of S or equal one, in which case DI is the index offset to S.
4. All such equivalence data is saved by group in the array QTABLE exactly as input via the call arguments. This information will be subsequently utilized by the Memory Assignment module of the Function Processor.

FUNCTION: COMMON COMMON Data Declaration

CALL: COMMON(NAME,S1[,SK])

ARGUMENTS: NAME -- Global (COMMON) block name.

S1,...,SK -- Order-dependent pointers to variables or arrays.

EFFECTS:
1. Places NAME in the symbol table as a block name.
2. Links each SI with NAME in the symbol table.
3. Sets the relocatability of each SI to "global COMMON."
4. The Function Processor assigns NAME a separate control section and gives each SI a relative offset based upon its order of declaration, precision, and size.
5. Multiple Global references to the same NAME are allowed, but only one reference for each SI.

FUNCTION: ITEM   ITEM DECLARATION

CALL: ITEM(NAME,MODE,SECMODE,PREC,S,R,FRAC,BIT,PK)

ARGUMENTS: NAME -- **The item name.**

MODE -- The item mode

SECMODE -- Secondary item mode (0=none)

PREC -- Precision in bits (bytes for texual items)

S -- Signed Flag (0=NO,1=Yes)

R -- Round Flag (0=NO,1=Yes)

FRAC -- Number of Fractional bits (for fixed point items)

BIT -- Bit starting position within allocated cell [generally appropriate only for table items]

PK -- Packing Density

EFFECTS:
1. The item name is placed in the symbol table (if an MLV or string).
2. The indicated attributes are flagged in the symbol table.
3. If within an internal procedure or function, the item is marked as "local". If it matches a previously occurring global symbol, a new entry is forced into the symbol table.
4. A check is made to insure that only one item declaration is active at any given time.
5. All the input parameters except the first are optional. A null parameter should be represented by successive commas (i.e., ,) if any parameters follow.

FUNCTION: INIT Data Initialization

CALL: INIT (-J,V[,I][,-J,V[,I]],0,M,C[,M,C]

ARGUMENTS:	V -- Variable, array, table, or table item name pointer
	I -- Integer-Valued subscript pointers for V (if array or table item)
	M -- Integer Repeat Values
	C -- Constant Pointers
	J -- Integer Increment Value for the following V.

EFFECTS:

1. The variables (V) are initialized to the constants (C).
2. There must be a one-to-one correspondence between the variables and constants; or, a single Variable may occur defining a base location for the constant list.
3. The repeat values M allow the following constant C to be repeated M times.
4. The variable pointers V (along with subscripts I) define the initial location to start storing the corresponding M constants C. This location is incremented by J cells each time.

GENERATED FL:

```
DATA   -J,V[,I],0,M,C[,M,C]
 .
 .
 .
 .
```

FUNCTION: TABLE Table Declaration

CALL: TABLE(NAME,TYPE,R,PACK,NUMEN,WDSENT,I,WDNUM,BITNUM,IPACK])

ARGUMENTS: NAME -- Table Name Pointer or MLV

TYPE -- Parallel (=0)

Serial (=0)

R -- Table Rigidity (0=rigid)

PACK -- Packing - 0-None, 1-medium, 2-dense, 3-tight

NUMEN -- Number of Entries

WDSEN -- Number of Target Words/Entry (0=compiler determined)

I -- Table Item Pointers

WDNUM -- Word position for I (-1=compiler determined)

BITNUM -- Bit position for I (-1=compiler determined)

IPACK -- Item packing density (0=none)

EFFECTS:

1. A [rigid] parallel or serial table is defined with designated packing density, number of entries, words per entry, and table items.
2. The **conventions** with respect to allocation of table items within target words are identical to those of the language SPL.
3. Each table item is linked with the table name NAME through the symbol table.

2.2.2 Expression Manipulation

Expressions are manipulated and resolved into a Polish string through the use of two stacks: PSTACK and QSTACK. These two stacks are always present whenever expressions are utilized as source elements.

PSTACK is the polish stack of operands and operators representing the expression; its associated pointer is PPOINT. QSTACK is a temporary stack for holding operators to be placed on PSTACK on a priority basis as well as operand computational modes; its associated pointer is QPOINT.

The hierarchies of each operator are found in the vector QLOPER. This vector is preset to the normal operator hierarchies but may be changed by the compiler writer via an OPERATOR Meta-Language declarative.

The format of PSTACK and QSTACK is described in section 2.3. Figure 2-1 gives the logic flow for the expression handling routines referenced in this section.

EXPRESSION MANIPULATION LOGIC

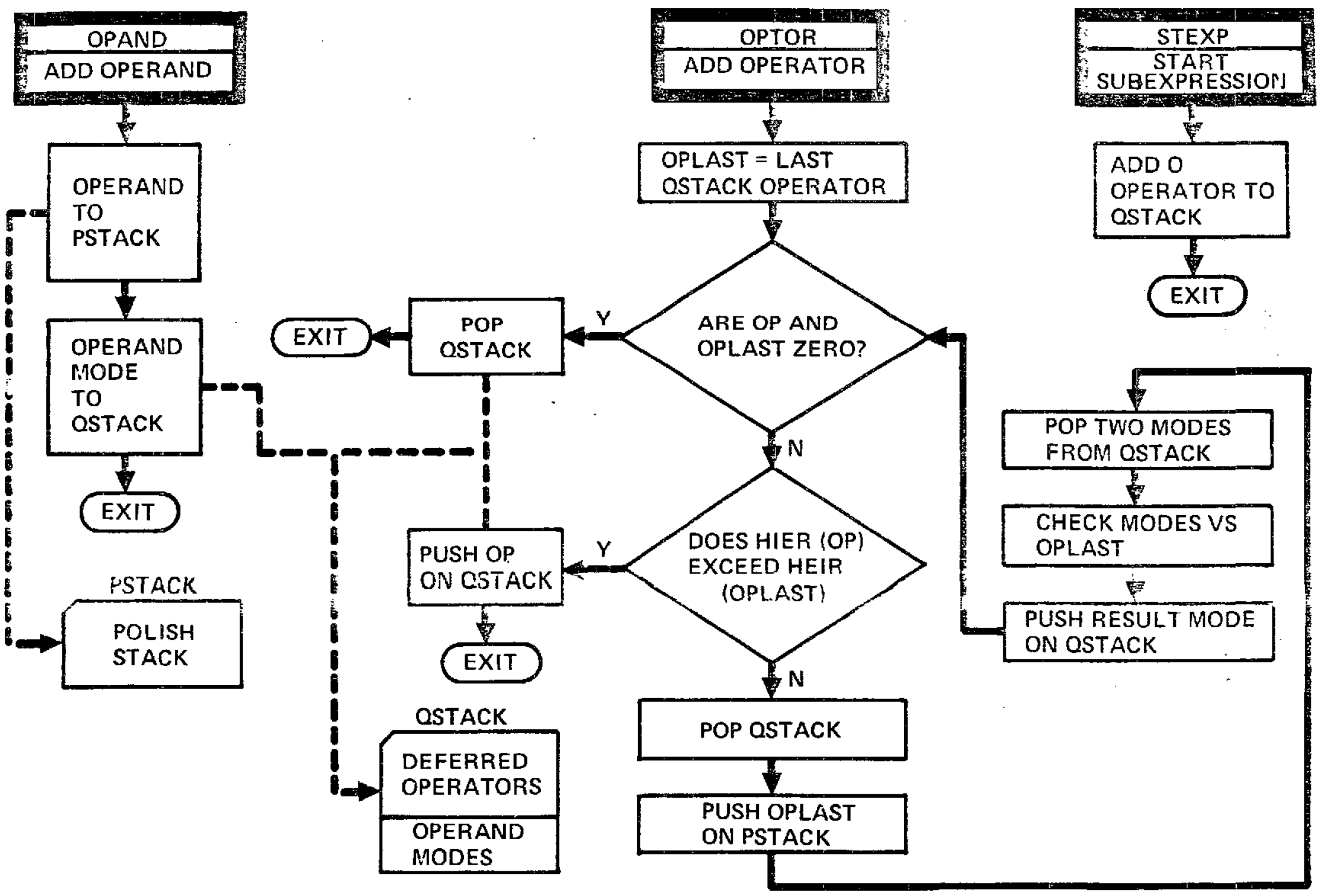


FIGURE 2-1

FUNCTION: STEXP   Begin A (Sub)Expression

CALL: STEXP

ARGUMENTS: None

EFFECTS:

1. Indicates the beginning of a distinct subexpression, i.e., a collection of operands and operators collectively representing a separate item (possibly embedded within a larger expression).
2. The effect of this function call is: 'PUSH 0 on QSTACK'
3. Subexpressions examples include:

   ```
   ...A+(B*C-E)-....
   ...B*A(E-1,D,F+2)
   A=B*C-E
   ```
4. A subsequent occurrence of an 'end expression' operator (i.e., OPTOR(0)) causes all stacked operators down to this point to be moved from QSTACK to PSTACK.

FUNCTION: OPAND   Add Operand to Polish String

CALL: OPAND (PTR)

ARGUMENTS: PTR -- the operand symbol pointer

EFFECTS: 1. The operand (PTR) is placed on the cumulative polish string PSTACK.

2. The effect of this function is equivalent to:

'PUSH PTR ON PSTACK'

3. The computational mode of the operand is fetched from the symbol table and is stored on QSTACK (the mode portion).

FUNCTION: OPTOR Add Operator to Polish String

CALL: OPTOR (OP)

ARGUMENTS: OP -- the operator number

EFFECTS:
1. The operator OP is added to the polish string PSTACK on a priority basis.
2. The declared hierarchy of an operator determines whether it is placed immediately on PSTACK or is temporarily stacked on QSTACK.
3. A 'terminate (sub)expression' operator (OP=0) causes all delayed operators (on QSTACK) down to the most recent 'begin expression' (0) operator to be placed on PSTACK on a last-in first-out basis.
4. A 'begin list' or 'end list' operator, signifying the beginning or ending of a parameter list for the last operand (procedure, array, table, or table item), is placed directly on PSTACK.
5. If the hierarchy of OP exceeds the hierarchy of the last entry on QSTACK, OP is placed on QSTACK (the hierarchy of 'begin expression' (0) is defined as zero).
6. If not case 5, the last operator on QSTACK is placed on PSTACK and QSTACK is popped once. The stacked operator is then checked for mode compatibility with the last two mode entries on QSTACK (mode portion) using table QIMODC. The two mode entries are then popped from QSTACK (mode portion) and are replaced on QSTACK by the result mode. The result mode is typically the maximum of the two modes depending on the operator (i.e., the result mode of a relational operator acting on two arithmetic operands is logical.
7. Step 5 is repeated.

FUNCTION: GEN   Generate An Expression

CALL: GEN(PI[,PI])

ARGUMENTS: PI -- Operands, operators, or Polish string stacks.

EFFECTS:
1. A polish string is formed as a composite of the arguments PI, and is passed through the Function Language.
2. The PI are formed in the resulting polish string in the order presented.
3. A operand which is all or a portion of a stack (typically PSTACK) is assumed to have been previously generated via calls to STEXP, OPTOR, and OPAND.
4. The composite expression is not mode checked for operand/operator mode compatibility (see OPTOR).

EXAMPLES:
1. GEN(ELEMENTS OF PSTACK)

   Pass the previously formed polish string PSTACK to the Function Language.
2. GEN(I,ELEMENTS OF PSTACK,.STORE)

   Pass a polish string for storing an expression result in PSTACK into the variable pointed to by I.
3. GEN(I,I,N,.PLUS,.STORE)

   Pass a polish string to perform: I=I+N

GENERATED FL:

GEN E [E is the resulting composite polish string formed from the PI]

2.2.3 Utility Functions

FUNCTION: UCON   Create A Universal Constant

CALL: UCON(ARG[,IM][,PREC])

.L

ARGUMENTS: ARG -- STRING, MLV, or integer value

IM -- the constant mode

PREC -- the constant target precision

EFFECTS:

1. Forms a machine - independent constant in the vector VALUE from the argument ARG.
2. If ARG is simply a value, it is placed as is in VALUE. The global cells IM and PREC are then set to the second and third input parameters. Default values for missing parameters are 'integer' and WORD.
3. If ARG is a string or MLV it is parsed to form the machine-independent value at VALUE. Legal constant types include:

   signed integers

   signed real and double-real with optional signed exponent

   complex

   fixed point

   Hex

   Octal

   Binary

   Literal Strings (Texual)

   The constant syntax is as defined in FORTRAN and SPL.
4. If the IM argument is missing, IM is set to the mode derived from parsing the ARG. If the argument IM is Hex, binary, or octal the appropriate base is used for the number conversion; otherwise, the

base 10 is assumed. Texual constants require an input IM of 'texual' and the entire string of MLV ARG defines the texual constant, i.e., no parsing function is performed.

5. If PREC is missing it is set to the minimal word size required to hold the constant value for integer, real, fixed point, and complex; and the minimal byte size required otherwise. If PREC is present as an argument, it must be greater or equal to the derived minimally required precision.

EXAMPLES:	*UCON(1)	Forms an integer constant of one target word.
	*UCON('1.0',REAL,32)	Forms a floating point constant of 32 bits.
	*UCON('ABCD',TEXUAL)	The texual constant ABCD is formed.
	*UCON($CON)	The MLV CON is parsed to determine the constant value.

FUNCTION: MKCON Make a Constant Entry

CALL: MKCON(ARG,IM,PREC)

ARGUMENTS:	ARG --	String, MLV, or integer value
	IM --	The constant mode
	PREC--	The constant target precision

EFFECTS:

1. Calls UCON to build a universal constant.
2. Places resulting CONVAL vector in the symbol table as a hashed entry.
3. Sets SYMP to resulting symbol pointer.

FUNCTION: STATID Statement Identification

CALL: STATID[(label[,label]$_n$)]

ARGUMENTS: label -- Statement Label Pointers (if any).

EFFECTS:
1. Defines the beginning of a new statement with labels label,....
2. The labels are eventually associated with the first instruction of a block of code.
3. The New Statement Identifier module is called with the argument labels (if any).

GENERATED FL:

```
STAT
[LDEF LABEL[,LABEL]n]  if labels
[BEGIN LABEL]
```

FUNCTION: LREF   Label Reference

CALL: LREF(LABEL[,LABEL])

ARGUMENTS: LABEL--Statement label pointers.

EFFECTS:
1. Indicates a reference to a program label within the body of a statement.
2. The label[s] are placed in the symbol table.
3. The Block Dependency Handler is called with the argument labels to handle block dependency.

FUNCTION: ADATA Assign Data to Control Section

CALL: ADATA(LAB,ARG)

ARGUMENTS: LAB -- Label pointer to assign to the data to be output.

ARG -- MLV - MLV character string

String - Literal string

Value - Data constant pointer

EFFECTS:

1. Assigns LAB to next cell in the data control section (if present).
2. Dumps the string (MLV or Literal) or indicated constant value at the label location.

GENERATED FL:

```
ADATA  LAB,ARG  [if LAB is present]
```

FUNCTION: MODIFY   Modify a Symbol

CALL: MODIFY (VI[,VI])

ARGUMENTS: VI -- Symbol Pointers

EFFECTS: 1. Tells the Function Processor optimizer that each VI is changed through execution of the current statement.

2. Sample usages - READ statements; direct code.

GENERATED FL:

```
MOD  VI[,VI]
```

FUNCTION: NEGATE Make Negative Constant

CALL: NEGATE[(I)]

ARGUMENTS: I -- Constant Pointer

EFFECTS: 1. If I is present the negative of the constant value is placed at VALUE; otherwise, VALUE itself is negated.

FUNCTION: SIGN   Get Constant Sign

CALL: SIGN(CI)]

ARGUMENTS: I -- Constant Pointer

EFFECTS: 1. Returns the sign (0=positive, 1=negative) of the constant pointed to by I (if present) or of VALUE.

FUNCTION: LDCON   Load a Constant Value

CALL: LDCON(I)

ARGUMENTS: I -- Constant pointer

EFFECTS: 1. Moves the value of the constant into VALUE.

FUNCTION: PUTCON

Put a Constant Value Into Symbol Table

CALL: PUTCON

EFFECTS: 1. Puts the constant value in VALUE into a unique position in the symbol table.

2. The resulting symbol pointer is SYMP.

FUNCTION: CONOP   Perform Constant Operation

CALL: CONOP(OP,I[,PREC])

ARGUMENTS: OP -- An operation to perform between constant values.

I -- Second constant operand pointer.

EFFECTS:

1. VALUE is replaced by VALUE OP I, where OP is

1	+
2	-
3	*
4	/

2. A mode conversion is performed if the modes of the two operands differ.
3. If PREC is specified, it must be large enough to carry the result. If not specified it is set to the minimal word size required.

2.2.4 PROGRAM DEFINITION

FUNCTION: PROG(NAME,REC,REN,M,L[,PI,RI]$_n$)

ARGUMENTS: NAME -- Subprogram Name (MLV, string, or pointer)

REC -- Recursive Flag (0=No, 1=Yes)

REN -- Reentrant Flag (0=No, 1=Yes)

```
M    -- Type -----  0 = Main program
                    1 = Subroutine subprogram
                    2 = Function subprogram
                    3 = Closed subroutine [see CLOSE function]
                    4 = Entry Point [see ENTRY function]
L    -- Linkage --- 0 = DEF symbol [an entry point]
                    1 = REF symbol [defined externally]
                    2 = Internal only
```

PI -- Parameter Pointers

RI -- Parameter Type (0=neither, 1=output, 2=input, 3=both)

EFFECTS:

1. NAME is placed in the symbol table (if an MLV or string) and is marked as a global procedure name of type M and linkage L.
2. The PI are marked as dummy (formal) parameters of type RI and are linked to NAME. They are considered to be local symbols.
3. An externally defined function (L=1) causes no FL generation - its Symbol Table entry contains all required information.
4. A global subprogram (L=0) causes the subprogram name pointer to be saved in cell QGSUBP as the active global subprogram.
5. A new block is created and initiated with the current block pointer saved. The current block will be restarted upon encountering the subprogram EXIT call. The new block is associated with the subprogram name in the Symbol Table.
6. An exit block is created and its label pointer is saved in a cell for subsequent use by the EXIT (and RETURN) functions.

GENERATED FL:

L=1 (global subprogram name):

```
          BEGIN      1               [the subprogram prolog block]
          [.exit]
         {MAININN}                     [m=0]
         {ESUBIN }   NAME, REC, REN    [m=1 or 2]
          EXDUMMY    1, P1, R1         [if any arguments]
            :
          EXDUMMY    i, Pi, Ri
           B         .begin          [entry point transfer label -
                                      see END function]
          BEGIN      2               [start block 1 - first body block]
```

L=2 (internal subprogram):

```
          BEGIN      1               [the subprogram prolog block]
          [.exit]
          ISUBIN     NAME, REC, REN
          INDUMMY    1, P1, R1
            :
          INDUMMY    i, Pi, Ri
           B         .begin          [first subprogram block]
          BEGIN      .begin
```

FUNCTION: CLOSE

CALL: CLOSE (NAME)

ARGUMENTS: NAME -- Closed subroutine name (MLV, string, or pointer).

EFFECTS:
1. Defines the start of an internal closed subroutine to be later terminated by an EXIT command.
2. The subroutine may not have parameters nor return a value.
3. A new block is initiated after saving the current block label. The closed subroutine is associated with the new block in the Symbol Table.

GENERATED FL:

```
BEGIN  .label
CLOSP   NAME
```

FUNCTION: ENTRY Entry Point Definition

CALL: ENTRY (NAME[,LABEL][,PI,RI])

ARGUMENTS: NAME -- Entry point name (MLV, string, or pointer)

LABEL-- Internal entry point label (assumed to be '*' if missing).

PI -- Parameter pointers

RI -- Parameter type (0=neither, 1=output, 2=input, 3=both).

EFFECTS:

1. Defines NAME as an entry point to the currently active procedure. The attributes REC, REN, and L of the active procedure are applied to NAME.
2. The PI are marked as dummy (formal) parameters of type RI and are linked to NAME. They are considered local symbols.
3. The LABEL identifies the transfer label for the entry point. Its absence implies the current location counter, causing a new block to be created and initiated.
4. The old [current] block is saved for later restoration (see EXIT function).
5. The new block (LABEL or created) is associated with the entry point in the Symbol Table.

GENERATED FL:

```
 BEGIN      1               [the entrance prolog block]
(EENTIN)    NAME,REC,REN    [L=1]
(IENTIN)                    [L=2]
(EXDUMMY    i,Pi,Ri         [L=1]
(INDUMMY                    [L=2]
   :
   B        [LABEL][.begin]
 BEGIN      [LABEL][.begin]
```

FUNCTION: END

CALL: END[(LABEL)]

ARGUMENTS: LABEL -- Transfer point upon entry.

EFFECTS:

1. Terminates compilation of all active programs.
2. If LABEL is present, its associated block is also associated with the transfer label of the global subprogram being terminated (see PROG function).

   If absent, the subprogram transfer label is associated with block 1, the initial code body block.
3. A flag is set signaling the end of compilation to the Main Driver program.

FUNCTION: RETURN   Return from Active Subprogram

CALL: RETURN[(E)]

ARGUMENTS: E -- the returned expression value (or null).

EFFECTS:

1. A transfer is made to the exit point of the active function (see PROG and EXIT) with the value of E, which is converted to the mode of the active function procedure.
2. Sets the Transfer of Control mode, thus requiring a label on the next statement.

GENERATED FL:

```
RETURN
GEN  E
GO   .EXIT   [the exit block for the active subprogram]
```

FUNCTION: EXIT

CALL: EXIT[(E)]

ARGUMENTS: E -- the returned expression value (or null).

EFFECTS:

1. The expression result value (if any) is converted to the mode of the active procedure.
2. The proper exit is made from the procedure or close.
3. All variables and formal parameters local to the procedure are unlinked from the symbol table.
4. The procedure exit block is initiated.
5. Any output parameters cause value return code to be generated.
6. For a termination of an internal function, the saved original code block for the previously active global program is restored.

GENERATED FL:

```
[GEN E]                   [if E present]
BEGIN .exitblock          [start the exit block
⎧EXRETDUM⎫
⎨        ⎬   i,P          [argument return code-type 1 or 3 args]
⎩INRETDUM⎭
    :
⎧MAINEX⎫
⎪ESUBEX⎪
⎨      ⎬ NAME,REC,REN  [terminate active subprogram]
⎪ISUBEX⎪
⎩CLOSEX⎭
BEGIN     .oldblock    [if not terminating the globally active
                         subprogram]
```

2.2.5 PROGRAM TRANSFER CONTROL

FUNCTION: IDXSW — Indexed Switch List Definition

```
CALL:  IDXSW (NAME[[,L][,C][-1,SNAME,IND]],T)
                  n
```

ARGUMENTS:

NAME	--	Switch Name (string, MLV, or pointer)
L	--	Label or location variable pointer
C	--	Closed subroutine name pointer
SNAME	--	Pointer to another item switch name
IND	--	Integer constant or variable specifying the index into the switch SNAME
T	--	Test flag (0=No checking)

EFFECTS:

1. Defines NAME as an indexed switch list containing labels, location variables, closes, or other switch references. Code will be generated to transfer to the I'th location parameter upon the reference "GO TO NAME (I)."
2. If the test flag is non-zero, code is generated to test for a valid switch reference.
3. A location parameter of zero implies no branch is to occur for the corresponding value of the index at invocation time.
4. A new block is created and a path is marked from it to all the blocks associated with the argument labels or closed subroutine names. A switch name argument causes all its associated blocks to be included also. A location variable pointer causes <u>all</u> blocks (including future ones) to be path associated with the new block. The new block is flagged in the Symbol Table entry for NAME.

GENERATED FL:

```
IDXSW NAME[[,L][,C][-1,SNAME,IND]],T
           n
```

FUNCTION: ITMSW   Item Switch Declaration

CALL: *ITMSW(NAME,SIP$[,K,L]_n[,K,C]_n[K,-1,SNAME,IND]_n$,T)

.L

ARGUMENTS: NAME -- Switch name (string, MLV, or pointer)

SIP -- Switch item pointer

K -- The pointer to the comparison constant for a given alternative.

L -- Label or location variable pointer

C -- Closed subroutine name pointer

SNAME -- Pointer to an indexed switch name or item switch name with a scalar item pointer.

IND -- Integer constant or variable pointer indicating the indexed or item switch reference "SNAME(IND)."

T -- Test Flag (0=No checking).

EFFECTS:

1. Defines NAME as an item switch containing labels, location variables, closes, or other item or indexed switch references.
2. The reference "GO TO NAME $[(I1[,IN]_n)]$"

   has the meaning:
   ```
   TEMP = SIP[(I1[,IN]n)]
   IF TEMP EQ K GO TO L
   IF TEMP EQ K GO TO C
   IF TEMP EQ K GO TO SNAME(IND)
        ⋮
   ```
3. A location parameter of zero implies no branch is to occur.
4. If the test flag is non-zero, code is generated to test for a valid reference.
5. A new block is created and is associated with NAME in the Symbol Table. A path is defined from the block to all blocks associated with labels, other switch lists, and closed subroutines. If a location variable is present, a path is marked to <u>all</u> other blocks (future and current).

GENERATED FL:

```
        ITMSW  NAME,SIP[,K,L][,KC][K,-1,SNAME,IND],T
                       n     n    n
```

FUNCTION: GSWCH Switch Transfer

CALL: GSWCH (NAME,[E]$[,E]_{n}$)

ARGUMENTS:	NAME	-- Switch List Name Pointer (Indexed or Item)
	E	-- Polish Expression String[s]

EFFECTS:

1. For index switches this call processes a switch transfer to the switch NAME at offset E. The conversion of the expression E to an integer value is done automatically.
2. For item switches, the expressions E define the index to the switch item pointer. The E are converted to integer values automatically.
3. The Block Dependency Handler is called to reflect a path from the current block to the block identified with NAME.
4. For an index switch the Transfer of Control mode is set if the switch NAME does not contain null branch points, thereby allowing a switch invocation with no transfer followed by a flow to the next statement.
5. For an item switch the expression EE is constructed from the E subscripts to reflect the following transformation:

   GO TO NAME[(E$[,E]_{n}$)] becomes

   GO TO NAME[(item-switch-name(E$[,E]_{n}$)]

GENERATED FL:

INSXFR	NAME[,E]	[Indexed switch transfer]
ITSXFR	NAME[,EE]	[Item switch reference]

[current block → switch block]

FUNCTION: GO(L)

CALL: GO(L)

ARGUMENTS: L -- Label or Location Variable Pointer

EFFECTS:

1. Processes a direct transfer to a label or through a location variable.
2. The Block Dependency Handler is called with argument L if a label.
3. The Transfer of Control mode is set.
4. If L is a location variable a path is constructed from the current to all other blocks.

GENERATED FL:

```
        GO  L
        [current block → .L]
     or
        [current block → [all blocks]]
```

FUNCTION: IXFER Indexed Transfer

CALL: IXFER L$\underset{n}{[}$,L],TEST,E

ARGUMENTS:	E	-- Integer Index Polish Expression
	L	-- Program label pointers
	TEST	-- Test Flag (0=No Test Code)

EFFECT:

1. The integer value of E is used as an index to transfer to the appropriate label L.
2. If TEST NE 0, E will be tested for negative or greater or equal to N, in which case no transfer is made.
3. The Block Dependency Handler is called for each L label.
4. The Transfer of Control mode is set.

GENERATED FL:

IXFER L$\underset{n}{[}$,L],TEST,E

FUNCTION: CXFER   Computed Transfer

CALL: CXFER(I[,L_n][,TEST)

ARGUMENTS: I   -- Location Variable Pointer

L   -- Program label pointers

TEST -- Branch Test Flag (0=No Test)

EFFECT:

1. Implies a transfer to the contents of I.
2. The legal values for I are the L.
3. If TEST=0, an indirect branch is made to I.
4. The Block Dependency Handler is called for each label L.
5. The Transfer of Control mode is set.

GENERATED FL:

```
          CXFER  I[,Ln][,TEST]
```

2.2.6 Conditional Transfer of Control

FUNCTION: IFCOND

CALL: IFCOND(E)

ARGUMENTS: E -- logical or relational expression

EFFECTS:

1. Defines the start of a collection of conditionally executed mode blocks, called a structure, corresponding to an IF-THEN-ELSE-type language construct.
2. The expression E defines the condition under which the initial condition block is to be executed.
3. Subsequent IFALT calls define alternative condition blocks to be executed within the scope of the active structure.
4. The structure is ended by a subsequent SEND call.
5. The Block Identifier is called to define the following blocks:
   a. The structure end (successor) block;
   b. The initial condition block.

   The current (structure predecessor) block and structure end blocks are saved in the Condition Stack.
6. The condition code is generated and the Block Initiator is called to activate the initial condition block.
7. An implied mode block length of one statement is set such that a new statement encountered later causes an automatic BEND to be generated.

GENERATED FL:

```
[.SE]
GEN  E, [.C11],.CONDBRANCH
BEGIN .C11
```

FUNCTION: IFALT

CALL: IFALT(E)

ARGUMENTS: E -- logical or relational expression.

EFFECTS:

1. Defines a conditionally executed block which is to be considered an alternative to the active structure.
2. Corresponds to an ORIF language construct.
3. The expression E defines the condition under which the block is to be executed.
4. The Block Identifier is called to create the condition block.
5. The structure predecessor block (in the Condition Stack) is activated and the condition code is generated.
6. The condition block is then activated.

GENERATED FL:

```
    BEGIN .SP
    GEN E,[.Cli],.CONDBRANCH
    BEGIN .Cli
```

FUNCTION: IFELSE

CALL: IFELSE

EFFECTS:

1. Defines the start of the last alternative block for the active structure.
2. Activates the structure predecessor block, outputs a transfer to the alternative block (call to GO), and then activates the alternative block.

GENERATED FL:

```
BEGIN .SP
GO .SA
BEGIN .SA
```

FUNCTION: BLOCK   Mode Block Identification

CALL: BLOCK

EFFECTS: 1. Identifies the start of a conditional or loop later terminated by a call to BEND or AEND.

2. For conditional mode blocks a call to BLOCK is required to override the implied mode block length of 1 statement.

3. For loop blocks:

   a. For a 'Test before execute loop' [see LOOP function] a branch to the loop test block is generated. The Block Dependency routine is called to reflect a path from the loop predecessor block to the test block.

   b. For a normal loop the Block Dependency routine is called to reflect a path from loop predecessor block to the body block.

   c. The Block Initiator is called to start the loop body block.

GENERATED FL:

```
B      .LTEST [for loop blocks]
START  .LBODY [for loop blocks]
```

FUNCTION: BEND

CALL: BEND

EFFECTS: 1. Identifies the end of a mode block initiated by the most recent BLOCK call.

2. For a loop block end:

   a. The Block Initiator is called to start the loop successor block.

   b. The Block Dependency routine is called to reflect the following block paths:

      current block [last of loop]⟶loop increment block [most recent]

      loop increment block⟶loop increment block [most recent to previous, for this loop]

      ⋮

      loop increment block [outermost]⟶loop test block

      loop test block⟶loop body block

      loop test block⟶loop successor block

3. For a conditional block end:

   a. For the currently active structure, a branch to the structure end (successor) block is generated.

GENERATED FL:

[See above]

FUNCTION: SEND   Structure End

CALL: SEND

```
        SEND
```

EFFECTS: 1. Acts like BEND for all active blocks within this structure.

2. Drops structure end block, thus ending the active structure, by calling the Block Initiator.

3. Deletes an entry from the Condition stack.

GENERATED FL:

```
        BEGIN  .SE
```

FUNCTION: AEND   End All Compound Statements

CALL: AEND

EFFECTS: 1. Closes all open loops and conditionals. Acts like repeated SEND and BEND calls until no more conditionals, loops, or structures are active.

GENERATED FL:

(See BEND and SEND]

Examples:

[FORTRAN]	Semantic Calls	Generated Function Language
IF(A.LT.B)X=X+1	IFCOND(A,B,<)	[.SE]
⋮		GEN A,B,[.C11],.BLT
		BEGIN .C11
	GEN (X,X,1,+,=)	GEN X,X,1,+,=
		[BEND]
	⋮	GO .SE
		[SEND]
		BEGIN .SE
IF(A)GO TO 10	IFCOND(A)	[.SE]
⋮		GEN A,[.C11],.BT
		BEGIN .C11
	GO (.10)	GO .10
		[.C11 $\rightarrow$ .10]
	⋮	[BEND]
		GO .SE
		[SEND]
		BEGIN .SE
[SPL]		[in block α]
IF A THEN B=C	IFCOND(A)	[.SE]
		GEN A,[.C11],.BT
ELSE B=D END		BEGIN .C11
	BLOCK	
	GEN (B,C,=)	GEN B,C,=
	BEND	
		GO .SE
	IFELSE	BEGIN α
		GO [.SA]
		[α $\rightarrow$ .SA]
		BEGIN .SA
	BLOCK	
	GEN (B,D,=)	GEN B,D,=
	BEND	
		GO .SE
	SEND	
		BEGIN .SE

```
                         Semantic Calls              Generated Function Language

IF A GR B THEN Q=C       IFCOND(A,B,>)               [in block ∝]
                                                     [.SE]
                                                     GEN A,[.C11],.BT
                                                     BEGIN .C11
                         BLOCK
                         GEN (Q,C,=)                 GEN Q,C,=
                         BEND
                                                     GO .SE
ORIF Z LS 8 A=B C=Q      IFALT(Z,8,<)                BEGIN  ∝
                                                     GEN Z,8,[.C12],.BLT
                                                     BEGIN .C12
                         BLOCK
                         GEN (A,B,=)                 GEN A,B,=
                         GEN (C,Q,=)                 GEN C,Q,=
                         BEND
                                                     GO .SE
ORIF Z GT 9 GO TO NEXT  IFALT (Z,9,>)                BEGIN  ∝
                                                     GEN Z,9,[.C13],.BGT
                         BLOCK
                         GO (.NEXT)                  GO .NEXT
                         BEND
                                                     GO .SE
ELSE A=C Q=B             IFELSE                      BEGIN  ∝
                                                     GO [.SA]
                                                     BEGIN .SA
                         BLOCK
                         GEN (A,C,=)                 GEN A,C,=
                         GEN (Q,B,=)                 GEN Q,B,=
                         BEND
                                                     GO .SE
END                      SEND
                                                     BEGIN .SE
```

	Semantic Calls	Generated Function Language
IF A EQ B IF D E=F	IFCOND(A,B,.EQ)	[in block α]
		[.SE1]
		GEN A,B,[.C11],.BEQ
		BEGIN .C11
	BLOCK	
	IFCOND(D)	[.SE2]
		GEN D,[.C21],.BT
		BEGIN .C21
	BLOCK	
	GEN (E,F,=)	GEN E,F,=
	AEND	[BEND]
		GO .SE2]
		[SEND]
		BEGIN .SE2
		[BEND]
		GO .SE1
		[SEND]
		BEGIN .SE1
[JOVIAL]		
IFEITH A NQ B $	IFCOND(A,B,≠)	[in block α]
		[.SE1]
		GEN A,B,[.C11],.BNE
		BEGIN .C11
BEGIN	BLOCK	
X=X+1$	GEN(X,X,1,+,=)	GEN X,X,1,+,=
END	BEND	
		GO .SE1
ORIF A-1 GR B $	IFALT(A,1,-,B,>)	BEGIN α
		GEN A,1,-,B,[.C12],.BGT
		BEGIN .C12
C=C-1 $	GEN(C,C,1,-,=)	GEN C,C,1,-,=
		[BEND]
		GO .SE1

```
                     Semantic Calls           Generated Function Language
ORIF A EQ 0 $        IFALT(A,0,.EQ)           BEGIN α
                                              GEN A,0,[.C13],.BE
                                              BEGIN .C13
BEGIN                BLOCK
  C=B+1$             GEN(C,B,1,+,=)           GEN C,B,1,+,=
  GO TO H1$          GO(.H1)                  GO .H1
  END                BEND
                                              GO .SE1
ORIF 1 $             IFELSE                   BEGIN α
                                              GO .SA
                                              BEGIN .SA
B=A$                 GEN (B,A,=)              GEN B,A,=
END                  SEND
                                              BEGIN .SE1
```

2.2.7 LOOP CONTROL

FUNCTION: LOOP Repetitive Loop Definition

CALL: LOOP(T,LV[,E])

ARGUMENTS: T -- Loop Type:

0 = No initial branch to limit code

1 = Implied branch to limit test prior to loop execution

LV -- Loop Induction Variable Pointer

E -- Loop initialization expression for induction variable LV.

EFFECTS:
1. Implies start of a repetitive loop mode block.
2. If T=1, a branch to the corresponding test code is generated prior to the first statement in the loop body.
3. Creates the loop body and successor blocks (.LBi and .LEl) by calling the Block Identifier. The resulting labels and the LV pointer are encoded into the Loop Control stack.
4. The current block label is also saved in the Loop Control stack.

GENERATED FL:

```
GEN  LV,E,=
LOOP T,LV,LB,LE
```

FUNCTION: PLOOP   Parallel Loop Definition

CALL: PLOOP (LV[,E])

.L

ARGUMENTS: LV -- PLOOP Induction Variable Pointer

E -- PLOOP initialization expression for LV

EFFECTS:

1. The start of a parallel loop to the currently active loop is implied.
2. The block preceding the first loop block is reactivated by calling Block Initiator.
3. An entry is made into the Loop Control stack.

GENERATED FL:

```
BEGIN BEFORE [The BLOCK preceding the loop]
GEN LV,E,=
 PLOOP LV
```

FUNCTION: LINCR   Loop or Parallel Loop Increment

CALL: LINCR[(E)]

ARGUMENTS: E -- Loop incrementation expression for active [parallel] loop.

EFFECTS:
1. The incrementation expression for the current loop or parallel loop is defined.
2. The Block Identifier is called to create an increment block.
3. The Block Initiator is called to activate the block and the increment expression is passed.
4. The block label is encoded in the Loop Control stack.

GENERATED FL:

```
BEGIN .LIi [loop increment block]
GEN LV,LV,E,+,=
LINCR LV[LV is the active [parallel] loop variable]
```

FUNCTION: LTEST   Loop Test Definition

CALL: LTEST(T[,E])

ARGUMENTS: T -- Truth Flag:

1 = Repeat loop when true (i.e., WHILE)

0 = Repeat loop when false (i.e., UNTIL)

E -- Relational Truth Expression

EFFECTS:

1. Defines conditions under which the loop is to be repeated (WHILE) or ended (UNTIL).
2. The UNIQUE loop test block for this loop level is created (if non-existent) and initiated by calling the Block Identifier and Block Initiator.
3. The block label is encoded in the Loop Control stack (if not already there).
4. The relational operator is reversed for a WHILE condition so that a false jump is always reflected in the test code, and is then replaced by a corresponding conditional branch instruction.

GENERATED FL:

```
BEGIN .LTi [Loop Test block for this loop level]
GEN E,.LEND,.COND BRANCH [.LEND is the block following the loop]
LTEST  LV[The loop variable for the active[parallel] loop]
```

Examples

[FORTRAN]	Semantic Calls	Generated Function Language
		[in block α]
DO 100 I=1,10,2	LOOP(0,I,1)	GEN I,1,=
:		LOOP 0,I[,.LB1,.LE1]
	LTEST(1,I,10,≤)	BEGIN [.LT1]
		GEN I,10,.LE1,.BGT
		LTEST I
	LINCR(2)	BEGIN [.LI1]
		GEN I,I,2,+,=
		LINCR I
	BLOCK	BEGIN .LB1
		[α→ .LB1]
100 CONTINUE	STAT(.100)	BEGIN .100
		[.LB1 →.100]
	BEND	BEGIN .LE1
		[.100 →.LI1]
		[.LI1 →.LT1]
		[.LT1 →.LB1]
		[.LT1 →.LE1]
[FORTRAN]		[in block α]
DO 200 I=1,100	LOOP(0,I,1)	GEN I,1,=
		LOOP 0,I[,.LB1,.LE1]
	LTEST(1,I,100,≤)	BEGIN [.LT1]
		GEN I,100,.LE1,.BGT
		LTEST I
	LINCR (1)	BEGIN [.LI1]
		GEN I,I,1,+,=
		LINCR I
	BLOCK	BEGIN .LB1
		[α→ .LB1]

```
                         Semantic Calls              Generated Function Language

DO 300 J=L,M             LOOP(0,J,L)                 GEN J,L,=
                                                     LOOP 0,J[,.LB2,.LE2]
                         LTEST(1,J,M,≤)              BEGIN [.LT2]
                                                     GEN J,M,.LE2,.BGT
                                                     LTEST J
                         LINCR(1)                    BEGIN [.LI2]
                                                     GEN J,J,1,+,=
                                                     LINCR I
    .                    BLOCK                       BEGIN .LB2
    .                                                [.LB1 → .LB2]
    .
300 CONTINUE             STAT(.300)                  BEGIN .300
                                                     [.LB2 → .300]
                         BEND                        BEGIN .LE2
                                                     [.300 → .LI2]
                                                     [.LI2 → .LT2]
                                                     [.LT2 → .LB2]
                                                     [.LT2 → .LE2]
200 CONTINUE             STAT(.200)                  BEGIN .200
                                                     [.LE2 → .200]
                         BEND                        BEGIN .LE1
                                                     [.200 → .LI1]
                                                     [.LI1 → .LT1]
                                                     [.LT1 → .LB1]
                                                     [.LT1 → .LE1]
[SPL]                                                [in block [illegible]]
FOR C=0 BY 2 UNTIL 101   LOOP(1,C,0)                 GEN C,1,=
                                                     LOOP 1,C[,.LB1,.LE1]
                         LINCR(2)                    BEGIN [.LI11]
                                                     GEN C,C,2,+,=
                                                     LINCR C
```

```
                        Semantic Calls              Generated Function Language

                        LTEST(0,C,101,.EQ.)         BEGIN [.LT1]
                                                    GEN C,101,.LE1,.EQ.
                                                    LTEST C
ALSO D=50 BY 1          PLOOP(D,50)                 BEGIN
                                                    GEN D,50,=
                                                    PLOOP D
                        LINCR(1)                    BEGIN [.LI12]
                                                    GEN D,D,1,+,=
                                                    LINCR D
ALSO E=5                PLOOP (E,5)                 BEGIN
                                                    GEN E,5,=
                                                    PLOOP E
                        BLOCK                       BEGIN
                                                    GO .LT1
                                                    [→ .LT1]
                                                    BEGIN .LB1
FOR F=100 BY -2 WHILE F GR 0
                        LOOP(1,F,100)               GEN F,100,=
                                                    LOOP 1,F[,.LB2,.LE2]
                        LINCR(-2)                   BEGIN [.LI21]
                                                    GEN F,F,-2,+,=
                                                    LINCR F
                        LTEST(1,F,0,>)              BEGIN [.LT2]
                                                    GEN F,0,.LE2,.BLE.
                                                    LTEST F
ALSO G=7 BY L UNTIL 91
                        PLOOP (G,7)                 BEGIN .LB1
                                                    GEN G,7,=
                                                    PLOOP G
                        LINCR(1)                    BEGIN [.LI22]
                                                    GEN G,G,1,+,=
                                                    LINCR G
                        LTEST(0,G,91.,EQ.)          BEGIN [.LT2]
                                                    GEN G,91,.LE2,.EQ.
                                                    LTEST G
```

	Semantic Calls	Generated Function Language
	BLOCK	BEGIN .LB1
		GO .LT2
		[.LB1 → .LT2]
		BEGIN .LB2
TEST(C)	TEST(C)	GO .LI11
		[.LB2 → .LI11]
		[BEGIN .LB21]
TEST	TEST	GO .LI22
		[.LB21 → .LI22]
		[BEGIN .LB22]
END	BEND	BEGIN .LE2
		[.LB22 → .LI22]
		[.LI22 → .LI21]
		[.LI21 → .LT2]
		[.LT2 → .LB2]
		[.LT2 → .LE2]
END	BEND	BEGIN .LE1
		[.LE2 → .LI12]
		[.LI12 → .LI11]
		[.LI11 → .LT1]
		[.LT1 → .LB1]
		[.LT1 → .LE1]

2.3 Internal Data Structures

Several control arrays are present within the Source Processor for the purpose of driving the parsing and semantic processing of the compiler source language. These arrays are typically allocated and initialized by the Meta-Compiler as a function of the declarative information supplied by the compiler writer in Meta-Language, such as lexical definition, language declaratives, encountered literals, and hash table, symbol table, and stack declarations.

The format and use of each control array is described below, with the information encoded within host words in symbolic format (the exact bit positions are to be determined at implementation time). Many of the arrays have an exact correspondence with a stack or array in the Meta-Compiler used to initialize the array in the BLOCK DATA initializer of the Source Processor.

Literal Vector - QLVECT

```
Format:  word 1:    n
         word 2:    C₁
           .
           .
         word n+1:  Cn
           .         .
           .         .
           .         .
```

n -- the number of characters in the literal.

C_i -- the i'th literal character in internal format.

Use: This array is used to store all literal operands in the Meta-Language compiler definition which are used as search operands. The array drives the Literal Scan syntax support submodule (QLITSC).

MLV Association Arrays - QSTART, QSIZE

Format:	QSTART	QSIZE
word i:	startpos	length

startpos -- starting position in the input image buffer (IMAGE) of the construct associated with mlv i.

length -- the corresponding length, in characters, of the mlv i.

Use: These two arrays identify the position and length of the language construct satisfying the most recent application of each mlv. They are reset by the Step-Up/Down support submodule when a true step-up is performed.

Terminal Driver - QTERMD

Format: (see stack LTERMD, section 1).

Use: This array contains the encoded information enabling the Terminal Detection support submodule to find terminal constructs in the input image buffer (IMAGE).

Argument Control - QPARGS

Format: (see stack QPARG, section 1).

Use: This array contains the argument descriptions and values associated with deferred and immediate semantic function calls. This information is utilized by the Argument Stacking support submodule in moving the arguments for a particular semantic call onto the control stack QCTAB.

Prescan Driver - QPRES

Format: (see stack NTABLE, section 1).

Use: This array controls the prescanning activity implied by PRESCAN Meta-Language statements. The process takes place in the Source Input module at the time a card image is appended to the current compiler language statement.

Illegal Mode Combinations - QIMODC

Format: (see stack IMODE, section 1).

Use: This array specifies which combinations of operands and operators are flagged as illegal by the Operator Language semantic submodule.

Field Specifiers - QFIELD

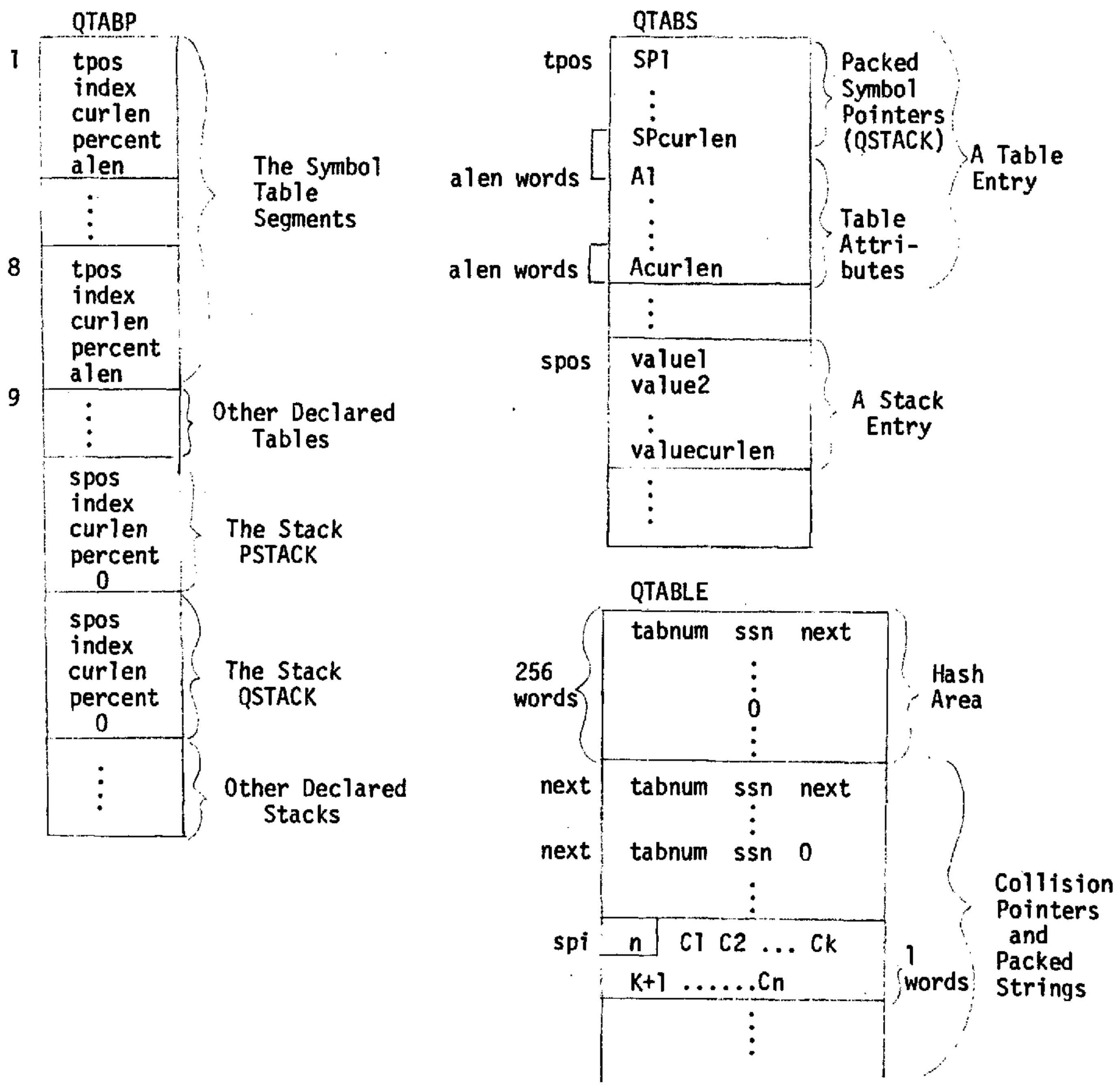
Format: (see stack LFIELD, section 1).

```
    word i:   startcol
    word i+1: endcol
    startcol -- starting column for field i.
    endcol --   ending column for field i.
```

Use: This array defines the field boundaries, if any, for language statements. The field information is used by the Source Input module to maintain cursor movement from a field to the next field.

Dynamic Table/Stack Control - QTABP, QTABS, QTABLE

Formats:



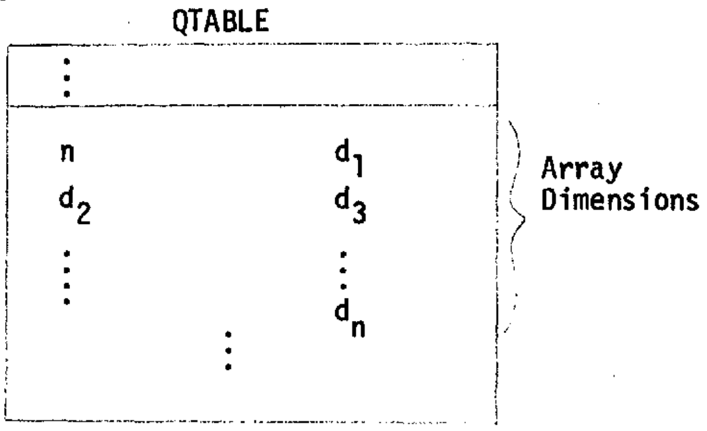
tpos --	dynamic QTABS starting position for a table.
index --	running stack or table pointer (declared).
curlen --	current stack/table length, in entries.
percent --	the declared percentage of the total QTABS area (dynamic storage) to allocate to the stack or table if it overflows.
alen --	number of attribute words/entry for a table (0 implies no attributes).
spos --	dynamic QTABS starting position for a stack.
Sp_i --	QSTACK relative pointer to the packed symbol representing the i'th table entry, i.e., symbol sequence number i.
A_i --	the packed attributes, alen words long, for the i'th table entry.
$value_i$ --	the i'th value contained in a stack.
tabnum, ssn --	the table number and symbol sequence number uniquely identifying a symbol within an arbitrary table.
next --	the thread pointer (QTABLE relative) to the next triplet (tabnum,ssn,next) identifying another symbol which hashes into the same slot within the hash area. If zero, no others remain on the thread. If an entire triplet at position i in the hash area is zero, then no symbols having a hash value of i have been placed in any table.
n --	the number of characters in a compressed symbol string (8 bits).
C_i --	the i'th string character in internal form, 6 bits. Thus, up to 63 language characters are allowed.
ℓ --	the number of host words required to represent a symbol of length n,

i.e., $$\ell = \frac{7+D*6}{qwdsZ} + 1$$

qwdsZ -- the host word size.

Use: These three arrays contain all information required to maintain dynamic symbol tables, hash tables, and stacks. Their entries are maintained by the table (QSTACK) and stack (QSTKSR, QSTKM) manipulation support submodules.

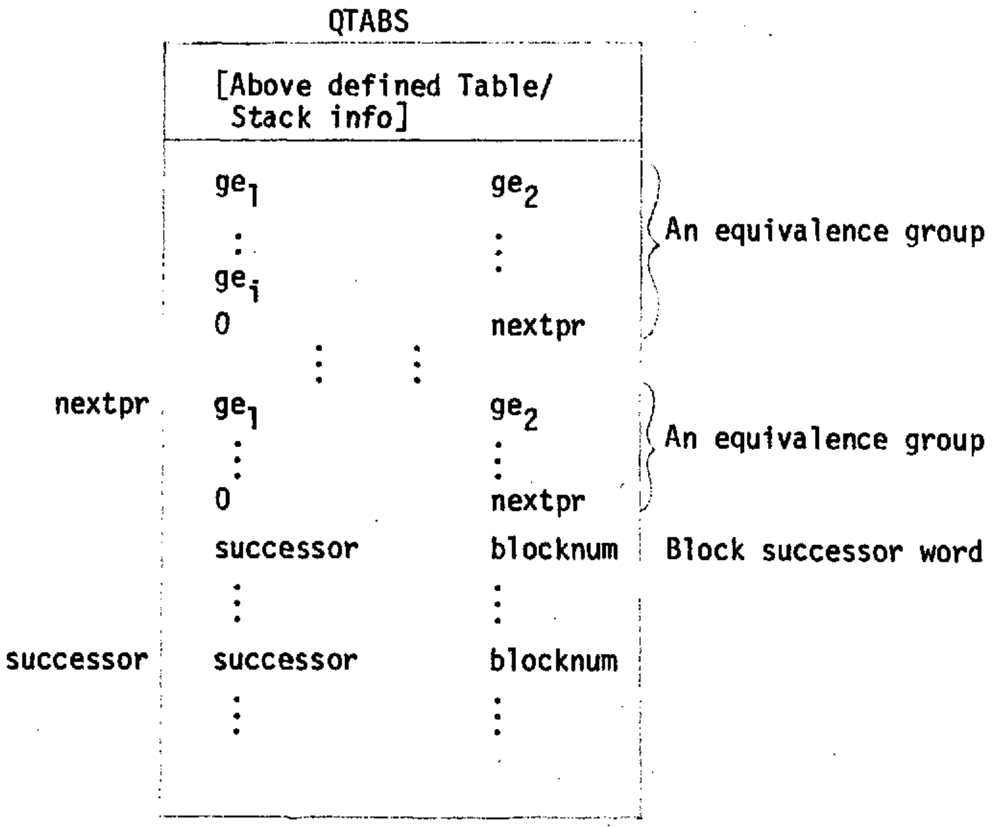
The array QSTACK is also utilized to save array dimension descriptor information as follows:



n -- number of dimensions declared for the array.

di -- symbol pointer to a constant or variable dimension.

The array QTABS is also utilized to save equivalence groups derived from the EQUATE function as well as successor blocks for the array QBIP (Block ID Table):



ge_i -- pointer to an equivalence group element (see EQUATE function).

nextpr -- offset to the next saved equivalence group (0 implies no more groups).

successor -- offset to the next successor identifier word (0 implies no more).

blocknum -- successor block number.

Statement Image - IMAGE

Format: word i: character i

character i -- the i'th character, in internal format, of the current statement being parsed.

Use: All statement parsers and syntax support modules manipulate language characters within the current statement, which is always contained within the IMAGE buffer, one character per word, right adjusted, in internal code. The internal codes for each meta-character are to be determined at implementation time.

Debug Control - QDEBUG

Format: word i: option i

option i -- 1 if the i'th option is selected, 0 if not.

Use: Used by the Error Report submodule to monitor the Source Processor execution as indicated by each selection option (see Volume I, DEBUG declaration).

Mode Flags - QLMODE

Format: word i: mode i

mode i -- 1 if computational mode i is legal within expressions of the compiler language, 0 if not.

Use: Referenced by the Expression Handler semantic support submodule for operand/operator mode analysis.

Operator Flags - QLOPER

Format: word i: hierarchy

word i+1: commutivity

word i+2: associativity

hierarchy -- the declared or default operator hierarchy.

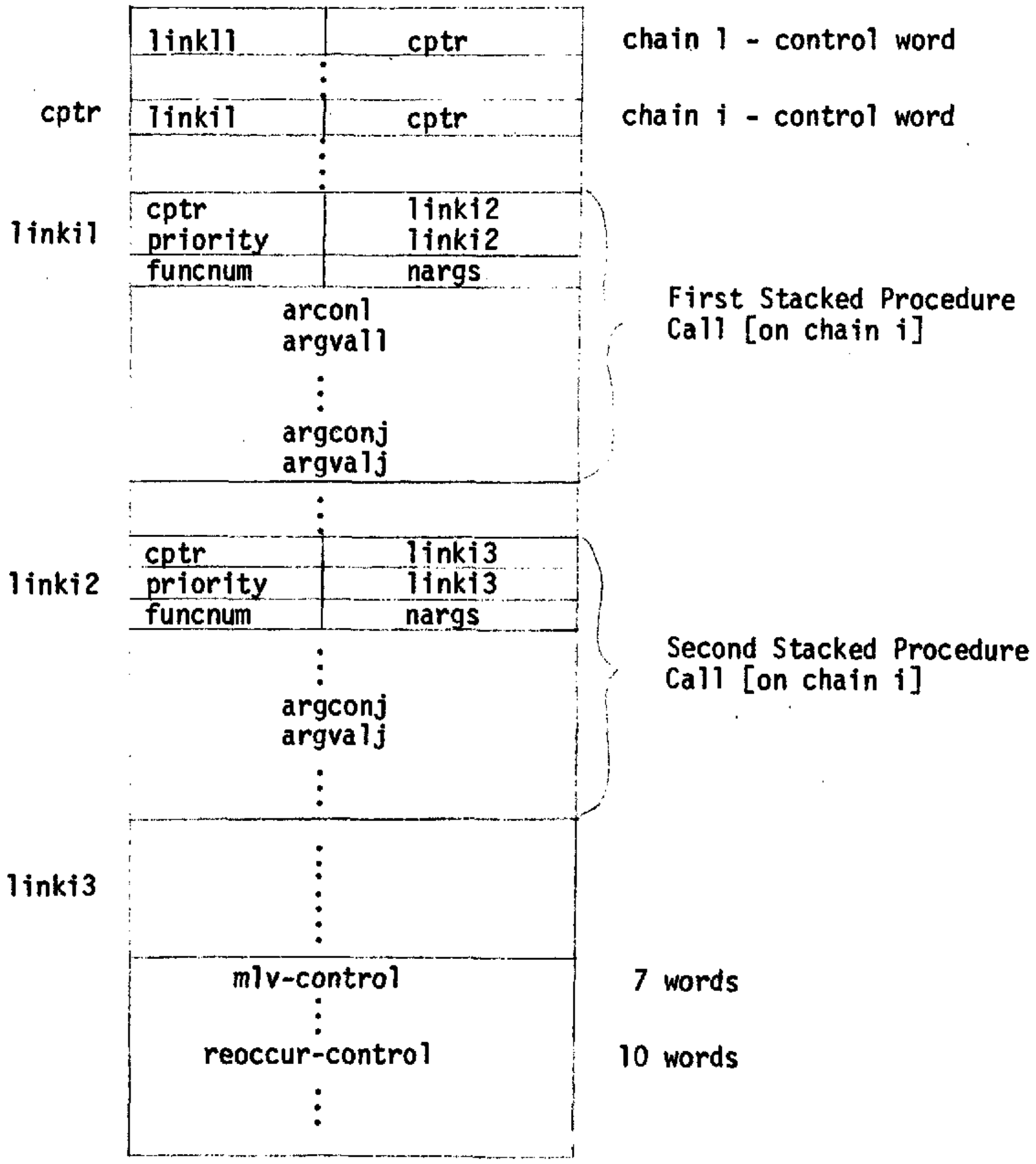
commutivity -- 1 if the operator is commutative, zero if not.

associativity -- 1 if the operator is associative, zero if not.

Use: Used by the Expression Handler semantic support submodule in forming a polish string representation of a language expression.

Execution Control - QCTAB

Format:



cptr --	pointer to the control word for the next chain (if zero, no more chains).
linkik --	pointer to the next deferred procedure call stacked on a chain, starting at linki1 which is specified in the chain control word. Zero implies this is the last call on the chain.
priority --	the priority level for the stacked call.

funcnum --	the function number of the semantic procedure being called.
nargs --	the number of argument words following.
argconj --	argument control word for argument j (see below).
argvalj --	argument value word[s] for argument j (see below).
mlv-control --	control information saved when step downs to an mlv occur.
reoccur-control --	control information saved when a reoccurrence element is initiated.

Use: This array is used as a dynamic control stack containing the following:

- All deferred procedure calls and their arguments;
- The control information for active step downs into mlvs and execution of reoccurrence elements.

The deferred call information is created by the Argument Stacking submodule (QARG10) when a deferred semantic request is made. The Deferred Execution module subsequently performs the semantic calls still active (i.e., "true") by chain and priority within a chain.

The control information is utilized by the Step Up/Down (QSTEP) and Reoccurrence Maintenance (QREOCB, QREOCE) support submodules. The information (CURSOR, etc...) is saved upon a step down into an mlv or reoccurrence, and restored upon stepping back up, at which time the QCTAB control cells are released.

The following gives the correspondence between argument descriptions and values as encoded within the QPARGS descriptor array and the argument descriptor and value words placed on QCTAB by the Argument Stacking submodule (refer to QPARGS format):

Argument Type	Corresponding QCTAB Argument Descriptor	QCTAB Value Word[s]
-1 to -5	not applicable	----
-6	-6	n stkvalue_1 : stkvalue_n
-7	-7	symbol-pointer
-9	-9	integer value
-10	-10	litptr
-11	-11	QSTART (mlvnum) QSIZE (mlvnum)

n --	the number of stkvalue cells following.
stkvalue_i --	the contents of i'th stack entry at this time for the stack referenced by the ELEMENTS OF argument.

Condition Stack - ICOND

Format:

word i: structure-end, structure-predecessor

structure-end -- a label identifying the successor block to the corresponding structure.

structure-predecessor -- a label identifying the predecessor block to the corresponding structure.

Loop Control Stack - LOOPC, LOOPCI

```
Format:                 LOOPC                        LOOPCI

                      ( n                        n:   m
        Active        ( loop-predecessor            ( loop-induction-variable
         loop         ( loop-successor              ( loop-increment
       (level i)      ( loop-body            2*m    ( loop-induction-variable
                      ( loop-test                   ( loop-increment
        Active        ( n                           (   :
         loop         (  :                          (   .
      (level i+1)     (  :
                                                        :
                                                        :
```

n -- the starting position in LOOPCI for the primary and secondary loop induction variable information.

m -- the number of loop variables at the current level.

loop-predecessor -- block label pointer for the loop predecessor block.

loop-successor -- block label pointer for the loop successor block.

loop-body -- block label pointer for the loop body block.

loop-test -- block label pointer for the loop test code following the loop body block (0 implies no test code).

loop-induction-variable -- a secondary or primary loop induction variable. The first entry is the primary variable.

loop-increment -- block label pointer for the corresponding increment code for the variable (0 implies no increment).

Use: These parallel stacks are used to save information about all currently active program loops. An entry is made on LOOPC and LOOPCI for each encountered LOOP call, and an entry is deleted upon executing the matching BEND call. A loop nest of K levels results in K LOOPC and LOOPCI entries to be made.

Block Identification Table - QBID

Format:

word i: successor-ptr, label   [for block i]

successor-ptr -- offset for word in table QTABS identifying the immediate successors to block i (see QTABS format).

label -- pointer to the label associated with the block.

Use: This array is used to identify a unique program block as well as its associated label and all of its immediate successor blocks.

Polish Expression Stack - PSTACK

```
Format:     operand-ptr
              :
            operator
              :
            operand-ptr
              :
            operator
              :
```

operand-ptr -- the representation of an expression operand as a symbol pointer of the form:

symbol-type, sequence-number

symbol-type -- the type of symbolic operand, which is the segment number to which it belongs in the Symbol table.

sequence-number -- the symbol sequence number within the Symbol table segment.

operator -- a numeric representation for an expression operand.

An array, table, table item, or procedure reference has the following special representation on PSTACK:

name-ptr	[array,table,table item,or procedure pointer]
'begin-list'	[a special operator defining a parameter list]
P_1	[parameter list of operands or subexpressions]
:	
P_N	
'end-list'	[a special operator ending a parameter list]

Use: This stack contains the reversed Polish representation of the expression being parsed. As expression parsing proceeds, operands and operators are continually added in the proper order until the entire expression is finally represented.

Polish Temporary Stack - QSTACK

Format:			
	word 1:	mode 1	operator 1
	word 2:	mode 2	operator 2
		⋮	⋮
	word n:	mode n	
	word m:		operator m

mode i -- the computational mode of the i'th operand stored on PSTACK.

operator i -- the i'th stacked operator temporarily saved until placed on PSTACK on a priority basis. A zero operator marks the start of a subexpression.

Use: This temporary stack actually contains two information stacks at the same time. The upper part of each word contains the saved computational modes for operands contained on the Polish Stack (PSTACK). The lower part of each word contains expression operators which are held temporarily because of hierarchy considerations.

3. FUNCTION PROCESSOR

The design of the Function Processor is presented in this section. The major modules and submodules are identified, and the supporting internal data structures are specified.

The module is in effect a combination of a universal, language independent and machine independent compiler with major portions of the Meta-Assembler. The compilation algorithms are imbedded in the local and global optimizers. The code generative functions are initiated by the PROC expander, which utilizes the code constructor portions of the Meta-Assembler.

3.1 Program Logic Modules

The Function Processor is organized as a collection of distinct FORTRAN coded modules performing the translation of function language into generated code. The modules are organized into three basic groups: FL term processing, global optimization, and code generation.

The FL term processing modules include FL term decoding and all local optimization routines. This overlay has the general function of transforming an FL program representation into a lower level n-tuple representation on the Code Table.

The global optimization overlay modules modify and permute the Code Table n-tuples within loop regions to produce a more efficient program representation.

The code generation overlay modules produce the actual target code from the Code Table n-tuple entries through PROC expansions.

Figure 3, the Function Processor block diagram, illustrates the interrelationships between the major modules and the internal and external data.

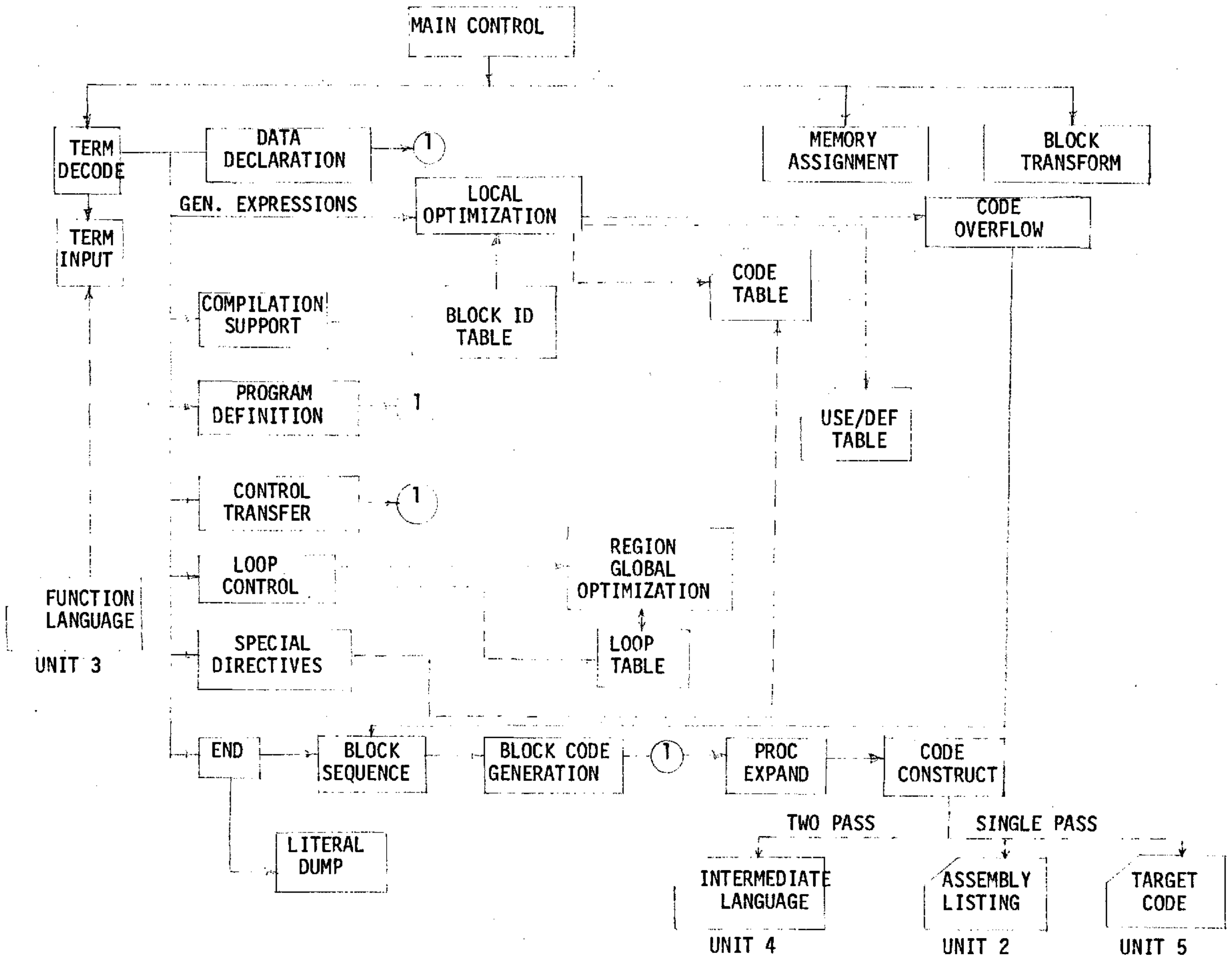


FIGURE 3. FUNCTION PROCESSOR BLOCK DIAGRAM

MODULE: Main Control

FUNCTION: Coordinates and controls the execution of all other logic modules of the Function Processor.

PROCESS:

The Main Control module initiates execution of the Function Processor and retains control at all levels. It performs the phased execution of a collection of overlay modules. Each module has a separate function relating to one of the following tasks: function term decoding and processing, local optimization, global optimization, or code generation.

The Block Transfer module is called to transfer the contents of the Block ID Table in preparation for inter-block code generation.

The Memory Assignment module is then executed to handle the assignment of addresses to all data structures via offsets to the various types of data control sections. The symbol table is transformed to a slightly different format.

The Term Decode module is then repetitively executed to process each input function term. Upon encountering the 'end' FL term, the Block Sequence module is called to generate code for any remaining blocks still active on the Code Table. Control is then returned to the overall compiler control (in the main program of the Source Processor) to terminate compiler execution (single pass mode) or to initiate the Operation Processor (two pass mode).

MODULE: Block Transform

FUNCTION: Restructure the Block ID table.

PROCESS:

The Block Transform module transforms the Block ID Table created during the execution of the Source Processor. The connectivity matrix defining all immediate paths between blocks is formed from the successor links formed by the Source Processor in tables QTABS and QBID.

MODULE: Memory Assignment

FUNCTION: Assigns an offset to each program data structure relative to the various data control sections.

PROCESS:

This module performs a scan through the Symbol Table, assigning target memory addresses relative to data control sections to all simple variables, arrays, tables, and table elements.

The first task performed is to assign all global data blocks. The global block names segment of the symbol table is scanned, with the following actions:

(1) The block name link points to the first symbol. That symbol is assigned address zero, the Item Length submodule is called to compute the length (in target addressing units), and the block name ISIZE attribute (the data section location counter) is updated accordingly.

(2) The symbol link is followed and if more symbols occur, they are assigned the current ISIZE value, and ISIZE is again updated.

At the end of the process, all global blocks will have been assigned with their computed lengths in their ISIZE attributes. The link pointers for each data symbol are reset to point to the data block name rather than the next name in the data list.

The next task is that of scanning the table names segment of the Symbol Table to resolve all TABLE declarations into equivalence chains:

(1) The table name link points to the first declared table item, which points to next item, etc... This chain had been previously constructed by the TABLE function of the Source Processor.

(2) For a programmer allocated table (one in which the word and bit allocations are declared), each table item is assigned an address field indicating the offset from the start of the table in addressing units. These values are computed from the declared word positions and the ratio of the target accessing and addressing sizes.

(3) For a compiler allocated table, a scan of all table items must be made to determine the optimal packing with target words depending on the declared packing density of the table and each item within. The result for each table item is a word and bit position relative to the start of the table as in case (2). The words per entry value is computed for the table.

(4) The result is an equivalence chain linking the table items and table name with each item having been assigned a relative offset to the table.

The next step is to scan the stored equivalence group data in array QTABS corresponding to EQUATE directives. The link attribute of each symbol in each equivalence group is used to form a circular threaded list for <u>each</u> item within a group as follows:

(1) Set OFFSET=0.

(2) Set Z = OFFSET + position of the item (0 for variables, computed for arrays from the subscripts and array declaration info).

(3) If the symbol link for the item is zero (i.e., the symbol is not part of another group or in a global data block or is not a table item), it is linked to the group. The symbol address field is set to Z.

(4) If the item is in a global block:

    (a) Scan all previously linked items in the group, replacing their links with the block name pointer, assigning them addresses within the block based on the difference between their address field, Z, and the assigned address of this symbol.

    (b) Scan the remaining group items (in QTABS), assigning each item an address in the data block as for case (a).

    (c) Update the data block length (ISIZE) if the length of an item causes an increase in the block size.

    (d) The relocatability attribute of each item is marked as global.

(5) If the item is already in an equivalence group, the two chains are merged:

   (a) Recalculate the offsets of the chain being built to be relative to those of the old chain:

   ```
   D = Z - assigned item address
   OFFSET = OFFSET - D
   etc...
   ```

   (b) Merge the two chains through any link. The order of the items in the composite chain (group) is immaterial.

(6) If the item is a table item, its link attribute defines the previously constructed chain derived from a TABLE declaration. The chain is merged into the current group as in case (5).

(7) Step (2) is repeated for the next group item. When all items have been processed the equivalence temporary storage area in QTABP is released.

The final task performed is that of scanning the variable and array Symbol Table segments to assign all symbols an offset relative to the Data Control Section. All symbols which are relative to a globally defined data block will have already been assigned:

(1) The symbols with the largest unit size (per symbol entry) are to be assigned first. A compiler variable had been set to the maximum unit size encountered in the language. After assigning all items with that unit size (steps 2-3), the next smaller unit size is assigned, etc... The data location counter is bounded for each unit size prior to assignment.

(2) For each symbol not having an equivalence link, the address field is set to next value of the data location counter (L), which is then incremented by the length of the symbol (determined by the Item Length submodule).

(3) For an equivalenced item:

    (a) The chain is scanned to find the minimum assigned offset (=K).

    (b) Each entry is assigned the address

        L - K + the entry offset

        The location counter L is then set to the maximum of itself and the assigned offset plus the item length.

    (c) The equivalence links are maintained for the purpose of providing information to the Local Optimizer module.

MODULE: Data Declaration

FUNCTION: Processes a declarative FL term.

PROCESS:

For a DATA function language term, a sequence of ORG and DATAC Operation Language PROCs are expanded by calling the PROC Expand module. The offset to each data symbol is computed from the supplied subscript pointers (if any), the item size, and the target accessing and addressing units.

For a assign data (ADATA) FL term, the ORG OL PROC is expanded with the label as an argument (the label is first assigned the next Data Location Counter value, which is then incremented by the constant or character string length). The DATAC PROC is then expanded with the data constant (or string) as an argument.

In either case, an ORG(0) expansion is done to restore the program location counter.

EXPANDED OL:

```
DATA V[,I],O,M,C[,M,C]:                  ADATA label, datap:
      n        n

     ORG (V+k)                           ORG (label)
     DATAC (PREC,M,C)                    DATAC (PREC,1,datap)
       :
       :
     ORG (0)                             ORG (0)
```

MODULE: Program Definition

FUNCTION: Processes all FL terms in the Program Definition FL category.

PROCESS:

The indicated action is performed upon encountering the following FL terms:

```
  o  ┌MAININ     name,rec,ren
     │ESUBIN
     {ISUBIN
     │EENTIN
     └IENTIN
```

The PUT submodule is called directly to place the matching OL term into block 1, the prolog block (which is active due to a preceding BEGIN 1).

```
  o  {EXDUMMY   i,Pi,Ri
     {INDUMMY
```

The PUT submodule is called directly to place the matching OL term into the block 1, the prolog block.

- CLOSP name

The OL operation 'CLOSP name' is placed in the current block (PUT).

```
  o  RETURN
     EXRETDUM   i,p
     INRETDUM   i,p
     MAINEX     n,rec,ren
     ESUBEX     n,rec,ren
     ISUBEX     n,rec,ren
     CLOSEX
```

The PUT submodule is called directly to place the matching OL term into the current block.

MODULE: Compilation Support

FUNCTION: Performs support functions implied by the compilation support category of FL terms.

PROCESS:

The indication action is performed upon encountering the following FL terms:

- STAT (statement identification)

  [no action]

- LDEF label $[,\text{label}]_n$ (label definition)

  [no action]

- BEGIN label (block initiation)

  The current block is terminated by entering a 'block transfer to 0' code entry on the Code Table.

  The block associated with label is then activated for all subsequent code up to the next BEGIN. The Block ID table entry for the block is modified as follows:

  (a) If the begin value is zero, it is set to the next available code table position.

  (b) If the begin value is non-zero the block is already present on the Code Table. If the block end value points to a 'block transfer to 0' entry, the end value is replaced by a 'block transfer to i', where i is the current code table length. The Set-Up module is then called to complete the block activation.

- MODIFY name $[,\text{name}]_n$

  The Use/Def table is used to mark the modification of the names within the active block.

MODULE: Block Set-Up

FUNCTION: Activates a block of code on the Code Table for insertion of additional operations.

PROCESS:

This module is called whenever new code is to be added to a previously formed block of code on the Code Table. The Block ID Table defines the block start and end positions. A single scan of all operators in the block is performed and the following actions are taken:

1. The Definition Position Table is created to identify the position of all operations defining n-tuple results which are n-tuple pointers*.
2. The Fold Table is updated for instructions setting variables to constants*.

* - Refer to the Local Optimizer, PUT submodule, step 3.

MODULE: Program Transfer

FUNCTION: Processes the Program Transfer of Control group of FL terms.

PROCESS:

The terms in this FL group generally cause a transfer of control to be performed from the current block to other blocks. The action indicated below takes place for each FL term:

```
∘ IDXSW  NAME[,-1,S,I] [,L] [,c],T
             n          n   n
```

The block associated with the label attribute of NAME is assigned the current location counter. The following OL PROCS are then expanded:

```
RES (NAME,0)            assigns NAME the current location counter
SWENT (T,NAME)          switch entry code, with possible test code
B  (L)                  for a label argument
:
SWEXIT (NAME)           for a null argument
:
:
BI (L)                  location variable, BI exists as a PROC
B (label)               for CLOSE calls, other switch references
:
RES (label,0)
CCALL (C)               close call to C
SWEXIT (NAME)
RES (label,0)
INSCALL (S,I)           indexed switch call
SWEXIT (NAME)
RES (label,0)
ITSCALL (S,SIP+I-bias) item switch call
SWEXIT (NAME)
:
```

The following transfer of control FL terms are entered onto the Code Table (via PUT) as delayed OL operations:

FL		Corresponding OL
INSXFR NAME,E		[Expression E→Local Optimizer] INSCALL (NAME,eptr*)
ITSXFR NAME,E		[Expression E→Local Optimizer] ITSCALL (NAME,eptr)
GO L		B (L) for a label
	or	
		BI (L) for a location variable
IXFER L$[,L]_n$,TEST,E		[Expression E→Local Optimizer] IXFER L$[,L]_n$,TEST,eptr
CXFER I$[,L]_n$,TEST		BI I [if no test]
	or	
		CXFER I $[,L]_n$

* - The pointer eptr is returned by the Local Optimizer as an operand pointer representing the expression result.

- ITMSW NAME,SIP$[,K,L]_n$ $[K,C]_n$ $[K,-1,S,I]_n$,T

  The block associated with the label attribute of NAME is assigned the current location counter. The following OL PROCs are then expanded:

```
RES (NAME,))                  assigns name the current location counter
CB  EQ,ACC,K,L                for a label argument
:
CB  EQ,ACC,K,L,,,INDIRECT     compare and branch indirect through
:                             location variable
CB  EQ,ACC,K,label            CLOSE, indexed or item switch call
:
SWEXIT (NAME)
:
RES (label,0)                 CLOSE call
CCALL (C)
SWEXIT (NAME)
:
RES (label,0)                 indexed switch call
INSCALL (S,I)
SWEXIT (NAME)
:
RES (label,0)                 item switch call
ITSCALL (S,SIP+I-bias)
SWEXIT (NAME)
:
```

MODULE: Local Optimization

FUNCTION: Incorporates an expression into the currently active block.

PROCESS:

A Polish expression stack is assumed to be the next entry in the FL input stream. The Polish entries are scanned and resolved into binary operations which are passed to the Binary Operator module to be placed onto the Code Table, identified with the currently active program block. The detailed logic is shown in Figure 3-1.

The temporary operand stack shown actually overlaps the input Polish vector itself, starting in the leftmost position. Stacked temporary operands have the same format as a Code Table operand (see 3.2). A result operand pointer is returned identifying the expression result.

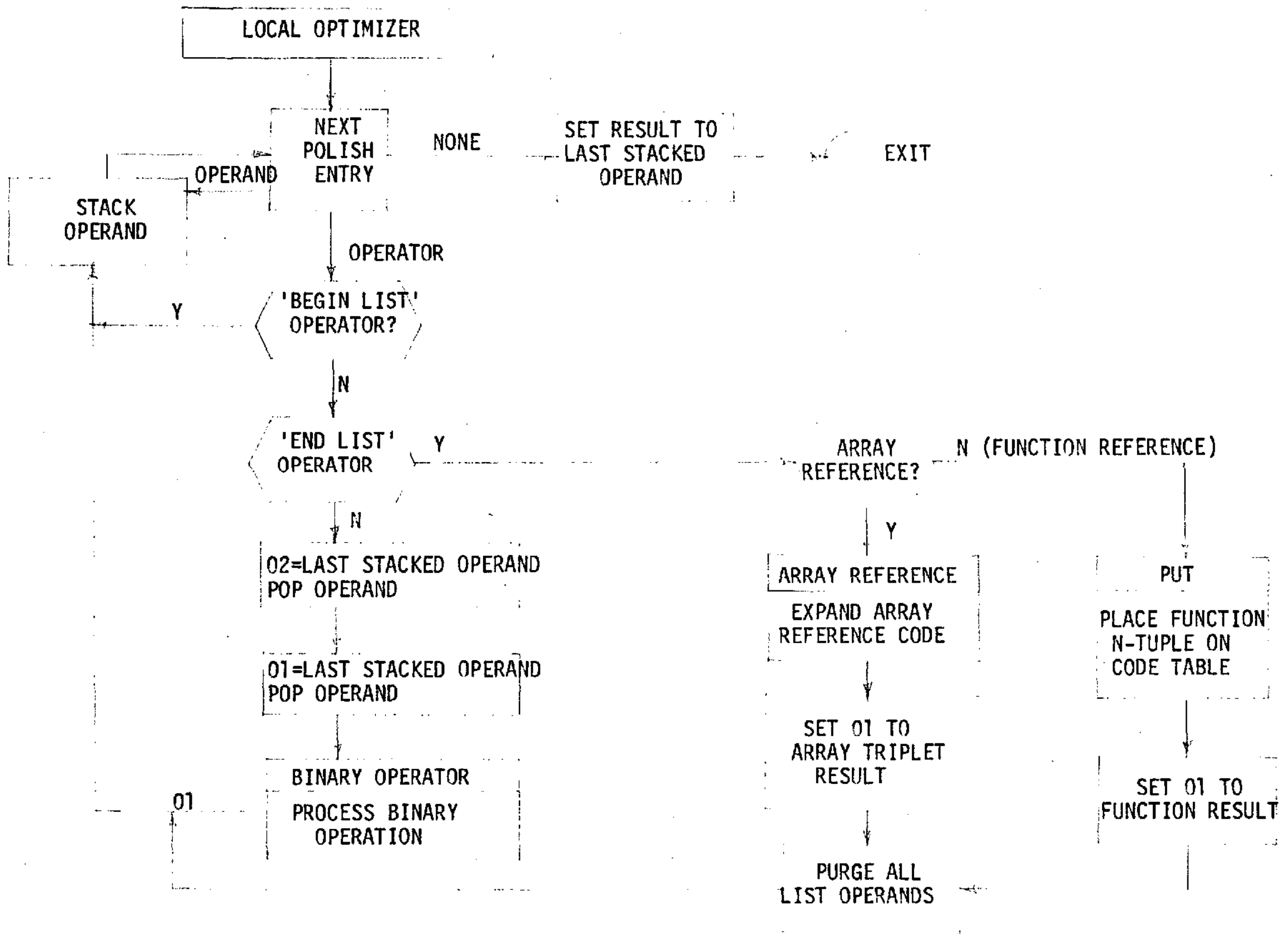


FIGURE 3-1. LOCAL OPTIMIZER LOGIC FLOW

MODULE: Array Reference

FUNCTION: Generates code for an array or table item reference.

PROCESS:

Code is generated for an array reference from the array name and subscript references in the temporary operand stack (see Figure 3-1). The result pointer for the generated code along with the array base and computed constant offset are placed onto the Code Table as a triplet entry (see 3.2, Code Table Format).

Given an array or table item reference:
A $(S_1,\ldots,S_n)$, where A is dimensioned as A $(d_1,\ldots,d_n)$, the following algorithm is applied to generate subscript code:

```
1. Set W = the number of addressing units per element of A;
       K = 0 [the computed constant offset].

2. If all $d_i$ are constants:

       a. If bias $\neq$ 0 [the declared array bias value, 0 or 1],
          $K = -W*[(\ldots(bias*d_{n-1}+bias)*d_{n-3}+\ldots)*d_1+bias]$

       b. Call Binary Operator repeatedly:

              $S_n*d_{n-1}$
              $\theta 1+S_{n-1}$
              $\theta 1*d_{n-2}$
              $\theta 1+S_{n-2}$
                 ⋮
              $\theta 1+S_1$

          If any $S_i$ is a constant, skip the above two computations
          involving $S_i$ and set

              $K = K+S_i*d_1*d_2..*d_{i-1}$
```

3. If some or all d_i are variables:

Call binary operator repeatedly:

```
$S_n$-bias
$\theta1*d_{n-1}$
$\theta1+S_{n-1}$
$\theta1$-bias
$\theta1*d_{n-2}$
  ⋮
$\theta1+S_1$
$\theta1$-bias
```

4. Place a subscript triplet (A, $\theta1$, K) onto the code table.

MODULE: Binary Operator

FUNCTION: Places an optimized binary operator onto the Code Table.

PROCESS:

The combination (θP,θ1,θ2) is examined relative to other operations encoded on the Code Table by a succession of optimizing submodules. The resulting operand pointer for the operation is returned as θ1. Figure 3-2 gives the overall logic flow.

Each operand has a computational mode and negation flag associated with it. For a symbol pointer, the symbol mode is fetched from the attributes. For an n-tuple result operand, the computational mode is set as a function of the modes of the n-tuple operands and operator.

The Unary Analyzer is executed to examine each operand sign to determine the result sign. The Commutative Re-orderer is called to order θ1 and θ2, and to check for folding or operator degeneracy. The Local Strength Reducer is then executed to check for operator strength reduction. The Redundancy Checker is then called to check for an operator match. A 'true' exit from any of the above calls implies a reduced operation, the result of which is set in θ1. A 'false' exit from all causes the Redundancy Checker to call the PUT module to add the (θP,θ1,θ2) combination to the Code Table as a new binary operation.

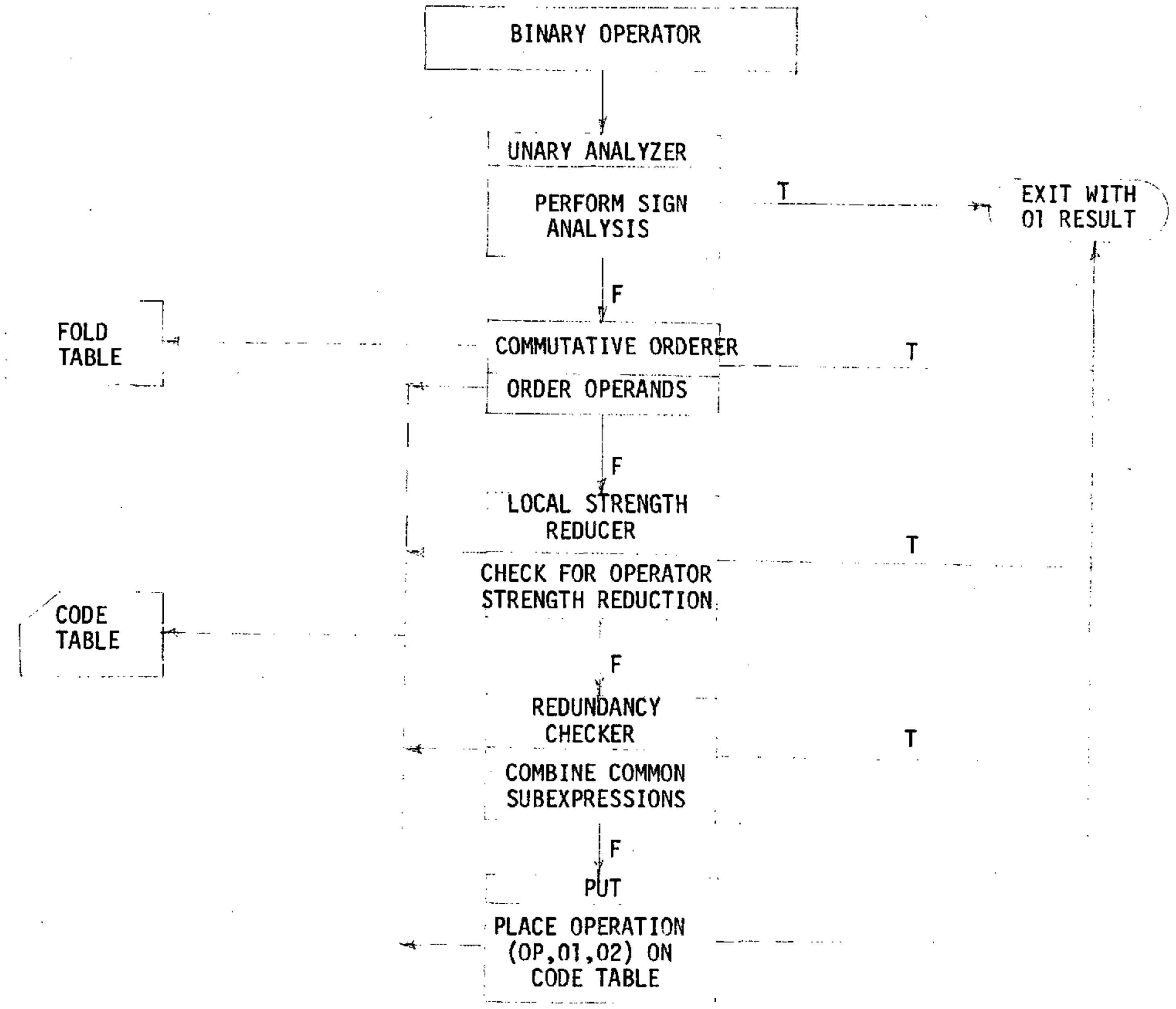


FIGURE 3-2. BINARY OPERATOR LOGIC FLOW

MODULE: Unary Analyzer

FUNCTION: Performs operator/operand sign analysis.

EFFECTS:

The operands of the current operator are examined for the probability of sign cancellation and propagation to the left. The goal is to remove negation signs from operands by flagging the result sign field, thus increasing the chances of sign cancellation.

The following transformations are made for the indicated operations:

$$\begin{array}{lcl}
 & & \underline{\text{Result Sign}} \\
\bar{A} * \bar{B} & \longrightarrow & + \; A * B \\
\bar{A} \,/\, \bar{B} & \longrightarrow & + \; A \,/\, B \\
\bar{A} \left\{\begin{matrix} * \\ / \end{matrix}\right\} \bar{B} & \longrightarrow & - \; A \left\{\begin{matrix} * \\ / \end{matrix}\right\} B \\
\bar{A} \left\{\begin{matrix} * \\ / \end{matrix}\right\} \bar{B} & \longrightarrow & - \; A \left\{\begin{matrix} * \\ / \end{matrix}\right\} B \\
\bar{A} + \bar{B} & \longrightarrow & - \; A + B
\end{array}$$

Also, if either operand is an n-tuple result with a negation flag, and the n-tuple operator is plus, is unreferenced [θP[R]=0], and has two operands with differing signs, both n-tuple signs are complemented, and the operand sign is made positive. Thus,

$$\begin{array}{lcl}
\text{if } t_i = \bar{A} + B & & t_i = A + \bar{B} \\
\bar{t}_i + c & \longrightarrow & t_i + c
\end{array}$$

If a unary operation is detected (i.e., $0 \left\{\begin{matrix} - \\ \text{NOT} \end{matrix}\right\} \theta 2$), the result sign is set and a true exit is made with θ1=θ2.

MODULE: Commutative Re-orderer

FUNCTION: Places θ1 and θ2 in standard order and detects operator degeneracy.

PROCESS:

If the operator (OP) is commutative, its operands θ1 and θ2 are arranged in canonical form:

1. If θ2 is an n-tuple and θ1 is not, the operands are swapped;
2. If the mode of θ2 is less than the mode of θ1, the operands are swapped;
3. If θ2 is a constant pointer and θ1 is not, the operands are swapped;
4. Otherwise, if θ1 < θ2, swap operands.

The following degeneracy check is then made with the indicated transformation followe by a true exit (if found):

$$\left.\begin{array}{l}
0 + \theta 2 \longrightarrow \theta 2 \\
1 * \theta 2 \longrightarrow \theta 2 \\
0 * \theta 2 \longrightarrow 0 \\
0 \,/\, \theta 2 \longrightarrow 0 \\
\theta 1 \,/\, \theta 1 \longrightarrow 0 \\
\theta 1 \,/\, \theta 1 \longrightarrow 1 \\
\theta 1 ** 1 \longrightarrow \theta 1
\end{array}\right\rangle \begin{array}{l}\text{θ1 is set to this result} \\ \text{prior to a true exit.}\end{array}$$

The Folder module is then called to check for operations between constant operands.

MODULE: Folder

FUNCTION: Detects operations between constant operands.

PROCESS:

Information saved within the Fold Table is used to detect an operation between two constant operands. The operands may be actual constants or variables which have been previously set to constants. The algorithm is as follows:

1. If the operator is not computational, exit false.

2. If θ1 is a constant, proceed to step 3. If θ1 is a variable, it is searched for in the Fold Table. If not found, exit false; if found, replace θ1 by its Fold Table entry constant pointer.

3. Perform step 2 with θ2.

4. Perform the constant operation (θP) between θ1 and θ2, and exit true.

MODULE: Local Strength Reducer

FUNCTION: Checks the current operator for possible transformation to a more basic operation.

PROCESS:

A check is made for exponentiation to an integer constant. If found, the following algorithm is performed:

```
1. J = θ2, θ2= θ1, T1= θ1

2. N = [$\log_2 J$]      [greatest integer function]
   I = 0

3. Call the Redundancy Checker with
         θ1 * θ2

4. I = I+1
   IF I< N, θ2 = θ1, repeat step 3;

5. J=J-2**I
   IF J $\leq$0, exit true.

6. θ2=T1, repeat step 2.
```

MODULE: Redundancy Checker

FUNCTION: Checks for match between the current operation and the other n-tuples composing the current block.

PROCESS:

A forward scan of the currently active block is performed to detect a possible match between the current (θP,θ1,θ2) binary operation and a stored n-tuple. If a match is detected, the matching n-tuple result pointer is returned to the Binary Operator module as the operation result.

An additional task is performed involving the combination of two n-tuples with similar operators, where the result of one is an operand of the other, and no operands of either n-tuple are changed in the n-tuples between them. A composite n-tuple is formed by combining the two separate ones.

The logic flow is illustrated in Figure 3-3. Heavy use is made of the 'definition position' of each operand, defined as follows:

1. Whenever a variable is modified, generally through appearing in the result field of an n-tuple (see PUT module), the position of the defining n-tuple is saved in the upper part of the compressed string pointer for the variable in the attribute table (QTABS).

2. Whenever an n-tuple is created (by PUT) with a new result field, the DEFPOS table indexed by the result pointer is set to the n-tuple position in the Code Table.

In checking for redundant operations, the scan time is considerably reduced since the scan begins at the maximum definition position for θ1 and θ2.

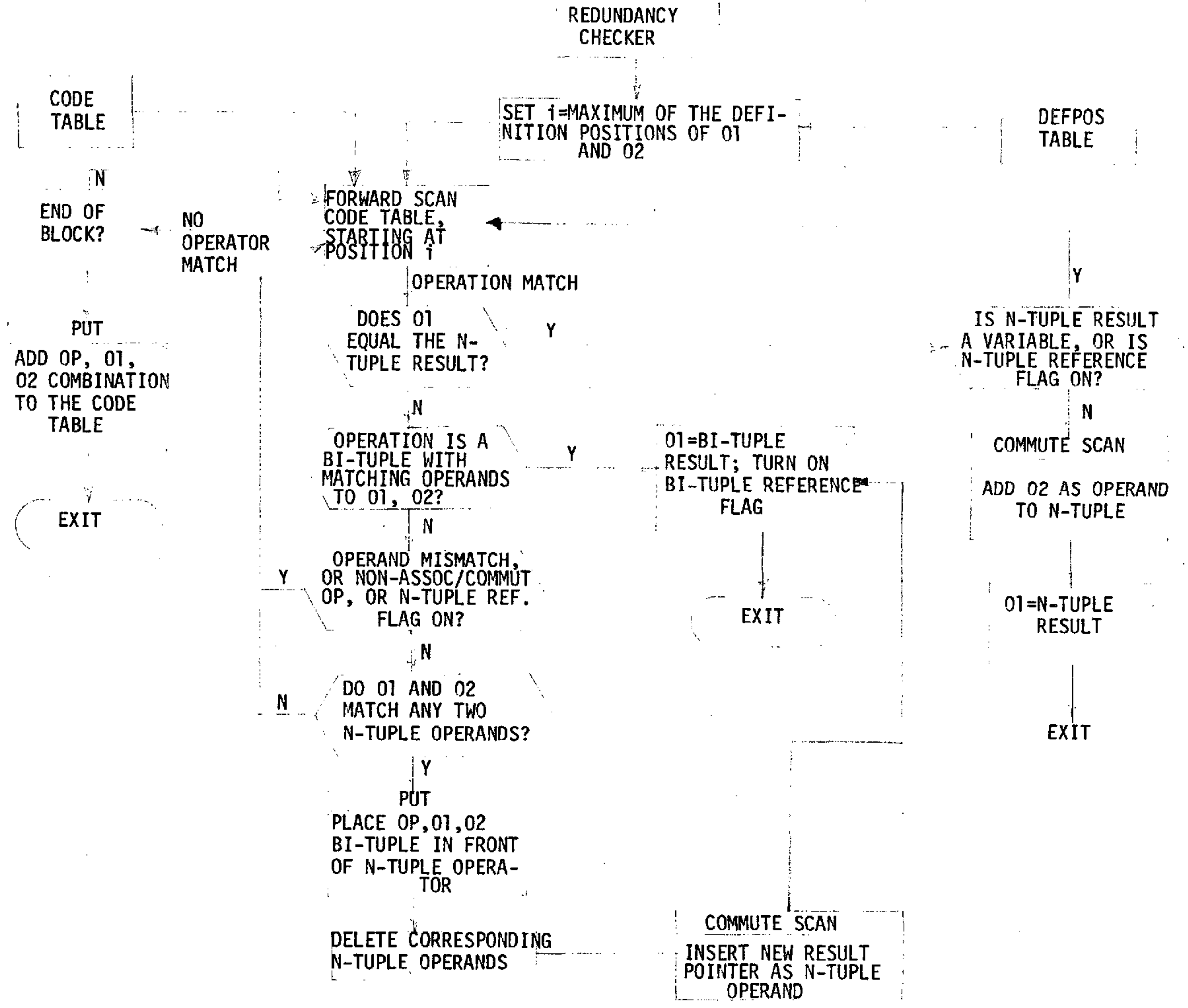

REDUNDANCY CHECKER
CODE TABLE
SET i=MAXIMUM OF THE DEFINITION POSITIONS OF 01 AND 02
DEFPOS TABLE
N
END OF BLOCK?
NO OPERATOR MATCH
FORWARD SCAN CODE TABLE, STARTING AT POSITION i
OPERATION MATCH
Y
PUT
ADD OP, 01, 02 COMBINATION TO THE CODE TABLE
DOES 01 EQUAL THE N-TUPLE RESULT?
Y
IS N-TUPLE RESULT A VARIABLE, OR IS N-TUPLE REFERENCE FLAG ON?
N
N
OPERATION IS A BI-TUPLE WITH MATCHING OPERANDS TO 01, 02?
Y
01=BI-TUPLE RESULT; TURN ON BI-TUPLE REFERENCE FLAG
COMMUTE SCAN
ADD 02 AS OPERAND TO N-TUPLE
EXIT
N
Y
OPERAND MISMATCH, OR NON-ASSOC/COMMUT OP, OR N-TUPLE REF. FLAG ON?
EXIT
01=N-TUPLE RESULT
N
N
DO 01 AND 02 MATCH ANY TWO N-TUPLE OPERANDS?
EXIT
Y
PUT
PLACE OP,01,02 BI-TUPLE IN FRONT OF N-TUPLE OPERATOR
DELETE CORRESPONDING N-TUPLE OPERANDS
COMMUTE SCAN
INSERT NEW RESULT POINTER AS N-TUPLE OPERAND

FIGURE 3-3. REDUNDANCY CHECKER LOGIC FLOW

MODULE: Commute Scan

FUNCTION: Inserts an operand pointer in the appropriate operand position of an n-tuple.

PROCESS:

If the n-tuple in question has a non-commutative operator, the operand argument is inserted in the rightmost position and control is returned.

For a commutative operator, the Commutative Orderer module is repeatedly called with paired operands as follows:

1. The current binary operands θ1 and θ2 are saved.

2. i=1, θ1=argument operand.

3. θ2= the i'th n-tuple operand.

4. The Commutative Orderer is called to arrange θ1 and θ2. The i'th n-tuple operand is replaced with θ1. For a True return, go to step 6.

5. A false return causes i to be incremented, and if more n-tuple operands exist, step 3 is repeated.

6. The saved θ1, θ2 operands are restored.

MODULE: PUT

FUNCTION: The current operation is placed onto the Code Table.

PROCESS:

The current operation and its operand fields are placed on the Code Table in the form of an n-tuple. The following actions are performed:

1. For a bi-tuple V=t, where V is a variable operand and t is an n-tuple result, the n-tuple defining t (i.e., DEFPOS(t)) is checked for a reference flag. If unreferenced, and if V is not modified subsequent to the n-tuple, the operand V replaces the n-tuple result field. The definition algorithm (see step 3) is then performed on V and the n-tuple position.

2. For an operator with n operands, 1+(n+1)/2 words are utilized to store the operator and its operands. If the Code Table overflows, control is passed to the Code Overflow module to free table space. The result field is set to the value of a temporary result count cell, which is then incremented. The table entry DEFPOS (result pointer) is set to the Code Table position of the created n-tuple. If any of the n-tuple operands are themselves n-tuple (temporary) result pointers, the reference flags for the defining n-tuples are set.

3. If OP is a replacement (=) operator, the following tables are updated:

    a. If the n-tuple result field is a variable pointer, a scan for the occurrence of the variable in the Fold Table is performed. If present, and the n-tuple is not of the form V=C, where C is a constant pointer, the Fold Table entry for V is deleted. For the case V=C, the Fold Table entry for V is updated or one is created for the constant C. The Use/Def table is then updated for V.

    b. The n-tuple Code Table position is marked in upper portion of the QTABS variable compressed string pointer, thus defining the most recent variable definition.

   c. Any variable equated to V (through an EQUATE function call) is also processed as in steps 3a, 3b.

4. If the operator is a function call, the following actions are taken:

   a. If the function arguments are marked as 'output' arguments, the actions indicated by steps 3a, 3b are performed on each n-tuple operand corresponding to an 'output' argument.

   b. A call to an external function causes all variables in global data blocks to be processed as in steps 3a, 3b.

MODULE: Loop Control

FUNCTION: Processes the loop definition category of FL terms.

PROCESS:

The directives processed by this module define a loop region to which the Region Global Optimizer module may be (optionally) applied. Loop parameters are encoded in the Loop Table for each FL term as follows:

1. LOOP t,V,lb,le
   A new entry is created on the Loop Table. The loop begin block (lb) and loop end block (le) numbers are saved. The master loop variable (V) is placed as the first primary entry. All other table entry fields are cleared.

2. PLOOP V
   The parallel loop variable (V) is added as the next primary variable entry.

3. LTEST V
   The active block number is saved in the Loop Table as the loop test block number. The compare operand of the last operator, a conditional branch, is stored in the limit field of the last (current) primary variable entry, representing the loop variable Test value.

4. LINCR V
   The active block number is saved in the Loop Table as the loop increment block number. The last operator is examined and the operand used to increment the loop variable in the operation is stored in the increment field of the last (current) primary variable entry.

5. BEGIN le [where le is the block number of the loop end block for the currently active loop]
   The start of the end (successor) block to the active loop cause the Region Global Optimizer overlay to be called under the following conditions:

a. The loop body, loop increment, and loop test blocks are all encoded on the Code Table; i.e., none have been dumped as yet via the Code Generator.

b. All successors to the loop body block which are also predecessors to the loop end block are also on the Code Table; i.e., the loop is complete.

The loop entry is then deleted from the Loop Table.

MODULE: Region Global Optimizer

FUNCTION: Optimizes an entire loop region, consisting of a collection of blocks on the Code Table composing the loop region.

PROCESS:

This overlay module consists of the following submodules to perform the indicated tasks:

1. A Code Mover to remove n-tuple operations from the loop body out of the loop.
2. A Strength Reducer to simplify additive and multiplicative operations on loop increment variables.
3. A Test Replacement checker to manipulate loop test parameters for possible removal of loop variables entirely.

The overall logic is illustrated by Figure 3-4. After activating the loop predecessor block (via Setup), a boolean vector of region, or loop, blocks to examine is formed as follows:

1. The loop body block is included;
2. All immediate successors to this block are included (derived from Block ID Table);
3. All their successors up to the loop successor block are included.

A single scan is then made of all blocks within the region to detect both operations to be moved, and operations on loop variables that may be reduced. As n-tuples are moved to the loop predecessor block (via calls to the Local Optimizer by the Code Mover and Strength Reducer), entries are placed in the Transform Table to cross-correlate n-tuple result pointers.

The Code Mover is applied to a given block only once, even if that block belongs to several nested loop regions. This situation is detected by maintaining a single boolean vector which is the logical or of all loop regions for previous calls to the Region Global Optimizer. For the current call, block i is only examined for strength reduction if bit i of the vector is set.

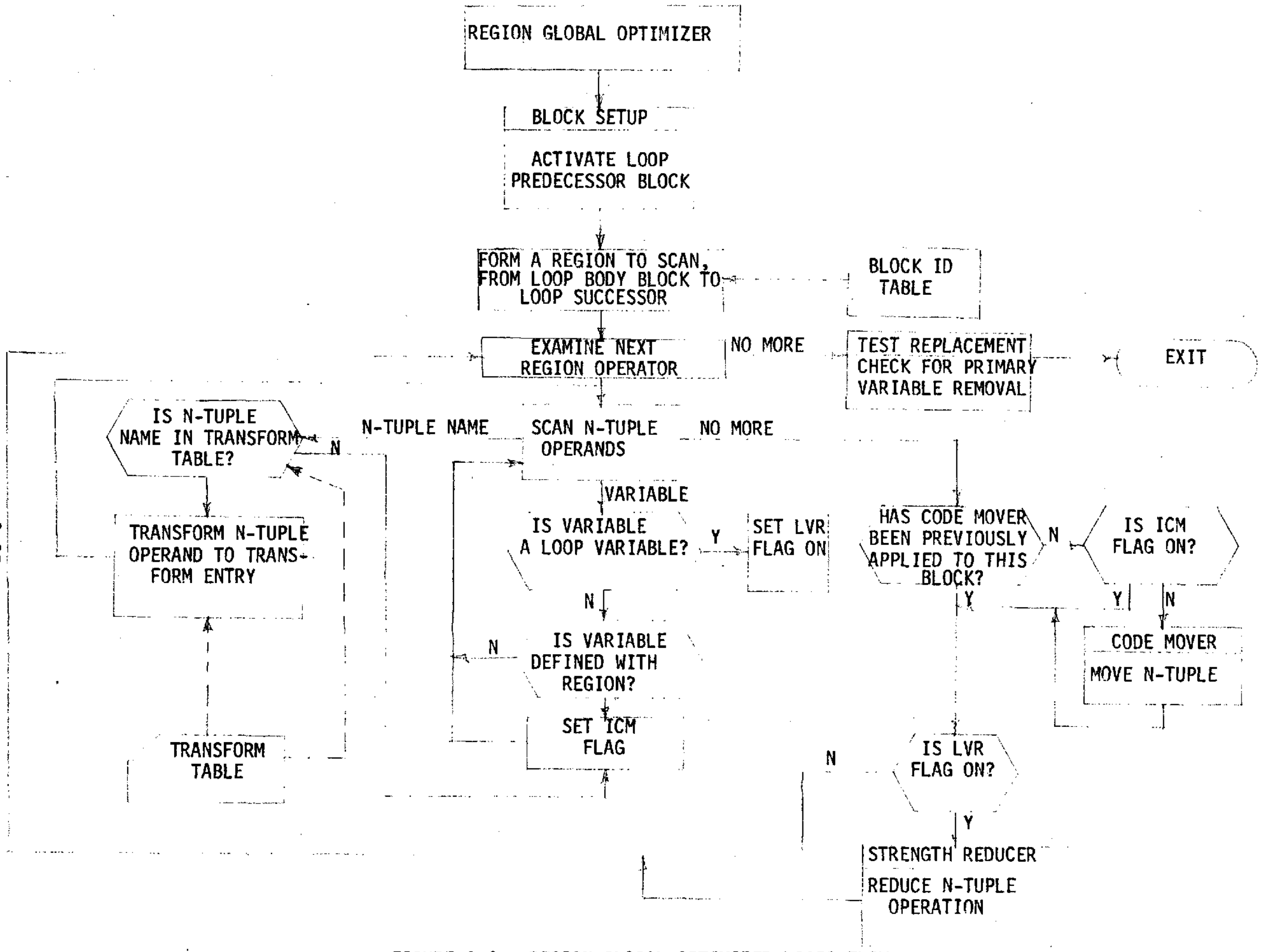

REGION GLOBAL OPTIMIZER
BLOCK SETUP
ACTIVATE LOOP PREDECESSOR BLOCK
FORM A REGION TO SCAN, FROM LOOP BODY BLOCK TO LOOP SUCCESSOR
BLOCK ID TABLE
EXAMINE NEXT REGION OPERATOR
NO MORE
TEST REPLACEMENT CHECK FOR PRIMARY VARIABLE REMOVAL
EXIT
IS N-TUPLE NAME IN TRANSFORM TABLE?
N
N-TUPLE NAME
SCAN N-TUPLE OPERANDS
NO MORE
VARIABLE
TRANSFORM N-TUPLE OPERAND TO TRANS-FORM ENTRY
IS VARIABLE A LOOP VARIABLE?
Y
SET LVR FLAG ON
HAS CODE MOVER BEEN PREVIOUSLY APPLIED TO THIS BLOCK?
N
IS ICM FLAG ON?
Y
N
Y
N
CODE MOVER
MOVE N-TUPLE
N
IS VARIABLE DEFINED WITH REGION?
SET ICM FLAG
TRANSFORM TABLE
N
IS LVR FLAG ON?
Y
STRENGTH REDUCER
REDUCE N-TUPLE OPERATION

FIGURE 3-4. REGION GLOBAL OPTIMIZER LOGIC FLOW

MODULE: Code Mover

FUNCTION: Removes invariant operations from a loop region into its unique predecessor block.

PROCESS:

This module is called by the Region Global Optimizer whenever an invariant operation is detected. The operation is first deleted from the loop region. Then, the Binary Operator submodule of the Local Optimizer is repeatedly called to place the n-tuple operation into the loop predecessor block, which is now active. The following analysis is then performed on the n-tuple result pointer of the operation.

1. If the n-tuple result pointer is itself an n-tuple pointer, an entry is made into the transform table with this pointer and the returned result pointer from the call to the Binary Operator. This will cause all references to the operation result to transformed into a reference to the result in the predecessor block.

2. For a variable n-tuple result (i.e., a definition of a variable):

   a. If the block containing the operation is the loop body block, and there is no use of the variable anywhere in the region (USE/DEF Table), the Binary Operator submodule is again called with the replacement operation: V=returned result pointer. This moves the definition itself into the predecessor block.

   b. If not case a, place the operation: V=<u>returned result</u> in place of the deleted operation in the current block.

MODULE: Strength Reducer

FUNCTION: Reduces an operation on a loop variable.

PROCESS:

This module is called by the Region Global Optimizer to process an operation having a reference to a loop variable active for the current loop region. The loop variable may be primary or secondary. The logic flow is illustrated by Figure 3-5.

Only addition and multiplication operations between loop variables and 'region constants' are reduced. A region constant is detected by the preceding Region Global Optimizer operand scan such that the ICM flag (see Figure 3-4) is not set when the Strength Reducer is called.

A delay mechanism is utilized to place a created loop variable with its initialization code into the predecessor block. This is done to detect a chain of reducible operations, all of which may be combined into a single operation associated with a created loop variable.

The detection of a reference to a secondary loop variable in a non-reducible operation causes the initialization code for the variable to be inserted into the loop predecessor block.

An entry is made into the transform table for a reduced operation to convert all references to the operation result into the created loop variable name.

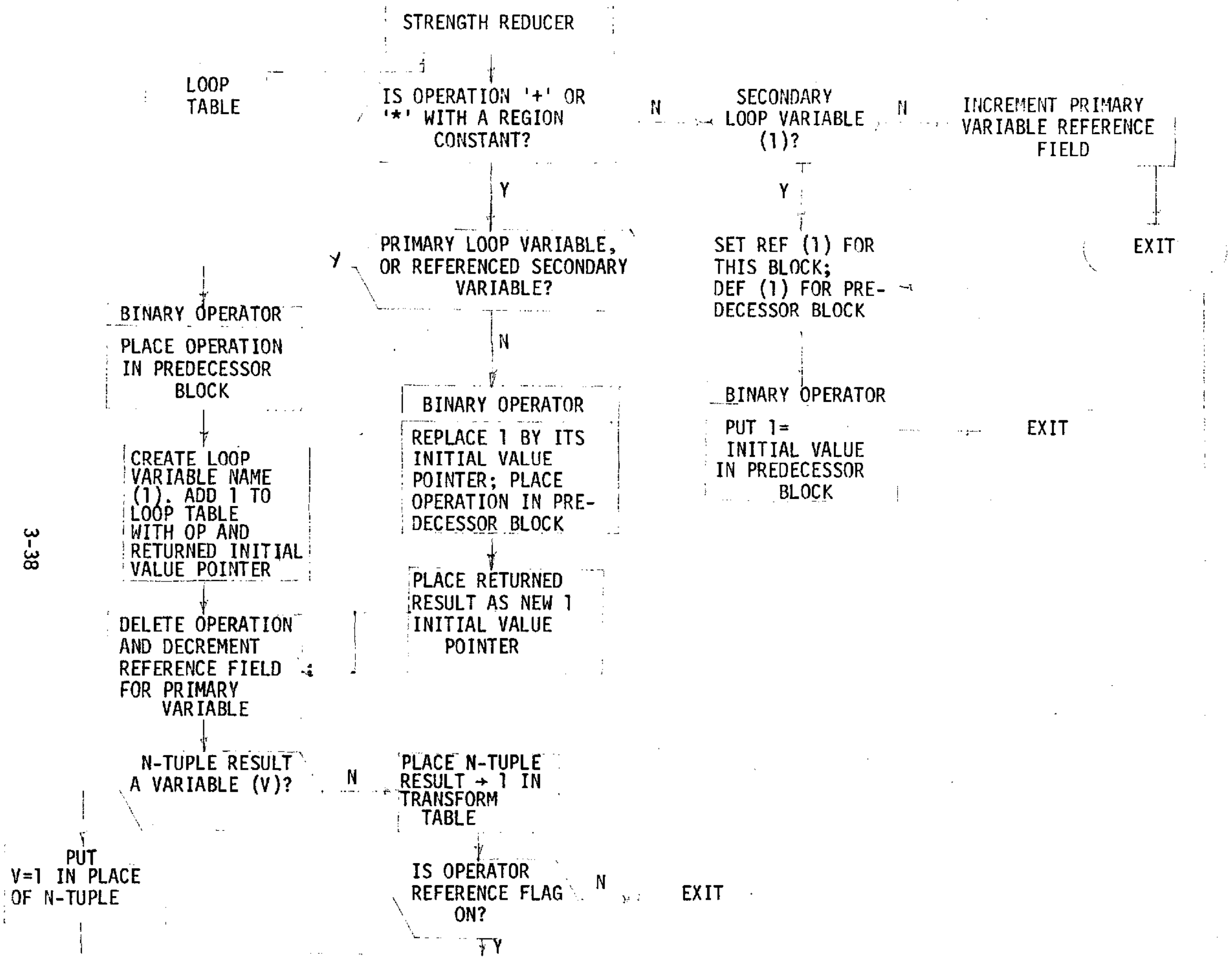

STRENGTH REDUCER
LOOP TABLE
IS OPERATION '+' OR '*' WITH A REGION CONSTANT?
N
SECONDARY LOOP VARIABLE (1)?
N
INCREMENT PRIMARY VARIABLE REFERENCE FIELD
Y
Y
PRIMARY LOOP VARIABLE, OR REFERENCED SECONDARY VARIABLE?
Y
SET REF (1) FOR THIS BLOCK; DEF (1) FOR PREDECESSOR BLOCK
EXIT
BINARY OPERATOR
PLACE OPERATION IN PREDECESSOR BLOCK
N
BINARY OPERATOR
REPLACE 1 BY ITS INITIAL VALUE POINTER; PLACE OPERATION IN PREDECESSOR BLOCK
BINARY OPERATOR
PUT 1= INITIAL VALUE IN PREDECESSOR BLOCK
EXIT
CREATE LOOP VARIABLE NAME (1). ADD 1 TO LOOP TABLE WITH OP AND RETURNED INITIAL VALUE POINTER
PLACE RETURNED RESULT AS NEW 1 INITIAL VALUE POINTER
DELETE OPERATION AND DECREMENT REFERENCE FIELD FOR PRIMARY VARIABLE
N-TUPLE RESULT A VARIABLE (V)?
N
PLACE N-TUPLE RESULT → 1 IN TRANSFORM TABLE
PUT V=1 IN PLACE OF N-TUPLE
IS OPERATOR REFERENCE FLAG ON?
N
EXIT
Y

FIGURE 3-5. STRENGTH REDUCER LOGIC FLOW

MODULE: Test Replacement

FUNCTION: Detects redundant tests and increments on primary loop variables.

PROCESS:

This module is invoked by the Region Global Optimizer as the last phase in the analysis of a loop region. The objective is to remove any test and increment code for loop variables which are unreferenced elsewhere in a loop region. The tests are replaced by alternative tests on secondary variables which are additive or multiplicative functions of the primary loop variable. The details of the process are shown in Figure 3-6.

The Loop Table is scanned for all primary loop variables associated with the active (last) loop. These variables include the master loop variable and all parallel loop (PLOOP) variables. If a loop variable is unreferenced in the loop except for its test and/or increment code, its test code is replaced by a test on one of its secondary variables, if any are present. The 'simplest' secondary variable is chosen is follows:

1. If any 1-region pointer is a constant pointer, choose it.
2. The next choice is any 1 with a variable 1-region pointer.
3. Choose any 1 otherwise.

If the replaced V has an increment code and is unreferenced in any loop successor block, its increment code is deleted from the loop increment block. A check is then made to see if the loop increment block has any computational code other than incrementation, a condition indicating that the V-increment expression is not a region constant since it otherwise would have been removed by the Code Mover.
If no such code is found, the region constant associated with each secondary variable (to V) is replaced by a multiplicative or additive function on the V-increment within the loop predecessor block.

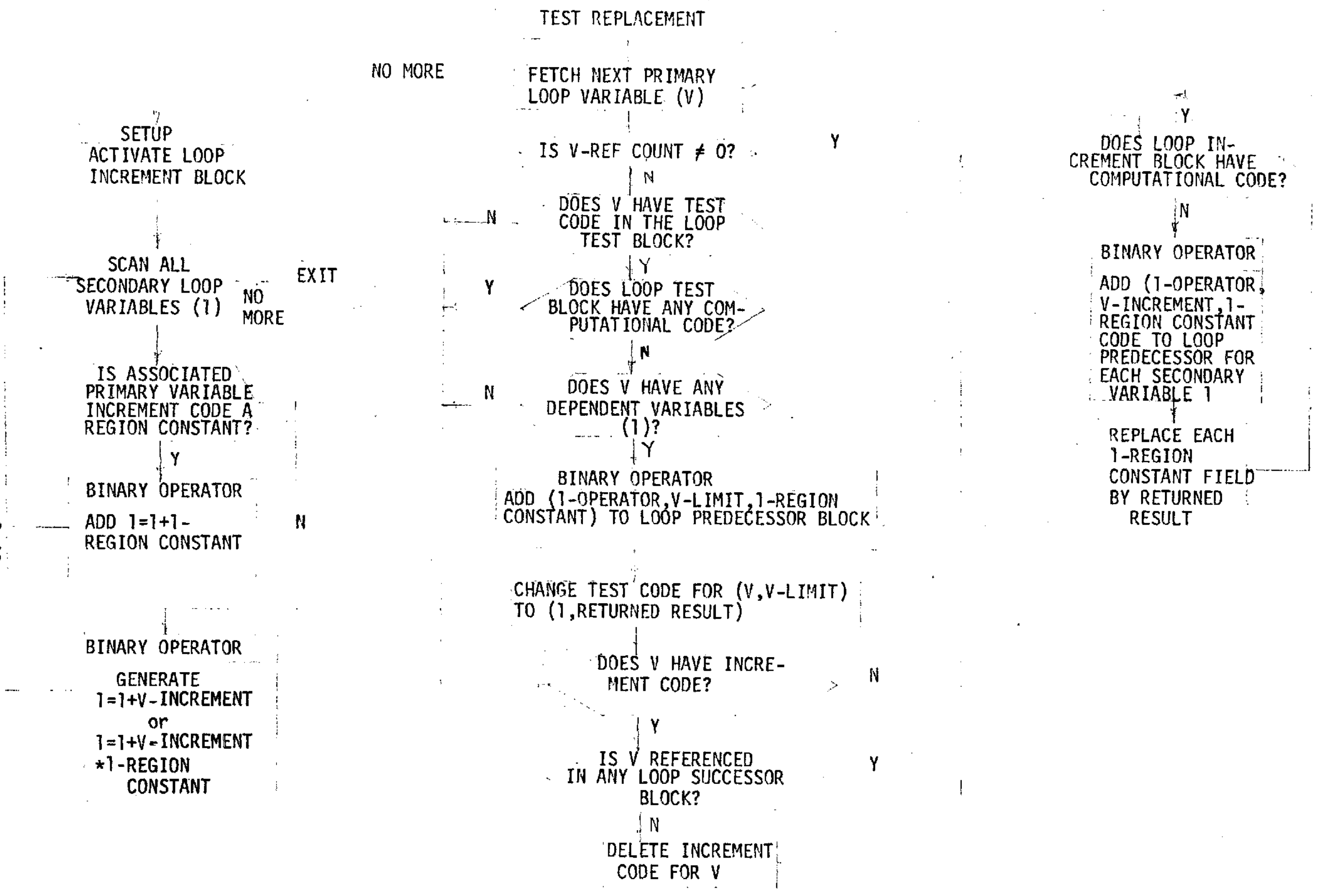

TEST REPLACEMENT
NO MORE
FETCH NEXT PRIMARY LOOP VARIABLE (V)
IS V-REF COUNT ≠ 0?
Y
N
DOES V HAVE TEST CODE IN THE LOOP TEST BLOCK?
N
Y
DOES LOOP TEST BLOCK HAVE ANY COMPUTATIONAL CODE?
Y
N
DOES V HAVE ANY DEPENDENT VARIABLES (1)?
N
Y
BINARY OPERATOR
ADD (1-OPERATOR,V-LIMIT,1-REGION CONSTANT) TO LOOP PREDECESSOR BLOCK
CHANGE TEST CODE FOR (V,V-LIMIT) TO (1,RETURNED RESULT)
DOES V HAVE INCREMENT CODE?
N
Y
IS V REFERENCED IN ANY LOOP SUCCESSOR BLOCK?
Y
N
DELETE INCREMENT CODE FOR V
SETUP
ACTIVATE LOOP INCREMENT BLOCK
SCAN ALL SECONDARY LOOP VARIABLES (1)
NO MORE
EXIT
IS ASSOCIATED PRIMARY VARIABLE INCREMENT CODE A REGION CONSTANT?
Y
N
BINARY OPERATOR
ADD 1=1+1-REGION CONSTANT
BINARY OPERATOR
GENERATE 1=1+V-INCREMENT or 1=1+V-INCREMENT *1-REGION CONSTANT
Y
DOES LOOP INCREMENT BLOCK HAVE COMPUTATIONAL CODE?
N
BINARY OPERATOR
ADD (1-OPERATOR, V-INCREMENT,1-REGION CONSTANT CODE TO LOOP PREDECESSOR FOR EACH SECONDARY VARIABLE 1
REPLACE EACH 1-REGION CONSTANT FIELD BY RETURNED RESULT

FIGURE 3-6. TEST REPLACEMENT LOGIC FLOW

The loop increment block is then activated through a call to Setup. The increment code for each secondary variable is inserted as follows:

1. If the associated primary variable (V) increment code is a region constant, generate

```
l=l+l-region constant
```

2. Otherwise, generate

```
l=l+v-increment        [for additive variables]
l=l+v-increment*l-region constant
```

MODULE: Code Overflow

FUNCTION: Creates free space on the Code Table whenever an overflow condition occurs.

PROCESS:

This submodule is called by the PUT submodule of the Local Optimizer whenever insufficient space remains on the Code Table to store the current n-tuple operation. The Block Sequence overlay module is called to choose the next block[s] on the Code Table to be dumped as generated code. The resulting released space on the Code Table is flagged as available and control then returns to the calling module.

MODULE: Block Sequence

FUNCTION: Determines the next program block or blocks to dump as generated code.

PROCESS:

One or more code blocks residing on the Code Table are dumped by calling the Block Code Generation submodule. The choice of the block[s] to be output proceeds as follows:

1. The last block to output (i) is maintained. The initial value for i is 1 (the prolog block).
2. The next immediate successor to i, determined by the Block ID Table, is examined. If the successor has already been dumped, another successor to i is examined until one is found which has not yet been dumped.
3. If the successor block is the exit (highest numbered) block, i is set to 1, the Register Initializer is called, and step 2 is repeated. If a complete scan is made and no successors other than the exit block are candidates, step 8 is performed.
4. If the successor block (j) is the predecessor block for an active loop, determined by a scan through the Loop Table, then i is set to the loop successor block and step 2 is performed.
5. The last two operations of the successor block (j) are examined for the following condition:
   a. The last operation is a GO b_k and is preceded by a COND BRANCH ... b_n.
   b. Block b_n has block b_k as a successor.

   The following transformation within block j is then made:

$$\begin{Bmatrix} \text{COND BRANCH} \ldots b_n \\ \text{GO } b_k \end{Bmatrix} \rightarrow \begin{Bmatrix} \overline{\text{COND BRANCH}} \ldots b_k \\ \text{GO } b_n \end{Bmatrix}$$

   [The conditional branch operator condition is logically reversed.]
6. The Block Code Generator is then called to generate code for block j.
7. If block j ends with a GO b, then j is set to b and step 4 is performed. Otherwise, a true exit is made.
8. Since only loop region blocks remain on the Code Table, the currently active loop predecessor block becomes i and step 2 is performed. The dumping of one or more loop blocks deactivates any loop analysis by the Region Global Optimizer for that loop.

MODULE: Block Code Generation

FUNCTION: Generates target code for a program block and frees the corresponding Code Table area.

PROCESS:

This module is called by the Block Sequence module to convert the encoded operations for a program block on the Code Table into target code. A single scan of the block is made, with code generation taking place through PROC expansions of lower-level PROCS associated with each n-tuple pseudo-operation.

During the operator scan, operator/operand information is unpacked from the Code Table n-tuple entry into global cells as follows:

OP -- current operation number.

RESULT -- operation result pointer.

RM -- computational mode of the result.

O1 -- pointer to the first operand.

NF1 -- negation flag for the first operand.

IT1 -- flag indicating the type of operand:

- 0 = symbol pointer.
- 1 = n-tuple temporary result name.
- 2 = target register.

IM1 -- computational mode of operand one.

IU1 -- the symbol type for variable (IT1=0) operands.

FB1,PREC1,... -- the unpacked symbol attribute information for variable operands, such as number of fractional bits, precision, etc...

O2,NF2,IT2... -- the same information as above for the second operand (if present).

As each operand is unpacked, the target register contents (RVALUE) are scanned to find a register containing the operand. If found, 0_i is set to the register number and ITi is set to 2.

For binary operations, the code expansion is performed on two operands at a time as follows:

1. The first two operands are unpacked as 01 and 02 (NF1,NF2, etc...).
2. The binary operation (OP,01,02) is expanded.
3. The result register REG (if any) is associated with the operation result pointer RESULT.
4. If more operands remain, 01 is set to RESULT, the next operand is unpacked as 02, and step 2 is repeated.

Prior to expanding a binary operation (step 2) or a general n-tuple operation, the operands are placed onto the PROC Control stack in preparation for a manually initiated PROC expansion. For an operation defined by the compiler writer, the PROC Expand module is then executed immediately. For a system-defined operation, certain predefined steps are taken prior to initiating code expansion for certain operators as defined below:

1. For commutative binary operators, the following transformations are made:

   $\overline{01} + 02 \longrightarrow 02 - 01$

   $01 + \overline{02} \longrightarrow 01 - 02$

   01 OP Register $\longrightarrow$ Register OP 01

2. If 02 is a symbol (IT2=0) having a bit starting or ending position in the middle of an addressing unit, then:

```
FCH ('A',02)          [Fetch 02 into an accumulator]
REG   02,IT2=2,etc...     [Replace 02 by a register]
```

The default code expansion for all system-defined operations are provided. All such PROC expansions are machine independent and require user supplied PROCs only at most basic level.

Subsequent to processing a complete operation successfully, which includes the true exit from the highest level PROC expansion, any deferred machine code stacked onto the tail end of QCTAB (see Code Construct module) is output via a call to the Code Dump submodule.

MODULE: PROC Expand

FUNCTION: Enters the expansion of a PROC skeleton contained in the PROC skeleton table (IPROC).

PROCESS:

This module performs the expansion of a PROC through interpretive execution driven by table IPROC and the PROC control stack (QCTAB). All expansions are recursive, allowing a PROC to call another PROC at any level, including itself.

The PROC skeletons are prepared by the Meta-Compiler and are placed on the Compiler Library Data file during the target definition phase. The skeletons compose the largest portion of a complete target definition data entry, which is read into table IPROC at the start of the Function Processor execution.

The PROC control stack (QCTAB) contains all arguments to all active PROCs as well as pointers to control nested PROC execution. The expansion of each PROC and the manipulation of QCTAB proceeds as follows:

1. To expand PROC number i, the current-string-ptr (CS) is set to the PROC start location, given by the i'th position in the PROC transfer vector preceding the PROC skeletons. If CS=0, the PROC definition is absent and a false exit is performed (step 8).
2. The current-element-ptr (CE) is set to CS+1.
3. The PROC element pointed to by CE is processed (see below). If false, step 7 is initiated to move to the next PROC string.
4. Advance CE to next element and repeat step 3. If no more elements, perform a true exit (step 6).

5. Perform a true exit: the last QCTAB entry is popped, CE and CS are reset to values at the previous level, and step 4 is repeated. If QCTAB was empty, a true exit from PROC Expand is made.
6. False string: CS is advanced and step 2 is repeated for the next alternative PROC string. If no alternative exists, perform a false exit (step 8).
7. False exit: the last QCTAB entry is popped, CE and CS are reset, and step 4 is executed. If QCTAB was empty, a false exit is made from PROC Expand.

Processing a PROC Element:

Element Type	Actions
1=Replacement	Evaluate operand 2 (see below); replace operand 1 variable with VALUE result.
2=Condition Test	Evaluate both operands and compare with condition operator to determine element truth.
3=NULL element	No action; element is true.
4=FALSE element	No action; element is false.
5=PROC call	A call entry is stacked onto QCTAB. The PROC operand pointers are copied directly. Step 1 is initiated with the called PROC number.
6=Code Request	A call entry is stacked onto QCTAB as for type 5. The Code Construct module is then called, returning the element truth status.
7=Support Function Call	The same actions are performed as for type 6, except the appropriate support function is called instead of the Code Construct module.
8=Substring Operand	A dummy call entry (CS,CE, and number of args=0) is placed on QCTAB. CS is incremented by one and step 2 is executed.

Evaluating a PROC Operand:

A general operand pointer of the form: (operand-type, operand-value [OVAL]) is evaluated, resulting in a returned VALUE, as follows:

Operand-Type	Returned VALUE
0=variable	The contents of the variable [PROC variable start offset + OVAL] is returned.
1=integer	OVAL is returned.
2=literal string	OVAL is returned.
3=location counter	The current location counter value + OVAL is returned.
4=symbol pointer	The contents of the variable is returned.
5=expression	The expression pointed to by OVAL is evaluated (see below).
6=PROC argument	The operand pointer on QCTAB corresponding to argument number OVAL is considered the operand to evaluate.
7=TYPE OF operand	The operand-type portion of argument OVAL is returned.
-K=simple expression	The following (secondary) operand pointer is examined by type: 0 - return variable contents + K 3 - return entire operand pointer; K 4 - return entire operand pointer; K 6 - replace secondary operand by indicated argument and re-examine.

Note: Only a 'simple expression' operand can be returned from the operand evaluator as a dual result rather than a single value. This allows 'variable + offset' or '$+offset' combinations to be identified as single operands.

Evaluation of an Expression:

Encountering an expression operand pointer causes the evaluation of the pointed to expression, which is stored in the PROC Expressions area of IPROC. The format of a expression is as described for table PEXP in the Meta-Translator (see section 1.2). Each operation is applied to the evaluated result of its two operands. If two or more 'dual result' operands are present, they may only be combined in a sensible manner, such as:

```
(V+10)+7
(V1+5)-(V2)
```

MODULE: End

FUNCTION: Terminates the Function Processor and forces the generation of all delayed code.

PROCESS:

The Block Sequence module is repeatedly called to convert all remaining n-tuple operations in blocks on the Code Table into generated code. The Literal Dump module is then called to output all referenced constants. Control is then returned to the Main Control to terminate the execution of the Function Processor.

MODULE: Code Construct

FUNCTION: Maps a code request made within a PROC into a target instruction.

PROCESS:

A code request is processed, resulting in the stacking of intermediate language (IL) code on the end of the PROC control stack (QCTAB). During the construction of an IL instruction, any error condition causes a false exit to be made back to the calling module (PROC expand).

The arguments to the code request are to be found as the latest entry on QCTAB. The first argument defines a pointer to the operator description within the IPROC operations section. The specified instruction format (iform) pointers define a collection of alternate formats to which the operation can be mapped. The formats are applied one at a time until one succeeds in constructing the operation; only if all fail, is a false exit made.

Each iform field is matched with a code argument, and the computed or supplied value corresponding to the argument is placed in the designated position within the target bit string being assembled. The associated values for each code argument/iform operand combination proceeds as follows:

IFORM OPERAND TYPE	ALLOWED MATCHING CODE ARGUMENTS
Operation Code--0	-

The op code specified within the operation definition defines the value to be substituted.

IFORM OPERAND TYPE	ALLOWED MATCHING CODE ARGUMENTS
Register Designator-RS	Variable (0), integer (1), expression (5), PROC argument (6).

The evaluated input argument value represents a target register of class 'S'. If the argument is not resolvable to a single absolute value, or is not of class 'S', a false condition occurs.

Control Field - F	Variable (0), integer (1), expression (5), PROC argument (6).

The evaluated input argument value is substituted as the field value. If the argument is not resolvable to a single absolute value, a false condition occurs.

Constant

No matching code argument is fetched. The constant field within the iform specification becomes the substituted field.

Memory Reference - M	All code argument types.

A memory reference iform type consists of up to four subtypes appearing in the same order as the corresponding code arguments. The subtypes define the memory addressing mode, optional index register, optional base register, and optional indirect flag. Each subtype corresponds to a code argument as follows:

MEMORY REFERENCE SUBTYPE	ALLOWED MATCHING CODE ARGUMENT
1. Location Counter - L	Any type.
Signed Displacement - SD	
Unsigned Displacement - D	

Any of the above three addressing mode specifiers typically require a matching memory reference argument (see below).

MEMORY REFERENCE SUBTYPE	ALLOWED MATCHING CODE ARGUMENT
2. Index Register - X	Variable (0), integer (1), expression (5), PROC argument (6).

The argument must be resolvable to a single value corresponding to a register of type 'X', i.e., an index register.

3. Base Register - B	Variable (0), integer (1), expression (5), PROC argument (6).

The argument must be resolvable to a single value corresponding to a register of type 'B', i.e., a base register.

4. Indirect Address - *	Variable (0), integer (1), PROC argument (6).

The argument is single valued and represents an indirect addressing flag. The above subtypes may be specified in the following combinations with the indicated meaning.

(1) L[,X][,*]   Direct memory reference with possible indexing and indirect addressing.

The computed effective address is assumed to be a memory reference plus the contents of register X (if present). If an indirect flag is present, the indirectness is assumed to be applied after computing L+(X), i.e., (L+(X)) is the effective memory address.

(2) [S]D[,X][,*]   Location counter displaced with possible indexing and indirect addressing.

The value [S]D is computed as a [signed] displacement from the current value of the location counter by subtraction from the memory reference location. Any indirectness is assumed to be performed last, i.e., ([S]D+current location counter + (X)) is the effective memory address.

(3) [S]D,B[,X][,*]   Base register, displaced with optional indexing and indirect.

If the code argument corresponding to [S]D is a [signed] resolvable value, it is directly substituted as are all remaining arguments. The supplied base B is assumed to be the selected base.

If the argument for [S]D is not resolvable, it defines a memory reference requiring automatic displacement from a base register. One base is selected from the 'B' class having a current assigned address within range of the displacement field width. If none are found, a base is automatically loaded with a program or data base location counter value within range of the memory operand.

The successful completion of the application of an iform to a code request results in a series of TXT code records formatted as IL object text. The IL is stored onto the control table (QCTAB), to be subsequently otuput as code by the Code Dump module upon a true exit from the highest level PROC.

MODULE: Register Manipulation

FUNCTION: Maintains the status of all target registers.

PROCESS:

This module consists of a collection of submodules for fetching, saving, and defining the contents of all target registers. Each register is identified explicitly by number or by association with one of 26 possible register classes. Each register is described by 6 flag words defining all aspects of the current status of the register. The register class and flag data is maintained in contiguous areas of the IPROC table.

The enclosed submodules are accessible to code generation process through support function calls invoked within PROCs. Each submodule has a particular register maintenance function as follows:

SVR(R) - Saving a Register

1. Register R is saved in a temporary storage variable if it is currently associated with an n-tuple name referenced in a future n-tuple operator. All such future references are replaced by the temp-name pointer.
2. Register R is freed and made available (FRG(R)).
3. If R is found to be a continuation of a lower numbered register (i.e., RVALUE (R)=1, the lower numbered register (up through R) is saved and freed.

FRG(R) - Freeing a Register

1. Frees the register R and marks it as being available. [I.e., RVALUE(R),...,RVALUE(R+WSIZE(R)-1)=0;WSIZE(R),...,WSIZE(R+WSIZE(R)-1)=0]

DFR(R,S,M,V) - Defining a Register

1. Defines the register R as containing the operand V, with size S. [I.e., RSIZE(R)=S;RVALUE(R)=V;RVALUE(RH),..,RVALUE(R+S-1)=1]
2. The mode of the register is set to M. [RMODE(I),...,RMODE(R+S-1)+M].

RSR(V) - Search Registers

1. Searches the target registers for the operand V.
2. Returns 0 if not found, or the register number if found.

IRG - Initialize Registers

1. Initializes all modifiable (i.e., RHOLD(R)=0) registers. [I.e., clears RVALUE, WSIZE cells]

GTR('T',SIZE) - Get a Register

1. Gets SIZE registers of type T (A,B,X,G, or other) from the register pool.
2. May call SVR to save and free registers if SIZE consecutive registers are not free (RVALUE=0).
3. Gets only non-hold registers (RHOLD=0).
4. Returns the value of the found register. Clears RVALUE(R) and sets WSIZE(R)=SIZE.
5. If SIZE consecutive registers are not free, then all registers containing variables are freed, and a second try is made before SVR is called.

GTS(M,PREC) - Get Temporary Storage Name

1. Creates a temporary storage cell in the symbol table of mode M and precision PREC.

$\underline{\text{FCH}_R(\text{'T'},\text{A})}$ - Fetch into a Register

1. Fetches the operand A and places it in a register. If A is a symbol, a register is fetched from the register pool and A is loaded into it. The sought after register is selected according to type 'T'. If A is a register not of type 'T', it is transferred to one of type 'T'.
2. The current accumulator cell REG is set to the register.
3. The register mode and size are determined by A.
4. If a specific register R is requested, it is force-loaded with A.

SETREG(R,SIZE,MODE) - Set Register Contents

1. Set current accumulator REG to R.
2. Set RMODE(R)=MODE, WSIZE(R)=SIZE.
3. The value of the current accumulator REG will be set to contain the current N-tuple upon completion of the processing of the current N-tuple.

MODULE: Code Dump

FUNCTION: Outputs intermediate language or target language along with an assembly listing.

PROCESS:

This module is called by PROC Expand upon a true exit from the highest level PROC. Any delayed code remaining on the control stack (QCTAB) is output and assembly listing lines are generated for each.

If a single pass compilation has been selected, the intermediate language (IL) stacked on QCTAB is sent directly to the text output file (unit 5) as Target Language, i.e., object text. Any references to future symbols in the IL are replaced by reference threads, to be followed by definition threads at the time each future symbol is defined (refer to Volume I, section 3, Target Language). If an assembly listing has been requested, a listing line is generated and passed to the print file (unit 2).

For a two pass compilation the stacked IL is passed to the Intermediate File (unit 4). Assembly listing lines, if requested, will be generated by the Operation Processor.

3.2 Internal Data Structures

Several control arrays are present within the Function Processor to drive the optimizers and code generative routines. The format and use of each array is described below in symbolic format.

Loop Table - LOOP

Format:

```
word i*4:      value-ptr-i, lp
               lb      lt
               li      le
                   .
                   .
                   .

value-ptr-i:   ref-count,limit    }  Primary Variable
               increment, V       }
                   .
                   .
                   .
               region-constant,op }  Secondary Variable
               assoc-ptr,l        }
```

value-ptr-i	-- pointer to the loop variables and their values for the active loop at level i.
lp	-- loop predecessor block number.
lb	-- loop body block number.
lt	-- loop test block number (0=no test code).
li	-- loop increment block number (0=no test code).
le	-- loop successor block number.
ref-count	-- number of references to this primary loop variable within the loop.
limit	-- primary variable limit code result pointer.
increment	-- primary variable increment code result pointer.
V	-- primary variable pointer.
region-constant	-- secondary variable initial value result pointer.
op	-- secondary variable operation type (additive or multiplicative function of associated V).
assoc-ptr	-- value-ptr-i pointer for the primary variable associated with the secondary variable.
l	-- secondary variable pointer.

Use:

The relevant information for all active loops is stored in this table dynamically by the FL term loop processors and the Region Global Optimizer modules.

Loop variables are of two types: primary and secondary. Primary loop variables include the master variable defined in the LOOP FL term and all parallel loop variables defined in subsequent PLOOP FL terms. Secondary loop variables are created by the Strength Reducer submodule and are associated with a corresponding primary loop variable.

Use/Definition Table - USE/DEF

Format:

$$\text{For variable } i \left\{ \begin{array}{lll} & \underline{\text{Use}} & \underline{\text{Def}} \\ \text{word } i{*}n: & u_{i1} & d_{i1} \\ +1 & u_{i2} & d_{i2} \\ \vdots & \vdots & \vdots \\ +n-1: & u_{in} & d_{in} \end{array} \right.$$

n -- the number of host words required to hold a boolean description vector for a block, computed as

$n = \frac{m-1}{qwdsz} + 1$, where m = the total number of program blocks;

qwdsz = the host word size in bits.

u -- the boolean use vector.

d -- the boolean definition vector.

Use:

This table defines all definitions and uses of variables within program blocks. A variable is defined (or used) in block j if bit j of the boolean vector d (or u) is set.

Definition Position Table - DEFPOS

Format:

word i: codeposition

codeposition -- Code Table position pointer.

Use:

For a given block, the i'th word in table DEFPOS gives the position of the n-tuple operator with the result operand i. The table is cleared at the start of each code block.

Fold Table - QFOLD

Format:

```
        word i:  varptr, conptr
                   :       :
```

varptr -- pointer to a variable.
conptr -- pointer to a constant.

Use:

QFOLD contains a list of all variables having a constant value in the currently active block. It is cleared whenever a new block is initiated. Entries are deleted or entered by the PUT module upon processing a replacement operator.

Block Identification Table - QBID

Format:

```
        word i*(n+2):  label,addr        )
                       begin,end          Block i
                       connectivity_1     information
                          :
                       connectivity_n
```

begin -- block start position in Code Table (0 implies the code for the block is not yet in the Code Table).

end -- block end position in the Code Table.

label -- pointer to the label associated with the block.

connectivity -- the connectivity vector for block i.

Block j is an immediate successor to block i if the j'th bit of connectivity is a one.

n -- $\frac{K+QWDSZ-1}{QWDSZ}$, where K = the total number of program blocks,
QWDSZ = the host word size in bits.

addr -- the address of the generated code for the program block (0 implies not yet generated)

Use:

BID defines all relevant information concerning the basic blocks of the program being compiled. The position of a block in the Code Table as well as all flow dependency relationships between blocks can be determined from the encoded information.

Code Table

Format:

```
                    :
                 n-tuple
                 n-tuple
                    :
```

```
n-tuple --  r, op, result, n
            nf1, 01, nf2, 02              )
            :                             >  n/2 words
            nf_{n-1}, 0_{n-1}, nf_n, 0_n  )
```

$$\left.\begin{array}{l} nf1,\ 01,\ nf2,\ 02 \\ \vdots \\ nf_{n-1},\ 0_{n-1},\ nf_n,\ 0_n \end{array}\right\} n/2 \text{ words}$$

r -- reference flag on if this n-tuple result is currently referenced as an operand to a future n-tuple.

op -- the n-tuple operator number.

n -- the number of n-tuple operands.

nf_i -- the negation flag for operand i.

0_i -- the i'th operand pointer.

r -- the result operand pointer.

An operand pointer has the form:

or	0,tabnum,symbol-sequence-number.	[symbol pointer operand]
	1,mode,quad	[quadruple result operand]

tabnum -- symbol table segment number.
symbol-sequence-number -- symbol position.
mode -- computational mode of the quadruple result.
quad -- quadruple result number.

Use:

PROC Skeleton Table - IPROC

This table contains the expansions for all system and compiler-writer-defined PROCs. Its structure is identical to that of record two of a Compiler Library Data file entry (see section 5.1).

PROC Control Stack - QCTAB

Format:

```
     word 1:  current-string-pointer   }
              current-element-pointer  }
              arg1                     }    Level 1
              arg2                     }    PROC, Code,
                :                      }    or Support call
              argn                     }
              n                        }

     word n+3:  current-string-pointer }
                  :                    }
                                       }    Level 2
                                       }
                                       }
                                       :
```

current-string-pointer -- the saved string position in table IPROC.

current-element-pointer -- the saved string element position in table IPROC.

n -- number of argument words.

argi -- the operand pointer to the i'th argument. The operand pointer format is discussed in the PROC Expand module.

Use:

This control stack supports the nested expansion of all PROCs by the PROC Expand module.

4. OPERATION PROCESSOR PASS II

This section presents the design of the Operation Processor Pass II module. This module is responsible for the conversion of Intermediate Language object text into Target Language.

This module is invoked only if a two pass compilation has been requested, in which case the Intermediate Language file is created by the Function Processor as preliminary object text.

4.1 Program Logic Modules

The major modules and submodules of the Operation Processor Pass II are described below.

Figure 4 presents a block diagram of the module logic.

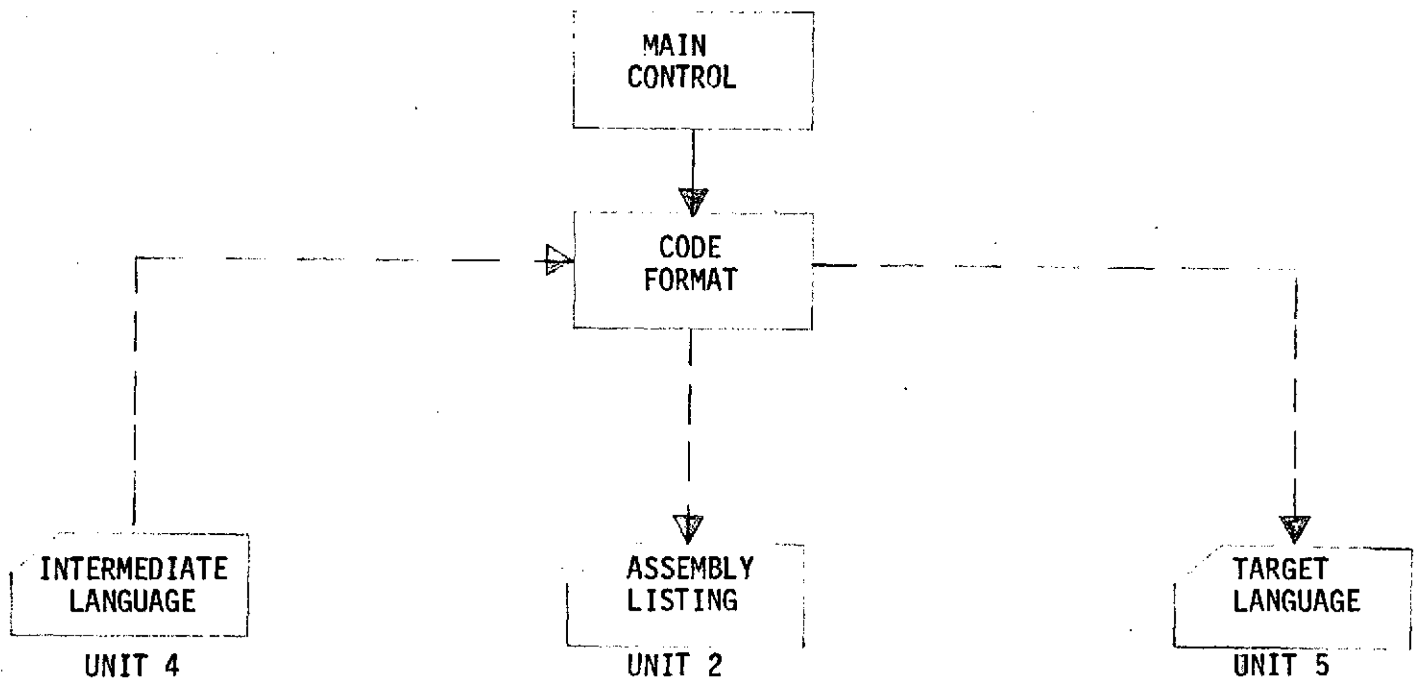


FIGURE 4 - OPERATION PROCESSOR PASS II BLOCK DIAGRAM

MODULE: Main Control

FUNCTION: Controls overall module execution.

PROCESS:

The Intermediate Language file (unit 4) is rewound and control is passed to the Code Format module to transform IL entries into Target Language.

MODULE: Code Format

FUNCTION: Transforms IL entries into Target Language.

PROCESS:

The module description and global symbol dictionary IL text is copied directly to the Target Language file (unit 5) since no modification is required.

The object text (TXT) records following contain the actual target instructions requiring possible modification. Each TXT item containing a symbolic reference to a program symbol is replaced by the relocatable or absolute address of the item as defined in the Symbol Table segment associated with the symbol. The modified object text is then passed on to the Target Language file as completed object.

If an assembly listing has been requested, a listing line is formatted and sent to the print file (unit 2). The operation mnemonic, assembled instruction value, operand name, etc... are derived from the IL text item.

The END IL item terminates execution and completes the compilation process.

4.2 Internal Data Structures

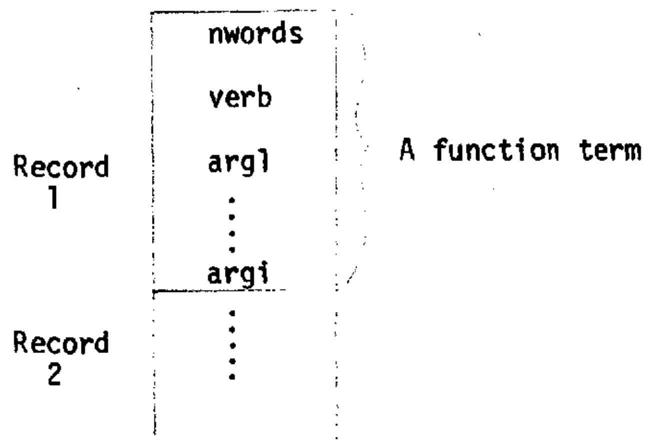
The only internal tables and arrays utilized are the IPROC array, containing the target description, and the various Symbol Table segments (tables QTABP, QTABS). The format and content of each are described in section 3.2.

5. EXTERNAL DATA STRUCTURES

The format of each data set file utilized by the C.W.S. system is described in this section. Each file corresponds to an intermediate language or library driving the execution of a processor. The formal definition of the structure of the corresponding language is described in section 1.

5.1 Function Language File Structure

The Function Language file contains a sequence of records, each describing a function term, with the following format:



nwords	--	the number of words in the record.
verb	--	the function verb number.
argk	--	the argument words.

5.2 Intermediate Language File Structure

The IL file represents an intermediate form of the Target Language. The format of the file contents are identical to that of the Target Language file described in the next section (5.3).

5.3 Target Language File Structure

Target code, or object text, is encoded on this file as a sequence of records, each record occupying 512* or less words:

Record Format

```
┌──────────┐
│ n        │
│ text1    │
│   .      │
│   .      │
│   .      │
│ textn-1  │
└──────────┘
```

n -- record word size (less or equal to 512*).

texti -- object text information as described in section 3, Volume I, Target Language.

The records are constructed so that the last text word ends an object text item description, i.e., no item description starts in one record and ends within another.

* - This record size is system-dependent.

5.4 Compiler Library File Structure

This file consists all target definition data as well as compiler initialization information.

A compiler initialization entry consists of a single record formatted as follows:

cid
qtabp-size
QTABP
qtabs-size
QTABS
qtable-size
QTABLE

cid -- compiler name in packed integer format (3 words).

qtabp-size -- the number of words for the following QTABP array (1 word).

QTABP -- the contents of array QTABP (qtabp-size words).

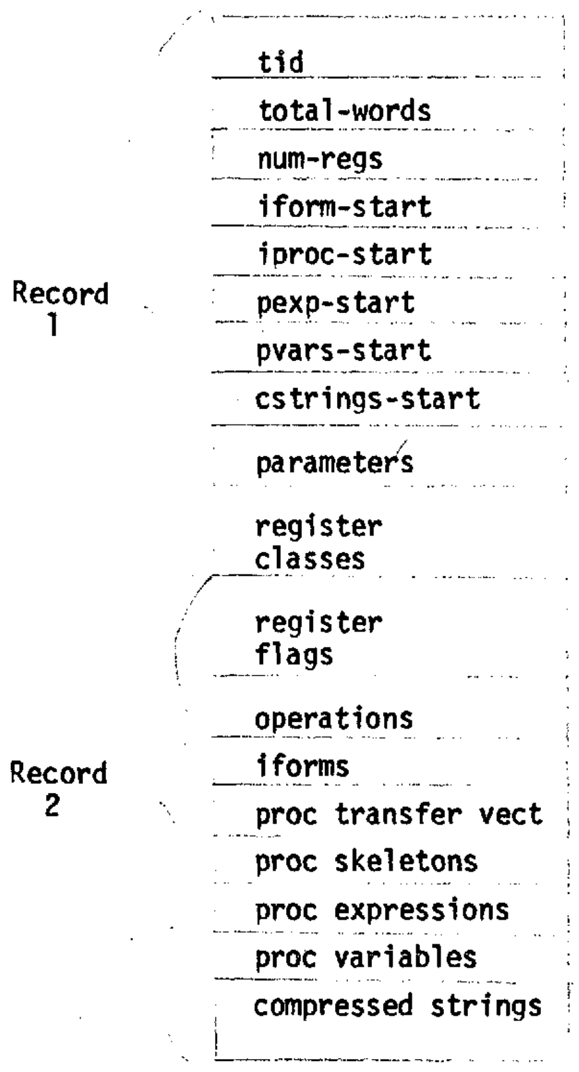
qtabs-size -- the number of words for the following QTABS array (1 word).

QTABS -- the contents of array QTABS (qtabs-size words).

qtable-size -- the number of words for the following QTABLE array (1 word).

QTABLE -- the contents of array QTABLE (qtable-size words).

A target definition entry consists of two successive records formatted as follows:



tid -- target identification name in packed integer format (3 words).

total-words -- total number of words in the entire target definition entry (both records).

num-regs -- the number of declared target registers.

iform-start -- starting word position of the instruction format descriptors.

iproc-start -- starting word position of the PROC skeletons.

pexp-start -- starting word position of the PROC expression operands.

pvars-start -- starting word position for the PROC variable storage area.

cstrings-start -- starting word position of the compressed string PROC operands.

parameters -- the target definition parameters (word size, addressing unit, etc...).

register classes -- the definition of all registers belonging to the 26 possible register classes (26 words).

register flags -- the status information for all target registers (num-regs *6 words).

operations -- the encoded target machine operations.

iforms -- the encoded instruction formats.

proc transfer vect -- the word starting positions for the definitions of each PROC.

proc skeletons -- the encoded PROCS definitions.

proc expressions -- the encoded PROC expression operands.

proc variables -- the storage area for PROC variables.

compressed strings -- the storage area for all packed integer strings representing literal PROC operands, operation names, etc...

he structure of each encoded data area is defined in Volume II, section 1.2, the escription of the internal data structures maintained by the Meta-Translator.

APPENDIX A: SYMBOL TABLE LAYOUT

The following describes the default attributes for the Symbol Table, which consists of eight segments. Each segment resides in a separate hash table and corresponds to a particular symbol type. The attributes are referred to by name, each name corresponding to a globally defined cell from which the attribute is packed or unpacked. The name descriptions are collectively defined at the end of the Appendix.

A.1 Segment 1 - Simple Variables

Attributes: IMODE,IFRAC,IRND,ISIGN,IRNGE,IRELOC,IPREC,IPKD,
ISMODE,IBIT,ILINK

A.2 Segment 2 - Arrays

Attributes: IMODE,IFRAC,IRND,ISIGN,IRNGE,IRELOC,IPREC,IPKD,
ISMODE,IBIT,ILINK,IDIM

A.3 Segment 3 - Tables

Attributes: IPAR,IRELOC,IRIG,INUMEN,IWDSEN,IPKD,ILINK

A.4 Segment 4 - Procedures

Attributes: IMODE,IRELOC,IREC,IREEN,IPTYPE,IPREC,ILINK,IBLOCK

A.5 Segment 5 - Global Block Names

Attributes: ISIZE,ILINK

A.6 Segment 6 - Switch Names

<u>Attributes</u>: ISIP,IBLOCK

A.7 Segment 7 - Labels

<u>Attributes</u>: IBLOCK, ILINK

A.8 Segment 8 - Constants

<u>Attributes</u>: IMODE,IEXP,IPREC

Attribute Name Definitions:

IMODE -- the computational mode of the item:

integer

fixed point

real (floating point)

complex

logical (masking)

boolean

texual

contexual (currently active mode is ISMODE)

status

location (contains an address)

binary (secondary mode only - see ISMODE)

octal (secondary mode only)

hex (secondary mode only)

IFRAC -- the signed fractional bit position for fixed point items.

IRND -- the rounding flag (0=no, 1=yes)

ISIGN-- the signed flag (0=unsigned, 1=signed item)

IRNGE-- the range flag (0=no range check, 1=range check)

IRELOC-- the item relocatability:

program relative (within the data section)

global block relative

absolute

externally defined

entry point

input dummy (subprogram parameter)

output dummy (subprogram parameter)

IPREC -- the precision of the item. Specifies the number of target bits to allocate for the item. For texual items, IPREC specifies the number of bytes to allocate.

IPKD -- the item packing density (for table items only):

none

medium

dense

tight

ISMODE -- the secondary mode of the item. For texual and contexual items, ISMODE defines the current computational mode (contexual) or conversion mode (texual) for the item.

IBIT -- the starting bit position of a table item within a host word.

ILINK -- the item link word to other symbols having a common property. The link is a symbol pointer containing both a type (table segment number) and a position. The meaning of ILINK for each symbol type is as follows:

(a) Simple Variables, Arrays, or Table Names

1. If within a global block, points to the next symbol within the block, or the block name if no more.
2. If equated to other symbols, points to the next symbol in the equivalence group.
3. If a table item, points to the table name.
4. If an argument to a procedure, points to the next procedure argument, or to the procedure name if no more.

(b) Procedure Names

Points to the first procedure argument.

(c) Global Block Names

Points to the first symbol in the block.

(d) Labels

Points to the next label (if any) which is also identified with the same block (IBLOCK) as this label.

IDIM -- the pointer to the dimension vector in QTABLE for an array item.

IPAR -- the parallel (=1) or serial (=0) flag for a table name.

IRIG -- the table rigidity flag for a table name (0=rigid).

INUMEN -- the number of table entries for a table name.

IWDSEN -- the number of words per entry for a table name.

IREC -- the recursive flag (0=no, 1=yes) for a procedure name.

IREEN -- the reentrant flag (0=no, 1=yes) for a procedure name.

IPTYPE -- the procedure type for a procedure name:

0 = main program

1 = subroutine

2 = function

3 = closed subroutine

4 = entry point

IBLOCK -- the pointer to a label associated with a program block (for procedures, switch names). For labels, contains the block number associated with the label.

ISIZE -- the computed size of a global data block.

ISIP -- the switch item name pointer for item switches.

IEXP -- signed binary exponent for constants. Indicates the IFRAC value for fixed point constants.