# ACT III COMPILER

## PRELIMINARY

## MANUAL

## CHAPTERS

I ACT LANGUAGE

II ARITHMETIC OPERATIONS

III SUBSCRIPTED VARIABLES

IV LOGIC OPERATIONS

V INPUT - OUTPUT OPERATIONS

VI STORAGE PROBLEMS

VII SOURCE - LANGUAGE SUBROUTINES

VIII DEBUGGING AT RUN _ TIME

IX STANDARD SUBROUTINES

X MACHINE OPERATION

XI THE OVERFLOW LOGIC MODIFICATIONS

Supplement - Description of the "Standard System" Tapes.

## TABLES

I EXAMPLE OF ACT - LANGUAGE CONSTANTS

II ERROR TYPES (COMPILE TIME)

III ARITHMETIC OPERATIONS

IV LOGICAL OPERATIONS

V INPUT _ OUTPUT OPERATIONS

VI FUNCTIONS

VII MISCELLANEOUS OPERATIONS

VIII MACHINE OPERATIONS

IX CODES FOR "aread"

CHAPTER I

# ACT LANGUAGE

## A. Types of words.

ACT language consists of words of five characters or less, each word followed by a conditional stop code (').

These words may represent operations, variables, constants or statement numbers.

### A.1. Constants

Positive numerical constants may be inserted directly in ACT-language in any of three forms, as follows:

a) Integers of five digits or less. These occupy one word.
b) Integers of more than five digits. These occupy two words; the first must consist of a + followed by one to four digits, and the second word must be one to five numeric digits. The largest integer the system can handle is 536,870,911.
c) Floating point constants occupy four words. If we write any number y in the form

$$y = a \times 10^{n},$$

where $0.1 \leq a \leq 1.0$
and $-32 \leq n \leq 31$

then this constant in ACT language is represented as follows. The first word consists of a decimal point followed by the first one to four digits of a. The second word contains the remaining digits of a (0 - 5). The third word is "e" if $n \geq 0$. The fourth word contains the absolute value of n as a one or two digit integer. Note that, although as many as nine digits may be given for a, the internal form of floating point numbers is such that they are rounded to the 24th binary bit, or roughly 3 in the 8th significant digit.

Table I contains some examples of ACT language constants, with their translation. Note that the word 0' represents both an integer and a floating point zero. Note also: -.9999'99999"e'o' will give a more accurate representation of 1.0 than .1"e'1'.

TABLE I

| ACT-language | Translation |
|---|---|
| 1' | 1 (integer) |
| 0' | 0 (integer or floating point) |
| 56789' | 56789 (integer) |
| +1'23456' | 123456 (integer) |
| +123'456' | 123456 (integer) |
| +123' ' | 123 (integer) |
| +123'000' | 123000 (integer) |
| +7000'00000' | error type 3 |
| .1''e'1' | 0.1 x 10 = 1 (fl. pt.) |
| .1''e'0' | = 0.1 (fl.pt.) |
| .1''e-'1' | = 0.01 (fl.pt.) |
| .1234'56'e'1' | = 1.23456 (fl. Pt.) |

| ACT - Language | Translation |
| --- | --- |
| .1''e'34' | error type 3. |
| .1''e-'34' | error type 3. |

A.2 Statement numbers.

A statement number is a single word, consisting of the letter s followed by an integer. Only statement numbers less than 192 are allowed.

The ACT language program is divided into segments called statements. For example, a single equation will normally occupy a statement. The end of a statement is signalled by a blank word ( a conditional stop with no preceding characters). A statement number occurring as the first word of a statement causes this statement to be assigned the given number. A statement number occurring at any other position in a statement is interpreted as a reference to the statement so identified. A second blank word at the end of a statement is interpreted as the physical end of the program being compiled.

A.3 Operations

The words listed in Tables IV -VIII are operations in the standard ACT III system. It must be emphasized here that this is not an inflexible list; it may be augmented or diminished as desired. This is discussed in detail in Volume 2 of the present manual. The individual operations are discussed more fully elsewhere; they may be classified roughly as arithmetic, input-output, functions, logical, bookkeeping, and machine operations. For present purposes, it will suffice to note that the standard ACT III tables regard all the words of Tables III - VIII as operations; if a given subroutine implementing it has not been included, its use causes an error indication (type 8). A statement must contain at least one operation; otherwise, it will produce no instructions in the machine language program.

A.4 Machine Language Operations

Although ACT III is primarily an algebraic compiler, provision is made to code in symbolic machine language for the convenience of users familiar with machine language who desire to perform special operations not otherwise available. Table VIII lists the machine language operation codes available. This includes all the codes except "p" and "i"; basic input and output may be obtained by the codes "daprt", and "rdhex", and "hxpch" described in Chapter V.

It is possible to incorporate library subroutines already available in machine language. This procedure is described in Volume 2.

A.5 Remarks

Remarks may be included in an ACT III source program in the following form. The remark must contain at least 6 characters; the sixth from the last character must be one of the letters z, b, y, r, i, d, m, n, p, e, u, t, h, c, a, s. (The "command" letters). The remark must be the first "word" in a statement (or it may follow the statement number) and is followed by a single conditional stop. In counting characters for the above purpose, do not count carriage return, upper and lower case, color shift, backspace. "Tab" counts as a character usually; it is wise to avoid using it in this context.

B. Errors in ACT-language programs.

This manual contains a number of rules which constitute the syntax of ACT-language. Some of these rules have been stated above; others will be found in connection with the treatment of special topics of ACT-grammar. If one of these rules of syntax is violated, an error indication is given, consisting of a carriage return, the letter "e", a space, and a digit giving the error type. These types are summarized in Table II, and are discussed in greater detail at the appropriate place in the text.

After such a display, the computer will stop. In all cases except the type 7 error at the end of a program, and type 2 indication, it is possible to correct the statement being processed and reprocess it by pressing "start".

TABLE II

ERRORS AT COMPILE TIME

| TYPE | MEANING | REMEDY |
|---|---|---|
| 1 | Symbol table full (max. 126) | Put some variables into regions. |
| 1 | Too many constants (max. 63) | Read in some as data |
| 2* | Storage exceeded | See Chapter VI |
| 3 | Incorrect constant | Correct tape and restart at beginning of statement |
| 4 | Improper use of "enter", "end", or "exit" | " |
| 5 | Invalid bracket count | " |
| 6 | Statement too large | Segment statement and restart at beginning of statement |
| 4 | Too many arguments in an "iter" statement | Correct tape and restart at beginning of statement |
| 7 | Statement number 191 | Correct tape and restart at beginning of statement |
| 7* | Undefined statement | Correct tape and recompile |
| 8 | 6-bit button up | Restart at beginning of statement |
| 8 | Invalid subscript | Correct tape and restart at beginning of statement |
| 8 | Operation Code not included in this package | Correct tape and restart at beginning of statement or recompile using correct T-tape |
| 9 | Invalid operand | Correct tape and restart at beginning of statement |

* See Supplement for special error displays

TABLE III

ARITHMETIC OPERATIONS

| Code | Prec. | Operands | Reference | Package | Error Type | Error Meaning |
|---|---|---|---|---|---|---|
| ; : | 0 | l.r. | II-C.1. | P-0 | | |
| + | 1 | l.r.a. | II-C.2. | P-1 | 1 | fl.pt. overflow |
| - | 1 | l.r.a. | II-C.2. | P-1 | 1 | fl.pt. overflow |
| x | 2 | l.r.a | II-C.2. | P-1 | 1 | fl.pt. overflow |
| / | 2 | l.r.a. | II-C.2.<br>IV-E | P-1 | 1<br>2 | fl.pt. overflow<br>rt. operand zero |
| abs | 3 | r.a. | II-C.2. | P-1 | | |
| i+ | 1 | l.r.a. | II-C.3. | P-0 | | |
| i- | 1 | l.r.a. | II-C.3. | P-0 | | |
| ix | 2 | l.r.a. | II-C.3. | P-2 | 3 | integer overflow |
| i/ | 2 | l.r.a. | II-C.3. | P-2 | 3 | rt. operand zero (usually) |
| nx | 2 | l.r.a. | II-C.3. | P-0 | | |
| iabs | 3 | r.a. | II-C.3. | P-0 | | |
| prev | 3 | a. | II-C.1. | P-0 | | |
| flo | 3 | l.r.a. | II-C.4 | P-2 | 1 | fl.pt. overflow |
| unflo | 3 | l.r.a. | II-C.4. | P-2 | 3 | integer overflow |
| fix | 3 | l.r.a. | II-C.4. | P-2 | 3 | integer overflow |
| x10p | 3 | l.r.a. | II-C.4. | P-2 | 1 | fl.pt. overflow |
| pwr | 3 | l.r.a. | II-C.2. | PELL | 1<br>2 | fl.pt. overflow<br>left operand neg.; or, left op. zero and rt. op. neg. |
| ipwr | 3 | l.r.a. | II-C.3. | P-2 | 3 | left op. zero, and right op. neg. |

TABLE IV

LOGIC OPERATIONS

| Code | Reference | Package |
|---|---|---|
| use | IV - A | P - 0 |
| trn | IV - A | " |
| iter | IV - D | " |
| if | IV - B | " |
| neg | IV - B | " |
| zero | IV - B | " |
| pos | IV - B | " |
| stop | IV - F | " |
| ret | IV - C.1. | " |
| set | IV - C.2. | " |
| to | IV - C.2. | " |
| bkp4 | IV - E | " |
| bkp8 | IV - E | " |
| bkp16 | IV - E | " |
| bkp32 | IV - E | " |
| oflow | IV - E | " |
| go to | IV - C.1. | " |

TABLE V

INPUT - OUTPUT OPERATIONS

| Code | Operands | Reference | Package | Error Type | Error Meaning |
|---|---|---|---|---|---|
| rdflo | l.r. | V - A.3. | P-2 | 1("flo") | fl.pt. overflow |
| read | r.a. | V - A.2. | P-1 | 1 | fl.pt. overflow |
| print | l.r. | V - B.2. | P-1 | | |
| dprt | l.r. | V - B.3. | P-1 | | |
| iread | r.a. | V - A.1. | P-0 | | |
| iprt | l.r. | V - B.1. | P-0 | | |
| punch | r. | V - B.4. | P-1 | | |
| ipch | r. | V - B.4. | P-0 | 3 | more than 7 digits |
| aread | r.a. | V - C | P-0 | | |
| aprt | r. | V - C | P-0 | | |
| cr | | | P-0 | | |
| tab | | | P-0 | | |
| daprt | | V - D | P-0 | | |
| reprt | | V - D | P-0 | | |
| rdxit | r. | V - A.4. | P-0 | | |
| rehex | r.a. | V - B.5. | P-0 | | |
| hxpch | r. | V - B.5. | P-0 | | |

## TABLE VI

## FUNCTIONS

| Code | Prec. | Operands | Reference | Package | Error Type | Error Meaning |
|---|---|---|---|---|---|---|
| sqrt | 3 | r.a. | IX - B | SQRT | 2 | arg. negative |
| ln | 3 | r.a. | IX - C | PELL | 2 | arg. neg. or zero |
| log | 3 | r.a. | IX - C | PELL | 2 | arg. neg. or zero |
| exp | 3 | r.a. | IX - C | PELL | 1 | fl.pt. overflow |
| sin | 3 | r.a. | IX - D | TRIG | 2 | arg. large; ans. has no significance |
| cos | 3 | r.a. | IX - D | TRIG | " | " |
| artan | 3 | r.a. | IX - E | ARTAN | | |
| randm | 3 | a. | IX - F | P-1 | | |

## TABLE VII
## MISCELLANEOUS OPERATIONS

| Code | Reference | Package |
|---|---|---|
| index | III - C | P-0 |
| dbind | III - D | P-0 |
| dim | III - A | P-0 |
| call | VII - B | P-0 |
| arg | VII - B | P-0 |
| enter | VII - A | P-2 |
| exit | VII - A | P-2 |
| end | VII - A | P-2 |
| trace | VII - C | P-2 |
| wait | VII - D | P-0 |

## TABLE VIII
## MACHINE OPERATIONS

| Code | Meaning |
|---|---|
| bring | b |
| add | a |
| subtr | s |
| mult | m |
| nmult | n |
| div | d |
| extrt | e |
| hold | h |
| clear | c |
| stadd | y |

TABLE IX

CODES FOR "aread"

| SYMBOL | CODE | SYMBOL | CODE |
|---|---|---|---|
| )0 | 04 | Aa | 72 |
| L1 | 0j | Bb | 0f |
| *2 | 14 | Cc | 6f |
| "3 | 1j | Dd | 2f |
| △4 | 24 | Ee | 4f |
| %5 | 2j | Ff | 54 |
| $6 | 34 | Gg | 5j |
| [illegible] 7 | 3j | Hh | 62 |
| Σ8 | 44 | Ii | 22 |
| 9 | 4j | Jj | 64 |
| Space | 06 | Kk | 6j |
| _- | 0a | Ll | 0j |
| =+ | 16 | Mm | 3f |
| : ; | 1a | Nn | 32 |
| ?/ | 26 | Oo | 46 |
| ]. | 2a | Pp | 42 |
| [, | 36 | Qq | 74 |
| TAB | 30 | Rr | 1f |
| Lower Case | 08 | Ss | 7f |
| Upper Case | 10 | Tt | 5f |
| Color Shift | 18 | Uu | 52 |
| Carr. Ret. | 20 | Vv | 3a |
| Back Space | 28 | Ww | 7j |
| ' | 40 | Xx | [illegible]a |
| | | Yy | 12 |
| | | Zz | 02 |

Summary of Operation Codes *Means Precedence does not apply.

| Code | Prec. | Use | Comments |
|---|---|---|---|
| [ | | | left bracket (maximum bracket depth = 7) |
| ] | | | right bracket |
| : | 0 | a':'b' | value of a substituted into b (a unchanged) |
| + | 1 | a'+'b' | floating point addition |
| - | 1 | a'-'b' | floating point subtraction |
| x | 2 | a'x'b' | floating point multiplication |
| / | 2 | a'/'b' | floating point division |
| print | 0 | i'print'a' | (i = 100c + s) print a as fl.pt. no. in field c columns wide, rounded to s significant figures |
| dprt | 0 | i'dprt'a' | (i = 100c + s) print a (fl.pt.) as decimal number in field c columns wide with s fractional digits |
| read | 0 | read'a' | read fl. pt. number and store in a |
| i+ | 1 | a'i+'b' | integer addition |
| i- | 1 | a'i-'b' | integer subtraction |
| ix | 2 | a'ix'b' | integer multiplication |
| i/ | 2 | a'i/'b' | integer division |
| nx | 2 | a'nx'b' | fast integer mult. handles large numb. incorrectly |
| ipwr | 3 | a'ipwr'b' | a raised to the b'th power (integer) |
| x10p | 3 | a'x10p'b' | a (fl. pt.) times 10 to the b'th power(b is integ) |
| use | 0 | use'sl' | transfer control to statement no. l |
| iter | * | iter'i'j'k'sl'' | increase i (integers) by j (pos.) and transfer to sl if the new value of i is not greater than k |
| iread | 0 | iread'a' | read integer and store in a |
| iprt | 0 | i'iprt'a' | (i = 100c) print a(integer) in field c columns wd. |
| abs | 3 | abs'a' | take absolute value of a (fl. pt.) |
| iabs | 3 | iabs'a' | take absolute value of a (integer) |
| punch | 0 | punch'a' | punch a (fl.pt.) in format with cond.stops for rd. |
| ipch | 0 | ipch'a' | punch a (integer)with cond. stop for reloading |
| aread | 0 | aread'a' | read one word in special alphabetic form into a |
| aprt | 0 | aprt'a' | print a as alphabetic information |
| if | 0 | if'a'neg'sl'' | classify a (fl.pt.) or integer) as negative, zero, |
| neg | 0 | if'a'neg'sl'zero's2'' | or positive (but not zero) and transfer |
| zero | 0 | if'a'zero'sl'pos's2'' | to the indicated statement in each case, |
| pos | 0 | if'a'neg'sl'zero's2'pos's3'' | or to next state. if none indicated |
| prev | 3 | prev'x'a' | saves result of previous statement for use as an operand. Must be first operation executed in statement. |
| cr | 0 | cr' | Execute a carriage return |
| tab | 0 | tab' | execute a tab |
| flo | 3 | a'flo'b' | convert b (integer) to fl.pt. counting last a digit as fractional (b unchanged) |
| rdflo | 0 | a'rdflo'b' | read integer, convert it to a fl. pt. number, counting last a digits as fractional & store in b. |
| unflo | 3 | a'unflo'b' | convert b (fl.Pt.) to integer, moving decimal point a places to right (b unchanged) rounded |
| fix | 3 | a'fix'b' | ditto, but fractional digits dropped without round. |
| index | * | index'i'j'' | set up i, j for use as subscripts (max.30) |
| dbind | * | dbind'i'j'' | set up i, j for use as double subscripts |
| dim | * | dim'a'10'b'50'' | reserve 10 sequential locations for a region 50 for b region, etc. |
| daprt | * | daprt's't'o'p' | print alphabetic information as given in source language. See special list for control characters |
| reprt | * | n'reprt'cr4'' | print the indicated character or control n times |
| stop | 0 | stop'' | stop. continue if "start" pressed |
| ret | 0 | ret;s20'use'sll'' | transfer to sll after storing a return address in s20, which must read s20'go to's0'' |

| | | | |
|---|---|---|---|
| trn | 0 | trn's1'' | transfer control to s1 if accumulator negative |
| sqrt | 3 | sqrt'a' | obtain square root of a (fl. pt.) |
| ln | 3 | ln'a' | obtain natural logarithm of a (fl. pt.) |
| log | 3 | log'a' | obtain common logarithm of a (fl. pt.) |
| exp | 3 | exp'a' | obtain the value of e raised to the ath power (f.pt |
| pwr | 3 | a'pwr'b' | a raised to the b'th power (fl. pt.) |
| sin | 3 | sin'a' | obtain the sine of a (a fl.pt. in radians) |
| cos | 3 | cos'a' | obtain the cosine of a (a fl.pt. in radians) |
| artan | 3 | artan'a' | obtain the angle whose tangent is a (fl.pt) radians |
| randm | 3 | randm' | generates a fl.pt. pseudo random no. between 0 and1 |
| set | 0 ) | set's20'to's11' | statement 20, which must have the form s20'go to 's0'' is made to read use 's11' |
| to | 0 ) | | |
| bkp4 | 0 ) | bkp4'use's20'' | On machines equipped with the overflow logic modification only, control is transferred to s20 if indicated breakpoint switch is down (on), otherwise to next statement |
| bkp8 | 0 ) | | |
| bkp16 | 0 ) | | |
| bkp32 | 0 ) | | |
| oflow | 0 | oflow'use's20'' | (Overflow logic only) transfer to s20 if overflow has occurred on a preceding i+, i- or n/ |
| rdhex | 0 | rdhex'a' | read a word of hexadecimal and store in a |
| enter | * | enter'sub'blok1' ' | denotes the beginning of a source-language subroutine whose name is "sub". blok1 is a dummy symbol to enable reference to a sequential block of data specified in the main program calling sequence |
| | see "call" | | |
| exit | * | exit' | return control from source-language subroutine to main program |
| end | * | end' ' | denotes the end of source-language subroutine |
| trace | * | trace'' | causes a source-language subroutine to be trace-compiled if the transfer control button is down |
| call | 0 ) | call'sub'arg'a'' | transfers to the source-language subroutine named "sub", and sets the dummy symbol "blok1" to refer to the actual region a |
| arg | 0 ) | | |
| wait | * | wait' | (one cond. stop) place at end of source-language subroutine tape. suspends compiling while next tape is loaded |
| hxpch | 0 | hxpch'a' | punches a as a hexadecimal word with conditional stop for reading by "rdhex" |
| bring | 0 | bring'a' | machine language instruction |
| add | 0 | add'a' | " |
| subtr | 0 | subtr'a' | " |
| mult | 0 | mult'a' | " (m) |
| nmult | 0 | nmult'a' | " (n) |
| div | 0 | div'a' | " |
| extrt | 0 | extrt'a' | " (e) |
| hold | 0 | hold'a' | " |
| clear | 0 | clear'a' | " |
| stadd | 0 | stadd'a' | " (y) |

Special Symbols

| | |
|---|---|
| remdr | remainder of previous "i/" operation (not "n/") |
| areg1 | a block of 192 locations available for data storage in Mode A only |
| areg2 | a block of 640 locations available for data storage in Mode A only (if more than the first 328 are used, the T-tape must be used for the next compilation) |

CHAPTER II

# ARITHMETIC OPERATIONS

## A. Syntax.

Operations may be divided syntactically in accordance with two considerations; the operand-answer relation and the precedence.

The precedence is an attempt to formalize the combinational rules of algebra, and gives an indication of the order in which different operations appearing in a statement are performed. Thus, algebraic functions, such as "sin", which are of precedence 3, are performed before multiplications and divisions (precedence 2), and the latter are performed before additions and subtractions (precedence 1).

Some operations require a left operand (l.); some require a right operand (r.); some have a numerical result or answer (a.). These requirements are listed in Tables III-VIII for the various standard operations. The result of any operation may be used as an operand for another operation; if an operation has no answer (e.g. "print"), it cannot be so used. Any failure to observe these syntactical rules will result in an error stop (type 9).

## B. Brackets [ ] and "rank".

Brackets may be used in any statement to modify the order of execution of operations. As a general rule, such brackets carry the standard algebraic significance. This may be formulated quantitatively by defining a "rank" for each operation as the precedence plus four times the bracket level. These ranks determine the order of execution of the operations in a statement in accordance with definite rules. If two neighboring operations have different ranks, the one of higher rank is executed first; if they have the same rank, the one on the left is executed first. Brackets may be nested to a maximum depth of 7; violation of this rule, or of the standard rules regarding unmatched brackets in algebraic expressions, causes an error stop (type 3).

Redundant brackets are, as a rule, ignored. In any case, they can do no more harm than adding a few milliseconds to the running time of the program. Thus if you are doubtful about precedence rules, play safe and insert brackets to remove possible ambiguity.

## C. Arithmetic Operations.

The arithmetic operations are listed in Table III. All these operations (except "abs" and "iabs") require both a left operand and a right operand. They fall naturally into two main classifications, integer and floating-point. As a general rule, integer operations are preceded by the letter i.

### C.1. The Substitution Operator ":" and Previous Result "prev".

We have no use for an equal sign, which represents a statement of fact rather than an operation. In its place we have the substitution operator ":". The statement

a':'b''

causes the value of a to be stored in the location labeled b. The previous value of b is destroyed in this process. The only other operations which can change the contents of memory are the input operations (see Chapter V).

It is to be noted that the substitution operator has no "result". Actually, the number stored remains in the accumulator, but the operation table is coded so that this may not be used directly as an operand. The purpose is to enable detection of some common programming errors. Sometimes it would be desirable to make use of this previous result. The operation "prev" serves the desired purpose. One example will suffice at this time; another example may be found in Chapter V, section A.5.

Suppose one desires to store the value zero in locations a, b, and c. The coding

```
→
0':'a''
0':'b''
0':'c''
```

will accomplish this purpose; however, it is necessary to recall the number "zero" at each step, which consumes two instructions and 30 milliseconds of running time (in this case; the discrepancy may be considerably larger in other cases). The coding

```
0':'a':'b':'c''
```

will not work because ":" has no "result"; however, this may be bypassed by writing

```
0':'a''
prev':'b''
prev':'c''.
```

Thus, whenever the result of the previous statement is needed for further use, the operation "prev" will avoid having to recall it from memory. "Prev" must be the first operation executed in a statement in order to accomplish this purpose.

C.2. Floating-Point Arithmetic.

The basic floating--point arithmetic operations are as follows:

| ACT Language | Meaning |
|---|---|
| a'+'b' | fl.pt.sum |
| a'-'b' | fl.pt.difference |
| a'x'b' | fl.pt.product |
| a'/'b' | fl.pt.quotient |
| a'pwr'b' | a rised to the power b |
| abs'a' | absolute value (magnitude) of a |

These operate on floating-point numbers in the indicated manner. If a result would have an exponent greater than 31, an error stop (type 1) occurs (at run-time--see Chapter VIII). If a result would have an exponent less than -32, it is returned as zero. It should be noted that zero is seldom obtained as the result of "+" or "-"; the problems of round-off and binary-decimal conversion make exact cancellation unlikely.

An attempt to divide by zero (or a number not in proper floating-point form) will give a type 2 error stop, as will the attempt to raise zero to a negative power, or a negative number to any power.

C.3. Integer Arithmetic.

Integer arithmetic is provided in the standard ACT III system primarily for use in counters and subscripts. The primary operations of value in this case are addition and subtraction.

```
a'i+'b'
a'i-'b'
```

These are simple machine operations. If the result exceeds 536,870,911, a machine overflow will occur. This stops the computer if a standard logic board is being used; it sets an indicator which can be tested later (see Chapter XI) if an overflow modification board is used.

For situations where products of small integers are required, the operation "nx"

```
a'nx'b'
```

may be used. This will give incorrect results if the product is greater than 134,217,727.

If products are required and the above limitation is too strict, the operation "ix"

```
a'ix'b'
```

is available. It is considerably slower in operation than "nx", and requires an extra subroutine. If the product exceeds 536,870,911, an error stop (type 3) will occur.

Division of integers in accomplished by the operation "i/"

```
a'i/'b'.
```

The quotient is, of course, an integer, and is determined by the requirement that the remainder must have the same sign as the denominator. This is consistent with the "modulo" concept of number theory. If a and b are both positive, it means that the quotient is the largest integer contained in a/b.

The remainder obtained in this division is stored in the special location (symbol) "remdr", and may be used at any later time in the program.

The absolute value of an integer is obtained by "iabs".

The expression

a'ipwr'b'

produces the result of raising a to the power b. An attempt to raise zero to a negative power gives a type 3 error stop. The answer is obtained by successive multiplications (if b is positive). If b is negative and a = 1, the answer is 1. If b is negative and a is greater than 1, the answer is zero.

## C.4. Mixed Arithmetic.

Several operations deal with both integers and floating-point numbers.

The operation "flo"

n'flo'a'

gives an answer the result of interpreting the "integer" a as a decimal number with n (integer) digits after the decimal point. (n may be positive or negative.) Some examples follow:

| n | a | result (fl.pt.) |
|---|---|---|
| 0 | 123 | 123.0 |
| 1 | 123 | 12.3 |
| -1 | 123 | 1230.0 |

The operation "unflo"

n'unflo'a'

converts the floating-point number ($a \times 10^n$) to an integer, rounding as needed.

| n | a | result (fl.pt.) |
|---|---|---|
| 0 | 15.73 | 16 |
| 2 | 1.822 | 182 |
| -2 | 2947.3 | 29 |

The operation "fix"

n'fix'a'

is similar to "unflo", except that no rounding occurs. Thus the results in the above sample cases would be 15,182,and 29 respectively.

It is not advisable to use floating-point numbers as counters, since the binary-decimal conversion problem introduces roundoff errors which will accumulate. An important use of the above three operations is in any case in which a counter is also needed for floating-point calculations.

The operation "x10p"

a'x10p'n'

produces the product of the floating-point number a by $10^n$ (n is an integer). The result is floating-point.

## CHAPTER III

## SUBSCRIPTED VARIABLES

### A. Regions

A series of sequential locations may be set aside as a "region". The statement (dim. = dimension)

dim'a'50'b'25''

reserves 50 locations for region "a" and 25 locations for region "b". As many regions as desired may be set aside in one dimension statement; more than one dimension statement may be used; and they may occur anywhere in the program. Note, however, that the names ("a" and "b" in the above example) should not have been used previously; if they have been, this statement changes their definition, so that the symbol "a" used before this statement refers to a different location from that assigned after this statement.

### B. Integer Subscripts

The word pair a'25' refers to the 26th word of region "a". It is not necessary that "a" be assigned a size of 25 or more, or even that "a" be previously named in a dimension statement. In the latter case, note the fact that all variables are assigned consecutive locations in the order in which they first appear in the program. Blocks reserved by dimension statements also fall into this order as they occur.

The first word of region "a" is a'0', or simply a'. Thus, in the above example, the fifty words of "a" are a'0' thru a'49'. Then comes b'0'. The word pair a'51' would thus have the same meaning a b'1'.

If variables c, d, and e are used (not in a dimension statement) in that order, then c'1' is the same as d', etc.

### C. Single Variable Subscripts.

The statement index'i'j''

sets up a mechanism by which the variables i and j may be used as variable subscripts. Up to 31 subscripts may be named in a single index statement. More than one index statement may occur, and they may appear at any place in a program. In the latter case, the same comments appearing in section A. for dimension statements apply.

The word pair a'i' refers to the (i + 1)st word of region a, where the current value at run-time of the integer variable i is used. If the value of i is not properly set before using it, the resultant location may be anywhere on the drum, with mysterious results. Note that "i" must have been previously named in an index statement, or an error stop will occur.

The words a'i'25' (or a'25'i') refer to the i + 26)th word of region "a". Again, i must be named previously in an index statement.

## D. Double Subscripts

The statement

dbind'm''

makes m available as a double subscript. Two locations are reserved for each symbol so named, referred to (in the case of "m") as m'0' (or simply m') and m'1'.

The word pair a'm' now refers to the location in region "a" corresponding to the matrix element in row m'0' and column m'1' of a matrix stored row-wise beginning at a'1'. The number of columns of the matrix (an integer) must be first placed in a'0'.

Other properties of double subscripts are as described for single subscripts.

## E. Adjacent Variables

If two variables (e.g. a and b) appear next to each other (as a'b') or if a variable and a constant or two variables and a constant appear together (not separated by operation words), they are interpreted in accordance with the preceding sections. If they do not satisfy the conditions laid out there, an error stop will occur.

COLUMN INDEX m'1'

a'0'
#COL.

ROW
INDEX m'0'

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | a1 | a2 | a3 | a4 | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| . | | | | | | | | |
| . | | | | | | | | |

## CHAPTER IV

## LOGIC OPERATIONS

### A. Logic Flow.

The normal flow of operations (at run-time) is from one statement to the next in sequence, as they appear in the source program. Several operations are available which may change this flow. The machine operations "use" and "trn" are two such. Thus the statement

```
use'sl''
```

transfers control unconditionally to statement number 1, and the statement

```
trn'sl''
```

transfers to statement 1 only if the result of the previous statement is negative (otherwise the normal flow occurs).

### B. "If" Statements.

The statement

```
if'a'neg'sl'zero's2'pos's3''
```

transfers control of statement 1 if a is negative, statement 2 if a is zero, and statement 3 if a is positive (non-zero). The variable "a" may be integer or floating-point. It may also be replaced by an expression with a result (integer or floating-point). Such an expression must be enclosed in brackets. It is not necessary to specify all of the tests ("neg", "zero", "pos"); any one or any two may be obmitted. The normal flow (to the next statement) is understood for the missing tests. It is necessary that the relative order of the three tests be preserved in the statement. Thus;

```
if'['a'+'b']'neg'sl'pos's3''
if'['a'-'b']'zero's5''
```

### C. Variable Connectors.

There are several ways, in addition to the "if" statement, of setting up variable transfers.

C.1.

The statement

```
ret's10'use's11''
```

where statement 10 (appearing elsewhere in the program) has the form

```
s10'go to's0''
```

first changes statement 10 to "use (next statement after the ret-use statement)", then transfers to s11. Thus a block of programming may be

set up, beginning at s11 and ending with s10, which may be called as a unit from various places in the program. Note the space in the word "go to".

C.2. The words "go to" and "use" are not interchangeable. The symbol "SO" is a dummy symbol with no special significance.
The statement

```
set's10'to's5''
```

where statement 10, as before, reads

```
s10'go to's0''
```

causes s10 to be changed to use 's5'', but the flow then proceeds to the next statement following the set-to statement.

C.3. Subscripted Statement Numbers.

The word pair s5'i', where i is named in an index statement, transfers to the i'th location before the beginning of statement 5. Thus the statement preceding statement five might read

```
use's10'use's15'use's12'use's3''
s5' (etc.).
```

In this case, if i = 0 (at run-time) s5'i' refers to s5 itself; if i = 1 it refers to the use's3', if i = 2 to the use 's12', etc. Note that the "statement dictionary" so constituted must all be contained in one statement.

Note that the word pair s5'2' (for example) refers to statement number (5-2) = 3, without reference to any such statement dictionary. Similarly, the combinations s5'2'i' and s5'i'2' are equivalent to s3'i'. Subscripted statement numbers must not be used after "to"; or after "use" in a "ret-use" statement; or after "zero" in an "if" statement.

D. Programming Loops.

The statement

```
iter'i'j'k's1''
```

has the following significance: increase i by the amount j (must be positive); if the new value of i is less than or equal to k, transfer to s1, otherwise go to the next statement.

As an example, the following coding will obtain the sum of 26 floating-point numbers in locations a'0' thru a'25'.

```
0':'sum''
prev':'i''
s1'a'i'+sum':'sum''
iter'i'1'25's1''
```

The arithmetic performed in an iteration statement is integer arithmetic. The four arguments of the statement need not be simple variables. The first argument (i in the above) may be a subscripted variable, the second and third arguments may be integer expressions of any complexity, and the statement number may be subscripted. Any such argument, however, must be enclosed in parentheses. Example:

```
iter'['m'1']'['m'1'i+'3'ix'j']'k'['s1'i']''
```

E. Breakpoint and Overflow Tests.

20400
U 515
↓

The statement

bkp4'use's15''

when used on a computer with overflow logic board (see Chapter XI) causes a transfer to statement 15 if breakpoint 4 is on; otherwise control passes to the next statement. The codes bkp8, bkp16, and bkp32 have similar meanings. If this is used with a standard logic board, and the breakpoint is on, control passes to the next statement. If it is off, a stop occurs. Pressing start will continue with the next statement. Execution of the sequence "one operation, manual input, start, one operation, normal, start" will transfer to statement 15.

The statement

oflow'use's15''

when used with the overflow logic modification transfers to s15 if the overflow indicator is on as the result of a previous i+ or i- operation. It should be noted, however, that the overflow is turned off during execution of a + or -; and that, if the overflow is on at entry to / (fl. pt. divide), it will cause a type 2 error stop.

Subscripted statement numbers must not be used with these tests.

F. Stops.

The statement

stop''

will cause the computer to stop. If the statement is numbered and was not trace-compiled, the statement number will appear at a q of 11 in the instruction register.

CHAPTER V

# INPUT-OUTPUT OPERATIONS

## A. Numerical Data Input.

Numerical data may be read into the computer (at run time) in either integer or floating-point format. The operations which achieve this result are "iread" (for integers), and "read" (for floating-point numbers). Also available is the operation "rdflo", which is a read-and-float procedure.

### A.1. Integer input.

The statement

```
iread'n''
```

causes a single word to be read from tape, interpreted as an integer, and stored in n.

The proper form for integer data is a sign (+ or space for positive, - for negative) followed by one to seven numeric digits. The + sign is recommended, since a space may be overlooked. This is followed by a conditional stop. The sign must be the first character read; it may be preceded by a carriage return, which enters nothing into the computer, but not by a tab, which enters as a space (unless, of course, you wish this to be recognized as a plus sign). not as + b.f

If it is desired to include textual matter with the data word, the last eight characters must constitute the data word, as follows: first character is the sign, followed by a seven-digit integer with leading zeros (or spaces).

### A.2 Floating-point Input.

The statement

```
read'a''
```

causes two words to be read from tape, interpreted as the mantissa and exponent of a floating-point number, and the resulting number stored in a.

The proper form for floating-point data is as follows. The first word consists of the sign and one to seven significant digits of the mantissa. The decimal point is understood to be immediately to the right of the sign. Non-significant trailing zeros may be omitted unless textual material precedes the sign. The second word contains the exponent, in the form of an integer as explained above.

### A.3. Read-and-float.

The statement

```
n'rdflo'a''
```

causes a single word to be read from tape in the form of an integer, interpreted as a decimal number with n digits following the decimal point, converted to a floating point number and stored in a. The decimal indicator n may be positive or negative.

A.4. Data Input Termination.

In batch processing, reading in lengthy tables of data, and other applications, it is necessary to include provision to vary the amount of data from one run to another. Suppose, for example, you write a program designed for batch processing, in which a set of data is read, computed results are printed, and then new data are read to repeat the process. One way to control the number of sets of data read is to incorporate in the program a separate "iread" (executed only once) which reads first the number of cases on the tape.

A more useful procedure is provided by the statement

```
rdxit's12''
```

execution of which (at run time) sets a special exit from the data read subroutines to statement 12. This exit is used whenever a blank word (conditional stop only) is read from the data tape. The placing of this statement at the beginning of the above program allows the batch processing to be terminated by merely placing an extra conditional stop after the last set of data. When this is read, control is transferred to s12, which may merely say "stop", or may start another section of the program. Several "rdxit" statements may be used in a program in this way, changing the data exit as required in various sections of the program. If a blank word is read before "rdxit" has been set, an error stop will occur.

Tables of varying length may be read in using this method. The sequence

```
rdxit's2''
1':'i''
s1'n'rdflo'a'i''
iter'i'1'99999's1''
s2' (etc.)
```

with read and float data words, loading into sequential locations $a_1$, $a_2$, etc., until the extra conditional stop is read, at which time control is transferred to s2. The value of i is then one more than the number of entries in the table. The upper limit 99999 is, of course, a dummy, since this many words would never be used.

A.5. General Comments on Data Input Operations.

The right operand of "read", "iread", or "rdflo" must, of course, be a simple variable. The statement

```
read' 'a'x'b' ''
```

will give no error indication, but will accomplish nothing useful in the running program.

The result of a data input operation may be used as an operand. Thus, if it had been desired in the above loop to accumulate the sum of the numbers as they were read in, the following statement could have been inserted before the iteration statement.

```
prev'+'sum':'sum''
```

Statement number 1 could, alternatively, have been changed to read

```
sl'['n'rdflo'a'i']'+'sum':'sum''
```

The advantage to be gained in either case is the time necessary to recall the indexed variable, which is about 225 milliseconds.

Instructive examples are included at the end of the chapter.

B. Numerical Data Output.

Remember: bkp 32 ON for high speed punch
bkp 32 OFF for flexowriter

B.1. Integer Output.

The statement

```
n'iprt'a''
```

(where n and a are both integers) causes the integer a to be printed or punched in a field n/100 characters wide, with leading spaces. Negative numbers have the sign printed immediately to the left of the high-order digit. A nine-digit integer may be in error by 1 or 2.

As an example, if n=1100 and a=5372, the routine will print 7 spaces followed by the number 5372. If a=-5372, then 6 spaces would be printed, followed by -5372.

If the field is too small for the number to be printed, the size of the field is respected. In this case, the sign is replaced by a p or m (for plus or minus), followed by enough of the high-order digits to fill the field. If n/100 = 1, a conditional stop is printed; if n/100 = 0, nothing is printed.

B.2. Floating Point Output.

The statement

```
n'print'a''
```

where a is a floating-point number and n = 100c + s (c and s integers), causes the floating-point number a to be printed in a field c characters wide rounded to s significant digits. The format consists of leading spaces as required, followed by a sign (space or -), decimal point and s digits of the mantissa, space, "e", and the exponent printed as an integer in a field three characters wide. If c is too small to make room for s digits, the value of s is decreased as required. If c is less than 7, the routine simply prints the exponent as an integer in a field of c characters.

B.3. Unfloat and Print (decimal print).

The statement

```
n'dprt'a''
```

where a is a floating point number and n = 100c + f (c and f integers) causes the floating-point number a to be unfloated and printed as a decimal number in a field c characters wide, rounded to f fractional digits. The format consists of leading spaces are required, followed by a sign (space or - ), integral digits (if any), decimal point, and franctional digits

(if any). If the number is too large to be printed as specified, f is decreased as required. If this measure fails, control is transferred to the "print" routine (see B.2.).

If f = 0 and the number is less than 0.5, it is printed as 0.

B.4. Compatible Output.

Routines are available for punching numbers in a form which may be used directly as input to another program. If legible output is desired, the statements

```
          punch'a''
and       ipch'n''
```

produce legible output for floating-point and integer data, respectively. The floating-point format consists of a sign and seven digits for mantissa, a conditional stop, a sign and two digits for exponent, a conditional stop. The integer format consists of a sign and seven digits (with leading zeros) followed by a conditional stop. If the integer contains more than seven digits, an error stop will occur.

B.5. Hexadecimal Input-Output.

The statement

```
rdhex'a''
```

causes a single hex word to be read and stored in a. The statement

```
hxpch'a''
```

causes a to be punched as an eight-character hex word with a conditional stop. This pair of operations is recommended for intermediate output of data required for another program when legibility is not required. The advantages are increased input-output speed and increased accuracy (in floating-point numbers, where the binary-decimal conversion causes inaccuracy).

B.6. General Comments on Data Output Operations.

All the output routines will operate on the flexowriter (breakpoint 32 up) or the high-speed punch (breakpoint 32 down). Since an output order has no "result", it may not be used as an operand.

Instructive examples are included at the end of the chapter.

C. Alphanumeric Input-Output.

The statement

```
aread'a''
```

will cause a single word to be read from tape, interpreted as alphanumeric information in a special two-digit-per-character code (four characters or less per word), and stored in a. The two-digit code is given in table IX. It is identical to tbe code used in the standard alphanumeric print subroutine 19.0. A blank word transfers to "rdxit" (see A.4.).

The statement

aprt'a''

causes the contents of a, interpreted as alphabetic information of up to five characters, to be printed. Five characters are possible, although words read in by aread" may contain no more than four. This limitation is due to the input format rather than the internal storage format. The output will run on the high-speed punch if breakpoint 32 is down.

If, through error, a numeric data word is in a, some of the characters may not be acceptable to the flexowriter. This may cause a stop during the print operation. No such stop will occur on the high-speed punch, since all characters are acceptable to it. If a is a negative data word, nothing will be punched.

D. Direct Alphanumeric Print.

The statement

daprt's't'o'p''

will cause the compilation of instructions to print the word "stop" (at run time). Up to 30 characters may be so indicated in a single statement, there is no objection to having a series of such statements. All characters acceptable to the flexowriter (including space) may be included in this way. The control characters are represented by the following mnemonic codes.

| | |
|---|---|
| lower case | lc1' |
| upper case | uc2' |
| color shift | color' |
| carriage return | cr4' |
| back space | bs5' |
| conditional stop | stop' |
| (or apostrophe | ap') |
| tab | tab6' |

Execution of the above statement does not affect the contents of the accumulator.

The statement

n'reprt'x''

(where n is an integer, and x stands for any character, as discussed above) causes the high-speed printing of the indicated character n times. If n is negative, nothing is printed.

E. Example.

The following pages contain undoctored flexowriter listings of a sample program showing input-output format. The first page contains the source-language program and the data tape. The second page is a record of the compilation, and the third page is the result of the actual run. Most of the features of numeric input-output format are demonstrated.

The headings on the data tape are for purposes of labelling the output (page V-8) and are included as a sample of this method of inserting headings. All fields in the output begin immediately under the first character of the heading, having been aligned by a "tab" in the source program. Note that the output is always right-justified, with leading spaces inseted where necessary to fill the field.

The two controls cr' and tab', not previously discussed, have the same effect as daprt'cr4' and daprt'tab6' respectively. If these operations stand alone, the shorter forms compile more rapidly, but if they are part of a series of printed characters, they should be included under the "daprt" for more rapid compiling. The output program is the same in either case.

Source Program

```
( Examples of input-output format ')

Part I -- integer format  '

sl1'      rdxit'sl2''
sl'       cr'iread'm''
          tab'iread'n''
          tab'm'iprt'n''
          use'sl''

Part II -- floating point format '

sl2'      rdxit'sl3''
s2'       cr'iread'm''
          tab'read'a''
          tab'm'print'a''
          tab'm'dprt'a''
          use's2''

Part III -- "rdflo" format '

sl3'      rdxit'sl4''
s3'       cr'iread'n'tab''
          prev'rdflo'a''
          tab'1608'print'a''
          use's3''
sl4'      stop'use'sl1'''
```

Data Tape

```
Examples of input-output format

Part I -- integer input-output

field             number            "iprt" output
+   1000'+12345'+900'-12345'+1000'+0'+300'+12345'+300'-12345''

Part II -- floating point format

field             number            "print" output  "dprt" output
+   1407'+1'-2'+1407'+1'+4'+1407'+1'+20'
+1204'+5555555'+1'+1204'+5555555'-4'+1204'+9999999'+8''

Part III -- "rdflo"

dec.pt.           number            output
       0'½1234'+5'-1234'-20'+1234''
```

## Compilation Record

Examples of input-output format '

Part I -- integer format '

```
sll'     rdxit'sl2''
sl'      cr'iread'm''
         tab'irea'n''
         tab'm'iprt'n''
         use'sl''
```

Part II -- floating point format '

```
sl2'     rdxit'sl3''
s2'      cr'iread'm''
         tab'read'a''
         tab'm'print'a''
         tab'm'dprt'a''
         use's2''
```

Part III -- "rdflo" format '

```
sl3'     rdxit'sl4''
s3'      cr'iread'n'tab''
         prev'rdflo'a''
         tab'1608'print'a''
         use's3''
sl4'     stop'use'sll'''
```

```
f 0414

s 01  0305
s C2  0326
s 03  0354
s 11  0302
s 12  0323
s 13  0351
s 14  0412
```

Program Output

Examples of input-output format

Part I -- integer input-output

```
field          number        "iprt" output
+   1000'      +12345'          12345
+900'          -12345'         -12345
+1000'         +0'                  0
+300'          +12345'       p12
+300'          -12345'       m12
'
```

Part II -- floating point format

```
field          number         "print" output   "dprt" output
+   1407'      +1'-2'         .1000000 e -2          .0010000
+1407'         +1'+4'         .1000000 e  4      1000.0002849
+1407'         +1'+20'        .1000000 e 20     .1000000 e 20

+1204'         +5555555'+1'     .5556 e  1            5.5556
+1204'         +5555555'-4'     .5556 e -4             .0001
+1204'         +9999999'+8'     .1000 e  9       99999988.54
'
```

Part III -- "rdflo"

```
dec.pt.        number       output
      0'       +1234'        .12339997 e  4
+5'            -1234'       -.12340003 e -1
-20'           +1234'        .12339997 e 24
```

# CHAPTER VI

## STORAGE PROBLEMS

NOTE: All of the allocations discussed below are taken care of by your T-tape and R-tape

### A. Mode A

In mode A operations, 32 tracks (of 64 words each) are allocated to the compiler and its temporary storage. Of these, one block of 3 tracks may be freely used for data. This is set apart as the region "aregl". Use of more than these three tracks (aregl'0' to aregl'191') will cause data to be written over the compiler itself.

Another block of 10 tracks is designated as areg2'0' to areg2'639'. The first five tracks of this region (0 to 319) may be used freely for data storage; the second five may be used, but in this case the whole T-tape must be used for the next compilation. If more than ten tracks are used in this region, the operating subroutines (P-tape) will be damaged.

Of the remaining 27 tracks (3-29 inclusive) the subroutines occupy the top part (from 29 down), the compiler program runs from track 3 up and data storage lies in between.

If a type 2 stop (storage full) occurs, several procedures may help:

a. compile without trace (saves 2 words per statement)
b. reduce all regions to bare minimum size
c. move all data to aregl and areg2
d. use less subroutines if possible
e. go to Mode B

### B. Mode B

In Mode B operation tracks 3 - 61 inclusive are available for the compiled program (from 3 up), subroutines (from 61 down) and data (from the subroutines down). However, it is not possible in any case for the compiled program to occupy more than 27 tracks (3 - 29 inclusive).

# CHAPTER VII

## PROCEDURES
## (SOURCE-LANGUAGE SUBROUTINES)

In many programs, it is desirable to have certain blocks of programming (at source-language level) set apart as closed subroutines which may be called upon at various points within the main program. Typical examples would be a plotting routine, matrix inversion, etc. The "retuse" language discussed in Chapter III may be used for this purpose, but it is not extremely flexible. The present chapter describes ACT III language provisions for this situation.

### A. Basic Codes

The statement

```
enter'plot''
```

is interpreted as the physical beginning of a source-language subroutine or procedure which is thereby assigned the symbolic name "plot".

The statement

```
exit''
```

executes a return from the procedure to the main program.

The statement

```
end''
```

is interpreted as the physical termination of a procedure. The block from an "enter" to the next following "end" is the procedure. Tests are incorporated which implement the following diagnostic checks.

a. An "enter" may not occur within a procedure
b. An "end" may not occur unless an unmatched "enter" has been processed
c. An "exit" must lie within a procedure

### A.1. Data Blocks; Indirect Addressing

Many procedures have single input and a single result. These may conveniently be placed in the accumulator. Often, however, a procedure will operate on an entire block of data.

A sample of this would be a matrix inversion procedure, which must know the location of the matrix to be inverted and the location in which the inverse is to be placed. This is accomplished by beginning the procedure with

```
enter'minv'1minv'2minv''
```

Thus the name "minv" is assigned, and the two symbols, lminv and 2minv, are set up as dummy references to the two data blocks. No space must be reserved (by "dim" statements) for these symbols. We suggest in all cases, to avoid duplication, that such dummy symbols be named as here, with a numeric digit followed by the first four characters of the name.

Any variable subscripts required by the procedure must be set up, preferably immediately after the "enter" statement, by "index" and "dbind" statements. Such subscripts should have names of the type indicated above.

Suppose we write, after the above statement,

```
dbind'3minv''
```

Then the variable

```
lminv'3minv'
```

refers to the element of the original matrix determined by the double subscript "3mniv". Specific words in a data block may be referred to by integer subscripts, as

```
lminv'3'
```

Note that "lminv" without a subscript should not be used; it does not refer to word zero of the block, but rather to the location in which the origin of the block is stored for purposes of indirect addressing. Thus word zero must be written out as

```
lminv'0'
```

The special subscript arithmetic, e.g. lminv'3minv'2' is not allowed when using indirect addressing.

B. Calling Upon a Procedure

The procedure headed by the statement given in A.l.

```
enter'minv'lminv'2minv''
```

is called upon in the main program as follows.

```
call'minv'arg'a'arg'b'
```

Here "a" and "b" are the actual names of the blocks for the original and inverse matrices, as set apart by "dim" statements in the main program. Assuming the procedure has been written and compiled, the above statement will cause the matrix "a" to be inverted and its inverse stored in "b". The same procedure may be called more than once, with different matrices to be operated upon.

It is necessary that the procedure be compiled before a statement calling upon it. It is not practical to incorporate a test for this condition; it is necessary, therefore, that procedures be placed routinely at the beginning of any program.

"a" and "b" may be subscripted; they may also be indirect address references.

C. Temporary Storage

In order to avoid duplication of symbols between procedures, or between a procedure and the main program, the following provisions are made for temporary storage.

If a procedure needs (for example) 5 locations for temporary storage, they are reserved by placing the statement

```
stop'stop'stop'stop'stop''
```

(one "stop" for each location required) immediately before the "enter" statement. These locations are then called by the name of the procedure, with subscripts 1 to 5 (do not use zero).

D. Sample Procedure

To demonstrate these features, we include in section G a program for a simple plotting procedure. The minimum scale value ($y_o$) and range ($y_{max} - y_o$) are taken from locations plot'1' and plot'2', and the number of scale divisions from plot'3' (all in floating point). The number to be plotted is in the accumulator. The number of scale divisions is assumed to be in plot'4' as an integer and we assume that tab stops are set at every 16 characters.

This procedure is called from the main program, after the proper numbers have been placed in plot'1' thru plot'4' and an initial carriage return has been executed, by the statement

```
bring'a'call'plot'',
```

or, if the number to be plotted has been calculated as the result of the previous statement, simply

```
call'plot''.
```

The procedure is written so that points are plotted using the character "o" and off-scale values are indicated by a red "x".

E. A Library of Procedures

It would be desirable to keep commonly-used procedures in a library form. A separate tape of a procedure like the one in section G may be kept, if it ends with the word

wait'.

If this tape is to be used with a new program, it is compiled first. When the "wait" is read, the computer stops while the next tape is placed in the reader. Simply pressing "start" resumes compilation.

The problems involved in duplication of symbols can be avoided by using the conventions which have been suggested above, and which may be summarized here.

a. For temporary storage, use locations reserved by "stop" codes preceding the "enter" statement.

b. All dummy symbols and index symbols should consist of a numeric digit followed by the first four digits of the name (separated by spaces if the name contains less than 4 digits).

Duplication of statement numbers between procedures and between a procedure and the main program is automatically avoided by the compiler. When the "end" code is read, all statement number references compiled so far are satisfied and the statement dictionary is erased. If any statement numbers have been called but not defined, the special "type 7" error display occurs. Thus, no statement number may be called before an "end", which is not defined before the same "end". Computation will automatically start at the statement immediately following the last "end" statement compiled. Thus, it is recommended that all procedures be compiled first, before any portion of the main program is compiled.

If you need to have second, third, or other entries and/or exits to a procedure, the block of words preceding the "enter" statement may be used. Thus, if one of the "stop" words before the "enter" (see section C) is replaced by (for example) use 's5', where s5 is defined in the procedure, it may be referred to in the main program or in a later procedure by the name of the procedure and the appropriate subscript. If you desire to have an alternate exit set by "set" or "ret" in the main program, you must use the above format, where s5 again must be defined in the procedure, although in this case it is really a dummy. Do not attempt to use a "go to" code in this situation.

When you are debugging a procedure, the following steps must be followed if you wish statement numbers printed out while tracing. First, the procedure must be preceded by a "use" statement transferring to the main program following the procedure. Second, the "trace" statement must be inserted after the "enter" statement. Third, the "end" statement must be omitted. Fourth, the main program for the test run must not duplicate statement numbers used in the procedure. Finally, no other procedure may be compiled after the one being tested in this way.

Symbols used within a procedure will appear in the symbol table printed at the end of compilation.

F. Trace-Compiling.

The tracing feature is automatically suspended within a procedure. This is done because procedures will normally be debugged already.

If it is desired to trace through a procedure (for debugging purposes), the statement

```
trace''
```

is placed immediately after the "enter" statement. This restores the trace-compile feature (depending on the setting of the transfer control button).

G. Sample Procedure

```
     Plot Procedure'
     stop'stop'stop'stop'stop''
     enter'plot''
     0'unflo' ' 'prev'-'plot'1' 'x'plot'3'/'plot'2' ':'plot'5''
c    cr''
     if'prev'neg'sl''
     if' 'plot'4'i-'plot'5' 'neg's2''
s3'  set's5'to's6''
s8'   'plot'5'i/'16' 'reprt'tab6''
     remdr'reprt' ''
s5'  go to's0''
s6'  daprt'o''
s4'  cr'exit''
s2'  set's5'to'sl''
     plot'4':'plot'5''
     use's8''
sl'  daprt'color'x'color''
     use's4''
     end''
     wait'
```

# CHAPTER VIII

## DEBUGGING AT RUN-TIME

### A. Trace-Compiling.

If, during compile time, the transfer control button is down (on), provision is incorporated in the compiled program to make use of certain valuable tracing features. If only a portion of a program needs checking, the transfer control button may be pushed or released at any time during compilation. Blocks of coding beginning with an "enter" statement and ending with an "end" statement (see Chapter VII) are not trace-compiled regardless of the position of the transfer control button.

### B. Error Displays.

Indorporated into the standard operating subroutines are a number of tests on the validity of the data which they are handling. For example, if a floating-point subroutine generates a number with an exponent greater than 31, an error condition is set up. The occurrence of such errors, which can be detected at run-time, but not at compile-time, causes a legible error display to be printed out in the following format: carriage return, "e", space, error type (a numeric digit), space, subroutine name; carriage return, the number of the last numbered statement executed, the machine location (decimal) in the main (compiled) program, and the contents of the accumulator at entry to the subroutine (usually the right operand), interpreted first as an integer and second as a floating-point number.

The meaning of the error type is indicated in connection with each subroutine. The "last numbered statement" will be printed as zero if the program was <u>not</u> trace-compiled.

The subroutine name printed out in the name of the actual operation in the <u>source program</u> being executed. For example, a type 2 error in "pwr" (argument regative) gives the name "pwr" even though the error is actually detected in the logarithm subroutine. The only exception to this is in "rdflo", where the name "flo" will be given.

### C. Statement Number Stopping.

If your program is trace-compiled, and you enter it with the transfer control button down and manual input on the flexowriter, a read is called. Type in a + sign followed by a statement number, release the transfer control button, and press "start". The program will run without interruption until it stops just before executing the statement identified by the number you typed in. Pressing start now will cause it to ask for another statement number. If you wish it to run without stopping, type "run".

### D. Tracing.

If, during run-time, the transfer control button is depressed, and if your program was trace-compiled, each statement will be traced. After each statement is executed, the following print-out will occur: carriage return, statement number, machine address of first instruction of statement, and result of statement, interpreted first as an integer and then

as a floating-point number. The transfer control button may be depressed or released at any time to turn the trace print-out on or off.

Suppose, for example, you know your program is all right as far as statement 10, but you want to trace from there on. The procedure is then as follows: transfer to 300 with transfer control down. When the light glows, type in +10. Release transfer control, and press start compute. The program will run at full speed and stop before executing statement 10. Press start. When the light glows, type "run". Depress transfer control and press start compute. The first line of tracing is always meaningless; the second line will be the trace of statement 10, and subsequent statements will be traced until the transfer control button is released.

# CHAPTER IX

# STANDARD SUBROUTINES

"P-tapes"

## A. Subroutine Packages

Considerations of machine language coding have made it necessary to group the operating subroutines into "packages" which may not be broken up. An attempt has been made to formulate these groupings on the basis of related content.

These basic or minimum packages may in turn be joined into larger packages. The contents of the various packages are indicated in tables III-VIII.

## B. "Basic Package" P-0. (9 tracks)

The basic package P-0 contains all of the basic service routines, including the common input-output loops, trace and error display, and others needed by all the other subroutines.

## C. Floating-Point Package P-1. (16 tracks)

The package P-1 contains everything in P-0 plus another group labelled FP, including the basic floating-point arithmetic (+, -, x, /, abs) and the floating-point input-output (read, print, dprt, punch). Optionally available in P-1 is a random number generator.

The operation

randm'

which requires no operands, has as result a floating-point number from a pseudo-random sequence uniformly distributed over the range zero - one.

## D. Package P-2. (21 tracks)

The package P-2 contains everything in P-1 plus the procedure codes (see Chapter VII) and ix, i/, flo, unflo, fix, ipwr, rdflo.

These additional portions may be separated:

CS (1 track) procedure codes.
ICP (4 tracks) "integer compatibility package".

## E. PELL (4 tracks)

PELL contains pwr, exp, ln, log. The FP group must have been previously assembled, or contained in the R-tape.

## F. TRIG (3 tracks)

TRIG contains sin, cos. The FP group must have been previously assembled.

## G. ARTAN (2 tracks)

The FP group must have been previously assembled.

## H. SQRT (1 track)

The FP group must have been previously assembled.

# CHAPTER X

## MACHINE OPERATION

### A. Assembly of Subroutines. (See Flow Chart A)

<u>Load</u>: SPAR A

<u>Console</u>: breakpoint 4 off, 8 off, 16 off, 32 on.
6-bit off. transfer control off.

Read appropriate R-tape (see A.1.). After reading the first two words, the program initializes tables, then reads the remainder of the tape. At end of R-tape, the computer stops.

Depress the 6-bit button, prepare a symbolic subroutine tape in the reader, press start. At end of tape, the computer stops. Repeat for as many subroutine tapes as desired.

When all desired subroutines have been assembled, switch to the flexowriter (manual input), and press start compute three times. If an error indication occurs at this time, consult the writeup of the subroutines used, or the SPAR manual. If it is unexplained, it probably indicates a damaged tape or a misread.

Set up the punch (breakpoint 32 down for the high-speed punch, up for the flexowriter) and press start. The tape punched will be the P-tape (assembled subroutines in hex) of all the symbolic subroutines assembled. Keep this tape, run a leader and press start. This will produce the T-tape (operation tables in hex). Repeat this procedure to produce the T* -tape (see D).

### A.1. Use and Preparation of <u>R-tapes</u>.

Six R-tapes are provided with the system. These are R-0-A, R-0-B, R-1-A, R-1-B, R-2-A, and R-2-B. An R-1 or an R-2 tape must be used if any floating-point arithmetic is desired.

To prepare R-tapes corresponding to your own P-tapes, after completion of section A above load the tape SPAR B. Prepare the punch as above and press start. The desired R-tape will be punched.

### B. Compiling a Program. (See flow chart B)

<u>Load</u>: ACT III A

<u>Console</u>: breakpoint 4 off, 8 off, 16 off, 32 off
6-bit off. transfer control off.

Read appropriate T-tape. After reading the first two words, the program initializes tables, then reads the remainder of the T-tape and stops.

If you have source-language subroutines on seqarate tapes, they must be compiled now. Follow the instructions below for each such tape. A stop occurs at the end of each tape.

Prepare your source-language program tape in the reader, depress 6-bit button. Depress transfer control if you wish to trace-compile. Press start.

At the end of the program, the computer will type out the last location used by the compiled program (in decimal) and a statement dictionary, listing the location of the first instruction of each numbered statement. After the search of the statement dictionary, a carriage return is executed and the symbol table is printed. (See Supplement).

B.1. Punching a Compiled Program.

The compiled program may be punched out as a hex tape either immediately after compiling or after testing. Four separate methods are available.

A. After the statement dictionary is printed, prepare the punch as usual and press start. The resultant tape will be a single block of hex with one checksum.

B. After the statement dictionary is printed, release the 6-bit button and transfer control. Load the tape marked ACT III B. Set up the punch and press start. The resultant hex tape will be in blocks of two tracks, each with its own checksum.

C. After testing, reload ACT III A (not necessary in mode A), transfer to 5449, set up the punch and press start. The tape produced is described in (a) above.

D. After testing, follow the directions in (b) above.

In any of the above cases, punch .0000300' manually on the end of the program tape. (a) and (c) not available with standard system. .0000300' is punched automatically in cases (b) and (d) above.

C. Running a Program.

Load: The proper P-tape, followed by your program tape.

Console: breakpoints are required by your program, 6-bit off, transfer control off (except for debugging; see Chapter VIII).

Prepare data tape (if any) in reader, prepare punch if necessary, press start.

Note: Immediately after compiling, the compiled program is present on the drum. For testing, load the P-tape (if necessary), and transfer to 300 to start your program.

D. Mode A and Mode B Operation.

31-1/2 62

In mode A operation, the subroutines (P-tape) and compiler (ACT III A) are on the drum simultaneously. Thus it is not necessary to reload these tapes, unless you wish to use a different P-tape. Also, unless you have used ACT III B or written data in more than the first 320 words of the region "areg2", it is not necessary to read in the whole T-tape. In this case, use only the T*-tape for a second compilation. This mode is evidently most convenient for installations not having a photoelectric reader. However, the limitations on program size are rather severe.

In mode B operation the various tapes must be reloaded for each use; we strongly recommend the use of ACT III B for punching the compiled program in this case.

## PREPARING THE STANDARD SYSTEM FOR USE

A. <u>To Prepare P-0-A or P-0-B</u>.

Follow flow chart "A" using for R-tape RI-A or RI-B, and for symbolic tape, S-0. Mark P, T, and R tapes thus produced (P-0-A, T-0-A, T*-0-A, R-0-A) or (P-0-B, T-0-B, T*-0-B, R-0-B).

B. <u>To Prepare P-1-A or P-1-B</u>.

Follow (A) above, replacing 0 by 1 everywhere. (bottom)

C. <u>To Prepare P-2-A or P-2-B</u>.

Follow flow-chart "A" using for R-tape R-1-A or R-1-B, and for symbolic tape CS-ICP*. The T and R tapes thus produced may be appropriately marked. They <u>include</u> all previous material. The P-tape should be marked "P-2-A minus P-1-A" (or "P-2-B minus P-1-B").

To produce a complete P-2-A or P-2-B: load P-1-A (or P-1-B) and "P-2-A minus P-1-A" (or "P-2-B minus P-1-B"). (The word "load" means place tape in reader and execute the sequence "OCNS" as described in flow chart "A".) Now load ACT III B, transfer to 3000. When light glows, type in 14002963 (A) or 46006163 (B). Set up punch as usual, press "START COMPUTE". Punch ".0000000' " manually at end of this tape.

D. <u>To Prepare P-4-B</u>.

Follow chart "A" using for R-tape R-1-B, and for symbolic tapes CS-ICP, PELL, TRIG, ARTAN, SQRT. The T and R tapes thus produced may be appropriately marked. They include all previous material. The P-tape should be marked "P-4'B minus P-1-B."

To produce a complete P-4-B, load P-1-B and "P-4-B minus P-1-B", and punch out from 3100 to 6163 using 13.2 or equivalent. Punch ".0000000' " manually at end of this tape.

*NOTE: When assembling CS-ICP assembly will stop with the tape only partially processed. Push "START" and assembly will continue.

B. The assembly of S-1 will stop before the tape is completely processed. Push start to include "randm".

# CHAPTER XI

## THE OVERFLOW LOGIC MODIFICATIONS

The ACT III system has been set up to take advantage of the special logic board installed at Parma Research Center, Union Carbide Corporation; however, it has been so designed that it will run satisfactorily on a standard logic board. In this chapter we present the ways in which the special logic may be utilized. First, however, since there are in existence several versions of this logic modification, it is worthwhile to describe the facility in detail.

### A. Nature of the Modifications.

The occurrence of an arithmetic overflow in addition, subtraction, or division does not stop the machine, but sets an indicator (the sign bit of the C-register) and computation continues.

All Z instructions with a zero sign bit ("+Z") have the same meaning as on the standard logic board.

Z instructions with a one in the sign bit ("-") are changed as follows:

| | |
|---|---|
| Track address 00: | stop (unchanged) |
| Track address 01: | do nothing if overflow indicator off; skip next instruction and turn indicator off if it is on. |
| Track address 04,08,16,32: | do nothing if corresponding breakpoint switch is up (off); skip next instruction if it is down (on). |

### B. Overflow Logic and SPAR

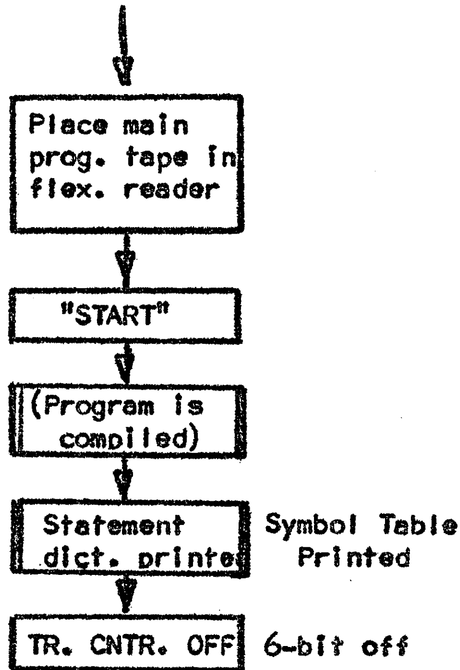
SPAR contains a test which determines whether the logic board on which an assembly is being performed is standard or modified. Special codes in SPAR language may be used so that certain instructions may be assembled only on one board, not on the other.

### C. Overflow Logic and the Operating Subroutines.

The symbolic tapes of the basic floating-point subroutines contain, in effect, two sets of coding, one of which is used if the tapes are assembled on a standard board, the other on a modified board.

The gains in the routines for the modified board involve decreased running times for the +, -, and / routines and improved accuracy in + and -.

None of the other routines have been coded in this dual manner.



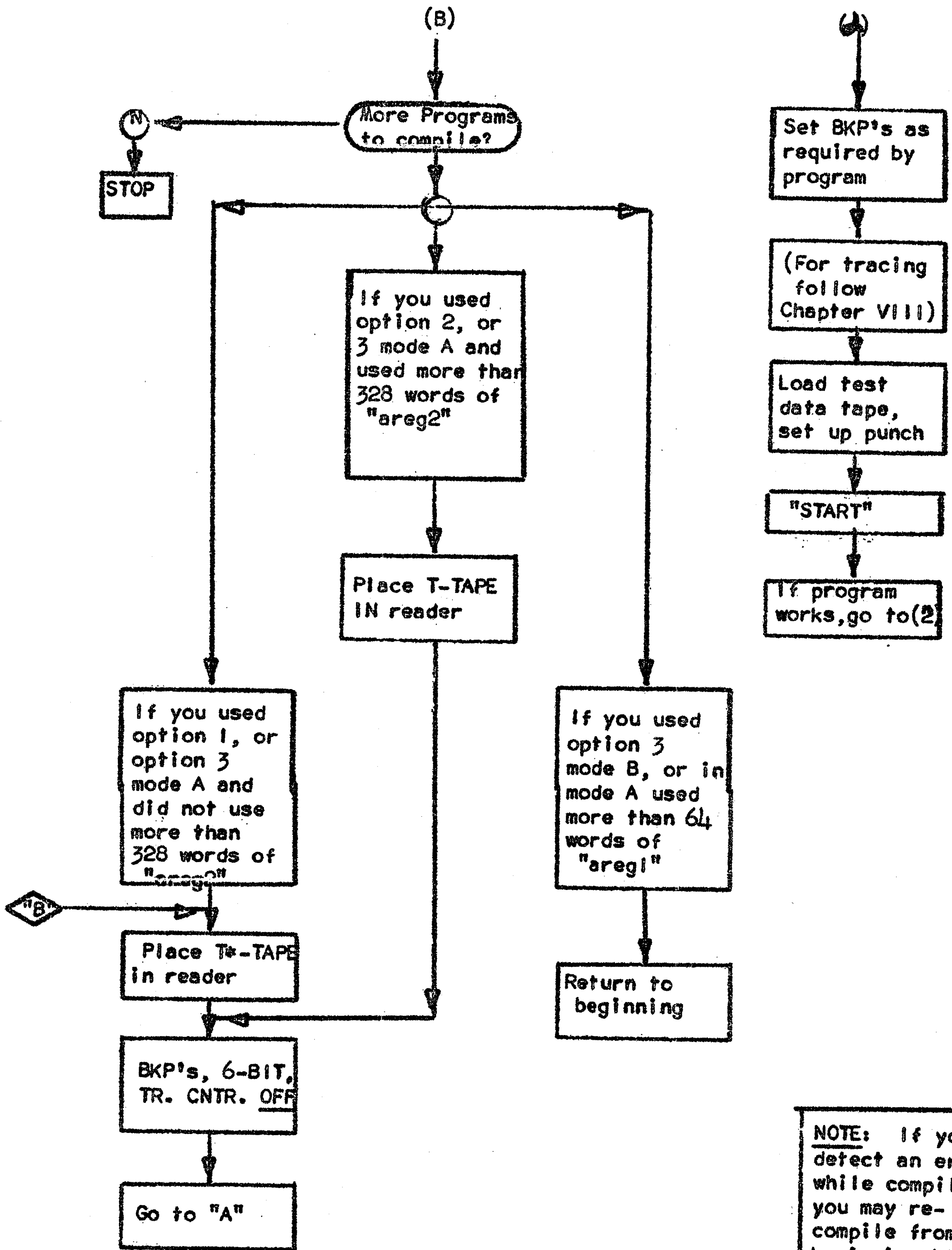

NOTE: Option 1 not available with tape ACT III A (S)

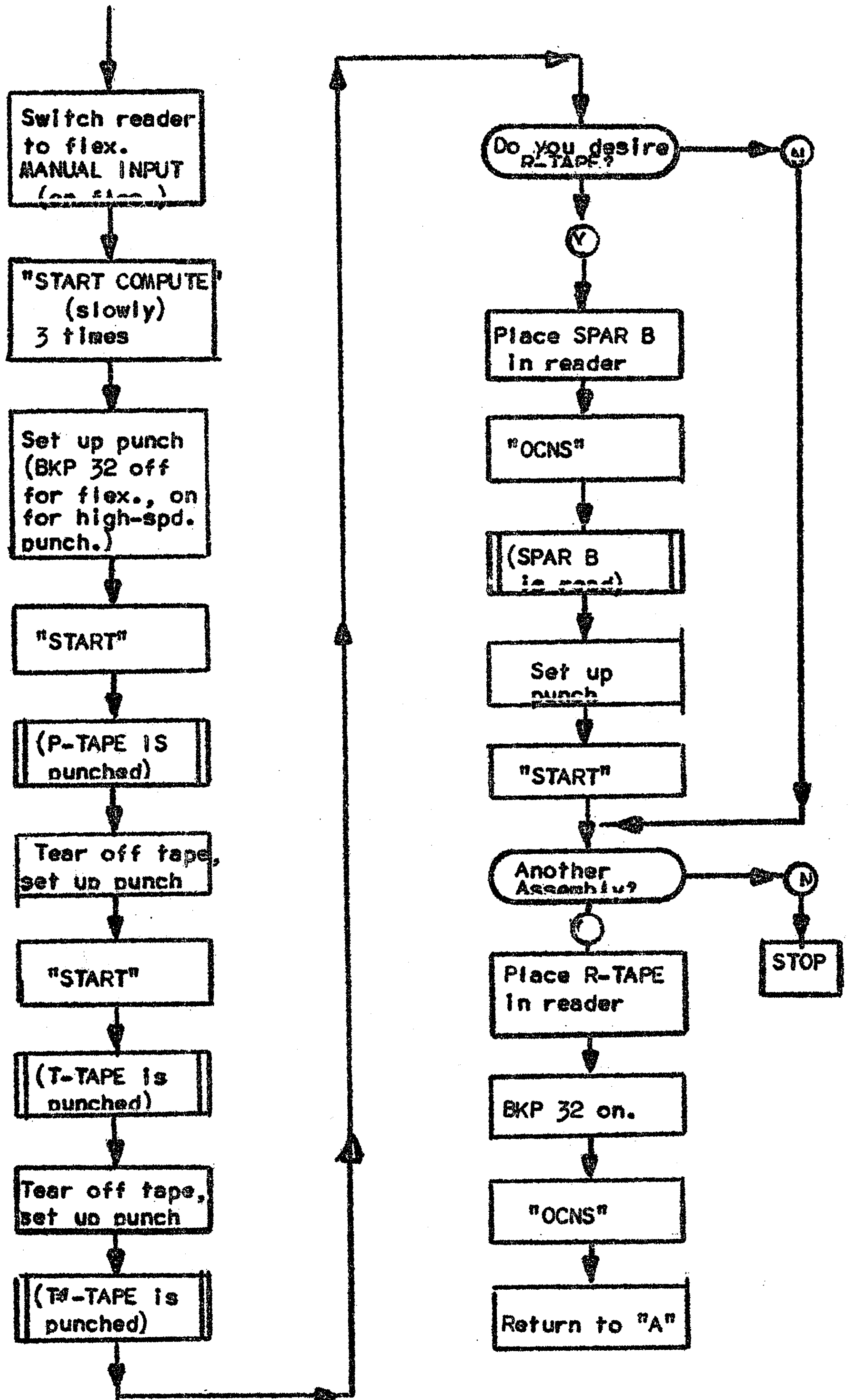
OPTIONS

① Set up punch
"START"
(Hex program is punched)
Punch ".0000300'" manually on hex tape at end of tape
File tape
(B)

② Place ACT IIIB in reader
"OCNS"
Set up Punch
"START"
Hex Program is punched

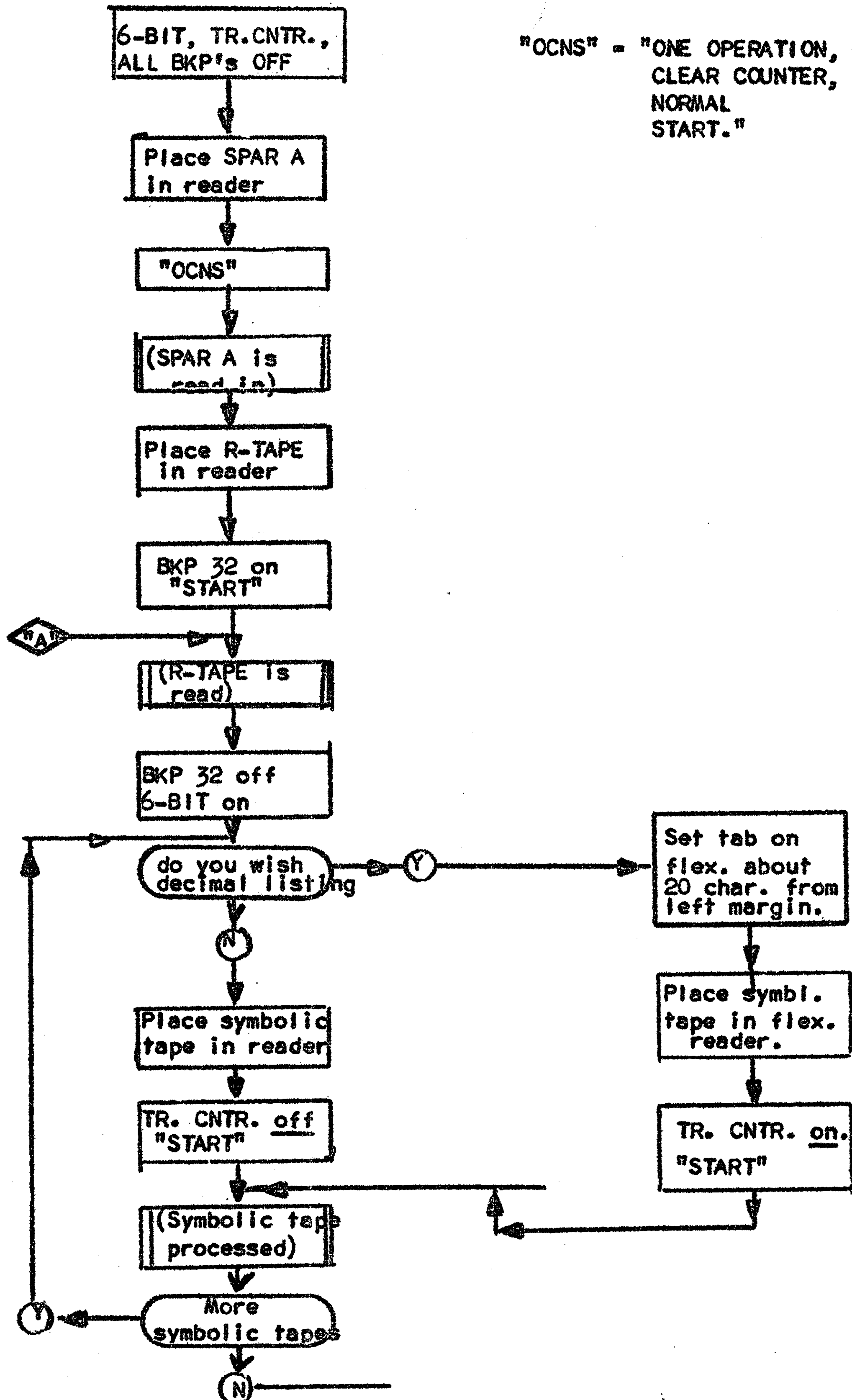
③ Place P-TAPE in reader
"OCNS"
(P-TAPE is read)
not necessary if P-TAPE on drum (mode A)
Switch to flex.; MAN. INPUT
"START"
Type ".0000300'"
"START"
(2)


(B)
More Programs to compile?
N
STOP
If you used option 2, or 3 mode A and used more than 328 words of "areg2"
Place T-TAPE IN reader
If you used option 1, or option 3 mode A and did not use more than 328 words of "areg2"
"B"
Place T*-TAPE in reader
BKP's, 6-BIT, TR. CNTR. OFF
Go to "A"
If you used option 3 mode B, or in mode A used more than 64 words of "areg1"
Return to beginning
(A)
Set BKP's as required by program
(For tracing follow Chapter VIII)
Load test data tape, set up punch
"START"
If program works, go to (2)
NOTE: If you detect an error while compiling, you may re-compile from the beginning by going to "B".



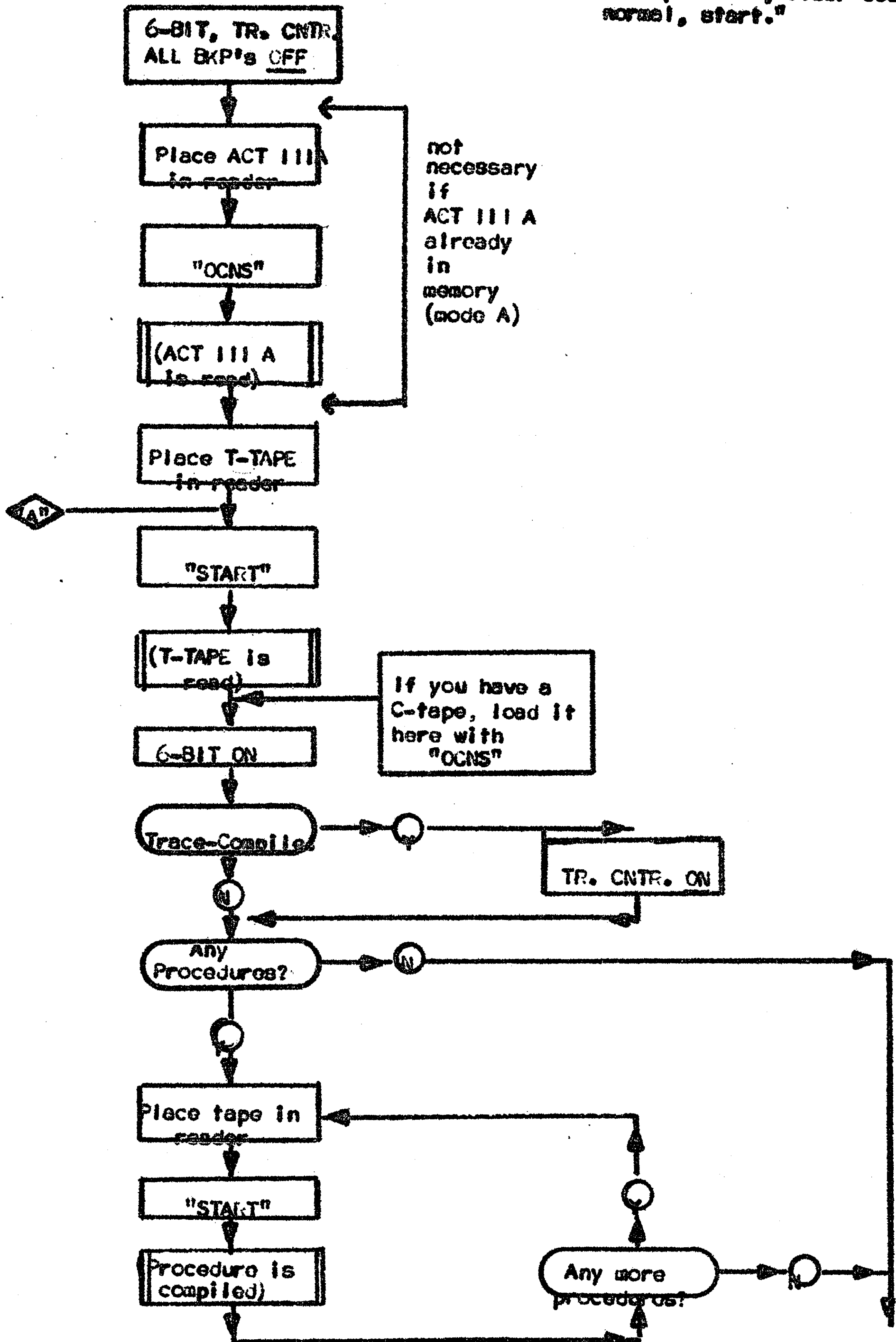
Switch reader to flex. MANUAL INPUT (on flex.)
"START COMPUTE" (slowly) 3 times
Set up punch (BKP 32 off for flex., on for high-spd. punch.)
"START"
(P-TAPE is punched)
Tear off tape, set up punch
"START"
(T-TAPE is punched)
Tear off tape, set up punch
(T-TAPE is punched)
Do you desire R-TAPE?
Y
N
Place SPAR B in reader
"OCNS"
(SPAR B is read)
Set up punch
"START"
Another Assembly?
N
STOP
Y
Place R-TAPE in reader
BKP 32 on.
"OCNS"
Return to "A"

FLOW CHART "A" - ASSEMBLY OF SUBROUTINES

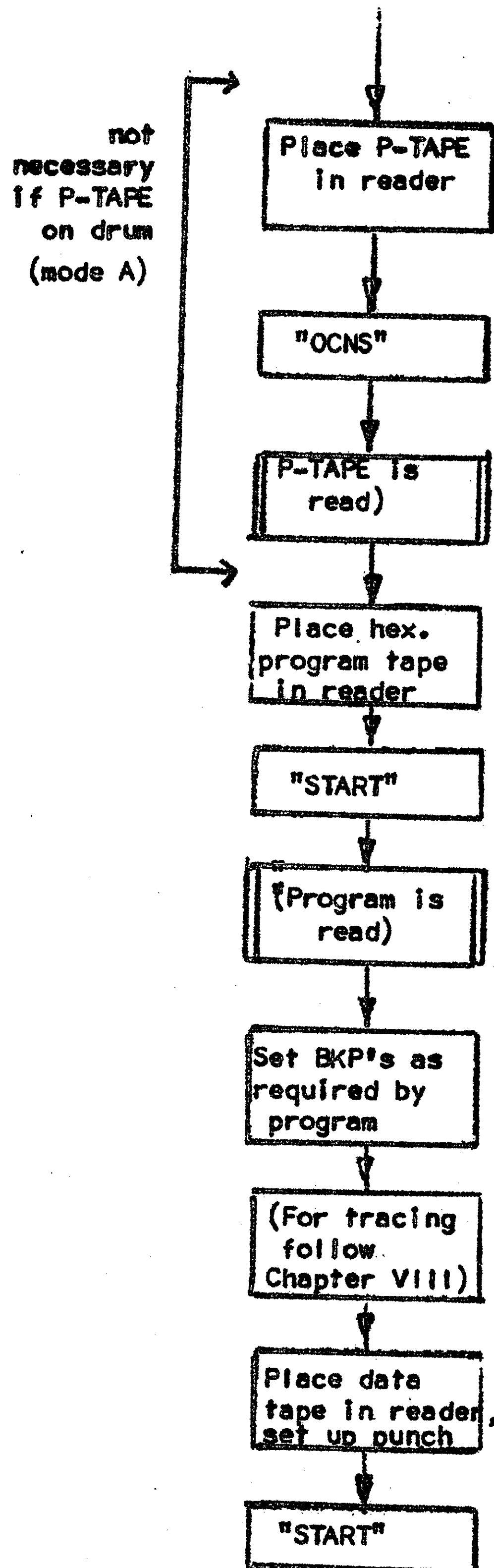


FLOW CHART "B" - COMPILING A PROGRAM

"OCNS" = "one operation, clear counter, normal, start."



FLOW CHART "C" - RUNNING A PREVIOUSLY COMPILED PROGRAM



PARMA RESEARCH LABORATORY

UNION CARBIDE CORPORATION

URM-21A
August 5, 1960

# Supplement to ACT III Compiler

by

H. J. Bowlden and R. R. Helms

## The Minimum System

This system is adequate for most requirements of installations possessing a photo-electric reader. It includes the following tapes:

A. ACT III A (S). This is the ACT III translation program, described in the attached sheets.

B. ACT III B. This is the program punch routine; its use is described in the manual.

C. ACT III C. This tape is used for partial compilations. See attached sheets.

D. P-4-B. This tape contains the subroutines necessary to implement all the codes described in the manual. It operates on the standard logic board.

E. T-4-B and T*-4-B. These are the corresponding operation table tapes needed for compiling programs which are to use P-4-B.

## Additional Tapes (available on request)

The following tapes are needed if you:

do not have a photo-reader, and desire to operate in Mode A;

or have an overflow logic board, and wish to take full advantage of it;

or need more space for data, and wfch to omit some of the subroutines;

or wich to add subroutines to the sy stem.

A. SPAR A. This is the symbolic assembler, as described in the manual.

B. SPAR B. The use of this tape is described in the manual.

C. R-I-A and R-I-B. These are the initializer tapes for Mode A and Mode B Assembly.

D. S-1. This is the symbolic tape containing the floating-point arithmetic, input-output, error and trace routines.

E. CS-ICP, PELL, TRIG, ARTAN, SQRT. These symbolic tapes are described in the manual.

F. R-4-B. This is the R-tape corresponding to P-4-B.

Also available, on special request, is the tape ACT III A (H), incorporating the hex-punch routine for option (1) of the flow chart B. This takes the place of the symbol table print, special error displays for types 2 and 7 errors, and restart features associated with these displays. Compilation speed is about 30 per cent slower than with ACT III A (S); the advantage for installations without photo-reader lies in the elimination of the necessity to load ACT III B and reload the T-tapes, when compiling short programs.

Also available is the tape S-0, the symbolic tape incorporating integer input-output, integer arithmetic (except ix and i/), error and trace. This may be of use if you do not wish to use the floating-point arithmetic, or if you wish to substitute some other arithmetic mode.

## Special Features of ACT III A (S)

This tape is a version of the ACT III A translator designed for increased compilation speed and improved facilities for handling partial programs and correction of certain source-program errors.

### Symbol Table Print-Out

After the printing of the statement dictionary (see flow chart B), a list of all symbols used is printed, with the machine language addresses assigned to each. In the case of regions, the address corresponds to word 0. Words 1, 2, etc., are found by subtracting 1, 2, etc., from this

address. Remember that these addresses consist of two digits for track and two digits for sector, and that each track contains 64 sectors. Thus, we have the peculiar arithmetic: 2400 - 1 = 2363.

Special Type 2 and 7 Error Display

In the case of type 2 (storage full) and type 7 (undefined statement number) errors, the usual error indication is not given. In its place, the following print-out occurs:

A. For type 2 errors only, a carriage return, and a simulated statement dictionary entry with statement number zero and address 0000..
   For type 7 errors only, a carriage return, the offending statement number, and the machine address of the instruction in which this statement is called. If more than one case occurs, they are all listed as they are found.

B. The f-address, statement dictionary and symbol table as given for successful completion of compilation.

Restarting Compilation After Type 2 or 7 Errors

If the necessary corrections can be made to the source program tape, or if you can stand by to catch the process and insert the corrections manually, the following procedure may be followed.

Roll the source tape back beyond the correction to the conditional stop code before a numbered statement. If a dimension statement is in the block thus contained, depress "manual input" on flexowriter, "start compute", and (at the read) type in the first symbol named in this dimension statement, and lift the "manual input" button. If no dimension statement is involved, simply press "start compute".

The compiler will reset and recompile over the previously compiled portion.

It should be noted that often if you are trace-compiling you may overcome a storage problem by compiling the last few statements without trace

(which takes up two words per statement). The above procedure makes this easy.

If you try to restart at a statement number not yet defined, a display like that for type 2 error will be repeated.

In the case of a type 7 error, the portion of the program which is not to be recompiled must not contain any references to statement numbers in the recompiled portion. This restriction does not apply to a type 2 error.

If it is necessary to get off the machine after a type 2 or 7 error display in order to correct your tape, the following process may be used to advantage if an appreciable part of the program does not need to be recompiled.

A. Load the tape ACT III C, set up the punch and press "start". The resulting tape contains a complete record of the compilation, as described under the description of ACT III C.

B. To recompile with this tape, load it as described under ACT III C. Then switch to flexowriter, "manual input", "OCNS". Type in .0005642 and press "start compute". Position your program tape at the conditional stop before the statement number and follow the remainder of the procedure described above.

<u>Partial Compilations (ACT III C)</u>

It is often desirable to use a large portion of a program (a long procedure, for example) as a common portion of many programs. The following method may be used to avoid the need for repeated compilation of this common portion.

Compile the common portion in the usual way. It should end with a "wait" code. Place the tape ACT III C in the reader, "OCNS" to load it. Prepare the punch and press "start". We shall refer to the resulting tape as a C-tape. It contains the compiled program segment, statement dictionary,

and symbol table in hex. After it is punched, you may obtain a printed statement dictionary and symbol table by disconnecting the punch and pressing "start compute".

To use the C-tape, initiate compilation in the usual way with the T-tape (or T*-tape). When this tape is in, place the C-tape in the reader, and "OCNS". After the C-tape has been read, place your program tape in the flexowriter reader and proceed as for standard compilations.

Note: A partial program must not go beyond location 2763.

A very frequently used C-tape may be duplicated onto the end of the T-tape by deleting the last code hg character punched on the T-tape.

10 72 2F 1F ' 0J ___ ___ ___ '

APR-1
Act III

# TABLE IX

## CODES FOR "aread"

| SYMBOL | CODE | SYMBOL | CODE |
|---|---|---|---|
| )0 | 04 | Aa | 72 |
| L1 | 0j | Bb | 0f |
| *2 | 14 | Cc | 6f |
| "3 | 1j | Dd | 2f |
| △4 | 24 | Ee | 4f |
| %5 | 2j | Ff | 54 |
| $6 | 34 | Gg | 5j |
| ¶7 | 3j | Hh | 62 |
| Σ8 | 44 | Ii | 22 |
| 9 | 4j | Jj | 64 |
| Space | 06 | Kk | 6j |
| _- | 0a | Ll | 0j |
| =+ | 16 | Mm | 3f |
| :; | 1a | Nn | 32 |
| ?/ | 26 | Oo | 46 |
| ]. | 2a | Pp | 42 |
| [, | 36 | Qq | 74 |
| TAB | 30 | Rr | 1f |
| Lower Case | 08 | Ss | 7f |
| Upper Case | 10 | Tt | 5f |
| Color Shift | 18 | Uu | 52 |
| Carr. Ret. | 20 | Vv | 3a |
| Back Space | 28 | Ww | 7j |
| ' | 40 | Xx | 8 |
| | | Yy | 12 |
| | | Zz | 02 |

I type
3a 9 J

ME
feel
one up
w/
mar
start