SYNTAX VERIFIER

Syntax Verifier is an interpretive routine for checking the format of input strings against a given definition. Further features permit it to do simple encoding and various housekeeping procedures associated with input verification.

The stored definition is a sequence of pseudo-instructions created to approximate Backus Normal Form (BNF) notation.*

Just as BNF is a defining language which operates on characters in a text string, the Syntax Verifier Executive Program operates on a character string. In order to group characters by functional class, however, and to avoid the long insertional statements made in many character manipulation programs, Syntax Verifier classifies all of the system's available input characters into two main classes - punctuational and arithmetic - each class containing 13 subgroups. All reference to this classification system (the Atom Table) are made via a 15-bit numerical representation; the 1's representing the "inclusive or" of the classifications which they describe. The punctuational class is selected by setting bit 3 to zero, the arithmetic by setting it to one. Note that a 15-bit representation of the logical "or" of any groups within a class may be formed. Thus, the number 4276Ø refers to left parenthesis, right parenthesis, left bracket, right bracket, and the two broken brackets which are the less than and greater than signs. (See the Atom Table Configuration)

*Backus, J.W., "The Syntax and Semantics of the Proposed International Language of the Zurich ACM-GAMM Conference", Intl. Conf. Proc., Paris, France, June, 1959.

Syntax Verifier is called by an IOT STV described below.

STV (731700)

The IOT STV is called with a character pointer in FSA pointing to the text which is to be verified (the input string) and with a word pointer in the AC to the Syntax definition, (expressed in pseudo-instructions). There are two returns from STV. Return 1, to the location immediately following the STV, is made if the input string failed the Syntax Verification. Return 2, the next location, is made if the string passed.

Registers of interest to the user are described below.

FSA	FSA is the register whose contents are a pointer to the character in the text string currently under test.
SUS	SUS is a register containing a 15-bit atom table definition. It is used in conjunction with certain overall string validation procedures to ignore those characters which are not important to the meaning being validated but are generally present for aesthetic or format purposes in the input string.
SUA	pseudo-instruction pointer
SUE*	push-down list maximum +1
SUG	current pointer to output buffer
SUP*	beginning pointer for push-down list
SUQ	beginning pointer to output buffer

*C(SUE) - C(SUP) must be divisible by 3, quotient is number of allowable nested "DIP" pseudo-instructions.

SUR	relocation register
SUT	multiplier for "ADD" pseudo-instruction

In discussing the pseudo-instructions of Syntax Verifier we speak of an input string as passing or failing the test which is made upon it. When the string passes, control is transferred to the "success" location, the second register following the STV command. The "failure" location is the one immediately following the STV. If the text string passes the test made upon it by a single pseudo-instruction, the pseudo-instruction pointer is moved to the next pseudo-instruction code. When an input text string fails to meet the requirements of the pseudo-instruction, Syntax Verifier enters its failure evaluation routine. This moves the pseudo-instruction pointer along the list of interpretive code, reading, but not interpreting instructions until it encounters a terminator (LAC) or an interpretive jmp (LAW) operation code.

Note that when a failure has occurred, FSA will be pointing to some arbitrary character within the text string. In the case of a success, FSA will be pointing to the first character in the string which has not been verified. If the entire string has been verified, FSA will be pointing to the EOM at the end.

THE ATOM TABLE

The use of the atom table has been described. Its configuration is shown in the table below:

ATOM TABLE CONFIGURATION

Bit No.	Value	CLASS: Punctuational	CLASS: Arithmetic
17	1	consonant (not y)	asterisk
16	2	digit	slash
15	4	carriage return-line feed	colon
14	1Ø	question mark	exclamation point
13	2Ø	space	left parenthesis
12	4Ø	hyphen	right parenthesis
11	1ØØ	plus sign	left bracket
1Ø	2ØØ	period	right bracket
9	4ØØ	vowel (a,e,i,o,u,y)	less than
8	1ØØØ	apostrophe	equal sign
7	2ØØØ	comma	greater than
6	4ØØØ	semicolon	all 12-bit characters
5	1ØØØØ	quotation	special*
4	2ØØØØ	any arithmetic character	any punctuation character

*This bit is used for all remaining characters, in particular:

number sign
dollar sign
percent sign
ampersand or and
at (for each) sign or VEL

THE OUTPUT BUFFER

The pseudo-instructions, DCH and IDX deposit information into an output buffer specified by the programmer. The address of first location of the buffer is designated by the contents of SUQ. SUG the current buffer address register, is referenced and appropriately incremented by DCH and IDX. When the buffer is used by a JMP subroutine, SUG must be incremented in the coding.

When a definition or sub-definition fails SUG is reset to the value it had before the definition or sub-definition was called. Output from a failure is thus automatically suppressed.

The output buffer is useful in tracing a complex definition, saving parts as they are verified for future manipulation. Since the main user program may refer to this buffer, it is clearly useful as a guide from branching.

THE SYNTAX VERIFIER MACRO

A macro is available for setting up registers SUQ, SUP and SUE, defining the Syntax Verifier output buffer and push-down list. It defines the symbol "OPB" as the beginning of the output buffer created. It is called as follows:

SYNTAXVERIFIER LOB, PDL

where LOB = length of the output buffer
and PDL = the number of push-down levels to be provided for.

The macro sets aside LOB + 3 "T" PDL registers whenever it is called.

PSEUDO-INSTRUCTIONS

The Syntax Verifier pseudo-instructions are not IOT's; they are registers which are interpreted by a portion of the Executive core other than the basic dispatch table. Each pseudo-instruction consists of an operation code and an address which represents a core address or a mask according to the operation code with which it is associated.

Any memory reference pseudo-instruction may specify indirect reference and/or relocation. Relocation is performed at every level of an indirect chain. Those operation codes which may specify indirect reference and/or relocation are starred in the pseudo-instruction list in Appendix 1. Bit 5 set to one (symbolic I) specifies indirect reference. Bit 4 specifies relocation (AND), adding the contents of the relocation register (SUR) to get the effective address. For example, XCT AND 3 when C(SUR) = DEF means XCT DEF+3. If DEF+3 contains I SUBDEF and SUBDEF contains AND 6, then XCT I AND 3 is equivalent to XCT DEF+6. The steps involved are as follows:

1) Relocate: XCT I DEF+3
2) Indirect (2 levels) XCT I SUBDEF: XCT AND 6
3) Relocate: XCT DEF+6

The pseudo-instructions are described below.

Pseudo-instructions which do a character-by-character check refer to the mask in register SUS and skip characters according to its value. The usefulness of SUS is illustrated by the following example, in which the validating string is QED⊙.

Setting SUS to 22Ø permits the input string to be any of those listed below, permitting the user to include space and period for format purposes.

QED
Q E D
Q.E.D.
Q. E. D.

Note also that this syntax definition accepts the following input string.

...Q...E .D . .

OPERATION CODE Ø (bits Ø,1,2)

This pseudo-instruction checks only one character in a string, and is the most elementary Syntax Verifier instruction. Bits 3-17 of the pseudo-instruction represent an atom table definition. When this pseudo-instruction occurs, the next character of the input string is looked up in the atom table. The bit found there is compared with the bit pattern in bits 4 through 17 of the pseudo-instruction; the class bit is checked against bit 3. If at least one bit is found set in the same position in both words, the character has passed the test. If there is no match, the character has failed.

When a character passes, the text pointer (FSA) is incremented one character. When a character has none of the indicated traits, subsequent pseudo-instructions are skipped as described in the section on the failure evaluation routine. When a pseudo-instruction of all zeroes is encountered, Syntax Verifier makes a special check to see if the next character is an EOM. If not, the action is the same as for character failure. FSA is never incremented past the EOM.

OPERATION CODE 5 Symbol - SAD

The SAD pseudo-instruction is equivalent to indefinite repetition of the Ø operation code until a failure occurs. On failure, with SAD, FSA remains pointing to the character which failed and the next pseudo-instruction in sequence is interpreted. The failure evaluation routine is never called by this pseudo-instruction. The use of this pseudo-instruction is illustrated in the definition for an integer given below:

```
INT,    SAD 2
        LAC 1
```

SAD 2 assembles a bit in position 16 of the pseudo-instruction and skips characters in the input string until it finds a character whose description in class Ø of the atom table does not have a bit in position 16 (i.e., is not a digit).

OPERATION CODE 1 (bits Ø,1,2) Symbol - XCT

XCT (operation code 1) effects a character-by-character check of an input string, for identity with a specific string of characters stored in core.

Bits 4 and 5 of this pseudo-instruction have the effect as described in the section on indirecting and relocation. The argument of XCT is a pointer to the stored text. The top two bits of the eventual pointer are used as the byte pointer. If the indirect bit is not set in the pseudo-instruction itself, the character string will be assumed to start in the leftmost third of the word. Relocation never takes place on a byte level, always a word level.

As FSA is incremented to a character it is looked up in the atom table and its class and group bits are compared with the bit pattern in register SUS. If a match is found, FSA is incremented one character for further validation. An EOM terminates the defining string and does not require existence of a corresponding EOM in the input string. XCT (CHARAC L#) is a no-op. If any character to be skipped (SUS) appears in the defining string, failure is assured. An identity check applies to a portion of the input string which may be the whole string. If all of the characters of defining text (except EOM) are found in the same sequence in the input string, the test is passed. At that time, interpretive control passes to the next pseudo-instruction with FSA set to the next character of the input string. If the test fails (i.e., if any character except the EOM in the defining string does not appear in the input string), Syntax Verifier enters its failure evaluation routine.

OPERATION CODE 3 Symbol - DIP

The DIP pseudo-instruction does not directly affect success or failure status nor the input string pointer. It acts as a second or nth call to Syntax Verifier for the purpose of writing sub-definitions. When a DIP is interpreted, the address portion points to the location in memory of the sub-definition which is to be checked. (as the AC when an STV is executed). If the input string passes the sub-definition, interpretive control is returned to the location following the DIP. In case of failure of the sub-definition as a whole, the DIP is treated as a simple pseudo-instruction which caused failure, and the failure evaluation routine is entered on the same level as the DIP. If the input string passes,

the input pointer setting is determined by the sub-definition. For example, if we wish to define an expression as a left parenthesis followed by an integer followed by a right parenthesis we may utilize the previous definition of an integer as below:

```
EXP,    Ø4ØØ2Ø    /left paren atom table definition
        DIP INT    /integer sub-definition
        Ø4ØØ4Ø     /right paren atom table definition
```

The fact that DIP's may be used recursively permits verification of input in the form of a list of elements. For example, a definition which will accept a list of integers separated by commas, (e.g., 1,2,3,4) is given below:

```
DEF,    DIP  INT /integer sub-definition
        ØØ2ØØØ   /comma atom table definition
        DIP  DEF
        LAC
        DIP  INT
        LAC 1
```

When the comma test fails, Syntax Verifier will proceed to the LAC in fail mode, with FSA pointing to the final integer. The integer sub-definition is the next alternative and the definition terminates with FSA pointing to an EOM.

OPERATION CODE 2 Symbol - LAC

Operation Code 2 characterizes a set of pseudo-instructions which act as terminators. Individual characteristics are represented by bits 15-17 of the pseudo-instruction, thus permitting LAC through LAC+7 as meaningful terminators.

LAC 2ØØØØØ

This operation code literally means "or". If a LAC is encountered by Syntax Verifier in normal interpretation the input string has passed all tests and is, therefore, valid under the BNF definition or sub-definition. If it is encountered in a sub-definition, it causes the DIP that called the sub-definition to have effectively succeeded.

Note that when alternative entries are such that one acceptable string is, in effect, a prefix of a longer one, the longer must precede it in the definition or sub-definition if this is not done, the longer string will pass the test, but FSA will not be pointing to an EOM. This is not a problem with the IDX pseudo-instruction, described later, which searches for the longest string.

FSA and SUG are left as they were before the LAC was executed and control is transferred to the instruction after the DIP (exception, see OPCODE 4 - ADD). If the LAC is encountered in the main definition, a special check is made to see that FSA points at an EOM. It if does, Syntax Verifier gives return 2 (success). If FSA points at a character which is not an EOM, the LAC is treated as if it were encountered by the failure evaluation routine. When encountered by the failure evaluation routine, the LAC command indicates that alternate definitions follow. FSA and SUG are reinitialized to their contents at the beginning of this sub-definition. If this is the main definition they are set to their initial values.

LAC 1 2ØØØØ1

LAC 1 indicates the end of a set of Syntax Verifier definitions separated by LAC's (or). If it is encountered in the main definition on the failure evaluation routine, it immediately causes a return to the failure location following the STV call. If encountered in a sub-definition by the failure evaluation routine, it causes the DIP which called the sub-definition to have effectively failed. If encountered during sequential interpretation it behaves precisely the same as LAC. (N.B. if in main definition FSA doesn't point to an EOM, the LAC 1 is treated as if it were encountered by the failure evaluation routine - see immediately above).

LAC 2, 3

Whenever bit 16 is set in a LAC instruction which is encountered in normal interpretation, it causes the LAC to make Syntax Verifier give return 2 without further checks. In the main definition, LAC 2 is equivalent to LAC Ø except LAC 2 suppresses the EOM check. In any sub-definition, LAC 2 causes immediate return 2 rather than return to the register after the DIP. The failure evaluation routine ignores bit 15 in LAC's. LAC 3 is clearly a LAC 1 with bit 16 set.

A program could be written as follows:

```
SEX,    XCT (FLEXO M##)
        LAC 2
        XCT (FLEXO F##)
        LAC 3
```

Examples of acceptable input strings:

"M" "M." "MALE" "MUMBLE"
"F" "F". "FEMALE" "FOO"

The remainder of the string may contain anything.

LAC 5 2ØØØØ5

LAC 5 is equivalent to:

```
LAC
LAC 1
```

LAC 7 2ØØØØ7

LAC 7 is equivalent to:

```
LAC 2
LAC 3
```

OPERATION CODE 44 Symbol - IDX

The low-order 14 bits of the IDX pseudo-instruction are interpreted in the same way as with the XCT command. IDX is used to compare an input string against entries and sub-entries in a dictionary, returning with an indication of the definition. IDX to a certain extent, thus, acts as a semantic verifier.

The address portion of the pseudo-instruction points, directly or indirectly, to the beginning of the dictionary. The IDX dictionary consists of a series of delimited text strings grouped together by some attribute which is important to be stored. Individual strings are delimited by single end-of-message terminators, groups by double end-of-message terminators, and the entire dictionary

by three end-of-message terminators. As in,

TEXT /M#MALE##F#FEMALE###/

The test provided by IDX results in the number of the group to which an input string belongs, a counter being indexed at every double end-of-message. The whole string is searched and the longest accepted string taken. The contents of the counter when membership is detected in a group (equal to one less than the ordinal number of the group in sequence) is left in the currently available output buffer register and may be used for further encoding purposes. The output buffer pointer (SUG) is incremented. If there is no entry which is acceptable within any group, the IDX fails, and the failure evaluation routine is entered.

The IDX command is interpreted under control of the mask in register SUS in exactly the same fashion and with exactly the same restrictions as the XCT command.

OPERATION CODE 14 Symbol - DCH

DCH saves the contents of the register specified in the address part of the pseudo-instruction. The contents are stored in the output buffer and the output buffer pointer (SUG) is incremented.

OPERATION CODE 6 Symbol - JMP

The JMP pseudo-instruction permits the programmer to call external, non-interpretive subroutines. The location to which control is to be transferred is specified directly or indirectly by the address portion.

The coding to which the command points is actual machine coding rather than Syntax Verifier pseudo code. This coding

may contain Executive IOT's. The facility for using Executive IOT's, including file access IOT's, means that the JMP command may be used to extend the power of Syntax Verifier beyond the detection of pure syntactic errors to the detection of file-dependent and moderately simple semantic errors. When JMP is executed the most recent machine language AC and IO are restored. Note that the restriction of common space means that the JMP subroutine cannot include a call to Syntax Verifier and, thus, cannot be used to test Syntax Verifier definitions unless the user has decided he has no use for the original call to STV. The coding entered via a JMP instruction is legally terminated by either of the IOT's described below.

STVW - 732000 (Syntax Verifier "win")

STVW returns control to the Syntax Verifier routine at the pseudo-instruction following the JMP command. The JMP is considered to be a Syntax Verifier pseudo-instruction and the input text string is considered to have been accepted by the JMP. The pointers within the Syntax Verifier program have not been changed by this action unless the machine coding specifically has modified them (register FSA should be given close attention here).

STVL - 732100 (Syntax Verifier "lose")

STVL returns control to the Syntax Verifier routine at the level of the JMP pseudo-command and then Syntax Verifier enters its failure evaluation routine in the same manner as for failure of an input string to pass a test.

OPERATION CODE 7 Symbol - LAW

The LAW pseudo-instruction unconditionally transfers interpretive control to the core location indicated by the address part. It should be used with caution because of the risk of getting into infinite loops. It may be useful as a definition shortener if the last several instructions of two or more definitions are identical. It can be invaluable when patching a definition.

Note that LAW transfers control from pseudo-instruction coding to other pseudo-instruction coding; JMP transfers to external coding.

OPERATION CODE 74 Symbol - SPO

SPO behaves precisely like LAW when encountered in normal interpretation. To the failure evaluation routine, however, it appears like a LAC Ø. Thus interpretive control transfer may be made a function of the results of the string check.

OPERATION CODE 4 Symbol - ADD

The ADD pseudo-command permits denumeration, an operation which is extremely awkward using normal BNF notation. The address part of the pseudo-instruction is taken to be the number of times the pseudo-instruction immediately following is to be interpreted. In effect, it is the same as rewriting the instruction that number of times and, therefore, is validly used only with those instructions which may logically succeed themselves such as Ø, SAD, JMP, etc. An example of the use of ADD is given in the BNF definition -

<UNI> : : = <DIG> <DIG> <DIG> <DIG> <DIG> <DIG> <DIG>

This statement defines <UNI> as being a succession of seven digits. The same definition may be written for the Syntax Verifier program utilizing ADD as:

```
UNI,    ADD 7
        2
        LAC 1
```

This definition is precisely equivalent to 7 2's in a row followed by a LAC 1 and expressly equivalent to the BNF definition. Since many of the tests for legality in input format with which we are dealing are denumeration tests, this pseudo-instruction provides a real saving in programming time and space.

The maximum legal multiplier is 3777 or 2047.

It is legal to follow ADD with a DIP command. This causes the DIP to be re-executed as many times as specified. When the sub-definition completes each time except the last time, control is transferred back to the DIP rather than the instruction following it. The current number of execution is saved on the pushdown list, so ADD's are legal even within a sub-definition.

APPENDIX A

EXAMPLE 1

The following definition uses the IDX instruction to check against a dictionary of acceptable input. The user may add to the dictionary, signifying his intention by typing a plus sign before his input string. An asterisk must follow the triple EOM which ends an IDX dictionary, and a dictionary entry may not contain an asterisk.

```
DEF,        SAD 2Ø
            1ØØ             /PLUS?
            SAD 2Ø
            IDX DICT
            SPO LOSE        /IF INPUT ALREADY IN DICTIONARY

            SAD 2Ø
            1ØØ             /DID IT LOSE AT PLUS OR DIP
            SAD 2Ø          /DIP, THEN ADD IT
            DCH FSA         /SAVE PTR. TO INPUT
            JMP ADDON
            LAC             /DO NOT ADD
            SAD 2Ø
            IDX DICT        /CHECK IF O.K.
            LAC 1

ADDON,      LAW DICT        /TO FIND END OF DICTIONARY
            DAC POINTER
            LCH I POINTER
            SAS (12ØØØØ     /ASTERISK
            JMP ADDON
            LAW I 1         /UNSTEP PTR. TWICE
            ADD POINTER
```

```
                IDC
                DAC POINTER
                LCH I OPB       /ADD TEXT
                DCH I POINTER
                SAS (74
                JMP .-3
                LAC (747412     /END OF TEXT, SET UP
                DCH I POINTER   /DICTIONARY END
                DCH I POINTER
                DCH I POINTER
                STVW

LOSE,           JMP TPTHR
                LAC 1

TPTHR,          LAW THERE
                TOS
                STVL

THERE,          TEXT /ALREADY IN DICTIONARY #/
```

EXAMPLE 2

This definition accepts as input the string "ALL", a string of the form XX9 where X represents any letter and 9 any digit, or a list of such strings separated by commas and terminated by an EOM. Plain EOM is also acceptable. Note that it is repetitive without using DIP.

```
DEF,        SAD 2Ø
            XCT ALL          /CHECK FOR ALL
            DCH (Ø           /SAVE INDICATOR
            LAC

LIST,       SAD 2Ø
            DCH FSA          /SAVE FOR REFERENCE
            ADD 2
            4Ø1              /2 LETTERS
            2                /DIGIT
            DIP COM          /COMMA?
            SPO LIST         /IF COMMA, GET LIST
            LAC 1

COM,        Ø                /EOM?
            LAC 2            /WIN UNCONDITIONALLY
            2ØØØ             /COMMA CHECK
            LAC 1

ALL,        TEXT /ALL#/
```

EXAMPLE 3

The following example accepts as legal input, a date in one of the following forms:

1/1/1966
1/1/66
1/1
"T" for today
"Y1" for yesterday
"Y2" for day before yesterday, and so on.

Since Exec III allows the user to decode date without the use of syntax verifier, this example is somewhat contrived. It was chosen because "DATE" is a common input and because readers are familiar with its syntax.

```
DATDEF,   SAD 2Ø
          DIP DAT1
          SAD 2Ø
          LAC 1

DAT1,     Ø                  /demand an EOM
          JMP DAT2           /EOM for "TODAY"
          LAC
          JMP DAT3           /initialize 5-register block
          XCT (FLEXO T#X     /"T" for "TODAY", no argument allowed.
          LAC
          XCT (FLEXO Y#X     /"Y" for "YESTERDAY"
          JMP DAT4           /get ARG to "Y"
          LAC
          JMP DAT5           /month
          4ØØØ2              /slash
          JMP NUMBER         /day
          DIP DAT6           /slash and year or else nothing
          JMP DAT7           /validate decoded date
          LAC 1
```

```
DAT2,     LAC (FLEXO T#X
          TYO                 /print "T"
          DCH I FSA           /replace "EOM" with "T-EOM"
          DCH I FSA
          STVL                /N.B. Treat as ordinary "T"

DAT3,     GTD
          DAC DATE
          TDNUM
          OPB
          STVW

DAT4,     DNM+1Ø
          STVL
          SPQ                 /"YO" illegal
          STVL
          CMA
          ADD DATE
          DAC DATE
          STVW

DAT5,     LAW OPB 2           /start output buffer ptr. to month
          DAC SUG

NUMBER,   DNM+14
          STVL
          DAC I SUG
          IDX SUG
          STVW

DAT6,     4ØØØ2               /slash
          JMP NUMBER
          LAC 5
```

```
DAT7,     LAW I 1ØØ.
          ADD OPB 4
          SPA
          ADD (19ØØ.          /supply century if typist didn't
          ADD (1ØØ.
          DAC OPB 4
          LAW OPB
          DCDTD               /B.C. DCDTD (decode time and date)
          STVL
          DAC DATE
          STVW

DATE,     Ø                   /definition leaves date here.
```

SYNTAX VERIFIER, PUT, GET, AND BGI 11-18-66 (STV,27)

```
/SYNTAX VERIFIER VARIABLES
SUA=BTEM+6
SUD=BTEM+7
SUE=BTEM+10
SUF=BTEM+11
SUG=BTEM+12
SUK=BTEM+13
SUP=BTEM+14
SUQ=BTEM+15
SUR=BTEM+16
SUS=BTEM+17
SUT=BTEM+20
```

```
/STV IOT AND SUBR
STV,      LAW IR1
.STV,     LIO I (USERAC)
.STVJ,    DIO D
          DAP I (BTEM)
          LAC I (USERPC)
          DAC I (BTEM+1)
          LAC I (FSA)
          DAC G        /TEXT PTR
          DAC I (SUD)
          LAC BO
          DAC I (SUT) /MULTIPLIER
          LAC I (SUQ)
          DAC I (SUK)
          DAC C        /OUTPUT PTR
          LAC I (SUP)
          DAC I (SUF) /PUSHDOWN PTR

/MAIN LOOP FOR INTERPRETATION
STVML,    LIO I D      /GET PSEUDO INSTR
          ISP I (SUT)
          IDX D        /IDX PC CONTINGENT ON MULTIPLIER
          DIO E       /SAVE CURRENT PSEUDO INSTR
          CLA
          RCL 3S
          ADD (LAC STVDTB)
          DAC A
B2,       XCT A        /CONSTANT 100000. DISPATCH LOOKUP
          SZL I
          IDA          /ADD 1 IF IN WIN MODE
          DAC A
          CLA"U"SWP"U"CLF 7          /CLEAR ALL FLAGS EXCEPT LINK
          RCL 3S
          IFI          /SET FLAGS 4,5,6 FROM BITS 3,4,5
          LIO E
          JMP A        /DISPATCH JMP

/IDX OPCODE
STV44,    DZM I (BTEM+5) /GROUP COUNTER
          JSP STVTRA  /GET TEXT PTR INTO E
          LAC G
          DAC I (BTEM+4) /SPACE FOR RESTORING
          DAC I (BTEM+2) /FARTHEST SUCCESSFUL SO FAR
STV4A,    JSP STVLE   /LCH I E; DAC A; SKIP ON NOT EOM
          JMP STV4C   /EOM; SUCCEEDED
STV4A1,   LAC G
          SAD I (BTEM+2)
          STF 1        /THIS WILL BE A NEW RECORD IF IT WINS
          JSP ATM     /LCH I G; DAC F
          LAC A
          SAD F
          JMP STV4A   /CHAR.S AGREE, LOOP
STV4B,    JSP STVLE   /FAILURE, WAIT FOR EOM
          JMP STV4D   /FOUND, TRY NEXT CASE
          JMP STV4B
```

```
STV4C,    SZF I 1      /SUCCESS; IS THIS A NEW RECORS?
          JMP STV4D    /NO, IGNORE
          STF 2        /SET WIN FLAG
          LAC G
          DAC I (BTEM+2) /REMEMBER HIS TEXT PTR
          LAC I (BTEM+5)
          DAC I (BTEM+3) /REMEMBER HIS GROUP
STV4D,    LAC I (BTEM+4) /ROUTINE TO TRY NEXT CASE
          DAC G
          CLF 1
          JSP STVLE
          JMP STV4D1   /2 EOM'S IN A ROW; NEW GROUP
          JMP STV4A1   /GO INTO CHAR. LOOP
STV4D1,   IDX I (BTEM+5)
          JSP STVLE
          JMP STV4E    /3 EOM'S IN A ROW; FINISHED
          JMP STV4A1

STV4E,    SZF I 2      /WAS THERE A WINNER?
          JMP STVLOS
          LAC I (BTEM+2) /YES, REMEMBER HIS PTR
          DAC G
          LAC I (BTEM+3) /SAVE HIS GROUP #
STV14,    DAC I C
          IDX C
STV4,     JMP STVML    /ALL DONE; ALSO LOSE MODE
          SZF 4
          JMP STV44    /IDX OPCODE
          LAI          /?
          SCM"U"IDA    /ADD OPCODE, CALCULATE MULTIPLIER
          DAC I (SUT)
STV0,     JMP STVML
          JSP ATM
          JSP REATM    /CHECK CHAR. AGAINST ATOM TABLE
          JMP STVML    /OK
STVLOS,   CLL"U"CML    /NG, ENTER LOSE MODE
          LAC BO
          DAC I (SUT) /CLOBBER MULTIPLIER
STV5,     JMP STVML
STV5L,    JSP ATM
          JSP REATM    /CHECK CHAR. AGAINST ATOM TABLE
          JMP STV5L    /OK, KEEP CYCLING
STV5UL,   LAC I (SUD)
          IDC
          DAC E
STV5UM,   LAW I 1
          ADD G
          IDC
          IDC
          DAC G
          IDC
          SAD E
          JMP STVML
          LCH G
          SAD ML6
          JMP STV5UM
```

```
STV1,     JMP STVML
          JSP STVTRA  /GET TEXT PTR
          LAC G
          DAC I (FSA) /UPDATE FSA SO IT CAN BE REFERRED TO
          LAC I E     /GET ARG
          SZF 4
          JMP STV14   /DCH OPCODE; SAVE ARG
STV1L,    JSP STVLE
           JMP STVML  /EOM, XCT OPCODE SUCCEEDED
          JSP ATM
          LAC A
          SAD F
          JMP STV1L   /KEEP INSISTING ON EXACT MATCH
          JMP STVLOS  /NO MATCH, FAIL

STV2,     JMP STV2L   /LAC IN LOSE MODE
          LAW 2       /LAC IN WIN MODE
          AND E
          SZA
          JMP STV2SP  /SUPER POP
STVPOP,   LAC I (SUF) /NORMAL POP
          SAD I (SUP)
          JMP STVNPP  /TOP LEVEL, CAN'T POP
          SUB C3
          DAC I (SUF) /UPDATE PUSHDOWN PTR
          DAC A       /TEMPORARY PUSHDOWN PTR
          LAC I A
          DAC I (SUD) /POP SUD
          IDX A
          LAC I A
          CLI         /PREPARE TO UNPACK POP SUT;SUK; PSEUDO PC
          SCR 6S
          SZL
          CLA         /NO MULTIPLIER IN LOSE MODE
          XOR B0
          DAC I (SUT)
          RIL 6S
          IDX A
          LAC I A
          RCL 6S
          DIO I (SUK)
          RAR 6S      /PSEUDO PC TO AC
          JMP STV7I   /PSEUDO GO
```

```
STVNPP,   LAC I (BTEM)
          SZL
          JMP STVRUR  /GIVE R1 ON CAN'T LOSE POP
         JSP ATM
          LAC F
          SAS ML4
          JMP STVWL   /NOT EOM, LOSE THEN LAC
STV2SP,   IDX I (BTEM)   /GIVE R2
STVRUR,   DAP H        /SUBR TO RESET USER'S REGISTERS
         LAC C
          DAC I (SUG) /OUTPUT PTR
          LAC D
         DAC I (SUA) /PSEUDO PC
          LAC G
          DAC I (FSA) /TEXT PTR
          LAC I (BTEM+1)
          DAC I (USERPC) /REAL PC (SOMETIMES REDUNDANT)
         JMP HEXIT

/DON'T LAW IN LOSE MODE
STV7L,    SZF I 4
          JMP STV7II
          DZM E        /LOOK LIKE LAC 0
/WIN ENTRY TO LAC IN LOSE MODE
STVWL,    CLL"U"CML    /ENTER LOSE MODE
STV2L,    LAC I (SUK) /RESTORE PTRS TO BEG OF LEVEL
         DAC C
          LAC I (SUD)
          DAC G
          LAC E
          SAD PUD      /(LAC H)
          JMP STV2SP
          LAW 5
          AND E
          SAS B17      /STAY IN LOSE MODE ON LAC 1
         CLL          /ENTER WIN MODE ON LAC 0; LAC 5
         LIO B17
          DIO E        /LOOK LIKE LAC 1 ON RECYCLE (LAC 5)
         SZA
          JMP STVPOP  /POP ON LAC 1; LAC 5
```

```
STV3,     JMP STVML
          LAC I (SUF) /PUSH OPCODE (DIP)
          SAD I (SUE)
          JMP STVLOS  /CAN'T PUSH, LOSE
          DAC A       /TEMPORARY PUSHDOWN PTR
          LAC I (SUD) /PUSH SUD
          DAC I A
          IDX A
          LAC I (SUT) /PACK PUSH SUT; SUK; PSEUDO PC
          LIO I (SUK)
          RIL 6S
          RCL 6S
          DAC I A
          IDX A
          LAC D
          RCL 6S
          RAR 6S
          DAC I A
          IDX A
          DAC I (SUF) /UPDATE PUSHDOWN PTR
          LAC C
          DAC I (SUK) /UPDATE "BEGINNING OF LEVEL" PTRS
          LAC G
          DAC I (SUD)
          LAC B0
          DAC I (SUT) /CLOBBER MULTIPLIER
          JMP STV7II  /NOW DO A "LAW" OPCODE

STV7,     JMP STV7L
STV7II,   JSP STVTRA
          LAC E
STV7I,    DZM D
          DAP D
STV6,     JMP STVML
          JSP STVTRA
          JSP STVRUR
          LAC E
          JMP G0

/STVL AND STVW IOTS
STVL,     CML
          LAC B0
          DAC I (SUT)
STVW,     LAC I (FSA) /UNRESTORE USER REGISTERS
          DAC G
          LAC I (SUG)
          DAC C
          LAC I (SUA)
          JMP STV7I
```

```
/ATOM TABLE SUBROUTINES
/ENTRY TO GET A CHAR. AND SKIP ALL IRRELEVANT (SUS) CHAR.S
ATM,    SUB B17
        DAP H
        LIO I (SUS) /PUT MASK IN IO
        LCH I G
        SAD ML6
        JMP ATMW    /WARNING
ATMRO,  DAC F       /SAVE CHAR.
        RAL 6S
ATMI,   ADD (LAC ATOM)
        DAC B
        XCT B       /LOOK UP ATOM TABLE ENTRY
        SZA I
        JMP XHEXIT  /R2, EOM ALWAYS FAILS
ATMWC,  DIO B       /SAVE MASK
        RIL 3S      /PUT CLASS BIT IN SIGN OF IO
        XAI
        SPA
        JMP ATMD    /DIFFERENT CLASSES
        XAI         /RESTORE AC
        AND B
        AND MR14
        SZA I
XHEXIT, IDX H       /NO BITS THE SAME, FAIL (R2)
        JMP HEXIT   /OK, R1

ATMD,   RIL 1S      /MOVE "OTHER CLASS" BIT INTO SIGN OF IO
        SPI I
        IDX H
        JMP HEXIT

ATMW,   LCH I G
        SAD ML4
        JMP ATMRO   /TREAT RUBOUT LIKE EOM
        RAR 6S
        IOR ML6
        DAC F       /12BIT CHAR. IN F
ATMWI,  LAC (404000)  /ATOM TABLE ENTRY FOR 12 BIT CHAR.S
        JMP ATMWC

/ENTRY TO TEST CHAR. IN F AGAINST MASK IN E
REATM,  DAP H
        CLA
        LIO F
        RCL 6S
        LIO E
        SNI
        SAS (74)
        JMP REATM1
        JMP STV5UL
REATM1, SAS C77
        JMP ATMI
        JMP ATMWI
```

```
/ROUTINE TO LCH I E, DAC A AND SKIP ON NOT EOM
STVLE,    DAP H
          LCH I E
          SAD ML6
          JMP STVLEW  /WARNING
STVLEI,   DAC A
          SAS ML4
          IDX H       /NOT EIM
          JMP HEXIT
STVLEW,   RCL 6S
          LCH I E
          RCR 6S
          JMP STVLEI  /12 BIT CHAR. IN A

/RELOCATABLE TRACING SUBROUTINE FOR STV
STVTRA,   DAP H
          LAW 7777    /TOP 6 BITS 0 FIRST TIME THROUGH
          AND E
          LIO E
STVTRL,   RIL 4S
          SPI
          ADD I (SUR)
          DAC E
          RIL 1S
          SPI I
          JMP HEXIT
          LAC I E
          AND (607777
          LIO I E
          JMP STVTRL

/OPCODE DISPATCH TABLE
STVDTB,   JMP STV0
          JMP STV1
          JMP STV2
          JMP STV3
          JMP STV4
          JMP STV5
          JMP STV6
          JMP STV7
```

```
/ATOM TABLE, SIGN BIT HAS CLASS - 0 MEANS EOM
ATOM,B13, 20          /SPACE
          400010      /!
B5,       10000       /"
          410000      /#
          410000      /$
          410000      /%
          410000      /&
B8,       1000        /'
          400020      /(
          400040      /)
          400001      /*
B11,      100         /+
B7,       2000        /,
B12,      40          /-
B10,      200         /.
          400002      //
B16,      REPEAT 10., 2  /DIGITS
          400004      /:
B6,       4000        /;
          400400      /<
          401000      /=
          402000      />
B14,      10          /?
ATOM+40,  410000      /@
B9,       400         /A
B17,      REPEAT 3, 1 /BCD
400       REPEAT 3, 1 /E FGH
400       REPEAT 5, 1 /I JKLMN
400       REPEAT 5, 1 /O PQRST
400       REPEAT 3, 1 /U VWX
400       1           /Y Z
          400100      /[
          0           /EOM
          400200      /]
B15,      4           /CRLF
ATOM+77,  /NO ENTRY FOR 77
```