KNUTH

2105
see C1001

IDA

CRD

VON NEUMANN HALL, PRINCETON, NEW JERSEY

MAINTENANCE MANUAL FOR PSYCO - PART ONE

The Princeton Syntax Compiler

Mr. Edgar T. Irons

January, 1961

23

Maintenance Manual

for

PSYCO

The Princeton Syntax Compiler

Part One

Edgar T. Irons

January 10, 1961

MAINTENANCE MANUAL FOR PSYCO

The Princeton Syntax Compiler

Part 1

It was originally intended that this manual should be issued for the first time complete in its final form. However, due to heavy demands on the author's time, it has become evident that the complete manual cannot be finished for at least several months. Since there is need at the present time for a detailed description of the Princeton Compiler, it was decided that as much as has been written on the compiler should be made available immediately as Part 1 of the final manual, and that the remaining material should be issued at a later time as Part 2.

This decision results in some references in the following document to sections which are not in Part 1, but for the most part, the material presented here is a self contained section of the final document. The material covered in Part 1 and the attached appendices is at least the most important to the thorough understanding of the compiler itself.

MAINTENANCE MANUAL
For
PSYCO - The Princeton Syntax Compiler

Psyco is a syntax directed compiler which operates on strings of symbols in some object language (currently ALGOL 60) and translates it into a string of symbols in some target language (currently the assembly language AR for the CDC 1604) according to tables of syntax and semantics constructed automatically from a meta-symbolic description of the translation. The maintenance manual for PSYCO is written in two parts.

The first part describes the translating mechanism itself: The metalanguage used to describe the translation, the routines which carry out the translation, and input output routines used by the compiler.

The second part describes the implementation of the translation of ALGOL 60 to CDC 1604 assembly language: The particulars of the algorithms used to carry out various ALGOL statements, the philosophy of storage of data and operation of the run-time program.

All the routines in the compiler and also the running programs compiled by the compiler assume the existence of a set of input-output translations which translate groups of characters received from input equipment into a set of internal symbols (0-360 BASE 8) upon which the compiler operates. One such set of routines is described in detail in section 1.3, but local equipmental restrictions may suggest some others as preferable. The compiler itself demands no particular numerical representation for any of the internal characters associated with any translation except for the following:

External character	Internal symbol (base 8)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
"space"	156
"carriage return"	155
"tab"	115
	157
	160
	161
	162
	163
	164
	165
	166
	167
	170

The first group of these symbols is the set of octal digits 0 through 7; the second group is the set of formatting characters for a typewriter (which suffice under simple conventions for most printing equipment); the third group is the set of metasymbols used to describe the translation. (See section 1.1). The complete list of

internal symbol representations for the translations of ALGOL 60 is given in Appendix 1.

Part 1. The Translating Mechanism

1.1 The Meta Language

1.1.1 Philosophy of the metalinguistic description

The translation specifications consist of a series of sentences, each one consisting of a syntax formula followed by a string of symbols designating the semantics of that syntax formula. The sentences have the following form:

> Let S be a syntax unit: either a metalinguistic variable or a symbol of the input language.
>
> Let P denote a semantic unit: either a symbol of the output language or a designator of a string of such symbols.
>
> Each sentence of the specifications then has the form:

$$\underbrace{S\ S\ S\ S\ \ldots\ S\ S}_{\text{Components}}\ =::\ \underbrace{S}_{\text{Subject}}\ \Longrightarrow\ \{\underbrace{PPPPPPPP\ \ldots\ P}_{\text{Definition}}\}$$

The syntax unit S following the metasymbol =:: in any sentence is the "subject" of the sentence, and the syntax units to the left of =:: are the "components" of the sentence. The string PPPP..PP between the metasymbols {and} is the "definition" of the sentence. Specifically, P may have one of the following three forms:

1. Any (p) symbol of an output language. The output language alphabet may contain any symbols, but when that alphabet does contain the symbols

   $$\{\ \wp\ '\ \exists\ [\ ]\ ;\ \}$$

   special conventions will hold in the cases described below.

2. An output string designator of the form

$$\underbrace{P^{11\ldots1}}_{\text{may be empty}} \; n \underbrace{[\![\, p \leftarrow PP..P \;;\; \ldots \;;\; p \leftarrow PP..P \,]\!]}_{\text{may be empty}}$$

where P and p are defined as above, and n is an integer designating a particular string.

If a string designator (2) is of the simple form $P\,n$ it denotes the string which is the <u>definition</u> of the sentence whose <u>subject</u> is the nth <u>component</u> to the left of =:: in the sentence containing the designator $P\,n$.

If the string designator is of the form

$$P\,n\,[\![\, p \leftarrow PP..P \;;\; \ldots \,]\!]$$

it denotes the same string but with substitutions made as indicated; namely, with the symbol p replaced at every occurance by the symbols **PPPPP..PP**, these substitutions being made one after the other from left to right. Examples are given below.

3. An output string function designator of the form

$$\underbrace{F^{11\ldots1}}_{\text{may be empty}} \; n \; \underbrace{[\![\, PPP..P \quad PP..P \quad \ldots \quad PP..P \,]\!]}_{\text{may be empty}}$$

where P is defined as above, and n is an integer designating a particular string function.

The output function designator F of 3 is used to specify a function of the strings **PPPPP..PP** enclosed between the brackets following the function designator. The integer n serves to identify a particular function which is relevant to some particular set of syntactic sentences. They serve to enhance the descriptive ability of the output language, and constitute part of the description of an input language.

Consider as an example of the use of string designators the following five sentences specifying a translation of an input string consisting of some series containing the letters a and b to an output string composed of the letters A, B, x and y, t, m.

a =:: letter ⟹ {A x}

b =:: letter ⟹ {B t}

letter =:: iden ⟹ {ρ_1}

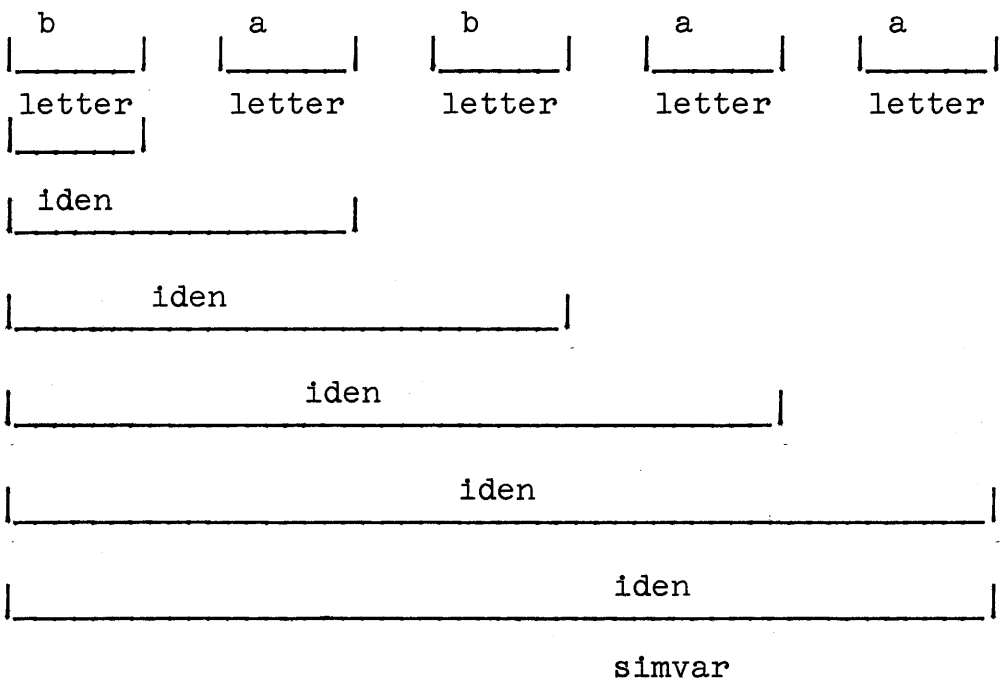
iden letter =:: iden ⟹ {$\rho_2 \rho_1 [\![t \leftarrow m]\!]$}

iden =:: simvar ⟹ {$\rho_1 [\![x \leftarrow y]\!]$}

The unique diagram of the input string

babaa

is



simvar

and the meaning of the final syntactic unit "simvar" is

BtAyBmAyAy

An example of function $\mathcal{Z}$ might be one whose value is a string of the characters 0 1 2 3 4 5 6 7 8 9, concatonated to represent the number of symbols in the parameter string of the function on any use. If we identify this function by the integer 1 and change the last syntactic statement on page 5 to

iden =:: simvar ⟹ {$\mathcal{Z}_1 [\![\rho_1]\!]$}

the meaning of the string of our example would now be merely the characters

The metasymbol ' serves as a left metaparenthesis counter to allow the output language to contain the metasymbols of the description, so that an input language may be described in terms of — and hence translated into — the meta language. This convention enables a translator to modify the set of translation specifications it is currently using according to the particular input string it is examining. This enabling convention depends on the use of the symbols {and} as metaparentheses. If in any definition of a syntactic sentence the number of 's following any occurrence of ρ or $\mathcal{F}$ is <u>not</u> equal to

$$(\text{the number of } \{\text{s to the left}) - 1$$

then the symbol ρ or $\mathcal{F}$ and its associated symbols

$$[\![\ ;\ \leftarrow\ ]\!]$$

will be treated as symbols of type 1 rather than in the way described above. The metasymbols {and} are always treated as symbols of type 1, when they occur in the string of a definition.

If the last sentence of the descriptions of the example were changed to

$$\text{iden} = : \text{simvar} \Longrightarrow \{\rho_1 =:: \underline{\text{realtype}}\ \{\rho_1'\,[\![x \leftarrow y]\!]\,\mathcal{F}_1\}\}$$

the translation of the string babaa would be:

$$\text{BtAxBmAxAx} =:: \underline{\text{realtype}}\ \ \{\text{BtAyBmAyAy}\,\mathcal{F}_1\}$$

1.1.2 The Particulars of notation for this implementation of the metalinguistic description.

1.1.2.1 The format of the metalanguage.

The format used in the actual implementation of the metalinguistic description differs in detail from that used in the section 1.1.1.

The external characters used as metasymbols in all the examples and descriptions in this manual are those characters on the CXCO typewriter-punch at the Princeton installation. The characters and their internal representation (after input translation) are

CXCO character	Translated representation recognized by the compiler
!	160
$	161
∃	162
⊣	163
⊥	164
⊢	165
#	166
¢	167
⊤	170

We must, of course, point out that these CXCO characters are tied to the translator in no way whatsoever. Any external representation for the metasymbols is acceptable, and can be used with the translation by redefining the external character set to the input routines (section 1.3). It is only the internal representation which is fixed for the compiler. However, for convenience, we shall use the CXCO character representation for the purposes of description.

The format for the actual implementation is given below in terms of the notation used in section 1.1.1. A rough correspondence between the symbols of the description in section 1.1.1 and those of the implementation is:

=::	!
{	$
}	∃

'	⫢
⫣	⫤
⫍	⫥.
←	#
;	¢
⟧	⫦
:=	!
{	$
}	⫧
⫍	⫥
⫣	⫤
←	#

The symbols

are exactly equivalent, that is the implementation characters are used exactly as the corresponding characters were used in 1.1.1.

For the other symbols, slightly different format rules are used. In particular

1. ⫢ serves the same function as ', but the ⫢'s <u>precede</u> rather than <u>follow</u> the ⫥ or ⫤ (i.e. ⫍ or ⫣) with which they are associated.
2. The metacomma ¢ is used as a left bracket (⟦) as well as a comma.
3. The closing bracket ⫦ (⟧) <u>must</u> always follow the simple string or function designator ⫥ or ⫤ .

The following examples point out the format differences.

1. ⫍'''...' n ⟦p←PP...P; p←PPP; ...; p←PP...P⟧

MAY BE EMPTY — MAY BE EMPTY

is equivalent to

⫢⫢...⫢⫥ n ¢p#PP...P¢p←PPP¢...¢p←PP...P ⫦

MAY BE EMPTY — MAY BE EMPTY

2. $\underbrace{\mathcal{I}^{11\cdots1}}_{\text{MAY BE EMPTY}}\ n\ \underbrace{[\![PP\cdots P;\ PP\cdots P;\ \cdots;PP\cdots P]\!]}_{\text{MAY BE EMPTY}}$

is equivalent to

$$\underbrace{\mathcal{H}\mathcal{H}\cdots\mathcal{H}}_{\text{MAY BE EMPTY}}\ \mathcal{E}\ n\ \underbrace{\mathcal{C}PP\cdots P\mathcal{C}PP\cdots P\mathcal{C}\cdots\mathcal{C}PP\cdots P}_{\text{MAY BE EMPTY}}\ .\mathcal{I}$$

Note that the $\mathcal{I}$ is always required. Hence

$\mathcal{P}_n$ is equivalent to $\mathcal{E}\,n\,\mathcal{I}$

and

$\mathcal{I}\,n$ is equivalent to $\mathcal{E}\,n\,\mathcal{I}$

As mentioned in section 1.1.1, the metasymbol ! (= :) is always ignored, and the meta symbol $\Longrightarrow$ is not used in the implementation at all. For examples of the metasymbolic description, see the ALGOL description in Appendix 2.

1.1.2.2 The form of machine language programs for the compiling functions $\mathcal{E}\,n\,\mathcal{I}$ $(\mathcal{I}\,n)$:

These programs are string translation or string generation programs which aid in the specification of the definition part of the meta-linguistic sentences. In general they are small programs to generate unique numbers which would be used as internal labels in a translation, or perform some translation of a highly specialized nature. The programs would generally be written in Assembly language, and would have to be in the machine during any compilation. The starting address of each function must be listed in a table of functions which the compiler references when it needs one of them during a translation. The specifications for such a program are:

1. Entry: The entrance address for the function must be entered in the table FNBASE, so that the contents of the lower address position of FNBASE+n is the address of the entrance for function n. (See the description of TRANSYM, Section 1.2.3.2)

2. Exit: Each function shall transfer to PI (in TRANSYM) when it has complied its action.

3. Storage: If any of the function programs use any index registers, they must save the contents of those registers. The Accumulator and Q are open to use (however, A is used for information transfer).

4. Transmission of input parameters: The input strings, if any, to the function will be in a portion of the output string (OTPX). The index of OTPX which marks the character preceeding the beginning of the first point string is in the lower address position of the accumulator upon entry. B6 contains the index of OTPX which marks the character following the end of the last input string. If there are several strings as parameters to the function they are separated by the 8 bit character (base 8) 377. If there are no input parameters (A[lower]) = (B6) upon entry.

5. Transmission of output: The functions must put their output strings in OTPX beginning where the first symbol of the input string was located and must mark the end of the output string by putting the ineex of the last symbol of the output string in B6. If r is the contents of A [lower] upon entry to the function, then upon exit:

(B6) = r + (the number of symbols put out).

If there is no output string then upon exit:

(B6) = r.

1.2 The subroutines comprising the compiler:

For the translation of ALGOL 60, PSYCHO is written as a two pass compiler. The first pass scans the input string according to syntax according to syntax tables for pass I and outputs syntactic sentences concerning the particular variables of the program being compiled. These syntactic sentences are added to the permanent Pass II symtax tables, and then the original input program string is scanned again according to the modified Pass II syntax tables, to output the final translation. This process is carried out by a group of subroutines each of which performs a well defined part of the total process. The operation of the subroutines is correlated by a "main program" which initiates the calls on the other routines.

In this section we first list the routines which comprise the compiler and outline generally their functioning. Secondly, we describe in detail the composition of the tables and storage areas used by these routines. Lastly, we give the detailed description of the routines and correlate these with the "listing diagram" which is found with the listing in Appendix 3.

1.2.1 The subroutines - general

1. ROUST (DIAGRAM). This subroutine is the heart of the compiler. It is programmed as a recursive subroutine (i.e. one which calls itself directly), and essentially its function is to construct a syntactic diagram of the object program, while simultaneously forming

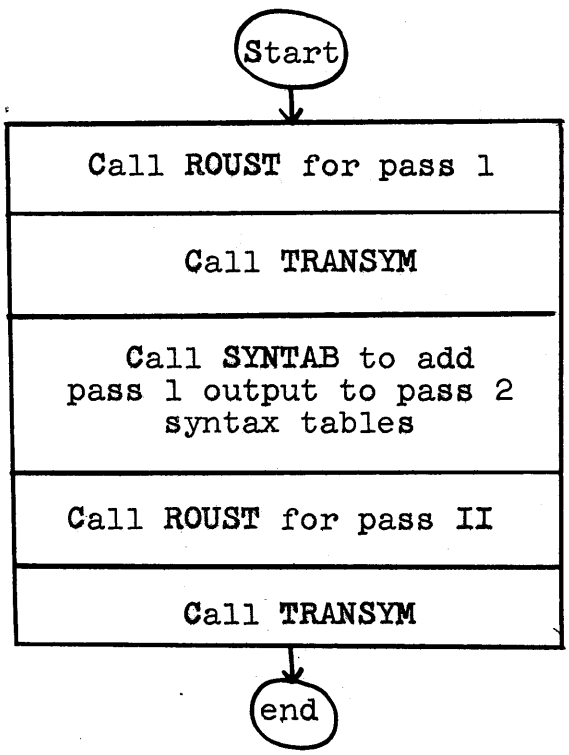
an output string (OTP) which indicates to TRANSYM which of the definitions of the metasentences are to be present in the target language program, and how these definitions are to be put together.

2. TRANSYM. This subroutine is also programmed as a recursive procedure. Its function is to form the final output string (OTPX) which is the target language program from the linked list (OTP) provided by ROUST.

3. SYNTAB. This subroutine is a relatively simple program to build the internal tables (STAB -- see 1.2.2) from the metalinguistic descriptions which are used by ROUST in its operation.

4. COSYN. This subroutine might in a sense be called the main program of PSYCO. Its function is to set initial values of constants, and execute the several calls of ROUST, TRANSYM, and SYNTAB which occur during a compilation. It is essentially an overseeing routine to correlate the action of the others.

The following is a simple flow diagram of COSYN, showing the overall operation of the compiling system:



5. Service routines: NEWSYN and GSCR.

a. NEWSYN is a service routine which calls SYNTAB directly to build new syntax tables STAB either from an in-memory (i.e. already input translated) string of symbols comprising a metalinguistic description, or from such a string available through the input translator (INTRAN). Its use is primarily to aid in making available to the compiler new metalinguistic descriptions, or corrected ones. The operating instructions for NEWSYN are found in Appendix 4.

b. GSCR. This is a Generalized String Correction Routine which allows completely general correction of any string of words (presumably though not necessarily internal symbols, one per word) from a correction list available from input equipment through INTRAN. The particular formats for this correction list, and the operating instructions for GSCR are found in Appendix 4.

1.2.2 Storage Allocation and Table Composition

This section describes in detail the tables and storage areas used by the subroutine outlined in the last section. For each table, the basic construction, and the methods of reference are given.

1.2.2.1 STAB (referenced MNEMONICALLY in the listing with B1, base address 0 or L (note that L is defined to be equivalent to 0)). STAB contains the internal representation of the metalinguistic description and is called the Syntax TABLE. It contains the metalinguistic description for both passes of the system and, since it is

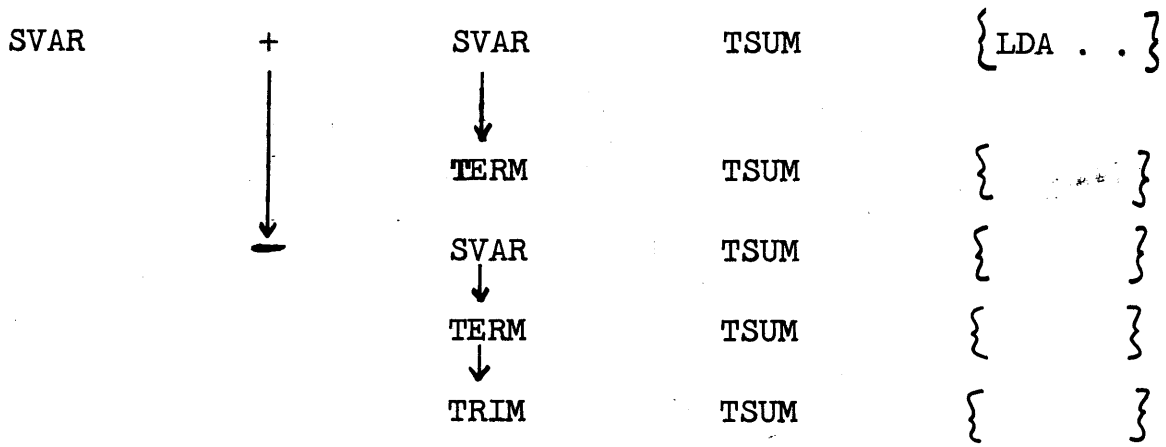
constructed as a semi-linked list, it makes no difference whether the sentences of the two passes are intermixed or not. (However, by convention, the syntactic sentences for the first pass occur at the end of STAB, ehile those for the second pass are in the end.) STAB is constructed in the memory from any metalinguistic description of the form outlined in section 1.1 by the subroutine SYNTAB and is used in the scanning process by ROUST.

The construction of STAB is dictated by the requirements of the scanning routine ROUST. ROUST is designed to examine groups of symbols in the input string and determine from STAB what syntactic catagories the symbols fall into. Hence, having found that the first n symbols fall into some syntactic category, A, ROUST must be able to find conveniently from STAB the names of all the syntac constructions (e.g. B, C, D) which can follow the construction A (concatonated on the right). Given these, ROUST examines the symbols of the input string following the nth to determine whether any group of them fall into the syntactic categories B or C or D ... Then finding, for example, that the input symbols fall into category B, ROUST must further be able to find from STAB the names of all syntactic categories which may follow the categories AB, and so on. (The exact process is explained 1.2.3.1)

These requirements make it desirable to store the syntactic information in STAB in a tree like structure which is illustrated diagramatically by the following examples: The five sentences:

```
SVAR    +    SVAR    =:    TSUM    ⟹    {LDA Q1
                                         ADD Q2}
SVAR    +    TERM    =:    TSUM    ⟹    {. . .}
SVAR    -    SVAR    =:    TSUM    ⟹    {. . .}
SVAR    -    TERM    =:    TSUM    ⟹    {. . .}
SVAR    -    TRIM    =:    TSUM    ⟹    {. . .}
```

would be linked as follows:



Notice in the above diagram that at every point where it is possible for a given syntactic construction to be followed by more than one syntactic element on different occasions, the alternatives are easily found by following the downward arrows. The progression through the tree by ROUST as it diagrams the input string is from left to right across the tree, following the topmost possible path. If it is discovered at any point that it is not legitimate to proceed horizontally across any path, (we will leave the discussion of how this decision is made to the discussion of the operation of ROUST -- section 1.2.3.1) then ROUST proceeds down the path beginning at the last junction passed (the nearest to the left). Proceeding in this manner, ROUST eventually must either decide that the whole branch of the tree will not select a diagram for the program, or it must eventually reach the right side of the tree. If

it reaches the right side of the tree, it then has available the output string (which is the definition of the syntactic sentence) for the section of the input string it has diagrammed in proceeding across the tree. Furthermore, it has available the name of the syntactic category which it will then assign to the construction discovered, and hence can continue the diagramming process starting at a new branch of the tree.

To realize a storage mapping of the syntactic sentences which allows ROUST to operate in this way, the convention is adopted that elements listed horizontally across the tree (as above) from left to right, will be stored consecutively in memory from smaller address to larger address in STAB. Where a junction in the tree is indicated by a downward arrow, a link address will be stored in the same word with the symbol from which the arrow departs. This link address is, of course, the address of the first symbol of the string toward which the arrow points. Hence, each word of STAB contains:

1. The internal code of a syntactic name or output symbol, and
2. A link address - the address 00000 indicating no link.

Specifically the format is:

000 (link address) 000 00 (symbol)

Note that the link address is the machine address of the symbol beginning the branch toward which the link points. (specifically NOT relative, but ABSOLUTE address)

For the example given above, the pertinent section of STAB would have the structure:

	LINK	SYMBOL
I 1	I 4	+
I 2	I 3	SVAR
	0	TSUM
	0	{
	.	.
	.	.
	0	}
I 3	0	TERM
	0	TSUM
	:	{
	0	:
	0	}
I 4	0	—
I 5	I 6	SVAR
	0	TSUM
	0	{
	.	.
	.	.
	0	}
I 6	I 7	TERM
	0	TSUM
	0	{
	:	.
	0	}
I 7	0	TRIM
	0	TSUM
	:	{
	.	:
	0	}

where In is the relavant machine address. Notice in this example that the <u>first</u> syntactic name (SVAR) is <u>not</u> stored explicitly in STAB. Thus the <u>first</u> syntactic name of the branch is taken effectively as the <u>name</u> of that branch. In other words, there is one and only one branch of the tree for each syntactic category possible in

any set of metalinguistic sentences. If ROUST is ever interested in examining any syntactic sentence beginning with SVAR, for example, it always comes to the branch of the tree which bears the "name" SVAR (I 1 in the example given), no matter which of the sentences beginning with SVAR it wishes to consider.

The only remaining question about the structure of the tree is how ROUST knows where the branch for any syntactic element is located. This information is provided by the table TRAN, which is essentially an index to STAB. (TRAN is described in detail in Section 1.2.2.2).

Having outlined the general structure and use of STAB, we now describe the conventions for determining which symbols are stored in STAB.

1. In the tabling of the components and subject of any syntactic sentence (i.e., all symbols to the left of $) the following symbols are deleted:

Symbol	Internal Code
"space"	156
"carriage return"	155
"tab"	115
!	160

2. In the tabling of the definition of the sentence, all symbols are entered into the table (consecutively) including both parentheses $ and 3 .

3. For the purposes of tabling, it is considered that a sentence ends with the last closing parenthesis 3 of its definition, and that the next sentence begins with the symbol immediately following that 3 .

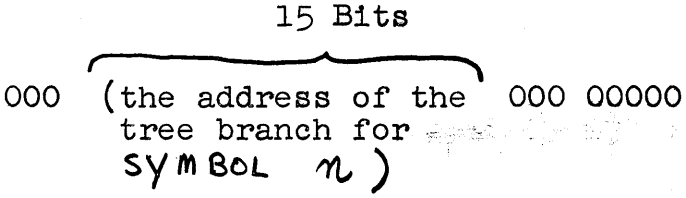
Note that under these conventions the opening parenthesis $ serves two functions: 1. to implicate the immediately preceding syntactic name as the subject of the sentence, and 2. to act as the opening parenthesis of the pair enclosing the definition.

1.2.2.2 TRAN (referenced memonically by use of TRANLOC, ONETRAN, and TWOTRAN)

As mentioned in the last section, TRAN is the table which is an index for STAB. Since PSYCO is programmed as a two pass compiler, and thus expects to reference two sets of syntax tables, there are in fact two such indices. One of them (ONETRAN) is the index for the Pass I Tables, and the other (TWOTRAN) is the index for the Pass II Tables. Since STAB contains the syntactic sentences for both passes, both ONETRAN and TWOTRAN refer to those branches pertinent to the pass with which it is associated.

TRANLOC is a single memory cell whose lower address position contains the address either of ONETRAN or TWOTRAN, whichever is currently in use. Since the use of either one of these tables is exactly the same, we shall henceforth speak of them both as TRAN unless we wish to specify one of them.

Since there are 360_8 (maximum) internal symbols possible in any specification, TRAN (i.e. each one) is 360_8 words long. If the code for any internal symbol is n then the contents of TRAN + n is:



The "link addresses" in TRAN, like the link addresses in STAB, are absolute addresses. If there is no tree branch in STAB for symbol n, then the link address in TRAN + n 00000.

For the implementation of ALGOL 60, there is a special group of internal symbols (343 to 357_8) which are used to qualify syntactic statements as being valid for only certain parts of the particular input string under consideration. These symbols are used only in the syntactic sentences added to the Pass II tables from the pass I scan. They appear in STAB as syntactic categories (i.e. like SVAR, TSUM) but the entry in TRAN for these symbols has a special form: If n ($335_8 \leq n \leq 360_8$) is the internal code for one of these symbols, then the contents of TRAN + n is:

400 (UPPER BOUND) 000 (LOWER BOUND)

where UPPER BOUND is the highest index of the input string for which any syntactic sentence containing this qualifier is valid

and LOWER BOUND is the lowest index of the input string for which any sentence containing this qualifier is valid.

The sign bit indicates that this entry of TRAN is for a qualifier rather than for a normal syntactic element.

It is a general policy in PSYCO that all tables may be relocated by reassembling the system having first changed the Equivalence psuedocommands which locate the tables. This policy also holds with the TRAN tables, but with the following restriction: The address of TWOTRAN must be

(the address of ONETRAN) + 400_8 .

Furthermore, at the end of TWOTRAN 5000_8 words of storage must be reserved for the Boolean precedence matrices ONEPREC and TWOPREC. A detailed memory map for the total 6000_8 storage area which holds these four matrices is given at the end of the next section (wherein ONEPREC and TWOPREC are explained.

1.2.2.3 PREC (referenced nemonically by referring to TRAN + k where $1000_8 \leq k \leq 5777_8$ and by the tags TRAN1, TRAN2, TRAN3, TRAN4, and TRAN5.)

PREC is a Boolean precedence matrix of dimensions 360_8 x 360_8. As for TRAN, there are two such matrices, specifically ONEPREC (used for Pass I) and TWOPREC (used for Pass II), and again since the use of either for its pass is the same, we shall refer to them as PREC for the following discussion unless we wish to speak specifically of one of the other.

The use of PREC in the scanning process is to determine when the scan is examining paths which are known in advance to be invalid for the symbols under consideration. Specifically, if p is one syntactic category and q is another (possible equal to p) then PREC [p,q] is true if

1. q is the subject of any sentence beginning with (i.e., whose first component is equal to) p OR
2. q is the subject of any sentence beginning with any syntactic element which is the subject of any sentence beginning with q OR

3. q is the subject of

⋮

and so on.

Stated mathematically: Let A and B be any syntactic sentences ($A \neq B$). Let a_i be the first component of sentence A, and a_f be the subject of sentence A, and define b_i and b_j similarly.

Define the operator $\equiv>$ as follows:

1. For any sentence A, $a_i \equiv> a_f$
2. If, for any other sentence B, $a_f = b_i$ then $a_i \equiv> b_f$

Now we may define the precedence matrix **PREC** as

PREC [p,q] = true if and only if $p \equiv> q$.

Since **PREC** is a Boolean matrix, its elements may be bits of words in memory, and indeed **PREC** is so stored. However, since **PREC** has dimensions $360_8 \times 360_8$ the number of words necessary to store each **PREC** is

$$\frac{360_8}{60_8} \times 360_8 = 2260_8 \text{ words.}$$

In other words there are five sections of each **PREC** each section being 360_8 words long. For convenience in referencing, each of the sections is stored 1000_8 words above the last, and the first section is stored 1000_8 words above **TRAN**. (See diagram at the end of this section.)

Furthermore the first element of PREC is the rightmost bit of the first word of PREC, the second element is the next bit to the left, and so on. (Note that the bit numbers on the console face correspond to the element numbers in the words of PREC). Specifically, the expression for calculating the address of any element of ONEPREC or TWOPREC is given by:

$$\langle \text{ONEPREC } [p,q]_W \rangle = (\text{ONETRAN}) + p + ((\text{greatest intgr in } \tfrac{q}{60})+1)1000$$

$$\text{BIT} = q \pmod{60}$$

where all numbers are base 8, and

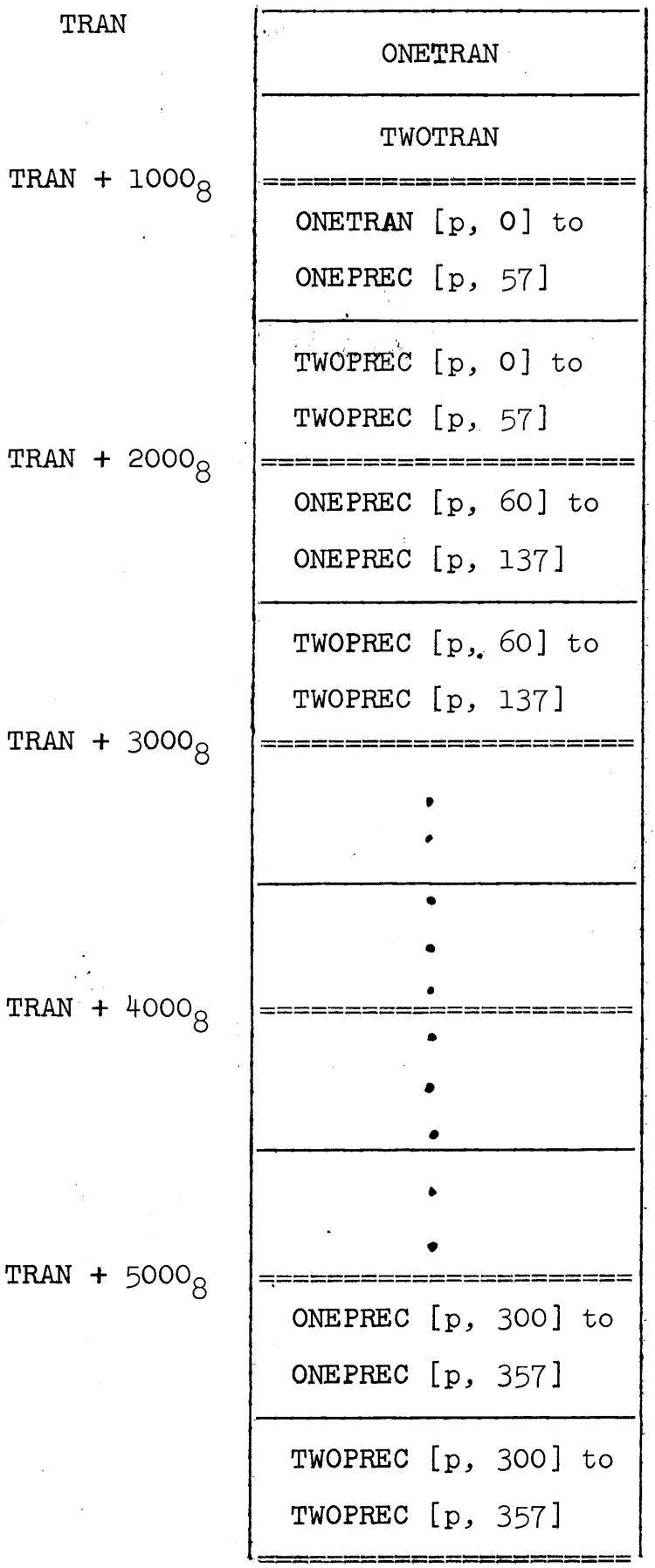
$\langle \text{ONEPREC } [p,q]_W \rangle$ is the address of the word containing the element ONEPREC [p,q], and

BIT is the octal bit number (with the rightmost bit being defined as the 0 bit) of the bit in that word which is the element.

Elements of TWOPREC are obtained by the same formula, with ONETRAN replaced by TWOTRAN.

Note that the formulas given define the address arithmetic used to obtain elements of PREC in ROUST and SYNTAB.

The following diagram shows the storage map for TRAN and PREC:


TRAN
ONETRAN
TWOTRAN
TRAN + 1000₈
ONETRAN [p, 0] to
ONEPREC [p, 57]
TWOPREC [p, 0] to
TWOPREC [p, 57]
TRAN + 2000₈
ONEPREC [p, 60] to
ONEPREC [p, 137]
TWOPREC [p, 60] to
TWOPREC [p, 137]
TRAN + 3000₈
TRAN + 4000₈
TRAN + 5000₈
ONEPREC [p, 300] to
ONEPREC [p, 357]
TWOPREC [p, 300] to
TWOPREC [p, 357]

1.2.2.4 INPUT (denoted mnemonically as S, indexed with B4)

INPUT is the storage area which holds the input symbol string. While ROUST is scanning during Pass I, it reads the input symbols into INPUT (by using INTRAN). The complete input string is retained in INPUT after it has been completely read in, so that it may be used by ROUST in the second pass scan. The input symbols are stored one per word in INPUT in the following format:

8 BITS

000 00000 000 00 (internal representation of input symbol)

1.2.2.5 OTP (mnemonically OTP, indexed with B3 in ROUST.)

OTP is the intermediate list which is the output of ROUST and the input for TRANSYM. There are two basic formats for words stored in OTP:

15 BITS 15 BITS 15 BITS

Format 1: 4 (X) (Y) (Z)

where Z is the address in STAB of the first symbol of a definition of some sentence (i.e., the first symbol after the first $ in that sentence). X is the index of the first symbol in the part of the input string which is being translated into the string which is the definition Z and Y is the index of the last symbol in the part of the input string which is being translated into definition Z.

15 BITS

Format 2: 000 00000 000 (W)

where W is n the address in OTP (not the index, but the absolute address) of an entry of Format 1.

The entries in OTP form essentially a set of "macro" orders indicating how TRANSYM should put together the definitions from STAB to form the translated output string from a pass of the compiler. You will recall that the definitions of sentences may contain string designators of the form $\wp_n$ $(\underline{6}\ n\ \overline{9})$ which specify that they should be replaced by the string which is the definition of a sentence whose subject is the nth component (to the left of $\Rightarrow$) in the first sentence. (Section 1.1.1). It is TRANSYM which causes this substitution to be made, and the form of OTP is dictated by the requirements of TRANSYM for making these substitutions (and by ROUST in generating the contents of OTP). At the time TRANSYM begins to untangle this list, it is supplied with the address in OTP of the word of Format 1 whose Z part is the address of the definition of the sentence whose subject was PROGRAM. In other words, TRANSYM is initially given an address in OTP which directs it to the definition of STAB which comprises the entire translated program (after appropriate substitutions have been made). For the pruposes of explanation let this address of OTP be OTP [j]. TRANSYM begins examining the symbols of this definition one by one, and if they are of the simple type, it puts them in the final output string (OTPX, section 1.2.2.6). However, if it comes across a string designator $\wp_n$ $(\underline{6}\ n\ \overline{9})$ TRANSYM must then begin outputing symbols from another definition. Which definition? This it determines by going back to OTP [j]. In OTP [j + n] it will find a link to the definition which will replace the string designator $\wp_n$. This link may be either of Format 1 or Format 2. If the link is Format 1, the Z term is the address of the definition TRANSYM wants. If the link is of Format 2, the W term is the address of OTP [k] (where $k < j$); OTP [k] will be of format 1, and the Z

term of OTP [k] will be the address of the definition TRANSYM must use. Whether OTP [j + n] is of Format 1 or Format 2 is determined by the value of n in any particular case. If

n = (the number of components of the sentence to which the definition of OTP [j] belongs),

then OTP [j + n] will be of Format 1, (I.e., a direct link). If k is less than m , then OTP [j + k] will be of Format 2, (i.e., an indirect link).

The structure of OTP may be explained somewhat more precisely by adopting the following notation:

Let M_i be the symbolic name of the definition of sentence i. We may then regard the definition as it stands as a symbolic "function" (as a MACRO in an assembler), and the string designators in a "call" of the macro are to be plugged in.

Assume that a macro call has the form:

$$M_i\ (P_1,\ P_2,\ \ldots\ P_n)$$

Then we would agree that the string P_1 should replace the indicator $\wp_1$, P_2 should replace $\wp_2$ and so on in the final copy of the macro. In the compiling system, however, observe that all the "parameters" of the macro call would be the names of other macros. (Not all macros, however, have parameters, so the process of untangling the macros is not an endless one). Then an actual macro call for the system might be:

$$M_i\ (M_j\ (M_k, M_l),\ M_n(M_o, M_p(M_q),\ M_r))\ .$$

In this case, M_i is the "main" macro, and its parameters are M_j, M_n, and M_r, some of which have parameters. M_k, M_l, M_o, and M_r are macros with no parameters.

This example could be the list generated by ROUST indicating the definitions comprising the translated program. However, if we specified that the list be given in the form of the example, it would not only be difficult to generate the list, but difficult to use it. In fact, this notation conveys exactly the same information as the elements of OTP do, but OTP is merely arranged in a more convenient form (a variation of the Polish form) which is parenthesis free. Symbolically, the above example could be rewiitten to fit in the form used for OTP as follows:

Address	Contents
1	M_k
2	M_j
3	1
4	M_l
5	M_o
6	M_n
7	5
8	M_p
9	M_q
10	M_i
11	2
12	6
13	M_r

In OTP, items 3, 7, 11, and 12 would be entered in Format 2 and the rest in Format 1. (Lines have been drawn in the diagram for expositional clarity, but notice that they are not necessary to specify the macro call structure if, as is in fact the case, the number of "parameters" of every "macro" is known).

With the macro calls in this form it is a simple matter to find the macro-parameters of any macro call. In order to find the nth parameter of a macro, we need only consult the nth word of the list after the word which mentions the original macro. For example to find the second parameter of M_i we consult entry 12 (since M_i is mentioned in 10) and find the reference to entry 6, which contains the name of the macro we are looking for, namely M_n.

In fact, not only is it easy to determine the parameters of the "macro" - definitions from this type of list, but the nature of the scanning method used in ROUST is such that this form of list is the easiest possible form to generate.

The details of how the list is generated, and used are explained in the sections on ROUST and TRANSYM (1.2.3.1 and 1.2.3.2).

1.2.2.6 OTPX (mnemonically OTPX, and indexed by B6)

OTPX is the output string from a pass through the compiler. The format for OTPX is identical to that for INPUT. At the end of the first pass, OTPX will contain a set of syntactic sentences which specify definitions pertinent to this compilation only (i.e., this particular input string). These sentences are added to the syntax

tables STAB for Pass II (by invoking SYNTAB) before pass II is begun.

During Pass II, the symbols placed in OTPX comprise the final output string of the compiler, and hence may be written on output equipment (using OUTRAN, section 1.3) as they are put into OTPX.

"6 OTPX" in the b and m terms of a command refers to the last symbol stored on OTPX rather than the next available cell of OTPX for storage as for most other tables.

1.2.2.7 SUBL (mnemonically SUBL, index N)

SUBL is a Substitution List used by TRANSYM to store the lists of substitutions to be made in strings called for by String designators of the form

$$P_n [\![p \leftarrow P P P \ldots]\!]$$

in definitions of sentences. There is one entry in SUBL for each substitution currently in force during the untangling operation of TRANSYM. Each of these words has the format:

15 BITS 8 BITS

000 (V) 000 00 (U)

U is the symbol for which the string named by V is to be substituted

V is the absolute address of the first symbol in the string to be substituted for the symbol U

The string named by V is stored in the table ISUBIX (section 1.2.2.8).

Entries are made in SUBL (and ISUBIX) immediately before TRANSYM begins to output the string designated by the

$$P_n \quad [\![p \leftarrow P\ P\ P\ \ldots]\!]$$

and these entries are cancelled when this string has been output. Every symbol which is output by TRANSYM is checked against the list in SUBL and, if that symbol appears anywhere in SUBL, the string V for the last occurance of the symbol U in SUBL replaces the symbol in the output list. In addition every symbol which is entered into ISUBIX is checked against the list in SUBL before it is entered in ISUBIX and, if the symbol appears in SUBL, the substitution is made as for symbols being put in the output list. This guarantees that all the strings in ISUBIX will have had all substitutions already made on them, and hence may be putout (either into ISUBIX or the output list OTPX) as they stand.

1.2.2.8 ISUBIX (mnemonically ISUBIX, indexed by using SUBIX)

ISUBIX holds the (fully substituted) strings to be substituted for the variables U listed in SUBL. SUBIX [lower] contains the absolute address of the cell in ISUBIX in which the last symbol of such a string was stored. The format of each such string is the same as the format for INPUT and OTPX; symbols are stored one per word with the symbol code in the rightmost eight bits of the word. The last symbol of each of these strings is followed by the symbol 377_8, this symbol indicating the end of the string.

1.2.2.9 BIN (mnemonically BIN, indexed with B5, and referenced as outlined below)

BIN is the storage area used to hold the quantities unique to any call of ROUST or TRANSYM. Since both of these subroutines are recursive (i.e., call themselves), each call of either of them must have a space to store its parameters and temporary quantities which is separate from the space used for the last call. In other words, when ROUST, for example, calls ROUST directly, the two operations of ROUST must use different storage areas, lest the second operation destroy data pertinent to the first operation. This separation of data storage is accomplished by using BIN as a "pushdown" list. The names of data unique to an individual call of ROUST or TRANSYM are given in the following table, along with their relative positions in BIN. (Note that equivalence statements in the listing make these assignments)

Name of datum	Relative location in BIN
For ROUST	
IVAR	BIN + 0
GOAL	BIN + 1
MAC	BIN + 2
REXIT	BIN + 3
QUAL	BIN + 4
COUNTS	BIN + 5
SW	BIN + 6
PAR	BIN + 7

For TRANSYM

EXIT	BIN + 0
RR	BIN + 1
VKEEP	BIN + 2
TE	BIN + 3
NKEEP	BIN + 4
SUBXKEEP	BIN + 5
SUBLKEEP	BIN + 6
P	BIN + 7
Q	BIN + 10_8

Whenever any of these quantities is referenced by ROUST or TRANSYM, B5 is used for the reference. The contents of B5 is incremented 10_8 each time ROUST calls itself, and decremented 10_8 each time ROUST exits to itself. (For TRANSYM (B5) is incremented and decremented 11_8). This insures that the data calls referenced on each level of operation of ROUST (or TRANSYM) will be unique to that level.

1.2.2.10 Overflow of Tables

All of the tables and storage areas listed above except TRAN and PREC must be capable of containing a number of symbols which is dependent on the size of the input string for any compilation. Hence, every time an entry is made into any of them, a check is made to insure that they have not grown to be larger than the space alloted to them. For each of the tables, there is a quantity which indicates the upper bound of the storage area alloted. The actual value

of this checking parameter depends on how the parameter is used to make the overflow check. The parameters are listed in the following table for each of the storage areas, and a formula is given for each which specifies its value in terms of the octal number (n) of words allotted to each particular table.

Table	Overflow Variable	Formula for value
STAB	NEGMAXTB	= -STAB - n
INPUT	INSTOP	= n
OTP	NEGMAXOT	= -n
OTPX	NEGOTPX	= -n
SUBL	(NSTOP)	= n
ISUBIX	(SUBXSTOP)	= ISUBIX + n
BIN	NEGBIN	= -n

Parentheses indicate that the value of the formula should be the quantity in the lower address position of the variable enclosed in parentheses, whereas variables not enclosed in parentheses should be equated to the value of the formula.

In the event that any of the tables should overflow during a compilation, a return jump is executed to OUTBOUND, which is a subroutine to handle the overflow (section 1.2.3.6).

1.2.3 The Subroutines of the Compiler -- Detail

This section gives a detailed explanation of the theory of operation of the subroutines of the compiler. The routines will be explained individually, and each explanation will be followed by a non-detailed flow chart of the routine. The policy followed in this manual will be to present several explanations of each routine, each one covering the same basic ideas, but on a different level of detail. The first and most general explanation of the routines has already been given in section 1.2.1. This section will present the material again with enough detail to give a thorough understanding of the ideas behind the programs, but will not deal in detail with the methods used to implement these ideas. The flow charts presented in this section uphold that philosophy, but provide the connecting link to the next level of exposition. The detailed explanation of the method is presented in the form of word descriptions of the individual algorithms which compose each routine. Both these word descriptions and the flow charts to be presented below follow the listing in format. That is, the boxes on the flow charts will correspond in position to the paragraphs of the detailed listing descriptions, and the paragraphs of the listing description will in turn correspond in position to the commands on the listing. Hence the exposition will "cascade" from the first brief description to the listing itself in an orderly, but ever expanding fashion. In essence, we exposit 1. The entire system, then 2. The routines of the system (this section), then 3. the parts of the routines, and finally 4. The commands which comprise these parts (the listing). The word descriptions of the
of the listing, and the listing are found in Appendix .

1.2.3.1 ROUST -- THE SYNTAX SCAN.

ROUST is a recursive subroutine which in effect diagrams the input string for a compilation according to the syntactic information about the system which is given by the metalinguistic description of the system. The two guiding input parameters of ROUST are IVAR and GOAL. IVAR is the address of a spot in the syntax tables STAB indicating the part of STAB from which ROUST should continue the diagramming process. GOAL is the name of a syntactic construction which ROUST is to try to build by diagramming the input symbols. When ROUST is called, it will begin at the spot specified by IVAR, and continue through the syntax tables, consulting the input string for guidance as it goes, until either (1) it has diagrammed the input string to the point where it has found the longest group of input symbols following the starting point which will form the syntactic construction specified as the GOAL, or (2) it has determined that there is no group of symbols after the starting point which will form the requested syntactic construction GOAL. During the process, a marker (in fact in B4) indicates which element of the input string must be the first of the requested syntactic construction. If ROUST finds the GOAL, it will move this marker to the next symbol after the last one included in the construction GOAL. If ROUST cannot find the GOAL, it will leave the marker where it was. Hence, at any given time in the diagramming of the input string, all the symbols to the left of the marker (i.e., at the first part of the string) will have been discovered to be in some syntactic category, and all the symbols to the right of (and the one "under") the marker will be un-diagrammed.

Each step through the syntax tables taken by ROUST causes another recursion of ROUST. That is, there is one level of recursion for every step ROUST has taken in the tables. Hence, if at any point ROUST discovers that it has mistakenly taken the wrong path, it may "back-up" and try other paths which might lead to a vaild diagram of the input string. Not until ROUST has examined all possible paths and found that none of them lead to a correct diagramming of the pertinent symbols of the input string will it indicate that the GOAL cannot be found.

The decision whether or not a path is a valid one is made by consulting the Boolean matrix PREC. Recall that PREC gives information whether the beginning syntactic component of any sentence can ever lead to another given syntactic element by progressing through the sentences from left to right. In the progression through the sentences, ROUST checks PREC every time it comes to the beginning of a new sentence to see whether or not it can get to the GOAL by following the path beginning at the new sentence. If PREC says yes, it continues through the sentence. If PREC says no, ROUST backs up to look for another path.

The progression through the syntax tables is as follows: Beginning at some sentence (we will explain how the process begins in a mement) ROUST proceeds left to right across the sentence checking to see whether the symbols in the input string will form (in order) the syntactic components specified by the sentence. As long as the input string meets the requirements of the components of the sentence, ROUST continues to the right until finally it reaches

the subject of the sentence. It then proceeds to the sentence whose first component is the subject it has in hand and continues through that new sentence.

Which sentence is used as the first one in any sequence is determined by the elements of the input string. In order to find the starting sentence, ROUST picks up the element of the input string just after the marker, and takes the sentence which has this symbol as its first component as the starting sentence of the sequence.

The tree form of STAB, of course, makes clear the alternative paths that may be taken in progressing through the sentences. Every time PREC indicates that it will not pay to continue through a sentence or the input string will not form a syntactic component in a sentence, ROUST backs up to the last junction in STAB that it passed, and tries the alternative path. If it has examined all the alternatives and is forced to back up to the beginning of the first sentence it started with, then it indicates that the input string cannot be diagrammed as requested.

The checking of syntactic components of a sentence to determine whether or not it is valid to continue through the sentence requires, of course, another call of ROUST, since, in fact, it is precisely the function of ROUST to perform such a check.

The progression through the syntax tables and the decision process involved are best illustrated by an example. Consider a system described by the following seven syntactic sentences.

A =:: SVAR ==> {0 A}

B =:: SVAR ==> {0 B}

C =:: SVAR ==> {0 C}

SVAR =:: TERM ==> {LDA ρ_1}

TERM ⊗ SVAR =:: TERM ==> {ρ_3
FMU ρ_1}

TERM =:: TSUM ==> {ρ_1}

TSUM + TERM =:: TSUM ==> {ρ_3
STA 0 t
ρ_1[t ← ti]
FAD 0 t}

These sentences would be written using the notation of the actual implementation as follows:

A! SVAR $Ø A3

B! SVAR $Ø B3

C! SVAR $Ø C3

SVAR! TERM $LDA 6 1 9 3

TERM X SVAR! TERM $6 3 9
FMU 6 1 9 3

TERM! TSUM $6 1 9 3

TSUM + TERM! TSUM
$6 3 9
STA Ø t
61 ¢ t # ti 9
FAD t 3

(Note: These sentences prescribe a very simple arithmetic scan similar to the one used in the implementation of ALGOL 60. Needless to say, the ALGOL specifications are more complicated, partly to gain increased efficiency in the output program, but the elemental ideas are the same.)

For the prupose of explaining the action of ROUST, we shall draw the tree storage map of these specifications in the form of a matrix in which the rows form the sentences of the description and the linkages are indicated by arrows. This form of map will allow us to speak more easily of areas and elements of the tree:

TRAN		STAB										
		1	2	3	4	5	6	7	8	9	10	
A	1	SVAR	$	ø	"SP"	A	$\overline{3}$					
B	2	SVAR	$	ø	"SP"	B	$\overline{3}$					
C	3	SVAR	$	ø	"SP"	C	$\overline{3}$					
SVAR	4	TERM	$	L	D	A	"SP"	$\overline{6}$	1	$\overline{9}$	$\overline{3}$	
TERM	5	X	SVAR	TERM	$	$\overline{6}$	3	$\overline{9}$	"CR"	"TB"	F	...
	6	TSUM	$	$\overline{6}$	1	$\overline{9}$	$\overline{3}$					
TSUM	7	+	TERM	TSUM	$	$\overline{6}$	3	$\overline{9}$	"CR"	"TB"	S	...

The Boolean precedence matrix PREC would have entries for the above table as follows:

	A	B	C	SVAR	TERM	TSUM
A	0	0	0	1	1	1
B	0	0	0	1	1	1
C	0	0	0	1	1	1
SVAR	0	0	0	0	1	1
TERM	0	0	0	0	1	1
TSUM	0	0	0	0	0	1

Note in the tree diagram of STAB that the arrows indicate links explicitly entered in the tabling of STAB. Entries across rows of the matrix would be entries in successive words of memory in the tabling of STAB, but there is neither an explicit nor implicit link between rows unless there is an arrow shown in the diagram. In the following examples, we will refer to "addresses" in the syntax table by giving ordered pairs (p,q). For example the ordered pair (7,3) picks the element of STAB which is at the intersection of row seven and column three, namely the element TSUM. Similarly (6,1) would be TSUM, (5,10) would be F, and so on.

We now give an example of the operation of ROUST in scanning the simple input string

A + B X C ;

To explain the scan of this statement, we list in columns some of the parameters at the various levels of ROUST, and list one row for each step through the tables. We assume that the initial call of ROUST set the goal as TSUM.

Input Symbol	IVAR	GOAL	PREC[(IVAR),GOAL]	Recursion Level	Comments
A	1,1	TSUM	1	1	continue at SVAR
	4,1	TSUM	1	2	continue at TERM
	5,1	TSUM		3	check for X, not finding,
	6,1	TSUM	1	3	continue with TSUM
(+)	7,1	TSUM		4	check for +, finding, continue across the same line.
	7,2	TSUM		5	call ROUST to check for TERM.
B	2,1	TERM	1	6	continue at SVAR
	4,1	TERM	1	7	continue at TERM
(X)	5,1	TERM		8	check for X, finding, continue across the same line.
	5,2	TERM		9	call ROUST to check for SVAR.
C	3,1	SVAR	0	10	note that we have found GOAL. Return since PREC [SVAR, SVAR] = 0
	5,3	TERM	1	10	note we have found GOAL, but do not return as PREC[TERM,TERM] = 1. Instead, continue at TERM.
(;)	5,1	TERM		11	check for X, not finding, try alternate.
	6,1	TERM	0	11	PREC[TSUM,TERM]=0, so return. Since we found the GOAL along the way, note success.
	7,3	TSUM	1	6	note we found GOAL, but PREC[TSUM,TSUM]=1, so continue at TSUM.
(;)	7,1	TSUM		7	check for +, since there is no +, and there are no alternatives, return but note that we found our GOAL along the way.

In this table, the entries under the column "Input symbol" give

1. If no parentheses surround the symbol, the symbol which we used to find the entry in TRAN which led to the correct spot in STAB.

2. If parentheses surround the symbol, the symbol with which a comparison was made to determine whether a possible path through the syntax table was valid.

The entries under the column "Recursion Level" give the number of times ROUST has called itself without returning. If e is the total number of times since the beginning of the diagramming process that ROUST has been called, and x is the total number of times it has returned (to the routine which called it), then

Recursion Level = e - x

at any time in the operation of the program. Note also that the Recursion level gives at any time the number of separate data groups in BIN for ROUST. (see BIN, section 1.2.2.7).

In terms of a diagram picture, the following diagram shows when the lines of the diagram would be drawn. The correspondence of this picture with the table above is given by the column "Recursion Level", which is repeated in the picture.

Recursion Level	"Diagram" of the input string				
	A	+	B	x	C
1	⌊svar⌋				
2	⌊term⌋				
3	⌊tsum⌋				
4		⌊ + ⌋			
5					
6			⌊svar⌋		
7			⌊term⌋		
8				⌊ x ⌋	
9					
10					⌊svar⌋
10				⌊ term ⌋	
11					
12					
6		⌊ tsum ⌋			
7					

This example shows that ROUST continues calling itself recursively until one of two things happens.

1. The precedence matrix PREC indicates that continuation forward at the path indicated by the subject of an entry is invalid
2. A check of symbols in the string, or a call of ROUST to check for a syntactic term indicates that the input sym-

bols will not form the necessary syntactic units to continue on the path through the syntactic statement.

When either of these conditions occurs, ROUST returns (reducing the recursion level by one, of course) to either the error exit or the normal exit depending on whether or not the GOAL requested was found on the level from which ROUST is exiting. If the error exit is chosen, a test if made on the level to which ROUST returned to see whether or not the GOAL was found on that level. If the GOAL was found, ROUST puts into the output string the output quantity for that level (see below) and exits by the normal exit for that level. If the GOAL was not reached, ROUST tries any other paths possible on the current level, and if there are no more paths to try, exits via the error exit for the current level.

To summarize, the diagramming decisions are made as the recursion level increases. That is, as long as the paths chosen continue to be valid, the recursion level increases with each step. When a path is discovered to be invalid, ROUST reduces the recursion level, and tries another path unless a GOAL has been reached somewhere along the way up. If, at any point during the backward progression (i.e., reducing the recursion level) the process reaches a point where a GOAL had been reached, then the downward progression is one where some output is given at each level, and an exit is made to the level below until the process comes to the point where the GOAL reached had been requested.

The downward progression is then one of two types:

1. Trying new paths or
2. Generating the output string

depending on whether or not a GOAL had been reached on the way up at some point. It is very important to note that the discovery of a GOAL does not necessarily mean that ROUST should return indicating sucess. It must return only when it has not only found the GOAL but also has used as many symbols in the input string as possible to do so.

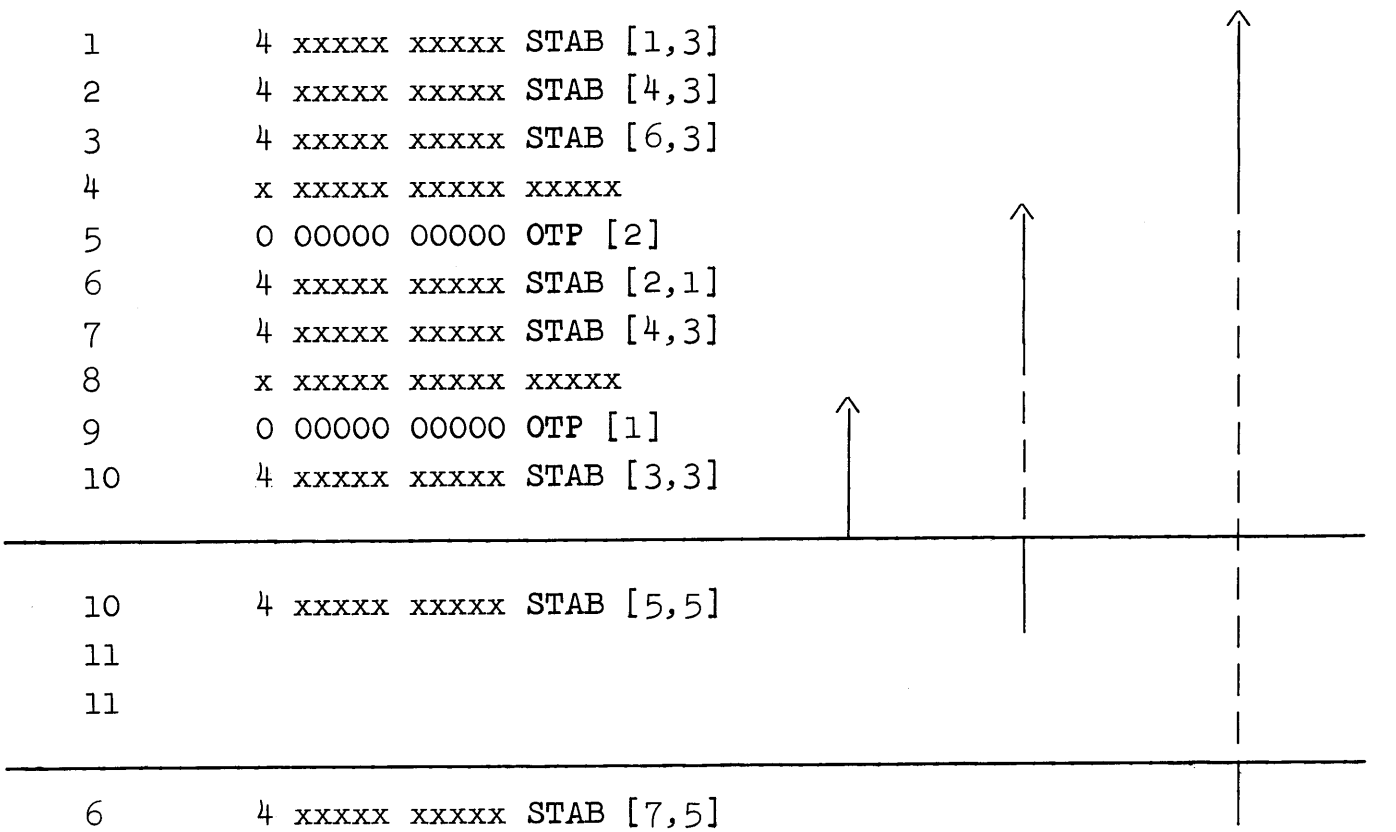
The generation of the output string, accomplished during some downward progression (i.e., reducing the recursion level) is accomplished by means of the quantity OTCEL which is local to the level of recursion. OTCEL holds the tentative output entry for the level of recursion. If the action at level P on the way up was the checking of components in the input string (perhaps by calling ROUST) then OTCEL for level P contains a type 2 OTP entry (the indirect type) referring to the section of OTP where the output links for the component being checked are in OTP. This address is put in $OTCEL_P$ when the GOAL was reached, say at level Q, (Q > P). The information that $OTCEL_P$ was to contain this quantity was passed to level Q as a parameter in the same manner that the GOAL was passed.

If the action at level P was to continue the diagramming process by starting at a new sentence in the syntax table (e.g., levels 1, 2, 8 ... in the example) then $OTCEL_P$ will contain an entry of type 1 for OTP (the direct link type) which gives the address in STAB of the definition corresponding to the sentence whose subject was the name of the new sentence selected for continuation.

The process is illustrated by the following table of output for the example given above. In this table we give the recursion levels as before, but now indicate the action taken on the downward progression rather than the upward progression. Hence to see the action taken as time progresses, one must start at the end of the table and work backwards. Assume that the output is built up in OTP starting at OTP [1].

Recursion level	Output
1	4 xxxxx xxxxx STAB [1,3]
2	4 xxxxx xxxxx STAB [4,3]
3	4 xxxxx xxxxx STAB [6,3]
4	x xxxxx xxxxx xxxxx
5	0 00000 00000 OTP [2]
6	4 xxxxx xxxxx STAB [2,1]
7	4 xxxxx xxxxx STAB [4,3]
8	x xxxxx xxxxx xxxxx
9	0 00000 00000 OTP [1]
10	4 xxxxx xxxxx STAB [3,3]
10	4 xxxxx xxxxx STAB [5,5]
11	
11	
6	4 xxxxx xxxxx STAB [7,5]

Time Sequence
(left to right, bottom to top)



The resulting output string in OTP may be placed in evidence by following the time sequence arrows. At the end of the entire process, OTP would have the following entries:

J	OTP [J]	
1	4 xxxxx xxxxx	STAB [3,3]
2	4	STAB [5,5]
3	0	OTP [1]
4	x	xxxxx
5	4	STAB [4,3]
6	4	STAB [2,1]
7	4	STAB [7,5]
8	0	OTP [2]
9	x	xxxxx
10	4	STAB [6,3]
11	4	STAB [4,3]
12	4	STAB [1,3]

where xxxxx indicates irrelevant quantities.

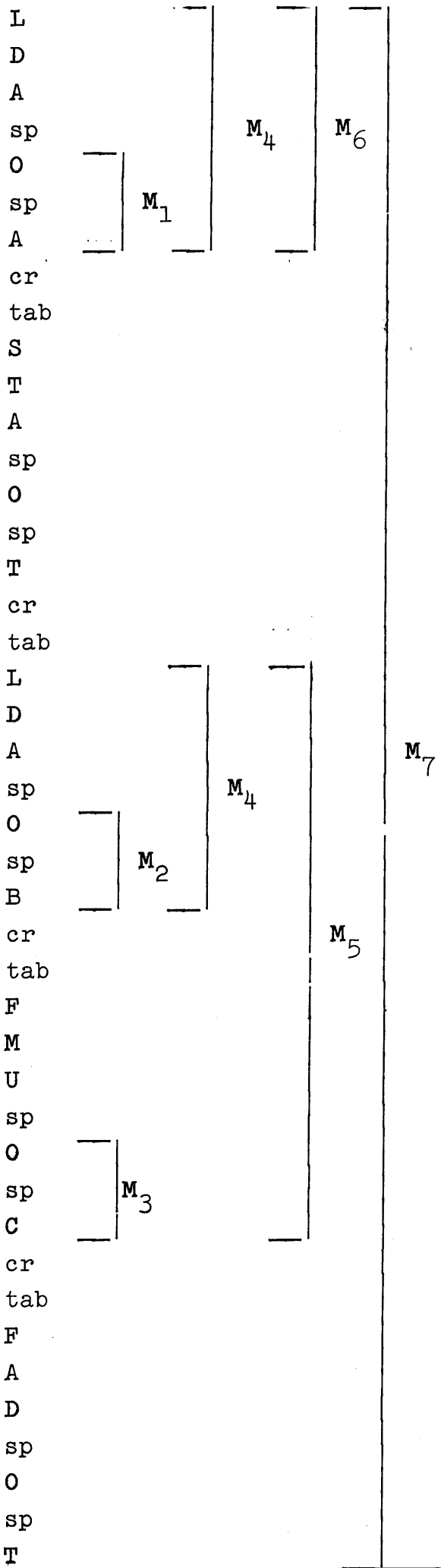
If we denote the definition in row k of STAB by M_k, then we may indicate the contents of OTP by:

J	OTP [J]
1	M_3
2	M_5
3	OTP [1]
4	xxxxx
5	M_4
6	M_2
7	M_7
8	OTP [2]
9	xxxxx
10	M_6
11	M_4
12	M_1

Expressed in more conventional function notation (see OTP section 1.2.2.5) the string in OTP would be:

$$M_7\ (M_5\ (M_3\ ,\ xxxxx,\ M_4\ (M_2)),\ xxxxx,\ M_6(M_4(M_1)))$$

Recalling the symbol strings which composed the definitions M_1 through M_7, the formation of the final output string (which TRANSYM generates from the string in OTP) would be:


L
D
A
sp
O
sp
A
cr
tab
S
T
A
sp
O
sp
T
cr
tab
L
D
A
sp
O
sp
B
cr
tab
F
M
U
sp
O
sp
C
cr
tab
F
A
D
sp
O
sp
T
M1
M2
M3
M4
M4
M5
M6
M7

which reproduced on printing equipment would yield:

```
LDA  0  A
STA  0  T
LDA  0  B
FMU  0  C
FAD  0  T
```

which is the translation of the input string specified by the metalinguistic syntax sentences.

In the example given in the preceding paragraphs, the only junction in the tree form of the syntax table was at the components following **TERM**. (X and **TSUM**). In this case, X was a component of a sentence and **TSUM** was the subject of a sentence. Since the tabling of the metalinguistic sentences is by the order of the sentences, it happened that we specified by ordering the sentences that **ROUST** should check for X after it had a **TERM** before it should decide to call the **TERM** a **TSUM** and continue at the sentence beginning with **TSUM**. Had we reversed the two sentences beginning with **TERM** in the original specifications, **ROUST** would have gone directly from **TERM** to **TSUM** without checking for the symbol X, and would, in fact, check for X after it had eventually discovered later that the input string would not form a **TSUM** at that point. It is clear that the diagramming process would still work in this case, since we would eventually check the input string against all the symbols which could follow **TSUM** (namely +) and find that X was not one of these, and so come back to the check for symbols which could follow **TERM** (namely X). However, since the successful paths through the syntax table are <u>all</u> eventually determined by the symbols of the input string, it would seem wise to check for symbols at any junction before wandering off to the beginnings of any other

sentences. Many times this will enable a decision to be made much more quickly (as is clearly the case with this example). Of course the whole question of which sentences are investigated first can be determined by the ordering of the sentences in the original set of specifications. But it is often difficult to see in a large set of specifications how the ordering will effect the efficiency of the diagramming process. At the very least, it is a big nuisance to do any more than a minimal amount of ordering of the sentences in the original specifications. Hence, ROUST has been programmed to check all the components following a syntactic element in the table before it uses a subject to start off on a new sentence. Hence if the two sentences in the specifications of the example beginning with TERM had been reversed in order, the diagramming process would actually have been the same as the one given. (Intuition dictates that the best solution to the problem of diagramming efficiency would be a thorough automatic analysis of the specifications as they were being tabled to determine the best ordering, but this seems to be a non trivial provlem, and was left for contemplation on rainy evenings. The expedient adopted of having ROUST check components before taking new sentences has, in practice worked well enough to justify postponing considerations of automatic ordering.)

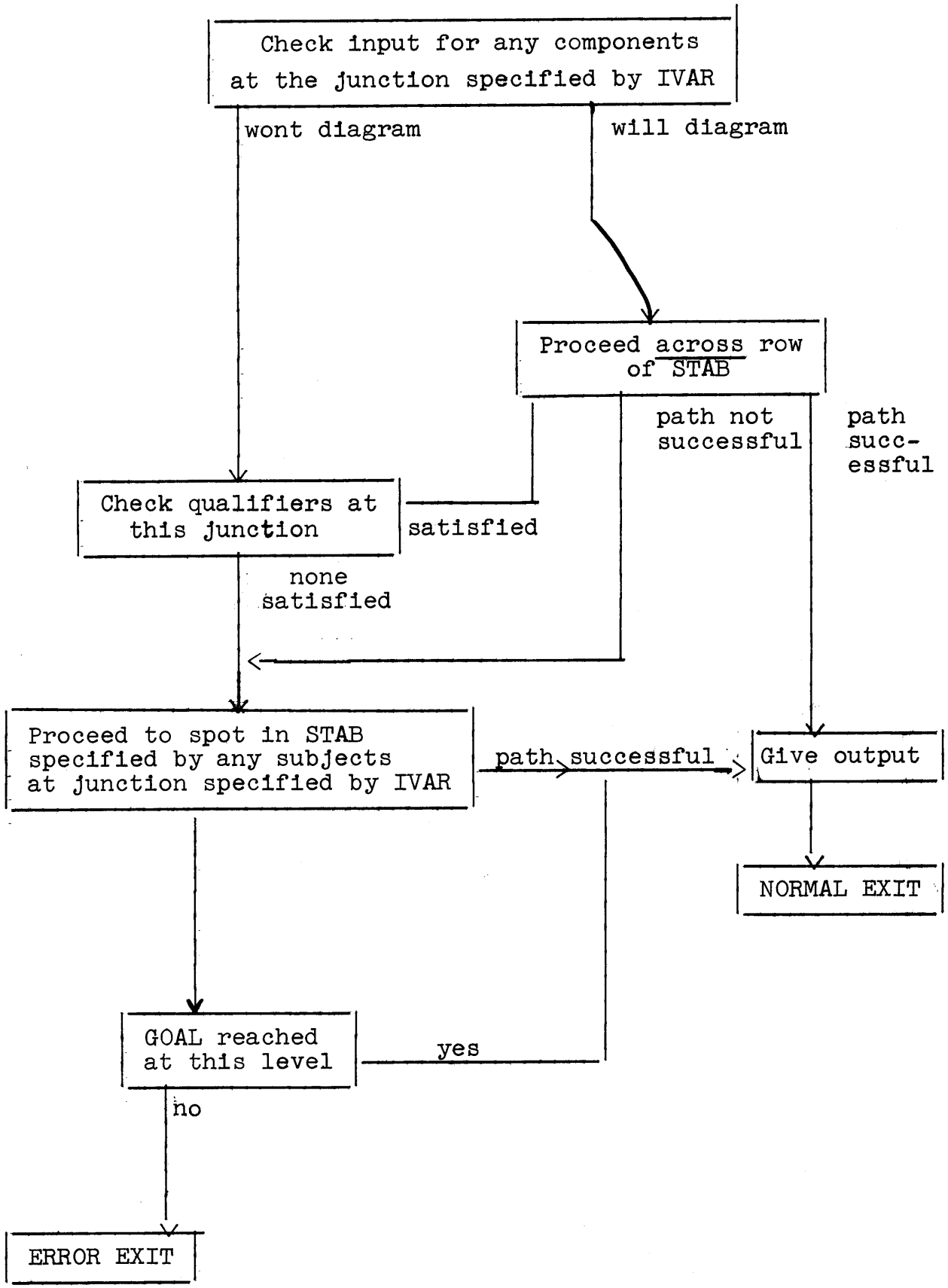
The other expedient adopted in ROUST to cope with a problem presented by the ALGOL language specifications, was to allow for special internal symbols which denote validity of syntactic sentences. The output of pass I of the compiler compiling an ALGOL program is a list of metalinguistic sentences specifying syntactic truths about the particular problem being compiled. These sentences essentially contain the information in the declarations of ALGOL, namely that

such-and-such a variable is a real variable, and another is an integer variable and so on. But ALGOL allows these declarations to apply only to parts of any problem, according to the block in which the declaration occurs, and hence if we specify that AXPQ, for example, is a real variable in one part of the program, indeed this does not imply that it is a real variable for the whole program; specifically it will not be if it is declared to be something else in another block. Therefore, it is necessary to specify in the syntactic sentences output by pass I over what range of the input string a given sentence is valid.

This specification of validity is accomplished by the use of special internal symbols 343_8 through 357_8. The compiling function $\mathcal{I}_5$, used during the first pass, enters the qualification words into TWOTRAN which give the range of the input string associated with each one of these special symbols (see section 1.2.2.2 -- TRAN). At each step through the syntax tables, ROUST checks the TRAN entry for each syntactic element of STAB that is passes to see if this TRAN entry is a qualification word (this decision is based on the sign bit of the entries of TRAN ... the sign bit is "1" for the qualification words, "0" for all other entries). If the TRAN entry is a qualification word, ROUST then checks to see whether the current index of the input string is within the limits specified by the qualification word. If the index is within limits, then the sentence in which the qualifier appeared is still considered to be valid; if not, it is considered to be invalid, and another path is investigated. However, since we do not wish to pass any junction without checking all sentence components at the junction, the qualifiers at any junction are not checked until all the components have been checked and found not to meet the restrictions imposed by the

input string. In addition, if there is more than one qualifier at any junction, the path following the qualifier with the smallest range containing the index of the input string is chosen, this convention being dictated by the block structure of ALGOL.

The following simple flow diagrams give with increasing detail the interconnections of the processes described in the preceeding paragraphs.


Check input for any components at the junction specified by IVAR
wont diagram
will diagram
Proceed across row of STAB
path not successful
path succ-essful
Check qualifiers at this junction
satisfied
none satisfied
Proceed to spot in STAB specified by any subjects at junction specified by IVAR
path successful
Give output
NORMAL EXIT
GOAL reached at this level
yes
no
ERROR EXIT

1.2.3.2 TRANSYM

TRANSYM is the subroutine which transforms the definition linking list which ROUST constructs in OTP into the final output string for a given pass of the compiler, forming this final output string in OTPX. As outlined in detail in the description of OTP (section 1.2.2.5) OTP is a Polish-like string which links the "functions" which are the definitions of sentences together. OTP is further arranged so that the link to the "parameter" strings of the definitions occur in successive words of OTP after the link to the definition in question. That is, for a definition referenced in OTP [k] the link to the nth parameter of that definition, (ρ_n) is found in OTP [k+n]. This arrangement of definition "links" allows TRANSYM to put together the definitions specified by OTP in a reasonably straightforward manner.

TRANSYM then, is basically composed of a set of interwoven subprograms which recognize the metasymbols which occur in a definition and take the appropriate action in each case. We shall explain here in general terms the action TRANSYM takes for each of the metasymbols.

1. The metasymbols $, ∃ and ∄ .

The symbols $ and ∃ are used as parentheses to enclose definitions in the syntactic sentences. However, since the definitions may be in face syntactic specifications, they may include definitions themselves, in which case special conventions must be put into effect while dealing with the inner definitions.

Of course, one of the primary duties of the parentheses is to indicate the end of the definition. ROUST specifies that the initial $ is not part of the definition. Furthermore, if the parentheses and $\overline{3}$ occur within the definition, they must be paired in proper parenthesis fashion; hence TRANSYM determines that it has reached the end of a definition when it has found one more $\overline{3}$ in the definition $'s.

Where the definitions of sentences are themselves syntactic sentences containing definitions, the 王 's serve to indicate which part of the syntactic sentences the parameter string designators ρ_n and function designators $\exists_n$ refer to. The convention is that if the number of 王's preceding a ρ or $\exists$ is equal to the number of $'s to the left of the ρ or $\exists$, then that ρ or $\exists$ applies to the outermost definition, and hence must be treated by TRANSYM. If not, the ρ or $\exists$ applies to an inner definition and in this case TRANSYM must pass along the ρ or $\exists$ together with the 王 's and all other metasymbols associated with the occurance of the ρ or $\exists$ (namely ¢, # or $\forall$) in their original form. That is TRANSYM must treat them as ordinary symbols. Hence, each time a series of 王 's is encountered, the number of them in sequence is counted, and compared with the current count of $'s thus far passed. If the counts are equal, the 王 's are <u>ignored</u> and the following ρ or $\exists$ is treated normally; if the counts are not equal, the 王 's and the metasymbols following them are passed on as normal symbols.

2. Paremater string designators, Im ¢P#PPP,,P¢P#PP.PI

TRANSYM fundamentally picks up the symbols in a definition in order and puts them into the output string OTPX. However, when a parameter string designator is encountered (assuming that it has the proper number of Ξ's preceding it) TRANSYM then calls itself, specifying the definition named by the parameter number n, to output the symbols of the parameter string in place of the designator. The primary input parameter of TRANSYM is the location in OTP of the link that specifies the definition TRANSYM must output. TRANSYM finds the address of the parameter string (definition) by consulting the entry in OTP which occurs n words after the link whose address was supplied as an input parameter to TRANSYM. In calling itself, TRANSYM specifies to the next level in the recursion the address in OTP of

1. this word (i.e., the one n words after the current OTP address) if the link in this word is direct, or
2. The address which this word contains if the link is indirect. (see OTP -- section 1.2.2.5)

When TRANSYM comes to the end of the second definition it exits to the last level of recursion, and TRANSYM continues on that level as before.

However, if the parameter string designator is followed by a substitution list, TRANSYM enters the indicated substitutions into a substitution tables SUBL and SUBX before calling itself to effect the instantiation of the parameter string. The variable to be re-

placed in the parameter string is entered into SUBL as described in 1.2.2.8, and the string following the # is entered into ISUBIX as indicated in 1.2.2.9. Since, however, the string specified to replace the variable may in general be composed of any of the allowable symbols of a definition, including the metasymbols, it is necessary to analyze the symbols before putting them into ISUBIX so that they are reduced to the "proper" string containing no metasymbols. Since this task is exactly the function of TRANSYM, we indeed call TRANSYM itself to reduce the substitution string to the simple form. In order that TRANSYM convert only the substitution string, we specify that TRANSYM shall exit whenever it encounters the metasymbols ¢ or ∑ since one of these two symbols must appear after each substitution string. Once all the substitutions have been entered into the substitution tables, TRANSYM proceeds to call itself for the instantiation of the parameter string as in the case where no substitutions were indicated. But when TRANSYM returns this time, any substitutions entered into the substitution tables for the instantiation of the parameter string are removed from the substitution tables. Therefore, at any given time during the operation of TRANSYM, the substifution tables contain a list of the substitutions currently in force.

Each simple symbol is checked against the substitution tables before it is put into the output string OTPX (see "simple symbols", below). Note that it is possible to have several strings specified to be substituted for the same variable in the substitution tables at one time. In this case the last entry (the latest in time) is the one valid for the substitution. Nested substitutions are effected nor-

mally by this process, since all the symbols in the substitution strings tabled have been reduced by TRANSYM and therefore have had substitutions indicated by the previous contents of the table done on them.

3. Function designators Ξ n ¢ PPP...P ¢ PPP..P ¢ ... Ξ

Function designators in the definition string of any sentence indicate that a special string function n should now be called to transform the parameters of the function designator into a string and place this "output" string in TRANSYM's output string OTPX. Hence, when TRANSYM encounters the metasymbol Ξ in its progress through a definition, it merely transfers control to the indicated function. If this function has parameters (some do not) these parameters (i.e., the strings enclosed in the meta brackets ⟦ and ⟧) are placed at the current end of the output string OTPX, where they may be used by the function being called. As for the substitution strings, the function parameter strings may contain any of the symbols legal in a definition, so TRANSYM is called to effect the reduction of these parameter strings to the simple form in exactly the same manner as for the substitution strings explained in the last paragraph. Since TRANSYM normally places the reduced string in OTPX, it is just left there for the function, and the first and last address of the parameter strings are supplied to the function so that it can locate them in OTPX.

4. Simple symbols

The main loop of TRANSYM sequences through the symbols of the definition TRANSYM is working on at any recursion level and as described above checks them for the metasymbols which indicate special action. Most of the symbols, however, will be simple symbols and, in this case, the symbol is merely transferred to the next available space in OTPX. However, before the symbol is put into OTPX, TRANSYM must consult the substitution tables to make sure that the symbol is not on the list of symbols to be replaced by a string. If the symbol is in the substitution list, then, of course, the string indicated by the substitution table is put into OTPX in place of the original symbol.

The following brief flow diagram indicates the relation of the operations described above to each other, and provides the position correspondence of the operations to the code in the listing which carries them out.

APPENDIX

USE OF THE INPUT-OUTPUT TRANSLITERATION ROUTINES

The translator will consist of three closed subroutines: INTRAN, OUTRAN, and DEFINE. We define a character as the unique action of an external equipment when directed by (or producing) an n-bit code, when n is the number of bits required to define all possible actions of any particular external equipment (usually 6). Examples of characters are

P p (. 3 { = carriage return lower case

Define a symbol as a group or string of characters which are to be represented in the machine by a p-bit code, where p is the number of bits required to represent the entire set of internal codes. Again, p may be 6 bits, but it is intended that the internal code be of any size desired by the programmer. Examples of symbols are

begin end step LDA EQS A 2

INTRAN:

INTRAN is a closed subroutine with one output parameter -- that parameter to be transmitted via the accumulator. A return jump to INTRAN will cause the following action: Characters will be read from some previously specified input equipment (see DEFINE) and translated according to a previously specified table, into one symbol. Control will then be returned by the return jump mechanism with the translated symbol appearing at the right end of the accumulator. If the symbols read from input equipment cannot be translated, a specified symbol distinct from the defined symbols will be given as the output. If there are no more characters available from the external equipment (as when the input operation has been concluded), a character distinct from the above characters will be returned as output.

At the time when INTRAN outputs the end-of-input character, it will restore itself for a new input string. Hence, the first time a transfer to INTRAN is executed after its output was the end-of-input character, it will initiate a new input from the same external equipment and using the same external code as for the last input operation (unless the code or equipment has been re-defined by operation of DEFINE).

OUTRAN:

OUTRAN is a closed subroutine with one input parameter, that parameter to be transmitted via the accumulator. A return jump to OUTRAN will cause the following action: The internal

symbol found in the right end of the accumulator will be translated to a string of symbols according to the previously defined table, and output on the previously specified output equipment. The last symbol supplied to OUTRAN in any output sequence must be the symbol used to denote end-of-input. Undefined symbols will not be translated, but will be ignored. In case there are several strings of characters representing the same internal symbol (synonyms), the output will be the one specified by the table (see DEFINE).

DEFINE:

DEFINE is a closed subroutine with several input parameters which, when executed, will construct translation tables to be used by INTRAN and OUTRAN and specify the external equipment to be used for input-output operations. These definitions will remain unchanged until DEFINE is executed again. The operation of DEFINE is determined by its input parameters:

1. If the A register is zero, the input-output equipment to be used by INTRAN and OUTRAN will not be redefined.

2. A non-zero upper address of A specifies the entry address of an "actuate input equipment routine" to be used with subsequent input operations. The specifications for such routines are found on page .

3. A non-zero lower address in A is the address of an "actuate output equipment routine" to be used with subsequent output operations. The specifications for such routines are found on page .

4. If the Q register is zero, the translation tables will not be redefined. If the Q register is positive and non-zero, the translation tables will be defined as those tables beginning in the location in memory whose address is in the lower address position of Q. If the Q register is negative, a new set of translation tables will be generated at the address specified in the lower address position of Q from the character string available at the input equipment specified in the A register. The terminal address of this newly defined translation table will be given as an output parameter in the lower address position of the accumulator. If the contents of the lower address position of A is zero, the table-defining character string will be expected to be available at the last specified input equipment. If new tables are being constructed (sign bit of Q=1) then the upper address potition of Q must contain either 200_8 or (the maximum internal symbol value) + 2 which ever of these is greater in value.

The character string defining the translation tables is designed so that the translating routine is completely independent of the particular equivalence table being used. Since DEFINE cannot have a priori knowledge of the representation of the characters of the definition string, it must assign meaning to these characters by virtue of their position in the string. However, the equivalence table may have the following properties:

1. A symbol may be composed of more than one character.
2. More than one symbol may be placed in correspondence with one internal code.
3. The length of the equivalence table is arbitrary.

Therefore it is necessary to define several metasymbols for punctuation before the body of the defining string may be presented. One of these characters is used to terminate each symbol. Another is used after a list of symbols equated to the same internal code. A third is used to terminate the defining string. (We might define the first to be a comma, the second to be a carriage return, and the third to be a Flexowriter stop code, for instance.)

The assignment of groups of symbols to internal codes is by position. Hence, following the group of characters defining the metasymbols is the group of symbols to be assigned the internal code zero, then the group of symbols to be assigned the internal code one, then two, and so on.

In order to describe the composition of the defining string more formally, define a format character as a character which is used in this string as a separator, or character to be used for defining the tables, and denote such characters by f[i] where i indicates the particular format character in question. There are 4 format characters which must occupy the first positions in the definition string:

Character 1. The upper Case character (if used) (denoted f[1].

Character 2. Any character (essentially a positioning character).

Character 3. The Lower Case character (if used) (denoted f[3]). If the alphabet used for the external code does not contain upper case and lower case characters (as for IBM equipment), Characters 1 and 3 must be the same character, though they may be any character in the alphabet. (In this case, the fact that they are the same character is the only information used by DEFINE.)

Character 4. Any character (as character 2, a positioning character)

Character 5. (denoted f[5]). This character will be used in the following string of characters to denote the end of a symbol.

Character 6. (denoted f[6]). This character is the character which will be used in the following string of symbols to denote the end of a character string associated with a particular internal symbol.

Character 7. Beginning with character 7 is a list of symbols which are to be ignored by INTRAN. Following this string of symbols must be the character f[6]. (This list may be empty, but the character f[6] must be present whether the list is empty or not).

Following the above string of characters will be a string of characters designating a symbol to indicate the end of input strings called by INTRAN. This symbol will also be used to indicate the end of the definition string. The symbol may be any string of characters. The symbol must be followed by the character f[6]. This ends the preface to the translation table.

Following the above string of characters is a list of symbols separated by the characters f[5] and f[6], which defines the translation. The list of symbols before the first f[6] is the list of symbols which INTRAN will translate into the internal symbol zero. The list of symbols following that f[6] and preceding the next f[6] is the list of symbols which INTRAN will translate into the internal character one, and so on. Every symbol in the list MUST be followed by the character f[5]. The lists may each be composed of an arbitrary number of symbols, and each of the symbols in the list may be composed of an arbitrary number of characters.

The characters f[5] and f[6] may be defined as symbols, but they may not be used as characters in a symbol composed of more than one character. (f[5] followed by f[5] defines f[5] as a symbol, and f[6] followed by f[5] defines f[6] as a symbol.)

The last character of the definition string must be followed by the "end-of-input symbol" which appeared at the head of the string.

The internal symbol one greater in value than the last defined internal symbol will denote the "end of input" symbol for INTRAN and OUTRAN.

The internal symbol two greater in value than the last defined internal symbol will denote the "error" symbol which INTRAN outputs upon discovery of an illegal input sequence.

Example

The following is an example of a definition string as it might have been prepared on a Flexowriter to define the Flexowriter codes to the translating routine. In order to give a complete indication of the characters punched on paper tape, non-printing characters have been noted at the right margin where it is not obvious that they have produced a punch.

In this example, f[3] is a comma, and f[4] is a carriage return.

```
Xx,   Preface, which was                UC X LC x , CR
      actually produced by              delete tape feed BS CR
      these  operations                 stop code CR

      0,
      1,
      2,
      3,
      4,
      5,
      6,
      7,
      8,
      9,
      a,
      b,
      c,
      d,
      e,
      f,
      g,
      h,
      i,
      j,
      k,
      l,
      m,
      n,
      o,
      p,
      q,
      r,
      s,
      t,
      u,
      v,
      w,
      x,
      y,
      z,
      :=,
      OR,
      AND,
      =,
      +,
      -,
      X,
      /,
      BEGIN, BEG,
      END,

                                        stop code
```

EXTERNAL EQUIPMENT ACTUATING ROUTINES

These routines, which may be many in number, will be closed subroutines providing a link between the translation routines and any external equipment which may be associated with the 1604 at various installations. It is intended that these routines should have no knowledge of ANY codes associated with the external equipment, but should only be capable of operating the equipment and passing along the information received from the equipment to the translator, or from the translator to the equipment. The purpose of so separating the functions of translation and equipment actuation is to minimize the programming effort necessary to make routines operating on one type of external equipment operate on another type. There will be two types of routines, those for input and those for output.

INPUT ROUTINES.

Each INPUT routine will be a closed subroutine (entered with a return jump.) Each time an INPUT routine is called it will provide one character from external equipment and leave it in the lower part of the accumulator. The accumulator will normally be positive on entrance. If the INPUT routine is entered with the accumulator negative the routine will adjust itself so that the next entrance will initiate a new input sequence and return. On this last type of operation of the INPUT routine, the contents of the accumulator will be considered to be irrelevant upon exit.

OUTPUT ROUTINES.

Each output routine will be a closed subroutine (entered with a return jump.) Each time an OUTPUT routine is entered, the accumulator will either be negative or will contain a character in the lower part of the accumulator. If A is positive, the routine will output the character it finds in A. If A is negative, the routine will adjust itself so that the next time it is entered a new output sequence will be initiated. [[Note that although the routines handle only one character for each operation, it is clear that they may not necessarily input or output one charachte at a time, since it is entirely possible for the routines to store characters in their own storage until a convenient number of characters for input or output have been accumulated.]

APPENDIX

DETAILED WORD DESCRIPTION OF THE LISTINGS

This appendix contains a detailed description of the parts of the compiler written to correspond directly to the listings of the compiler. In these descriptions, the tags in the listings are listed in order at the left margin and the description of the commands following these tags in the listings follows the corresponding tags.

TRANSYM: Detailed word description of the listing:

TRANSYM: Set P and Q to 0. P counts the left parentheses $ in a definition being outputted, and Q counts the "parentheses counters" ⌶.

PI: {L[B1] is the next symbol in the definition being outputted. If this symbol is $, add 1 to P and go to B203, which will put the $ in the output string and start over at PI with a new symbol}

B200: {If L[B1] is a closing parenthesis ⌶, check P to see if it is the outermost one. If it is, return, if not, go to B203 to put it in the output string.}

B201: {If L[B1] is a left parenthesis counter ⌶, then look at the following symbols L[B1] to see how many of them are also ⌶ s. If the number of them equals the number of left parens we have passed so far (according to P) then

B204 discard all the ⌶ s and proceed to SUBS to perform the function requested by the next symbol. If the counts are <u>not</u> equal then put into the output string all the ⌶ s, and proceed normally}.

B202 {We arrive here if the symbol L[B1] was neither ⌶ nor $, or the ⌶ s indicated that parenthesis counts were not equal. Test P to see if substitutions

are to be made. If so, proceed to SUBS which will check the symbol against the rest of the special metasymbols. Othersise...}

B203: {Output the character under consideration L[B1]. Check PUTIKR to see whether we are outputting into the output list final output characters or function or string substitution parameters. If the former, dump OTPX onto output equipment by calling ASSEMBLE. In either event, put the symbol in the output list, up the index of the list and go back to PI to go through the works again for the next symbol in the definition under consideration.}

SUBS: {Check to see if the symbol L[B1] is any of the metasymbols $\underline{6}$, $\underline{5}$, ¢ , or $\underline{9}$. If it is any of them, proceed to the routine which handles that type of symbol (getting the address of the routine from TLISTLNK). The routines are PARAM, FCN, RETURN, and RETURN respectively. If the symbol is none of these, then it must be a simple symbol.}

SIMSYM: {Having arrived here, the only thing left to check is whether the simple symbol we know we have is in the substitution list or not. If it is NOT, proceed to B203 to put the symbol in the output list. If it <u>is</u> in the substitution list, set up the loop which outputs the string which the substitution list says is to be substituted for the symbol.}

B210: {This begins a loop which outputs the symbols in the substitution list in place of the symbol for which substitution has been indicated. The loop is terminated by finding the character 377_8 in the string.}

CVRT: {This is a short closed subroutine which converts the octal digit L[B1+1] and the longest string of octal digits following it to an octal number. It advances B1 in doing so so that L[B1] is the first non digit character after the digit string when the routine exits. The octal number is left in the accumulator upon exit.}

PARAM: {This routine is the one invoked when L[b1] is found to be **6** . We first find the number of the "parameter" of the definition that is wanted (by looking at the octal digits following the **6** , then add RR to this local digit to give us the machine address of the link word in OTP which tells us where the definition wanted is. RR always contains the address in OTP of the link which specified the definition we are currently working on. Therefore adding n to it gets the address in OTP of the nth parameter of the definition. (See OTP, section 1.2.2.5). We store the address of this parameter link in TE, and then check to see whether the next symbol is a **9** . If it is, we proceed to MACFORPM, which calls TRANSYM to start working on the definition which is specified as the parameter (whose link we have

just stored in TE. If the next symbol is not a 9 , we know that the string designator has some substitution specifications attached to it, and we proceed to enter these in SUBL and ISUBIX before going to MACFORPM to start working on the new definition.}

STARTSUB: {Recall that strings to be substituted for a symbol may, in general contain any symbols of which may be put in any definition. Since TRANSYM is designed to handle all these symbols, the easiest way to handle all the cases that might turn up in a substitution string is to call TRANSYM to straighten them out to their final form (i.e., handle string designators function designators, etc. which might be in the substitution string.) Hence, at this point, we do, in fact, call TRANSYM. In doing so, we must, of course, increase B5 by 11 so that the data we are now working on will not be destroyed by the call. At the same time we make sure that all the quantities we will need when TRANSYM comes back are safely tucked away in BIN (namely we store SUBIX; the contents of B1 and B6, etc.) Also we pass along the value of RR, since the next call of TRANSYM will be working on the same definition that we are now working on. (See the last paragraph for explanation of RR). Just before going to TRANSYM we change the value of PUTIKR so that the output symbols from TRANSYM will remain in OTPX where we can recover them, rather than being dumped out

on external equipment. (See B203, above.) We also must store the address to which we wish TRANSYM to return in EXIT (for the next level). We indeed know that TRANSYM will come back after having dealt with the substitution, since the symbols ¢ and ↯ will cause TRANSYM to return to whatever called it. (See SUBS above).

B213: {Here begins a "start-in-the-middle" loop. Notice that
B212 TRANSYM comes back to B212. Beginning at B212, we set up the loop to transfer the output string provided by TRANSYM to ISUBIX so that it will be available as a substitution string. We remembered to tuck the contents of B6 away in BIN before calling TRANSYM, so we can tell how many symbols were in the final string by comparing that value to the present contents of B6. Having set up the transfer loop, the end test for the loop is made <u>first</u>, rather than after one transfer since it is possible to specify null strings to be substituted for a symbol. We then proceed around the loop the main part of which begins at B213 until the string has been transferred to ISUBIX.}

B214: {Having put the substitution string in ISUBIX, we add a 377_8 to the end of the string to mark its end and then check the next symbol in the definition to see whether there are more substitutions specified. If there are, we go back to STARTSUB to do the whole operation over

again. When finally we come across the $\overline{9}$ which indicates the end of the list of substitutions in the string designator, we proceed to MACFORPM.}

MACFORPM: {Having by now entered all the substitutions which were indicated into ISUBIX, we may proceed to call TRANSYM to go to work on the definition which was specified by the string designator, and the address of whose link is stored in TE (see PARAM above). We have not yet determined, however, whether this link is a direct or indirect link. We check this by looking at the sign bit of the link (see OTP, 1.2.2.5). If the link is indirect, we get the entry in OTP whose address is given by the indirect link. Having done this, we now know that we have a direct link, since we are allowed only one level of indirection in OTP. We supply this direct link to the impending call of TRANSYM, storing it in RR (in the next space in BIN). (This, of course, will allow the next level of operation of TRANSYM to find the parameters of the new definition (if it has any in the same manner this level did (see PARAM above)). As for every recursive call of TRANSYM, we must tuck away in the section of BIN reserved for this level of recursion everything we wish to have when TRANSYM returns. This we do. We supply the exit address to the next level, and call TRANSYM. When TRANSYM returns, we know that all the output symbols which should have replaced

the string designator which caused the call of TRANSYM have indeed been put into OTPX and that B6 has been incremented appropriately, so we may now continue considering the symbols of the definition we were working on at this level. We do this by transferring to PI after having restored to their proper places the things we tucked away in BIN}.

FCN: {This part of TRANSYM is executed when it is discovered that the symbol under consideration is the function designator 5 . The first thing done here is to return jump to CVRT to get the number of the function which should be executed. This number is then added to the base of a transfer vector (the first entry of the transfer vector being in FNBASE) to give the address to which we must jump for the execution of the function. This address is stored in TE. We then test to see whether or not the function is to have any parameters by looking at the first symbol after the function number. If that symbol is a 9 , the function has no parameters, and we go to EXECUTE which transfers to the function.}

B220 {We arrive here only if the function has parameters. The action taken if indeed the function does have parameters follows the same line as for a string designator with a substitution list. We call TRANSYM to output into OTPX

the untangled string of symbols which is the first parameter of the function. In doing so, we pass along to the next level of TRANSYM the value of RR, since the next level of TRANSYM will be working on the same definition that this one is working on. Also PUTIKR is set to indicate that TRANSYM should retain the output string in OTPX rather than putting it on external equipment, since, of course, the function must have the parameter available for its operation.

TRANSYM will return after having unravelled all the symbols after the metacomma ¢ up to the first ¢ or ⨝ . (This because these symbols cause TRANSYM to return to the routine which called it --- see SUBS, above). When TRANSYM returns, we put the character 377_8 in the output list to indicate the end of the first parameter string and then check to see whether or not there are any more parameters of the function. If the next symbol in the definition string is a ¢ , then there are more parameters and we go back to B220 to have the next parameter built up in the output string. When we finally have converted all the parameters, we then proceed to EXECUTE, which finally calls the function.}

EXECUTE: {By the time the program has arrived here, all the parameters of the function to be called will have been built up in the output list (if it had any). Having saved the contents of B6 before we started converting

the parameters (in VKEEP) we may enter this quantity into the accumulator and transfer to the address we have kept in TE, which causes control to be turned over to the function in question. This function must be written to leave its output in OTPX and exit to PI. Hence, after the function has done its job, it returns to PI where TRANSYM begins operating on the next symbol of the definition.}

RETURN: This is a short exiting routine which gets the address in EXIT (which presumably had been put there by the routine which called TRANSYM) and, after having decremented B5 by 11 to reset the pushdown list of data to the proper place for the last level of TRANSYM, control is returned to the calling routine.}

ROUST: Detailed Word Explanation of Listings:

There are 8 parameters of ROUST which are local to the level of recursion. These are addressed, as are all such quantities in these routines with B5. Before giving the detailed explanation to follow of the sections of ROUST, we describe briefly these 8 parameters and their function.

IVAR (BIN+0) holds in the upper address position the absolute address in STAB which was specified as the point in STAB which ROUST should consider in the diagramming process on this level. ROUST uses the lower address position of IVAR as temporary storage to hold the contents of B1 at the beginning of ROUSTS operation on current level of recursion, so that B1 may be reset upon exit (B1 holds the address of STAB currently of interest)

GOAL (BIN+1) holds in the lower address position the internal symbol which is the GOAL for the current level of recursion. The upper address position of GOAL is the index of the input string which marks the symbol of the input string which will be the first symbol of the group which will form the GOAL should the GOAL be reached. If the GOAL is reached this quantity will be assigned to the position X of the Format 1 OTP entry which designates the definition associated with the discovered GOAL, and will

subsequently be used to form a qualification word in TRAN if this formation is called for. (See OTP, section 1.2.2.5, and the description of ALGOL function Compile function number 5, section 2.2.1.5)

MAC (BIN+2) holds in the lower address position the address of the cell in BIN (on a lower level of recursion) into which this level should put the address of OTP wherein begins the output for a discovered GOAL, if indeed the goal is reached on this level. The upper address position of MAC holds the address in STAB of the entry following the satisfied qualifier with the smallest range of the qualifiers at the junction of STAB considered at this level. If no such qualifiers exist at the junction in question, then this quantity (i.e., the upper address position) must be 00000.

REXIT (BIN+3) holds in the lower address position the address for the normal exit from ROUST. The upper address position of REXIT holds the address for the error exit from ROUST.

QUAL (BIN+4) holds the qualification word last found at the junction under consideration by ROUST which:

1. has a range including the index of INPUT
2. has the smallest such range of the qualification words thus far passed at the junction.

The initial value of QUAL at any level is set to the value of QUALCON, namely 400 77776 001 77776 indicating the range to be +77776 to -1)

QUALCON (and hence the initial value of QUAL) thus indicates the largest possible range of the index of INPUT, so no matter what the value of this index is it will always be contained in the initial range in QUAL. If no qualification words are passed whose range includes the index, then this initial range will be taken to be the range of validity of the sentence parts eminating from the junction, and therefore all these paths will be considered legitimate.

COUNTS (BIN+5) holds in the lower address position the value of B3 upon entrance to the current level of operation of ROUST, and in the upper address position the value of B4 at the same time. Hence COUNTS records the index of the INPUT string (B4) and the index of the OTP string (B3) upon entrance so that these quantities may be reset to those values if the error exit is taken from this level.

SW (BIN+6) is a switch parameter. During the operation of ROUST on the current level SW is -1 if the requested GOAL has not been reached at this level and 0 if the GOAL has been reached at this level.

PAR (BIN+7) (denoted as OTCEL at some points in the description)

holds the output word which will be placed in OTP at the current level if indeed there is to be anything placed in OTP from this level. Hence the address of PAR

for the current level is specified as a parameter of ROUST when ROUST is called to check a component of a sentence this parameter being known as MAC at the next level of recursion. (see MAC above).

Of these "parameters" some are assigned values by the last (i.e., lower) level of recursion: namely IVAR [upper], GOAL, MAC [lower], and REXIT; the rest of the parameters are used on each level as temporary storage, and have no relavent value upon entrance to ROUST, namely: IVAR [lower], QUAL, COUNTS, SW, and PAR.

The following pages give the detailed word description of the listing of ROUST

ROUST {The initial section of ROUST sets up local parameters for the operation of ROUST on the current level. In particular, it stores B3 and B4 in COUNTS so that in the event of an error exit from this level they may be reset, it sets the initial values of SW, MAC [upper], QUAL, and IVAR [lower]. In the operation of ROUST, B1 at any time contains the absolute address of the one element of STAB which is of interest at the exact moment. Logically, when ROUST calls itself therefore, it should save the contents of B1 in BIN, since a different section of STAB will be investigated by ROUST on each level of recursion. It turns out to be more convenient to have ROUST save and reset the contents of B1 upon entrance and exit than to have ROUST savethe contents of

B1 before calling itself, and reseting B1 after each return. Therefore, B1 is stored in IVAR [lower] upon entrance, and reset from IVAR [lower] on (either normal or error) exit.}

COMP {Here begins the section of ROUST which checks all the syntactic elements of STAB at the junction specified by IVAR [upper] to see whether they are components (or qualifying components) of a sentence. The subjects of a sentence among these syntactic elements are dealt with later in the program, (in the section labelled LINK). We begin by checking the first syntactic element at the junction, namely the one whose address is contained in IVAR [upper]. We check first to see whether this element is a qualifier by looking at the sign bit of its entry in TRAN. If the element is a qualifier, we transfer to LIMITEST which is the section of ROUST which deals with qualifiers. If the element is not a qualifier, then we check to see whether the element is the subject of a sentence by testing the element after the one in question. If the element in question is a subject, the element of STAB following it will be a $ (internal character 161). If the element in question is not a subject, then it must be a component of a sentence and we proceed to the processing of this component. If the element is a subject, then we proceed to LINK which changes B1 to the address of the next element at the junction (see LINK).

In processing the component of a sentence, we wish to initiate a call on ROUST to see whether the elements of the input string will form the syntactic structure named by the component. In order to do this, we first read the next column of the input string. Since ROUST moves back and forth across the input string, it may be that the element has already been read into the machine. We check to see whether this is the case by adding CURIN to the contents of B4. CURIN in a global (i.e., not local to the level of recursion) parameter of the routine whose value is the negative of the index of the last element of the input string which has been read into the machine. Hence if the addition of CURIN to the contents of B4 yields a positive number we know that the symbol we want has not yet been read into the machine, and we return jump to GETNS to have the symbol read in. Otherwise we merely increment B4 so that it will pick out the next element of the input string.

Having properly incremented B4 (and, if necessary, read in another character), we now test this character to see whether it is <u>equal</u> to the syntactic component which we must form to proceed across the sentence. (Recall that basic symbols of the language are considered to be syntactic elements). If the symbol of the input string is the same as the syntactic element, then we set SW to 0 to indicate that the element has been found. However,

it is possible that even though the two are equal that the symbol may be the first element of a syntactic construction which is again the element, so we must check PREC to determine whether this symbol will lead through a sequence of paths in STAB to form the same symbol. (Consider for example the syntactic specification in ALGOL

; comment NSX! ; $3

which indicates that a semi-colon followed by the ALGOL word "comment" followed by any string X not including; or "end" or "else" is a member of the syntactic category semicolon). This consultation with PREC is carried out at CANGOQ.

CANGOQ Whether or not the syntactic element is equal to the element of the input string which has just been read in, we perform here a check on PREC to see whether the input element can ever lead us to a path which goes eventually to the syntactic component which we must at this point form from the input string. This check is tantamount to asking the question:

Is PREC [input element, syntactic component] = TRUE.

If the answer to this question is yes, we then proceed to call ROUST to form the element from the input string, but if the answer is no, we don't bother to call ROUST since PREC indicates that ROUST could never form the syntactic element from the part of the input string which begins with the input symbol. Instead, we go to SWT1 to check whether the input element was equal to the syntactic element.

The asking of the question PREC [x, y] is detailed under the description of PREC in section 1.2.2.3, where we give the formula for calculating the value of PREC [x,y]. The orders at CANGOQ down to the AJP to SWT1 are merely a set of commands to evaluate this formula, and the AJP jumps to SWT1, or does not, depending on the value of PREC [input element, syntactic category]. If the jump to SWT1 is not executed, we then begin the recursive call of ROUST which will determine whether or not the elements of the input string do in fact form the syntactic element, and if so will diagram the input string elements according to the syntax tables and output the links in OTP. We supply the parameters to ROUST as follows:

1. Supply to IVAR [upper] the address in STAB of the syntactic structure which eminates from the input sumbol. This address, of course, is found by consulting TRAN.
2. Supply the syntactic element which we desire ROUST to form (namely the component of the sentence which we are currently considering) as the GOAL for the next level.
3. Supply to GOAL [upper] the index of the input string which is associated with the input character under consideration.
4. Supply the address of PAR for this level of recursion as the memory cell where ROUST should store the address

in OTP where it puts the output associated with this diagramming of the input string to form the requested GOAL to MAC.

5. Supply the normal and error exits for this call of ROUST to REXIT.

We then increase the index (B5) of BIN by 10 as for every recursive call of ROUST, and transfer to ROUST.

CHASE is the normal exit from this call of ROUST, and at CHASE begins the section of ROUST which continues the sequence across the syntactic sentence. SWT1 is the error exit from this call of ROUST.

SWOF SWOF is a short routine which sets SW to -1 in case that the input element under consideration was not equal to the syntactic category denoted by the sentence component under consideration in the preceeding sections. After setting SW to -1, control goes to CANGOQ (above) whence it goes if the two quantities were equal, (SW having been set to 0 in the latter case)

SWT1 Whenever it is determined that the syntactic component under consideration in this section of ROUST cannot be formed from the input string (either from PREC or from the call of ROUST) we arrive at SWT1. Here we check to see whether the input element itself was equal to the syntactic component. If so, we proceed to CHASE to continue

investigating the sentence one of whose components we have just found in the input string. If not, we then proceed to do the whole process over again for the next syntactic element (if any) at the junction in STAB currently under consideration. Before doing so, however, we carefully reduce (or which is the same, reset) the value of B4 which we had incremented before starting the above process of checking}.

LINK: We arrive at LINK whenever a component check or the discovery of an illigetimate path demands that ROUST progress to the next element at the junction of STAB under consideration at the current level of recursion. We proceed to the next syntactic element at the junction by following the arrow, if any, leading out of the junction; this is, of course, accomplished by proceeding to the link address in the word of STAB currently under consideration. Since B1 contains the address of the element of STAB under consideration, we set the contents of B1 to the link address of the word whose address is in B1. Having done this, we now check to see whether the link address was zero. If not, we have not checked all the elements at the junction to see whether there are any more components, so we proceed back to COMP to test the next element (whose address is now in B1). If the link address <u>is</u> zero, then we have checked all the components at the junction, (and have found that the

input string will satisfy none of them), and we proceed to check the qualification elements and the subjects at the junction.

Anticipating the check for subjects at the junction we first reset SW to indicate that no GOAL has been found, and reset B1 to the address of the first (lowest in memory) element of STAB at the junction (this address having previously been stored in IVAR [upper]). We then test to see whether any satisfied qualifiers were present at the junction. During the check for components (at COMP) we also tested for qualifiers; and upon finding one transferred to LIMITEST. LIMITEST in turn tested to see whether the qualifier was satisfied, and indicated the passing of a satisfied qualifier by storing in MAC[upper] the address of the entry in STAB following the satisfied qualifier (see LIMITEST). If no satisfied qualifiers were passed, MAC[upper] will be zero, since we set it to zero upon entering ROUST. At this point, however, we wish to follow the path indicated by the satisfied qualifier with the smallest range (if in fact there were any) before checking the subjects at the junction. We therefore test MAC[upper] and if it is non zero, we proceed to SNAKE to follow the indicated path. If, however, MAC[upper] is zero, then there were no qualifiers, and we proceed to check any subjects present at the junction.

To do this, we check STAB[B1+1] which will be the meta-symbol $ (internal character 161) if indeed the first syntactic element at the junction is a subject. If the first element is not a subject, we proceed to LNKSUB which will set B1 to the address of the next element at the junction, and repeat the test for a subject.

EQTEST Having arrived at EQTEST, we know that the STAB[B1] is a subject, and we proceed to the processing of that subject. Since it is possible that the GOAL for this level may be reached at this or some higher level, and since if it <u>is</u> reached we must be prepared to specify the definition for the subject under consideration to OTP, we build in OTCEL the Format 1 OTP word which will be placed in OTP after the GOAL has been reached. It is in fact not known at this time whether this definition will be the one specified from this level. The processing of subjects at any given level of recursion is continued only as long as the GOAL has not been reached; hence, if the GOAL is reached at this level or a higher level, we will already have supplied the correct output word. If the GOAL is not reached on a path eminating from this subject, then we will (through LINKSUB) pass to the processing of the next subject at the current junction, and will at that time specify a new definition in OTCEL, namely the one associated with the new subject. The Format 1 OTP word is constructed by

putting the inserting in 15 bit patterns from left to right, the current index of the input string, the index of the input string which was in effect when we first began looking for the current GOAL, and finally the address of the definition which is associated with the subject which we are now considering. (Note in the listing that PAR is the mnemonic used for OTCEL).

The next step in the processing of the subject is to test to see whether the subject under consideration is equal to the GOAL. If the two are equal we set SW to zero to indicate that the GOAL has been reached on this level. Note that this does not imply that we may stop the scaning process. In particular, we may still be able to reach the same GOAL on a higher level and use up more symbols in the input string in so doing. Since we with to use as many symbols of the input string as possible in reaching the GOAL, we merely note that we have reached the GOAL, but continue the scanning process until we have determined that it will not be possible to reach it again by further scanning. To allow for the possibility that we might have reached the GOAL for the last possible time, we enter the address of the next empty space in OTP into the memory cell specified by MAC[lower]. This address was provided by the level of ROUST which first requested the GOAL which we are currently seeking. The next available space in OTP will be the location IN

OTP of the definition which corresponds to the requested GOAL, if indeed this is the GOAL which diagrams the largest number of input symbols (since we will enter no information in OTP on higher levels of recursion unless we again reach the current GOAL.)

COMPATST Whether or not the subject was equal to the GOAl, we arrive at the COMPATST. It is here that we test PREC to determine whether it will be possible to reach the GOAL starting at the tree branch of STAB which begins with the subject we are considering. (Note again, that the test of PREC to determine whether or not to continue diagramming is independent of the test for having reached a GOAL). The test made is asking the question

PREC[subject , GOAL] = "true"

If the answer is yes, we proceed to continue the diagramming process. If the answer is no, we then test SW (AJP 0 SWTST) to see whether the GOAL has been reached at this level. The locating of the pertinent element of PREC is done by evaluating the address and bit formulas given in the description of PREC (section 1.2.2.3).

If the test of PREC indicates that we may continue the diagramming process, we do so by calling ROUST recursively to continue the diagramming at the tree branch beginning with the syntactic element which is the subject we are

considering (LDL 1 0, etc.). We specify as the error exit for this call of ROUST the tag STWST, which is the tag of the routine which checks to see whether we have found the GOAL on this level.

ROUSTX ROUSTX continues the supplying of parameters to ROUST for the recursive call, but this section of the call is used by several calling sequences, since the last parameters are the same for several of the calls (see SNAKE, and CHASE). Since these calls of ROUST are merely to continue the diagramming process (while remembering by virtue of the recursion the paths taken to get this far) but <u>not</u> to specify a new goal as when checking for components of a sentence, we pass along to the next level of recursion <u>the GOAL of this level</u> as well as MAC whose contents specify the address of the memory cell into which we must put the OTP address of the output for the GOAL should we reach it.

The normal exit for these calls of ROUST is GFOTP, which is the routine to which we go when the GOAL has been reached at this or a higher level of recursion. (The fact that ROUST returns by the normal exit indicates that the GOAL has in fact been reached at a higher level, and that this level is only obliged to enter its contents of OTCEL into OTP and return to <u>its</u> normal exit).

SWTST We arrive at SWTST under one of the following two conditions:

1. PREC indicated that the subject under consideration could not lead to the GOAL.

2. PREC indicated that the subject could lead to the GOAL, but the ensuing call of ROUST determined that in fact it dit not.

Hence, we know now that the GOAL could not be reached at a higher level of recursion, if the further recursion was begun at the tree branch of STAB named by the subject being considered. Therefore we test to see whether the GOAL was reached at this level. If so, with no further adieu, we proceed to GFOTP to transfer the contents of OTCEL (PAR) to OTP and return via the normal (we found the GOAL) exit. If on the other hand the GOAL was not reached on this level either, then we proceed to try the next subject at the current junction to see if it will lead to (or become) the GOAL. The progression to the next subject is effected by LNKSUB.

LNKSUB We arrive at LNKSUB when we wish to proceed to the next syntactic element at the junction of STAB under consideration either because

1. The element at the junction is not even a subject or
2. The element is a subject, but has been shown to not lead to the specified GOAL.

We progress to the next element at the junction by setting the contents of B1 equal to the link of STAB[B1], as at LINK. We then test to see whether B1 contains zero. If so, then there are no more elements at the junction to be considered, and we have at this time no alternative ot other than to proceed to the error exit of ROUST, namely ERRX, for we have investigated all the possibilities for satisfying the GOAL at this level and have had no success.

SNBACK At SNBACK we test the current element of the STAB junction we have been considering at this level of ROUST to determine whether this element is a subject of a sentence. This test is made by testing the element following it for the symbol $ (internal symbol 161). They will, of course be equal if the element is a subject. If the element is not a subject, we proceed to LNKSUB to try the next element. If the element is a subject, then we proceed to EQTST to process the new subject.

CHASE CHASE is the sub program which enacts a recursive call of ROUST to proceed across a syntactic sentence in STAB when we have successfully found that the input string will diagram into a component. (This section is a continuation from the component checking section at the beginning or ROUST). For this call of ROUST, we wish to pass along the GOAL and merely specify that the next level of recursion should begin working on the next element (or junction

if there is an arrow eminating from the element) in the syntactic sentence under consideration at the present time. We accomplish these ends by specifying to IVAR[upper] on the next level the address in STAB of the next consecutive element in STAB. The error exit for this call of ROUST is STW1 + 1 which will, in the event that the continuation across the sentence turns out to be an invalid path, continue investigating syntactic elements at the junction of this (i.e., the current) level, after resetting the input string index.

The normal exit for this call is GFOTP, where we put the current contents of OTCEL (PAR) into OTP and go to the normal exit. Note that in this case OTCEL will contain a Format 2 entry for OTP which contains the address in OTP of the definition corresponding to the Syntactic component of a sentence into which we diagrammed part of the input string. Since the last part of the call is identical to the call resulting from a subject investigation, we proceed to ROUSTX +1 to complete the call.

RESTJ Here we reset the index of the input string (B4) and proceed to SNBACK having come from the error exit of the call of ROUST resulting from a satisfied qualifier. (see SNAKE, following, for more detail)

SNAKE: At SNAKE we initiate a recursive call on ROUST having determined (see LINK):

1. that none of the components (if any) at the current junction could be formed from the input string, and

2. that there was a qualifier at the junction which is satisfied.

We arrive at SNAKE with the address in STAB of the element following the satisfied qualifier in the accumulator (having just tested this quantity and discovered it to be non zero). Since the discovery of a satisfied qualifier means simply that we should continue the diagramming process with the elements of the specifications following the qualifier, we merely supply the address found in the accumulator to IVAR [upper] of the next recursion level. We supply RESTJ as an error exit for this call so that in the event that the elements of STAB following the qualifier cannot lead to a correct diagram of the input string, we can reset the index of the input string and proceed to the checking of subjects at the current junction just though there had been no qualifiers. Since the rest of the call of ROUST is the same as the call occuring in the subject investigations, we proceed to ROUSTX for the completion of the call.

ERRX: We arrive at ERRX when we have investigated all the

STAB elements at the current junction and have found that none of them lead to a successful diagramming of the input string. We reset the indices of the input and output strings, and get the error exit (found in REXIT [upper]) into the accumulator so that it may be planted in a jump to effect the error exit.

BOTHXIT: BOTHXIT serves to effect the final exit procedure from a call of ROUST. The address to which we must exit is in the accumulator when we arrive at BOTHXIT; hence we plant this address in a jump command, and after resetting B1 to the value it had on entrance to ROUST, we decrement B5 (to reduce the level of recursion) by 10 and exit to the specified address.

GFOTP: We arrive at GFOTP after having returned successfully from a "continuation" call of ROUST. At this time there will be in PAR (OTCEL) the proper information to be put into OTP. This word is put in OTP, the index of OTP is incremented, a check is made to insure that OTP has not overflowed, and then exit is made to the normal exit of ROUST. The following three cases are distinguished.

1. Arrival at GFOTP from the normal exit of a "subject investigation" call on ROUST. In this case the contents of PAR will be a Format 1 OTP entry giving the address in STAB of the definition corresponding to the

selected subject.

2. Arrival at GFOTP from a "normal continuation" call of ROUST (originating at CHASE). In this case the contents of PAR will be a Format 2 OTP entry. This entry is placed in PAR by virtue of a "component checking" call of ROUST. Note that it is impossible to arrive at CHASE without having found that the input string formed one of the components at the junction of STAB under consideration. If the discovery that the input string formed this component was made by a call of ROUST, the indirect link to the definition associated with the component in question will have been put in PAR by virtue of PAR having been specified in the "component checking" call as the cell into which this quantity should be put. The indirect link would have actually been put there from the EQTEST section at a higher level of recursion. If on the other hand, the discovery that the input string formed the component was made by a simple equality test on the component and an element of the input string, there will be no relevant contents of PAR, but the irrelevant word must none the less be placed in OTP so that position correspondence will be maintained in OTP. (In other words, the element +, for example, will never have a definition associated with it, but it is still counted in referring to the components as though it had one. In fact, in the definition following the + in some sentence no references will be made by a Pn

to the + since it would make no sense to do so, but since the + is counted as a legitimate syntactic element just like the others, one must still treat it in the OTP list as though it had a definition attached to it.)

3. Arrival at GFOTP from a "qualifier" call. In this case, as in the case for + above, no relevant information will be in PAR, but again the irrelevant word must be placed in OTP to maintain the position of the entries following.

GFNDX: Here we merely initiate the normal exit from ROUST by getting into the accumulator the normal exit address and transferring to BOTHXIT where this address will be used to effect the normal exit.

LIMITEST: It is at LIMITEST that the check is made to discover whether or not a qualifier is satisfied. This test falls in two parts:

1. The current index of INPUT, in B4, is compared with the upper and lower bounds specified in the qualifier word. If the index is not within these bounds, the qualifier is considered to be unsatisfied and control passes to LINK where the next element at the junction under consideration is investigated.

2. If, on the other hand, the index is within the bounds, the qualifier is then compared with the last satisfied qualifier (at the current junction) to determine which of the qualifiers has the smallest range.

If the new qualifier has the smallest range, it replaces the old one, in QUAL, and the STAB index of the element following the new qualifier is put into MAC[upper]. If, on the other hand the old qualifier has the smallest range, it is left as the valid one, and control passes back to LINK. Note that if no qualifiers have been passed so far at the current junction, the new qualifier will be selected, since the initial value of QUAL indicates the largest possible range for any qualifier, and thus the new qualifier automatically has a smaller range than the "old" one.

The bounds are stored in the qualifier words with the upper bound in the upper address position and the lower bound in the lower address position. Because of this storage, it is possible to make the test on both bounds with one subtraction. Furthermore, both the test for the index of INPUT and the comparison of the two qualifiers can be made in the same way. The quantity which must be less than the upper bound is put into the upper address position of a test word (LOCN) and the quantity which must be greater than the lower bound is put into the lower address position of the test word. The test word is then subtracted from the qualifier. If the result of the subtraction yields a "0" (corresponding to +) to the left of the upper address position and a "1" (corresponding to minus) to the left of the lower address position, then the test word is within the bounds of

the qualifier. Otherwise it is not within the bounds. The result of the subtraction is compared with the TSTBITS to determine whether the correct bit pattern is obtained, thus determining whether the test word passed or failed the comparison with the qualifier. When the index comparison is made the upper and lower address positions of the test word both are set to the value of the index. When the new and old qualifiers are compared, the new one is subtracted from the old since the criterion for passing this test is that the new one should be within the range of the old one.

Note finally that the new qualifier is already in the accumulator when we arrive at LIMITEST since we tested the sign bit of this qualifier to determine whether or not it was a qualifier.