EDWARD FREDKIN
AUGUST 1961

PRELIMINARY DECAL MANUAL PDP-N-1

INTRODUCTION

The Digital Equipment Corporation Compiler, Assembler and Linking Loader program for PDP-1 is called DECAL. DECAL has complete assembly program facilities.

An important feature of DECAL is its function as an algebraic compiler. This means that the appearance of the algebraic statement:

```
a <=  b + c
```

in a program, has the same effect as writing

```
lac b     ...load the accumulator with contents of c
add c     ...add to accumulator the contents of b
dac a     ...replace contents of a with c + b
```

in assembly language. This provides added brevity and convenience to the programmer in writing his programs, especially for complicated algebraic expressions, where a symbolism may be used which corresponds directly to the mathematical notation used in writing formulas.

Another important feature of the DECAL system is the Linking Loader which provides for:

a) Relocation of binary programs: The origin of any program is determined at read-in time.

b) Symbolic cross-reference between programs: This enables programs to call on other programs which may be compiled separately, or exist as library tapes. References to such programs are punched out in their symbolic form during assembly by DECAL. These are replaced by their values at read-in time.

Several notes are in order:

(i) This preliminary manual should be taken as referring solely to the DECAL F tape.

(ii) This is the first draft of a manual for DECAL which is under development, but already useful as described below. It is hoped that this will provide adequate interim information for programmers who wish to use the current version of DECAL, (DECAL F tape).

(iii) Certain exceptions and intricacies of the

system have undoubtedly been overlooked or under-emphasized. Please inform us of any difficulties.

FUTURE FEATURES

DECAL will progress through various phases, the currently completed being Phase I (or DECAL F tape). The progression of phases will be, more or less, as follows:

Phase II

Phase II will include a description of the internal DECAL symbols and subroutines of practical general use.

Phase III

Phase III will allow subscripts and indexing. Subscripts will be able to be multi-dimensional and arithmetic expressions. The subscripts will not be allowed to contain subscripted variables.

Floating point arithmetic will be completely incorporated in arithmetic expressions.

Phase IV

Four Linking Loaders will be made available: Low Linking Loader (LLL), High Linking Loader (HLL), Compact Low Linking Loader (CLLL), and Compact High Linking Loader (CHLL).

Phase V

This final phase will allow for the use of published algorithms in the ALGOL language. Other features include:

1. Complete address arithmetic facilities.
2. Completely general subscripting and indexing facilities.
3. Automatic constants assignments. The constants will be available to the LL so that all constants of all programs are assigned to a minimum number of registers.

DECAL PHASE I

COMPILER CHARACTERISTICS

A. Input

Input to the compiler consists of a symbolic tape. In the following, a "program" is normally just the contents of one tape. A "system" consists of a number of programs. A "program" consists of a series of "statements", which are in turn composed of a series of "symbols". DECAL may be thought of as operating on a string of symbols provided by a particular tape. Normally a statement will occupy just one line (see B); while symbols appear between blanks on the line (symbols; below). In other words, statements are normally delimited by "carriage returns", and symbols by "spaces" or "tabs"; but there are many exceptions to this which give more power to the system. The exact specifications are enumerated below

Format: Spaces and tabs may be used in any desired number between symbols. Extra carriage returns (giving blank lines) are permitted.

Suggested Format: Type statements one to a line. Start with the label symbol (if any), then "tab" and type the rest of the statement. Comments may also be typed on a line.

Characters: Input is a string of legal eight-bit characters as listed in PDP-1 Manual, F-15A. Tape feed, blank tape, and error code punches are ignored. Non-printing characters are represented below as follows:

Space	
Tab	
Carriage Return	
Upper Case	
Lower Case	
Black	
Red	
Back Space	

Character Classes: Characters are divided into several classes for the purpose of defining symbols. Case shifts are remembered (but are filtered out) and are used to distinguish upper and lower case characters. At the start of a compilation,  and  are assumed.

Class 0: Illegal codes (e.g., codes 12-17, 32, 37, 52, 53, 60, 76)
Class 1: Δ ↦
Class 2:)] ([→ '(single quote)]
Class 3: 0123456789
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
? ¯ (overstrike)
Class 4: ~ ↓ ∨ ∧ < > ↑
Class 5: ,.• (center dot)
Class 6: " (double quotes)

Symbols: Formed from one or more consecutive characters as follows:

Class 0 characters halt the compiler when read from tape.

Class 1 characters delimit any symbol but are otherwise ignored.

Class 2 characters are single character symbols, having a particular significance in the DECAL system.

Class 3 characters are those normally used by the programmer to form location symbols, constants, numbers, etc. in instruction statements, and to form the variables of algebraic symbols already defined in DECAL are either standard ALGOL connectives, or consist of three characters. Consequently, programmers should avoid the use of three-character symbols for their program variables.

Class 4 characters are normally used to form symbols which are the operators in algebraic statements (instruction generators). Some Class 4 characters have a special significance to DECAL when appearing in the address part of an instruction statement (e.g., ' + -). The use of Class 4 symbols as operators eliminates the necessity of spacing between operator and operand, as long as the operands are of Class 3.

Class 5 characters are used to form some symbols of special significance to DECAL. Other Class 5 symbols are available to the programmer, but

are not normally used.

Class 6 the character " has a special use, when it is desired to form symbols of mixed class (see (i) below).

Normalization: Symbols are normalized with respect to case by filtering out all case shifts from the input string, supplying case shifts only as needed.

Thus, a symbol is any string of characters that satisfies the following conditions:

Either

The characters immediately before and after the string are " (the Class 6 character, which we call a symbol bracket), and the string does not contain a "

Or

ii) The string consists of just one Class 2 character

Or

iii) The string consists of characters of like class (Classes 3, 4, or 5), and the characters immediately before and after the string are of a different class.

E.g., consider the following strings of characters on tape:

(i) symbol␣⇑ A⇓a⇑A⇓a⇒( )↓

(ii) s␣t␣a␣t␣e␣m␣e␣n␣t↓

z•, z•, z•,

(iii)⇑+ A␣⇓

(iv) "⇒ß⇑↑⇓ .a"

These break down into symbols

(i) symbol
AaAaAa
(
)
↓
(5 symbols)

(ii)
s	e	z
t	n	;
a	t	z
t	↓	;
e	z	z
m	;	;

(18 symbols)

(iii) +
A
(2 symbols)

(iv) ⇨ᛋ⇳↑⇵. a
(1 symbol)

Integer Number: A symbol beginning with a numeral (0, 1, . . . 9) is assumed to be an Integer Number. An Integer Number is a sequence of consecutive numerals terminated with a non-numeric character. Integers are always octal, except when the Data Word dec is used.

Symbol Types: The class of characters which comprises a symbol has no effect upon its type, although, by convention, different classes are commonly used for different type symbols, e.g., bs ps ss wd as oc, Type 3; ig, Type 4. The symbol type is determined either by previous definition in the DECAL system, or by its context in program. Any symbol appearing in program is assigned to a particular type when read from tape. Except for symbols of types un and ss, every symbol has a numerical value. The type and value of a symbol are entered by DECAL in the DECAL symbol table upon definition. The various symbol types and the significances of their values follow:

Action operator (ao): The value of an ao is the address of a subroutine in the DECAL system. Whenever an ao symbol is encountered in program, its associated subroutine is called. The assignment of types and values to symbols (except un) is done by ao's. A symbol is assigned to type ao by the ao dao (See L).

Instruction Generator (ig): Used as an operator in algebraic statements. The value of an ig is the address of the first of a series of "pattern words" in memory. When an ig is encountered, the pattern words are combined with the values of the appropriate operators to assemble a series of instructions. Assignment is done by the ao dig (See M).

Constants (wd, as, oc): These are used in the formation of the output Word on the LLD tape. These types must be distinguished from the types bs and ps below principally for the sake of the LLD relocating feature. The value of a constant is an 18-bit quantity.

If the left 6 bits are zeroes the type is as (address-size). If the right 6 bits are zeroes, the type is oc (order-code). Otherwise, the type is wd (word). Assignment is done by the ao's ewd, eas, eoc (See N).*

Program Symbol (ps): The value of a ps is the relative address of an instruction in program. Type ps is distinguished from type as above, so that the LLD may adjust the contents of registers in which a ps was used in assembly. Assignment is done by the ao :. (center dot; period; period). A ps symbol is expunged by the ao fin only.

Block Symbol (bs): Used exactly the same as ps, only a bs may also be expunged by the ao blk. This enables the programmer to divide his programs into blocks, and use bs symbols independently in each. Assignment is done by the ao : (center dot; period).

Unassigned Symbol (un): A bs or ps may be used before it is assigned. At the time of its use it will be assigned to type un, and reassigned on the appropriate ao. In connection with certain ao's, it is necessary to use ps and bs symbols only. At the end of any compilation, there should be no un symbols remaining. (Note: References below to un symbols refer to occurrences of symbols that are un at the time of reading.)

System Symbol (ss): In a system, it may be necessary to refer to bs and ps symbols defined only in other programs of the system. Such a symbol, which refers to another program compiled separately, is of type ss. Assignment is done by the ao dss. A symbol may be used by other programs as an ss only if the ao ' (single quote) is used. This involves the "linking" feature of the LLD. At load time ss symbols are evaluated as the locations where ' was used, and are substituted at the appropriate points in other programs of the system.

Numbers: Numbers have immediately available values (their octal values), hence are not included in the symbol table.

* In DECAL II, the distinctions between wd, as, oc will be waived.

Data: In connection with the ao's bci and dec, any character string has an immediately available value, which is a function of the CONCISE III representations of the individual characters.

B. Simple Statements

Simple statements are separated by a ; or a ↵. (Empty statements are permitted). Simple statements are of one of the three following types. A simple statement is assumed to be an algebraic statement unless it starts with a symbol which identifies it as of another type.

Simple Algebraic Statement	Results in a sequence of instructions in object program.
Simple Instruction Word Statement	Introduced by a symbol of the type wd, as, or oc. Results in one instruction in object program.
Data Word Statement	Introduced by an ao.

C. Algebraic Statements

An algebraic Statement consists of a string of operators, operands, and parentheses. The operators are ig symbols.

e.g., go to $\Rightarrow, \Leftarrow, \equiv, \supset, \wedge, \vee, \sim, =, \neq, >, <, \leq, \geq$
+, -, x, /. (See section Q)

Operands are bs, ps, un, or ss symbols (but not numbers). For each operator, a level is defined (lv0, lv1 . . . . . .lv7). For the ig's now in DECAL these are:

lv0: goto $\Rightarrow \Leftarrow$
lv1: $\supset \equiv$
lv2: $\wedge \vee$
lv3: $\sim$
lv4: $= \neq > < \leq \geq$
lv5: + -
lv6: x/mpy dvd

If no parentheses are present, instructions for the higher order operators of the string are compiled first, and, for expressions at the same level, the leftmost is compiled first.

e.g., $a>b \vee \sim b<c$ goto d

may be written $((a>b) \vee (\sim(b<c)))$ goto d and produces the same program as the sequence of instruction statements.

```
lac a          ...calling sequence for >
jda >
lac b
```

```
                jda >           ...(b < c is equivalent to c > b)
                xct ~           ...does cma
                ior al          ...(a > b) V ~ (b < c)
                xct goto        ...does spa
                jmp d
                .
                .
                .
                .
                .
al:             loc             ...a > b
```

Temporary storage register (e.g., al, in the above example) are automatically supplied at the end of the current program.

The presence of parentheses indicates that the parenthesized expression is to be compiled first, before proceeding with the rest of the statement. Parentheses may be used to any required depth. It should be noted that the operators currently available in DECAL correspond in both notation and significance to those of ALGOL. Many features of DECAL (e.g., the : to indicate a location) have been included with a view to the ultimate incorporation of ALGOL within the system.

D. Simple Instruction Word Statements

i) A simple instruction word statement must be introduced by a constant symbol (wd, as, or oc), and consist of a string of symbols of types wd, as, oc, bs, ps, ss, un, or numbers.

ii) If no ss or un symbols are involved, an instruction word is assembled, which is the inclusive "or" of the values of the symbols. If an odd number of bs or ps symbols are used, the output tape is coded to indicate that the instruction word is "relocatable", and its address part will be corrected at "load time" according to the "relocation constant". This deals with the simple cases of address arithmetic in which (a - b) should not be relocated, while (a - b + c) is relocatable.

iii) If an ss or un symbol is used, only the first 6 bits of the instruction word are assembled. The output tape is coded to indicate that the address part is to be filled in by the LLD. If the symbol is un, then the LLD expects to pick up its value at another location on the same tape.

If the symbol is ss, the LLD gets its value from some other tape loaded at the same time. Consequently, the only other symbols that should appear with ss or un symbols are oc or numbers, with the right 12 bits zero.

E. Defined Expression (DE)

A number os ao's to be described make use of a "Defined Expression", which consists of octal numbers and/or symbols whose value is known to DECAL at that point in the compilation, i.e., wd, as, oc, bs, ps symbols and +, - signs, but not un, ss. That is, the symbols appearing in a Defined Expression must have been previously defined in the program. A DE is terminated by a) or ;. In the absence of + or - signs, a logical "or" is taken of the values (as in the evaluation of the address parts of instruction statements).

F. Action Operators in Instruction Word Statements

i) Most of the PDP instructions are represented in DECAL as wd or oc constants. Some instructions, however, have been given special treatment:

a) szs is defined as an ao, which plants the value of the succeeding defined expression (DE) in a skip instruction: i.e., szs S produces 6400S0, and has the action "skip" on zero sense switch "S".

b) All "rotates" and "shifts" are defined as ao's, which plant a number of "ones" equal to the value of the succeeding defined expression (DE) in the appropriate "shift" or "rotate" instruction: e.g., rcl 9 produces 663777 and has the action "rotate combined left 9 bits".

The above may not be preceded by any other part of the instruction word statement.
e.g., szf 1 szs 2 produces a diagnostic print.

ii) The symbol . in the address part of a statement is treated as a bs whose value is the current value of the location counter (the relative address of the instruction)

iii) The symbols +, - in the address part of a statement change the mode of combination of the succeeding symbol into the rest of the statement to an "add" or "subtract", respectively. Address arithmetic cannot be done with un or ss symbols.

iv) The symbol / in the address part of a statement adds a defer bit 10000 to the resulting word.

v) For sequence-break instructions, the following ao's are used:

chn appearing in the address part of an instruction word plants the value of the DE that follows at the appropriate point in the instruction, e.g.,

```
isb chn 17 produces 721752
dsc chn 6 produces 720650
```

bac, bio, bpc, bjm are used in the address part when referring to the sequence-break storage locations for accumulator, in-out register, program counter, and the sequence-break jump respectively. e.g.,

```
jmp  'bpc 7 produces 610035
(35 is the program counter storage location for
channel 7)
```

chn, bac, bio, bpc, and bjm, may not be the first symbol in a statement.

G. Labels, Location Symbols, and Variables

Algebraic, Instruction Word, and Data Statements may be labeled by a bs or ps. Such labels are actually location symbols, in that the symbol has associated with it the corresponding location of the (initial) instruction of the labeled statement. If the purpose of the statement so labeled is to provide a storage register(s) for a variable, then the labeling symbol may be used as a variable (operand) in an algebraic statement appearing in the program.

The labeling is accomplished by punching the desired symbol at the beginning of a statement followed by a : for bs or a :. for a ps. The insertion of a ' causes the preceding symbol to be transmitted as a symbol defined by the program, and available as an ss to other programs of the system. The ' must be followed by a : or :. For example:

```
name:  a + ct = d
xyz:.  jmp art
abc':  b + c = a
abd':. jmp X
```

Any number of labels may be applied to a statement. Labels are neglected when classifying a statement as "algebraic", "instruction word", etc.

H. Data Word Statements

Octal: oct followed by an octal integer of not more than six digits. Initial zeroes may be omitted. (oct is an as with value 0)

Alphanumeric: bci (binary-coded-information) followed by one separating character (usually a Δ or .) followed by a string of characters terminated by a .. The resulting data is packed three characters per word with zeroes filling out the last word if necessary

Any character may be included in the string, although special handling of the ' and . is required. To include either a . or ' in the resulting data, it is necessary to precede each instance by a '.

Decimal: dec followed by a decimal number of the form:

±123.456 ±789

or of the form:

±123

The first case will result in a two-register floating point number and the second in a one-register integer. The plus signs are optional, as is every other part in the floating-point case which is not essential to specify the value, or to distinguish it from the integer case. If the decimal point is omitted from the first case, it is assumed to lie to the right of the number.

J. Statement Parentheses

For certain ao's it is necessary to generalize the definition of a "statement". A compound statement either

i) is a simple statement, as already defined (delimited by ; or ↵).

or

ii) lies between two occurrences of the statement parentheses beg and end. Compound statements may contain other compound statements, so intermediate occurrences of beg and end must count out in the usual fashion for parentheses.

K. Tape Terminators

Tapes are terminated by the ao's stp or fin. The former permits the read-in of several physically separate tapes which comprise a single program. The CONTINUE button will read-in the next tape. The latter finishes the compilation process, and expunges all symbols defined during the program from the symbol table, unless fix was used. In order that the symbols stp and fin be recognized, they must be delimited by a terminal character. Any character other than one in Class 3 will suffice, but period (.) is suggested. The period may be followed by a ↵ to restore the carriage.

L. The Action Operator dao

dao defines the preceding symbol as a new ao by the compound statement which follows it. When the dao is encountered, the defining statement is assembled directly into the compiler storage area in DECAL. The defining statement, normally using beg and end, must be capable of a complete one-pass compilation, so may contain no un or ss symbols. The compiled program is called on every subsequent occurrence of the new ao symbol. Return to the main program is achieved through the MAC return rml. An example of the use of a dao is as follows:

```
X dao
   jmp rml
```

(may be used to make the symbol X irrelevant). A complete description of dao will be given in a later phase.

M. The Action Operator dig

dig defines the preceding symbol as a new instruction generator by the statement which follows it. The action of dig is precisely the same as that of dao, except that the new symbol is now defined as an ig in the symbol table. The associated sequence of registers in the compiler storage area will now be decoded every time the new symbol is encountered in an algebraic statement. The ig operator may have either one or

two operands. For one operand, the operand occurs after the ig symbol. For two operands, these occur separated by the ig symbol. They are referred to in the encoded registers by the numbers 1, 2, respectively, in the address part. Other codes are

i) ths (as 3000); when encountered in decoding, results in the address part of the appropriate word being treated as an ss, where the appropriate symbol is the labeling symbol for the ig. At load time, the LLD will expect a tape on which the labeling symbol is defined as the location of a subroutine, or an instruction, to be executed by the generated program.

ii) lst (as 400); indicates the last register in the encoded sequence.

The first register of the sequence contains some necessary information for the compiler, and does not result in a register of decoded program. The codes for the first register are:

iii) lv0, lv1 . . . . . lv7 (oc 04 - 14 in bits 0 - 3); indicate the level of precedence of the ig (see C).

iv) op1, op2 (as 100, 200); the number of operands expected.

v) rs1 (as 10); number of results (currently no choice).

vi) cmt (as 2000); indicates that the two operands (op2) are interchangeable.

vii) nlc (as 1); indicates that the first generated instruction word is not lac 1; in the absence of nlc, a lac 1 is inserted without being specified in the encoded program.

Examples:
```
mpy dig
beg lv6 op1 rs1 nlc
jda ths
lac 1 lst end
```
(a tape with mpy must be supplied at load time).

```
= dig
  beg
  lv1 op2 rs1 cmt
  xct ths
. xor 2 lst end
```
(a tape with =': cma must be supplied).

The instruction generator is restricted as follows:

i) The full address part is used in decoding the instructions, the only acceptable constants must occupy bits 0-6, thus cla, "law 0", etc., may not be used (hence, the circumlocution is ≡ above).

ii) The only acceptable references from the address part are to the operands, or to just one associated location.

iii) It is not possible to use ao's to control the compilation process, or to use "nested" iq's.)

N. Other Action Operators

blk introduces a new block. If SW 2 is ON, the bs's previously defined will be printed. If SW 3 is ON, the un's will be printed. All bs's are expunged from the symbol table.

dss declares as type ss the symbols which follow in the same statement. (i.e., up to ; or ↵)

*eas equates the preceding symbol as an as to the DE which follows in the same statement.

*eoc equates for oc, as above.

ewd equates for a wd, as above.

fix fixes the symbol table for subsequent compilations; prints current bs and un.

lve leaves a gap in the program of length given by the DE which follows in the same statement.

xsy expunges from the symbol table the symbols which follow in the same statement. (i.e., up to ; or ↵)

... the remainder of statement is comment material and is therefore ignored.

O. List of Constants Supplied

Symbol	Type	Value	Operation
add Y	oc	400000	C(AC) ⇐ C(Y) + C(AC)

*This feature will be eliminated in future tapes. ewd may be used instead.

Symbol	Type	Value	Operation
and Y	oc	20000	C(AC) ⇐ C(Y)∧C(AC)
cal Y	oc	160000	jda 100
dac Y	oc	240000	C(Y) ⇐ C(AC)
dap Y	oc	260000	$C_{6-17}(Y) \Leftarrow C_{6-17}(AC)$
dio Y	oc	320000	C(Y) ⇐ C(IO)
dip Y	oc	300000	$C_{0-5}(Y) \Leftarrow C_{0-5}(AC)$
dis Y	oc	560000	divide step
dzm Y	oc	340000	C(Y) ⇐ 0
idx Y	oc	440000	C(Y) ⇐ C(AC) ⇐ C(Y) +1
ior Y	oc	40000	C(AC) ⇐ C(Y)∨C(AC)
iot	oc	720000	in-out transfer group
isp Y	oc	460000	idx Y; spa
jda Y	oc	170000	dac Y; jsp Y + 1
jmp Y	oc	600000	C(PRC) ⇐ Y
jsp Y	oc	620000	lap ; jmp Y
lac Y	oc	200000	C(AC) ⇐ C(Y)
law Y	oc	700000	C(AC) ⇐ Y
lio Y	oc	220000	C(IO) ⇐ C(Y)
mus Y	oc	540000	multiply step
nop	oc	660000	no operation
opr	oc	760000	operator group
sad Y	oc	500000	if C(AC) ≠ C(Y) then skip next instruction

Symbol	Type	Value	Operation
sas Y	oc	520000	if C(AC) = C(Y) then skip
sft	oc	660000	shift group
skp	oc	640000	skip group
sub Y	oc	420000	C(AC) ⇐ C(AC) - C(Y)
xct Y	oc	100000	execute instruction in Y
xor Y	oc	60000	C(AC) ⇐ ∼(C(AC) ≡ C(Y))
skip group:			
sma	oc	640400	if B_0 (AC) = 1 then skip
spa	oc	640200	if B_0 (AC) = 0 then skip
spi	oc	642000	if B_0 (IO) = 0 then skip
sza	oc	640100	if C(AC) = 0 then skip
szf f	oc	640000	if Flag (f) = 0 then skip
szo (szs see F (i))	oc	641000	if Overflow Ind = 0 then skip
usk	oc	640600	skip (unconditional)
operate group:			
cla	oc	760200	C(AC) ⇐ 0
clf f	oc	760000	Flag (f) ⇐ 0
cli	oc	764000	C(IO) ⇐ 0
cma	oc	761000	C(AC) ⇐ ∼ C(AC)
hlt	oc	760400	halt
lap	oc	760300	C(AC) ⇐ C(PRC)
lat	oc	762200	C(AC) ⇐ C(test word)
stf f	wd	760010	Flag (f) ⇐ 1

Symbol	Type	Value	Operation
in-out transfer group:			
asc	wd	720051	activate sequence-break channel
cnv	wd	720040	convert, analog to digital
dpy	wd	720007	display (C(AC), C(IO))
dsc	wd	720050	deactivate sequence-break channel
esm	wd	720055	enter sequence-break mode
isb	wd	720052	initiate sequence break
lsm	wd	720054	leave sequence-break mode
ppa	wd	720005	punch paper tape, alphanumeric
ppb	wd	720006	punch paper tape, binary
rcb	wd	720031	read converter buffer
rpa	wd	720001	read paper tape, alphanumeric
rpb	wd	720002	read paper tape, binary
rrb	wd	720030	read relay buffer
srb	wd	720021	set relay buffer
tyi	wd	720004	type in
tyo	wd	720002	type out
(see also ao's chn, bac, bio, bpr, bjm, section F(v))			
shift-rotate group (see F (i)):			
other program constants:			
loc	as	0	
oct	as	0	
..	as	0	
rml	as	124	return to MAC from a called subroutine

Symbol	Type	Value	Operation
constants for dig statements			
lst	as	400	last pattern word
ths	as	3000	"this" symbol
cmt	as	2000	commutative operator
lv0	oc	200000)	
lv1	oc	240000)	
lv2	oc	300000)	
lv3	oc	340000)	precedence levels
lv4	oc	400000)	
lv5	oc	440000)	
lv6	oc	500000)	
lv7	oc	540000)	
nlc	as	1	not lac 1 as first pattern word
op1	as	100	one operand
op2	as	200	two operands
rs1	as	10	one result

P. Error Detection

DECAL detects and prints about 30 types of errors. The format of the print in each case is a three-character code in red followed by the last defined symbol (or NS if there is none) followed by the current symbol. For the cases ich and fpe, the offending character is printed as a three-digit octal integer.

Code	Error	Action
bos	bracket or separator (where it shouldn't be)	ignore

Code	Error	Action
cas	compiler algebraic statement in dig or dao (not permitted).	gives a try anyhow
ciw	compiler improper word	store anyhow
dda	duplicate definition attempted	do not re-define
dds	duplicate definition of symbol attempted by dss	do not re-define
fpe	parity error (this error stop is presently deactivated)	halt; load desire 6 bit character in test word (right-justified) and continue
iaa	illegal address arithmetic	try anyhow
ias	instruction word in algebraic statement	handle as instruction word
ich	illegal character	treat as space
idn	improper decimal number	proceed with conversion
iig	incorrect ig	ignore ig
ino	incorrect number of operands	gives a try anyhow
nas	number not allowed in algebraic statement	treat as oct
nds	number in dss	ignore number
nfs	not first symbol in statement (when defining)	proceed with definition
nxs	number in xsy	ignore number
sdi	symbol definition indefinite (expression contains undefined symbol)	define as zero
spu	symbol previously used when definition attempted	do not define

Code	Error	Action
tmd	too many digits in octal numbers	ignore left digit
tmo	too many operands	new operand replaces previous
tmr	too many results	ignore excess
xcs	exceeded compiler storage	halt; continue with store over last word
xmp	exceeded MAC push-down list	halt; no recovery
xps	exceeded MAC protected storage (commonly occurs with incorrectly written instruction statement	halt; no recovery
xst	exceeded symbol table	halt; no recovery
X)	right paren without left	ignore paren
(X	right paren missing	treat current symbol as right paren
8,9	8 or 9 in octal number	take mod 8
,	comma found out of place	ignore comma

Q. Available ig's

Appearances of the left hand expressions in program cause the same output as the sequence of instruction statements on the right.

```
a+b             lac a
                add b

a-b             lac a
                sub b

a⇒b             lac a     (⇒ is (equal sign; greater-than sign)
                dac b
```

axb	lac a	(integer multiply)*
	jda imp	
	lac b	
a/b	lac a	(integer divide)*
	jda idv	
	lac b	
	loc idv	(halts on error; address points to routine in which error occurs)
mpy a	jda mpy	(fractional multiply)*
	lac a	
dvd a	jda dvd	(fractional divide)*
	lac a	
	loc dvd	(halts on error; address points to routine in which error occurs)
a ≤ b	lac b	(≤ is ("less-than" sign; equal sign))
	dac a	
b goto a	lac b	
	spa	
	jmp a	
a≡b	lac a	(≡ is (understrike; equal sign)
	xct ~	*
	xor b	
a≥b	lac a	
	xct ~	*

*Calling sequence for associated subroutines. If these are used, the appropriate tape must be loaded at run time (supplied with DECAL).

```
                ior b
a∧b             lac a
                and b
a∨b             lac a
                ior b
~a              lac a
                xct ~       *
a=b             lac a
                jda =       *
                sas b
a≠b             lac a       (≠ is (vertical bar; equal sign))
                jda =       *
                sad b
a>b             lac a
                jda >       *
                lac b
a<b             lac b
                jda <       *
                lac a
a≥b             lac a       (≥ is (understrike; greater-than sign))
                jda ≥       *
                lac b
a≤b             lac b       (≤ is (understrike; less-than sign))
                jda ≤       *
                lac a
```

R. Available Action Operator

For use in place of oc symbols:

Symbol	Associated Value	Significance
ral	661000	Rotate AC left
rar	671000	Rotate AC right
rcl	663000	Rotate combined left
rcr	673000	Rotate combined right
ril	662000	Rotate IO left
rir	672000	Rotate IO right
sal	665000	Shift AC left
sar	675000	Shift AC right
scl	667000	Shift combined left
sar	677000	Shift combined right
sil	666000	Shift IO left
sir	676000	Shift IO right
szs (See F(i))	640000	Skip on zero sense-switch

For use in place of as symbols:

Symbol	Significance
chn	Channel number
bac	Sequence-break storage for AC
bpc	Sequence-break storage for PRC
bio	Sequence-break storage for IO
bjm (See F(v))	Sequence-break jump

For use in address part of instruction word statements; (these do not have these significances elsewhere).

Symbol	Significance
/	Defer bit 10000 (F(iv))
→	Location counter (F(ii))
+	Change composition to "add" (F(iii))
-	Change composition to "subtract" (F(iii))

For use in algebraic statements:

if	None (a dummy, for purposes of ALGOL format)
then	None

For use in data statements:

dec	Decimal convert
bci (See H)	Binary coded information

Other action operators:

blk	End of block (N)
dao	Define action operator (L)
dig	Define instruction generator (M)
dss	Declare system symbols (N)
eas	Equals address-size (N)
eoc	Equals order-code (N)
ewd	Equals word (N)
fin	End of program (K)
fix	Fix symbol table (N)
lve	Leave space in program (N)

Symbol	Significance
stp	Pause; end of tape (N)
xsy	Expunge symbols (N)
:.	Location of ps (G)
:	Location of bs (G)
...	Comment (N)
/	Location of usable ss (G)

S. Operating Instructions

1. Place DECAL F tape in reader and press READ-IN.

2. Place program to be compiled in reader, turn on punch, and start at 400.

3. After first program has compiled, subsequent programs may be compiled by pressing CONTINUE or by starting at 400.

Linking Loader for DECAL, Phase I

At present, there is a low Linking Loader (LL) occupying 0000-1777. The symbol table of LL occupies 1400-1777. SA = 400. This present LL will allow relocatable loading of LL subroutine tapes (output of DECAL) in any order.

Operating Procedure:

1. Clear memory, if desired.

2. Load LL Binary tape (uses Hi P+L loader); identified as BIN Low Linking Loader F. Program stops at 100.

3. Set test address switches to 400.

4. Load reader with LL tape (output of DECAL) to be loaded. Remember that what was last punched by DECAL is first loaded by LL. I.e., place tape backwards in the reader keeping the feed holes in normal orientation with respect to the reader. Turn the reader on.

5. Press START. The tape should read-in, then come to a stop. (See Note d following for description of typeouts during use of LL).

6. Computer halts at 502. To load additional tapes, relocated to follow programs already loaded, put subsequent tapes in reader (see (4) above), and press CONTINUE. After each program is loaded, the computer will halt at 502.

7. After all tapes are loaded, put SS 6 up and press CONTINUE. If the system has been properly loaded, the typewriter will type out fin in black. This means you are all set to run your program. If there are still any undefined system symbols, the typewriter will type out in red the required system symbols. E.g., the following would be typed out all in red.

```
rq      symbol-1
rq      symbol-2
...     ...
rq      symbol-n
fin
```

After type-outs, computer again halts at 502.

8. To load missing subroutines to define the missing system symbols, just put SS 6 down, and resume procedure from step (6).

NOTES:

a. Programs are normally loaded into 2000 and up; each program being relocated to follow those already loaded. To start loading a system at some arbitrary loaction above 2000 (say 4100), set test word switches to desired address (4100), and put SS 2 up, when loading first program START at 400. After this program is loaded put SS 2 down, and all subsequent programs will be relocated to follow the first program by continuing from 502. (Press CONTINUE)

b. START at 400 causes the symbol table to be initialized. CONTINUE at 502 leaves symbol table unchanged.

c. Re: Check sum error. If there is a check sum error noted when the program has been read, the LL will type-out: CKS, and halt at 502. In order to reload the erroneously loaded program use the following procedures:

1. Reload program tape into reader.

2. Set test word switches to 415.

3. Press START.

This procedure will cause the program to be loaded in the same space it was previously loaded. However, since there may have been system symbols defined during the first loading, this second loading might cause LL to erroneously type-out dda (duplicate definition attempted). Such type-outs during a reloading after check sum error should be ignored.

d. The following is a description of the type-outs during use of LL

1. First word is not checksum. This is typed if the LL tape being loaded is not properly loaded in the reader.

2. pgm xxxx
   sst yyyy
   ll zzzz

   The above is normally the first type-out during loading of an LL tape. xxxx is the octal address of the first register of the program. yyyy is the octal address of the first temporary storage used by algebraic stateme of the program (if any). zzzz is the address of the last location used by the program including temporary storage.

3. Example of type-out normally following above.

Symbol-1	aaaa
Symbol-2	bbbb
- - - -	----
Symbol-n	nnnn
end	

Symbol-1, Symbol-2, and symbol-n represent certain symbols (ss) defined in the program by apostrophes ('). aaaa is the octal address which is the definition of symbol-1; bbbb is the octal address which is the definition of symbol-2; and nnnn is the octal address which is the definition of symbol-n. The symbol end signifies program has been read in.

4. dda eeee
   Symbol ffff

Error diagnostic type-out for duplicate definition attempted. eeee is the octal address which is the present value of the location counter; Symbol is an example of some symbol of the program identified by an apostrophe (') as a system symbol (ss). When LL sees this symbol and finds it already defined in the symbol table, it types out the above. ffff is the octal address which is the definition of the symbol found in the symbol table. LL does not change the definition in the symbol table.

5. cks gggggg hhhhhh

This is the error diagnostic type-out for checksum error. gggggg is the octal representation of the computed checksum. hhhhhh is the octal representation of the read checksum. To reload the program see instructions, not C.

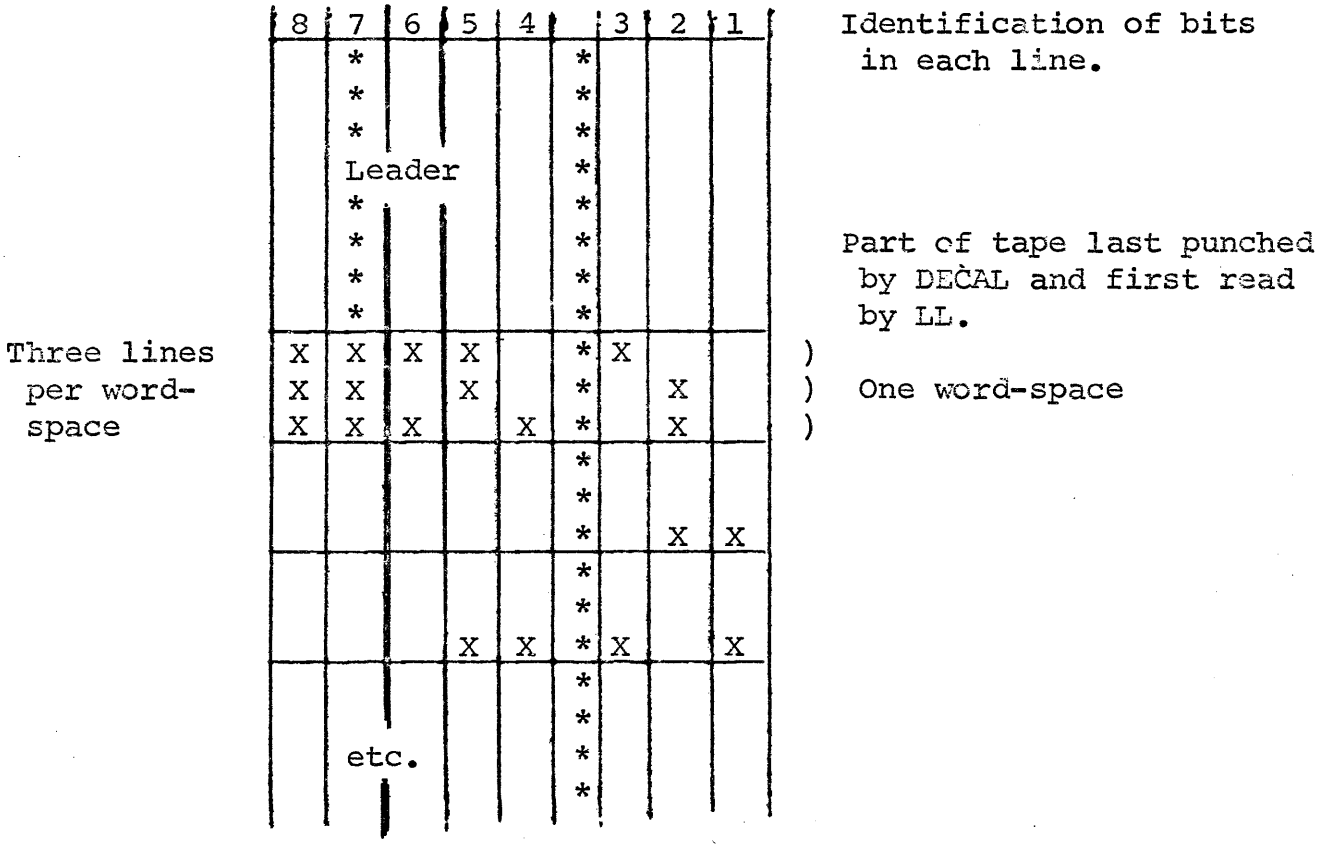
6. rq symbol (in red).

See operating procedure Step 9.

How To Read A LL Tape

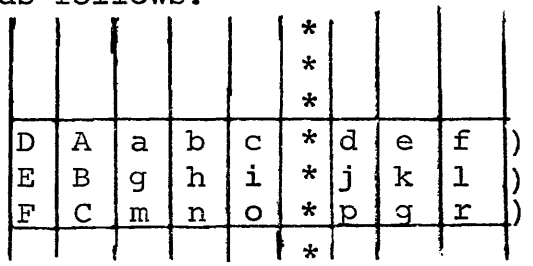
The following is intended to allow DECAL and LL users to become familiar with the operation of LL, and to be able to read LL tapes and understand how they will be loaded by LL.

The LL tapes should be held so that the last part punched (first part to be read) is put at the top. The feed holes are to be toward the right. Below is a schematic representation of a properly oriented LL tape for visual reading.



X: indicates punched information bit
*: indicates feed hole

The LL tapes consist of a sequence of words punched on the tape. That is, there are 3n lines on the tape where n is the number of words on the tape. We will now direct our attention to the arrangement of information within each word-space (3 lines) on the tape. We note that there are 8 hole spaces per line. I.e., there are 24 bits per word space. Consider the bits in a word space labeled as follows:



Now consider these bits as consisting of two parts: the Word and the Code, as follows:

Word: abcdefghijklmnopqr

Code: ABCDEF

The word is a string of 18 bits where a is the high order bit and r is the low order bit.

The Code is a string of 6 bits where A is the high order bit and F is the low order bit.

For convenience, we will represent the word as a 6 octal digit number, and the code as a 2 octal digit number.

We now present a description of the meaning of the various codes which appear on LL tapes.

Code	Meaning
00:	Normally this code means that the word sharing its word-space is to be loaded directly into memory as a non-relacatable word. This code also appears with words which consist of concise codes for 3 characters of alphanumeric information, one character per line. It also appears with the second and third words on the tape respectively: the number of words of temporary storage required by the algebraic statements in the source language programs; and the total number of words of memory (registers) required by the program, i.e., the program size, not counting temporary storage.
01:	This code identifies the word it accompanies as a relocatable word. The location of the first word in the program (the last word of program read by LL) is added to the accompanying word and the result is stored in memory.
04:	This code identifies the accompanying word as having an address which refers to temporary storage. The location of the first register following the program, i.e., the first register of temporary storage for the program, is added to the accompanying word, and the result is stored in memory.
05:	This code identifies the accompanying word as having an address which refers to a block symbol (bs) or

program symbol (ps) which was undefined at that point in the compilation of the source language program which resulted in this word. To find the address part to be daped into the word, look at the address part of the register whose location is equal to the sum of the address part of the word and the location of the first word of the program. The address part of this found register is daped into the word and the result is stored in memory. (See code 11 to understand how the appropriate address is stored into the found register).

07: This code identifies the accompanying word as having an address which refers to a system symbol (ss) identified in the source language program by a dss statement. It also indicates that the word(s) to follow are the concise codes for the alphanumeric representation of the system symbol (ss) of the address reference. (See codes 00 and 70.) The alphanumeric information is read by LL into the next available portion of memory reserved for the symbol table. The symbol is tested to determine whether or not it is already in the table, and if it is in the table, whether or not it has been defined, (i.e., that the address corresponding to this symbol has been determined. See code 10.) If the symbol is defined, the defining address is daped into the word and the resulting word is stored in memory. If the symbol is undefined or not already present in the table, a list is extended or initiated so that when the symbol becomes defined, the definition will be daped into the word, already stored in its appropriate register in memory. In the meantime, the address of the word is set to establish the appropriate list, and the result is stored in memory. If the symbol was not already in the symbol table, it is added to the symbol table.

10: This code accompanies a word which is 000000. It informs the LL that the word(s) to follow consist of concise codes for alphanumeric information. (See codes 00 and 70.) This alphanumeric information is a system symbol (ss) which appeared in the source language program with an apostrophe (') at a point corresponding to the present word-space. LL reads the alphanumeric information into the next available part of the symbol table, and checks to see if the symbol is already in the table or not. If not, the symbol is added to the symbol table and the value set equal to the LL location counter which points to the previous word stored in memory. If the symbol is already in the symbol table, its value is

set the same way, and the value is daped into all words of the system which referenced this symbol. This is made possible by the list set up at the appearances of code 07 at earlier points in the loading of the system. (See code 07.)

11: This code accompanies a word of the form 00XXXX. That is only the address part is non-zero. This word space corresponds to that point in the source language program where a block symbol (bs) is identified by colon (:) and for a program symbol is identified by a colon-period (:.). This code causes the LL to set the address part of the contents of the register, whose location is defined below, equal to the LL location counter which points to the previous word stored in memory. The definition of the location (where the location counter value is put into the address part of the register) is equal to the sum of the address part of the word and the location of the first register (origin) of the program. That is, the LL adds the address part of the word to the origin of the program being loaded to determine the location. Then the location-counter is daped into that location. (See code 05.)

12: This code accompanies a word which is 000000. This is the first thing punched on the tape by DECAL and the last thing read by LL while loading the LL tape for a program. This code causes LL to check the checksum and then halt at 502.

13: This code accompanies a word of the form 00XXXX, i.e., only the address part is non-zero. This code and word are punched by DECAL at that point in the compilation where a lve (formerly leave) statement appears. This code causes LL to update the location counter by the quantity appearing in the word.

70: This code accompanies the last word of a string of words (maybe only one word) which contain alpha-numeric information.

77: This code accompanies a word which is the checksum for the program LL tape. This is the last thing punched by DECAL and the first thing read by LL.

Summary of Operation of LL

When STARTing at 400, LL sets the origin of program to 2000, if ss2 is down, and to the contents of the test word switches if ss2 is up. When CONTINUEing at 502, LL sets the origin of program equal to previous origin, plus size of previous program, plus number of words of temporary storage of previous program, if ss2 is down. If ss2 is up, LL sets the origin of program equal to contents of test word switches. LL sets the location counter equal to the origin of current program plus size of current program.

The first word-space read by LL contains code 77 and a word which is the checksum which is the sum of all words and codes. This checksum is stored so that it can be checked at the completion of loading the program.

The second word space read by LL contains code 00 and a word of the form 00XXXX, whose address equals the number of temporary storage registers required by the program being loaded.

The third word space read by LL contains code 00 and a word of the form 00XXXX, whose address equals the size of the program. LL then reads one word space at a time and operates on it as indicated in the description of codes presented above.

When starting to load the first word of program read by LL (last word of program punched by DECAL), the location counter points to the register just past the last register to be occupied by the program (not counting temporary storage if any). Just before storing a program word in memory (Codes 00, 01, 04, 05, 07), the quantity one (1) is subtracted from the location counter, and the resulting value of the location counter points to the register into which the word is to be stored.

The last word space which LL reads contains a code 12 and a word 000000, which signifies the end of the tape.

EXAMPLE

Symbolic Program

```
...coin subroutine
...July 11, 1961
...
dss random
coin':b:        loc
                dap a
                lac b
                spa
                jmp a
                jsp random
                spa
                cma
                sub b
                sma
                idx a
a:              jmp
fin.
```

Decal Printout

```
ss coin         0000
un

bs

coin            0000
b               0000
a               0013

ps

ss

random

as


oc

wd

ao

ig

LINKING LOADER STORAGE MAP

pgm     4000
sst     4014
ll      4013
coin    4000
end
```