University of Pennsylvania
Department of Computer and Information Science
Moore School of E.lectrical Engineering
Philadelphia, Pennsylvania 19104

Technical Report

MODEL PROGRAM GENERATOR: SYSTEM AND PROGRAMMING DOCUMENTATION Fall 1980 version

by

A. Pnueli, K. Lu and N. Prywes

CABINET

Submitted to
Information System Program
Office of Naval Research
Under Contract N00014-76-0-0416

COMPUTER SCIENCE DEFT,
TECHNICAL REPORT FttM

PUM
80-38

Moore School Report

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MODEL PROGRAM GENERATOR: SYSTEM AND PROGRAMMING DOCUMENTATION Fall 1980 Version		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) A. Pnueli, K. Lu and N. Prywes		8. CONTRACT OR GRANT NUMBER(s) N00014-76-0-0416
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer and Information Science Moore School of Electrical Engineering University of Pennsylvania, Philadelphia, PA 19104		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 049-153
11. CONTROLLING OFFICE NAME AND ADDRESS Information Systems Program Office of Naval Research Arlington, VA 22217		12. REPORT DATE March 1980
		13. NUMBER OF PAGES 244
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution and Reproduction in Whole or in Part Permitted for Purposes of the USA Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

MODEL
Very High Level Languages
Program Generators
Compilers
Graph Analysis
Nonprocedural Languages
Data Description
Array Graphs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The MODEL system is a program generator. It accepts as input a specification of a program, in the nonprocedural Very High Level MODEL language, and produces a useable program in the PL/1 High Level language. The MODEL language has facilities essentially for: i) declaration of data structures of source (input) and target (interim and output) data; ii) equations which define target data. The MODEL system also performs extensive checking of the user specification and corrects the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entere

20. (continued)

specification in many cases of incompleteness, ambiguity, or inconsistency.

The MODEL system consists of four phases: individual statement syntax and semantic analysis, global semantic analysis, scheduling of program events and code generation. The global analysis is based on a graph representation of the specification denoted as Array Graph.

This report concerns the system organization and individual algorithms used in the Fall 1979 version of the MODEL system.

PREFACE AND ACKNOWLEDGEMENT

The MODEL version described in this report represents a revision and enhancement of previous versions. Its primary new capability is the accepting of equations where a dependent array variable depends on other elements in the same array. This capability is however supplemented by several other capabilities. The MODEL language has been supplemented with declaration of subscripts. Many checking and correction procedures have been added, particularly in the areas of dimensional analysis of array and in use of subscript. The scheduling and code generation phases are entirely new.

This report is organized similar to documentation of previous versions of the MODEL system and some parts have been incorporated in this report.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF TABLES

1 OVERVIEW

This document describes the algorithms and mechanisms of the MODEL Processor, which is a software system performing a program writing function. The MODEL Processor (hereafter called the Processor) has been designed to automate the program design, coding and debugging of software development, based on a non-procedural specifications of a program module in the MODEL language. As shown in Figure 1, a program module is formally described and specified in the MODEL language, whose statements are then submitted to the Processor. The set of MODEL statements describing a program module is referred to as a specification. The Processor, performs the analysis (including checking for the completeness and consistency of the entire specification), program module design (including generating a flowchart-like sequence of events for the module), and code generatior functions, thus replacing the tasks of an application programmer/coder. The Processor's capability to process a non-procedural specification language is built on application of graph theory to the analysis of such specifications and to the program generation task.

Another important function of the Processor is to interact with the specifier to indicate necessary supplements or changes to the submitted statements.

The Processor produces a complete PL/1* program ready for compilation as well as various reports concerning the specification

*Another version of the system produces COBOL code.

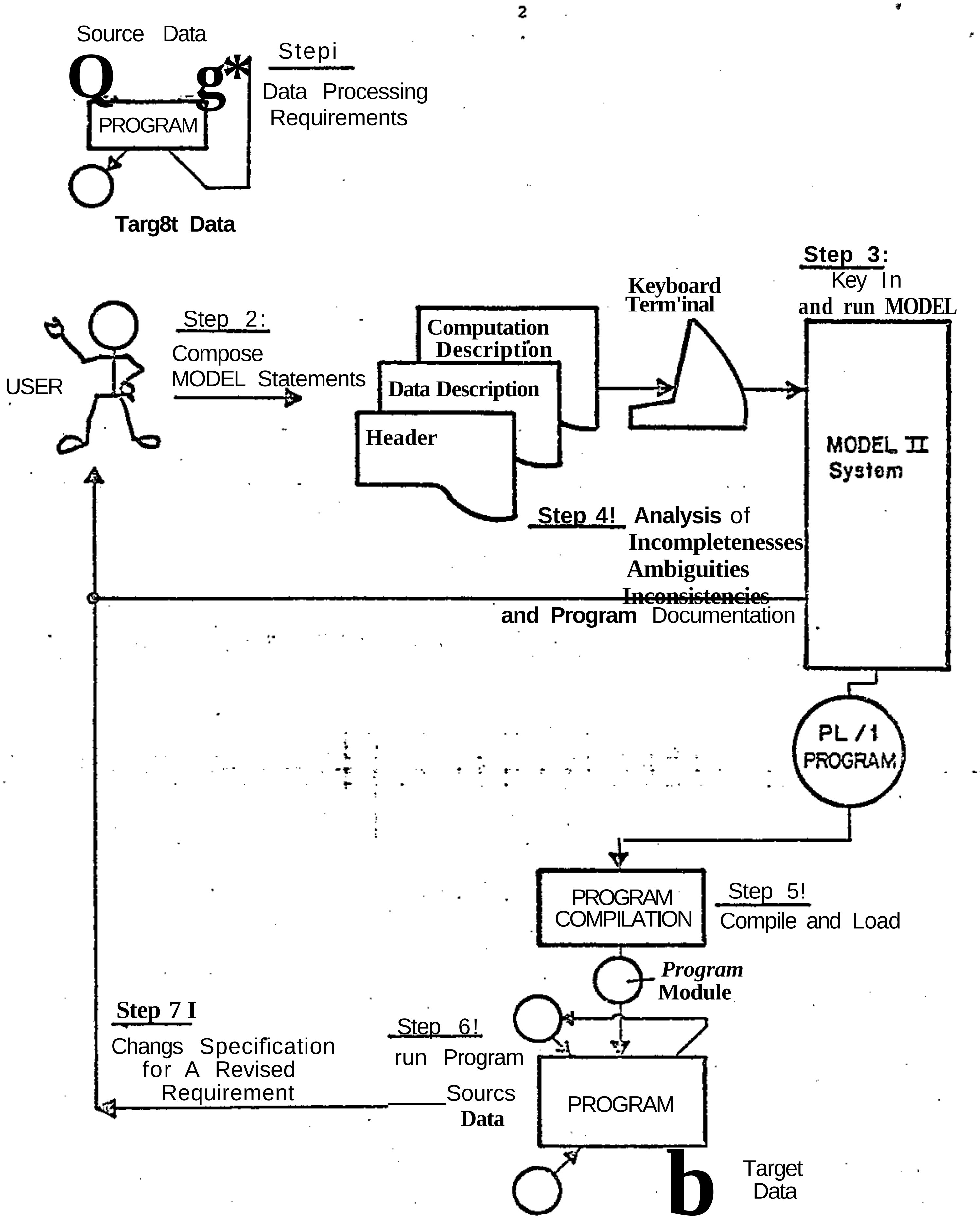


Figure 1 The Overall Procedure For Use of MODEL

and the generated program. The Processor output reports include a listing of the specification, a cross-reference report, subscript range report, a flowchart-like report of the generated program, and a listing of the generated program, all to be described fully later.

Processing of a specification written in MODEL by the Processor consists of four phases shown in the system flowchart of Figure 2, which is the first refinement of Figure 1. Some of these phases represent adaptations of known but state-of-the-art technology, while other phases involve more novel innovations in analysis of the specification and in the design and code-generation for the application program.

Each of the four phases depicted in Figure 2 is discussed below.

Phase 1: Syntax Analysis of the MODEL Module Specification

In this phase, the provided MODEL Specification is analyzed to find syntactic and some semantic errors. This phase of the Processor is itself generated automatically by a meta-processor called a Syntax Analysis Program Generator (SAPG), whose input is syntax rules provided through a formal description of the MODEL language in the EBNF language (yet to be discussed). In this manner, changes to the syntax of MODEL during development can be made more easily.

A further task of this phase is to store the statements in a simulated associative memory for ease in later search, analysis, and processing. Some needed corrections and warnings of possible errors are also produced in a report for the user. Also, a cross-reference report is produced.

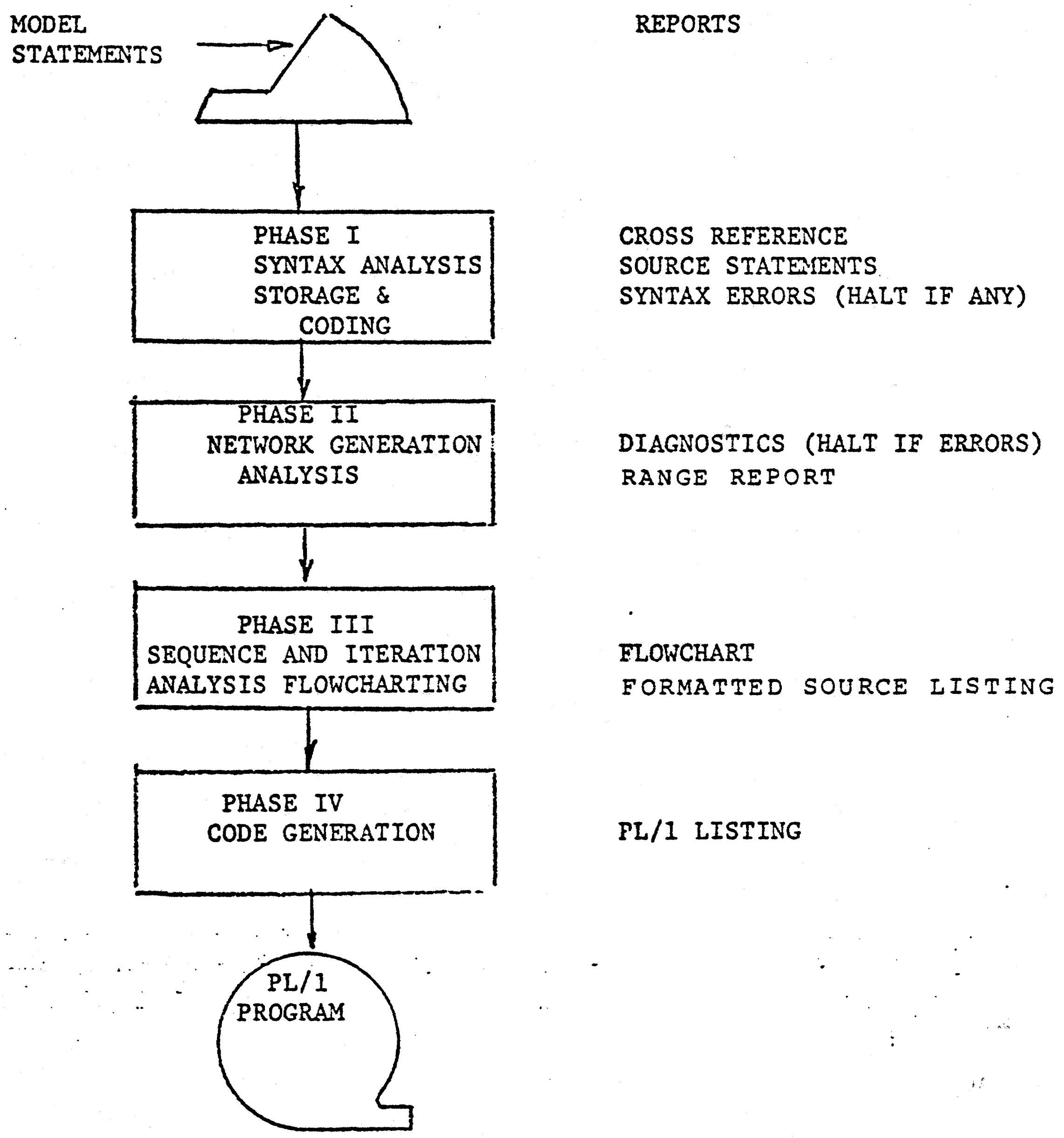


FIGURE 2 Phases of the MODEL II Processor

A description of the Syntax and Statement Analysis phase is covered in detail in Section 2.

Phase 2: Analysis of MODEL Specification

In this phase, precedence relationships between statements are determined from analysis of the MODEL data and assertion statements. The specification is analyzed to determine the consistency and completeness of the statements. Each MODEL statement may be considered to be an independent stand-alone statement. The order of the user's statements is of no consequence. However, in analysis of the statements, precedence relationships are determined based on statement components. These relationships are used to form the nodes and directed edges of an array graph (yet to be discussed) on which completeness, consistency, ambiguity, and feasibility of constructing a program can be checked. Various omissions or errors are corrected automatically, especially in connection with use of subscripts. Reports are produced for the user indicating the data, assertions, or decisions that have been inadequately described, assumptions that have been made by the Processor, or contradictions that have been found. In addition, a report showing the range of each subscript is generated.

Explanation of this process is covered in Section 3.

Phase 3: Automatic Program Design and Generation of Sequence and Control Logic.

This phase of the Processor determines the sequence of execution of all events and iterations implied by the specification, using graph theory techniques. It determines also the sequence and control logic of the desired program. The result of this phase is a flow of events, sequenced in the order of execution. Thus, the output of this phase is similar to a program flowchart of the desired program. It is subsequently used to produce a flowchart-like report* At the end of this phase it is also possible to produce a formatted report of the specification. This phase is presented in detail in Section 4.

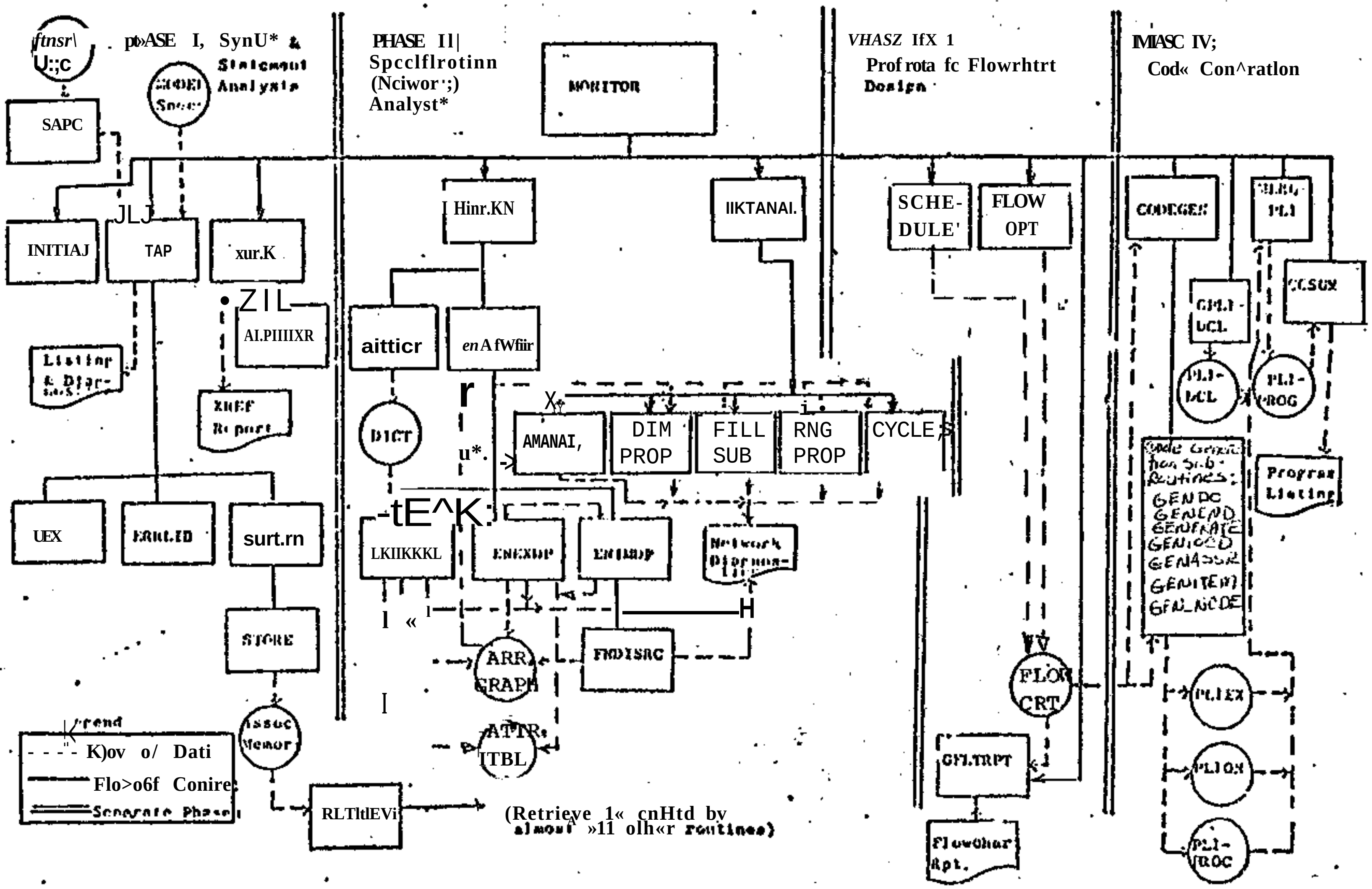
Phase .4 % Code Generation

At this point in the process it is necessary to generate, tailor, and insert the code into the entriea of the flowchart to produce the program. In particular, read and write Input/output commands are generated whenever the flowchart indicates the need for moving records. The assertions are developed into PL/I assignment statements. Wherever program iterations and other control structures are necessary, program code for them is generated. Declarations for object program data structures and variables are generated. Code is also generated for recovery from program failures when bad data is encountered during program execution. The product of this phase is a complete program in a high level language, PL/1, ready for compilation and execution. A listing of the generated program is produced.

The remainder of this report expands on the above phases. Figure 3 provides a tree diagram of the major modules, as well as the overlay structure of the Processor. The names of the modules in this diagram are referenced throughout the remainder of this report wherever the corresponding task is explained. As seen at the top of Figure 3, a MONITOR governs the execution of the different phases of the Processor, and does not allow succeeding phases to proceed without the success of the previous phases. At the second level of Figure 3, the major phases of the Processor are named (1) SAP (Syntax Analysis Program), Section 2; (2) NETGEN (Network Generation) & NETANAL (Network Analysis), Section 3; (3) SCHEDULE (Schedule events and generate flowchart), Section 4; and (4) CODEGEN (Code Generation), Section 5. Below this level of Figure 3, the diagram shows the names of the modules subordinate to each of these phases. Each of these subroutines is discussed at length throughout this report*Figure 3a provides an alphabetic index of the names of the major modules and the sections in which they are discussed.

In order to exemplify the nature of the Processor phases throughout the analysis, design, and program generation phases, a sample case problem is described in Section 3 and specified in MODEL. The processing of that sample problem is followed throughout the various phases for tutorial purposes.

Figure 3 Major Modules of the MODEL Processor



Major Modules	Section
ADD_TO_WHILE	5.5.3
AMANAL	3.3.12
BYTE_CALC	5.6.1
CGSUM	5.10
CHECK_VIRT	5.3
CODEGEN	5.5.1
CRADJMT	3.3.2
CREASIM	3.3.1
CRDICT	3.3.1
CRSVAR	5.3.2
CYCLES	3.3.13
DECLARE_STRUCTURE	5.8
DIMPROP	3.3.9
DOASS	3.3.6
DO_FLD	5.6.3
DO_GRP	5.6.3
DO_REC	5.6.1
DRAW_EDGES	3.3.9
ENEXDP	3.3.4,3.3.5
ENHRREL	3.3.3
ENIMDP	3.3.8
ENT_HIER_ADJ	3.3.3
ERRLIB	2.2.3
EXTEND_STRUCTURE	3.3.9
EXTRACT_COND	5.5.2
FIELDPK	5.4.1
FILLSUB	3.3.10
FLOWOPT	4.2
FNDISRC	3.3.6
GENASSR	5.5
GENDO	5.1.3
GENEND	5.1.4
GENERATE	5.1.2
GENIOCD	5.6
GENITEM	5.4
GEN_NODE	5.2
GFLTRPT	4.3
GPL1DCL	5.8
LEX	2.2.1
MERGPL1	5.9
NETANAL	3.3.9
NETGEN	3.3
OPTIMIZE_LIST	4.21
PRINT	5.5.3
RENUMBER	3.3.7

Figure 3a
Index of Major Modules

Major Modules	Section
RETRIEVE	2.3.5
RNGPROP	3.3.11
SAP	2.1
SAPG	2.1
SCAN	3.3.7,5.5
SCHEDULE	4.1
SCHEDULE_COMPONENT	4.1.2
SCHEDULE_GRAPH	4.1.1
SEARCHC	4.1.5
SIMPLE#	3.3.1
STORE	2.3.4
STRONG	4.1.5
SUPLIB	2.2.2,2.2.4,2.2.5
UNPACK	5.6.2
XREF	2.4
$	5.3.4

Figure 3a
Index of Major Modules

2. SYNTAX STATEMENT ANALYSIS

The first phase of the MODE! processor analyzes the syntax and other local semantics of indi^ idual statements. Advanced state-of-the-art syntax analysis techniques are used here which have proved to be invaluable. Specifically, the capability to generate the parser automatically has enabled rapid development changeso In addition to checking the MODEL statements for syntactic and $ome semantic errors, this phase also stores the statements in an internal associative form for later processing.

2.1 EBNF, SAPG, and the SAP

2>1.1 Specification of MODEL using EBNF and the SAPG.

The Syntax Analysis Program (SAP) for the MODEL statements is generated automatically by a Syntax Analysis Program Generator (SAEG)• As shown in Figure 4, the SAPG produces the Syntax Analysis Program (SAP) for analyzing MODEL statements, based on a specification of the MODEL language expressed in the EBNF/WSC (Extended Backus Normal Form with Subroutine Calls) meta language.

The EBNF/WSC includes the traditional concepts of BNF. BNF uses sequences of characters enclosed in angle-brackets < > called non-terqiinals to give names to grammatical units, for which substitutions may be made. It also uses sequences of characters not enclosed in brackets which are in the object language (in this case MODEL). BNF consists of a series of production rules or substitution rules of the form "As%*B"• "A" is a single non-terminal symbol and ^{tf}B" is one or more alternative sequences of terminal or non-terminal symbols that can be substituted for A.

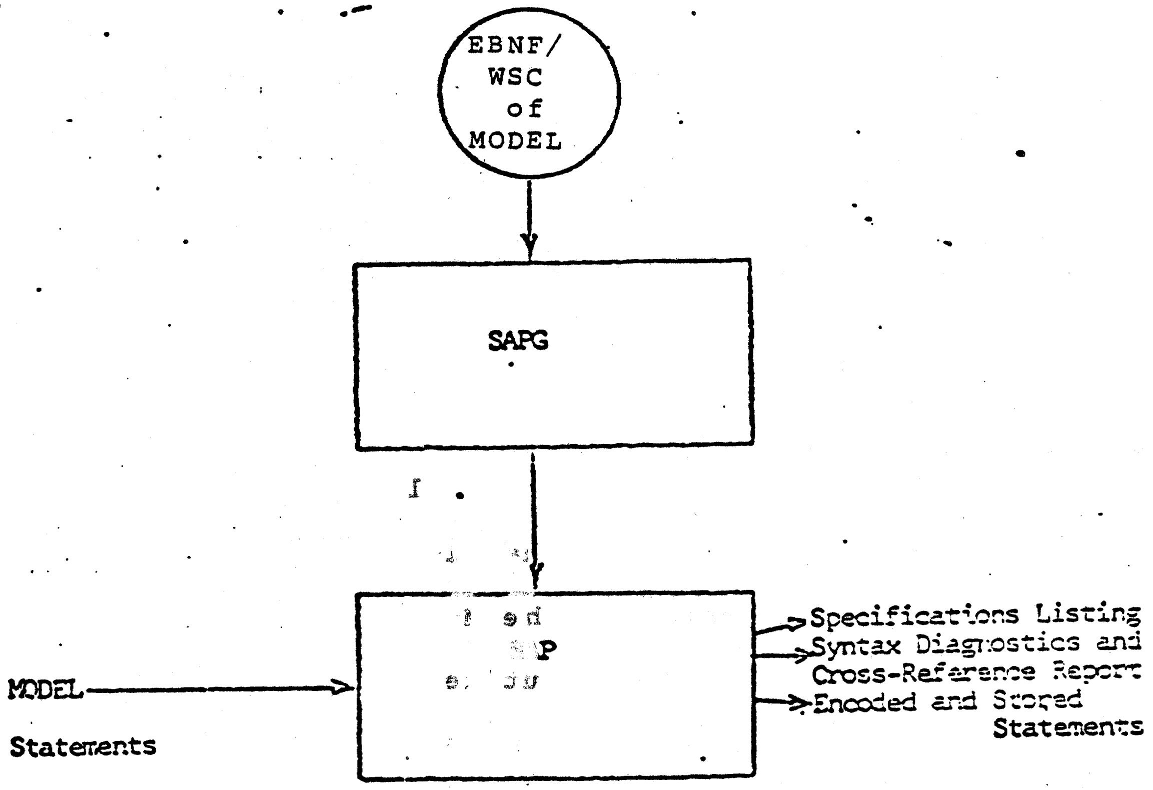


Figure 4

Block Diagram of SAPG and SAP

Th alternatives are separated by the meta-symbol "|". To facilitate language description, BNF was extended to EBNF with two more well-known meta-symbolss [ ] representing optionality and [ ]* representing zero or more repetitions.

The specification of MODEL that is input to the SAPG consists not only of the syntax specification of MODEL, but also of subroutine names embedded within the EBNF; therefore the name "EBNF with Subroutine Calls" (EBNF/WSC). The SAPG provides a capability to branch to these subroutines upon successful recognition of a syntactic unit. Thus, they can complete the SAP to enable it to check some of the statement semantics, to encode/ to produce error messages, and to store t KODEL statements for later retrieval. The invocations of these subro in >s are generated automatically by the SAPG, while the supporting subroutines themselves are written manually. The definition of the MODEL language in EBNF/WSC appears in Figure 5« The subroutines to be invoked are indicated between slashes (/•••/). Note thit subroutine calls are made after the successful recognition of syntactic units up to that point.

The SAP generated by the SAPG according to the EBNF/WSC is supplemented and linked with the routines. The SAP accepts statements in MODEL and checks them for syntactical correctness, and local semantics. It produces a listing of the statements, syntax diagnostics, an encoded stored version of the MODEL statements, syntax trees for the assertions and a cross-reference report.

```
<MODEL_SPECIFICATION>: := < "<KODE L_BOD Y_ST!*TS >  /CL*ERRF/  "{*
        /ST^T^FL/    <HOQEL_SPt.CIFICATION>~
<MOOeL_fcODY_STKTS>r:«  "  /UNRECS/
     MODULE  <MODULE.NA«£_ STM>
    ISOURC?  <SOURCE~FILtS_ST.M>
    |TARGfcT  <T4RGET~FILES~STrtT>
    |  3,'_ E\'D^  /ENDINP/
     I  <FILE_STMT>
     |  /A'SSINIT/  <ASSERTICNS>  /STRHS/
<ASSERTIONS>::=/£RKASS/<CONOITIONAL>  |
                    /SETTG//SVASSR/<SUB VARIAPLE>/SVCMP1/
                    <"<1S>/SWNAOP/"<<ODL  OH_KHS>
<C0NDIT10f*AL>::-IF  /SVAAS1/  /SV0P1/"/$£IBIT/  /SETSR/  /ER*BOOL/
                    <POOL£AN_EXPKF SSI CN>  /S\^C^P1/  /ERMTHEN/
                    TH£.«  /SVNXOP/  <S1J»PL£.ASSERT1ON>  /SVNXCMP/
                    i^ELSE  /SVNX-OPy  <ASSfeRT10N>  / S VfcX CMP/"{  /STALL/
<ASSFRTIO^>: :=  /SftirASS/  <COKDITIGMAt>  I  <SIhPL£_ASSERT2GN>
<DCL_OR_RHS>::=/INTODDL/<DATA DESC STMT>/FREETMP/
                    I  /ERMRHS/<I**TCAS>/SttSR/<300LEAN.£XPRESSI0N>/SVNXCMP/
                      /STALL/<ENDCHAR>
<INTCAS>::=/INTOASS/
<SiftPL£.aS$£RTICN>:;=/SETTG//SVASAE1  <SU^.VARIABLE>
                          /SVCMP1/  /ERffQ/  =  7SVN»)IOP/  /SETSR/
                          <6CCLfcAN^EXPStSSI0N>  /SVNXChP/  /STALL/  <ENDCHAR>
<SU8_VAR!ADLF>::= <EAChR>  /SVEACH/  /SETy/AR/  /AOLF*/  /STR.CON/
     "                 I  /EACHINT/  /SETSU3V/  <VAR>  /SVC"P1/  /SVT65R/
                    .    i"i/SV;,X(;P/  /I^CL^VL/  /:hTSIT/  /f RM &A C H/
                         <BG0L£AN^EXPRESSION>  /SV*\XCMP/  (  ,/SVNXQp/
                         <BGOLEkH_£XPkESSiON>/SVNXChP/"<-
                        '•'/ERKERF/)/DECIEVL/"{/STALL/
<bCCLEAN.EXPf?£SSION>::=  /VRb£1/  /SVBEXP/  <COND_EXP>  |
                              <BOOLEAN TERM>  /SVCMP1/
                               <"<oR>  /SVNXOP/  <800LCAN^TERM>  /SVNXCMP/"C*
                              /iTALL/
<COUD^EXP>::*IF  /SVCONO/  /CUN'DBL/  <&OOLE *:>*.£ X PRE S i IOW>  /SVCMP1/
                 /ThENCE/  THEN  /$VNXCP/  <fc00LE*N.EXPRESS IOK>  /SVNXCI^P/
                 /ELSECE/  ELSc  /SVNXCP/  <BOOLEJ»fc*EXF»P E$$ ION>  /SVNXCMP/  /STALL/
<0R>::=  /QR_REC/
<faOCL£AN.TER?->: :=  /wRfeTI/  ZiVoTI/  <E 0OL6 *N.F A CTOR >  /SVC*^P1/
        "               <"i  /SVNXOP/  <POOLEAN_FACTO&>  /SVNXC^P/"<*  /STALL/
<BOGL£AN^FACTOR>::=  /*R£E1/  /SVBF1/  <COMCAT£NATICN>  /SVCMP1/
        *                <"<R£LATION>  /SVNXCP/  <C0NCATcNATION>  /$VNXCtfP/"<*  /STALL/
<REL*TION>::*  /RFLREC/
<CONCATILSHTION>::-  /uRCCM/  /SVCCN/  <ARITH..EXP>  /SVC^PI/
                        <"  <CONCAT>  /SVNXOP/  <ARITh_EXP>  /SVNXC^P/"<*  /STALL/
<CCNCAT>::=  /CATR£C/
<ARITH_C-^P>J:=  /VRAE1/  /SVAt/  <"<SIGN>  /SVOPt/"<
       ~             <TERrt>  /SVC?*P1/  i"<OP?>>  /SVNXOP/  <TERM>  /SVNXCMP/"{*  /STALL/
 <TERf^>::=  /.RTERri/  /SVTEK>r/  <FACTOKS  /SVC^P1/
              {••<MOFS>  /sy^xcF/  <FACTGP>  /SVMXC^P/"<*  /STALL/
 <FACTGR>::=  /WRFAC.1/  /SVFAC/  C""*  /SVOP1/"{  <PRIftARY>  /SVCWP1/
                  <"<£XPON>  /SVNXOP/  <FRIfiARY>  /SVNXChP/"<*  /STALL/
<EXP0N>::s  /HXPR^C/
 <PRI^ARY>::=  /«RPRX.V1/  /SVPRIF/  <IS_PRI«>  /SVCwPt/  /STALL/
```

Figure 5: Definition of MODEL Language in EBNF/WSC

```
<IS_PRIM>::= ( <BOOLEAN_EXPRESSION> /ERMERP/ )
             | /SETNUM/ <NUMBER> /STNUM/ | <STRING_FORM>
             | <FUNCTION_CALL> | <SUB_VAPIABLE>
<STRING_FORM>::= ' /SETSTRN/ {" <STRING>  /SVSTRNG/"{
                 /ERMISS/
                 ' /ADLEX/ {"B /STBIT/ /BITERR/ <B_SUFX>"{ /STNUM/
<FUNCTION_CALL>::= <FUNCTION_NAME> /STFUN/
                   /SETFUNC/ {"(/SVNXOP/ <BOOLEAN_EXPRESSION>
                   /SVNXCMP/ {",/SVNXOP/ <BOOLEAN_EXPRESSION>
                   /SVNXCMP/ "{* ) "{ /STALL/
<FUNCTION_NAME>::= /FNCHECK/
<VAR>::= /SETVAR/ /INITGNM/ /GNMERR/ <NAME> /ADLEX/ /MKGNM/
         {". /ADLEX/ /GNMERR/ <NAME> /ADLEX/ /MKGNM/"{* /STR_CON/
<B_SUFX>::= /BITSTR/
<EACHR>::= /EACHRCC/
<GNAME>::= /INITGNM/ /GNMERR/ <NAME>  /MKGNM/
    {" .  /GNMERR/ <NAME>  /MKGNM/ "{ *
<STRING>::= <STRING_CONST>
<OPS>::= /OPREC/
<MOPS>::= /MOPREC/
<TEST>::= /TESTBIT/
<MODULE_NAME_STMT>::=        /MODUL1/: /MODUL2/ <NAME>
    /STMOD/ <ENDCHAR>
<SOURCE_FILES_STMT>::=        {"<FILE_KEYWORD>"{ /SRCFL1/ /INITSFL/ :
    <SOURCE_FILELIST>      /STSRC/ <ENDCHAR>
<FILE_KEYWORD>::=FILES|FILE
<SOURCE_FILELIST>::= /SRCFL2/ <NAME> /SVSRC/
     {", /SRCFL2/ <NAME> /SVSRC/"{*
<TARGET_FILES_STMT>::=        {"<FILE_KEYWORD>"{ /TARFL1/ /INITTFL/ :
    <TARGET_FILELIST>     /STTAR/ <ENDCHAR>
<TARGET_FILELIST>::= /TARFL2/ <NAME> /SVTAR/
    {", /TARFL2/ <NAME> /SVTAR/ "{*
<DATA_DESC_STMT>::=  <DATA_DESCRIPTION> <ENDCHAR>
<DATA_DESCRIPTION>::=
  <FILE> /SVFILE/ /FILERR1/  <FILE_DESC> /STFILE/ <STORAGE_DESC> /STDEV/
  |<RECORD_STMT>
  |<GRP_FLD>
  |<INT_STMT>
  |<SUB_STMT>
<GRP_FLD>::=<GROUP_STMT> | <FIELD_STMT>
<SUB_STMT>::=<SUBSCRIPT>/MEMINIT/ /SVMEM/ {"( <OCCSPEC> )"{ /STSUBST/
<SUBSCRIPT>::= SUB | SUBSCRIPT
<FILE>::= FILE|REPORT
<RECORD_STMT>::= <RECORD> /MEMINIT/ {"("{ <ITEM_LIST> {")"{
   /STREC/
<RECORD>  ::=  REC | RECORD
<ITEM_LIST>::= /ITEMC1/<ITEM> {"{","{ <ITEM>"{*
<ITEM>::=<NAME> /SVMEM / {" . <NAME> /SVMEM/ "{* {"("{ {"<OCCSPEC>"{{")"
<OCCSPEC>::=  <STAR> /SVSTAR/
                 |              <MINOCC>/SVMNOC/ {"<MAXOCC>"{
<STAR>::= /STARREC/
<MINOCC>::=<INTEGER>
<MAXOCC>  ::=  {":/ITEMER2/"{<INTEGER> /SVMXOC/ /CKMNMX/
              |            <INTEGER> /SVMXOC/ /CKMNMX/
```

Figure 5 (continued)

```
109     <GROUP_STMT>::= <GROUP>/MEMINIT/ {"("{ <ITEM_LIST> {")"{ {","{
110                 {" TABULATED /SVTAB/ "{    /STGRP/
111     <GROUP>  ::=  GRP | GROUP
112     <FIELD_STMT>::=  <FIELD> /SVFLD/ <FIELD_ATTR>  /STFLD/
113     <FIELD>  ::=  FLD | FIELD
114     <FIELD_ATTR>::= {"("{  <TYPE> /SVFDTP2/{" <LENG_SPEC>"{
115        {","{  {"<LINE_SPEC>"{  {","{  {"<COL_SPEC>"{  {")"{
116     <LENG_SPEC>  ::= ( /FLDERR4/
117          <MIN_LENGTH> {" <MAX_LENGTH> "{ /FLDERR5/    )
118                        |  <MIN_LENGTH> {"<MAX_LENGTH>"{
119     <MIN_LENGTH>  ::=                <INTEGER> /SVMNFLN/
120     <LINE_SPEC>::= LINE /LCERR/ (<INTEGER> /SVLINE/)
121     <COL_SPEC>::= COL /LCERR/ (<INTEGER> /SVCOL/)
122     <TYPE>::= /FLDERR3/ <PIC_DESC>
123         | <STRING_SPEC>  | <NUM_SPEC>
124     <PIC_DESC>::= <PIC_TYPE> /PICERR1/ /SVPIC/
125                 ' {" <STRING> /SVPICST/ "{ ' /STPIC/
126     <PIC_TYPE>::= PIC | PICTURE
127     <STRING_SPEC>::= <STRING_TYPE> /S' TRTP/
128     <STRING_TYPE>::= CHAR | CHARACTER   BIT | NUM | NUMERIC
129     <NUM_SPEC>::= <NUM_TYPE> /SVNUMTP/ {" <FIXFLT> /SVMOD/ "{
130     <NUM_TYPE>::= BIN | BINARY | DEC | DECIMAL
131     <FIXFLT>::= FIX | FIXED | FL | FLOAT | FLT
132     <MAX_LENGTH>::= {":"{ <INTEGER> /SVMXFLN/
133              | , /FLDERR2/ <SINTGR> /SVSCALE/
134            |                <INTEGER> /SVMXFLN/
135     <SINTGR>::= - /INTERR/ <INTEGER> /NEGATE/ | <INTEGER>
136     <NUMBER>::= <INITNUM> /NUMERR/ <RECNUM>
137     <RECNUM>::= /RECNUM/
138     <INITNUM>::= /INITNUM/
139     <SIGN>::= + | -
140     <RECG>::= RECORD | GROUP
141     <KEY>::=KEY|SEQUENCE
142     <CODE>::=EBCDIC|BCD|ASCII
143     <ANY>::= <NAME>|<INTEGER>
144     <NO_TRKS>::= 7|9
145     <DENSITY>::= 200|556|800|1600|6250
146     <PARITY>::= ODD|EVEN
147     <TYPEDSK>::= 2314|2311|3330|2305 | 3330-1
148     <ORG>::=ORG|ORGANIZATION
149     <ORG_TYPE>::= /DSKER2/ISAM|SEQUENTIAL|SAM|INDEXED_SEQUENTIAL
150     <INT_STMT>::= <INTERIM> /SETINTR/ <GRP_FLD>
151     <INTERIM>::= INT|INTERIM
152     <ENDCHAR>::= /SEMI/ <END_CHAR> /STMTINC/
153     <END_CHAR>::= /SVENDC/
154     <STRING_CONST>::=/CHARSTR/
155     <NAME>::=/NAMEREC/
156     <INTEGER>::=/INTREC/
157     <IS>::= IS | =
158     <FILE_STMT>  ::=  FILE {"NAME"{ {"<IS>"{ /MEDER1/ <NAME>/SVFLNM/
159                                <FILE_DESC>/STFILE/
160                           .      <STORAGE_DESC> /STDEV/ <ENDCHAR>
161     <FILE_DESC>  ::= <RECG> {"NAME"{ {"<IS>"{ /FILERR3/<NAME>/SVRCNM/
162                                     {"{"("{<STAR>/SVSTARF/{")"{"{
```

Figure 5 (continued)

```
                            ("STORAGE <"NAMFft< {"<1S>"( /FILERR5/ <NAME> /SVSTNK/M(
                            (M<KtY> (#'NAMEff( {"<IS>"{ /FILERR6/ <NAME> /SVKEY/M(
                            (M<OfcG> {*•<!$>••{ <ORG_TYPE> /SVONG3/M(
       <STORAGEwDESC>   22= {•VDEVICt (M<IS>-i( <DEVICE>lH /SVDEV/
                  {"RECORD /MEDER2/"L{"FORMAT {"<IS>"{ <REC_FMT>"{/SVRECF/
                  <P_LK._RtCwVOL>
                  {M<TAPt_DESC>w{ (ii<DISK.DFSC>*i{
                  {fiHARDWARE"(  {"SOFTWA H Eff{                    %'
        <DEVIC£>   22- /hfOERt/  TAPE ! DISK/SETDEVB/
            . |   I^HD /SETDtVC/     | PRINTER /StTDFVP/
               j PUNCH /SETDEVU/     | IERI^IMAL /SFTDEVI/   ^
       <REC^f«T>   22= /RCFER1/ F1XED|VAR1AbLEIVAR_SPANNED lUNDEFIUED
       <bLK REC VOL>   ::=
             ~    "(*• (fiMAXM{ /KCFER2/ BLOCKS17E (ff<IS>"{ <1NTEGER> /SVBLK/ tf{
                  (M {MMAX/MtDfcR4/w( RECORDSIZE {"<1S>M{ /RCF ER 3/<INT EGER >/S VRCSZ /•*(
               ("VOLUME- {itNAHEM( {M<1S>°( /KEDER5/ <NA-HE >/S VVOL/ {§if/MEDER5/<NAME>ii{* "iL
       <TAPE^DESC>   s:=  {"<TRACKS> {••<!$>"{ /P ARF HR/<NO_TR KS >/S VTR K2/ fl(
                             ("PARITY (ff<I S>-§( /PARFRR/ <PAR ITY >/ SVP AR2 / " (
                             ("DENSITY ("<I S>#-( /PARERk'/ <OFI«SITY> /SVDEN2/M(
                           ' {•• ••{fiTAFEii( LAPEL fM<IS>9f( <LAPfcL_TYPE>/ SVLA82/fi(
                             ("START {MFIL£fi< {M<IS>f#( /PARFKRA <1NTEGER> /SVSTFL2/M(
                            {il(MCHAKM( CODE <-i<IS>-6t <CCDE> /SVCC/ fi(
       <TRACICS>   22=   NO_TRK$  I  TRACKS
       <LAbEL_TYPE>   2 S=*/?1ED ER3/ ID«_SIDJAMSI _STD |NUNE f BYP AS S
      <D1SK^DE5C>   :2=  ("UNIT (if<IS>MC /D5KER4/ <TYPEDSK> /SVUNIT2/°(
                             ("<C Y^IUDtRSV/SVUC YL / {#l<IS>"{ /PARERR/ <INTE6ER> /SVQTY2/W(
       <CYLINDL^S>   ;:=  NO^CYLS  I  CYLINDERS
       <HARDUARf>22=  ( "{MC OMPUTE R'H >tODFL  (M<IS>>§( <kUV>
       <SOFTwANF>22=  (M(#iOPEKATINGM<  SYSTEM  {l{l<IS>ifi{ <ANY>"{
```

Figure 5s Continued

```
1EXTERNAL FUNCTIONS AND/OR SUBROUTINES
 CLREPRF   STMT_FL   UNRECS    ENDINP    ASSINIT   STRHS     ERMASS    SETTG     SVASSR    SVCMP1
 SVNXOP    SVAAS1    SVOP1     SETBIT    SETSR     ERMBOOL   ERMTHEN   SVNXCMP   STALL     INTODDL
 FREETMP   ERMRHS    INTOASS   SVASAE1   ERMEQ     SVEACH    SETVAR    ADLEX     STR_CON   EACHINT
 SETSUBV   SVTGSR    INCLEVL   ERMEACH   ERMERP    DECLEVL   WRBE1     SVBEXP    SVCOND    CONDBL
 THENCE    ELSECL    OR_REC    WRBT1     SVBT1     SVBF1     RELREC    WRCON1    SVCON     CATREC
 WRAE1     SVAE      WRTERM1   SVTERM    WRFAC1    SVFAC     EXPREC    WRPRIM1   SVPRIM    SETNUM
 STNUM     SETSTRN   SVSTRNG   ERMISS    STBIT     BITERR    STFUN     SETFUNC   FNCHECK   INITQNM
 QNMERR    MKQNM     BITSTR    EACHREC   CPREC     MOPREC    TESTBIT   MODUL1    MODUL2    STMOD
 SRCFL1    INITSFL   STSRC     SRCFL2    SVSRC     TARFL1    INITTFL   STTAR     TARFL2    SVTAR
 SVFILE    FILERR1   STFILE    STDEV     MEMINIT   SVMEM     STSUBST   STREC     ITEM01    SVSTAR
 SVMNOC    STARREC   ITEMER2   SVMXOC    CKMNMX    SVTAB     STGRP     SVFLD     STFLD     SVFDTP2
 FLDERR4   FLDERR5   SVMNFLN   LCERR     SVLINE    SVCOL     FLDERR3   PICERR1   SVPIC     SVPICST
 STPIC     SVSTRTP   SVNUMTP   SVMOD     SVMXFLN   FLDERR2   SVSCALE   INTERR    NEGATE    NUMERR
 RECNUM    INITNUM   DSKER2    SETINTR   SEMI      STMTINC   SVENDC    CHARSTR   NAMEREC   INTREC
 MEDER1    SVFLNM    FILERR3   SVRCNM    SVSTARF   FILERR5   SVSTNM    FILERR6   SVKEY     SVORG3
 SVDEV     MEDER2    SVRECF    MEDER6    SETDEVB   SETDEVC   SETDEVP   SETDEVU   SETDEVT   RCFER1
 RCFER2    SVBLK     MEDER4    RCFER3    SVRCSZ    MEDER5    SVVOL     PARERR    SVTRK2    SVPAR2
 SVDEN2    SVLAB2    SVSTFL2   SVCC      MEDER3    DSKER4    SVUNIT2   SVUCYL    SVQTY2
1RECURSIVE PRODUCTIONS
 MODEL_SPECIFICATION
 CONDITIONAL
 ASSERTION
 SUB_VARIABLE
 BOOLEAN_EXPRESSION
 COND_EXP
 BOOLEAN_TERM
 BOOLEAN_FACTOR
 CONCATENATION
 ARITH_EXP
 TERM
 FACTOR
 PRIMARY
 IS_PRIM
 FUNCTION_CALL
```

Figure 5: Continued

2.1.2 How the SAPG Produces the SAP

The SAPG is a small compiler in itself in that it processes a specification in the language EBNF/WSC and produces a program (SAP). It performs this in three passes over the set of productions.

In pass 1, each production is scanned, and its components are encoded into a set of tables. Non-terminal symbols appearing on the left-hand-side of a production (new production names) are put into a symbol table, while non-terminals appearing on the right-hand-side of a production are put into a work table. Terminal symbols in a production are put into a terminal symbol table. Subroutine calls are put into yet another table.

In pass 2, the symbolic references in the work table (i.e., non-terminals on the right-hand-side of the original production) are resolved. Pass 2 checks that each right-hand-side non-terminal symbol in the work table is defined, and links it to the corresponding entry in the symbol table. Undefined non-terminals as well as circularly-defined non-terminals can be detected in these table searches.

Pass 3 of the SAPG is the code-generation phase that produces the SAP in PL/1. It is only entered if no errors were encountered in the previous phases. For each EBNF/WSC production, a PL/1 procedure is generated. Each one returns a bit: 1 if the recognition was successful; 0 if it was unsuccessful. The exclusive nature of EBNF production rules and alternatives is effected by generating nested PL/1 IF-THEN-ELSE statements.

Repetition zero or more times is effected by generating a GO TO to the statement testing for recognition. Subroutine names embedded in the EBNF/WSC get a CALL generated for them in place. Calls to other subroutines not explicit in the EBNF/WSC are also generated. These include "housekeeping" subroutines of the SAPG and calls to LEX, a subroutine to scan and return the next token in the object language.

To illustrate the code that the SAPG generates, consider the following representative production rule in the EBNF/WSC and the PL/1 code that corresponds:

```
<FIELD_STMT> ::=FIELD /SVFLD/<FIELD_ATTR> /STFLD/
```

The PL/1 code that is generated for it by the third pass of the SAPG would be the following:

```
FIELD_STMT:  PROCEDURE RETURNS(BIT(1));
CALL $MARK;
CALL LEX;
IF LEXBUFF='FIELD' THEN DO;
CALL LEXENAB;
CALL $POPF;
CALL SVFLD;
IF FIELD_ATTR THEN DO;
IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
CALL STFLD;
CALL $SUCCES; RETURN('1'B);END;
ELSE DO; CALL $SUCCES; RETURN('1'B); END;
END;
ELSE DO; CALL $FAIL; RETURN('0'B); END
END FIELD_STMT;
```

The above code generated by the SAPG would become one procedure in the SAP. Note that the names that the language definer uses in the production rule ar*e preserved in the generated SAP code* The subroutines beginning with dollar signs ($) are "housekeeping" routines that are internal to the mechanisms of SAPG-generated code.

2»2 Supporting Subroutines for BBNF of MODEL

A refined system flowchart of the SAPG and SAP showing the types of supporting routines appears in Figure 6. The manually-written syntactical supporting routines are of one of several types*i*

(1) a lexical analyzer which returns tokens of syntactic units to the SAP for analysis,

(2) statement semantics checking routines;

(3) error message handling routines;

(4) encoding routines to compact information for further efficient processing; and

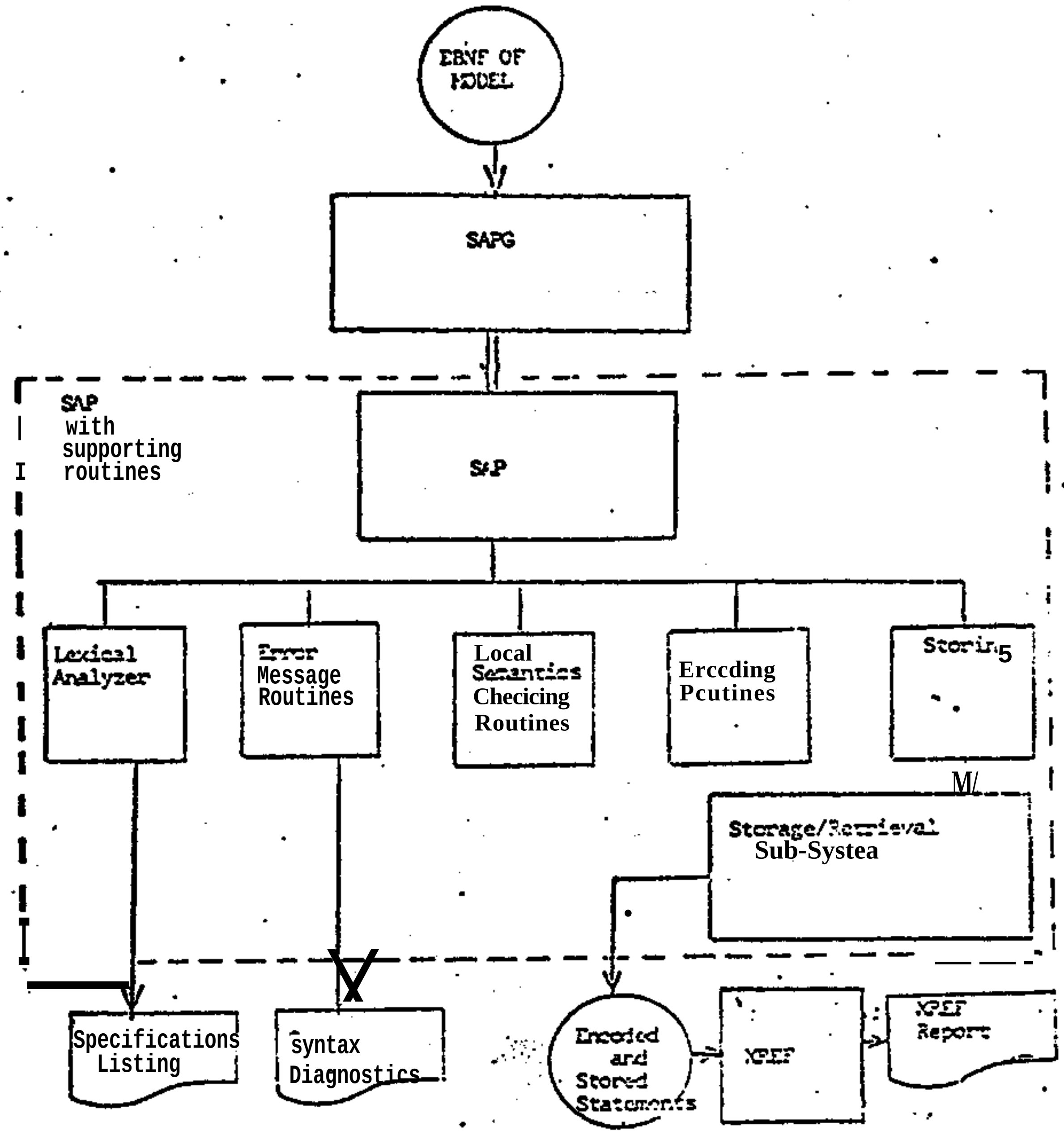
(.5) statement storage routines.

The cross-reference report produced during this phase is generated by a manually-written program (XREF) and is described in Section 2.7.

A discussion on how to decide where to insert subroutines as well as a tabular summary of all routines used appears in Section 2_e3 .

~~2.2«>1 The Lexical Analyzer~~

The purpose of the lexical analyzer is to scan for syntactic units or "tokens", using such delimeters as blanks and certain punctuation marks, and to return tokens to the Syntax Analysis,



FUure 6

More Oetullccf View Of SAPG end SAP V/!ch
• Suppcrtlnc; Subroutines

Program (SAP) for syntactic checking* The automatically-generated SAP calls upon the lexical analyzer (LEX) whenever it needs the next token. The lexical analyzer is based on the finite state machine concept. Each state of the machine corresponds to a condition in the lexical processing of a character string. At each state, a character is read, an action is taken based on the character read (such as concatenating the current character to previous ones or returning the entire token to the SAP), and the machine changes to a new state. The character classes for the MODEL Language, for the purposes of lexical analysis, appear in Table 1. These classes divide the entire character set into categories such as illegal characters , delimeters, "normal" characters, etc. The state transition matrix for the MODEL language appears in Table 2_e The rows of the matrix represent the character classes of the previous character, while the columns represent those of the current character. The entries in the matrix indicate the action to be taken and the next state. The action taken in each state is summarized in Table 3. The actions involve such steps as concatenating of a character, ignoring a character, detecting an illegal character, returning a complete token to the SAP, etc., and setting a "next state".

2.2,2 Statement Semantic Analysis

Some of the semantics of the specification statements can be checked during the syntax analysis phase. Such routines can check that a range or condition on a syntactic unit is locally correct. These routines do not and cannot check the overall consistency, completeness, or correctness of the logic of the MODEL specification, a task which is performed by a later phase

Class	Character Set	Explanation
0	A B ... Y Z _ # @	Characters in names
1	(space)	Delimeter
2	0 1 2 ... 9	Numerals
3	. (+ &) ; , % : ' ''	Delimeters in various contexts
4		
5	<	Delim in logical expr.
6	|	"OR" symbol
7	*	Mult. Or comment if with "/*"
8	~	"NOT" symbol
9	-	minus symbol
10	/	Division or comment if with "/*"
11	>	Delim in logical expression
12	=	Delim for keywords & log. Expr.
13	all others	Illegal

Table 1

Character Classes for MODEL Language

Character Class (next) / (current)	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	2	1	2	2	2	2	2	2	2	2	2	2	7
1	1	3	1	5	1	1	1	1	1	1	1	1	1	7
2	1	2	1	2	1	2	2	2	2	2	2	2	2	7
3	2	2	2	2	2	2	2	2	2	2	2	2	2	7
4	2	2	1	2	2	2	2	2	2	2	2	2	2	7
5	2	2	2	2	2	2	2	2	[illegible]	2	2	2	1	7
6	2	2	2	2	2	2	1	2	2	2	2	2	2	7
7	2	2	2	2	2	2	2	[illegible]	2	2	2	2	2	7
8	2	2	2	2	2	1	2	2	2	2	2	1	1	7
9	2	2	2	2	[illegible]	2	2	2	2	2	2	1	2	7
10	2	2	2	2	2	2	2	6	2	2	2	2	2	7
11	2	2	2	2	2	2	2	2	2	2	2	2	1	7
12	2	2	2	2	2	2	2	2	2	2	2	2	2	7
13	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Table 2

State Transition Matrix for MODEL Lexical Analyzer

Action 1:	Concatenate next character to current token
Action 2:	End word with next character
Action 3:	Skips blanks sequence
Action 4:	Reserved (never taken)
Action 5:	Scan forward one character and save as token
Action 6:	Comment bracket; scan to end of comment
Action 7:	Illegal character(s); print error message

Table 3

Lexical Analysis Actions

of the Processor. An example of a local semantics checking routine is one which checks the range of a numeric computation. For instance, if a group is said to occur n to m times, a subroutine exists to check the 0 < = n < m < 32768. These manually-written routines are invoked automatically by the SAP by virtue of their specification in the EBNF/WSC of the MODEL language for the SAPG. The semantic checking routines are listed in Table 4.

2.2.3 Error Message Stacking Rout[illegible]

These are subroutines which [illegible] error diagnostics to print out upon recognition of a syntac[illegible]-incorrect user statement. Upon reaching incorrect syntactic [illegible], the automatically generated SAP does not print its own messages, but expects the corresponding diagnostics to be on an "error stack". For this purpose, subroutines have to be written to give a MODEL user effective information when statements have been incorrectly composed. Specifically, an error message has to be stacked for each expected terminal symbol in the MODEL language in case the token is missing or incorrect. If the expected token is found, the SAP simply pops the corresponding error message and continues; if the expected token is missing or incorrect, the SAP pops the corresponding error message, prints the statement number and message, scans for the end of the statement delimeter (;), and continues. The routines that stack such error message codes are the ones ending the letters "ER" or "ERR. (e.g. RECERR). Each routine's syntax error message pinpoints the token that is incorrect, missing, unexpected, or misspelled.

Table 4

Semantics Checking Routines

(Inserted in the EBNF/WSG after the token(s) to be checked or for other action)

NAME	WHAT IT DOES
ASSINIT	Initializes number of sources/targets to assertion
CATREC	Recognize the operator *|J^{f}•
BITSTR	Check that an alleged bit string contains only the digits, 0, 1
CKMNMX	Checks proper range for minimum and max!
EACHINT	Initializes flag for FOREACH existence
EACHREC	Recognizes FOREACH phrase
EXPREC	Recognizes the operator **
FNCHECK	Check that a candidate name is a recognized function name
FSASS	Prints frame before first assertion
FRUITER	Prints frame before interfile relationship
FRINTRM	Prints frame before interims
GETLIB	Gets input from library
INITQNM	Initializes number components to qualified name
INITSFL	Initializes source file list
INITTFL	Initializes target file list
INTOASS	Returns 1 if the currently scanned statement is an assertion and not a data description statement
INTODDL	Records that the statement scanned is a data description statement

Table 4:

Semantics Checking Routines (continued)

NAME	WHAT IT DOES
INTR[illegible]	Recognizes integers
MEM[illegible]	Initializes number of members of record or group
MKQNM	Concatenates qualified name components
MOPREC	Recognizes a multiplication operation, i.e. '*' or '/'
NAMEREC	Name recognizer; checks not keywords
OPREC	Recognizer for the operators '+', '-'.
OR_REC	Recognizes the alternation operator '|'
RECNUM	Recognizes and scans a number
RELREC	Recognizes any of the relations <,>, > =, <= , =, >, <, =.
RESETBT	Clears a flag which if set signify an assertion is being scanned
SETBIT	Used to set and reset a bit that indicate whether the statement is an assertion or a data description statement.
SETSR	Sets a flag which signals that the right hand side of an assertion is being scanned. This will cause all names encountered to be added to the "source" list for the assertion
SETTG	Signals that the left hand side of an assertion is being scanned
STARREC	Recognizes a '*' for indefinite repetition
SVASSS	Saves the actual assertion itself during the scanning of a statement
SVENDC	Recognizes a ';' as an end of statement character

One product of the syntax analysis phase is the Error Diagnostics Report containing the messages. Each message gives the diagnostics provided by the error routine and provides the exact location of the ferror so that it can be corrected and resubxaitted by the user easily. If no syntax errors are found during the syntax analysis phase, a message is sent thai: "NO ERRORS OR WARNINGS DETECTED", and the Processor proceeds to the next phase. But, if error diagnostics were produced, a flag is set to diable continuation of analysis and design beyond the syntax checking phase.

The error aespages and stacking routines are listed in Table 5.

2.2.4 Encoding User Statements

These supporting routines encode some of the MODEL specification into an internal representation. Although all of the names provided by the user specification are kept intact in internal form for use by the object program, many of the descriptions and attributes are encoded for more compact and efficient processing later. For example, the description in a FIELD statement enters an internal table where the type of field is encoded CO for character, 1 for binary, 2 for numeric, etc.), and the field length type is encoded 0 for fixed length, 1 for variable length). One encoding routine is written for each such statement type* Each routine is invoked automatically after recognition of the syntactic unit by the SAP. The invocation is automatically generated as part of the SAP by the SAPG by virtue of its specification in the EBNF/WSC. The internal format of the tables is given in the next section in conjunction with the discussion of the internal associative

Table 5

ERROR MESSAGE STACKING ROUTINES

NAME	CODE	ERROR MESSAGE
BITERR	BITERR	A bit string contains characters other than 0 or 1#
CONDBL	CONDBL	Missing boolean expression in conditional expression
DSKER1	DISK01	Left pa'~en missing in DISK atem nt
	DISK02	Rig'.t pa-en missing in DISK st. ame t
DSKER2	D2SK03	*Orga *xiza* ion type missing or illegal in DISK statement
DSKER3	DISK04	Internal name missing or illegal in DISK statement
DSKER4	DISK05	Type disk missing or illegal in DISK statement
DSKER5	DISK06	Left par*;n missing in SPACE spec in DISK statement
	DISK07	Right paren missing in SPACE spec in DISK statement
DSKER6	DISK08	Units missing or illegal in DISK statement SPACE spec
	DISK09	Comma missing after units in DISK statement SPACE spec
	DISK10	Quantity missing or illegal in DISK statement SPACE spec
ELSECE	ELSEECE	Missing keyword THEN in conditional expression
ERMASS	ERMASS	Assertion missing after the keyword THEN.
ERMBOOL	ERBO0L	No Boolean expression after the keyword IF.
ERMEACH	EREAdH	No expression after the keyword '('

Table 5

ERROR MESSAGE STACKING ROUTINES (continued)

NAME	CODE	ERROR MESSAGE
ERMERP	ERMERP	Missing Right Parenthesis
ERMEQ	ERMEQ	Keyword '=' is missing
ERMISSR	ERMRP	Missing Right parenthesis
ERMISS	ERISSS	Missing string after quote
ERMPHS	ERMRHS	No expression after the keyword '='
ERMRHS	ERMRHSR	Error is recognition of a right hand side of an assertion
ERMTHEN	ERMTHN	Keyword THEN missing
EXPREC	EXPERR	Wrong structure for the exponent part of a floating point constant
FILERR1	FILE01	Left paren missing in FILE or REPORT statement
	FILE02	Right paren missing in FILE or REPORT statement
FILERR2	FILE03	Keyword missing in FILE or REPORT statement
FILERR3	FILE04	Record name missing or illegal in FILE or REPORT statement
FILERR4	FILE05	Character code missing or illegal in FILE or REPORT
FILERR5	FILE06	Medium name missing or illegal in FILE or REPORT statement
FILERR6	FILE07	Keyname missing in FILE or REPORT statement
FLDERR2	FLD07	Maximum length missing or illegal in variable length in FIELD statement
FLDERR3	FLD02	Invalid/missing field type in field/interim stmt
FLDERR4	FLD04	Missing/invalid length in field/ interim stmt

Table 5

ERROR MESSAGE STACKING ROUTINES (continued)

NAME	CODE	ERROR MESSAGE
FLDERR5	FLD05	Missing right parenthesis after field-type in field/interim
INTERR	INTERR	'-' sign is not succeeded by an integer
ITEM01	ITEM01	Name missing or illegal in item list
ITEMER2	MAXER1	Missing/invalid max. no. of occurrences of item
LCERR		Badly formed line or column number for statement
LIBERR	LIB01	Left paren missing in library call
MEDER1	FILENM	Missing/invalid file name after keyword FILE
MEDER2	RECFMT	FORMAT missing/misspelled after RECORD in storage stmt.
MEDER3	TPLLBL	Invalid/missing tape label
MEDER4	RECSZ	Keyword RECORDSIZE missing/misspelled after MAX
MEDER5	VOLNAM	Missing/invalid volume name (external or internal)
MEDER6	DEVTYP	Invalid/missing device type
MINERR	MINER1	Number of occurrences of item missing or illegal
	MINER2	Colon or right paren missing
MODUL1	MODUL1	Colon missing after keyword MODULE
MODUL2	MODUL2	Name missing or illegal in MODULE statement
NUMERR	NUMERR	Error in assembly of a number constant
PARERR	PARERR	Tape spec parameter missing or illegal
PICERR1	PICER1	An error in a picture specification
QNMERR	QNMERR	Qualified name illegal

Table 5

ERROR MESSAGE STACKING ROUTINES (continued)

NAME	CODE	ERROR MESSAGE
RCFER1	RECF01	Record format missing or illegal
RCFER2	RECF02	BLOCKSIZE keyword missing in record format specification
	RECF03	Blocksize value missing or illegal in record format spec.
RCFER3	RECF04	Record size value missing or illegal in record format spec
RPTERR	RPT01	Left paren missing in REPORT statement
	RPT02	Keyword REPORT_ENTRY missing
	RPT03	Report entry name missing
	RPT04	Right paren missing in REPORT statement
RPTNER	RPTN01	Left paren missing in REPORT_ENTRY statement
	RPTN02	Right paren missing in REPORT_ENTRY statement
SEMI	SEMI	Semi-colon missing at end of statement
SRCFL1	SRCFL1	Colon missing after keyword SOURCE FILES
SRCFL2	SRCFL2	Name missing or illegal in source file list
SVMNFLN	LNGER1	Specified length is inappropriate for specified length is inappropriate for specified type of data element (0 or too long)
SVMXFLN	LNGER2	Specified maximum length is inappropriate for the described data type, or is smaller than the minimum length specified.
SVPICST	PICER2	Length of picture specification is too small or too big (< 31)
	PICER3	Bad structure of picture string specification

Tablv 5

ERRO' MESSAGE STACKING ROUTINES (continued)

NAME	CODE	ERROR MESSAGE
	PICER4	Illegal character in picture specification
SVSCALE	PRECR1	The fraction point offset is outside of the bounds -128 <p <127 or inappropriate for the data type described
TAPCRR	TAPE01	Left paren missing in TAPE statement
	TAPE02	Right paren missing in TAPE statement
TARFL1	TARFL1	Colon missing after keyword TARGET
TARFL2	TARFL2	Name missing or illegal in TARGET file list
THENCE	THENCE	Missing keyword ELSE in conditional expression.
TLABERB	TLAB01	Keyword INT^NAME missing in tape label description
	TLAB02	Internal name missing or illegal in tape label description
TRMERR	TRMER1	Left paren missing in TERM description
	TRMER2	Right paren missing in TERM description
UNRECS	UNRfiCS	Unrecognizable statement
VOLERR	VOLER1	VOL^NAME keyword missing
	VOLER2	Volume name missing or illegal
WRBE1	WRBE1	Badly formed boolean expression
WRBT1	WRBfl	Badly formed boolean term
WRCON1	WRC0N1	Badly formed concatenation of expressions
WRFAC1	WRFAC1	Badly formed factor
WRPRIM1	WRPRIM	Badly formed primary
WRTERM1	WRTERM	Badly formed term

storage of the MODEL statements.

The encoding and saving routine are listed in Table 6.

2.2.5 Statement Storage Routines

These routines collect the strings of names and other vital information in the MODEL statements, and pass them to the STORE system, which is a sub-system in itself to store the statements for later processing. Such storage-invoking routines are called at the end of scanning each MODEL statement, and are the ones that begin with the letters "ST". (e.g. STFLD, STREC, etc.). The storage subsystem described below (STORE), which is called by these routines, stores the MODEL statements in a simulated associative memory that facilitates later retrieval.

On analyzing the assertions (computational statements) a syntax or derivation tree which represents the assertion is generated and stored. This representation facilitates later analysis and scanning of the assertion, as well as systematic transformations. The tree representation is reconverted into text form in the code generation phase.

At the end of the syntax pass, we have the entire set of MODEL statements stored in a convenient storage system for further analysis. The storing subroutines which invoke the use of the STORE system act as an interface between t e automatically generated SAP and the storage system presented below. The storage system is an extension to the capabilities of the SAPG since it is general purpose in nature and is independent of the nature of

Table 6:

ENCODING/SAVING ROUTINES

NAME	WHAT IT DOES
DECLEVL	Decrements the Index level of a subscripted variable (It is an entry in ASSINIT)
INCLEVL	Increments the Index level of a subscripted variable (It is an entry in ASSINIT)
INITNUM	Initialize scanning a numeric constant
SETDEVB	Set device flag in media description to imply disk storage
SETDEVC	Set device flag in media description to imply that input is from cards
SETDEVP	Set device flag in media description to imply PRINTER
SETDEV1	Set device flag in med,ia description to imply a terminal
SETDEVU	Set device flag in media description to imply a card punch
SETFUFC	Initiate a node in the syntax tree to store a function reference
SETNH1	Set for assembling a constant number
SETS!>N	Initiate a node in the syntax tree to store a string constant
SETSUBV	Initiate a node in the syntax tree to store a subscripted variable
SETVAR	Initiate a node in the syntax tree to store a variable name
STALL	Stores a node in the syntax tree after all its components have been defined
STBIT	Se s the current string contained in the temporary node tfcfbe a bit string
STDEV	Store devices Tape or disk
STFUN	Stores a node in the syntax tree which contains a function name
STNUM	Concludes the assembly of a constant number (possibly floating point)

Table 6

<u>ENCODING/SAVING ROUTINES</u> (continued)

NAME	WHAT IT DOES
STPIC	Concludes the storing of a picture type specifi cation
STR_CON	Stores a node in the syntax tree which contains a general constant
STRHS	Stores on assertion in the associative memory (an Entry point in ASSINIT)
SVAAS1	Sets a node to contain a conditional assertion
SVASAE1	Sets to define a node containing a simple assertion
SVASNM	Saves assertion name in assertion storage entry
SVASSR	Same as SVASAE1
SVBEXP	Sets a node for storing a boolean expression
SVBF1	Sets a node for storing a boolean factor
SVBLK	Saves blocksize in disk/tape storage entry
SVBT1	Sets a node for storing a boolean term
SVCC	Encodes character code
SVCMP1	Save in a node the recently scanned syntactical unit as the first descendent
SVCOL	Saves column number in field storage entry
SVCON	Sets a node for storing a concatenation of expressions
SVCOND	Sets a node for storing a conditional expression
SVDDNM	Saves data description statement name
SVDEN	Saves density in tape storage entry
SVDEN2	Save density for tape, or giving warning
SVDSK	Encodes disk statement type as disk
SVDEV	Set device name to storage name, and save device: Tape or Disk.
SVEACH	Saves FOREACH name in assertion storage entry

Table 6

ENCODING/SAVING ROUTINES (continued)

NAME	WHAT IT DOES
SVPAC	Sets a node for storing a factor
SVFCN	Saves function name in assertion storage entry
SVFDTP	Encodes field type
SVPDTP2	Save field type, including NUM & DEC
SVFILE	Encodes file statement type as FILE
SVFLD	Encodes field statement type as FLD
SVFLNM	Save file name. Call SVFILE, Set default names for record a storage, and reset device bit (DEVBIT).
L 'XNCR	Saves increment in -disk storage entry
SVINNM	Encodes INTERIM statement type as INTR
SVINTNM	Saves internal label name in disk storage entry
SVINTN	Saves internal label name in tape storage entry
SVKEY	Saves key field in file storage entry
SVLAB	Encodes label type in tape statement
	O«none, 1*IBM_STD, 2«ANSJ_STD, 3=BYPASS
SVLAB2	Save label for tape, or give warning
SVLBNM	Saves library name in file storage entry
SVLINE	Saves line number in field storage entry
SVMEN	Saves member name in record/group storage entry
SVMNFLN	Saves minimum field length in FIELD statement
SVMNOC	Saves minimum number of occurrences in record or group storage entry
SVHOD	Marks the mode as FIXED or FLOATING
SVMXFLN	Saves maximum field length in FIELD statement
SVMXOG	Saves maximum number of occurrences in record or group storage entry

Table 6

ENCODING/SAVING ROUTINES (continued)

NAME	WHAT IT DOES
SVNUMTP	Marks the data type as a numeric data type (BINARY or DECIMAL).
SVNXCMP	Saves the next assembled syntactical unit in a syntax node which is its ancestor.
SVNXOP	Saves the next delimiter associated with the assembled syntactical unit or separating it from its successor
SVOP1	Saves an initial delimiter associated with phrase such as unary 1_1 or 'IF'
SVORG	Encodes organization type in DISK statement S=sequential; I=ISAM;
SVORG2	Saves organization for disk, or give warning
SVPAR	Saves parity in tape statement
SVPAR2	Saves purity for tape, or give warning
SVPIC	Denote the data as 'picture'
SVPRIM	Sets for assembling a phrase for a PRIMARY
SVPICST	Saves the picture specification string
SVQTY2	Save quantity for disk, or give warning
SVQOTY	Saves track quantity in disk storage entry
SVRCNM	Saves record name in file description storage entry
SVRCSZ	Saves record size in tape/disk storage entry
SVRECF	Encodes record format on tape/disk storage entry; 0=FIXED, 1=FIXED BLOCK, 2=VARIABLE
SVRENM	Saves report entry name in report storage entry
SVRLSE	Encodes space release indicator in disk storage entry 1=release; 0=no release;
SVRPT	Encodes report statement type as REPT storage entry
SVSCALE	Saves the scale factor specified in the precision specification of the data type

Table 6

ENCODING/SAVING ROUTINES (continued)

NAME	WHAT IT DOES
SVSR	Saves source name to assertion in ASSR storage entry
SVSRC	Saves source file name in source storage entry
SVSTARF	Records and saves the repetition specification '(*)' in a file statement.
SVSTFL	Saves start file in TAPE storage entry
SVSTFL2	Save start file# for tape, or give warning
SVSTNM	Saves storage name in FILE storage entry
SVSTRNG	Transfer an assembled string constant (may be character or bit) from the general buffer into a special temporary storage. The final storage of the node will be done by STR_CON.
SVTAB	Sets tabulated indicator in group storage entry
SVTAPE	Encodes tape statement type as TAPE
SVTAR	Saves target file name in target storage entry
SVTERM	Initialize a node to store a phrase for a TERM
SVTERM	Encodes terminal statement type as TERM
SVTG	Saves target name to assertion in ASTG storage entry
SVTGSR	At the end of scanning of an assertion two additional storage entries are made One for the list of source variables used in the assertion (type ASSR) and one for the list of target variables defined by the assertion (type ASTG). SVTGSR calls for routines SVSR and SVTG respectively to perform these storage operations.
SVTMUN	Saves tape unit number of tape storage entry
SVTRK	Saves number of tracks in TAPE statement
SVTRK2	Save #Tracks for tape, or give warning
SVTRMNM	Saves terminal name

Table 6

ENCODING/SAVING ROUTINES (continued)

NAME	WHAT IT DOES
SVUCYL	Save units as CYL for disk, or give warning
SVUNIT	Encodes disk units in DISK storage entry
SVUNITS	Saves space units in DISK storage entry
SVUNIT2	Save unit for disk, or give warning
SVVOL	Saves volume name in disk/tape stora e entry

the language specified, and could be used for processing other languages.

The storing routines are listed in Table 7.

2.2.6 Housekeeping Routines

Finally, there are just a few "housekeeping" type subroutines which need not be written by the language definer because they are provided by the SAPG, but which need to be included in the EBNF/WSC.

The house eping routines are listed in Table 8.

2.2.7 An In To Sap Routines

The sub utine ames used in the specification of MODEL can be classifi into on of the following five types of subroutines: error messa stacking routines, encoding/saving routines, storing routines, semantics checking routines, and housekeeping routines. Tables 5-8 provide an alphabetical listing of the routines within each category. In the case of error message routines, the error codes and their meanings are shown. For the other types of routines, their name and tasks are shown.

2.3 The String Storage and Retrieval Sub-System

2.3.1 Introduction

The store routines that are referred to in the EBNF description of MODEL, utilize a general-purpose mechanism for storing source language strings. A similar mechanism is used later for retrieving these source language strings. The following system, basically, consists of a directory structure, described in section 2.3.2 and the format of storage entries described in Section 2.3.3. There are also two main procedures:

Table 7

<u>STORING ROUTINES</u>

(inserted at the <u>end</u> of each type of statement of the EBNF/WSC in order to call STORE to put the statement in the associative memory)

NAME	STMT WHAT PT STORES
STCARD	Stores CARD statement
STDISK	Stores DISK statement
STFILE	Stores FILE statement
STFLD	Stores FIELD statement
STGRP	Stores GROUP statement
STMOD	Stores MODULE statement
STPNCH	Stores PUNCH statement
STPRNT	Stores PRINTER statement
STREC	Stores RECORD statement
STRPT	Stores REPORT statement
STRPTN	Stores REPORT-ENTRY statement
STSRC	Stores SOURCE FILES statement
STSUB	Stores SUBSCRIPT statement
STTAPE	Stores TAPE statement
STTAR	Stores TARGET files statement
STTERM	Stores TERM statement

Table 8

"HOUSEKEEPING" ROUTINES

(inserted in the EBNF/WSC in order to perform services provided by the SAPG)

NAME	WHAT IT DOES
ADLEX	Adds a subpart of a floating point constant to its full representation
CLRERRF	Clears "errors" flag every statement to indicate no syntax errors yet in next statement
ENDINP	Executed upon end-of-file to print last line and wrap-up
FREETMP	Frees &*loc^rion of a temporary data structure which was nef ler ly allocated
NEGATE	Negates th value of a negative integer constant to deri e i correct representation
STMTJFL	Scans for e? of statement delimeters when unrecognizable statement encountered
STMTINC	Increments :he statement number; called at end of each statement

(1) STORE for storing source language strings collected during syntax analysis. STORE is described in Section 2.3.4.

(2) RETRIEVE for accessing previously stored source language strings, based on a variety of "keys". RETRIEVE is described in Section 2.3.5.

Additionally a set of routines specified in EBNP parses and stores the assertions. Section 2.3.6 describes the format of stored assertions. Section 2.3.7 describes the routines that store the parsed assertions. These routines have also been referred to in the description of saving and encoding routines in Section 2.2.5.

The STORE procedure accepts strings which are formed by the subroutines called during syntax analysis. It stores the strings in memory which we call "storage entries" while building "directory entries" in a directory of certain names designated as keys. By building a directory, the strings are stored "associatively" in the sense that statements can later be retrieved based on their content. This capability is crucial to a "non-procedural" language processor since the statements can be input in any order.

2,3.2 The Directory and Storage Structure

The storage entries (the strings to be stored) consist of two parts:

(1) the key names to be entered in the directory which include the names the user provided in the MODEL statements for naming data, assertions, etc. These are the names by which we may want to retrieve information Tater.

(2) auxiliary data from the source language strings including the encoded information in table form. This information is not used as the basis of retrievals.

Each storage entry will contain information from a given MODEL statement. They will appear in memory in the order in which they are processed.

The directory consists of an entry for each key name. Each directory entry points to the first storage entry containing that key name. A linked-list is then maintained from the first storage entry with that key name to other storage entries containing the same key name. A "branch and bound" binary tree structure was chosen for the directory itself to make tree modifications and searching for key names efficient. That is the first key name entered in the directory becomes the root of the directory tree; the next key is entered "above" or "below" it in the tree by lexicographic order; etc.

Each directory entry has the following form:

Key name	Ptr-to-first	Up-pointer	Down-pointer

where

"Keyname" is a string of (up to) 10 characters (padded with blanks)

"Ptr-to-first" is a pointer to the first storage entry containing the "key name".

"Up-pointer" and "Down-pointer" are pointers to other directory entries, whose key names are up or down, respectively, in the lexicographic sense.

Each storage entry has the following form:

```
------------------------------------------------------------------
N       namel    ptrl     . . .   Name n   Ptr-n    Ptr -to-data
------------------------------------------------------------------


                                                    -------------
                                                     other data
                                                    --------------
```

where

N is the number of key names in the storage entry string.

Name (i=1 to n) is a key name of a variable.

Ptr (i=1 to n) is a pointer to the next storage entry with the same key name.

Ptr-to-data is a pointer to auxiliary data from the source language statement.

Figure 7 depicts an example of three storage entries and a directory consisting of only three entries, X,Y, and Z, where Y is the directory tree apex. Such a structure was partially motivated by similar ideas in the "multi-list" file organization.

2.3.3 Storage Entries Format and Tables For MODEL Statements

The STORE mechanism, described in the next section, is called by SAP's storing subroutines to store the MODEL statements for retrieval (by RETRIEVE) in the later phases. For each type of MODEL statement, the key names in it are stored in its storage

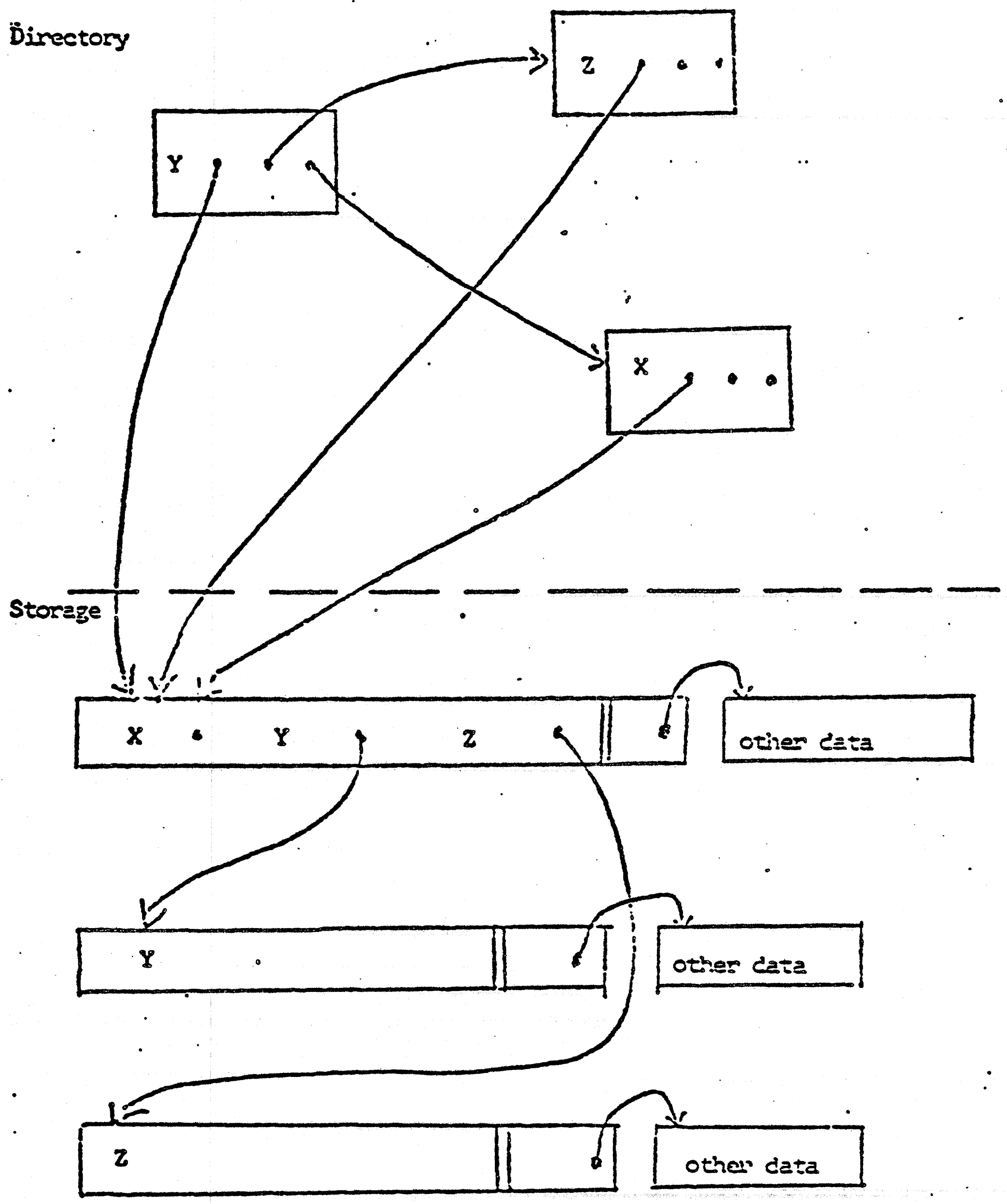


Figure 7
Sample Directory and Storage Entries

entry. The non-key information in the MODEL statement (information which is not used to specify retrievals) is kept in description tables, which are connected (by STORE) to the corresponding storage entries as was shown above. Table 9 summarizes the internal format of the storage entries and the corresponding description tables for each type of MODEL statement. The left name in each entry is the name of the statement being stored. The middle column shows the information appearing in the corresponding storage entry (with the pointers omitted due to lack of space). The right column shows the additional encoded information, if any, from the statement. The key names beginning with a dollar sign ($) in the storage entries are not user-provided, but are inserted by the system for its own information. The last name in each storage entry, for example, identifies the type of statement, while the name beginning with a "$P" identifies the parent file in which a data item appears.

2.3.4 The STORE Procedure

The STORE(S,D) Procedure has two parameters, S and D. S is the string containing the key names which are to be stored and to be entered in the directory. D is a pointer to previously built auxiliary data from the source string. The latter usually is an encoded form of non-key source language information.

Algorithm STORE shows the storing procedure. Section 2.3.2 already depicted the data structures that STORE creates.

STORE receives the key names from S and creates a storage entry for it (Steps 1-3). It checks if they are in the directory (Steps 4-5, subroutine SEARCH_DIR). If the key is in the

Table 9 Storage Entries Format for MODEL

MODEL Statement Schema	Storage Entry Key Names	AuxHilary Descriptions
		Type Stmt#.
MODULEi module-name	module-name $MOD0LB	MODL n
SOURCE FILIiSi sx, 32t....s$_n$	$SOURCE si s$_2$ ••• s$_n$	SRCF n r
TARGET FILESi ti# t$_2$i ... t$_m$	$TARGET tj, t$_2$ ••• t$_m$	TARP n
filename IS FILE(^ROUP j. l* rl STORAGE IS 3. RECORD KEY IS k, ORG IS O)	^filename r a k $FILE.:	FILE n ORQ-Code Key-flag ls-star 0-SAM 0 no sort 0-no repet.]- ISAM key for r «,,,,,> u v ' i~8 ort key lm* repeati
record-name IS RECORD (m$_1$. m$_2$,...,m$_n$)	record-name m$_1$ m$_2$ ••• m$_n$ $Pfile $RECD	BECD n #members members ^subscripts first sub. second sub.
group-name IS GROUP m$_1$,m$_2$,...,m$_n$)	group-name *i} 102 ••• m$_n$ $Pfile $GRP	GRP n (same as record) in 0

MODEL Statement Schema	Storage Entry Key Namea	Auxtlllary Descriptions Typo	 Stmtff.			
field IS FIELD (fieldtype (minlength : maxlength)	fieldname $Pflle $FLD	FLD	n	fieldtype O~char l^binary 2«numeric 3=decimal ^binary floating 5-bit 6«=docinial floating 7«picture	**length type** O«fixed invariable	min/max scale factor picture string (if type -7)
SOURCE: $s_1, s_2, \dots, s_n$	assertion-name a^ $s_2 . \cdot . a_n$ $ASSERT	ASSR	n	tfnames components		
TARGET: $t_1, t_2, \dots, t_m$	assertion-name $t_1, t_2, \dots t_m$ $ASSERT	ASTG	n	tfnames components		
assertion-name:	assertion-name 5ASSERT	ASTX	n	Pointer to syntax tree		
subscript-name IS SUB [seRJi;P3l[(range)]	**subscript name $$SUB**	$SUB	n	range		
storage-nai^e IS						
CARD	storage-name **$CARO**	**CARD**	**n**			
TAPE (...)	**$TAPE**	**TAPE**	**n**	**tape-attributes**		
DISK (..•)	$DISK	**DISK**	**n**	disk-attributes		
TERM (...)	$TEKM	**TERM**	**n**	term-attributes		
PUNCH (...)	$PNCH	PNCH	n	punch-attributes		
PRINTER (...)	$PRMT	**PKNT**	**n**	print-attributes		

Table 9 (continued) Storage Entries Format for MODEL

Algorithm STORE : The Store Procedure

Parameters: S=string of keys to be stored;
D=pointer to other data

(see Section 2.3.2 for diagrams of Data Structures)

[Subroutines called: CHECK_DIR, GENERATE_ENTRY]

Step 1. Count #KEYS.

Step 2. Allocate the storage entry for S (call it SE, according to the format shown).

Step 3. Connect PTR_TO_DATA in SE to D.

Step 4. For each key name, perform steps 5 through 11.

Step 5. If key exists in the directory (Algorithm CHECK-DIR), then go to step 7; else go to step 6.

Step 6. Create a directory entry for this key. (Algorithm GENERATE-ENTRY)

Step 7. Let DE=this directory entry.

Step 8. If PTR_TO_FIRST in DE already points to a first storage entry with this key name, then go to step 9; else go to step 11.

Step 9. Get the next storage entry in the list.

Step 10. If it is the last in list, then go to step 11; else go to step 9.

Step 11. Add the new SE to the list.

Step 12. Return.

directory, then it follows the "pointer-to-first" which points to the first storage entry with that name (Steps 7-8). The array of strings in each storage entry is scanned until the key name is found. If its "next" pointer is null (end-of-list), then it is set to point to the newly created storage entry (Steps 8-11). If it is not, the process is repeated until a null (end-of-list) pointer is found (Steps 9-10). If the current key name is not found in the directory, it is entered in the appropriate spot in the lexicographical position in the directory (Step 6, subroutine CREATE_DIR) and the pointer in the directory is set to point to the newly created first storage entry (Steps 7-8).

2.3.5 The RETRIEVE Procedure

RETRIEVE(E,D,S,N,P) is the procedure for retrieving desired storage entries, by searching through the data structures depicted in Figure 7 and Table 4. It is invoked by many routines described in subsequent phases of the Processor. It has five input parameters as indicated. RETRIEVE finds all the storage entries in which the given key name or expression of key names, E, appears and furthermore checks whether the first characters of data associated with the storage entries match the string D. That is, RETRIEVE finds all the storage entries with keys satisfying the logical expression E and other data D. RETRIEVE starts its search at directory entry S, normally the root node of the directory, and it returns a list of pointers P, to those storage entries which satisfy the request by the calling program. The number of storage entries satisfying the request is returned in N.

The logical expression used to retrieve strings can be any boolean expression involving "key" names or names in the MODEL statements in disjunctive normal form, where the first key in each term is non-negated. For example, consider the following statement by a calling program:

```
     CALL RETRIEVE(KEYS, '',START, N,P);
```

KEYS might contain the string value 'PRICE & ¬QUANTITY|EXTENT'. This makes RETRIEVE find all storage entries (which correspond to all statements in the MODEL specification) in which PRICE appears and QUANTITY does not appear, or statements in which EXTENT appears. The null second parameter means that the auxiliary data portion of each statement is immaterial. RETRIEVE would then start its search and return a list of pointers in P to those storage entries which satisfy the condition, and N would be set to the number of such statements that satisfy the condition.

Algorithm RETRIEVE is shown on the following page. An example showing the retrieval mechanism to retrieve all storage entries with key names "B" and "C" is given in Figure 8. The diagram shows in parentheses the steps that correspond in the algorithm. RETRIEVE starts by getting the leading key name of the first conjunct (Step 1) and searches the directory for it (Step 2). If found, it puts the list of pointers to all storage entries with that name in a temporary list (Steps 3-7). If there are other names in the conjunct (Steps 10,14), then RETRIEVE eliminates the pointers in the temporary list to storage entries that do not have the other terms in the

Algorithm RETRIEVE : The Retrieve Procedure

Parameters: E=logical expression string; S=pointer to beginning of directory (input);
P=list of pointers satisfying E; N=number of satisfying entries

(see Figure 7a for diagrams of data structures)

```
Step 1. Get leading key name K of next conjunct from E. If
no more, go to Step 22.
Step 2. Check directory for K (standard binary tree search
in subroutine SEARCH-DIR given earlier).
Step 3. If found, then go to step 4; else go to step 1.
Step 4. Set PSE=PTR_TO_FIRST (pointer to first storage entry
with K)
Step 5. Add PSE to W list (temporary list of pointers)
Step 6. If K in PSE storage entry points to another storage
entry with K, then go to step 7; else go to step 8.
Step 7. Set PSE to next storage entry in the list, go to
     Step 5.
Step 8. If end of E, then go to step 20; else go to step 9.
Step 9. Get next symbol in E.
Step 10. If symbol='&' then go to step 14; else go to step
11.
Step 11. If symbol='|' then go to step 12; else error
return.
Step 12. Add list of pointers in W to list of pointers in P
without duplication.
Step 13. Go to step 1.
Step 14. Get next symbol.
Step 15. If symbol='~' then go to step 16; else go to step
18.
Step 16. (Case of conjoining negated term) eliminate
pointers in W to storage entries which also contain next key
name in E.
Step 17. Go to step 8.
Step 18. (Case of conjoining non-negated term) eliminate
pointers in W to storage entries which do not contain next
key name in E.
Step 19. Go to step 8.
Step 20. Add list of pointers in W to list of pointers in P.
Step 21. Set N=number of pointers in P list.
Step 22. Return.
```

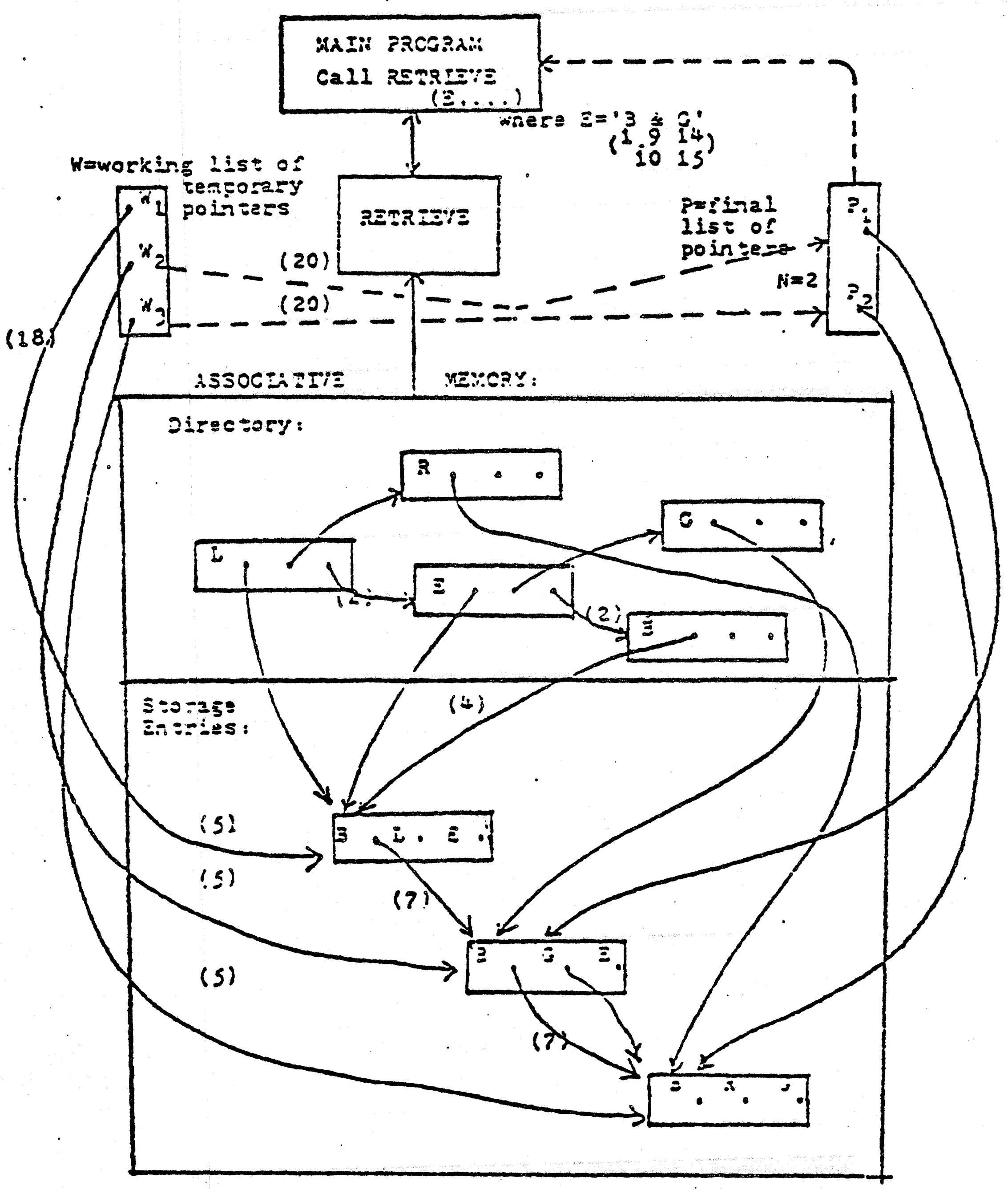


Figure 8 Example of Retrieval Mechansim

conjunct (Steps 14-16). If there are more conjuncts in the expression, then the process is repeated and the additional pointers are added to the list (Steps 12-13). When the end of the expression is reached, the list of pointers to the satisfying storage entries and the number of pointers are returned (Steps 20-22).

2.3.6 Storage Structures For Assertion Statements

Analysis of an assertion statement causes three storage entries to be made for the statement. (See also Table 9). The first entry is of type ASSR and contains a list of all the names which are sources to the assertion. These are all the names which appear on the right hand side of each equal sign, (including subscript expressions) and within boolean condition expressions. The second entry has the type ASTG and contains a name which is the targets of the assertion, i.e. it's value is defined by the assertion. Assertions will have only a single target. The third entry, of type ASTX, contains in it's main part just the assertion label (system generated if not provided by the user) and a keyword $ASSERT. Its auxiliary data contains a pointer to the syntax tree which represents in a parsed form the body of the assertion.

2.3.6.1 The Syntax Tree for an Assertion

The syntax tree of an assertion is constructed out of mutually linked nodes. There are nodes of two types: Non terminal nodes which have descendants and terminal nodes which have no descendants and represent an atomic syntactical units such as identifiers, numeric and string constants. Each node corresponds to a phrase in the parsed assertion, and if it is non terminal the list of its descendants represents the further breakup of this phrase.

<u>2,3.6.2 The Structure of Non-Terminal Nodes</u>

The structure of Non Terminal nodes is as follows:

TYPE	*n*- Number of Sons	Delimit #1	Pointer to Son #i	...	Delimit #n	Pointer to Son #n

where

<u>"type</u> is an integer code identifying the syntactical type of the phrase according to the following legend:

0.- Conditional Assertion. Examples If A«B THEN C«p.

1 - Simple Assertion Example: A«B

2 - Conditional Expression. Examples IF A > B THEN C ELSE *0.*

5 - Boolean Expression. Example: (A=B) | (C«D)

6 - Boolean Term. Examples (A > 5) & C <« 3

7 * Boolean Factor* Example: C * 7

8 - Concatenation. Example. AllJI'END[1]

9 ~ Arithmetical Expression. Example: A*B+C*D

10 - Term. Ex.: A*B

11 - Factor. Ex.: A**2

12 - Primary.Ex.: A,B(I+1), (A+B)

13 - Function Ex: SUM(A,I)

14 - Subscripted Variable. Ex: A(FOR_EACH.A)

<u>"Number of Sons</u>" is the number of components or subphrases the indicated phrase is broken into. Thus if the phrase is "A+B" it is of type 9 (Arithmetical Expression) and it is parsed further into the subphrases "A" and "B". The '+• delimiter will be stored as delimiter no, 2 in the current node.

The delimiters are encoded as integers according to the following legend:

1 - * '(Blank - No delimiter)

2 - 'IF' (keyword)

3 - 'THEN'

4 - 'ELSE'

5 - •-'

6 - '+'

7 - '-'

8 - '*' (Standing for multiplication)

9 - •/*

10 - '**• (Exponentiation)

11 - 'I' (Alternation - Logical 'or')

12 -.'&'

13 - '||' (Concatenation)

14 - '-»' (Negation)

15 - '('

16 - ')'

17 - V

18 -'->>'

19 - ' >»'

20 - ' ¬i < •

21 - .' <-•

22 - '-!»'

23 - »> '

24 - ' < '

"Delimiter #i, i=1, ..n" are the delimiters separating the subphrases. The first one is any delimiter prefixing the whole phrase such as the '-' in the phrase -A or the ' ' in the phrase ' (A <B & B <C)'

"Pointer to Son #i, i=1,..n" these are pointers to other nodes which represent the subphrases into which the current phrase is parsed.

2.3.6.3 The Structure of Terminal Nodes

Terminal nodes are used to store constants such as variable names, string or numeric constants. Their structure is as follows:

Type	Constant-Length	Constant

where

"Type" is an integer code identifying the type of the constant according to the following legend:

20 - A character string constant. Ex.: 'ABC'

21 - A function name. Ex.: SUM

22 - A numeric constant.Ex.: 3.14

23 - A variable name. Ex.: PAY

24 - A bit string constant. Ex. '1001'B

"Constant Length" is the length of the character string representing the constant. It will be 3 for storing the variable name PAY.

"Constant" is the actual character string representing the constant.

During later processing (Module ENEXDP), all the terminal nodes which refer to non constants (types 21,23) are converted to a different format; referred to as Variable Terminal Nodes:

Type	Node#

'Type' As before is an integer code identifying the type of the name according to the following legend:

25 - Variable type. The associated name is a variable and NODES is the dictionary entry number of this variable.

26 - Subscript type. This stores the name of a subscript. NODES refers to a dictionary entry number. This dictionary entry can be of one of the following types:

'GRP' or 'FLD', which must be repeating. If this entry name is X then the name of the subscript is FOR_EACH.X.

'$SUB' - This is either a subscript declared by the user or one of the system supplied free subscripts SUB1 to SUB9

'$' - This is a free subscript added by the system. It is one of the subscripts $1 to $9.

'$I' - This is a loop variable added by the system for lack of a user provided name. In any of the latter three cases the name of the subscript is the name of the entry.

27 - Function Name. NODE# is a running index in a list of functions recognized by the system. See [ ] for the list.

An an overall example consider the syntax tree for the assertion:

```
IF A-B | C < D $ E < = * F
    THEN X(FOR_EACH.X) = (Y + 2) *T|| '$' ;
    ELSE X(FOR_EACH.X) * '0' ;
```

It is described in Fig. 9, with the modification that delimiters are represented by themselves rather then in their encoded form, to improve readability.

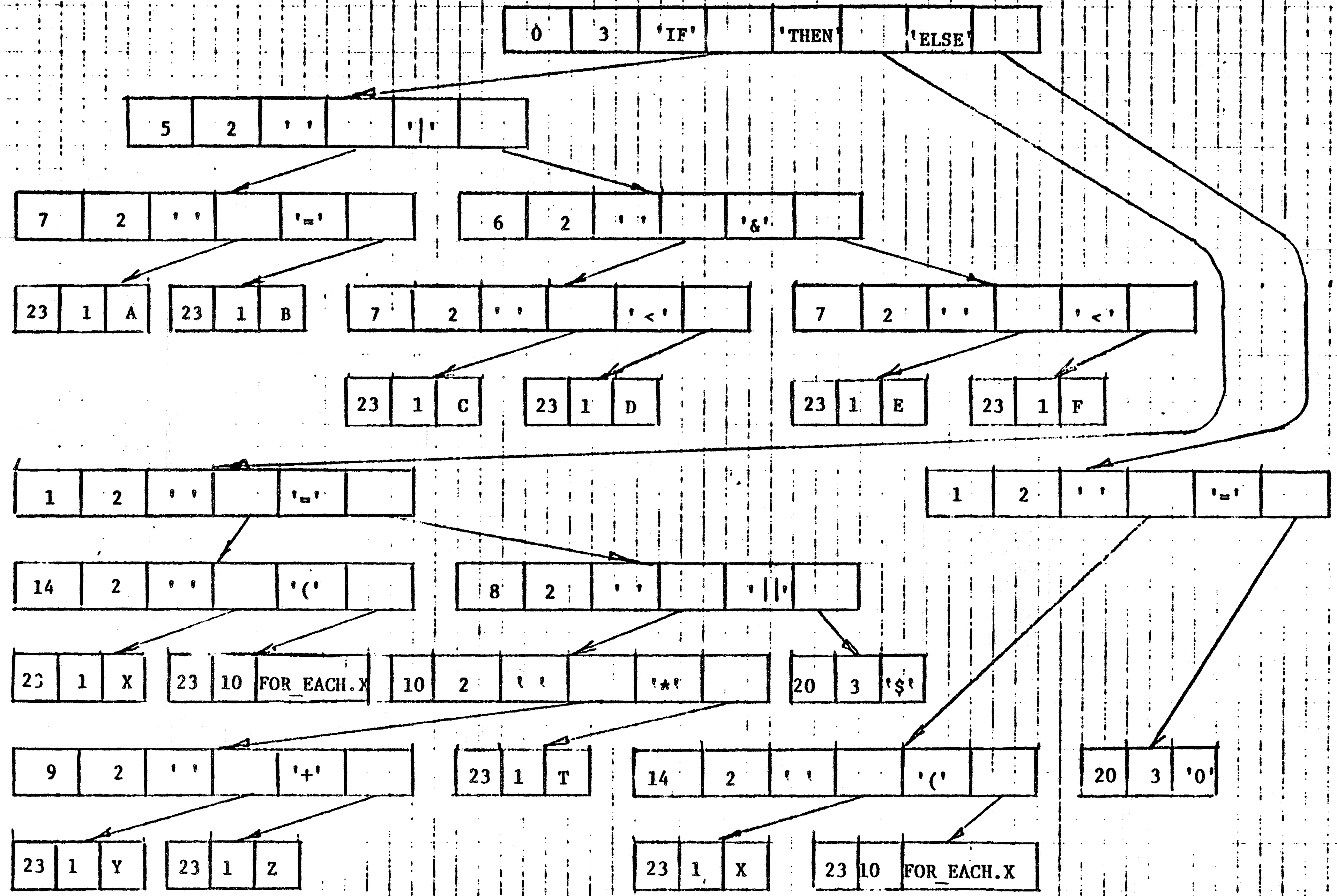


Figure 9 Syntax Tree For Example - Assertion

2.3.7 The Syntax Tree Construction Routines

Several routines are responsible for the construction of the syntax tree of an assertion. They may be classified and described as follows:

Setup Routines: On entering a parse for a phrase of a certain type (by SAP) an appropriate setup routine is called. This routine allocates a temporary node area (temporary since we do not know yet how many subphrases or components it will have), assigns a type number corresponding to the type of the phrase and resets a component count to 0. There is a setup routine corresponding to each phrase's type. They are for the non terminal types (listed in increasing type code order):

SVAAS1, SVASSR (SVASAE1) SVBEXP, SVBT1, SVBF1,

SVCON, SVAE, SVTERM, SVFAC, SVPRIM, SETFUNC, SETSUBV.

For the terminal types (codes > 19), a string area is allocated and a type variable is assigned too. The terminal types setup routines are: SETSTRN, STFUN, SETNUM, SETVAR. No setup routine exists for bit string since the distinction between it and a character string can be made only at the end of its scanning.

Save Routines: These are common to all non terminal phrases. They alternately store delimiters and pointers to components, increasing the "number of sons" counter appropriately. These are all stored in the temporary node storage area.

SVOP1 - Stores a first delimiter. If this routine is not called the first delimiter is always set to 1 (= '').

SVCMP1 - Stores a pointer to the first component.

SVNXOP - Stores the recently scanned delimiter in the next available delimiter slot. Then increment the "number of sons" counter.

SVNXCMP - Stores a pointer to the recently assembled subphrase in the next available component slot.

Storing Routines : These finalize the node structure, after scanning of the phrase is complete. Since size of strings and number of sons are known by this time, a permanent node space is allocated and the contents of the temporary storage entry transferred there. The temporary storage area is then freed.

STALL - This is the storing routine for all the non-terminal nodes. It first checks to see if the assembled node is not trivial. It will be trivial if it contains only one component and the first delimiter is blank. In this case no permanent storage is made for this node. This check eliminates redundant nodes in the syntax tree. If the node is not trivial, a permanent allocation is made for it and the proper contents transferred these.

For the terminal nodes we have separate storing routines.

STNUM - Stores a numeric constant

STFUN - Stores a function name

SVSTRNG - Transfers a string constant to the storage area before calling on STR_CON.

STBIT - Stores a bit string

STR_CON - A common routine for storing all constants. It Allocates a permanent node storage and transfers type, length and string into it.

2.4 Cross Reference and Attribute Report

A useful product of the Syntax and Statement Analysis Phase is a cross-reference report, produced by a cross-reference program (XREF) whose input is the encoded and stored MODEL specification. The XREF report provides an alphabetical listing of all the names provided by the user, and some of the reserved special names (such as CHOICE). For each name, the report provides the statement number in which the entity was described, the statement numbers of statements in which it is referenced, and the attributes or other known characteristics regarding the name.

For example, if field X is described in a given statement and is used in various other MODEL statements, such as in assertions, the cross-reference list would provide the original statement number in which it is described, a list of all the field's attributes as well as the names of the file or files in which it is a member, and a list of statement numbers which reference the given field name.

The cross-reference report is produced by the XREF module. It produces the report by traversing the directory and producing each line by successive uses of RETRIEVE to get the corresponding references. A bubble-sort is used to alphabetize the listing (in a subroutine named ALPHDIR).

3. Analysis of MODEL Specification

3.1 Introduction and Background

3.1.1 An Illustrative Example

In this phase the MODEL Processor analyses the MODEL specification by use of directed graphs. This introductory sub-section presents and exemplifies the background and terminology involved in this phase, and describes the tables, and other data structures that are built from a MODEL specification.

In order to exemplify the algorithms and data structures used, a sample problem is presented below and described using the MODEL language.

Section 3.2 provides an overview of the processes involved in this phase, and Section 3.3 discusses them in greater detail.

The statements of a MODEL specification consist of a series of descriptions of the following:

(1) files, which are designated as source files, target files, (or both). The description of each file may include the physical storage medium.

(2) components of each file; i.e. records, groups, fields, as well as assertions for defining data-dependent parameters.

3) assertions giving logical and arithmetic relationships that define the various target and interim fields.

A small sample set of MODEL statements is provided in for discussion purposes. This example is used here and in subsequent sections as a vehicle for explaining the various algorithms.

The specification of the example is shown in Figure 10(a). Statements added by the system are shown in Figure 10(b).

```
/*********************************************************************/
/*                                                                   */
/*                   MINSALE MODULE SPECIFICATION                    */
/*                                                                   */
/*********************************************************************/

   1      MODULE: MINSALE,
   2      SOURCE: TRAN, INVEN;
   3      TARGET: SLIP, INVEN;

/*********************************************************************/
/*                                                                   */
/*                          FILE DESCRIPTIONS:                       */
/*                                                                   */
/*********************************************************************/


/*********************************************************************/
/*                                                                   */
/*                 DESCRIPTION OF TRAN        FILE                   */
/*                                                                   */
/*********************************************************************/

   4      FILE TRAN
   4               RECORD SALEREC(*)
   4               STORAGE SALEDECK
   4               DEVICE CARD
   4               FORMAT FIXED
   4               BLOCKSIZE 3200
   4               RECORDSIZE 80;
   5                           SALEREC IS RECORD (CUST ,STOCK ,QUANTITY);
   6                                  CUST  IS FIELD(CHAR(5));
   7                                  STOCK# IS FIELD(CHAR(7));
   8                                  QUANTITY IS FIELD(CHAR(3));

/*********************************************************************/
/*                                                                   */
/*                 DESCRIPTION OF INVEN       FILE                   */
/*                                                                   */
/*********************************************************************/

   9      FILE INVEN
   9               RECORD INVREC
   9               STORAGE INVDISK
   9               KEY STOCK#
   9               ORG ISAM
   9               DEVICE DISK
   9               FORMAT IS VARIABLE
   9               MAX BLOCKSIZE IS 6800
   9               MAX RECORDSIZE IS 17;
  10                           INVREC IS RECORD(STOCK#,SALPRICE,GOH);
  11                                  STOCK# IS FIELD(CHAR(7));
  12                                  SALPRICE IS FIELD(NUMERIC(5));
  13                                  GOH IS FIELD(NUMERIC(5));
```

Figure 10(a) MODEL Specification of The MINSALE Example

```
/* * * *** **** ******** * * * * ******** ************************************* ************* A/
/*                                                                                              */
/*                            DESCRIPTION OF SLIP        FILE                                   */
/*                                                                                              */
/***** * ****************** ************************ **********************/

  U       FILE  SLIP
  U                RECORD  SLIP*EC<*>
  U                STORAGE  S*L£DISK
  14               DTvICF  DISK
  U                F.CkPAT  IS  FIXED
  1^               bUCKSIZL  IS  ]i'Ct
  U                rv^CORDSI^T  IS  SC;
  15                        SLIPKEC  IS  RCCCRD  (CUST ,STOCK-,CHANGE);
  U                                   CUST    IS  H FLDCCH A.H (5) );
  17                                  STOCK  IS  FILLM(CHAR(4));
  1?                                  CHARuE  IS  FIcLD  (Nu^6PIC(P));
          /* * * * * * * * * * . * * * * * - * * * * * * * * * * * * * * * * * * * * * " * * * * + * * * * . * * - * * * * * * * * * * * /
  19          CHA^u?  =  QUANTITY  *  CL^.iUVEr« •SALPRICE;
  ^0          NF*.1**WEN •ttC-H  =  DLL.INwE•«•<*Oh  -  QUANTITY;
x ^1          POINTt>% •OLi>. INVR£C  «  TftAN# STOCK *;
          /*  * * * * * * * * * * * * * * * * * * * * *  * * * * * * * * * * * * * * *  * * * * . * * * * * * * * . * * * * /
```

Figure 10(a) Continued

```
   1       SLIF.STOCK =TRAN.STOCK *;
   1       SLIP.CUST^=TRAN.CUST-;
   1        NEs^•iNVk.N1.SALPR1C£=OLD.iNVEu•SALPRICE;
   1         NEU;*INVF.N.STOCK—OLD*INVEK.STQCK J;
   1           SYSGENI  IS  GROUP(WE1-••INVRcC(*));
   1            $YSGE*\?  IS  GRCUPCOLD.INVREC(*));
   1           SYSGEf*7  IS  GROUPCPOINTEft.OLD•IKfvREC(•))';
```

Figure 10(b)s Statements Added By The MODEL System

It is referred to as MINSALE. It describes a module whose input is sale transactions (consisting of a customer number, stock number, and quantity desired) and an inventory file of items (consisting of a stock number, price, and quantity on hand). The output is a sale slip report (consisting of the customer number, stock number, and charge) and the updated inventory file with the new quantity on hand after the sale. The cross reference report is shown in Figure 10(c).

3.1.2 The Array Graph

The preparer of the MODEL specification gives each entity in his statements -- file, field, assertion, etc. -- a symbolic name. In this phase, each name is related by the Processor to other names in one of several ways. Hierarchical relationships exist when one data item contains another, such as when a file contains a record, a record contains a field, etc. A pointing relationship exists when a field of a record in one file is used to compute a key to a record. A dependency relationship exists between a field and an assertion when the field is a source variable of the assertion and between an assertion and its target field.

All of these are precedence relationships, in that the former in some sense must precede the latter and is said to be a predecessor (also known as a precedent) of the latter, while the latter is a successor (also known as a direct descendent or dependent) of the former. The various types of precedence relationships that are implicit or deduced from a MODEL specification are summarized below. Each type of precedence relationship has a corresponding predecessor and successor types. The types of precedence relationships

```
1                                          CROSS REFERENCE  AND ATTRIBUTES REPORT
0   NAME                STATEMENT          ATTRIBUTES                      REFERENCES
                        DESCRIPTION
0
 AASS13                  19                ASSERTION
 AASS14                  20                ASSERTION
 AASS15                  21                ASSERTION
 CHARGE                  13                FIELD, NUMERIC(    8) IN           19, 15
                                           FILE SLIP
 CUST#                   16                FIELD, CHARACTER(    5)            15
                                           IN FILE SLIP
 CUST#                    6                FIELD, CHARACTER(    5)             5
                                           IN FILE TRAN
 INVDISK                  9                DISK NAME                           9
 INVEN                    9                FILE,SOURCE,TARGET,                20, 20, 19,  3,  2
                                           SORTED/KEYED
 INVREC                  10                RECORD,( 7 SUB-MEMBERS),           21,  9
                                           IN FILE INVEN
 MINSALE                  1                MODULE NAME
 NEW                                       RESERVED WORD                      20
 OLD                                       RESERVED WORD                      21, 20, 19
 POINTER                                   RESERVED WORD                      21
 QOH                     13                FIELD, NUMERIC(    5) IN           20, 20, 10
                                           FILE INVEN
 QUANTITY                 8                FIELD, CHARACTER(    3)            20, 19,  5
                                           IN FILE TRAN
 SALEDECK                 0                CARD NAME                           4
 SALEDISK                14                DISK NAME                          14
 SALEREC                  5                RECORD,( 3 SUB-MEMBERS),            4
                                           IN FILE TRAN
 SALPRICE                12                FIELD, NUMERIC(    5) IN           19, 10
                                           FILE INVEN
 SLIP                    14                FILE,TARGET,UNSORTED                3
 SLIPREC                 15                RECORD,( 7 SUB-MEMBERS),           14
                                           IN FILE SLIP
 STOCK#                  17                FIELD, CHARACTER(    4)            15
                                           IN FILE SLIP
 STOCK#                  11                FIELD, CHARACTER(    7)            10,  9
                                           IN FILE INVEN
 STOCK#                   7                FIELD, CHARACTER(    7)            21,  5
                                           IN FILE TRAN
 TRAN                     4                FILE,SOURCE,UNSORTED               21,  2
```

will have direct implications on the program to be generated. For example, a record must be read before any of its component fields can be used. A key to a record must be available before the record being pointed to can be accessed* A value of a field which is a source (or input) to an assertion must be available before invoking the procedure embodying the assertion, A field which is a target of an assertion is only defined after the procedure is called. These and other requirements of the program to be generated are implied by the precedence information conveyed in a directed graph that is referred to in the following as an array graph.

An array graph is a pair <N,A>: a set,of nodes N*{ N1,N2# .-•", ,N-m} and a set of ordered pairs ("edges" or "arcs") A = {A1#A2,...,Ap} where each Ai is an ordered pair (Nj,Nk) representing an edge from node Nj to node Nk. In other words, A is a relation on N x N. Each node may have 0, 1, or more edges emanating from it.

Each edge (Nj,Nk) from node Nj to node Nk is a member of one of a set of different types of relations and is labeled by one of the possible labels.

An example of a labeled directed graph appears in Figure 11, which corresponds to the example of Figure 10. Each node of this graph represents the name of one of the entities in the MODEL statement, including files, records, groups, fields,

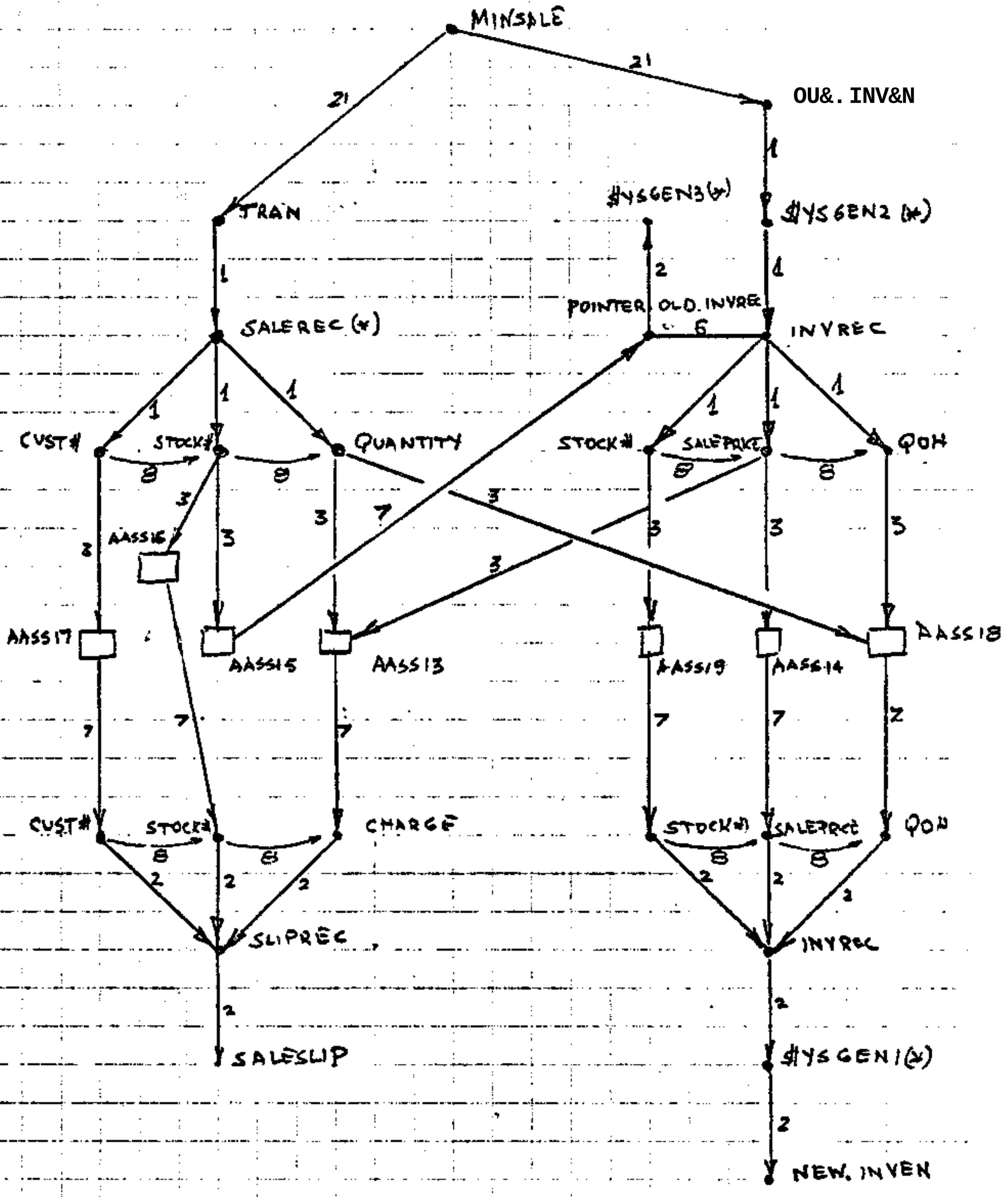


Figure 11: Array Graph For The Example of Figure 10 (Edges of Types 10,11, 12 and 18 omitted

assertions, etc. Each node has 0, 1, or more edges emanating from it pointing to successor nodes; i.e. to nodes to which it is precedent.

Generally, each MODEL statement of Figure 10 corresponds to one node. The exceptions of the one-to-one correspondence are the following:

(1) Files that are both input and output (such as INVEN in the example) as well as their component records, groups, and fields are described only once in MODEL, but become two nodes in the digraph -- one for the "old" or source data and one for the "new" or target data.

(2) The list of source and target files in the header of the MODEL specification do not correspond to any node because the file statements themselves correspond to the nodes for files.

(3) Qualified names prefixed by a key, such as POINTER, SIZE names, etc., constitute interim data. They are not described explicitly by the MODEL user, but are used in context. However, in the array graph, such names do correspond to nodes and have successors, predecessors, etc.

3.1.3 Representation of Edges

Edges are tied to their source and target nodes by edge-lists associated with these nodes.

Each node has in its attribute list the following four entries:

SUCC_LIST - A list of the edges emanating from the current node.

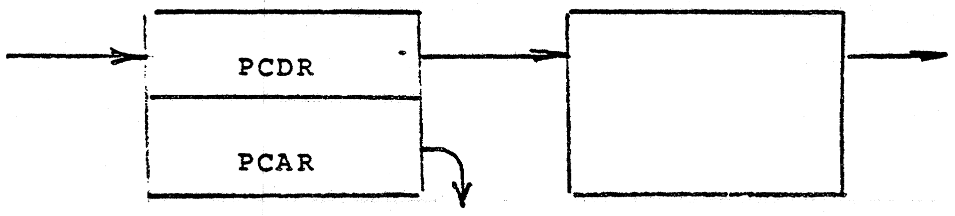
#SUCCESSORS - The number of edges of the successors list.

PRED_LIST - A list of the edges coming into the current node.

#PREDECESSORS - The number of edges on the predecessor list.

An edge list has the following format:

PPAIR:



PCDR - A pointer to the next list element.

PCAR - A pointer to an edge structure.

Consequently when an edge is created it will be entered into the successors-list of its source and into the predecessors-list of its target. Similarly when an edge is deleted it will be deleted from both these lists.

Every edge going from the node S to the Node T has the uniform format:

$$T(U_k,..U_1)\overset{t}{\leftarrow}S(J_m,..J_1)$$

where is the type of the edge, as described below.

k the dimensionality of T,

m the dimensionality of S,

each subscript expression J_i, $1 \le i \le m$ is in of one of the forms:

a# U£ for some 1 £ A £ k

b* U^-l for some 1 £ *I* £ k

c. uj7~c for some 1 £ A £ k and an integer constant c > 1

d# E standing for a general unanalyzed expression.

Consequently in our representation of such edges we do not have to specify the left hand side which is obtainable from the attributes of T. An edge will be completely specified by giving:

5 - The source of the edge^

T - The target of the edge,

t - The type of the edge

6 « k-m - The difference between the dimensionality of the target and the source.

$j_1 \dots j_m$ - The list of subscript expressions for the source variables.

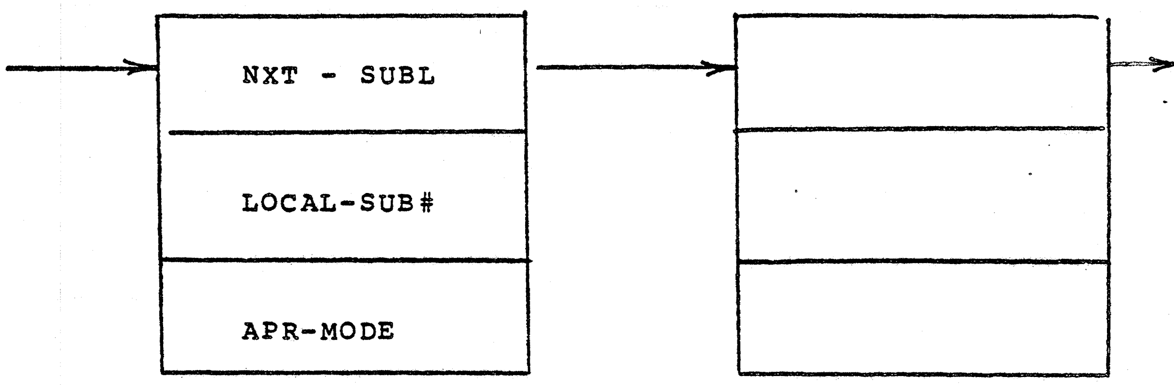
Indeed, an edge is represented by a structure:

EDGE:

SOURCE	:	S
TARGET	:	T
EDGE-TYPE	:	t
DIMDIF	:	6
SUBX	:	A pointer to the subscript expression list.

The subscript expression list is composed of elements of the following form:

EDGE-SUBL:



Here:

NXT-SUBL - A pointer to the next list element.

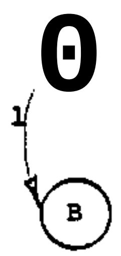
LOCAL-SUB# - For cases a,b,c this gives ℓ, i.e. the ordinal number of the subscript as it appears in $T(U_k,..U_1)$.

APR_MODE - Distinguishes between the cases. It has the value 1,2,3,4 corresponding to cases, a,b,c,d above respectively.

Note that in case c we do not retain the constant $c > 1$.

3.1.4 Edges and their Types in the Array Graph

Type 1 Hierarchical Source Edge:



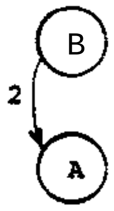
Drawn between a node A in an input file and any of its immediate descendants, B. If U is the local subscript list of B, then we have the following cases:

If B is repeating then

$$B(U_k, \ldots U_1) \leftarrow A(U_k, \ldots U_2)$$

Otherwise $B(U_k, \ldots U_1) \leftarrow A(U_k, \ldots U_1)$

Type 2, Hierarchical Target Edge:



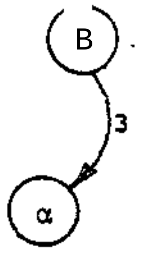
Drawn between a node B in an output file or interim structure and its immediate ancestor A.

If B is repeating then

$$A(U_k, \ldots U_1) \leftarrow B(U_k, \ldots U_1, E)$$

Otherwise $A(U_k, \ldots U_1) \leftarrow B(U_k, \ldots U_1)$

Type 3. Source to Assertion Edge



The local subscript list of α contains all the subscripts which appear either on the lhs or rhs of α. Subscripts which appear on the rhs only are considered to be reduced. For each instance of B in α we draw an edge:

$\alpha(U_k, .. U_1) \leftarrow B(J_m, .. J_1)$

where m is the dimension of B.

The order of the subscript J_i is as stated in the assertion.

Each of the J_i can assume one of the following forms:

1) U_j for some $1 \le j \le k$

2) U_j-1 for some $1 \le j \le k$

3) U_j-C for some $1 \le j \le k$ and an integer constant $C > 0$.

4) E .- standing for all other subscript expression forms.

The order of the subscripts U_j is discussed in connection with type 7.

Type 4. Not used

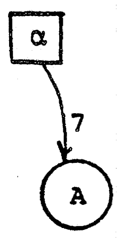
Type 5. Pointing Relationship Edge

For every record X which belongs to a keyed file (SAM or ISAM) we draw an edge:

$X(U_k, .. U_1) \leftarrow POINTER.X(U_k, .. U_1)$.

Type 6. Not Used.

Type 7. Assertion to Target Edge



The local subscript list of α is arranged to have all the lhs subscripts appearing in the order to the left of all the reduced subscripts. Consequently we have an edge:

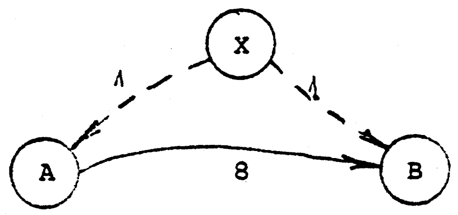
$A(U_k, .. U_1) \leftarrow \alpha\ (U_k, .. U_1, E, E, .. E).$

The number of E's is the number of subscripts of source variables of α which are reduced in α. The order of U_j is as stated in the target variable of α.

Type 8. Siebling Order Edge

These edges are drawn between siebling data items in an input/output file provided they are:

1. Below the record level, or
2. Belong to a sequential file.



The edge drawn is:

$B(U_k, ... U_1) \leftarrow A(U_k, .. U_m, E).$

m=2 if B is repeating, and m=1 otherwise.

The number of E's is the number required to fill out the complete dimension of B (could be only 0 or 1).

Type 9. ENDFILE Edge.

Let F be an input file and R the last record type specified for the file (the rightmost or youngest siebling record node in the tree). Then the following edge is drawn:

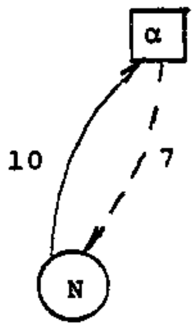
$ENDFILE.F(U_k, .. U_1) \leftarrow R(U_k, .. U_1).$

This edge is drawn only if the user has explicitly mentioned ENDFILE.F in an assertion. Note that the user does not specify the dimensionality of the ENDFILE.F variable. The dimensionality of ENDFILE.F is the same as that of R. This is used in the module DIMPROP to automatically assign the correct dimensionality to ENDFILE.F.

Type 10. Virtual-Subscript^Data _ To Assertion Edge,

These edges connect assertions, which refer to a virtual subscript,to their source and target variables that use a virtual subscript. These connections combine with type 7 edges to make the assertion and its target participate in the same strongly connected component. In the SCHEDULE procedure this will ensure a continuous iteration loop for the nodes that use virtual subscripts.

Feedback from target variable node to the assertion node:



Let the type 7 edge bes

7

N(u_kr..*D^{v}r.-Uj) + $^{a(}$^$_k$* •• U$_v$#«.O$_{1\#}$Er..E)

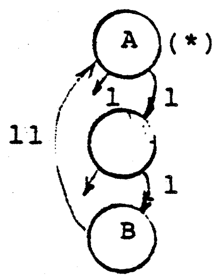
where u^v is a virtual subscript, m is the dimensionality of a and 5 * m-k is the number of E's and the number of reduced subscripts We draw a feedback type 10 edge from N to as

$\alpha(U_m, \ldots, U_{v+\delta}, \ldots U_1) \overset{10}{\leftarrow} N(U_m, \ldots U_{v+\delta}-1, \ldots U_{\delta+1})$

This edge ensures that we do not activate α for $U_m, \ldots U_{v+\delta}$, until $N(U_m, \ldots U_{v+\delta}-1, *, \ldots *)$ is completely utilized.

<u>Type 11.</u> <u>Virtual Subscript Edge - From Source Field Descendent to its Predecessor</u>.

Let A be a repeating node in an input structure, such that the repetition is virtual. Let B be its descendant which is a field.



Then we draw an edge:

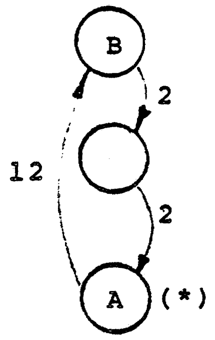
$A(U_k, \ldots U_1) \leftarrow B(U_k, \ldots U_2, U_1^{-1} E, \ldots E)$

The number of E's is the number required to fill up the correct number of subscripts of B.

The meaning of this edge is that since U_1 is a virtual repetition we can have only one instance of the structure with the predecessor A in memory. Consequently this edge ensures that we do not process A for the subscripts $U_k, \ldots U_1$ until we have utilized all the components of the structure for the subscripts $U_k, \ldots U_2, U_1-1$, namely, the previous instance of the structure.

Type 12. Virtual Subscript Edge - From Target or Interim Precededecessor to its Field Descendant.

Let A be a repeating node in a target or interim structure, and let B be its leftmost descendant which is a field.



We draw the type 12 edge:

$B(U_k,..U_m,U_{m-1},..U_1) \leftarrow A(U_k,...U_m,U-1)$

where k-m+1 is the dimensionality of A. The rationale is again avoiding processing the next instance of A until the previous instance is completely defined. Since in output and interim structures the processing of a structure begins with its leftmost field descendant and terminates in its head we made the beginning depend on the end for the previous subscripts value.

Type 13. SIZE edge.

This edge is drawn between the variable SIZE.X (if explicitly mentioned by the user) and the variable X. It has the form

$X(U_k,..u_1) \leftarrow SIZE.X(U_k,..U_2)$.

A 'SIZE' array always has one dimension less than the array it refers to. This is used by the system to assign the proper dimensionality to the structure SIZE.X.

Type 14 END Edge

This edge has the form:

$$X(U_{jv}, \ldots U_{x}) \leftarrow \text{END}.X(U_{jv}, \ldots U_{\wedge} \wedge Ur^{1})$$

The truth value of END.X(U. ,..U$_o$,U-1) determines whether X(U_k,-. .U_i.) is within range.

Type 15 FOUND Edge

Let R be an input keyed record, then we have the edge.

$$\text{FOUND}.R(U_{XL}, \ldots IK_{J.}) \leftarrow R(U_{JC}, \ldots U_{X})$$

The reason for this edge is that 'FOUND' is defined only after the record was read.

~~Type 16 NEXT Edge~~

Let X be a field in an input file, then we have the edge

$$\text{NEXT}.X(U_{K}, \ldots U_{X}) \leftarrow X(U_{K}, \ldots U_{X})$$

This reflects the fact that NEXT.X is read and defined only after X is read.

~~Type 17 SUBSET Edge - Target~~

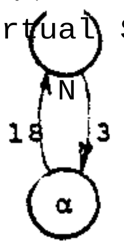
If R is a target record we have the edge

$$R(U_{Jv}, \ldots U_{X}) \leftarrow \text{SUBSET}.R(U_{JC}, \ldots U_{X})$$

This edge ensures that the SUBSET condition is evaluated before the writing of the record.

Data Edge

Type 18 Virtual Subscript-From Assertion To Its Source



For every type 3 edge from a node to an assertion of the form

$\alpha(U_m,..U_\ell,..U_1)\leftarrow N(J_k,..J_v,..J_1)$

where m is the dimensionality of α, k the dimensionality of N, J_v denotes a virtual subscript of N and U_ℓ is the corresponding virtual subscript in α.

J_v has to be of the form U_ℓ or U_ℓ-1 for some $1 \leq \ell \leq m$. U_ℓ may also be reduced in α. The order of the subscripts of α is determined by the order of the subscripts in the target variable of α, plus any reduced subscripts.

We draw the type 18 edge:

$N(J_k,...J_v,...J_1) \leftarrow \alpha(U_m...U_\ell....U_1)$

where each U_i $1 \leq i \leq m$ is may be equal to one of the subscripts of N or to a subscript in another source variable in α. However if $i = \ell$ then $U_i = J_v - 1$.

<u>Type 19</u>. <u>SUBSET Edge - Input</u>

If R is an input record and the user mentioned the variable SUBSET.R we draw the edge:

$SUBSET.R(U_k,...U_1)\leftarrow R(U_k,..U_1)$

This edge ensures that the 'SUBSET' condition will be checked after the record is physically read, enabling skipping the processing of its fields.

<u>Type 20.</u> <u>LEN Edge</u>

If the length of a varying length field X is specified by a LEN.X expression we draw the edge:

$X(U_k,...U_1)\leftarrow LEN.X(U_k,..U_1)$

This ensures that the field is processed only after the LEN.X expression is evaluated.

Type 21. MODULE NAME TO FILE EDGE

This edge indicates the precedence of a MODULE node over the FILE nodes. The MODULE and FILE nodes are sealer.

<u>3.2 Overview of Sub-phases in Network Creation and Analysis</u>

The array graph of a set of MODEL statements is a crucial factor in the MODEL Processor's ability to sequence operations and to detect many inconsistencies and incompletenesses. Table 10 shows a summary of the ten steps or sub-phases involved in the creation and analysis of the array graph (or "network"), and in the determination of the attributes of all the graph nodes. The ten sub-phases are described in sub-sections 3.3.1 through 3.3.13.

Each node, in particular the assertion nodes, may represent an action which should be performed repeatedly, say for each input record read. This will be the case if the assertion either uses a repeating field (directly or indirectly) or defines a repeating field. The requirements for such repetitions may be quite complex and nested, for example an assertion defining a repeating field within a repeating record within a repeating group. In MODEL II the need for repetition is expressed by associating with each node subscripts. One of the form of a subscript variable is FOR_EACH.X where X is some repeating structure. This form explicitly associates an assertion with the repetition on X. The list of these special variables (or repetitions) associated with a node is called the <u>subscript structure</u> of the node. We will compute the subscript structure for each node. This structure is later used to construct the proper iteration control and guide

	Step Name	Summary of Tasks and/or Relationships Searched
1	Creating Dictionary (CPDICT)	Creates a dictionary of all names assigning a "node" number to each (3.3.1)
2	Entering Hierarchical Relationships (ENHRREL)	Searches for hierarchical relationships between a parent and descendant data (3.3.3)
3	Attribute Computation (ENHRREL)	Computes various attributes for each of the dictionary entries, and a list of the iteration subscripts. (3.3.4)
4	Entering Explicit Value Dependencies (ENEXDP)	Searches for explicit value dependency relationships given by assertions (3.3.5)
5	Finding Implicit Predecessors (ENIMDP)	Searches for implicit predecessor to nodes with no explicit predecessor (3.3.8)
6	Calculating Dimensionalities (DIMPROP)	Calculates the dimensionalities of all the variables in the specification, considering the initial declarations and deductions from the edges drawn between nodes. (3.3.9)
7	Filling Up Missing Declarations and Subscripts (FILLSUB)	Based on the previous calculation of dimensions, structures are extended and all subscripted references expanded to the full dimension of their variables. (3.3.10)
8	Assigning Ranges to Subscripts (RNGPROP)	Each iteration for a node is assigned a range. The range specification is "propagated" from nodes which depend on them. (3.3.11)
9	Graph Analysis (AMANAL)	Analyses the array graph to ensure that certain error conditions do not exist (3.3.12)
10	Cycle Detection	Diagnostic Search for possible cycles (3.3.13)

Table 10
Steps in Network Creation and Analysis

the sequencing.

One of the major tasks during this entire phase is detecting logical errors and reporting them to the user. In parallel to searching and entering precedence relationships certain kinds of logical errors are detected, and messages are sent to the user. Further error analysis takes place after the Processor constructs the graph array. A summary of all the error messages produced in this phase, as well as the conditions for their generation is included in Section 3.4.

3.3 Sub-phases of Network Creation and Analysis

This section supplies greater detail on each of the sub-phases of network creation and analysis, and the logical errors that are detected by each phase. References are made to the message numbers of Section 3.4.

3.3.1 Creating a Dictionary of Names and Numbers of Nodes

The Create Dictionary (CRDICT) procedure creates a dictionary of names, assigning a node number to each. These names correspond to the nodes of the array graph. The dictionary data structure (DICT) is an array of strings. An entry is made in the dictionary for each distinct, fully qualified name of each file, record, group, field, or assertion named in the user's MODEL specification; each name roughly corresponding to a statement in the specification. For example, a field name entry corresponds to a field description statement, an assertion name entry corresponds to an assertion statement, etc.

However, there are exceptions to the correspondence between dictionary names and statements in MODEL. If a file is described in MODEL to be both a source and target file, its component record, groups, and fields (described once in the MODEL specification) appear in two separate entries in the dictionary (DICT) because they represent two distinct entities ("OLD" and "NEW"). Furthermore, there are several types of "special names" in a MODEL specification that can be the source or target of an assertion and which become entries in the dictionary. These include names with any of the following prefixes: POINTER, SIZE, LEN, CHOICE, SUBSET, END, ENDFILE, NEXT, FOUND, SUBSET, and declared subscript names. Such special names may be omitted in data description statements.

Instead their description is implicit and the Processor later generates the appropriate statements. They all become nodes in the array graph and therefore need dictionary entries.

Algorithm CRDICT shows the details of the Create Dictionary Procedure. It goes through each entry of the directory and retrieves the corresponding statement (Steps 1-3). Each name is fully qualified with the filename, "OLD" or "NEW" qualifiers, etc, and is entered in the dictionary (Steps 4-8). It also creates entries for the special names explained above (Step 9), and for the subscripts and loop variables (step 10). After this subphase we will refer to each dictionary entry also as a node.

After the dictionary is created all subsequent analysis is performed referring to node numbers which are the ordinal number (index) of the nodes in the dictionary,, In the analysis we have often to retrieve for a given name (possibly qualified) its node number.

The routine DICT# (NAME) returns the node number corresponding to the name NAME. A binary search is conducted on the alphabetized dictionary.

Since the user is not required to always specify the fully qualified name, it is often the case that only a partially qualified name is given and its node number required•

One case is handled by the routine DICTN(NAME) which operates as follows: It tries to find NAME in the dictionary. If it succeeds the node number is returned. Otherwise we check if the name has a prefix. If it has a prefix other than 'OLD' or 'NEW' this prefix is dropped and the search reattempted. If the prefix is NEW or OLD we look for the next component, try to drop it and search again.

Another case is when we are given a name which consists of only the last component. In order to retrieve the code number dictionary entries for simple (unqualified) names are maintained by the algorithm CREASIM.

<u>Algorithm CRDICT: Creating the Dictionary</u>

```
[Subroutines called:  RETRIEVE]

Step 1.  Get next directory entry.

Step 2.  If there are more directory entries, then go to Step 3;
else go to Step 9.

Step 3.  RETRIEVE statements (storage entries) in which the name
is described.

Step 4.  Branch on statement type:

      RECD, then go to Step 7;
      FLD or GRP, then go to Step 6;
      FILE, then go to Step 7;
      Others, then go to Step 5.

Step 5.  Enter name in next entry of dictionary as is; go to Step 1.

Step 6.  Qualify name with its parent file; go to Step 7.

Step 7.  If corresponding file is both a source and a target file,
then go to Step 8; else go to Step 5.

Step 8.  Enter name in dictionary twice:  once with "NEW".  and
once with "OLD".  prefix; go to Step 1.

Step 9.  Using RETRIEVE find all SIZE, LEN, POINTER, END, ENDFILE,
NEXT, FOUND & SUBSET names and enter each one in the dictionary
once.

Step 10. Create system subscripts and loop variables as follows:

         a.  Standard free subscripts SUB1,..SUB9
         b.  System added subscripts $1,..$9
         c.  System loop variables I7,..I9
```

<u>Algorithm CREASIM:</u> <u>Create Simple Dictionary</u>

A special attribute called UNIQUE is added to each node. It is formed as follows:

1. If the node is a special name, i.e. beginning with a prefix excluding NEW_OLD then the UNIQUE field is left blank.
2. If the node's name has a last component which is unique then UNIQUE is assigned the last component.
3. Otherwise UNIQUE is assigned the full name.

<u>Algorithm SIMPLE#:</u> <u>Search a UNIQUE List</u>

Input: A name, presumably a node name.

Output: A node number denoting the dictionary's entry matching the input name.

1. Do first a regular name search by calling DICT#(NAME). If a positive result was returned this is the result.
2. Otherwise extract the last component of NAME and search for a match in the UNIQUE list. The first node whose UNIQUE field matches the last component of NAME is the returned result.
3. If no match is found, the following Error Message is printed:

   "Name error: the following name is missing from the simplified dictionary - NAME".

The algorithm SIMPLE#(NAME) retrieves the code number for a simple name NAME.

3.3.2 Creating the Array Graph and Entering Precedence Relationships Within It

Algorithm CRADJMT (Create Edge Matrix) outlines the creation of the edge structures, including its allocation (Step 1), its initialization (Step 2), and the invocation of subroutines that detect and enter precedence relationships within it (Steps 3). The procedure then proceeds to call other routines (DIMPROP, FILLSUB and RNGPROP) which detect and enter subscript related information. In these subsequent procedures, values are entered in the appropriate edge structures. Certain logical inconsistencies and incompleteness in the MODEL statements can be detected during the construction and analysis of these structures.

Since we have the full dimensions of the nodes only after DIMPROP, which in turn relies on the edges, most of the edges are created without any subscript list. Edges types 3,7 are created with partial subscript lists which are later extended. All other edges are first created with an empty subscript list which is filled up later in FILLSUB. However the field DIMDIF is defined as soon as the edge is generated. This is so because when we find an edge:

$$A(U_1) \leftarrow B$$

we will decide that $\dim(A)-\dim(B) \geq 1$. This means that whenever B's dimension is extended we will extend A's dimension by at least the same number of additional dimensions. Thus corresponding to such an edge we will retain a DIMDIF field of 1.

Algorithm CRADJMT; Creating the Edge Matrix

[Subroutines called: ENHRREL, DIMPROP, FILLSUB, RNGPROP]

Step 1. Allocate edge structures

Step 2. Initialize edge structure

Step 3. Call ENHRREL (Enter Hierarchical Relationships).

Step 4. Call ENEXDP (Enter other Dependency Relationships).

Step 5. Call DIMPROP (Calculate Dimensionalities of variables)

Step 6# Call FILLSUB (Extend structures as needed and fill up subscripts in assertions and in edges)•

Step 7. Call RNGPROP (Identify the ranges of subscripts)

3.3.3 Entering Hierarchical Relationships

Edges of types 1,2,8,9,11 and 12 between files, records, groups, and fields by the routine named ENHRREL (ENter HieRarchical RElationships).

Algorithm ENHRREL consists of parts A,B, and C- Part A enters hierarchical relationships edges of types 1 and 2, Entering the hierarchical types is accomplished by retrieving all the file descriptions (A1,A2) and successively finding the components of each. By means of a recursive procedure (Step A3, ENT^HIER^ADJ) that "climbs" down the implicit hierarchic data structure, each component's direct descendant statements are retrieved in turn and the hierarchical relationship between a parent and its direct descendants is successively entered (Steps 1-7 of ENT^HIER^ADJ).

If the current file is an input file then the edge type is 1

If the current file is an output file then the edge type is 2. Furthermore, if the node is not a lowest level field (Step 10), then its descendants are found, in turn, and the procedure is invoked recursively to insert the hierarchical relationships with their descendants.

Note that the hierarchical relationships "1" and "2" are reversed in direction for precedence purposes (Step 7) because, for example, a record of an input file must be read before its component groups and fields are available, while the record of an output file must be written after its component groups and fields attain a value.

<u>Algorithm ENHRREL: Enter Hierarchical And Structural Relationships</u>

```
Part A:  Enter edges of types 1,2,8,11 and 12 in all files
    A1.  Retrieve all files
    A2.  Get next file name, if none go to Part B
    A3.  Call ENT_HIER_ADJ(Father name, Son name)
         This routine enter edges of type 1,2,8,9,11 and 12
         recursively, until it reaches all field nodes.
   A4.   Go to Step A2
Part B:  Enter edges of types 2,8, 12 in data structures where a data
         node has no parent.  Such structure is assumed to be interim
         and statements for parents will be added later.
    B1:  Find all data nodes which have no parent.
    B2:  Get next node with no parent of none go to Part C.
    B3:  Assign a name to a parent file.
    B4:  Call ENT_HIFR_ADJ(Parent-file-name, head-of-found-structure-name)
         to enter recursively type 2, 8,9,12 edges in the structure
    B5:  Go to step B2.
Part C:  Enter edge of type 9 between ENDFILE.file-name and the records
         in the file.
    C1:  Search directory for all sequential source file names.
    C2:  Get next.  If none go to Part D
    C3:  If the directory does not already have ENDFILE.file-name, add
         a new entry to directory.
    C4:  Search for all record descendents of the file-name.
    C5:  Get next record descendent.  If none go to C2.
    C6:  Enter an edge of type 9, δ=0 between the ENDFILE.file-name node
         and the record node.
```

Algorithm ENTJHIERJUDJ ENTERING HIERARCHICAL DATA STRUCTURES IN ADJACENCY WATKiX

Enter -Hierarchical Relationships in Weighted Adjacency Matrix (a recursive routine) [Subroutines called: RETRIEVE]

Step 1. Qualify parent and direct descendant names.

Step 2. Let i=*dictionary number of parent.

Step 3. Let ^dictionary number of direct descendant.

Step 4. If current file is source only, then go to Step 5;
if current file is target only, then go to Step 6?
if current file is source and target, then go to Step 7*

Step 5. (source only) Draw a 'I[1] edge from i to j, 6 = 1 if j is repeating and 6«0 otherwise; go to Step 8.

Step 6. (target only) Draw a '2' edge from j to i, 5 • -1 if j is repeating and 6*0 otherwise* go to step 8.

Step 7. (source and target)
Set i=dictionary number of "OLD" parent.
Set j«dictionary number of "OLD" direct descendant.
Perform Step 5
Set i-dictionary number of "NEW" parent.
Set j=dictionary number of "NEW" direct descendant.
Perform Step 6.

Step 8. RETRIEVE direct descendant storage entry

Step 9. If one direct descendant storage entry is found, then call it 'son[1] and go to Step 10;
if no direct descendant storage entry found, then go to Step 15;
if more than 1 direct descendant storage entry found, go to Step 16.

Step 10. If type of Son is record, group, or report entry, then go to Step 11;
if type of son is field, then go to Step 14; else system error.

Step 11* Get all of the son's direct descendants.

Step 12a. If the son is an item in a file which is either below the record level or an item in a sequential (actually unkeyed) file, then for each pair of successive direct descendants, k and m, we draw an [f]8 * edge between k and m. 5«0 since this edge is not used in dimension propagation.

Step 12bo (Virtual Self Dependence - Input)
If the son k is a virtual repeating structure in an input file, then draw an edge of type '11' between its rightmost descendant and itself. 6=0.

Step 12c. (Virtual Self Dependence - Output)
If the son is a virtual repeating structure in an output file or interim structure, then draw edge of type '12' between the son k and its leftmost descendant. 6=0.

Algorithm ENT_HIER_ADJ (continued)

```
Step 13.  For each descendant, call ENT_HIER_ADJ recursively to
enter hierarchical relationships between it & its descendants
(go to Step 1).

Step 14.  (field:  no further direct descendants) Return.

Step 15.  Print incompleteness message (#6); go to Step 17.

Step 16.  Print inconsistency message (#4); go to Step 17.

Step 17.  Return.
```

During the scanning of the structure, the structure relationships ('8') are also created in Part C. This relationship happens whenever we have a group or a record which is either an input or an output, belongs to a sequential file or is below the record level. In this case we consider the list of items which are descendants of the group or the record in the order in which they were specified and introduce an '8' edge between any two successive ones. This ensures for example in the specification:

```
                              IN IS FILE (GROUP IS IN_GRP)
                              IN_GRP IS GROUP (A,B)
```

that reading record A and its processing will precede reading record B and processing it.

Certain errors can be detected during this process (Steps 15 and 16 of ENT_HIER_ADJ). If at a given node the indicated descendants do not exist and therefore cannot be retrieved (e.g. if a record X is described to have fields A and B but field B is never (described), then the file layout is poorly-defined due to incompleteness. Likewise, if at a given node more than one descendant with the identical name can be found in the same file (e.g. field X of a given file is described twice with two different sets of attributes), then the file is ill-defined due to an inconsistency. Such problems are reported to the user in the Network Analysis Report in a manner similar to the following (Message numbers 6 and 4, respectively):

```
ERROR(INCOMPLETENESS):  Need a description of X
or
ERROR(INCONSISTENCY):   X is described more than once.
```

Part C in the algorithm ENHRREL defines a precedence relation of type 9 between all nodes describing variable ENDFILE.X. This represents the fact that the ENDFILE.X variable has a meaning only after the relevant records have been read, and hence should be tested only after reading these records.

Routine ENHRREL also constructs several attribute tables for the dictionary entries (nodes). They are described in the next section.

3.3.4 Node Attribute Table

The routine ENHRREL which systematically scans all the data structures also constructs several attributes tables. These tables record different properties of the program nodes (dictionary entries), The list below describes for each of the tables its structure, its significance and the mode of its computation (HOC)• Some of these attributes are generated later and are brought here for completeness* Most of these attributes are stored in a table haying one line for each node. The names of the entries in the table all begin with X, Thus the component DICT, has the table name XDICTc In addition we have for each component a function which gives the node number (t) retrieves this component value, e.g. DICT(I)o *In* some cases the type of the function and the table entry are different, this will be mentioned in the description*

In the list below each table contains DICTIND (the number of dictionary entry) elements, one for each dictionary entry (node).

ENDB - BIT(1) -	is '1' for a node by the name X if there exists a name END.X in the dictionary. This means that the repetition associated with the node X will be terminated by an END.X condition rather than by a repetition specification.
XENDB-INTEGER -	gives the node number of the END.X variable or 0. MOC: For a node by the name X search the dictionary for the name END.X. Computed in ENEXDP.
EXISTB-BIT(1) -	is '1' for a node by the name X if there exists a name SIZE.X anywhere in the dictionary. This means that the repetition associated with the node X will be terminated by count which will be given in the variable SIZE.X. MOC: For a node by the name X, search the dictionary for the name SIZE.X. Computed in ENPTREL.
XESISTB-INTEGER-	Gives the node number of the SIZE.X variable or 0.
INP-BIT(1) -	Is '1' if the node is a data item in an input file. (could be a group, a record or a field) <u>MOC</u>: When scanning a input file all of its descendants have their INP entry set to '1'.
XINP-INTEGER -	Is 1 if input and 0 otherwise.
KEYED-BIT(1) -	Is '1' if the node is a data item in a file for which a key name was specified.

XKEYED-INTEGER - If the file is keyed this contains the node
MOC: When scanning a file with a non blank key name all its number of the field used as key descendants have their KEYED entry set to '1'.

LEN_DAT-INTEGER -The length in bytes of the data item. Applies only to input/output items and not to interim variables. If the item itself or any of its subitems are of variable length the maximal length or number of iterations will be taken if available.
MOC: If the item is a field its length is calculated using the declared (maximal) length and the field's type in the procedure BYTE_CALC. For items on a higher level LEN_DAT of a direct descendant is computed, its product with the (maximal) repetition count is added to the LEN_DAT of its parent. In this way the size of the parent is computed by accumulating the sum of the length of its descendants. For files this is the maximum LEN_DAT of its records.

XLEN_DAT-INTEGER-The table entry name.

MAX_REP-INTEGER -(Table name XMAX_REP). The maximal repetition count which was declared for the item. If not declared as repeating MAX_REP = 1. If an exact count was specified (MIN=MAX) this count is assigned to MAX_REP.
MOC: The value is retrieved from the storage entry of the respective node.

NRECS-INTEGER -	(Table name XNRECS) This count is meaningful only for files, and holds for each file node the number of different records (record types) contained in the file, as declared by the user. For nodes other than files this count is always 0. MOC: For each node corresponding to a record the NRECS count for the node corresponding to its parent file is incremented.
OUP-BIT(1)	Is '1' for items in output files* Table entry XOUP is integer and is > o if item is in output file. MOC: When scanning an output file, the OUP entries for all its descendants are set to '1'.
PAREC-INTEGER	(Table name XPAREC) For items below the record level this will hold the node number of the item's parent record. MOC: When scanning descendants (not necessarily immediate) of a record, set their PAREC entry to the node number of the record*
PARFILE-INTEGER	(Table name XPARFILE) Holds the node number of the parent file for all input-output items. MOC: When scanning descendants of a file, set their PARFILE entry to the node number of the file.

PDIM-INTEGER

(Table name XPDIM)

Holds the dimension allocated to the item in memory. Every record, whether it is repeating or not, is allocated a single area in memory. Every repeating item below the record level is allocated space according to the maximal number of repetitions. The "physical" dimension of an item below the record level (given by PDIM) will therefore be the number of repeating items (including itself) which appear in the ancestry line from its parent record to itself. Thus in the example.

```
Ex.S:         A IS FILE(   B(*))
              B IS GROUP ( C(*))
              C IS RECORD( D(5))
              D IS GROUP (E(2))
              E IS FIELD
```

The PDIM of A,B,C,D,E will be respectively:

0,0,0,1,2

<u>MOC:</u> All items on and above the record level are assigned PDIM=0. Then going down the file tree if an item is not repeating

PDIM(item) = PDIM (its ancestor)

if it is repeating then

PDIM (item) = PDIM (its ancester) + 1

REPTNG - BIT(1)	Is '1' an item is repeating. This is the case if in its specification the maximal repetition > 1 or is specified as (*). Table entry XREPTNG is integer which is positive if item is repeating. MOC: Retrieve the storage entry for its ancestor (where repetition is specified) and check the above.
SUBREC - BIT(1)	Is '1' if the item is below the record level. Table entry XSUBREC is an integer which is positive if item is below record level. MOC: Set the SUBREC entry for a node to '1' if either the SUBREC for its ancestor is '1' or its ancestor is a record.
SUBSLST - PTR	This entry holds a pointer to the local subscript list associated with the node. These are all the subscripts which the node depends on. If the list is empty PTR = NULL, otherwise it points to a list of local subscripts.

The local subscript list is a list of the following structures:

LOCAL_SUB:

NXT_LOCSUB

REDUCED

SUBTYPE

SUBID

IDWITH

RANGE

NXT-LOCSUB - Is a pointer which points to the next structure in the list.

REDUCED Is positive only in subscripts which are reduced in an assertion. It is zero otherwise. This field is meaningful only in assertion nodes.

(SUBTYPE,SUBID) Specify the name of the subscript.

SUBID is the node number of the node associated with the subscript. SUBTYPE distinguishes between four types of subscript names.

1. SUBTYPE • 1, this is a subscript of the form FORJBACH.X associated with the node X.* X has then to be a repeating data node. SUBID is the node number of X,

2. SUBTYPE = 2, A subscript declared by the user as a running subscript.

3. SUBTYPE » 3, A standard free subscript.
One of the list SUB1,SUB2,..SUB9 /

4. SUBTYPE - 4 - A subscript added by the system in the fill-up process. This is one of the first $1,..$9.

In cases 2-4 the subscript appears in the dictional as an independent node entry and SUB(l) contains then its node number. **Cases.*** 1,2 can have a direct range specification, in case 1 by specifying the size of the associated data node X, and in case 2 by explicitly specifying a range in the declaration of the.subscript or including a SIZE or END statement. Cases 3 and 4 are essentially free subscripts and their ranges have to be deduced separately for each statement.

RANGE	Is a node number of a node which has an explicit range specification. If subscript U_i in an assertion has as range entry the number referring to some data node X, then this means that the range of U_i in the current assertion is the same of the declared range (size) of X.
IDWITH	Is used in the scheduling process and will contain the nesting level of the loop variable with which the subscript is identified.

Let us consider for illustration several types of nodes and their local subscript lists, as well as some relevant edges.

Consider the declarations

```
F IS FILE(C(*))
C IS GROUP(R)
R IS RECORD(L(3))
L IS FIELD
```

The local subscript list of L is

```
L: (FOR_EACH.L, FOR_EACH.G
```

The lists for G and R are

```
(FOR_EACH.G)
```

and the list for F is empty.

Consider the assertion:

α:A(I,J)=B(I)+C(J)

The assertion α gets the subscript list (J,I). Note that the list always goes from the least to the most significant subscripts (right to left).

Associated with this assertion will be the edges $3{:}\alpha(I,J)\leftarrow B(I)$ with the subscript expression list (2) and $3{:}\alpha(I,J)\leftarrow C(J)$ with the subscript expression list (1).

Later we may find that B is actually two dimensional while A is three dimensional. This will be reflected in modifying the assertion into

J: A($1,I,J) = B($1,I) + C(J).

The local subscript list of α will be modified to (J,I,$1) while the edge from B to α will contain the subscript expression list: (2,3).

Consider an assertion

$\beta{:}X(I) = \sum_{J} A(I,J)$

The assertion β is given the local list $(\bar{J},I)$ where $\bar{J}$ is marked as reduced. The edge leading into this assertion is

$3{:}\beta(I,J)\leftarrow A(I,J)$ with the subscript expression list (1,2) and the edge leaving into the target variable is:

$7{:}\quad X(\text{FOR\_EACH.X})\leftarrow\beta(\text{FOR\_EACH.X},E)$

<u>MOC</u>: For data nodes this list is obtained by scanning the structure tree bottom up starting at the current *node,* and listing the names of all the repeating data nodes which are encountered. For assertions we take the list of subscripts appearing on the left hand side of the assertion and precede it by a list of the additional subscripts which appear on the right hand side but not on the left hand side. These additional subscripts are marked as reduced. Later when subscripts are addded by the system these lists will be updated

VARS-BIT(l) - (Field entry XVARS, an integer) This entry is 'l' if the structure of the item involving any descendants below the record level is variable. Thus if any subitem has a variable number of repetitions or a variable length the item is assigned a VARS entry of "I'. This will later determine if in reading a record we have to unpack each field or can read the whole record as a single string overlaying the corresponding data structure.

	MOC: Whenever a subrecord item has a varying length or a varying number of repetitions, the VARS entries for both itself and its parent are set to '1'.
VARYREP - BIT(1)	(Field Entry XVARYREP). Is set to '1' if the item has a varying number of repetitions.
	MOC: Inspect in the node's parent storage statement if MAX_REP > MIN_REP or MIN_REP = -1 signifying a '*' repetition, set VARYREP to 1.
VIR_DIM-BIN	Integer (Field entry name XVIR_DIM) This gives for each item the conceptual (virtual) dimensionality that it has. Regardless of the physical memory allocation it counts the number of repeating structures (including itself) which exist on the ancestry line between an item and its parent file. For all input output items this will be equal to the ultimate size of the SUBSLST list.
	MOC: If the item is repeating then VIR_DIM VIR_DIM(item) = VIR_DIM (parent) +1 else VIR_DIM(item) = VIR_DIM(parent)

MAINASS - PTR	(Field name XMAINASS) For each node this entry contains a pointer to the storage statement defining this node. It is prepared in CRDICT.
XDICT - CHAR(32)	Holds the name of the node.
MAMESIZE - Integer -	The length of the node's name.

The corresponding function DICT(I) uses XDICT and NAMESIZE and returns the nodes name as a CHAR(32) varying string.

ISSTARRED	(Field name XISSTARRED, positive if true), is true if the data item is repeating and has a virtual repetition.
DICTYPE	(Field entry name XDICTYPE) CHAR(4) - Specifies the type of the node. It is set during the dictionary creation and whenever new nodes are added to the dictionary. Its possible values are the following:

^{f}ASTX* - An assertion node.

'GRP^1 - A group

'FILE^1 - A file.

^{f}RECD^f - A record

'MODL^1 - The specification name.

'DISK^1, 'PRINT^1, 'CARD^1, 'TAPE^f , 'TERM^1, 'PNCH' - denoting a storage media of the corresponding type,

•SPCN^1 - A special name with a reserved prefix:

(END,SIZE,LEN,POINTER,NEXT,SUBSET,

•ENDFILE,FOUND)

'$SUB' - User or system declared subscripts, including the standard subscripts SUB1,SUB2,...SUB9.

'$$' - System added subscripts: $1, .. $9

'$I' - System loop variables I1,...I9.

"SUCCESSORS - INTEGER - The size of the successor list.

SUCC_LIST - Pointer - A pointer to the list of successors - list of edges emanating from the current node.

#PREDECESSORS - Integer - The size of the predecessor list.

PRED_LIST - Pointer - A pointer to the list of edges coming into the current node.

UNIQUE-CHAR(32) - The smallest name by which this node can be identified. If the last component of DICT is sufficient to uniquely identify this node then UNIQUE is set to this last component. Otherwise, UNIQUE is set to DICT.

FATHER - Integer, The node number of the immediate ancestor of the current node.

SON1 - Integer - The node number of the first (leftmost) immediate descendant of the current node.

BROTHER - Integer The node number of the immediate right neighbor of the current node, or the next immediate descendant of FATHER.

ORGANIZATION - Integer	Equals 1 if this item is a member of a file which is not sequential. It equals 0 otherwise.
TERMC - Integer	Specifies the termination criterion condition for the node if one is explicitly specified: It accepts one of the values: 1- Constant limits given in the repetition specification 2- An END.X variable exists for the current node X. 3- A SIZE.X variable exists for the current node X. 4- This node is a last record or group in an input sequential (unkeyed) file.
RANGEP - Integer	If a termination criterion is not explicitly specified, the system will attempt to deduce one. RANGEP points to the node whose termination criterion (explicitly given) is the same as that of the current node.
NXTNEED - Integer	This entry is positive for all nodes which are referred to by a NEXT prefixed variable. It will also be set for records containing such nodes. These nodes are

	restricted to nodes in input sequential files.
PRVNEED - Integer -	This entry is positive for all nodes which are virtual repeating structures and have a reference of the form A(...I-1,..) is a subscript position in A corresponding to the current node. This will cause these nodes to be declared as repeating of size 2. X(1) will refer to the previous value of x abyte X(2) will refer to the current X. After the loop containing X, X(2) will be moved to X(1).
USED - Integer	This entry is set to 1 for each node which is chosen as a logs variable name. This will govern the selective declaration of the system generated subscripts: SUB1,..SUB9,$1,..$9,$I1,..I9.

3.3.5 Entering Dependency Relationships

Dependency relationships are entered to indicate that a node j, such as a field or assertion depends on the value of another node, i, and that therefore i is precedent to j. These relationships are detected and entered by the routine ENEXDP (ENter Edges for explicit DePendency). Some dependency relationships are explicit in the MODEL statements, while others are implicit and are deduced or assumed by the Processor.

The main tasks of ENEXDP are:

A) Draw edges of the types:

5,13,14,15,16,17,19,20 associated with the special names with reserved prefixes: POINTER,SIZE,END,FOUND,NEXT,SUBSET(Output), SUBSET(Input), LEN respectively.

B) Analyze assertions. Transform the leaves in the assertion syntax tree to reflect node numbers, subscript numbers and function numbers, replacing all the variable name leaves. Deduce and insert subscripts in the case of implicit reduction. Form the local subscript list for the assertion. Generate edges type 3 and 7 into and out of the assertion.

C) Call ENIMDP to generate additional assertions for the definition of fields lacking an explicit source.

Draw edges for special names.

Algorithm ENEXDP: Enter Edges For Explicit Value Dependencies

The algorithm consists of three Tasks, A,B and C.

Task A

This task is performed by the main body of the procedure ENEXDP.

Each node is examined for having a reserved prefix as a first component in its name. In the following let NODE denote the examined node and TRGT denote the subject of the special name, i.e. the suffix to the reserved prefix.

1. If PREFIX = 'POINTER' then verify that TRGT is a keyed record and draw an edge.

   5: TRGT $\leftarrow$ POINTER.TRGT, $\delta=0$.

2. If PREFIX = 'SIZE' then verify that TRGT is repeating and draw an edge:

   13: TRGT(I) $\leftarrow$ SIZE.TRGT, $\delta=1$

   Note that this implies that the dimension of SIZE.X is smaller by 1 than that of X.

3. If PREFIX='END' then verify that TRGT is repeating and draw an edge.

   14: TRGT(I) $\leftarrow$ END.TRGT (I-1), $\delta=0$

4. If PREFIX='FOUND' then verify that TRGT is a keyed record and draw an edge:

   15: FOUND.TRGT $\leftarrow$ TRGT, $\delta=0$

   This will make FOUND.R depend on the record R.

5. If PREFIX = 'NEXT', verify that TRGT is an item below the record level in an input sequential file and draw an edge:

16: NEXT.TRGT «- TRGT , 6=0

This will make NEXT.X depend on X.

6. If PREFIX-^{1}SUBSET^1 then verify that TRGT is a record. If it is an output record we draw an edge %

17: TRGT+SUBSET.TRGT, 5=0.

Otherwise it must be an input record and then we draw the edge:

19: SUBSET.TRGT«-TRGT, 5=0

7. If PREFIX^'LEN^1 then we draw an edge:

20; TRGT«-LEN.TRGT, 5=0

All these edges (including 13,14) are drawn with an empty list of subscript expressions, i.e. SUBX=NULL. The module Fillsub constructs later the subscript expression list according to the edge type.

Task b

Transform Assertions and Draw Edge in and out of the Assertions (types 3 and 7)•

This task is performed by the procedure DOASS which is called for each assertion node.

Task c

We call on ENIMDP to detect non input fields for which no defining assertion is given. ENIMDP creates new defining assertions for such fields and enter then into the syntax analysis phase by recalling SAP. After coming out of ENIMDP, we call once more on DOASS for each of the newly created assertions.

3.3.6 Procedure DOASS

The syntax tree for the assertion is retrieved. Let POINT(1) point to the node representing the L.H.S. of the assertion and POINT(2) represent the R.H.S. We first call SCAN(POINT(1),1,1,0) to transform and construct the local subscript list for the L.H.S. Then, we call SCAN(POINT(2), 0,1,0) to transform and construct edges and augment the Local subscript list for the R.H.S.

When SCAN creates the local subscript list, subscripts are added from the left. Each subscript added to the list is assigned a sequence number which specifies its rank of joining the list or its position measured from the right. Thus in the assertion.

$$\alpha:\quad A(I,J) = \sum_{k} B(I,K,J)$$

we will construct the local subscript list (K,J,I) assigning right position numbers 3,2,1 to K,J,I respectively. The leaves in the assertion referring to subscripts refer to these right position numbers. Similarly when we construct the type 3 edge connecting B to α

$$3:\quad \alpha(I,J,K) \leftarrow B(I,K,J)$$

a subscript expression list (J,K,J) is constructed for this edge. This list is also represented by references to the right position numbers: (2,3,1).

However at the end of the process we would like to have all the references changed to left position number according to the positions of the subscripts in the local subscript list measured from the left. Thus we will have to change the subscipt expression list in the edge above to (2,1,3).

Consequently both the local subscript list and type 3 edges are first generated locally by SCAN.

Then after this is done we resequence the reference numbers of the local subscripts. We rescan the syntax tree for the assertion changing all nodes referring to subscript numbers. We then modify all the subscript expressions list in all the edges and enter these edges into the array graph.

3.3.7 Subprocedure SCAN

SCAN is presented with a node in the syntax tree and is responsible for performing the following tasks:

1) Transform the descendant leaves of this node which are of type variable-name (23) into one of the types: variable number type (25), subscript number type, (26) or function number type (27).

2) Augment the local subscript list by the new subscripts which appear among the descendants of the given tree node.

3) Construct type 3 edge for each instance of a subscripted variable appearing as or among the descendants of the given tree node. These edges will contain a list of subscript expressions.

4) Check special and reduction functions, and if no explicit reduced subscript is given, one is automatically deduced.

<u>Algorithm DOASS</u>

1. Call SCANCPOINTd) ,1,1,0) to scan the L.H.S. subtree of the assertion,
2. Call SCAN(POINT(2),0,1,0) to scan the RVfi.SVsubtree of the assertion,

3* Call RENUMBER to modify all references to subscripts in the syntax tree from right positions to left positions«

4, For each locally generated edge, modify the subscript expression list to refer to left positions, and enter the edge into the graph,

5« Check for subscripts which appear on the R_eH,S« but not on the L.H.S which are not explicitly reduced, Mark then as reduced and issue a warning;

^{f}ENEXDP: SOME SUBSCRIPTS APPEAR ON THE RHS BUT NOT ON THE LHS. SELECTION IS IMPLIED FORs SUB1,SUB2,,..',

6* Generate a type 7 edge from the assertion to its target with the following subscript lists

$(E \hat{} \bar{S} | \cdot «£ y\bar{d}_i\ f | «\bar{S}_k)$

E appears in any position corresponding to reduced subscripts.

<u>Algorithm SCAN(ROOT,LEFT,LEVEL,PARTYPE)</u>

The parameters are:

Root - A pointer to the tree node.

Left - An integer being positive if this node is in the target subtree of the assertion, and equal to zero if the node is in the right hand side subtree.

Level - An integer specifying the depth of the node in the tree.

Partype - An integer, giving the type of the parent of this node.

Description of the algorithm:

1. If the node is a leaf, go to Step 18.
2. I=1, no. of descendants call recursively SCAN(PONT(I),LEFT,Level+1, Node.Type)
3. If the node is not a subscripted variable go to step 12
4. {Subscripted variable}. Scan each of its descendants and construct a subscript expression list element as follows:
5. If the descendant is a simple subscript then LOCAL_SUB# = Subscript number, APR_MODE = 1.
6. Otherwise if the descendant is of the form I-1 we Set: LOCAL_SUB# = subscript number of I
   APR_MODE = 2.
7. Otherwise if the descendant is of the form I-C for C > 0 we set LOCAL_SUB# = Subscript number of I
   APR_MODE = 3.

Algorithm SCAN (continued)

8. Otherwise we set APR_MODE = 4

9. If left > 0 check that APR_MODE = 1, otherwise issue an error message: 'ENEXDP: A GENERAL EXPRESSION APPEARS AS A LEFT HAND SIDE SUBSCRIPT AT - ass-name'.

10. If Left=0, generate a local edge of type 3 from Root to the assertion with the subscript expression list as created in steps 5-8. Set its DIMDIF field to the size of the subscript expression list.

11. Exit.

12. If the node is not a function call, exit.

13. {Function Call}. If not a special function (array function) exit.

14. {Special Function}, Check that level = 1. Otherwise issue an error message: 'ENEXDP: A SPECIAL FUNCTION APPEARS AT AN INTERNAL LEVEL'.

15. If no explicit subscript list appears in the function call, generate one by taking the most recent subscript added to the local subscript list. This is based on the assumption that the summed variable will be explicitly subscripted and will transform A = SUM(B(I)) into A=SUM(B(I),I).

16. If the special function is a reduction function we mark all the subscripts in its parameter list as reduced by setting their REDUCED field in the local subscript list to 1.

Algorithm SCAN (continued)

17. Exit.

18. If the node type is not a variable name (type 23) go to step 24.

19. {Variable Name} If the name is of the form FOR_EACH.X and X is repeating or it refers to a dictionary node which of type subscript ('$SUB') then this name is a subscript name. Otherwise go to step 23.

20. {The name refers to a subscript} Check if a subscript by that name already appears in the local subscript list. If it does not, create a new entry in the local subscript list with this name. go to step 22.

21. {Name already in local subscript list} If left > 0 issue an error message: 'ENEXDP: Two LEFT SUBSCRIPTS COINCIDE'

22. Create a tree leaf of type: Subscript Number(26) referring to the right position of the corresponding entry in the local subscript list. Exit.

23. {The name refers to a variable name}. Construct a tree leaf of type: Variable Number(25) referring to the dictionary entry number corresponding to this variable. Create an edge of type 3 from this entry to the assertion with an empty subscript expression list, DIMDlF=0. Exit.

Algorithm SCAN (continued)

24. If the node type is not a function name (type 21) Exit.

25. {A function name}. Searches for this name in the function list. Create a leaf node of type Function Number (27) referring to its index in the function name list. Exit.

<u>Algorithm RENUMBER</u>

This procedure scans all the leaf nodes of the syntax tree which are of type 'subscript number' (26) and transform their reference number from right position to left position. If the final size of the local subscript list is #LOCALS then this transformation is done by Subscript:

(Left_Position)»# LOCALS+1-(Right Position).

3.3.8 Finding Implicit Predecessors (ENIMDP)

If a field in some target file is not defined via some explicit user's assertion, then the Processor tries to find an implicit source for the field, using a set of successive rules. Also, further analysis is made of the array graph and certain kinds of inconsistency and incompleteness errors are detected. Details of entering such implicit relationships and detecting corresponding errors are in the process called ENtering IMplicit DePendence (ENIMDP), and its subroutines, described here.

First, interim variables are checked to make sure that they have a predecessor. The HASSRC ("HAS SouRCe") function determines whether a node has an explicit predecessor. If an interim field corresponds to node j, then the node is checked to see if it has an explicit predecessor. If so, then the field has a source; otherwise, a message is sent to indicate its absence (Message number 3):

ERROR(INCOMPLETENESS): Need an assertion that describes how to obtain interim name X.

Secondly, all the fields in target files are checked to determine whether they already have an explicit predecessor via the HASSRC function. If a given field in a target file (a field corresponding to, say, node j already has an explicit source by virtue of a user's assertion, then it has an entering edge of type 3. Otherwise, the field has no explicit source the FNDISRC routine (FIND Implicit SouRCe) is called to find a same-named field in another file or a same-named interim field as its source using a set of successive rules in the following order of priority. The idea here is to make some reasonable assumption for a plausible predecessor if at all possible. The following rules are used by the FNDISRC Algorithm.

Rule 1: If the target field having no explicit predecessor is in a file which is both a source and target file, then the value in the corresponding field in the old record is taken as the value of the field in the new record (Message 10 is printed).

Rule 2: If Rule 1 does not apply, then the Processor tries to find a same-named field in a source file. If one is found, it is assumed to be the source and is so indicated in a message containing the assumed assertion (Message 10). If more than one same-named field in a source file is found, then the first is taken as a source and a message is sent to indicate that there was an ambiguity, and the assumed assertion is printed (Message 11).

Rule 3: If no predecessor for the field is found by the above means, then the Processor tries to find a same-named interim field. If one is found, it is taken as the source and a message is sent to indicate that (Message 10). If more than one is found, the first is taken and a message is sent to indicate that there was an ambiguity (Message 11).

Rule 4: If the above efforts are unsuccessful, the Processor tries to find a same-named field in another output file. If one is found it is taken as the source with a corresponding message given to the user (Message 10), and if more than one is found, then one is taken with a corresponding message to the user regarding the ambiguity (Message 11).

Rule 5: In the above cases, the Processor tries to find "implicit" sources for a field if none is given explicitly. If all this still fails to find some field which can be construed to represent the current field's source, then an error message is sent to the user to the effect that the current field has no assertion describing how it is obtained, and that therefore such an assertion is needed (Message 3).

In the above cases where an assumption is made regarding an implicit precedence, the corresponding assertion is printed to the user. A warning is printed as follows: "In the absence of any other relationship, the following assertions have been assumed:", followed by the assumed assertions. The warning (Messages 10 and 11) is produced by the PRSRCWRN routine (PRint SouRCE WaRNing).

The resulting list of such assumed assertions becomes a permanent part of the documentation. The assumed assertion is written out to evaluate whether it agrees with the users original intention or whether some of the statements must be changed and the specification resubmitted.

Each of the assumed assertions is added to the MODEL specification by the procedure CREATASS(I,J) which creates a new assertion.

NODE#J = NODE#I.

CREATASS(SOURCE.TARGET)

•SOURCE[1] - The dictionary number of the source variable.

•TARGET[1] - The dictionary number of the target variable.

The assertion text is created by retrieving the names Of SOURCE, TARGET:

'name(TARGET) - name(SOURCE); END;•

This text is placed in a character string CARD and a pointer to it placed in GN TXPTR which is a pointer variable known to the lexical analysis LEX is SAP,

Then SAP is called. Whenever SAP uses LEX to fetch the next input token, LEX checks first GNTXPTR. If it points to a non empty character string then the next input is taken from this character string. Thus SAP will process the given assertion and form-for it the appropriate entries in the associative memory including the syntax tree. Next it will read the END statement which will cause it to clear GNTXPTR and exit,

On exit we retrieve the assertion name given to the new assertion and create a new entry in the dictionary.

3.3.9 Dimension Propogation (DIMPROP)

This procedure calculates the final number of dimensions (referred here as dimension) for each node in the array graph. Initially every data node is given a dimension specified by its declaration. Thus, for example, a consequence of the definitions:

```
F IS FILE(G)
G IS GROUP(R(*))
R IS RECORD(X(5))
X IS FIELD
```

F and G are assigned dimension 0, R is assigned dimension 1 and X is two dimensional. Data which are not declared (END.X) or are declared as interim fields, not belonging to any higher structure, are (initially) assigned dimension 0.

The process of dimension propagation considers dependencies between nodes and infers a requirement for the dimension of the target of an edge (or an assertion) based on the dimension of the source. Thus if together with the above specification we also had

```
Y IS INTERIM FIELD
Y = X+1
```

we will infer that the dimension of Y should be at least the same as the dimension of X. This inference is based on the assumption that X is actually an abbreviation for X($2,$1) (which will in fact be fully developed into this form later). Assuming in general that unless selection is explicitly specified all rhs subscripts should also appear on the lhs, the full expansion of the complete statement will be:

```
Y($2,$1) = X($2,$1) + 1
```

From this we infer that the should have at least two dimensions. This interpretation is based on the following two rules:

1. Missing subscripts are always inserted on the left of a specified (or empty) string of subscripts.

2. All implicit subscripts that appear on the rhs must also appear on the lhs.

Assume for example that the variable U has been declared as one dimensional and the user specified Z=X+U.

This will be completed into:

```
Z($2,$1)=X($2,$1)+U($1)
```

If instead the user had specified:

```
Z(I)=X(I)+U
```

The completion would have been into

```
Z($1,I)=X($1,I)+U($1)
```

which has of course a different meaning.

The inference of dimensions from assertions is extendable to other edges which also reflect dependencies. For example having the edge

```
5:  R ← POINTER.R
```

we would expect R to be at least of the same dimension as POINTER.R. Since for each possibly different value of POINTER.R we have to retrieve a possibly different R.

The dimension propagation sometime proceeds from target to source. This is the case for example for the edge:

```
13:  X(I)←SIZE.X
```

Here we will want to infer the dimension of SIZE.X which is normally not declared, and make it one less than the dimension of X.

The general dimension propagation will consider therefore both <u>forward</u> and <u>backward</u> propagation. The propagation and its direction depends on the type of the edge. A summary of the algorithm is as follows.

We use an array C for representing the current dimension of a node. Let D represent the initially declared dimension of the nodes. Let N denote the set of nodes in the graph (specification). Below is a simplified algorithm for DIMPROP. This will be followed by the description of the more efficient algorithm in use.

```
1.  For each $n \in N$  Let $C(n) \leftarrow D(n)$
2.  Consider an edge e: $t \leftarrow s$ of type T
    and DIMDIF field ($\delta$) for the edge connecting s to $t.s,t \in N$.
3.  If $T \in \{1,2,3,5,7,9,15,16,19\}$ then
       {Propagate forwards}:
    if $C(s)+\delta > C(t)$ then $C(t) \leftarrow C(s)+\delta$
4.  If $T \in \{13,14,17,20\}$ then
    {Propagate backwards}:
    If $C(t)-\delta > C(s)$ then $C(s) \leftarrow C(t)-\delta$
5.  Repeat steps 2-4 until either
    a.  No further change in the C's is observed
    b.  One of the $C(n), n \in N$ exceeds a given threshold
        (in our case 20).
```

In case (a) we say that the process has converged. In order to verify that the process has converged we have to scan the C vector and check that none of the elements has been modified.

Case (b) is due to a cycle in the graph which if pursued will cause an endless increase in the dimensions. This of course is an error and is flagged as such. Consider for example the (erroneous) specification:

```
G IS GROUP(F(*))
F IS FIELD
IF I=1 THEN H(I)=5 ELSE H(I)=F+1
IF I=1 THEN F(I)=6 ELSE F(I)=H+1
```

The first assertion is interpreted as stating that the dimension of H is larger by 1 than that of F i.e. C(H)>C(F). The second assertion stated in turn that C(F)>C(H).

Applying our algorithm to this specification will result in endless loop of alternately incrementing C(H) and C(F). In order to avoid this we added the overflow case(b) and we check for one of the dimension getting too high.

In order to make the algorithm more efficient we introduce a queue Q which will hold all the nodes whose calculated dimension could possibly be altered.

The more efficient algorithm is:

```
1.   For each n£N let C(n)**D(n), put n in Q,
2m   If Q is empty ~ exit.
3.   Pick a node ntQ, remove it from Q«  Let d-<-0.
4,   For every incoming edge, from s to n of type
     Te{1,2,3,5,7,9,15,16,19} Let d«-ma.x(d,C(s)+6)
5e   For every outgoing edge, leading from n to t of
     type Te{13,14f17#20} Let
     d«-max (d,C(t)-6)
6e   If d S C(n) £O t£ step 2*
1.   {A new updated value} Let C(n)^de
8«   For every incoming edge, leading from s to n of
     type Te{13,14,17,20} put s on Q.
9.   For every outgoing edge, leading from n to t of
     type Tc{1,2,3,5,7,9,15,16,19} put t on Q*
10.  If d > Threshold then halt and issue an error message;
     there exists a propagation cycle.
```

In the program DIMPROP, D is represented by the attribute entry VIR-DIM, C by the array CALC_DIM.

The edges along which forward propagation should take place are characterized by the characteristic array IS_DPROP(I) which is positive for I's of the appropriate edges. Similarly backward propagating edges are characterized by the array BACK_DPROP.

The queue Q is represented by a linked list whose beginning and end are respectively pointed to by FRONT and BACK. The procedure PUT-NODE(n) will put the node n into this list if it is not already there. For quick reference we also use the integer array ONLIST(I) which is positive for nodes number I which are currently in the list.

The DIMPROP algorithm description is stated below.

For completeness we review here the initialdimension assignments of D (VIR__DIM) and the 6 values associated with each edge type*

Let a: $A(I_{ic} ..I_i)$ * $f(...B.(J_m ..J_i) . . .)$ be a typical assertion. For each instance of a subscripted variable such as A(.I, ,..I-), B(J- ,.*J.) we define an apparent dimension as the number of subscripts actually appearing in it. This is smaller or equal to the actual dimension of the appearing variable. The apparent dimension of the assertion itself a is defined to be the number of distinct subscript names appearing on both sides of the assertion.

The initial dimension VXR^DIM of any node n£N is defined as followsz

1. If the node is a data node, this is the dimension as implied by the declaration and the structure in which it is a member.
2. If the node is an assertion then its initial dimension is set to be its apparent dimension as described.

Following is a list of the 6 values associated with each type of edge$_o$

Type 1: <5 « 1, 0 according to whether the target is a repeating item*

Type *2%* 6 « -1,0 according to whether the source is a repeating item*

Types 5,9,14,15,16,17,19,20 all have 6 « 0.-

Type 3: Associated with an instance of a subscripted variable B in an assertion α:

δ = (apparent-dimension(α))-(apparent-dimension (instance of B))

Thus in the assertion:

α: A(I,J) = SUM(B(K,J),K)

The apparent dimension of B is 2, the apparent dimension of α is 3 ((I,J,K) being the local subscript list) and the $\alpha \leftarrow B$ edge has $\delta = 1$

Type 13: $\delta=1$

Type 7: δ=-(numbered reduced subscripts).

This is based on the premise that if the assertion is

α:A(I_k,..I_1)=f(...)

and the local subscript list is

(I_k,..I_1,J_m,..J_1)

J_m,..J_1 being the reduced subscripts, then the generated edge is

A(I_k,..I_1)$\leftarrow\alpha$(I_u,I_1,E.,,E)

whose dimension difference is -m.

δ for the other edges is irrelevant

<u>Algorithm DIMPROP: Dimension Propagation</u>

Calls EXTEND_STRUCTURE

1. Apply the dimension propagation algorithm*

2«, For every node n£N, compare VIR_DIM(n) with CALC^DIM(n). If CALC_DIM(n)=VIRJDIM(n) check next node. If all nodes checked - Exit.

3e { CALC-D.IM(n) >VlRJDIM(n) }• Verify that node is either a special name, a group or a field in an interim structure, or an item in a keyed pointed file. If none of the above hold issue an error message:

•DIMPROP: AN INCOMPATIBLE DIMENSION HAS BEEN COMPUTED FOR AN INPUT-OUTPUT NODE-n[1]

4e Update VIRJDIM(n) «-CALC-DIM(n) •

If the node is not a record nor the top level in an interim structure go to 2 to consider the next node.

5. {Node is either a record or the top level of an interim structure}

Call EXTENDED_STRUCTURE(n, father of n, Difference between CALCJDIM(n) and VIR-DIM(n)),

6. Return to step 2 to consider the next node*

This subprocedure EXTEND_STRUCTURE called by DIMPROP defines additional repeating nodes between the node and its father in the case of a pointed file, or above the top level in case of an interim structure. The number of nodes to be defined is the difference between the calculated and initial dimension.

<u>Algorithm EXTEND_STRUCTURE</u>

Parameters: BOTTOM,TOP,#DIM.

Calls DRAW_EDGES

BOTTOM	is the node above which additional structures have to be generated.
TOP	is its current father. It is the file node in the case of a pointed file and empty if this is an interim structure.
#DIM	The number of additional structures and hence dimensions required.

```
1. If TOP ≠ 0 remove all current edges between TOP
   and BOTTOM.
2. Set LOW to BOTTOM and repeat the following #DIM
   times. When done go to Step 8.
3. Generate a unique NEW_NAME = '$YSGEN$_i$$YSGENi'. Let
   OLD_NAME = DICT(LOW).
4. Generate the text:
   'new_name IS GROUP(old_name(*)); END;'
   and call SAP to process it.
5. Generate a new dictionary entry, m with the following
   field values:
     XDICT←NEW_NAME
    XDICTYPE←'GRP'
    XREPTNG,XISSTARRED←1
    XSON1←LOW. Set REPTNG(LOW), ISSTARRED(LOW)←1 too.
6.  Set HIGH←m and call DRAW_EDGES to draw necessary
    edges between LOW and HIGH.
```

Algorithm EXTEND_STRUCTURE (continued)

```
    7.  Set LOW←m, and return to step 3.
    8.  If TOP≠0 set HIGH←TOP and call DRAW_EDGES to draw
        necessary edges between the newly created top item
        and the old file node above it.
```

Algorithm DRAWJB0GES

Draw edges between two data nodes LOW and HIGH, the second being an immediate ancestor of the first,

```
1. . If items belong to an interim structure or output file
     go to step 4.
2.   {Input items}.  Draw a type 1 edge from HIGH to Low.
3.   If LOW is repeating find its rightmost field descendant s
     and draw a type 11 edge from s to LOW.
     Exit.
4«   {Output items}#  Draw a type 2 edge from LOW to HIGH
5.   If LOW is repeating search its leftmost field descendant s
     and draw a type 12 edge from LOW to s«
```

3.3.10 Filling Missing Subscripts in Edges and Assertions (FILLSUB)

This module performs the following tasks:

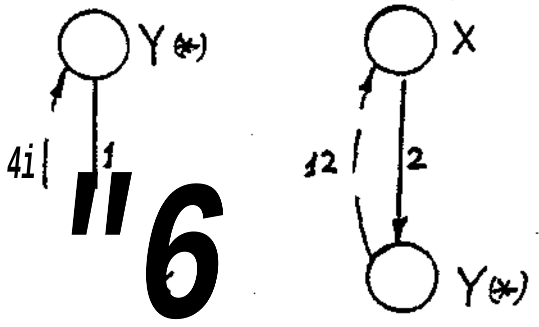
a) Generates and fills up the subscript list for each node.

b) Fills up missing subscripts in the syntax tree for assertions•

c) Generates and fills up the subscript expression lists of all the edges.

d) Draws edges of types 10 and 18 connecting nodes with virtual subscripts to assertions.

e) Draws edges of type 11 and 12 connected from FIELD type nodes X to their ancestors Y, if Y is repeating and FOR EACH.Y is a virtual subscript.



f) For each interim variable END.X and SIZE.X^ the system has built the top level structure if they are arrays themselves,in DIMPROP procedure. Now copy the symbol attributes XISSTARRED and XMAX_REP from the ancestors of X to the ancestors of END.X. This information will be used in the GFLIDCL procedure.

g) Draw edge of type 21 from module name to every file name. This will make the scheduler put the module name in the beginning of the flowchart,

h) Draw edges of types 24 and 25 connecting nodes X to nodes SIZE.X or END.X,if SIZE.X or END,X has virtual subscripts. It is similar to the reverse edge for edge type 3*

Task a: Local Subscript List Generation.

If the node X is a data node, its subscript list is (displayed from last to first) s

(FORJEACH.A_k,... FOR^EACH.A^

Where $A_{]c}$,..A_1'is the list of the repeating ancestors of X in a top down order. If X itself is repeating then. A_x « X.

If the node is an assertion node then it has already been assigned a partial subscript list in ENEXDP* This is the list of apparent subscripts in the assertion, i.e* all the subscripts appearing either on the L.H.S or the R.H.S of the assertion. Let the assertion be of the form

$$a:A(I_{k\#}..l_x) \bullet f(..)$$

Let the R.H.S contain the subscripts $^J w$»«$^J{}_m$ not appearing on the L.H.S and hence assumed to be reduced. Then the partial list assigned to a is {I_k,..$I_{1\#}J_m$,•.J^) and its apparent dimension determined to be d « k .+ m. As a result of the dimension propagation process we will have had recomputed a new dimension for α , $C \geq d$. This will cause $n = C-d$ new subscripts to be added to the list of α which now appears as:

$$(\$n,..\$1,I_k,..I_1,J_m,..J_1)$$

The new subscripts are respectively named \$1,..\$n.

<u>Task b</u>: Filling Up Missing Subscripts in the Assertions.

Consider an instance of a subscripted variable in an assertion $A(I_j,..I_1)$. The calculated dimension VIR_DIM for A yields a value d which should be $d \geq j$. If this is not the case an error message is produced. If $n = d-j>o$ we add n new system added subscripts \$1 to \$n, modifying the instance into $A(\$n,..\$1,I_j,..I_1)$.

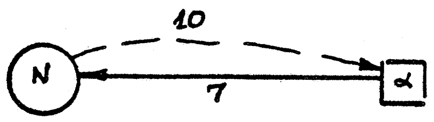
Note that the new subscripts are always added on the left.

<u>Task c</u>: Fill Up the Subscript Expression List for the Edges.

All the edges except types 3 and 7 have been generated with an empty subscript expression list . According to the edge type and the known dimensions of its source and target we generate a subscript expression list. Edges of types 3 and 7 have already a partial list based on their apparent appearance in the assertion. We augment these by adding the subscripts corresponding to \$1,.. \$n where n = dim(A) - apparent dimension in instance.

<u>Task d</u>: Drawing Edges of Types 10 and 18.

Type 10 edges connect a target node to an assertion, whenever one of the dimensions of the target node is virtual. It always reverses a type 7 edge.



Consider first the simple case that α contains no reduced subscripts, then the type 7 edge appears as:

$N(U_k,..U_v,..U_1) \overset{7}{\leftarrow} \alpha(U_k,..U_v,..U_1)$ where we assume that v, $1 \le v \le k$ is the position of the virtual repetition in N. This means that the value of N for subscripts $U_n,..U_1$ depends on the activation of α for the same subscripts. However, before activating α and producing a new value to be stored in $N(U_k,..U_1)$ we have to ensure that the previous value has been used by an assertion needing N. The fact that U_v is virtual implies that N's values for different U_v occupy the same memory space. Thus we draw a '10' edge:

$\alpha(U_k,..U_v,..U_1) \overset{10}{\leftarrow} N(U_k,..U_v-1,..U_1)$

In the case of the presence of reducing subscripts in α we have the 7 edge:

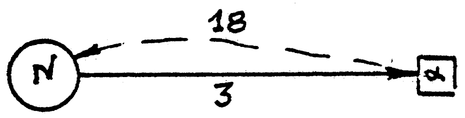
$N(U_k,..U_v,..U_1) \overset{7}{\leftarrow} \alpha(U_k,..U_v,..U_1,E,E..E)$

where m = number of E's, i.e. the number of reduced subscripts. We draw then the '10' edge:

$\alpha(U_k,..U_v,..U_1,J_m,..J_1) \overset{10}{\leftarrow} N(U_k,..U_v-1,..U_1)$

A type 10 edge has to be drawn between N and α for each virtual position in N.

Type 18 edges reverse 3 edges and are drawn from an assertion α to one of its sources N corresponding to a virtual position in N.



Corresponding to a type '3' edge:

$$\alpha(U_m,..U_1) \overset{3}{\leftarrow} N(J_k,..J_v,..J_1)$$

with a virtual position v in N we draw the edge:

$$N(U_k,..U_1) \overset{18}{\leftarrow} \alpha(K_m,..K_1)$$

where the K_i are subscript expressions. $J_k,..J_1$ may in general be a permutation or even partial selection of the $U_k,..U$, and $K_m,..K_1$ is an inverse of the permutation. Also the expression corresponding to the virtual position in N should be reduced by 1.

Assuming J_v to be U_1 or $U_1-\ell$ so that the virtual position in α is the ℓ'th), we take

$$K_\ell = U_v-1$$

K_i for $i \neq \ell = U_p$ if $J_p = U_i$ for some $1 \le j \le k$.

Otherwise $K_i = E$.

<u>Algorithm FILLSUB</u>

Subprocedure used: UPDATE_ASSR,APOD_SUB_REF,CONSEQ,EXPR.

1. Consider in turn each node n∈N.
2. If the node is a data node construct for it the subscript list by upwards tracing of its repeating ancestors.
3. If the node is an assertion it already has a partial subscript list. We augment it by adding on the left subscripts of the form \$1,\$7,.. up to the assertion's dimension. We also call UPDATE_ASSR to fill the missing subscripts in the syntax tree of the assertion.
4. Consider in turn each incoming edge e, entering the node n. Let its type be T. We branch according to the edge's type to different routine, each filling the subscript expression list for an edge of the appropriate type. The adding of an element is done by ADD_SUB_FEF.
5. Scan again each node n∈N which is a data node.
6. Consider in turn each incoming edge e entering n.
7. If the edge e is of type 7 (assertion to data node) then for each virtual position o in the local subscript list of the node n perform step 8.
8. Construct an edge of type 10 reversing the type 7 edge e relative to the virtual position v.
9. Consider in turn each outgoing edge of type 3 (data node to assertion).

Algorithm FILLSUB (continued)

10. Repeat step 11. for each virtual position o in the local subscript list of n.

11. Construct an edge of type 18 reversing the type 3 edge, relative to the virtual position o.

<u>Subprocedure UPDATE_ASSR(ROOT)</u>

This procedure adds missing subscripts in the syntax tree of the assertion pointed to by the pointer ROOT.

The syntax tree is systematically scanned. For each instance of a subscripted variable $A(I_m,..I_1)$, let d be the calculated dimension of A (as obtained in DIMPROP). If $d>m$ then let $e = d-m$ and the instance is replaced by the instance $A(\$Ie,\$I1,I_m,..I_1)$, adding the system generated subscripts \$I1 to \$Ie.

<u>Subprocedure ADD_SUB_REF (SUB#,APRM,TOBACK)</u>

This procedure adds an element to the subscript expression list associated with an edge. It is assumed that two pointers FRONT and BACK have been set to point respectively at the first and last elements of the list. SUB# is the local subscript number in the local subscript list of the node into which the edge enters. A subscript expression element is allocated. Its LOCAL_SUB# field is set to SUB#, its APR_MODE field to APRM. This element is linked to either the back or front of the list according to whether TOBACK is positive or zero respectively. Since subscript expressions are listed from right to left, adding to the back of the list means adding subscripts on the left.

<u>Subprocedure CONSEQ(LOW,NUM)</u>

A procedure for adding 'NUM' subscript expression elements to the back (left) of the list pointed at by the pointers FRONT and BACK respectively. The elements refer to the local subscript positions: Low, Low+1, . .Low__NUM~l. The APR__MODE of all of them is set to 1.

<u>Subprocedure EXPR(NUM)</u>

A procedure for adding 'NUM' subscript expressions of type 4 (general expression) to the back (left) of the list pointed at by the pointers FRONT and BACK.

3.3,11 Range Propagation (RNGPROP)

This module calculates the range of subscripts and dimensions in the specification. Range, or termination criteria must be calculated for:

1. -Each repeating data structure$_e$
2. Each local subscript associated with a node.

Basic termination criteria are always associated with the nodes themselves. Termination of local subscripts is indicated by references to the repeating node which has a range identical with that of the local subscript. A repeating node is said to have a direct range specification if its size was specified by a constant, by an 'END' or 'SIZE^1 descriptor or implied by an end of file encounter. The attribute vector TERMC(OICTINP) of integers provides range information for nodes with direct range specification as follows: It has the values:

1. If the repeating variable has a constant upper limit• This limit is found in the attribute vector MAX_REP(n) for the node n.
2 • If the range is specified by an END.X descriptor.
3. If the range is specified by a SIZE descriptor.
4. If the range is implied by reading an end of file. This criterion applies to any record or group above the record level which is last in its peer group in an input file. It may apply in combination with any of the preceding criteria and then the preceding are

marked in the TERMC array, but both criteria are checked in the generated program.

A repeating node is said to have an indirect range specification if no direct range was specified but one can be inferred from the assertions. For such a node n, TERMC(n)=0, but another array RANGEP(DICTIND) will point to the node which has the same range. Thus RANGEP(n)=m where m has a direct range specification which was inferred from the assertions for n. We limit ourselves to range inferences which are identical with a direct range of another node. If both TERMC(n)=RANGEP(n)=0 after RNGPROP is done, and n is a repeating structure, this is an error and will be flagged as one.

The range specification of local subscripts is always indirect by pointing to a node which has the same range specification. The field RANGE in the structure LOCAL_SUB will be set to the node number which has a range identical with that of the local subscript.

The general process of assigning ranges to repeating nodes and local subscripts can be summarized as follows:

a) Initially, assign direct ranges by defining TERMC for all these nodes which have direct range specification. Assign range pointers to all local subscripts of the form FOR_EACH.X or declared subscripts X such that X has been assigned a direct range.

b) We start an iterative process which successively attempts to assign additional ranges to local subscripts and consequently to repeating nodes. Note that a node is assigned an indirect range (by setting RANGEP) only through the range assignment of a local subscript. The rules for range propagation are the following:

L Whenever a local subscript of the form FOR_JEACH.X is assigned a new range, we set X to have the same indirect range specification, by setting RANGEP(m) where m is the node number- of X« (This is reflected in the procedure UPDATE^SUB)•

2_o For every edge of type « 1, 3, 5, 7, 9, 15_f 16 and the form:

$$A(I ,..!.,..I_1) «- B(..,1.C-'c3_t..)$$

where I_j or I_3-c appears in the k^fth position of B, then if the j'th local subscript in A has no range specification while the k'th local subscript of B has a range specification we assign this specification to I_j «the j'th local subscript of A. This is called forward range propagation.

3, For every edge of type = 2, 3, 7_f 14 and of the form:

$$B(\bullet..1,\bullet.I^{\cdot}) + A(...1^{\cdot}[-c],..)$$

where $I^{\cdot}$ or I_v-c appears in the j^fth position of A, we propagate the range specification of the k'th local subscript in B into the j'th local subscript of A. This is called backwards range specification.

In order to make the iterative process more efficient we maintain a queue of nodes to be processed. The queue is represented by a list of elements of the type PAIR which is linked forward. The pointers CFRONT and CBACK point respectively to the first and last elements in the list. The procedure PUT_NODE (NODE#) adds the node NODE# to the back of the queue, after checking first that this node is not already in the queue. This check is done by consulting the integer vector ONLIST which is an auxiliary record of the nodes which are in the queue. ONLIST(n)>0 if node number n is in the queue. The function REMOVE_CANDIDATE returns the node which is first on the queue. It also appropriately updates the CFRONT, CBACK pointers and the ONLIST array.

Another table expressing dependency of nodes is represented by lists pointed to by the array PROPTO(DICTIND). PROPTO(n) for the node x whose node number is n, points to a list of all the nodes one of whose local subscripts has the name for _EACH.X. Correspondingly, whenever the node n is assigned a range we rescan all the nodes which are in the list PROPTO(n).

<u>Algorithm RNGPROP:</u>

1. Initialization: Initialize the candidate queue (represented by CFRONT, CBACK, ONLIST) to an empty queue. Set the table FORPROP to be 1 for the edges along which we do forward propagation, namely: 1, 3, 5, 7, 9, 15, 16. Similarly set the table BACKPROP to 1 for edge types 2, 3, 7, 14. Also allocate and clear the arrays TERMC, RANGEP, PROPTO.
2. Determine Direct Ranges: Examine in turn each node n. If the VARYREP field of its attribute table is zero, we set its TERMC component to 1 (fixed size range). If its ENDB field is positive we set TERMC to 2. If its EXISTB field is positive TERMC is set to 3. Also, put each node on the candidate queue.
3. Examine Local Subscripts: Check in turn all the local subscripts of node n=1,..DICTIND. Let one of these subscripts have the name FOR_EACH.X or a declared subscript X, where the node number of X is m. If X has a direct range then we set the range pointer of the local subscript to m. If X has no direct range we put n on the list PROPTO(m), so that if later X is assigned a range we will reschedule the scanning of n.
4. This major step iterates the propagation of ranges along edges. It repeats the following substeps until the candidate queue becomes empty.

Algorithm RNGPROP (continued)

4.1 Let I be the element on the queue's top. Remove it from the queue. Spread its list of local subscripts into the auxiliary arrays ASUBT, ASUBID, ARANGE denoting respectively the local subscript type, identity and range. If the subscript is named FOR_EACH.X, or a subscript variable X, check if X has a direct or indirect range specification. If one is available assign it to the range of the local subscript.

4.2 For each incoming edge to the node I if it is one along which forward propagation is to be performed, carry out the forward propagation.

4.3 For each outgoing edge of the right type perform backwards propagation from the local subscripts of the target node into the local subscripts of I.

4.4 If any additional local subscript has been granted a range by the steps 4.1 - 4.3, update the RANGE field in all the local subscripts of node I. Then add to the candidate queue all the nodes which are either:

a) Connected to I by forward edges which are back propagatable.

b) Having I connected to them by edges which are forwards propagatable.

c) Are on the list PROPTO(I).

4.5 Return to 4.1 to consider the next element of the candidate queue.

Algorithm RNGPROP (continued)

5. Print a report for the ranges of the nodes and local subscripts. This is done through the procedure REPORT_RANGES described below.

6. This step defines some additional attribute arrays as following:

   a) If a node X which is an input field has an outgoing edge of type 16 it means that there is a variable by the name NEXT.X. If I is the node number of X and J is the node number of NEXT.X, we set NXTNEED(I)=J. Besides we also set NXTNEED(K)=J where K is the node number of the record which contains X as its subfield.

   b) If the node X, number I, is a record with an incoming edge of type 5, it means that there is a variable called POINTER.X. In this case we set

      PTDTO(I)=1

   c) Checking all subscript expressions in all edges (and hence in all the assertions) we verify that the only expressions appearing in virtual positions are of the form I and I-1. Also, if a virtual position which corresponds to the node number J contains somewhere a subscript of the form I-1, we set

      PRVNEED(J)=1

The following procedures perform auxiliary tasks within RNGPROP:

SEARCH_EDGE(ROOT,TYPE) - This function searches an edge of a given type in an edge list ROOT. ROOT can be the predecessor edge list PRED_LIST or the successor edge list SUCC_LIST of a node. TYPE is an integer between 1 and 24 specifying the sought edge type. If there exists an edge of this type in the given list,the function returns a pointer to the edge. Otherwise it returns the empty pointer null.

UPDATE_DEPENDENTS(NODE#) - This procedure is called to enter into the candidate queue all the nodes whose ranges or the ranges of their local subscripts would be influenced by the range just determined for the node NODE#. It enters into the queue all the nodes which are connected to NODE# by incoming or outgoing edges along which propagation is implied. It also enters into the queue all the elements of the list PROPTO (NODE#).

REMOVE_CANDIDATE - This function returns the element which is first on the candidate queue and removes it from the queue. It also updates ONLIST appropriately.

GETLMN(ROOT,N) - This function returns a pointer to the element number N of a linked list. ROOT points to the first element of the list.

PUT_NODE(NODE#) - This procedure adds the node number NODE# to the end of the candidate queue provided it is not already in the queue. ONLIST is updated appropriately.

ENTERICRIT(NODE#,CRIT#1) - This procedure enters a termination criterion CRIT# which is a number between 1 to 4 into the table TERMC(NODE#). It checks first that this node did not have any previous criterion. If an attempt is made to redefine the criterion for a node the following message is issueds

RNGPROP: THERE IS A MULTIPLE TERMINATION CRITERION FOR VARIABLE variable BOTH $crit_1$ AND critj.

"variable" is the node name.

$crit_1$ and $crit_2$ are each one of the clauses:

CONSTANT LIMITS

END.X SPECIFIED

SIZE.X SPECIFIED

END OF FILE

ENTER_REQ(NODE£,TRGT) - This procedure adds the node TRGT to the list pointed to by PROPTO(NODE#). It is called whenever the node TRGT contains the local subscript FOR_EACH.x where X is the name of node NODE#. This list will be used later whenever the range of the node NODE# is determined to trigger a rescan of the subscript list of the node TRGT.

UPDATE_SUB(J,NEW) - This procedure is called to assign a range to a local subscript number j. The range is given by NEW which is a node number whose range is identical to that of the J' th subscript. It is assumed that the local subscript list of some node has been copied into the tables $ASUBT_{\#}$ ASUBID, ARANGE and J refers to the J^{r}th component of these tables. If

NEW=O or NEW=ARRANGE(J) then no new information is provided and we exit immediately. Otherwise if ARRANGE(J)>O we have a contradictory range specification and the following error message is issued:

RNGPROP: THE SUBSCRIPT - sub HAS BEEN ASSIGNED TWO DIFFERENT RANGES $range_1$ AND $range_2$ THE FIRST ONE IS RETAINED. "sub" is the subscript name. "$range_1$" and "$range_2$" are range specifications of the respective nodes.

Then ARRANGE(J) is set to NEW and the variable UPDATED incremented to mark that at least one subscript was granted a range. If the subscript has the name FOR_EACH.X we check whether node X (number n), has a range specification and compare it with the range given by NEW. If the ranges are contradictory the following error message is printed:

RNGPROP: A MULTIPLE RANGE ASSIGNED TO THE DATA NODE node IN ASSERTION assertion THROUGH THE LOCAL SUBSCRIPT sub THE FIRST RANGE IS-$range_1$ AND THE NEWLY ASSIGNED IS-$range_2$.

Where "node" is the node name, "assertion" the assertion name, "sub" the subscript name, "$range_1$" and "$range_2$" are the contradictory range specifications.

If there is no contradiction then the node X is assigned the indirect range pointer NEW by setting RANGEP(n)=NEW where n is the node number of the node X.

<u>REPORT_RANGES</u> - This procedure prints a report for all the nodes and the local subscripts. The report contains the node and

subscript names and their respectively assigned ranges. The report is organized as follows:

First, under the heading BASIC RANGES we print all the nodes with TERMC>O, i.e. these with direct range specifications. Next, under the heading DEPENDENT RANGES we print all the nodes with RANGEP>O with the format:

node_1 SAME AS node_2

Then under the heading:

RANGE OF SUBSCRIPTS IN ASSERTIONS

We print for each assertion node:

-assertion name-

followed by

$$\begin{array}{ll} \text{sub}_1 & \text{range}_1 \\ \vdots & \vdots \\ \text{sub}_n & \text{range}_n \end{array}$$

3.3.12 Graph Analysis

Although by this time many logical errors in the MODEL statements have been detected during the construction of M, such as the inconsistencies, ambiguities, and incompleteness explained in the previous sections, some of the analysis can be done only after the construction of the graph is complete.

Some examples of the analysis performed at this stage are as follows:

a) If a given row, i, of matrix M corresponds to a field that has no direct descendants, i.e.

$$(\nexists j)(Mij=3)$$

then it is an "unused" field. If the unused field is an output field, then of course there is nothing unusual. If the unused field is a field in a source file, then a warning is sent to indicate that the field is not used in any assertion (Message 5). If the unused field is an interim field then the digraph is incomplete since there is no assertion involving the field, and an error message is sent to this effect (Message 5).

b) If the node, say j, corresponding to a "keyed" input record has no "pointing" source, (i.e. an ISAM file that has no assertion "pointing" to its records)

$$(\nexists i)(Mij=5)$$

then there is no assertion telling how that file relates to other files. The digraph is thus disconnected and therefore incomplete. In such a case, the user is warned that the two or more source file are defined but that there is no relation between the two (Message 8).

c) If a field, j, has more than one assertion as its source, *i.e.* there exist k and 1 such that Mkj=Mlj=7, then a warning message is sent to the user indicating that the two assertions can only hold if they are under mutually exclusive choices, and a corresponding message is sent to the user (Message 9).

d) Another check that needs to be madeis that the targets of all assertions may not themselves be a field in a source file; i.e. if Mij-3 where i corresponds to an assertion, then j may not correspond to a field in a source file (Message 12).

Note that if any errors have been detected during the construction *ox* during the post-analysis; of the array graph, the error count flags the Processor not to proceed to subsequent phases, but to let the user resubmit a corrected specification.

3.3.13 Cycle Detection

Another important type of analysis performed here is the detection of cycles that might exist in the graph. This is necessary to give the MODEL user feedback about possible errors regarding circular definitions.

In order to detect the existence of cycles in the directed graph, we perform a depth-first search systematically scanning all the nodes and edges. This search can be described by the CYCLES algorithm.

Since the full analysis of cycles is done jointly with the scheduling, the current check for cycles is only a preliminary diagnostic check. If it fails then the graph is unschedulable. On the other hand if it passes the test here it may still fail in SCHEDULE.

In actual implementation of the algorithm (in the module CYCLES) several data structures are needed, in addition to those mentioned. They are discussed below:

SUCCL(DICTIND)PTR - Is a compact representation of the graph. Each entry here points to a list of edges. We omit from these edges all these with subscript I on the left hand side and I-c for c>o on the right hand side.

<u>Algorithm CYCLES (THERE_ARE)</u>: Detect cycles in the graph G. Set 'THERE_ARE' to <u>true</u> ('1'B) if cycles exist.

```
1.  Let L be an empty list.
2.  If the graph is empty (no remaining nodes) - terminate.
3.  Pick an arbitrary node of the graph and place it in the
    list L.
4.  Let n be the last element in L.  If n has no successors in the
    graph go to Step 7.  Otherwise let n' be its next successor
    (considered in some ordering).
5.  Check if n' already appears in the list L.  If it does not,
    add n' to the end of L and return to step 4.
6.  A cycle has been detected.  Print the segment of the list
    from the previous appearance of n' to the end.
    Set 'THERE_ARE' to true.  Return to step 4.
7.  (No successors to n).  Remove n from L and delete n and all
    its incident edges from the graph.
    If L is empty return to step 2, otherwise return to step 4.
```

LIVE(DICTIND) BIN - Rather than actually deleting nodes and edges from the graph as is called for in step 7, which is an expensive operation, we maintain a characteristic array LIVE. LIVE(I)=1 if the node is still considered to be in the graph. Otherwise, if LIVE(I)=0 the node is considered to be dead and deleted.

IN-CYCLE(DICTIND) BIN - This characteristic array facilitates the check performed in step 5, if a node n' already appears in the list. Whenever a node n is added to the list, we set In-CYCLE(n)=1. Whenever a node n is deleted we set IN_CYCLE(n)=0. In order to test whether a node is already in the list we only have to check if IN-CYCLE(n)=1.

CYCLE(DICTIND) BIN - Is the program representation of the list L.

It is important to note that CYCLES will not print out all existing cycles, but will detect the presence of a cycle if one (or more) exist.

Consider for example the operation of CYCLES on the graph of figure 12. We list below the status of the list L and major steps performed:

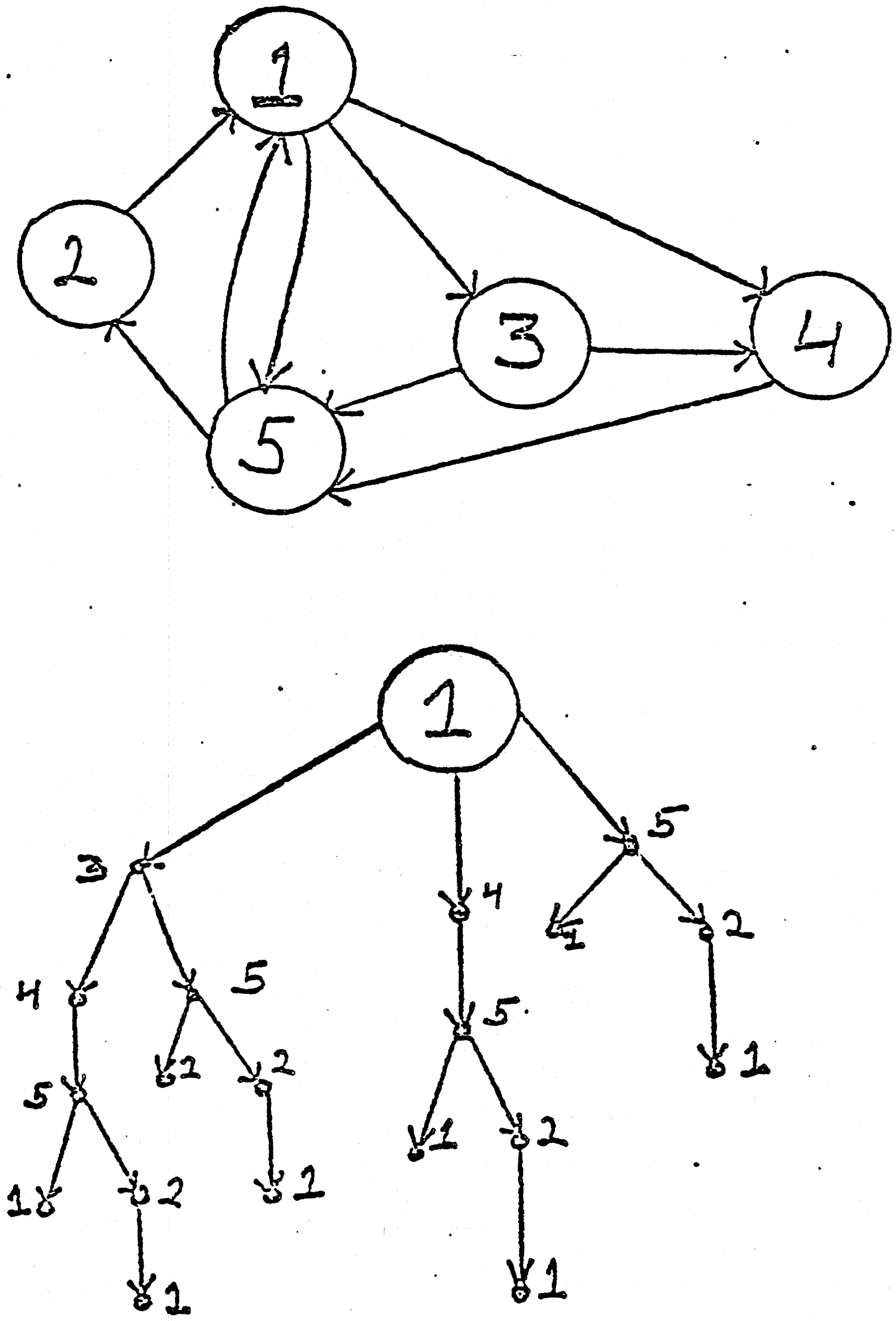


Figure 12   Cycle Enumeration of a Sample Digraph

L: (1)
L: (1,3)
L: (1,3,4) (Successors are taken in increasing order)

L: (1,3,4,5)

Consider 5's first successor, it is 1. A cycle has been detected.

Print: 1,3,4,5

L: (1,3,4,5,2)

Consider 2's successor, it is 1. A cycle is detected.

Print: 1,3,4,5,2

L: (1,3,4,5) Node 2 deleted
L: (1,3) Node 4 deleted
L: (1) Node 3 deleted

L: ( ) Node 1 deleted

End of algorithm.

Note that when we had (1,3) for the second time we did not consider 5 as a successor since it has been deleted. Thus of the 8 existing cycles, only 2 have been printed.

Note that the order of cycles printed by the algorithm is by lexicographic order of the node numbers. Since the corresponding dictionary has been previously alphabetized, the algorithm prints the distinct cycles in alphabetical order.

An example of an illegal cycle in a MODEL digraph would be a set of circular assertions such as the following:

A=B+C

B=C+D

D=C+A

In this example, A depends on B, B depends on D, and D depends on A, an inconsistent cycle. In such a case a message would be sent to the user in the Network Analysis Report indicating the assertions causing the problem (Message 7).

In summary, the above algorithm enumerates some of the distinct cycles in the specification. If there are illegal cycles, the Processor would not proceed to further stages but would let the user re-submit a corrected specification. Normally, however, no cycles would exist and the Processor proceeds to subsequent phases of analysis and design.

3,4 Summary of Errors Detected During Array Graph Analysis Phase

Message 1:

ERROR(INCOMPLETENESS):

Need to know how to obtain field X.

When Issued/Example;

If field X is in a target file or an interim, but no assertion exists that describes how X is obtained and nothing can be deduced.
Example:
X IS FIELD(...)
where X is a field in a target file, but no assertion exists which obtains X.
Issued by routine: FNDISRC

Message 2:

ERROR(INCONSISTENCY):

X is described more than once [Contradictory descriptions of xl

When Issued/Example:

If X is described in 2 or more data description statements in the same file.
Example:
X IS FIELD (CHARC2));
X IS FIELD (NUMERICC9));
where both pertain to the same file; or
Issued by routine: ENHRREL

Message 3:

ERRORCINCOMPLETENESS):
Description of Group or Field X in Y missing.

When Issued/Example:

Y (a file, record, or group) is described to have descendant X, but X is nowhere described.
Example:
Y IS RECORD(X,V,U);
Y IS FIELD (..'•)
U IS FIELD(...)
i.e. description of X is missing.

Issued by routine: ENHRREL

Message 4s

ERRORCINCONSISTENCY):
The following groups of items are circularly described:
• • •

When issued/Example:

When items are described circularly«
Example;
As X=Y+Z?
Bs V=X+W;
C: Y=V+U;

Issued by routine: PRCYCLES (which is called by CYCLES enumeration)

Message 5:

WARNING(POSSIBLE INCOMPLETENESS).:
Nothing is obtained from X.

When Issued/Example:

X is a field in a source file or is an interim name, but it is never used elsewhere in the specification.
Example5
X IS FIELD(...);
X is never used elsewhere in this specification of the module (intentionally or inadvertently).

Issued by routine: AMANAL

Message 6:

WARNING(POSSIBLE AMBIGUITY).
X is given a value by assertions Al, A2, . ..; they must be under mutually exclusive conditions.

When Issued/Example:

More than one assertion describes how X is obtained; may be alright if under mutually exclusive conditions.

```
Example:
Al: SOURCE: CHOICE.C1,Y;
    TARGET: X;
    •«•
A2: SOURCE: CHOICE.C2, W;
    TARGET: X;
    •••
```

This could be alright if Cl and C2 are mutually exclusive.

Issued by routine: AMANAL

Message 7:

WARNING(APPARENT INCOMPLETENESS):
Following assertion assumed:
"X=Y"

When Issued/Example:

When
(1) X was not assigned a value by means of an explicit assertion; and
(2) it was possible for the Processor to find an implicit predecessor using the first applicable of the following rules:
(a) X is in a file which is both source and target, so OLD name is assigned to the NEW name.
Example: NEW.X=*OLD.X;

(b) Y has the same name as X, except that Y appears in one of the source files.
Example: F.X=G.X;
where F is the target file, and G is the source file with the same-named field.

(c) Y has the same name as X, and Y is an interim field.
Example: F.X=INTERIM.X;

(d) Y has the same name as X, and Y is in another target files and already has a value itself.
Example: F.X=G.X;
where G is another target file with the same-named field, which already has a value assigned to it.

Issued by routine: FNDISRC (Rules 1-4)

Message 8:

WARNING(APPARENT AMBIGUITY):
Following assertion is assumed:
"X=Y";

When Issued/Example:

When
(1) X was not assigned a value by means of an explicit assertion; and
(2) the Processor determined an implicit predecessor using the first applicable of the following rules:

(just like the previous set of messages, except that here there is more than one candidate for a predecessor, because of multiple same-named fields in different files, so the first such candidate found is arbitrarily chosen and printed to the user).

(a) (see 7b).

(b) (see 7d).

Issued by routine: FNDISRC (Rules 1-4)

Message 9:

ERROR(INCONSISTENCY):
Field X is a souce-file field and cannot be the target of assertion A.

When Issued/Example:

When X is described to be in a file that is source to the module and X is described to be the target of an assertion.
Example:

```
SOURCE FILES: F,...;
...
F IS FILE(...);
  X IS FIELD (...); (in file F)
```

A: SOURCE: Y;
   TARGET: X;

Issued by: AMANAL

Message 10:

SEMANTIC ERROR: ENEXDP: THE SPECIAL NAME-POINTER.P POINTS TO A NODE WHICH IS NOT A KEYED RECORD

When Issued/Example:

When a POINTER type assertion of the form POINTER.P=F is given, but P is not the name of a keyed record.

Issued By: ENEXDP

Message 11:

SEMANTIC ERROR: ENEXDP: THE SPECIAL NAME-END.X POINTS TO A NON REPEATING NODE

A name of the form END.Y is allowable only when Y is a data name which is repeating. The above message is issued when an "end" name X is detected which does not satisfy this requirement.

Issued by: ENEXDP

Message 12:

SEMANTIC ERROR: ENEXDP: THE SPECIAL NAME-SUBSET.Y POINTS TO A NODE WHICH IS NOT AN OUTPUT RECORD

A name of the form SUBSET.Y is allowed only when Y is an output record name. The above message is issued when Y does not refer to a record name.

Issued by: ENEXDP

Message 13:

SEMANTIC ERROR: ENEXDP: A VARIABLE NAME OR SUFFIX-X IS UNRECOGNIZED IN ASSERTION-A

A name X which is in an assertion A is not found in dictionary, hence not defined in the specification.

Issued by: ENEXDP

Message 14:

SEMANTIC ERROR: ENEXDP: THE SPECIAL NAME-FOUND.X POINTS TO A NODE WHICH IS NOT A KEYED RECORD

A name of the form FOUND.X is allowed only when X is a keyed record name. The above message is issued when X does not refer to a keyed record name.

Issued by: ENEXDP

Message 15:

SEMANTIC ERROR: ENEXDP: THE SPECIAL NAME-SIZE.X POINTS TO A NON REPEATING NODE

A name of the form SIZE.X is allowed only when X is a repeating node. The above message is issued when X is not a repeating node.

Issued by: ENEXDP

Message 16:

SEMANTIC ERROR: ENEXDP: THE SPECIAL NAME-LEN.X POINTS TO A NODE WHICH IS NOT A FIELD

Issued by: ENEXDP

Message 17:

SEMANTIC ERROR: ENEXDP: A SPECIAL NAME-NEXT.X POINTS TO A NODE WHICH IS NOT AN INPUT FIELD

A name of the form NEXT.X is allowed only when X is an input field name. The above message is issued when X does not refer to an input field name.

Issued by: ENEXDP

Message 18:

NAME ERROR: the following name is missing from the simplified dictionary - X. This implies ambiguous use of X as a simplified name.

The single component name X is missing from the simple name dictionary.

Message 19:

ENHRREL: END FILE PREFIXES A NON EXISTENT FILE - name.

This message is issued when a name of the form ENDFILE.X is encountered and X is not declared as a file.

Message 20:

name UNDEF:

A message produced by ENHRREL when an item "name" is listed as a descendant of a group, a record or a file but there is no definition of "name" itself.

Message 21:

SEMANTIC ERROR: SCHEDULE: NO RANGE DETERMINED FOR LOOP VARIABLE AT LEVEL-M AT CYCLE-N1, N2,...

In a strongly connected component, we have found a subscript candidate for the loop but there is no range defined for this subscript variable, it is an error.

Issued by: SCHEDULE

Message 22:

SEMANTIC ERROR: DIMPROP: THE DIMENSION PROPAGATION IS IN AN INFINITE LOOP!

The nodes involved are also listed.

Issued by: DIMPROP

Message 23 :

SEMANTIC ERROR: DIMPROP: THE I/O NODE-X HAS INCOMPATIBLE DIMENSION. THE DIFFERENCE IS: N

If node name X is in input or output file and not in a keyed file. The dimension of X can't be extended.

Issued by: DIMPROP

Message 24:

SEMANTIC ERROR: RNGPROP: AN ILLEGAL SUBSCRIPT EXPRESSION IN A VIRTUAL POSITION. SUBSCRIPT 2S : X IN A DEPENDENCY OF T ON S

If there is an edge from node S to node T and X is a virtual subscript position of node S, the subscript expression of X should be either I or '2-1'. Otherwise, above message will be issued.

Issued bys RNGPROP

Message 25s

SEMANTIC ERROR: RNGPROP: THERE IS A MULTIPLE TERMINATION CRITERION FOR VARIABLE-X BOTH Tl AND T2

In range propagation procedure, we find that different (not equal) multiple termination criterions, Tl and T2, are assigned to variable X.

Issued by: RNGPROP

Message 26:

WARNING: RNGPROP: THE SUBSCRIPT-S HAS BEEN ASSIGNED TWO DIFFERENT RANGES: Rl AND R2 THE FIRST ONE IS RETAINED

In range propagation procedure, we find that both ranges Rl and R2, equal or different, are assigned to subscript S. We will arbitrarily choose Rl as its range,

Issued by: RNGPROP

Message 27:

SEMANTIC ERROR: SCHEDULE: A CYCLE DETECTED

In SCHEDULE procedure, if there is a strongly connected component which has more than one node and at least one node doesn't have available subscript candidate, it is a cycle. The node names in the strongly connected component are all listed.

Issued by: SCHEDULE

Message 28:

SEMANTIC ERROR: SCHEDULE: NO CANDIDATE SUBSCRIPT IN CYCLE

In SCHEDULE procedure, if there is a strongly connected component which has more than one node and every node has an available subscript candidate but we just can't find a subscript candidate out of them, it is a cycle. The nodes in the strongly connected component are listed.

Issued by: SCHEDULE

Message 29:

WARNING: SCHEDULE: A RANGE CONFLICT IN NODE X BETWEEN THE ALREADY ASSIGNED RANGE: R1 AND THE NEWLY IMPLIED RANGE: R2

In a strongly connected component, if we find there is a subscript candidate for the loop and the range of the subscript candidate in different node is conflicting, this possibly is an error.

Issued by: SCHEDULE

4. AUTOMATIC PROGRAM DESIGN AND DETERMINATION OF SEQUENCE AND CONTROL LOGIC

This section is concerned primarily with the creation of a flowchart for the specified program based on the array graph. It also performs additional checks of consistency and produces messages and a flowchart report. The constructed flowchart is used in the subsequent code generation phase to generate the program for the MODEL specification.

This phase consists essentially of three parts described in respective subsections. The first part is the SCHEDULE procedure which creates a preliminary schedule table. It consists of two recursive subprocedures SCHEDULE_GRAPH and SCHEDULE_COMPONENT which essentially order the nodes in the array graph into a linear order to which it adds iteration control statements. This linear order can be interpreted in the next phase, generally, entry by entry, to create the desired program. SCHEDULE also checks for circular definitions which are reported as errors.

The second part consists of the procedure FLOWOPT. Its task is to process the schedule produced by SCHEDULE and reorder the entries as appropriate to enlarge scope of the iterations, thereby producing a flowchart for a more efficient program.

Finally, the third part consists of the procedure GFLTRPT whose task is to produce a flowchart report that is available optionally to the user.

(1) The P(n) local subscript in node n,Ip(n),is still available.

(2) In any edge N(I_m, . . I_i)-*-M(J_s, . . JjJ where the J_i are expressions involving I^r».I_m we require that $J_{p(m}$ j=$I_{p(n)}$ C-c] i.e. the position corresponding to the loop variable is consistent in all edges. Since in the assertion or dependency corresponding to this edge the loop variable is to be identified with $*_{p(n)}$, we require that $I_{P(n)}$ occupies the position allocated to the loop variable in M,

If we cannot find a candidate identification satisfying (1) and (2) the graph cannot be scheduled and we issue an error message.

Otherwise we set all the ID.WITH fields of the local subscripts $I_{pIn)}$ .for each nGN to 1 thus noting that these subscripts have *been* identified with the loop variable at level 1. We also remove all edges of the form.

N(I_m, . . 1^)«•(. . •Ip(n)-?*••} f o r c > o since they imply dependency on values from the previous iteration of the same loop. Denoting the graph thus modified by G^1, which may have ceased to be strongly connected by the removal of the edges, we call S=SCHEDULE_GRAPH(G^1,1+d) to Schedule G^1. Let us denote the newly introduced loop variable by v^• Then the schedule returned by the current procedure is: for v^ do^ *S* end {v^}.

When the program will be generated, all the subscripts whose IDWITH field has been set to 1 will be replaced by v^.

4.1.3 Representations

A graph is represented by a list of elements of type GNODE, each having the following fields:

NXT_GNODE - A pointer to the next element in the list.

NODE_ID - The node number (in the directory) of the element.

SUXL - Pointer to a list of edges connecting this element to its successors. Initially this is identical to the SUCC_LIST list. But as the process proceeds some of these edges are removed from this list.

A strongly connected component is represented by the structure COMP having the fields:

NXT_COMP - Pointing to the next component.

NODE_LIST - Pointer to a graph which comprises the component.

A schedule is a list of schedule elements each of which is either a node-element or a for-element.

A node-element is declared as a structure NELMNT having the fields:

NXT_NLMN - Pointer to the next element in the schedule.

NLMN_TYPE - An integer, always equal to 1 for node elements.

NODE# - The node number.

A for-element is declared as a structure FELMNT having the fields:

NXT_FLMN - Pointer to the next element in the schedule.

FLMN_TYPE - Always equal to 2, denoting this is a for-element.

ELMNT_LIST - Pointer to a schedule which is the scope of the for-loop.

FOR_NAME - The node number of the loop variable which can be a FOR_EACH.X and then FOR_NAME is the node number of X, or it can be a declared subscript.

FOR_RANGE - The node number specifying the range of the loop variable in the loop.

4.1.4 The Main Program of Schedule

The main body of SCHEDULE starts by constructing a graph, i.e. a linked list of structures of the type GNODE representing the complete specification. Each I=1,..DICTIND is allocated an element structure with its NODE_ID field equal to I and its SUXL field pointing to a copy of the list SUCC_LIST(I). We also set the array NODEP(I) to point to the graph's element. This is necessary since the edges in SUXL refer to node numbers which we should translate to graph's elements. We then call once:

```
          FLOWCRT=SCHEDULE_GRAPH(MAING,1)
```

where MAING is a pointer to the complete graph, 1 is the initial level, and FLOWCRT is a global pointer to the schedule which later procedures use to retrieve the schedule.

4.1.5 Finding Strongly Connected Components

One of the basic processes in the procedure is that of finding maximal strongly connected components. We follow Tarjan's algorithm based on depth first search as described in Aho, Hopcroft and Ullman's book: The Design and Analysis of Algorithms. The main part of the algorithm is the recursive procedure SEARCH(V) which is presented with a node v and identifies all the strongly connected components reachable from V. It utilizes the arrays DFNUMBER, LOWLINK which are preset to $\emptyset$ for all nodes, the global variable COUNT and a stack called STACK. In our implementation we also employ an array ONSTACK which is positive for node n if it is currently on the stack.

STRONG(G) is a function which accepts a graph as a parameter and returns a sorted list of its strongly connected components. The procedure SCHEDULE_GRAPH has been described above.

Algorithm SEARCHC(v)

```
1.  COUNT: = COUNT+1, DFNUMBER(v), LOWLINK(v): = COUNT.
    Put v on the stack.
2.  Repeat the following substeps for each node w a direct
    descendant of v.
    2.1  If DFNUMBER(w)=o this is a new node not searched before.
         We call SEARCHC(w) and then let LOWLINK(v)=min
         (LOWLINK(v), LOWLINK(w)).
    2.2  Else, if DFNUMBER(w)>o and w is on the stack, then let
         LOWLINK(v)=min (DFNUMBER(w), LOWLINK(v)).
3.  If LOWLINK(v)< DFNUMBER(v) return.
4.  Else, LOWLINK(v)=DFNUMBER(v) and this is a root of a strongly
    connected component.  All the elements (above and including v)
    on the stack are successively unstacked and linked together
    into a list - a subgraph which is defined as a component.
    This component is placed at the head of a list of components
    pointed to by the variable COMP_LIST.  In addition we maintain
    a running component number COMP_CNT and set the array
    COMP#(w)=COMP_CNT for each w in the current component.
```

Note that the algorithm returns a list of components which are ordered consistently with the dependency order.

<u>Algorithm STRONG(G)</u>

1. Clear the stack, the component count, the list of components and the variable count. For each veG set

   DFNUMBER(v)=o

2. For each veG such that DPNUMBER(v)«o call SEARCHC(v) to add the components reachable from v to the top of the component list,

3* Delete from the graph all the edges which connect nodes in different components*

4. Return as a result the component list»

Algorithm SCHEDULE_COMPONENT(G,1)

```
1.  For each node n∈G compute the number of free local subscripts.
    These are local subscripts whose IDWITH=0 which implies that
    they have not yet been identified with any loop variable.
    Let MINFREE be the minimal number of free subscripts over
    all n∈G.
2.  If MINFREE=0 and |G|=1 we return a schedule of one node
    element containing the single node in G.  Exit.
3.  If MINFREE=o and |G|>1 this is an error.  The message:
    SCHEDULE: A CYCLE DETECTED is printed and then the procedure
    PRINT_CYCLE is called to print the remaining cycle.  Return
    an empty schedule and exit.
4.  Otherwise we have to search for candidate identification.
    We start by constructing in the array stack (denoted here
    by S) a list of the graph nodes such that for every i>1 the
    node S[i] has an edge incoming from some S[j] j<i.  This is
    done by the following iterative process:
    4.1  Let S[1] be the first node in G.  Let I=1.
    4.2  Repeat the following steps as long as I< |G|.
         4.2.1  Let n:=S[I].
         4.2.2. For each descendant of n which is not already
                on S, add it to S.
         4.2.3  I:=I+1, return to 4.2.1.
```

Algorithm SCHEDULE_COMPONENT (continued)

5. Let IDF be the node S[1]. Let POS range over all the free subscripts of IDF. Repeat steps 6-13 for each available subscript.

6. Clear POSITION for all nodes in the graph, and then set POSITION [IDF]:=POS.

7. Repeat steps 8-12 for I=1 to |G|.

8. Let n:=S[I], POSJ=POSITION(n) and consider each edge from node n to any other node t.

   $t(I_m,..I_1) \leftarrow n(E_s,..E_1)$

   Consider the subscript expression E_{POSJ} which corresponds to the identified position in n.

9. If E_{POSJ} is not a simple expression ($I_j[-c]$) or if the subscript is reduced then POSJ cannot be identified with a loop variable for the strongly connected component, exit to step 14 to consider the next value of POS.

10. If $E_{POSJ}=I_j[-c]$ for some $\lambda < j \leq m$, check if POSITION [t]>o. If POSITION [t]>o and POSITION [t]≠j there is a conflicting identification in the subscripts of t. Exit to step 14.

11. If POSITION [t]=o set POSITION [t]:=j.

12. Return to step 8 for the next I.

13. Arriving here means that a complete identification was successfully performed. Go to step 16.

14. The identification starting with position POS for node IDF has failed. If another free subscript for IDF is available set POS to it and return to step 6.

Algorithm SCHEDULE_COMPONENT (continued)

15. Arriving here means that no identification is possible. The message: "SCHEDULE: NO CANDIDATE SUBSCRIPT IN CYCLE" is printed, followed by a list of the nodes in the graph. Return the empty schedule and exit.

16. A successful identification! We proceed to determine name and range for the loop variable and to delete edges which are of the form

$$A(,..I_p,..I_i..) \leftarrow B(,..I_p - c,..)$$

where p is the identified position of A.

17. Allocate a for-element for the schedule with empty FOR_RANGE, FOR_NAME fields.

18. Scan each node in the graph, $n \in G$. Let p=POSITION[n].

19. Examine the local subscript I_p. Set its IDWITH field to 1_P, the level parameter. If I_p has a range and FOR_RANGE is empty yet set FOR_RANGE to the range of I_P.

20. If FOR_RANGE has a previous value which is different than the range of I_p, print the following warning message:

    SCHEDULE: A RANGE CONFLICT IN NODE node-name$_1$
    BETWEEN THE ALREADY ASSIGNED RANGE: range$_1$ AND
    THE NEWLY IMPLIED RANGE: range$_2$

    "node_name$_1$" is the name of node n.
    "range$_1$" and "range$_2$" are respectively node names whose range is assumed by the loop variable and the local subscript I_P.

21. If I_p has a range and FOR_NAME is empty yet assign to FOR_NAME the name associated with I_p, provided it does not

Algorithm SCHEDULE_COMPONENT (continued)

coincide with the names assigned to the loop variables of the enclosing loops. These names (or node numbers of the names) are kept in the array PAST_NAMES.

22. Delete from the graph any edge of the form

$$t(\quad) \leftarrow r(E_s,..E_p,..E_1)$$

where E_p is of the form I_k-c for some k and c>o. These correspond to dependencies on values created during the previous iteration and hence should be ignored.

23. Repeat steps 18 to 22 for all nodes in the graph.

24. If FOR_NAME is still empty define a system name of the form $I1.

25. Save FOR_NAME in PAST_NAMES(1).

26. If FOR_RANGE is empty issue the error message:

"SCHEDULE: NO RANGE DETERMINED FOR LOOP VARIABLES AT LEVEL - AT CYCLE - "

followed by a printout of the component.

27. Call SCHEDULE_GRAPH (G,1+1) to further schedule the component. Set the field ELMNT_LIST of the for-element to the schedule returned by SCHEDULE_GRAPH.

Return as a result the for-element.

The following subprocedures are defined with SCHEDULE_COMPONENT:

PRINT_CYCLE: Prints the names of all the nodes in the current component.

Algorithm SCHEDULE_COMPONENT (continued)

CONCATENATE (A,B): Concatenate the list B to the end of the list A. A and B are pointers to general lists.

FREE_PPAIR_LIST (LIST): Frees the space allocated to a list of PPAIR structures pointed to by LIST.

4.2 Loop Optimization (FLOWOPT)

The schedule generated by SCHEDULE was designed for correctness with no considerations for efficiency. Its tendency was to split large loops into small ones wherever possible. Considering efficiency it is much more economic to have maximal loop scopes. The advantages are not only in reduced overhead which is involved with the maintenance of separate loops but also in possible saving in memory space.

Thus by merging the two loops:

```
          for J do C(J) = A(J)*2
          for J do B(J) = A(J)+C(J)
```

we would reduce the required dimension of C if it is an interim variable not used elsewhere:

```
          for J do
               C = A(J)*2
               B(J) = A(J)+C
          end J
```

The main part of FLOWOPT just calls on the recursive procedure OPTIMIZE_LIST (FLOWCRT, 1) with parameters FLOWCRT pointing to the complete schedule, and 1 being the initial level.

4.2.1 OPTIMIZE_LIST (LIST, LEVEL)

This procedure optimizes a schedule by recursively performing two operations on it:

(a) Omitting all nodes which imply no action and loop consisting exclusively of such nodes.

(b) Merging every two contiguous loops whose loop variables have equal ranges and where the subscript positions identified with the loop variables are the same in the two loops.

Since a schedule is a structured object we use a function FORM_LIST (ELMNT_LIST) which spreads a schedule ELMNT_LIST into a flat list of nodes. These lists are all represented by the array NXTLINK which contains for a node n its successor in the flat list. FORM_LIST returns as a value the node number of the first element in the flat list.

Algorithm OPTIMIZE^LIST (LIST/ LEVEL)

1. Let CURRENT point to successive elements in the schedule list. PREVIOUS points to the previous element,
2. If the current element is a, node element go on to step 13 to continue scanning the schedule*
3. The current element is a for-element. Let NEXT point to the next element in the schedule* Call FORMALIST to form a flat list of all the nodes within the scope of the for-element CURRENT. Let CLIST be the first node in the flat listo
4. Scan the li$t pointed to by CLIST. If any of the nodes in the list is either an assertion or a data item (field, group, record or a file) in an input or output file, then the list should not be cancelled. Continue at step 6.
5. Otherwise the for-list is cancelled. We skip the for-element pointed to by CURRENT and omit it from the list SCHEDULE* Go on to step 13 to continue scanning the schedule«
6. Test if the element pointed by NEXT can be merged with CURRENT, A boolean variable MERGE is set to 'false' if such a merge is impossible. The testing is done in steps 7-10.
7. If NEXT is empty or points to a node-element, or points to a for-element with a range which is different from that of CURRENT then MERGE is set to 'false'. Go to step 12 to advance CURRENT and NEXT to the next two elements.

Algorithm OPTIMIZE_LIST (continued)

8. Form a flat list out of the nodes of the NEXT element. Let NLIST be the pointer to the flat list.

9. Run over the nodes in the NLIST. For each $n \in$ NLIST set POSITION (n) to the position of the local subscript which is identified with the loop variable at level LEVEL. This is a local subscript whose IDWITH field is equal to LEVEL.

10. Consider now all the nodes in CLIST (the list of nodes in the current for-element). Let $n \in$ CLIST. Find the position of its local subscript corresponding to the level LEVEL. Let it be POS. Consider every edge

$$m(,..I_p,..) \leftarrow n(E_s,..E_{POS},..E_1)$$

where $m \in$ NLIST, p=POSITION(M).

Verify that $E_{POS}=I_p[-c]$. If this is not the case set MERGE to 'false' and go to step 12. This check confirms the consistency of the loop variable positions between the variables in CLIST and the variables in NLIST.

11. If the tests have been passed for all edges from CLIST to NLIST, the two for-elements can be merged. This is done by calling the procedure CONCATENATE which concatenates the lists of elements from the current for-element and the list of elements from the NEXT for-element. Then merge the flat lists CLIST and NLIST. Set NEXT to the element following the element just merged and return to step 6 trying to increase the scope of CURRENT even farther.

Algorithm 0PTIMI2E_LIST (continued)

12. No more merges of contiguous elements are possible., Call OPTIMIZEJLIST (CURRENT, LEVEL+1) recursively to optimize the schedule nested within the CURRENT for-element with level LEVEL+1..

13o Advance CURRENT to the next element. If the schedule scanning is not complete return to step L Otherwise exit.

<u>Subprocedures</u>:

FORMALIST (ELEMENT): Forms a flat list of all the nodes enclosed within the schedule ELEMENT. All the elements in the schedule are scanned. If an element is a node element it is added to the list. If it is a for-element we call FORMALIST recursively with its contained element list and append the returned flat list to the accumulated flat list.

CONCATENATE!(A,B): A procedure which concatenates two element lists,

4.3 The Flowchart Report (GFLTRPT)

This module produces a report of the schedule . The report includes a line for each node delineating its type, name and attributes and the action associated with it. In addition it describes the iteration structure working the opening and closing of loops.

The actual task is performed by the recursive procedure PRINT^SCHEDULE. The main part of GFLTRPT produces a title line with captions heading the appropriate columns of the report and then calls P.RINT^SCHEDULE with a pointer to the full schedule, given in FLOWCRT.

PRINT^SCHEDULE (ELEMENT): This procedure prints the report of the schedule pointed by ELEMENT. In general this schedule is a list of elements. All node-elements are printed directly. For for-elexae.nt the report includes statements describing the opening and closing of the top loop associated with this top element. We then call PRINT^SCHEDULE recursively to report the elements on the higher level.

The operation of PRINT_jSCHEDULE is again split into several subtasks, and can be described as follows:

1. Consider each element of the element list pointed to by ELEMENT.
2, If the element is a node-element call PRINT_NODE to print a report line for the node. If the node is a for-element, call PRINT_F0R to report

the opening of a loop, call PRINTJSCHEDULE (ELEMENTJLIST) to report the schedule within the loop's scope, and then call PRINT_J3ND to report the closing of the loop.

3. Consider the next element in the schedule. If the schedule is not finished return to step 1. Otherwise exit.

Each line in the report consists of the following fields arranged sequentially:

CC - for controlling the skipping of lines.

NODE# > The node's number„

NAME - The node's name.

DESCRIPTION - The type of the node.

EVENT - The action associated with the node,

PRINT—TOR: This procedure reports the opening of a loop corresponding to a given for-element* The line printed has in its DESCRIPTION field the message ITERATION and its EVENT field iss

FOfiL name UNTIL termination*

Here "name" is the loop variable name. "Termination" is the termination condition. If there is no determined termination condition it assumes the value:

WHO KNOWS? (ERROR)

signifying an error.

PRINT^SNDs This procedure prints a report line with a DESCRIPTION field: END ITERATION FOR name

where "name" is the name of the loop variable*

PRINT_NODE: This procedure prints a report line for a node. The NODE# and NAME fields are respectively the node's number and name.

DESCRIPTION assumes one of the following values:

RECORD IN FILE file

ASSERTION

MODULE NAME

FILE

SPECIAL NAME

GROUP [IN RECORD record] [IN FILE file]

FIELD [IN RECORD record] [IN FILE file] [TARGET OF ASSERTION: assertion]

DECLARED SUBSCRIPT

FREE SUBSCRIPT

SYSTEM SUBSCRIPT

STORAGE DEVICE OF TYPE: type

Here "file" stands for a file name.

"record" stands for a record name.

"assertion" stands for an assertion name.

"type" stands for a device's type.

The EVENT field is usually blank except for the following cases:

For an input record: READ RECORD

For an output record: WRITE RECORD

For a module name: PROCEDURE HEADING

For an input file: OPEN FILE

For an output file: CLOSE FILE

5. Code Generation

This phase of the Processor proceeds after array graph construction, specification analysis, program design, and schedule creation have been completed. Recall that had there been user errors during syntax analysis or specification analysis, then neither the flowchart creation nor the code-generation phases would be reached. As seen in Figure 13 the code generation phase accepts as input the schedule tables produced in the previous phase, and produces as output a complete PL/1 program ready for compilation.

5.1 Generation of PL/1 Program

The control program for generating the complete PL/1 program (GENPL1), as shown in Figure 13, accepts the tables of attributes and the schedule table created during the previous phase as input. This phase produces, as output, the complete PL/1 program and a code-generation report. The files to which code is written are described below.

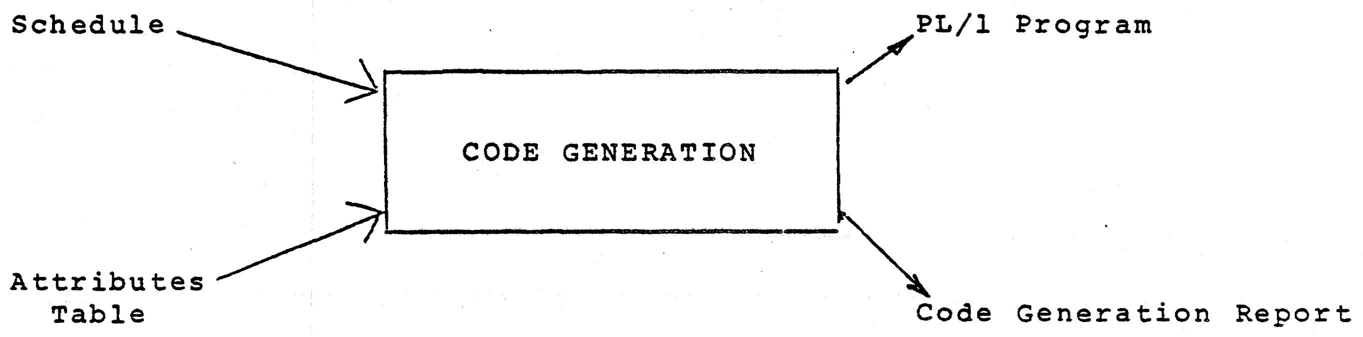


Figure 13

Overview of the Code Generation Phase

Generating the PL/1 program code, as can be seen in Figure 14, is accomplished by processing the schedule list described above and invoking the appropriate code-generation sub-routine. Algorithm CODEGEN describes the generation of PL/1 code. The executable PL/1 code is generated by examining the elements of the schedule one at a time, and invoking the code-generation routine that corresponds to the type of operation. These include code-generation routines for input-output operations, for invoking and writing of object assertions and for generating control structures.

The executable PL/1 code is written out to the "PL1EX" file, while associated PL/1 "ON" conditions are written to the "PL1ON" file. The PL/1 procedures (which contain assertions plus functions) are written to the PL1PROC file. The PL/1 code for declaring the object data items is written to a PL1DCL file.

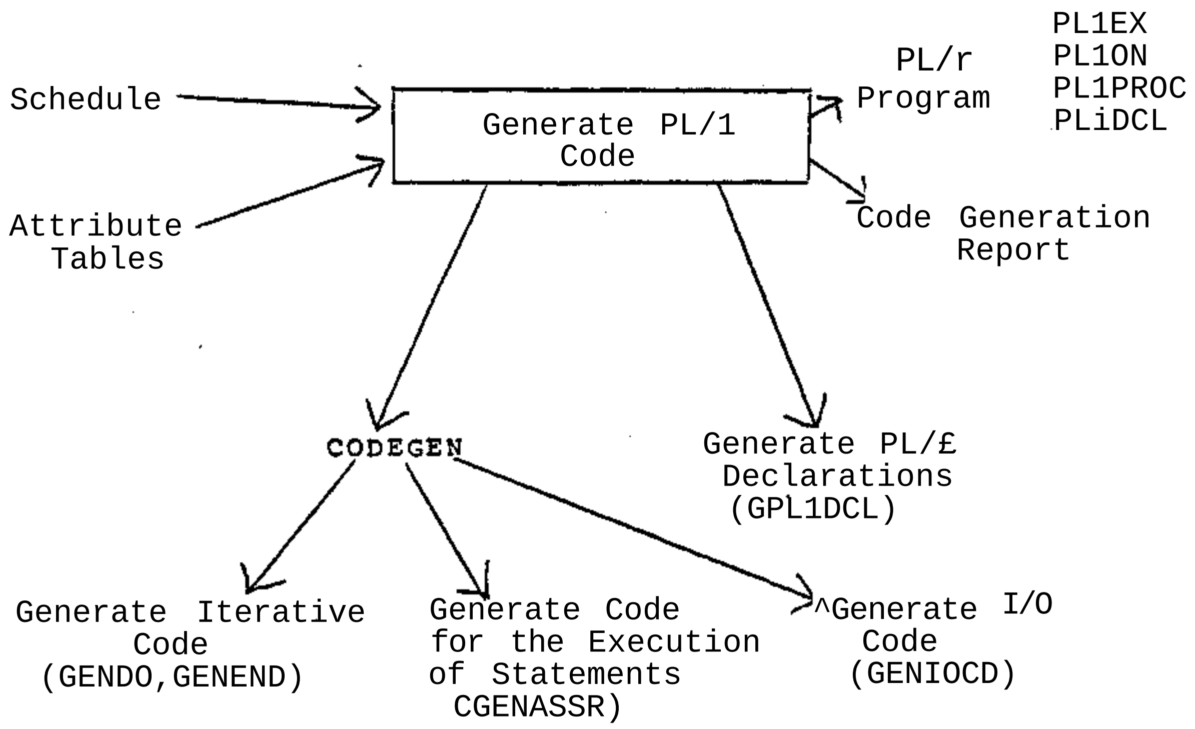

Schedule
Attribute Tables
Generate PL/1 Code
PL/r Program
PL1EX
PL1ON
PL1PROC
PLiDCL
Code Generation Report
CODEGEN
Generate PL/£ Declarations (GPL1DCL)
Generate Iterative Code (GENDO,GENEND)
Generate Code for the Execution of Statements CGENASSR)
Generate I/O Code (GENIOCD)

Figure {4

Components of Generating PL/I Code

5.1.1 Generate PL/t Program (CODEGSN)

This procedure generates the code for the PL/£ program. It takes care of all parts of the generation except for the declarations of variables and files which is done by GPL10.CL.

Initially we open all the output files and then generate the following standard instructions which open every generated pro.gram. These instructions are routed to the file PL1EX:

```
ALLOCATE ERROR, ACC_ERROR
ACC_ERRROR »''O'B
ALLOCATE $ERR_LAB
$ERR_LAB « END_PROGRAM.
```

The following declarations are routed to PL1DCL :

```
DCL (ERROR, ACC_ERR, NOT_DONE) CTL BIT(l)
DCL $ERR_LAB LABEL CTL
```

The following instructions are sent to PL1ON:

```
ON ERROR
BEGIN
     IF ERRORF_BIT THEN WRITE FILE (ERRORF)
           FROM ($ERRORJ3UP)
     ERROR - 'l'B
    . GO TO $ERR_LAB
END
ERROR_RESTART:
```

The procedure GENERATE (FLOWCRT, 0) is called then to perform the actual generation.

5.1.2 A Scan of the Schedule and Generation; GENERATE(LIST.LEVEL)

This recursive procedure scans the schedule given by the list of elements LIST at level LEVEL, It calls lower level procedures to process the different types of elements.

1. Scan each element of the list LIST.
2. If the element is a node-element call GEN-NODE. Return to 1 and repeat until the list is all empty.
3. If the element is a for-element do the following;
   - 3.1 Call GENDO to produce a code for opening a loop.
   - 3.2 Call GENERATE recursively with the list of the elements within the loop's scope and level * LEVEL-Hl.
   - 3.3 Call GENEND to generate the termination of the loop.

5.1.3 Open a Loop (GENDO)

This procedure produces the code necessary in order to open a loop. The loop variable name FOiySAME and the termination criterion are taken from the fields FORENAME and FOR_RANGE in the for-element being scanned.

The following instructions always precede each loop opening:

```
ALLOCATE ERROR, ACCJBRROR
ACCJERROR « 'O'B
ALLOCATE $ERR_LAB
$ERR_LAB « LOOfJENDc
```

The "c" following LOOP_END is a unique number assigned to the loop. The purpose of these statements is to ensure that an error occurring within the loop control will be directed to

END_LOOPc which is a label immediately preceding the loop's end.

We then construct the do-statement itself. If the termination criterion given is that of a fixed upper limit or given through a SIZE variable, the string DO_ST is initialized to

```
DO name = 1 TO upper
```

where "upper" is either a constant number or a variable of the form SIZE$X.

Thus the two basic forms for the loop opening are according to the termination criteria:

a) If an upper limit is specified then:

```
        DO name = 1 TO upper [WHILE (condition)]
```

b) An upper limit is not specified:

```
           name = 0
        DO WHILE (condition)
           name = name+1
```

Here "name" is the loop variable. "Condition" is the termination condition in case b and may contain additional conditions in both cases.

If the range is specified by an END.X descriptor, we add NOT_DONE to the condition and the following statements before the loop's beginning:

```
ALLOCATE NOT_DONE
NOT_DONE = '1'B
```

NOT_DONE will be set to '0'B whenever the appropriate END.X variable is set to 'true'.

If there is an end-of-file condition associated with the iteration, either as the main termination condition, or because this is an iteration on an input record or group above the record level which are last in their peer group, we add:

¬ ENDFILE$f to the condition "condition".

5.1.4 Close a Loop (GENEND)

This procedure produces the code needed to close a loop. First we check all the nodes that have been accumulated in the list PREDLIST. This list, local to each invocation of GENERATE accumulates all the variables which are defined in the loop and whose dimension corresponding the loop variable is virtual. The actual range declared for such dimension is 2 and in each iteration we compute (or read) A(...,2,..) and may refer to the previous element as A(...,1,..). When the loop is done we should perform the transfer: A(...,1,..) = A(..,2,..). Elements are put on the list PREDLIST by the procedure CHECK_VIRT which is called whenever processing a node which is a field or a group.

After producing a sequence of these shifting operations we produce the label:

```
          LOOP_ENDc:
```

where "c" is the unique count associated with the current loop. If the termination criterion for the loop was through an END.X descriptor we also produce the code:

```
          IF END.X = SELECTED THEN NOT_DONE = 'O'B
```

This has to be done at the end of the loop since the determination of END.X at a given iteration determines whether this iteration will be the last.

After this we produce the following statements:

```
$TMP_ERROR = ACC_ERROR
FREE ERROR, ACC_ERROR
FREE $ERR_LAB
IF $TMP_ERROR THEN ERROR, ACC_ERROR = '1'B
```

If the termination criterion was through an END.X descriptor we also produce:

```
FREE NOT_DONE
```

5.2 Code Generation for a node (GEN_NODE)

This procedure generates the code associated with a single node. It branches according to the type of the node to different parts dealing with the different types of the nodes. In the following "node" always refer to the node's name.

5.2.1 Program Heading

If the node is the module name (type MODL) we produce the code:

```
name:  PROCEDURE OPTIONS (MAIN)
```

This code is routed to the file PL1DCL.

5.2.2 Files

If the node is a file node (type FILE) we generate first three names. "File stem" is the file name, removing the "NEW" or "OLD" prefixes if there are any. "Name" is the original name and "file suff" is the file-stem with the addition of S for source only files, T for target only and U for update files (both source and target).

1. If the file is an input file we produce the statement:

```
OPEN FILE (file suff)
```

2. If the file is a sequential input file and an end-of-file is explicitly mentioned by the user or needed to terminate iterations we produce the declaration:

```
DCL ENDFILE$file stem BIT(1)INIT('0'B)
```

   routed to PL1DCL. The statement:

```
ON ENDFILE (file suff) ENDFILE$file stem = '1'B
```

   is sent to PL1ON.

3. If the file is input sequential and unkeyed we send the declarations:

```
DCL name_S CHAR (length) VARYING INIT('')
DCL name_INDX FIXED BIN
```

   "Length" is the maximum length of records in the file. "Name_S" is the name of a buffer into which records in the file are read. "Name_INDX" is a variable used to scan the buffer for unpacking input fields.

4. If the file is an output file we produce the statement:

```
CLOSE FILE (file suff).
```

5.2.3 Records

If the node is a record (type RECD) we call GENIOCD to produce the code for the reading and writing of records.

We also call CHECK_VIRT to check if the record has a virtual dimension.

5.2.4 Groups and Fields

To process groups and fields we call the procedure GENITEM. We also call CHECK_VIRT to find if the node has a virtual dimension.

5.2.5 Special Descriptors

We check if the node has a virtual dimension. Then if the node is of the form SUBSET.X we produce the code

```
If SUBSET$X THEN GO TO END_LOOPc
```

"c" is the unique count associated with the current loop. This will cause transfer of control to the end of the current loop if SUBSET$X has just been set.

5.2.6 Assertions

If the node is an assertion we call the procedure GENASSR to produce the code for an assertion.

If the type of the node is not one of the recognized types the following error message is generated:

```
CODEGEN:  AN ILLEGAL TYPE - TYPE: type FOR NODE: name
```

5,3 Auxiliary Procedures Within CODEGEN

Following is a description of some auxiliary procedures within CODEGEN/

5.3.1 Checking a Virtual Dimension (CHECK_VIRT)

This procedure checks a given node for being a repeating node with virtual dimension* If it is a virtually dimensional variable, the physical range assigned to it is either 1 or 2 depending on whether there is an explicit reference to A(...I-1) if A is the dimensional node,

If the node is virtual and a previous value is explicitly required, the node is added to the list PREDLIST.

Consequently, before the loop's end the following statement will be produced:

A<...1) - A<..*2)

for each A which was placed on PREDLIST.

5.3.2 Constructing an Instance of a Subscripted Variable (CRSVAR)

This procedure creates the text of a subscripted variable for the use of other code generation procedures. This procedure is not used in assertions. The list of local sub-scripts of the node is scanned. For each physical subscript we retrieve the IDWITH field which gives the level of the loop variable identified with this subscript. The array LOOP_VARS(1) contains for level 1 the name of the loop variable on level 1, placed there by GEN_DO. Therefore for each physical dimension number I we take LOOP_VARS ((LEVEL+1)-(OFFSET+D) as the subscript constructed. OFFSET is an additional parameter enabling a shift. For virtual dimensions we usually take the "current" value which is 1 or 2 according to whether there is an explicit requirement for the previous value of the same variable. A parameter named CASE specifies whether the last subscript is required to be the previous (case=1), the current (case=2) or the next (case=3). Therefore if the rightmost subscript is physical we use sname-1, sname or sname+1 as the subscript where "sname" is the subscript according to the LOOP_VARS stack. If the rightmost subscript is virtual we use 1, 2, or 3 according to the parameter CASE.

5.3.3 ADD_TO_WHILE (COND)

This procedure adds a condition to the string by the name of WHILE_COND which accumulates all the conditions to be included in a DO statement. If the string is empty it is set to COND. If it contains previous conditions we add to it the string 'L'|| COND.

5.3.4 $

This procedure converts a qualified name which contains reserved prefixes. If RES is any reserved prefix then the name RES.X should be converted to RES$X. To be more precise:

Names of the form NEXT.X are converted to X.

Names of the form NEW.X and OLD.X are converted to
NEW_X and OLD_X respectively.

Names of the form RES.X, where RES is any of the prefixes

SIZE, END, ENDFILE, SUBSET, POINTER, FOUND, LEN are converted into RES\$$\tilde{X}$. Where $\tilde{X}$ is X in which each appearance of a dot: '.' has been replaced by an underline: '_'.

5.4 Packing and Unpacking of Input/Output Fields (GENITEM)

This procedure is called for nodes which are input/output fields and may require packing and unpacking of information into or from an input/output buffer. The code for reading or writing the buffer is generated in association with the record node. After defining the names of the buffer and of the packing counter, and checking whether packing or unpacking is actually required the procedure calls an auxiliary procedure: FIELDPK which generates the code itself. For output fields a special check is made to determine whether the current field is the leftmost in the record in which case a code for initializing the packing counter is produced.

1. If the field is not a member in a v[illegible]able structure record, i.e. containing any fi[illegible] or group whose range is determined by a SIZE or [illegible]descriptors, or a field whose length is determined by a LEN descriptor we exit immediately. Packing and unpacking is done only for fields in variable structure records.
2. Determine the name of the record containing the current field. Let it be REC. Then we construct a buffer name: REC_S and a buffer index name REC_INDX. Let the field's name be in the variable "field".

3. Determine whether this field is the leftmost field in an output record. If the field is leftmost and not repeating or contained in any repeating groups we issue the code:

   ```
   REC_INDX=1.
   ```

   initializing the packing index. Note that in the input case the index is initialized immediately after the reading of the record.

   If the field is leftmost and output but contained in several loops with the loop variables $I_1, .. I_m$ respectively, we generate the code:

   IF I_1=1 & ..I_m=1 THEN REC_INDX=1.

4. In all cases call FIELDPK with an appropriate parameter: 1 - for packing, 0 for unpacking, for the actual code generation.

5.4.1 FIELDPK

This procedure produces the actual code for the packing or unpacking operation. Available to it are the field's name, buffer name and index name as well as the field's type.

1. If the length type of the field is fixed, i.e. specified in the declaration we compute its length directly. If the field's type is 'C', 'N' or 'P' denoting respectively character, numeric or picture we take the declared length. Otherwise we call BYTE_CALC with the declared length and type to compute the

length of the field in bytes. The string representing the length is stored in "lenstring".

2. If the length of the field was not declared exactly (but only by specifying lower and upper bounds) we check that there exists a length descriptor for this field. If none exists we issue the error message:

   FIELDPK: NO LENGTH SPECIFICATION FOR THE FIELD-field.

3. If a length descriptor is found we set:

   lenstring = BYTE_CALC (length-descriptor, field-type).

   This will cause an execution time call to BYTE_CALC in order to compute the byte-length of the field during run time.

4. If the field is an input field we generate the instruction:

   UNSPEC(field) = SUBSTR(REC_S, REC_INDX, lenstring)

   Otherwise we generate:

   SUBSTR(REC_S, REC_INDX, lenstring) = UNSPEC(field).

   Here:

   "field" - is the field name properly subscripted (through call to CRSVAR)

   "lenstring" - the length specification

   If the field is of type *C* the UNSPEC qualities will be omitted.

5. Generate the following code for incrementation of the buffer index:

   REC_INDX = REC_INDX + lenstring.

5.5 Generating Code for Assertions (GENASSR)

This procedure generates the code for assertions. If the assertion contains a special function such as SUN!, AMIN or AMAX we modify the assertion to perform the needed computation as well as provide initializing statements for the variable holding the cumulative result• In addition this procedure will also transform assertions containing conditional expressions into conditional assertions. Thus, an assertion of the form:

```
          Y « IP (IF X > 0 THEN Y > 0 ELSE Y <»0)
                  THEN X*Y ELSE -JX*?
          will be transformed into:
          IF X > 0 THEN IF Y > O THEN Y « X*Y
                                      ELSE Y = -X*Y
          ELSE IP Y <«O THEN Y * X*Y
                        ELSE Y » -X*Y.
```

Apart from these two special transformations the main task of GENASSR is to transform the syntax tree representation of the assertion into a string representation acceptable by the PL/I compiler. The transformation is carried out by a recursive climb on the syntax tree, combining for each node the string representations of the descendant subtrees into a string representation of the tree rooted at that node.

The overall execution of GENASSR can therefore be summarily described as:

1. Treat the case of (special) array functions.

2, Transform assertions with conditional expressions into conditional assertions.

3. Form the string representation of the assertion.

5,5.1 Transforming Array Functions

This subtask is performed in the body of GENASSR. Array functions are functions which may operate on array elements as they are generated and do not need the complete array at one time. Array functions may be divided into reduction functions which produce a single result for an array, and running-value functions which for each array element A[l] produce a value which depends only on the array segment A[l],.$_o$A[X]o

Examples of reduction functions are SUN, AMIN, AMAX which compute for a given array its sum, minimal and maximal values respectively. Their running-value counterparts are respectively RUN_SUM, RUN_MIN and RUN^MAX which compute the similar function on each array segment. The format for the use of any of these functions is:

$$Y - FCN\ (E, J_{1\#}.\bullet J_k)$$

where E is the array element, generally dependent on J_1,..J_V, and Jw'-Jfc $^{t\wedge ie}$ running subscripts. For a running value function the left hand side should be of the form $Z(J_{if}..J_k)$ or some permutation thereof.

As displayed above, the use of array functions is currently restricted to the top level of simple assertions. E itself may be a conditional expression.

The statement above is transformed as follows:

1. We add the statement

   IF $(J_1=1$ [&..(J_k-1)] THEN Y = initial

   as a preceding statement. Its role is to initialize Y to an initial value.

   'Initial' is a value computed by the procedure INIT_VAL and depends on the function FCN and on the type of Y. It will be 0 for summation functions,

   the minimal value possible for the type of Y for a maximization function, and

   the maximal value possible for the type of Y for a minimization function.

2. If FCN = 'SUM' or 'RUN_SUM' we modify the assertion into Y = Y+E.

   If the function is a minimization or maximization function, the statement is modified into:

   Y=MIN(Y,E) or Y=MAX(Y,E) respectively

   The modification is done by changing the pointers and fields in the syntax tree, and creating new nodes if necessary.

5.5.2 Transforming Conditional Expressions into Conditional Assertions

This task is carried out by the procedure SCAN which uses the auxiliary procedure EXTRACT_COND.

5.5.2.1. EXTRACT_COND (ROOT,COND,LEFT,RIGHT)

This procedure identifies and extracts the leftmost conditional expression in a given expression pointed to by ROOT.

If a conditional is found the (pointer to the) condition is returned in COND and its first (THEN) and second (ELSE) subexpressions returned in LEFT and RIGHT respectively. If the analyzed expression contains no conditional expression the procedure returns NULL.

Its operation can be described as follows:

1.1 Inspect the top level of the given expression.

1.2 If it is a conditional expression, return respectively its condition, THEN subexpression and ELSE subexpressions , exit.

1.3 If the expression is a simple expression, i.e. a constant or a variable, return NULL and exit.

1.4 If the expression is a compound expression, scan each of its descendants by calling EXTRACT_COND recursively. Consider the first COND, LEFT and RIGHT which are returned such that COND $\neq$ NULL. In general, a compound expression is of the form:

$$E = g(E_1,..E_m)$$

Assume that the recursive scanning of $E_1,..E_m$ produces first COND $\neq$ NULL for E_i $1 \leq i \leq m$, returning also the THEN and ELSE subexpressions L, R respectively. Then the current call for E returns:

COND as the condition,

$g(E_1,..E_{i-1},L,..E_m)$ as LEFT, and

$g(E_1,..E_{i-1},R,..E_m)$ as RIGHT.

Thus the overall effect of EXTRACT_COND on an expression E is to extract a condition C if one exists in E (returned as COND), and then to compute E_1, which is E when C is true, and E_2,which is E when C is false. E_1 and E_2 are returned in LEFT and RIGHT respectively. Described in another way we look for C, E_1 and E_2 such that the following equivalence holds:

$$E = \text{IF } C \text{ THEN } E_1, \text{ ELSE } E_2.$$

In particular this gives:

$$g(E_1,..E_{i-1}, (\text{IF } C \text{ THEN } L \text{ ELSE } R),..Em) = \\ \text{IF } C \text{ THEN } g(E_1,..E_{i-1},L,..E_m) \\ \text{ELSE } g(E_1,..E_{i-1},R,..E_m).$$

5.5.2.2. SCAN(IN)

The procedure SCAN effects the complete transformation of assertions containing conditional expressions into conditional assertions. The procedure is presented with an assertion pointed to by IN, and return a pointer to the transformed assertion.

1 If the assertion is a simple assertion, go to step 5.

2 Assertion is a conditional assertion of the forms IN: IF COND THEN S_1 ELSE S_2 where S_x, S_2 are statements.

3 Call EXTRACTjCOND to check whether COND contains a conditional expression. If it does then EXTRACTJCOND returns C,L,R such that:

```
COND * IF C THEN L ELSE R.
```

We transform IN into:

```
IN:  IF C THEN IP L THEN Sx
                     ELSE S2
          ELSE IF R THEN Sx
                     ELSE S2
```

The value returned is SCAN(IN^1) which involved a recursive call to SCA13.

4 Assume that the statement IN is as above but COND contains no embedded conditions. In this case we return the statement:

```
IF COND THEN SCAN(Sx) ELSE SCAN(S2)
```

obtained by two recursive calls to SCAN for the assertions S_1 and S_2. Exit.

5 The assertion is a simple assertion:

```
y • E.
```

Call EXTRACTJCOND(E). If it returns NULL, we return the assertion Y « E unchanged. Otherwise

EXTRACT_COND returns C,L,R such that
E = IF C THEN L ELSE R.
We return the result of

```
SCAN (IF C THEN Y = L
                ELSE Y = R)
```

5.5.3 Transforming the Assertion into String Form (PRINT)

This procedure is presented with a pointer to an assertion represented by a syntax tree and converts it into string representation.

The procedure branches according to the type of node to be printed:

1. If the node is a subscripted variable $A(E_1,..E_m)$ we generate the string 'A('. We then scan each of the subscript expressions E_1 to E_m and add them to the string according to the following subcases:

   1.1 If the subscript at position i corresponds to a record level and the variable was prefixed by 'NEXT.' then

   1.1.1 If the position is virtual we insert the subscript value '3'.

   1.1.2 If the position is physical and the expression Ei is a constant c, we insert the value of c+1.

   1.1.3 If the position is physical and E_i is an expression we insert PRINT(E_i) $||$ ' + 1'.

1.2 If the subscript position i is a virtual position then:

   1.2.1 If it is a simple subscript and the position is associated with an input record to which the prefix NEXT is applied somewhere, but definitely not in the current variable, then we insert the subscript '2'.

   1.2.2 If the position is not 'NEXT' qualified and the subscript is I then no subscript is inserted.

   1.2.3 If the subscript expression is I-1, then '1' is inserted. It is assumed that in these cases the physical allocation for the virtual position will be at least 2:'1' standing for the previous value, and '2' for the current value.

2. For all other compound nodes we call PRINT recursively to convert the descendants and insert between them the string representation of the separators, operators and delimiters stored in the OP_CODE fields of the code. This string representation is available in the string array KEYS.

3. For atomic nodes we use either the variable name, directly or through its node number. Loop variables (subscripts) are accessed through the level indication

available in their IDWITH field which is used as an index to the array LOOP_VARS. Function names are retrieved by their function number indexing the table FCNAMES.

5.6 Generating Input/Output Code (GENIOCD)

The routine for generating input/output code (GENIOCD) is invoked by the generate PL/1 code control routine after reading an element that corresponds to a record node. It accepts as input the node number corresponding to the element. This routine generates the Pl/1 READ, WRITE, or REWRITE Statements with the appropriate parameters based on the flowchart table entry, as well as any control code or condition code associated with the input/output operation.

To summarize the different statements generated by GENIOCD for the different cases we use a table format (Table 11) for DO_REC instead of an algorithm form for the sake of clarity. Each of the different cases is preceded by the conditions defining the case followed by the statements which are generated for the case. The upper case letters represent part of the actual Pl/1 string being generated, whereas the lower case letters are the metanames of the items obtained from the flowchart table during program generation.

Several preparatory steps are taken before branching to the different cases.

1. Definition of Names. Derived from the record name we generate several variable names to be used in the code.

   Let the record name be designated by R,

   1.1 If R is of the form OLD.X or NEW.X we define RECNAME as OLD_X or NEW^X respectively.

   1.2 Otherwise we define RECNAME as R.

   1.3 RECBUF is defined as recnamej,

   1.4 RECINDX is defined as recname^INDX.

   Consider now the file which is parent to R. Let it be denoted by F.

   1.5 Set FILENAME to F.

   1.6 If F is of the form OLD,X or NEW.X set FILENAME to OLD_X or NEW_X respectively and FILESUFF to filename U.

   1.7 Otherwise set FILESUFF to filename S if the file is a source and to filename T if the file is a target*

   1.8 Set "EOF to ENDFILE$filename.

   1.9 Retrieve the keyname associated with the record, if one exists, and assign it to KEYNAME.

   1*10 Set FOUND to FOUND$filename.

2. Issue the declaration. DCL recbuf CHAR (len^dat(n)) VARYING INIT(^{lf}). This declares a buffer for the record into which and out of which the information

will be read or written. 'Len_dat(N)' here gives the buffer length.

3. If the record has a variable structure/ packing and unpacking will be called for and we therefore issue the declaration:

```
DCL recindx FIXED BIN
```

4. If the file is an output file of fixed structure issue the transfer:

```
recbuf = recarea
```

'recbuf' is the record buffer defined above while 'recarea' is the name of the internal structure allocated to the record. It might be subscripted. If the file is an output file of variable structure the movement of data from the record area into the record buffer is done piecewise and instruction for its execution generated in conjunction with each of the output fields belonging to the record.

5. If the record is an output record and a SUBSET condition was specified for it we enclose the code for writing the record by the condition:

```
IF SUBSET$record THEN DO;
     code
END
```

5.6.1 DO_REC

The procedure DO_REC produces the code for the reading and writing of records. It branches according to the cases in Table 11.

Algorithm BYTE^CALC (Length, TYPE, #BYTES)

A routine for calculating the length in bytes of a data of a given length and a given type.

Length - The length of the field in units appropriate to the type.

TYPE - The field type (one of C,N,P,T,F,B,L ,E)

#BYTES - The calculated length in bytes

```
1.  If TYPE - •C^I|^IN^I|^IP^I then #BYTES - Length

2*  If TYPE + 'T^1 (Bits) then #BYTES - [length/8]

3.  If TYPE •F^1 (Decimal) then #BYTES * [(length+1)/23

4.  If TYPE w 'B^1 (Binary) then
       if length <16 then #BYTES » 2
                otherwise #BYTES = 4

5o  If TYPE » 'L^1 (Binary Floating) then
      if length <22 then #BYTES * 4
      if 22 length < 54 then #BYTES • 8
           otherwise #BYTSS • 16

6.  If TYPE » ••£' (Decimal Floating) then
      if length <7 then #BYTES * 4
      if 7 ^length <17 then #BYTES » 8
            otherwise #BYTES « 16
```

Case 1: An Input Sequential and Nonkeyed Record.

The following code is produced:

```
"IF recbuf «• 'f THEN DC-
      READ FILE (filesuff) INTO (recbuff) ;
END;
ELSE recbuf - filebuf11
```

If the record is of fixed structure we issue:

```
      "recarea » recbuff"
```

to move the information from the record buffs into the internal record structure.

Otherwise, for variable structure records we only produce

```
      "recindx » 1"
```

to reset the unpacking index. The movement of the data to the individual fields will be done in conjunction with the nodes corresponding to the fields (see GENITEM).

Next we read and unpack the data for the NEXT record.

```
"IF -l£NDFILE$FILE THEN DO;


READ FILE (filesuff) into (filebuf);"
```

if there is a reference to NEXT fields we should move or unpack the data from the next record. If the record has a variable structure we call the procedure UNPACK to

Table 11

DO_REC Inpur Output Transformations From Flowchart Table to Pl/1 produce the moving code, and then reset

```
            "recindx = 1"
```

Otherwise, for fixed structure record we produce

```
            "next_record_area = filebuf"
```

where 'next_record_area' is the internal structure component reserved for representing the next record fields.

Case 2: Input, Sequential and Keyed Record.

Ensure that the following declarations have been issued:

```
            DCL FOUND$rec BIT(1)
            DCL PASSED$rec BIT(1)
```

Issue now the code:

```
            FOUND$rec, PASSED$rec = 'O'B
DO WHILE ¬ENDFILE$file & ¬ PASSED$rec;
            READ FILE (filesuff) INTO (recbuf);
             If the record is of variable structure issue
                    "recindx = 1"
            and call UNPACK to unpack its fields,
            otherwise issue:
                    "recarea - recbuf" .
            If keyname = POINTER$rec THEN
                    FOUND$rec, PASSED$rec = '1'B;
            ELSE IF keyname > POINTER$rec THEN
                    PASSED$rec = '1'B
```

Table 11 (continued)

DO_REC Input Output Transformations From Flowchart Table to PL/1

Case 3: Input, Nonsequential (Indexed or Random), Keyed Record.

```
     Verify that the declaration
           "DCL FOUND$rec BIT(1)"
     has been issued.  Then issue the code:
           FOUND$rec = '1'B;
     READ FILE (filesuff) INTO (recbuf) KEY (POINTER$rec)
     ON KEY (filesuff) FOUND$rec = 'O'B
     If the record has fixed structure issue
           "recarea = recbuf"
     otherwise issue
           "recindx = 1".
```

Case 4: Output, Sequential Record

```
     WRITE FILE (filesuff) FROM (recbuf)
```

Case 5: Output, Nonsequential, Keyed and an Update Record (both NEW and OLD specified)

```
     REWRITE FILE (filesuff) FROM (recbuf) KEY (POINTER$rec)
```

Case 6: Output, Nonsequential and Keyed Record.

```
     WRITE FILE (filesuff) FROM (recbuf) KEY  (POINTER$rec)
```

Table 11 (continued)

DO_REC Input Output Transformations From Flowchart Table to PL/1

5.6.2 Unpacking Variable Structure Records (UNPACK)

If a record is of fixed structure its data can be moved between the record buffer and the internal structure area by a single PL/I assignment such as:

"recarea = recbuf"

If however the record is of variable structure the data movement will be performed by individual transfers, one for each field. The transfer statements will be interleaved with other statements which compute the variable parameters of the record structure such as fields' lengths and dimensions. These parameters can depend on earlier fields in the same record. The transfer instructions in the variable structure case are generated in conjunction with the schedule elements associated with the field nodes. There are however two cases in which the unpacking of information has to be done immediately after the reading of the record. The first case is that of reading the NEXT version of a record in order to access fields referred to by a NEXT.F reference. The other case is that of a sequential search on a file for a given key value. Here, also, we must, unpack the record in order to access the key field immediately after reading. In both cases we make the following simplifying assumption: The fields referred to by a NEXT.F reference, or used as a key value must all be in a prefix of fixed structure of the record.

Consequently in the abbreviated version of the unpacking process described here/ we may unpack only up to and excluding the first field or group which depend on a variable parameter* By assumption this must have included all the needed fields of the record. Of course,.a full unpacking will take place prior to the actual processing of the fields, duplicating some of the abbreviated unpacking performed here.

The procedure UNPACK accepts two parameters: NODES and CASE. NODE* identifies the field into, which data should be moved. CASE=2 implies unpacking from RECBUF to the "current" record area, and will be used for a sequential search. CASE=3 implies unpacking from PILEBUF to the "next" record area, and is used for reading the NEXT fields referenced.

The main part of UNPACK can be described as follows:

1. If the node is a repeating group or field we check for the termination criterion of the repetition. If it is not a constant repetition we exit.

   1.1 Otherwise open a loop: Define a loop variable of the form UNP#n, and generate the declaration

       DCL UNP#n FIXED .BIN

   1.2 Then issue the code

       DO UNP#n«l TO maxrep (nodet!

   1.3 Call the subprocedifres DO^GRP or *d*y_FLD to issue code for the unpacking of the node or its descendants.

1.4 Issue an 'END' code terminating the loop.

2. Otherwise if the node is not repeating then:

   If the node is a group or a record call DO_GRP, otherwise call DO_FLD

5.6.3 Unpacking Groups (DO_GRP) and Fields (DO_FLD)

Two subprocedures complete the unpacking.

DO_GRP: This procedure considers in turn each descendant of the node NODE#. For each descendant D it calls UNPACK (D,CASE) recursively.

DO_FLD: This procedure is responsible for producing code for the unpacking of a field. It uses the procedure FIELDPK to expand the code itself. For description of FIELDPK see 5.4.1. The procedure distinguishes between two cases according to the value of the parameter CASE.

For CASE=3 the code produced is (assuming a fieldname F and a record name REC).

```
F(..3,..) = SUBSTR (filebuf, FILEINDX, Length).
```

The '3' designates assignment to the 'next' version of the record.

If CASE=2 the following code is produced:

```
F(..2,..) = SUBSTR (recguf, RECINDX, Length).
```

The '2' designates assignment to the 'current' -sion of the record. If the record has no 'next' reference it might be allocated only a single copy (version) and then the '2' subscript will be dropped. This is managed by FIELDPK.

5.7 Generating the Program Error File

If there is any error during the execution of the generated program, an input record, for which the error occurred is written to an error file, ERRORF. This error file must be described in the JCL used to execute the generated program, by including a dd statement of the form.

//ERRORF DD DSN = <dsname> is the error file name and the <dd_parameter > are the same parameters used in the dd statement of the source file.

The required code for writing the bad input record to the error file is generated by the routine GENIOCD. For example, in the DEPSALE example, the following PL/I code is generated:

```
ON ERROR BEGIN;
  IF ERRORF_BIT THEN
     WRITE FILE(ERRORF) FROM $ERROR_BNF
  GO TO #ERR_LAB
  END
```

Note that if no dd statement for the error file is specified, then ERRORF_BIT is '0', in which case, the record that caused the error is completely ignored.

If the I/O mode is WR we check whether there exists a dictionary entry of the form SUBSET.recname. If there exists such, we precede the I/O code (which is enclosed by a DO-END pair) by the statement:

"IF SUBSET$recname THEN".

After the I/O code we check again to see whether a variable of the form SUBSETSrecname is defined. If there is such a variable we produce the codes

"SUBSETSrecname «.^{l}'l^{t}B;""

The main sequential input file (there must be exactly one) is read in a special way as shown in case 1 of Table 9* It is always read one record ahead so that filename^ (filename) always contains the <u>next</u> record to be used. If the NEXT option is used, i.e. there is somewhere a reference to an item of the form

NEXT.A

where A is an item in the main input file, several special actions take place.

The area for the record is defined as having dimension 3. Subscript 2 will always refer to the 'current' version of all fields in the record, while subscript '3' corresponds to the 'next^{1} copy.

<ins>5.8 Generating PL/1 Declarations (GPL1DCL)</ins>

This procedure generates most of the declarations for the structures defined by the user as well as those added by the system. Some additional declarations are generated by the other procedures during the code generation.

The main part of GPL1DCL can be described as follows:

1. For each file F in the specification (available from the list FILIST) call

   ```
   DECLARE_STRUCTURE(F)
   ```

   to declare F and all its descendants.
2. For each node N in the specification which is a group or a special reserved prefix name and has no parent, call

   ```
   DECLARE_STRUCTURE(N)
   ```
3. For each system or standard subscript which has been referenced or used (tested through the array USED) we issue the declaration:

   ```
   DCL subname FIXED BIN
   ```

5.8.1 Declaring a Structure (DECLARE_STRUCTURE)

This procedure declares a complete structure. It issues the declarative: DECLARE, and then proceed to call DCL_STR (N,1,0).

5.8.1.1 DCL STR (N,LEVEL,SUX)

This recursive procedure produces a declaring clause for each node N in the structure. 'LEVEL' is the current level in the structure while SUX is a termination criterion stating whether there is a next item on the same level (younger brother) or a descendant.

1. Some preliminary transformations are made on the declared item name.
   - 1.1 File names of the form NEW.F and OLD.F are modified to NEW_F and OLD_F respectively.
   - 1.2 All other names excluding special names are reduced to their last component.
2. For special names the resulting declaration is:

   For SIZE, LEN and POINTER names:

   name FIXED BIN,

   while for all other reserved prefix names it is

   name BIT(1),
3. The declaration includes in general the following items:

   LEVEL - The component level.

   Name - The declared name.

Repetition - The repetition indicator.

Type - The data type.

The type is determined as follows:

For Character fields	- CHAR(len)[VARYING]
For Binary fields	- BIN FIXED(len, scale)
For Numeric fields	- PIC '99..9'
For Fixed fields	- DEC FIXED (len, scale)
For Binary Floating	- BIN FLOAT (len)
For Bit string fields	- BIT(len)[VARYING]
For Decimal floating fields	- DEC FLOAT(len)
For Picture fields	- PIC 'picture'

In the above 'len' is the specified or default length for the field. The VARYING option is taken if the length is specified (for strings) by a minimal length < maximal length.

The repetition is defined by the repetition counter. If different from zero and denoted by R we append the string '(R)' after the declared name.

If the repetition is virtual we set the repetition to 3 if this is a record level for a record containing a 'next' referenced field. It is set to 2 if there is a reference to X(..I-1,.-) with I-1 in the position corresponding to the current level. In all other cases of virtual repetitions we omit the repetition indicator completely.

4. For each of the descendants of the node m, call DCL_STR(M, LEVEL+1, termination) recursively.

5.9 Other Code-Generation Supporting Routines

Certain routines have been found to be useful to all the code generation routines.

The "WRite PL/1" routine (WRPL1) is called by each of the code generating routines in order to write out the PL/1 code. Two parameters are passed to this routine: the string of Pl/1 code to be written and the output file to which it should be written. WRPL1 takes the string containing one or more generated PL/1 statements and it outputs the PL/1 statement in the format and syntax that the PL/1 compiler expects. It ensures that the statement fits in columns 2 to 72 of each card necessary for the statement produced and generates sequence numbers in columns 73 to 80 of each card image.

The WRite DeCLarations routine (WRDCL) does the same for writing PL/1 declarations and indents the declarations according to the level numbers for readability. It is called by GPL1DCL in order to write out each declaration. It is passed two parameters: the string containing the declaration, and the level in the tree. The file to which the declarations are written is PL1DCL.

5.10 Code Generation Summary

The "Code Generation Summary" routine (CGSUM) has the task of wrapping up the code generation phase and writing a report to the user.

First, the different files with the generated PL/1 program (PL1DCL, PL1ON, PL1EX, PL1PROC) are merged (by MERGPL1) into one object PL/1 file (PL1OBJ) which can be subsequently compiled. Secondly, a Code Generation Summary Report is written which lists the generated PL/1 program to the user, and prints out the total number of lines generated. While the PL/1 listing would not be of much use to the average MODEL user, it would provide a deeper understanding for the more sophisticated user or system programmer for insight or debugging. This is analogous to the way that a PL/1 compiler can list a pseudo-assembly language listing for the object program that it generates, which can be of occasional use to certain users.

This routine also generates a few lines of statistics about the generated program that might be useful for the user, including the number of PL/1 statements generated and the amount of computer time used to generate the program.

The result of this entire code generation process is thus a complete PL/1 program ready to be compiled by the PL/1 compiler.