cs37

COGENT 1.2

OPERATIONS MANUAL

BY

JOHN C. REYNOLDS

TECHNICAL REPORT CS37

APRIL 22, 1966

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

COGENT 1.2

OPERATIONS MANUAL

John C. Reynolds

Computer Science Department, Stanford University

and

Applied Mathematics Division, Argonne National Laboratory

April 22, 1966

(Work performed partially under the auspices of the U.S. Atomic Energy Commission, and partially under the Computer Science Department, Stanford University.)

PREFACE

This document is an addendum to the COGENT Programming Manual (Argonne National Laboratory, ANL-7022, March 1965, hereafter referred to as CPM) which describes a specific implementation of the COGENT system, COGENT 1.2, written for the Control Data 3600 Computer.

Chapters I and II describe a variety of features available in COGENT 1.2 which are not mentioned in CPM; these chapters parallel the material in Chapters II and III of CPM. Chapter III of this report gives various operational details concerning the assembly and loading of both COGENT-compiled programs and the compiler itself. Chapter IV describes system and error messages.

Familiarity with the contents of CPM is assumed throughout this report. In addition, a knowledge of the 3600 operating system SCOPE, and the assembler COMPASS is assumed in Chapter III.

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER I

EXTENSIONS TO THE COGENT LANGUAGE

All features described in Chapter II of CPM have been implemented in COGENT 1.2. The following additional features have been added to the language:

A. Constants

The format of constants in COGENT has been extended to allow a much greater variety of list structures to be represented. The productions (CPM, p. 54)

```
(constant) ::= ((open phrase class name⟩/⟨object string⟩)|

     (positive integer)
```

should be replaced by

```
(constant) ::= ((open phrase class name)/ (object string⟩)|

     ($IDENT,⟨positive integer⟩/⟨identifier object string)))

     (positive number)!-(positive number⟩|

     $$⟨object character representative⟩|**|

     $CSB(⟨template constant⟩⟨constant synthesis string))

(identifier object string) ::= ⟨empty⟩|

     (identifier object string⟩⟨object character representative⟩

(integer) ::= (positive integer⟩|-⟨positive integer)

(floating-point digit string) ::= *(digit string⟩| (digit string⟩*|

     (digit string⟩*⟨digit string)

(positive floating-point number) ::= (floating-point digit string⟩|

     (floating-point digit string⟩B

(unscaled positive number) ::= (positive integer))

     (positive floating-point number)

(scale factor) ::= E⟨integer⟩|Q⟨integer⟩

(positive number) ::= (unscaled positive number))
```

(positive number)⟨scale factor)

(template constant) ::= ⟨constant⟩| (name)

(constant synthesis string) ::= ⟨empty⟩|

(constant synthesis string⟩,⟨constant synthesis item)

(constant synthesis item) ::= ⟨empty⟩| (constant)) (name)

1. A constant of the form ((open phrase class name⟩/⟨object string)) denotes the list structure obtained by parsing the object string with respect to the goal specified by the phrase class name (CPM, p. 54).

2. A constant of the form ($IDENT,n/s) where s is an identifier object string, denotes an identifier element in table n containing a string of the output codes for each character in s . The string s is not parsed, and need not conform to the object language syntax.

3. Constants of the form (positive number) or -(positive number) denote positive and negative number elements respectively. Within an unscaled positive number, an asterisk indicates floating-point and acts as a decimal point, while the letter B indicates an octal representation. An unscaled positive number may be followed by one or more scale factors of the form En (or Qn). These scale factors are interpreted from left to right and cause the denoted value to be multiplied by 10^n (or 2^n) without changing the mode. When the mode is integer, the denoted value is truncated to an integer after each scale factor multiplication. For example,

399Q-2E2 denotes the integer 9900_{10}

1*24BE2 denotes the floating-point number 131.25_{10}

The use of scale factors is subject to the following restrictions:

a. Within each scale factor, the integer n must satisfy $-1023_{10} \leq n \leq 1023_{10}$.

b. If the mode is floating-point, then the initial unscaled positive number, as well as the result of applying each scale factor, must fall within the representable range of normalized double-precision floating-point numbers in the 3600.

4. A constant of the form $$⟨object character representative) denotes an integer number element giving the output code for the object character representative. For example, if the character description in a program is

$CHARDEF($EF) = (101)100.

then

$$A denotes 21_8

$$(( ) denotes 74_8

$$($EF) denotes 100_8

5. The constant ** denotes the dummy element.

6. The format $CSB(⟨template constant⟩⟨constant synthesis string⟩) is provided to allow constants to denote list structures with mixed syntax. Its effect is analogous to a synthetic assignment statement, but the indicated synthesis is carried out when the COGENT program is compiled, instead of when it is executed. Specifically, the list structure denoted by a constant with this format is obtained by copying the structure denoted by the template constant and replacing each parameter element with index i by the value of the ith constant synthesis item (numbered from left to right). If the ith item is empty, or if the synthesis string contains fewer than i items, then any parameter element with index i will be copied without replacement.

In COGENT 1.2, when a constant synthesis item replaces a parameter element, the list structure denoted by the synthesis item is itself copied. This situation, which prevents the $CSB-format from denoting list structures with common sublists (except identifier elements), may be altered in future versions of COGENT.

Within the $CSB-format, either the template constant or any constant synthesis item may be a name instead of a constant; in this case the name must be a pseudo-constant.

B. Expressions

The following should be added to the productions describing compound expressions (CPM, p. 64)

⟨compound expression⟩ ::= $SB(⟨template expression⟩⟨synthesis string⟩)

A compound expression with this format is evaluated as follows:

1. The template expression and all expressions in the synthesis string are evaluated. The order of evaluation is undefined and will be chosen to optimize code. If the evaluation of any of these expressions fails, then the evaluation of the entire compound expression fails, without evaluating further subexpressions or performing step 2.

2. An instantiated copy of the value of the template expression is formed and taken as the value of the compound expression.

In effect, this type of compound expression is similar to a synthetic assignment statement, i.e.

⟨name⟩ = $SB(⟨template expression⟩⟨synthesis string⟩) .

is completely equivalent to

⟨name⟩/=⟨template expression⟩⟨synthesis string⟩.

The advantage of the $SB-expression is that it allows list synthesis to be performed within a larger compound expression. For example,

OUTPUT($SB((TERM/(FACTOR)*(FACTOR)), X, Y)).

C. Interjections

The following productions should be added to the syntax of COGENT:

⟨comment character⟩ ::= ⟨normal character⟩|(|)| ,

⟨comment string⟩ ::= ⟨empty⟩| ⟨comment string⟩⟨comment character⟩

⟨interjection⟩ ::= $COMMENT ⟨comment string⟩.|

$TITLE ⟨comment string⟩.

⟨COGENT program⟩ ::= ⟨interjection⟩⟨COGENT program⟩

⟨character definition sequence⟩ ::=

⟨character definition sequence⟩⟨interjection⟩

⟨primary production sequence⟩ ::=

⟨primary production sequence⟩⟨interjection⟩

⟨secondary production sequence⟩ ::=

⟨secondary production sequence⟩⟨interjection⟩

⟨declaration sequence⟩ ::= ⟨declaration sequence⟩⟨interjection⟩

⟨generator definition⟩ ::= ⟨generator definition⟩⟨interjection⟩

⟨statement⟩ ::= ⟨statement⟩⟨interjection⟩

⟨statement label⟩ ::= ⟨statement label⟩⟨interjection⟩

Generally, an interjection may appear anywhere in a COGENT program where a character definition, production, declaration, generator definition, or statement may appear. One or more interjections may also appear at the beginning of a COGENT program.

1. An interjection beginning with $COMMENT has no effect on the compilation of a COGENT program.

2. An interjection beginning with $TITLE has no effect on the program produced by the COGENT compiler, but affects the printed listing

produced by the compiler. In general, each page of printed output from the compiler will be headed by a line giving a title, the current date, and a page number. The title field in this heading will be the comment string contained in the last-encountered $TITLE-interjection, or if no $TITLE-interjection has been encountered, the title field will be blank. Whenever a $TITLE-interjection is encountered, the next line printed by the compiler will be ejected to a new page.

In deriving a title field from the comment string in a $TITLE-interjection, blanks will be deleted, and if the string exceeds 87 (non-blank) characters, it will be truncated to the first 87 characters.

CHAPTER II

ADDITIONAL PRIMITIVE GENERATORS AND INTERNAL VARIABLES

All of the primitive generators and internal variables described in CPM are available in COGENT 1.2, except the dump primitives DUMPV, DUMP1, and DUMPALL. These three primitives may be called in COGENT 1.2 programs, but they will simply output a system comment and return the dummy element as their result.

This chapter describes the additional primitives and internal variables which have been implemented in COGENT 1.2, as well as various generalizations of earlier primitives described in CPM.

A. Internal Variables (CPM, p. 71)

1. The following internal variables have been added to the system:

rdm (random number). This internal variable is reset whenever the primitive generator RANDOM(X) is called. Successive calls of RANDOM will cause rdm to cycle through its legal values in a pseudo-random manner,

Initial value: 1 Legal values: odd integers such that

Used by: RANDOM. $1 \leq rdm \leq 2^{47} - 1$.

Reset by: RANDOM,

pct (point count). This internal variable is reset whenever a sequence of character output codes contained in the character buffer is converted into a number, either by the primitives DECCON, OCTCON, or FLOATCON, or by the syntax analyzer when controlled by a $DEC/, $OCT/, or $FLOAT/ special label. If the character sequence in the buffer does not contain any non-digits, then pct is set to the dummy element. Otherwise pct is set to the number of digits following the last non-digit in the sequence.

Initial value: dummy element. Legal values: $0 \leq pct < 1023_{10}$,

plus dummy element.

Reset by: DECCON, OCTCON, FLOATCON, syntax analyzer.

sno (S-medium logical unit number). This internal variable is provided to allow the comments and error messages produced by various system routines to be written on an output device different than the P-medium device (specified by the internal variable pno). If the value of sno is the dummy element, then system comments will be written on the output device whose SCOPE logical unit number is given by pno. But if $1 \leq sno \leq 80_{10}$, then the comments will be written on the output device whose unit number is given by sno. If sno = 0, then all system comments will be suppressed.

Initial value: dummy element. Legal values: $0 \leq sno \leq 80_{10}$,

plus dummy element.

Used by: system routines which output messages.

dno (D-medium logical unit number). This internal variable is intended to control the output of the dump primitives in the same manner that sno controls system messages. Since the dump primitives are not implemented in COGENT 1.2, dno has no effect in this version of the system, although its standard setting and evaluating primitives are available.

Initial value: dummy element. Legal values: $0 \leq \underline{dno} \leq 80_{10}$, plus dummy element.

Used by: DUMP1, DUMPV, DUMPALL (eventually, but not in COGENT 1.2).

2. The following standard primitives (CPM, p. 77) are available for setting and evaluating the new internal variables described above:

SETIVRDM(X)	sets rdm.
SETIVPCT(X)	sets pct.
SETIVSNO(X)	sets sno.
SETIVDNO(X)	sets dno.
IVRDM( )	evaluates rdm.
IVPCT( )	evaluates pct.
IVSNO( )	evaluates sno.
IVDNO( )	evaluates dno.

3. Zero has been made a legal value for the internal variables pnon, o , and bno (CPM, pp. 74-76). When any of these variables has the value zero, output produced for the corresponding output medium will be suppressed. The various output primitives will still function in their normal manner, but the actual records produced will be discarded rather than sent to an output device.

B. Testing Primitives (CPM; p. 79)

PSLARGER(X, Y). X and Y may be arbitrary list elements. PSLARGER fails unless $X > Y$ according to an arbitrary but fixed ordering of all list names; otherwise it returns the dummy element. The ordering defined by PSLARGER satisfies:

1. If $X > Y$ and $Y > Z$ then $X > Z$.

2. For any list names X and Y exactly one of the following holds: $X > Y$, $Y > X$, or $X = Y$ (where equality is defined in the sense of the primitive EQLIT).

C. Marking Primitives

Two one-bit components called mark1 and mark2 have been added to all normal list elements containing one or more name-components, i.e, to all non-literal normal elements. These mark components have the following properties:

1. All non-literal normal elements produced by the syntax analyzer, as well as all such elements appearing in constant list structures have both mark components set to zero.

2. When a non-literal normal element is created by an instantiated copy (e.g., by a synthetic assignment statement), the mark components are copied without alteration,,

3. When two non-literal normal elements are compared by an analytic assignment statement, the comparison fails unless both mark components match.

The following primitive generators test and manipulate the mark1 component:

TSTMARK1(X)	leaves mark1 unchanged.
SETMARK1(X)	sets mark1 to 1 .
CLRMARK1(X)	sets mark1 to 0 .
CMPMARK1(X)	complements mark1.

The argument of each of these primitives must be a non-literal normal element. Each of these generators will return a dummy result if the mark1 component of this element is 1, and will fail if it is 0 . However, before returning or failing, the generators will reset the mark1 component as shown above. These marking primitives must not be used to reset elements which appear in constant list structures.

The following primitives test and manipulate the mark2 component:

TSTMARK2(X)	leaves mark2 unchanged.
SETMARK2(X)	sets mark2 to 1 .
CLRMARK2(X)	sets mark2 to 0 .
CMPMARK2(X)	complements mark2.

Their operation is similar to that described above.

D. Arithmetic Primitives (CPM, p. 81)

RANDOM(X). X must be an integer or floating-point number element. RANDOM first resets the value of the internal variable rdm to $\rho(\underline{rdm})$, and then returns the result $(X \cdot \underline{rdm})/2^{47}$. This result is computed and returned in the same mode as X .

The function ρ is chosen so that successive values of rdm cycle through the odd integers between 1 and $2^{47} - 1$ in a pseudo-random manner. The currently-used definition of ρ is

$$\rho(\underline{rdm}) = (5^{15} \cdot \underline{rdm}) \bmod 2^{47}$$

E. Output Primitives (CPM, p. 96)

To increase the flexibility of output operations, a number of facilities have been added to COGENT 1.2, including the insertion of page headings in printed output, the conversion of BCD card images into equivalent binary images, the insertion of checksums in binary card images, and the suppression of certain system messages.

1. The Tape Table

To control these added facilities, a tape table has been introduced. This table, which exists during the running of all COGENT programs, is indexed by SCOPE logical unit number, and contains the following entries for each unit number:

a. A paging flag.

b. An integer called the line count.

C. An integer called the page count.

d. A list name called the title,, which must be either an identifier or the dummy element.

e. A C-medium conversion flag.

f. A B-medium checksum flag.

When program execution begins, all flags in the tape table are turned off, all line and page counts are set to zero, and all titles are set to the dummy element.

2. Paging

Whenever a P-medium record image, or a print-line image produced by a system comment routine (or an image produced by a dump generator, when these generators are implemented) is sent to an output unit with logical unit number i, then if the paging flag for i is off, the record is outputted without alteration. But if the paging flag is on, the following occurs:

a. If the carriage-control character of the record image is a blank (indicating single-spacing) the line count for unit i is decreased by one. If the carriage-control character is 0 (indicating double-spacing) the line count is decreased by two.

b. If the line count is negative, or if the carriage-control character is 1 (indicating page ejection), then:

(1) The page count for unit i is increased by one.

(2) A special print line containing a carriage-control character of 1 (page ejection), the current date, and the page count is written on unit i . If the title entry for unit i is an identifier (rather than the dummy element), the character string of this identifier will also appear in this print line.

(3) The line count for unit i is set to 56.

(4) The carriage-control character in the record image is set to 0 (double-space).

c. The record image is written on unit i .

If the title is an identifier with more than 87 characters, only the first 87 characters will be printed. The title identifier should not contain any output codes larger than 74_8.

3. <u>BCD-to-Binary Card Image Conversion</u>

Whenever a C-medium record image is sent to logical-unit number i, then if the C-medium conversion flag for unit i is off, the record is outputted without modification. But if the C-medium conversion flag is on, then the record will be replaced by a binary record which is equivalent, i.e., which will cause the same card to be punched.

4. <u>Checksum Insertion</u>

Whenever a B-medium record image is sent to logical unit number i, then if the B-medium checksum flag for unit i is off, the record is outputted without modification. But if the checksum flag is on, a checksum for the card image will be computed and inserted in bit positions 25-48, corresponding to columns 3 and 4 of the card. This checksum is computed according to the standard conventions for CDC 3600 binary cards.

5. <u>Tape-Table Primitives</u>

The following primitives set or reference entries in the tape table. In all cases, the argument LUN must be an integer number element denoting a SCOPE logical unit number. All of these primitives except PGCNT return the dummy element.

PAGE(LUN, T). T must be an identifier element or the dummy element. The paging flag for logical unit LUN is turned on, the title entry is set to the value of T, and the line count is set to zero.

NOPAGE(LUN). The paging flag for unit LUN is turned off.

PGCNT(LUN). Returns an integer number element giving the page count for unit LUN.

CLRPGCNT(LUN). Sets the page count for unit LUN to zero.

CMDCNV(LUN). The C-medium conversion flag is turned on.

NOCMDCNV(LUN). The C-medium conversion flag is turned off.

BCHKSM(LUN). The B-medium checksum flag is turned on.

NOBCHKSM(LUN). The B-medium checksum flag is turned off.

The argument LUN = 0 is allowed for all of the tape-table primitives. When LUN = 0 the tape table is not altered, and the dummy element is returned.

6. Running-Message Suppression

Most of the messages produced by system routines are produced either at the beginning of program execution or at program termination. However, certain messages called running messages may occur at arbitrary points during program execution. These include:

a. A comment whenever list storage recovery occurs.

b. A comment whenever 100 successive characters are read in the ambiguity mode.

The output of these messages is now conditioned by a running-message flag; if this flag is off, the running messages will be suppressed.

The running-message flag is-turned on when program execution begins, and may be altered by the following primitives:

RUNMSS( ). Turns the running-message flag on,

NORUNMSS( ). Turns the running-message flag off.

Both of these no-argument primitives always return the dummy element.

F. Tape-Control Primitives (CPM, p. 103)

The following tape-control primitives have been added:

SKIPR(LUN). Skips one record on the tape denoted by LUN. The dummy element is returned.

MASTLUN(LUN). Returns an integer number element giving the master logical unit number for the unit LUN. The master logical unit number is the unit number to which unit LUN has been equivalenced by SCOPE control cards or by SCOPE itself. In the absence of any equivalencing, the master number is LUN itself.

All of the tape-control primitives now accept the argument LUN = 0; when LUN = 0, the primitives perform no action and return the dummy element. When LUN ≠ 0, the primitives act by means of appropriate calls of the routine IOP., which is now used for all input-output operations in COGENT. This routine is described in the CDC 3600 FORTRAN Maintenance Manual.

Two characteristics of the tape-control primitives should be noted:

1. The tape-control primitives do not affect the tape table. Thus, for example, if a tape which is being paged is rewound, appropriate tape-table primitives should be called to reset the page and line counts.

2. The routine IOP. blocks records on SCOPE unit 61 when this unit is magnetic tape. On this unit, the primitives BSPR and SKIPR will move the tape by physical rather than logical records.

G. Running-Status Primitives

The following primitives are provided to furnish information about the running status of a COGENT program:

DATE( ). Returns an 8-character tableless identifier giving the date on which program execution began in the format mm/dd/yy .

TIME( ). Returns an 8-character tableless identifier giving the current time of day in the format hhmmb-ss, where b indicates a blank. In the character-string components of the identifiers returned by DATE and TIME, standard output codes (CPM, p. 39) are always used, even when otherwise overridden by character definitions.

CLOCK( ). Returns an integer number element giving the number of milliseconds remaining before program termination, i.e., before the program will be automatically terminated by SCOPE.

ICOUNT( ). Returns an integer number element giving the number of calls of the input editor which have occurred since program execution began.

FREELIST( ). Returns an integer number element giving the number of machine words in free list storage, i.e., the number of words available for creating list structures before the next list storage recovery.

COLLECT( ). Similar to FREELIST( ), except that a list storage recovery is performed before free list storage is counted.

CHAPTER III

OPERATING INFORMATION FOR COGENT 1.2

A. Subprogram Structure

In relocatable binary form, a COGENT program always consists of the following subprograms:

PROG	BETADATA	ANAGEN
STACK	IOP.	ARITHMTC
LIST	ALLOC.	SCAN
	INITIAL	OUTPUT
	INEDITOR	IDENT
	INRPEXIT	MISCPRIM
	SYNTAX	DUMP
	SUBGEN	GARBCOLL

(Each of these subprograms contains an entry point whose name is the same as the subprogram name.)

The three subprograms PROG, STACK, and LIST are produced by the COGENT compiler. The remaining subprograms are the same for all COGENT programs and are called the running deck. The two subprograms IOP. and ALLOC. are an input-output buffering package taken from the CDC 3600 FORTRAN library; the remaining subprograms in the running deck are written especially for COGENT. INITIAL is the main subprogram.

Two numbered common blocks occur. /1/ is a 1600_8-word bit table referred to by GARBCOLL. /2/ is one-word block referred to by INITIAL; its only use is to insure that the loader will be overlaid (when the STACK-bank is 0).

It should be emphasized that since the COGENT compiler is written in its own language and compiled by itself, it is merely a particular case of a COGENT program. Thus in relocatable binary form, the compiler consists of three subprograms PROG, LIST, and STACK, plus the same running deck as would be used with any other COGENT program.

B. Loading

1. Bank Allocation

Tne four subprograms PROG, STACK, IOP., and ALLOC. may be placed in arbitrary banks (except for a restriction on STACK described below); we will refer to these banks as the PROG-bank, STACK-bank, etc. The remaining subprograms are restricted as follows: BETADATA, LIST, INITIAL, and common block /2/ must go into the STACK-bank; all other subprograms and common block /1/ must go into the PROG-bank.

To insure proper bank allocation, all of the subprograms except PROG, STACK, IOP., and ALLOC, contain (in COMPASS) bank pseudo-instructions which allocate these subprograms (and also the two common blocks) to the same bank as PROG or STACK. (Thus in binary- form these subprograms begin with bank control cards rather than IDC cards.) Therefore to specify bank allocations while loading a COGENT program, it is only necessary to provide a bank control card which gives absolute allocations for PROG, STACK, IOP., and ALLOC. Tnis card should preceed the first binary deck.

2. Further Restrictions

The following additional restrictions are imposed on the loading of COGENT programs. Violations of these restrictions will cause program termination immediately after loading.

a. Tne subprogram STACK must be the first-loaded (highest-addressed) subprogram in the STACK-bank. Furthermore, either the STACK-bank must be bank 0, or else no subprogram may be placed in a bank whose bank address is larger than the STACK-bank. These restrictions are necessary to allow the pushdown stack to be protected by the bounds register, so that the exhaustion of the pushdown stack causes a bounds fault.

b. The subprogram INITIAL must be the last-loaded (lowest addressed) subprogram in the STACK-bank, and must be immediately preceeded by LIST (so that the lowest address in LIST is one larger than the highest address in INITIAL). The routine INITIAL allocates a portion of available memory in the STACK-bank, as well as the area occupied by INITIAL itself, to be used for non-constant list storage. These restrictions insure that these two areas are contiguous and that they are adjacent to the area in LIST wnich contains constant list structures.

c. Tne entry point LISTCHCK in the subprogram LIST must have an absolute address of less than or equal to 70000_8. This restriction assures that no list element will have an address within the range of literal list names. (This restriction will not be violated as long as the assembly control parameter STACKLEN in STACK is larger or equal to 4096_{10}.)

3. Additional Comments

The restrictions discussed in-the preceeding sections still permit a variety of memory allocations for a COGENT program. Normally, the optimum choice of allocation will be determined by two goals: (i) The program must fit within the total memory available; (ii) The available memory (remaining after loading) in the STACK-bank should be as large as possible to maximize the size of list storage. Maximizing list storage will increase the speed of a COGENT program by reducing the frequency of list storage recoveries.

To acheive these goals, the following should be noted:

a. The subprogram DUMP contains only the primitive generators DUMPV, DUMP1, DUMPALL, SETIVDNO, and IVDNO. It may be omitted for any COGENT program wnich does not call these primitives. (In COGENT 1.2

DUMPV, DUMP1, and DUMPALL are dummy routines which merely produce system comments.)

b. Large COGENT programs contain a very large number of entry points and external symbols. In the case Where the PROG-bank is 0 and the STACK-bank is not 0, the program size is limited by the collision of program space with loader tables. To minimize this limitation, it is advisable that the last-loaded subprogram should go into the PROG-bank and should have reasonable size but only a small number of entry points and external symbols. The best candidate for this position appears to be GARBCOLL.

c. By means of assembly control parameters, the running deck may be assembled in either a "safe" or a "fast" version (see section C.1). The fast version is significantly shorter.

4. Suggested Memory Allocations

The following memory allocation for two banks has been used extensively and is recommended for most programs. If the program (and/or the SCOPE resident) is large enough to cause memory overflow in bank 0, then IOP. and ALLOC. should be moved to bank 1 and placed between BETADATA and LIST.

Bank 0	Bank 1
IOP.	(SCOPE drivers)
ALLOC.	STACK
PROG	BETADATA
INEDITOR	LIST
INRPEXIT	INITIAL
SYNTAX	(available memory)
SUBGEN	Common block /2/
ANAGEN	
ARITHMTC	
SCAN	
OUTPUT	↑ increasing address
IDENT	
MISCPRIM	
DUMP	
GARBCOLL	
(available memory)	
Common block /1/	
(SCOPE resident)	

The following memory allocation is suggested for one-bank loading. The COGENT compiler itself is too large to fit in one bank, but many smaller COGENT programs can be run with one bank.

Bank 0

(SCOPE drivers)
STACK
BETADATA
IOP.
ALLOC.
PROG
INEDITOR
INRPEXIT
SYNTAX
SUBGEN
ANAGEN
ARITHMTC
SCAN ↑ increasing
OUTPUT address
IDENT
MISCPRIM
DUMP
GARBCOLL
LIST
INITIAL
(available memory)
Common blocks /1/ and /2/
(SCOPE resident)

C. Assemblv Control Parameters

As written in assembly language, most of the subprograms comprising a COGENT program contain one or more assembly symbols called assembly control parameters. These symbols, which are defined by EQU pseudo-instructions near the beginning of the subprograms, are provided to allow various characteristics of the program to be modified easily. The following is a complete list of the assembly control parameters in COGENT 1.2, giving the name of each parameter, the value given to the parameter in the subprogram versions on the COGENT 1.2 Master Tape, the names of the subprograms containing the parameter, and for each subprogram the COSY line number of the card defining the parameter;

Assembly Control Parameter	Value on Master Tape	Subprograms Using Parameter	Cosy Card Number
SAFE	1	ARITHMTC	5
		SCAN	5
		OUTPUT	5
		IDENT	5
		MISCPRIM	5
VERSCOPE	5	INRPEXIT	5
		OUTPUT	7

cont.

Assembly Control Parameter	Value on Master Tape	Subprograms Using Parameter	Cosy Card Number	
MINBUFF1	0	INITIAL	11	
MINSNSK1	1500	INITIAL	12	
MINLIST1	510	INITIAL	13	Storage Allocation Parameters for One-bank Loading
EXCSNSK1	10	INITIAL	14	
EXCLIST1	80	INITIAL	15	
MINBUFA2	0	INITIAL	17	
MINSNSK2	3000	INITIAL	18	
MINBuFB2	0	INITIAL	19	Storage Allocation Parameters for Two-bank Loading
MINLIST2	510	INITIAL	20	
EXCSNSK2	0	INITIAL	21	
EXCLIST2	100	INITIAL	22	
MINLIST	500	GARBCOLL	29	
PAGESIZE	59	OUTPUT	6	
DELBLANK	1	INEDITOR	5	
PSEOF	1	INEDITOR	6	
EOFCODE	101B	INEDITOR	8	
STACKLEN	4096	STACK	2	

1. SAFE

The parameter SAFE conditions the assembly of instructions which test the validity of arguments of primitive generators. When SAFE = 1, a safe version of the subprogram is assembled in which the calling of primitives with invalid arguments (in most cases) will cause abnormal program termination. When SAFE = 0, a fast version is assembled which is shorter and considerably faster than the safe version, but in which invalid arguments will have unpredictable consequences.

2. VERSCOPE

The parameter VERSCOPE should be set to 5 or 6, depending upon the version of SCOPE being used. When VERSCOPE = 6, code will be assembled to select interrupt on abnormal termination in order to close (output buffers for) logical unit 61 after an abnormal termination.

3. Storage Allocation Parameters

As soon as a COGENT program has been loaded, the routine INITIAL will determine the extent of available storage (the storage areas between subprograms and common blocks in each bank, plus the area occuppied by INITIAL itself) and divide this storage into areas for three purposes: list storage, the syntax stack, and input-output buffers. The amount of storage used for each purpose is determined by the eleven assembly control parameters called storage allocation parameters.

(To avoid confusion, it should be noted that a COGENT program contains two distinct storage areas called the stack (or pushdown stack) and the syntax stack, The stack is located in the subprogram STACK, and is used to store the local variables, input variables, and various temporary quantities used by generators. The syntax stack is located in available memory in the PROG-bank, and is used to store return addresses for recognizers, i.e., syntax-analysis subroutines,)

Two cases are distinguished and controlled by separate sets of parameters: one-bank loading, where the PROG- and STACK-banks are the same, and two-bank loading, where the PROG- and STACK-banks are different,, In both cases, the various storage areas are assigned minimum amounts specified by the parameters named MINxxxxx, and then the excess storage is divided among the areas according to percentages specified by the parameters named EXCxxxxx.

The allocation formulas for one-bank loading are:

excess = available storage size - MINBUFF1- MINSNSK1 - MINLIST1

syntax stack size = MINSNSK1 + excess x EXCSNSK1 / 100

list storage size = MINLIST1 + excess x EXCLIST1 / 100

buffer size = MINBUFF1 + excess x (100 - EXCSNSK1 - EXCLIST1) / 100

If the excess is negative, an error message will be written and abnormal termination will occur.

In two-bank loading, the syntax stack is allocated in the PROG-bank, and list storage is allocated in the STACK-bank. I/O buffers may be allocated in either bank or both. The allocation formulas are:

PROG-bank excess = available storage in PROG-bank - MINBUFA2 - MINSNSK2

STACK-bank excess = available storage in STACK-bank - MINBUFB2 - MINLIST2

syntax stack size = MINSNSK2 + PROG-bank excess x EXCSNSK2 / 100

PROG-bank buffer size =

MINBUFA2 + PROG-bank excess x (100 - EXCSNSK2) / 100

list storage size = MINLIST2 + STACK-bank excess x EXCLIST2 / 100

STACK-bank buffer size =

MINBUFB2 + STACK-bank excess x (100 - EXCLIST2) / 100

If either the PROG-bank or STACK-bank excess is negative, an error termination will occur. (For either one- or two-bank loading, slight deviations from the above formulas will occur; e.g., the syntax stack will always contain an even number of words.)

The storage allocation parameters must always satisfy the following restrictions:

All parameters must be non-negative.

MINLIST1 $>$ 510

EXCSNSK1 + EXCLIST1 $\leq$ 100

MINLIST2 $\geq$ 510

EXCSNSK2 $\leq$ 100

EXCLIST2 $\leq$ 100

In some cases there may be available storage in other banks than the PROG- or STACK-bank; this storage will be used entirely for I/O buffers. Specifically, if a bank b is not the PROG- or STACK-bank, and if either the STACK-bank = 0 or b < STACK-bank, then all available storage in b will be allocated for I/O buffers.

4. MINLIST

Tnis parameter determines the minimum size of free list storage. If a list storage recovery produces less than MINLIST words, then the program will terminate. MINLIST should always be at least 500.

5. PAGESIZE

This parameter appears in the routine which controls paging. it gives the maximum number of print lines (including the heading line) per page.

6. DELBLANK, PSEOF, and EOFCODE

These parameters control the assembly of the input editor., If DELBLANK $\neq$ 0, the editor will delete blank characters., If PSEOF $\neq$ 0, the editor will interpret a pseudo-end-of-file card (with asterisks in columns 1 to 72) as an end-of-file, EOFCODE gives the input code to be outputted by the editor for an end-of-file. The values DELBLANK = 1, PSEOF = 1, and EOFCODE = 101B are used for the standard input editor, i.e., the Version of the input editor which must be used with the COGENT compiler itself.

7. STACKLEN

This parameter determines the number of words used for the pushdown stack. When necessary it may be altered to provide a larger stack, or to provide more list storage or I/O buffers at the expense of the stack. However, a lower limit is imposed on STACKLEN by the requirement that location LISTCHCK in the subprogram LIST must have an absolute address less than or equal to 70000_8 (See section B.2.c. of this chapter), The exact lower limit depends 'upon the particular program and loading arrangement, but STACKLEN $\geq$ 4096 will always be sufficient.

STACKLEN is the only assembly control parameter which appears in a subprogram produced by the compiler rather than in the running deck,, Thus, in addition to altering STACKLEN by a COSY correction, it is also possible to alter the compiler itself to produce a different EQU for STACKLEN, Tnis may be done by changing the large constant in the generator PROGEND (in the version of the compiler written in its own language), and then recompiling the compiler.

DO The Compiler

As mentioned earlier, tne COGENT compiler is merely a specific case of a COGENT program, consisting of the subprograms PROG, STACK, and LIST. obtained by compiling the compiler, plus the usual running deck. Specifically, the compiler is a program which reads input in the COGENT language from SCOPE logical unit number 60, and produces COMPASS card images on logical unit 1, plus an output listing on logical unit 61. (Note that logical unit 1 is a programmer-defined unit.)

1. Loading

Like all COGENT programs, the compiler is subject to the loading restrictions discussed in section B of this chapter; since it is a very large program, the comments in B.3 are especially pertinent. The subprogram DUMP may be omitted, and the "fast" versions of ARITHMTC, SCAN, OUTPUT, IDENT, and MISCPRIM should be used.

The memory allocation for two-bank loading given in section B.4 is recommended, (with DUMP omitted) although it may be necessary to place. IOP. and ALLOC. in bank 1 rather than bank 0 if the SCOPE resident is large. The compiler is too large for one-bank loading.

2. input

Tne input program to be translated by the compiler must appear as a sequence of BCD card images on logical unit 60, Tnis program should be immediately preceeded by the RUN-card which initiates execution of the compiler, and immediately followed by either an end-of-file or a pseudo-end-of-file card (asterisks in columns 1-72). If the compiler terminates normally, it will leave tape 60 positioned immediately beyond the end-of-file or pseudo-end-of-file card,

3. COMPASS Output

The primary output of the compiler is a sequence of BCD COMPASS (not COSY) card images on logical unit 1. If the compiler terminates normally, this tape will have the following format:

```
IDENT          PROG
  .
  .
END
SCOPE
IDENT          STACK
  .
  .
  .
END
SCOPE
IDENT          LIST
  .
  .

EM;
SCOPE
end-of-file
```

and the tape will be rewound by the compiler.

Tne COMPASS output will include REM cards indicating the beginning of each generator, each recognizer (syntax analyzer subroutine), and various tables. In addition, many machine instructions will have one or more names, phrase class names, or statement numbers listed in their comment fields; these comments are provided to identify various internally generated assembly symbols appearing in the corresponding instructions.

4. Printed Output

The compiler also produces printed output on logical unit 61. This output includes an image of each card in the COMPASS output, plus

a. A table of all primary productions (with compound productions reduced to sets of simple productions) along with their production code numbers. This table appears at the beginning of the printed output.

b. A similar table of all secondary productions, appearing between the COMPASS-instruction listing of the syntax analyzer and the listing of the production code number table.

C. Various system comments and error messages.

d. At the end of the printed output, a count of the number of error messages.

Tne printed output does not include a listing of the input program.

5. Output Volume

Tne compiler frequently produces a very large valume of both COMPASS and printed output. As an extreme example, when the compiler is used to compile itself, it reads 1491 cards and produces about 28000 COMPASS card images plus 622 pages of printed output. Because of this volume of output:

a. Tne COMPASS output should not be punched directly onto cards. It is usually better to use the COMPASS assembler to convert this output into COSY desks.

b. For large programs such as the compiler itself, the COMPASS output may extend onto a continuation reel.

E. 'The COGENT Master Tape

Tne COGENT system is normally distributed in the form of a COGENT Master Tape. Tnis tape contains an 80-character (SCOPE 6.x-type) label with the name (COGENTb1.2bMAS), followed by two files.

Tne first file consists of 21 binary COSY decks for the following subprograms (in order):

STACK
BETADATA
IOP. } versions for use witn SCOPE 5.4
ALLOC }
IOP. } versions for use with SCOPE 6.0
ALLOC. }
LIST
INITIAL
PROG
INEDITOR
INRPEXIT
SYNTAX
SUBGEN
ANAGEN
ARITHMTC
SCAN
OUTPUT
IDENT
MISCPRIM
DUMP
GARBCOLL

The subprograms STACK, LIST, and PROG are specific to the COGENT compiler, while the remaining subprograms constitute the running deck and are used with all COGENT programs. Tne versions of IOP. and ALLOC. are taken from the FORTRAN library of a particular 3600 installation (Argonne); at other installations it may be necessary to replace these subprograms by local versions.

Tne second file of the master tape contains BCD card images giving the program for the COGENT compiler in its own language. Some of the card images in this file contain sequence numbers in columns 77-80; these numbers pertain to an earlier version of the compiler and should be ignored,

F. Illustrative Job Decks

Tne following sections give illustrative job decks for the preparation of a COGENT system from the Master Tape, and for the compilation, assembly, and execution of a COGENT program. Numerous variations are possible to meet the needs of particular installations, users, or programs, In the following, the 'symbol "?" is used to denote a 7-9 punch, and the symbol "/c" is used to denote a 11-0-7-9 punch.

1. To prepare a load-and-go tape of the COGENT compiler to be used with SCOPE 6.0 or 6.1:

```
?JOB, charge number,id,20
?EQUIP,1=(COGENTbl.2bMAS),SV
?EQUIP,2=(COGENTbl.2bBIN),SV
?FILE,2
/cBANK,(0),PROG,(1),STACK,(0), IOP.,ALLOC.}  or if       {... ,(1),IOP.,ALLOC.
?FILE END                                    necessary
?COMPASS,Y=1,X=2,L,R,M
          COSY      STACK
          COSY      BETADATA
          BYPASS    2      }      {                    BYPASS    4
          COSY      IOP.   }      {                    SCOPE
          COSY      ALLOC. }  or  {?FILE,2
                                  {     Binary relocatable decks for local
                                  {     versions of IOP. and ALLOC,
                                  {?FILE END
                                  {?COMPASS,Y=1,X=2,L,R,M
          COSY      LIST
          COSY      INITIAL
          COSY      PROG
          COSY      INEDITOR
          REPLACE   5
VERSCOPE  EQU       6
          COSY      INRPEXIT
          COSY      SYNTAX
          COSY      SUBGEN
          COSY      ANAGEN
          REPLACE   5
SAFE      EQU       0
          COSY      ARITHMTC
          REPLACE   5
SAFE      EQU       0
          COSY      SCAN
          REPLACE   5
SAFE      EQU       0
```

```
          REPLACE   7
VERSCOPE  EQU       6
          COSY      OUTPUT
          REPLACE   5
SAFE      EQU       0
          COSY      IDENT
          REPLACE   5
SAFE      EQU       0
          COSY      MISCPRIM
          BYPASS    1
          COSY      GARBCOLL
          SCOPE
```

2. To produce a safe version of the running deck (on binary relocatable cards) for use with SCOPE 6.0 or 6.1:

```
?JOB,charge number,id,10
?EQUIP,1=(COGENTb1.2bMAS),SV
?COMPASS,Y=1,P,L,R,M
          BYPASS    1
          COSY      BETADATA
          BYPASS    2     ⎫
          COSY      IOP.  ⎬  or if local versions of        ⎧BYPASS   5
          COSY      ALLOC.⎪  IOP. and ALLOC. are to be used ⎩
          BYPASS    1     ⎭
          COSY      INITIAL
          BYPASS    1
          COSY      INEDITOR
          REPLACE   5
VERSCOPE  EQU       b
          COSY      INRPEXIT
          COSY      SYNTAX
          COSY      SUBGEN
          COSY      ANAGEN
          COSY      ARITHMTC
          COSY      SCAN
          REPLACE   7
VERSCOPE  EQU       6
          COSY      OUTPUT
          COSY      IDENT
          COSY      MISCPRIM
          COSY      DUMP
          COSY      GARBCOLL
          SCOPE
```

3. To compile, assemble, and execute an COGENT program using the two-bank allocation suggested in section B.4 of this chapter:

```
?JOB,charge number,id,total time limit
?EQUIP,2=(COGENTb1.2bBIN),SV
?EQUIP,1=SV                                (include to save compiler output)
?LOAD,2
```

```
?RUN,,compilation time limit,print limit,1           (compilation)
     COGENT Program to be compiled
     Card with asterisks in columns 1-72
?COMPASS,I=1,X,P,L,R,M                               (assembly of PROG)
?COMPASS,I=1,X,P,L,R,M                               (assembly of STACK)
?FILE,69
     Binary relocatable deck for BETADATA
?FILE END
?COMPASS,I=1,X,P,L,R,M                               (assembly of LIST)
/BANK,(0),PROG,(1),STACK,(0),IOP.,ALLOC.
     Binary relocatable decks for: IOP.
                                   ALLOC.
?LOAD,69
     Binary relocatable decks for: INITIAL
                                   INEDITOR
                                   INRPEXIT
                                   SYNTAX
                                   SUBGEN
                                   ANAGEN
                                   ARITHMTC
                                   SCAN
                                   OUTPUT
                                   IDENT
                                   MISCPRIM
                                   DUMP
                                   GARBCOLL
?RUN,execution time limit,print limit,1              (execution)
     Data cards
```
With SCOPE 6.0 or 6.1, this type of job cannot be used if the program is so large that the compiler output on logical unit 1 runs onto a continuation reel. In this situation, compilation and assembly must be performed as separate jobs,

CHAPTER IV

SYSTEM AND COMPILER ERROR MESSAGES

Two types of "messages" occur in COGENT: System messages are produced by routines in the running deck, and may appear during the execution of any COGENT program, including the compiler, Some of these messages indicate the cause of an error termination, while others may occur during the operation of a correct program, Compiler error messages are produced only by the COGENT compiler; these messages never cause program termination,

A. System Messages

In general, system messages are outputted to the logical unit specified by the internal variable sno (See II.A.1). During the operation of the compiler these messages appear on logical unit 61.

For each message described in this section, a reference by sub-program name and COSY line number is given to the code within the running deck which produces the message.

1. Initialization Message (INITIAL, 152)

As soon as a COGENT program begins execution, the subprogram INITIAL will output a message giving the current time of day and a description of the allocation of available memory:

```
EXECUTION STARTED AT hhmm -ss

ALLOCATION OF AVAILABLE MEMORY
b fffff LIST STORAGE sssss
b fffff SYNTAX STACK sssss
b fffff I/O BUFFERS sssss
```

The octal numbers b, f, and s are the bank, lowest address, and length of each section of storage, If I/O buffers occur in more than one bank, then a line will be written for each bank containing buffers,

If INITIAL causes an error termination because of improper loading order or inadequate available memory, the initialization message will not occur,

2. Running Messages

These messages may occur anytime during program execution. Their output is conditioned by the primitives RUNMSS and NORUNMSS (See II.E.6).

a. List Storage Recovery (GARBCOLL, 386)

The following message occurs at the completion of each list storage recovery:

```
(LIST STORAGE RECOVERY, f WORDS RECOVERED.  p WORDS IN ACTIVE PUSHDOWN
                    STACK.  ELAPSED TIME t MS)
```

where f is the size of the recovered free list storage area, and t is the time (in milliseconds) taken by the recovery routine. The quantity p is the number of active words in the pushdown stack (excluding the sub-stack) and represents the memory in use for the input variables, local variables, and temporary storage of all generators in the calling chain at the instant when recovery occurred. All numbers in this message are decimal.

b. Ambiguity-Mode Character Count (SYNTAX, 430)

An ambiguity-mode character counter is initialized to zero whenever the ambiguity mode is entered by the syntax analyzer, and is incremented each time a character is read in the ambiguity mode. Whenever this counter passes through a multiple of 100_{10}, its content is printed,' along with the number of calls of the input editor which have occurred since program execution began:

```
(c CHARACTERS HAVE BEEN READ IN AMBIGUITY MODE.  INPUT EDITOR HAS BEEN
                        CALLED n TIMES)
```

where c and n are decimal numbers.

3. Normal Termination Message (INRPEXIT, 79)

Whenever a program terminates normally, the current time is printed:

```
NORMAL TERMINATION AT hhmm -ss
```

4. Abnormal Termination Messages

The running deck routines detect a variety of errors, all of which cause abnormal termination, In all cases, a system message is produced with the general form:

```
ABNORMAL TERMINATION AT hhmm -ss AFTER n CALLS OF THE INPUT EDITOR
TERMINATION DUE TO specific error message
```

where n is decimal, The following are specific system error messages:

a. Initialization Errors

... IMPROPER ORDER OF LOADING

(INITIAL, 219). A violation of the loading restrictions discussed in III.B.2.

... INADEQUATE AVAILABLE MEMORY IN BANK b

(INITIAL, 224). A negative excess quantity, as discussed in III.C.3.

b. Syntactic Errors

... ILL-FORMED INPUT STRING

(SYNTAX, 488). The string of characters produced by the input editor cannot be parsed according to the primary productions.

... AMBIGUOUS INPUT STRING

(SYNTAX, 437). The string of characters produced by the input editor can be parsed according to the primary productions in more than one way.. This error does not occur until an entire goal specifier has been parsed,

c. Storage Exhaustion Errors

... LIST STORAGE EXHAUSTION

(GARBCOLL, 400). A list storage recovery has recovered less than MINLIST words, (See III.C.4).

... LIST STORAGE EXHAUSTION BY COPYING n

(SUBGEN, 55). An attempt to create an instantiated copy of the list structure named n has used all of free list storage. The name n is an absolute octal list name, i.e., either the absolute address of a list element, or a literal name. This error will occur if an attempt is made to copy a cyclic list structure,

... BOUNDS FAULT BY INSTRUCTION AT LOCATION n (PROBABLY SYNTAX STACK EXHAUSTION)

(INRPEXIT, 63). The execution of the instruction at the 18-bit absolute octal address n has caused a bounds interrupt. Since the pushdown stack is protected by the bounds register, this error will occur if the push-down stack is exhausted, which will occur if an infinite recursion or a recursion over a very large list structure is attempted. Note that this error message is incorrect; it should indicate the probable exhaustion of the pushdown stack, not the syntax stack.

... SYNTAX STACK EXHAUSTION

(SYNTAX, 60). The syntax stack has been exhausted by an excessively deep recursion within the syntax analyzer.

... CONVERSION BUFFER EXHAUSTION BY SYNTAX ANALYZER

(SYNTAX, 164). The syntax analyzer, under the control of a character-packing special label, nas attempted to store an excessive number of characters in the conversion buffer, The number of characters in this buffer, plus the number of characters with output codes larger or equal to 75_8, must not exceed 1016_{10} for identifiers or 1024_{10} for numbers.

... CONVERSION BUFFER EXHAUSTION BY IDENT,CIDENT,DECCON,OCTCON,OR FLOATCON

(SCAN, 690).

... CONVERSION BUFFER EXHAUSTION BY IDENT OR CIDENT

(IDENT, 145). Both of these messages indicate that an identifier- or numbercreating primitive has attempted to store an excessive number of characters in the conversion buffer. The limits on the buffer size are given above.

d. Illegal Argument Errors

Within the subprograms ARITHMTC, SCAN, OUTPUT, IDENT, and MISCPRIM there are a large number of checks for illegal arguments of primitive generators, These checks, which are only assembled if the assembly control-parameter SAFE = 1 (See III.C.1), lead to the following error messages:

... ILLEGAL VALUE n ASSIGNED TO name of one or more internal variables

if the primitive is a standard setting generator for an internal variable, or

... ILLEGAL ARGUMENT n GIVEN TO name of one or more primitive generators

for all other primitives, In either case, n is the illegal argument, given as an absolute octal list name.

e. Miscellaneous Errors

... CALL OF ABEXIT FROM LOCATION n

(INRPEXIT, 86). The primitive generator ABEXIT has been called, The absolute octal address n is the biased return address given to ABEXIT by the calling generator,

... ERROR IN READING LUN n

(OUTPUT, 871). An irrecoverable parity error has occurred in reading logical unit number n (decimal), i.e., a READ CHECK BUFFER DECIMAL call of IOP. has returned with bit 19 of the A-register set. This message will be written on the system comment unit (logical unit 64) as well as the unit specified by sno.

... FAILURE OF GENERATOR CALLED BY SYNTAX ANALYZER FROM LINKAGE ADDRESS n

(SYNTAX, 188). A generator called by the syntax analyzer has failed; n is the absolute octal address of the generator linkage (in PROG) from which the generator was called.

... ILLEGAL LIST ELEMENT FOUND DURING STORAGE RECOVERY AT LOCATION n

(GARBCOLL, 97). During the search over all active list elements performed by the list storage recovery routine, a non-literal (absolute octal) list name n has been found which is not the address of a valid list element.

... ILLEGAL LIST ELEMENT FOUND DURING ANALYSIS AT LOCATION n

(ANAGEN, 23). During the execution of an analytic assignment statement, a non-literal (absolute octal) list name n has been found in the template list structure which is not the address of a valid list element.

... ILLEGAL INSTRUCTION AT LOCATION n

(INRPEXIT, 50). An illegal instruction at the 18-bit absolute octal address n has been executed. A few illegal instructions appear in the running deck at program points which should not be reached except under extraordinary circumstances,

... UNEXPECTED INTERRUPT

(INRPEXIT, 71). This message occurs (if the assembly control parameter VERSCOPE = 6, see III.C.2) for all abnormal terminations which do not produce other system messages. The most common cause is exceeding a time or print limit set by SCOPE,,

5. Dump Message (DUMP, 8)

The dump primitives have not been implemented in COGENT 1.2. A call of the primitives DUMPV, DUMP1, or DUMPALL will cause the system message

(DUMP GENERATOR-CALLED FROM LOC r WITH ARG n. SORRY, BUT THESE GENERATORS ARE NOT CODED YET)

to be written on the unit specified by sno. The absolute octal numbers n and r are the argument of the dump generator (if any) and the biased return address given to the dump generator by its calling generator, The dump generator will return the dummy element.

B. Compiler Error Messages

Compiler error messages are produced by the COGENT compiler, rather than by the running deck. Unlike system error messages, they do not cause abnormal termination, so that a single compilation may produce several error messages. These messages appear in the printed output on logical unit 61, with the standard format

```
****************ERROR TYPE k ERROR COURT IS n*****....
```

followed by enough asterisks to fill out the print line. The numeral k indicates the type of error, while n is the ordinal of the appearance of the error message in the compiler output.

At the end of the compiler output, a message will appear giving the total number of error messages. Since the compiler will give a normal termination even when one or more compiler error messages occur, it is advisable to check this final error count before running a compiled program. It is also advisable to check for error messages in the subsequent assembly of a compiled program. Certain types of errors, such as undefined or multiply-defined statement numbers, will not be detected by the compiler but will lead to assembly errors.

The following is a list of all- error types by their numerals.. For each type a reference is made to the generator or generators in the COGENT compiler (as written in COGENT, i.e., file II of the master tape) which detect the error. Some error types, designated as <u>unusual</u>, indicate that the compiler is behaving in an unforseen manner, either because of faulty design of the compiler or because the input program Contains some error for which specific checks have not been-built into the compiler.

1. (SAL on EXEC). The compiler 'has attempted to compile code to store into, or call as a generator, some quantity denoted by a compound expression (unusual).

2. (MOVE). Either (i) a pseudo-constant appears in a position in which only a variable is allowed (i.e., a position indicating assignment to the variable). or (ii) a pseudo-constant which does not denote a generator element appears in a position indicating a generator to be called.

3. (GENEND). Either (i) a generator is defined with a name which has not been declared (even implicitly) as a generator name, or (ii) more than one generator has been defined with same name (and under the same declaration).

4. (PCON). A name which is not a pseudo-constant has been used in a position where only a pseudo-constant is allowed.

5. (CANA). An analysis string contains more than 50 items.

6. (PARSCON). A constant has been encountered which cannot be parsed according to the total set of productions.

7. (PARSCON). A constant has been encountered which has more than one parsing according to the total set of productions. The compiler will make an arbitrary choice of the parsing to be used.

8. Not used,

9. (TRANCON). In parsing a constant, a $NOP/ special label has been found on a production with more than one phrase class name in its construction string, The compiler will disregard the $NOP/ special label,

10. Not used,

11, (SETCON or CONDEF1). Within some constant, a parameter has been found whose index is zero or larger than 50. The index will be replaced by 1. This error will be found when code is generated for the constant, rather than when the constant is actually read,,

12. (OBSETA). A left-hand constant in an identifier declaration does not denote an identifier element, The erroneous item in the declaration will be ignored,

13. (PROGDEC). A local declaration has appeared in the main declaration sequence.

14. (IDENTDEF). An identifier element within some constant list structure contains too many characters. This error will be found when code is generated for the constant, rather than when the constant is actually read.

15. (UHDEF). The entity on the left-hand side of a character definition is not an object character representative. Tne definition will be ignored.

16. (RECFLOW). A phrase class mentioned in the primary syntax description is empty, i. e,, the primary productions give no method for constructing any phrase of the class. This message is a warning rather than an error; the compiler will generate a syntax analyzer which will never recognize any phrase of the empty class.

Each empty phrase class gives rise to two error messages.: a type 16 message when the flow graph of the recognizer for the phrase class is being calculated, and a type 18 message when code for the recognizer is actually generated. The location of the type 16 message does not indicate the name of the corresponding empty phrase class, but each type 18 message will appear immediately before the compiled recognizer for the empty phrase class.

17. (CHRVEC). A member of a terminator sequence is not an object character representative.,

18. (RECCOMP). A phrase class mentioned in the primary syntax description is empty. This message is a warning rather than an error; See 16.

19. **(PRODCOMP).** A primary production with a $NOP/ special label nas more than one phrase class name in its construction string.

20. (GOALCOMP). A goal in a goal specifier is not a phrase class name.

21. **(PROGBEG).** The total set of productions, after the reduction of compound to simple productions, contains more than 1024_{10} productions.

22. **(TRANCON).** In parsing a constant, a parameter has been found in a portion of-an object string which is to be converted into a number or identifier element, i.e., which is parsed under the control of a character-packing special label. This message is a warning rather than an error; the parameter will be ignored in creating the number of identifier.

23. **(PROGOUTZ).** The compiler-has attempted to output an instruction which violates the format of COMPASS cards (unusual).

2.4. **(PDCNTABG).** While generating the production code number table or the character scanning table, the compiler has been unable to find any production with a given code number (unusual).

25. **(GENLINK).** A production label contains an improperly declared name, i.e., a name which is neither a generator name nor a universal own variable. This message appears when the corresponding generator linkage code is generated, not when the erroneous production is read.

26. **(CONDEF).** An inconsistency has been found while generating code for a constant (unusual).

27. **(OUTCODE).** An ill-formed constant of the form $$< object character representative > has been encountered,

28. **(OBIDENT).** A parameter has been found in an identifier object string.

29. **(CLANA).** In an analysis statement whose template expression is a constant, an analysis item appears which is not matched by any parameter in the template.