FOA P rapport
C 8265-11(64)
November 1970

A DATA BASE STRUCTURE FOR A QUESTION-ANSWERING SYSTEM

E Sandewall och K Mäkilä

FÖRSVARETS FORSKNINGSANSTALT
PLANERINGSBYRÅN

Stockholm

FOA RAPPORTKATEGORIER

Rapporter avsedda för spridning utanför FOA utges i följande kategorier:

FOA A-rapport. Innehåller huvudsakligen för totalförsvaret avsedd och tillrättalagd redovisning av ett, som regel avslutat, arbete. Förekommer som öppen (A-) och hemlig (AH-) rapport.

FOA B-rapport. Innehåller för vidare spridning avsedd redovisning av öppet vetenskapligt eller tekniskt-vetenskapligt originalarbete av allmänt intresse. Utges i FOA skriftserie "FOA Reports" eller publiceras i FOA utomstående tidskrift, i vilket senare fall särtryck distribueras av FOA under beteckningen "FOA Reprints".

FOA C-rapport. Innehåller för spridning inom och utom FOA (i vissa fall enbart inom FOA) avsedd redovisning av arbete, t ex i form av delrapport, preliminärrapport eller metodikrapport. Förekommer som öppen (C-) och hemlig (CH-) rapport.

FOA-RAPPORTS STATUS

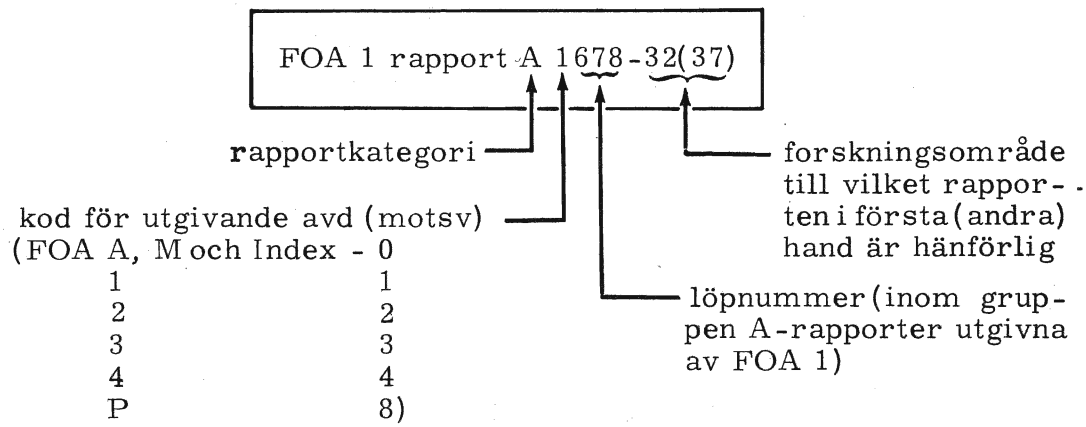
FOA-rapports status är att författaren (författarna) svarar för rapportens innehåll, t ex för att angivna resultat är riktiga, för gjorda slutsatser och rekommendationer etc.

FOA svarar - genom att rapporten godkänts för utgivning som FOA-rapport - för att det redovisade arbetet utförts i överensstämmelse med "vetenskap och praxis" på ifrågavarande område.

I förekommande fall tar FOA ställning till i rapporten gjorda bedömningar etc - detta anges i så fall i särskild ordning, t ex i missiv.

FOA-RAPPORTS REGISTRERING

Fr o m den 1. 7. 1966 registreras FOA-rapport enligt följande exempel:



Försvarets forskningsanstalt
Planeringsbyrån
104 50 Stockholm 80

Research Institute of National Defense
Operations Research Center
S-104 50 Stockholm 80, Sweden

FOA P rapport
C 8265-11(64)
November 1970

A DATA BASE STRUCTURE FOR A QUESTION-ANSWERING SYSTEM

Erik Sandewall, Kalle Mäkilä

Abstract: This report describes a data base structure for expressing binary relations, and a key-punch oriented data language for specifying such a data base.

The first four sections give a general, and essentially machine-independent description of the data base and the language.

The last two sections describe an IBM 360 implementation of this kind of data base and contain:

(1) a detailed specification of the data base organization

(2) a brief description of the routines, that make the translation from the data language to the data base

Search key: Artificial intelligence, Question answering, Data base, Data language, Proberty structure, Computer.

Sammanfattning: Denna rapport beskriver en databasstruktur för att lagra binära relationer, och ett stansorienterat dataspråk för att specificera en sådan databas.

De första fyra sektionerna ger en allmän, och väsentligen maskinoberoende beskrivning av databasen och dataspråket.

De två sista sektionerna beskriver en IBM 360 - implementation av detta slags databas och innehåller:

(1) en detaljerad beskrivning av databasens struktur

(2) en översiktlig beskrivning, av de rutiner, som översätter från dataspråket till databasen.

FOA kostnadsnummer 820M111 (Frågebesvarande system)

CONTENTS

0. Introduction.

This report is intended to specify the following:

(a) The SPB data structure (mainly as in "A set-oriented property structure representation for binary relations, SPB", Sandewall sept. -69) (section 1)

(b) PROPLAN, a formal language for expressing facts and questions in SPB in a convenient form. (sections 2-4)

(c) An IBM 360 implementation of the SPB data structure, a bit-level specification of the various fields used in the internal data base. (section 5)

(d) Routines that translate from PROPLAN into the data structure described in section 5. The users communication with these routines. (section 6)

1. The property structure.

The property structure is designed to represent sets of objects and binary relations connecting these sets. Thus we have two kinds of entities; nodes that represent sets of objects and arcs that represent the binary relations. To each node is connected a property list which contains information on all the arcs connected to this node.

Nodes.

In the final form of the property structure there are two types of nodes, constants and variables. The difference is that for variables some of the arcs on the property list are marked as defining properties. If there are other nodes in the base which satisfy the defining properties of a variable, then it can be concluded that a subset relation holds between the corresponding sets.

Arcs.

An arc connecting two nodes e.g. a and b is described by two properties. The property list of the node a contains a property that describes the connection with b, and similarly the property list of b contains a property describing the connection with a. Each property contains the following five items of information:

$\langle$ s,d,q,r b $\rangle$

(b) a pointer (or other means of reference) to the node at the other end of the arc

(d) the direction of the arc ("coming" or "going")

(r) a binary relation

(s) the presence or absence of a negation sign

(q) information on how the arc is "connected" at each end for example existential or universal quantifier.

2. Introduction of PROPLAN, simple statements.

In typical applications, a property structure receives a continuous input stream of new relations (which are stored in the structure) and of questions (which are answered from the available data). The data base grows in this process; new constant symbols (names for sets of objects in the universe) may be introduced; but the set of binary relations is assumed to be given once and for all. The present section will describe the SPB data language PROPLAN, which is a keypunch-oriented notation for the input to the data base. It is intended that this language shall be used

(1) by humans who want to communicate with an SPB data base
(2) as the output of programs which translate from natural language into a formal notation.

A text in the data language consists of an ordered sequence of statements. There are statements for introducing a new node in the data base, for adding more arcs, and for describing a node which has already been put into the data base.

There are also a few possibilities of forming more complex statements and questions using simple statements. These are described in the next two sections.

In this section we shall describe the various types of simple statements in detail.

2.1 Declarations.

A declaration may have any of the forms

CONSTANT x_1 , x_2 , --- , x_n ;

VARIABLE x_1 , x_2 , --- , x_n ;

DUMMY z x_1 , x_2 , ---- , x_n ;

ENDOFDEF x_1 , x_2 , ---- , x_n ;

which each x_i is a new alphameric identifier, and z is a code (e.g. THE,THIS) which indicates a choice of subprogram call.

The effect of a CONSTANT or VARIABLE statement is to introduce for each x_i, a node, which can thereafter be referenced by the identifier x_i.

The effect of a DUMMY declaration is to introduce each x_i as a name for a node which (normally) is already in core, and which is to be retrieved using information in succeeding statements, and using a sub-program referenced by the indicator z.

In the ENDOFDEF statement each x_i has been introduced in an earlier VARIABLE or DUMMY declaration, and has not yet occured in an ENDOFDEF statement. The purpose of the statement is to tell the assimilator (i.e. the program that stores data in the data base) that no more assertions or descriptions involving (explicitly or implicitly) a phrase ... DEF x_i ... are to be expected. Such a terminal signal is necessary before the variable or dummy can be used by deduction routines.

In the case of DUMMY variables ENDOFDEF also initiates the process of retrieving a matching node and transferring the properties (apart from the definition) to this node.

2.2 Assertions.

An assertion has the form

$$\cdot(q_a \;\; a \;,\; d \; s \; r \;\;,\; q_b \;\; b)\;;$$

where

a and b — are alphameric identifiers which have been introduced in previous declarations or descriptions;

q_a and q_b — are "connection codes", which may be either of ALL, SOME, ITS, DEF, THAT or blank. Blank is synonymous to ALL.

d — is either blank, or the code REVERSE;

s — is either blank, or the code NOT;

r — is a name for a binary relation. The names for the various relations should be specified in an input table before the process of assimilation is started (see 6.2)

The meaing of an assertion

$$(\; q_a \;\; a,\; d \; s \; r \;,\; q_b \;\; b \;)$$

is (in the notation of memo 6)

$$a \;\; \left[q(d's'(r))\right] \;\; b$$

where

$$d' = \begin{cases} Id & \text{if d is blank} \\ Rev & \text{if d = REVERSE} \end{cases}$$

$$s' = \begin{cases} Id & \text{if s is blank} \\ Neg & \text{if s = NOT} \end{cases}$$

and q is determined from the following table:

q_a	q_b	q
ALL	ALL	Aa
ALL	SOME	Ae
ALL	DEF	Ad
ALL	ITS	At
SOME	ALL	Ea
SOME	SOME	Ee
THAT	DEF	Bd
DEF	ALL	Da
DEF	THAT	Db
DEF	DEF	Dd
DEF	ITS	Dt
ITS	ALL	Ta
ITS	DEF	Td
THAT	THAT	Bb

2.3 Descriptions.

A description is a more compact way of writing a declaration plus a sequence of assertions. The form of a description is either of

CONSTANT x f_1 f_2 --- f_m ;

VARIABLE x f_1 f_2 --- f_m ;

DUMMY z x f_1 f_2 --- f_m ;

where z is a code like above, and each f_i is an assertion fragment. Each assertion fragment has the form

(q_a , d s r , q_b b)

where the components are like above. If q_a is blank, the first comma should be omitted.

A description is equivalent to a declaration plus a sequence of assertions, and can be converted to that form by the following algorithm:

(1) Insert x after the q_a in each f_i.

(2) Insert a semicolon before each f_i.

2.4 Abbreviations.

The expression

SINGLEVARIABLE x f_1 F_2 --- f_m ;

is an abbreviation for

VARIABLE x f_1 f_2 --- f_m ; ENDOFDEF x ;

A similar convention is made for DUMMY statements.

The symbol DISJOINT is introduced as an abbreviation for "NOT EQUAL" (where EQUAL stands for an equality relation between objects). Similary we introduce abbreviations OVERLAP, SUBSET, SUPERSET and OCCUR. The complete definitions are as follows:

abbreviated assertion	full assertion
(q_a a , DISJOINT , q_b b)	(q_a a , NOT EQUAL , q_b b)
(a , OVERLAP , b)	(SOME a , EQUAL , SOME b)
(q_a a , SUBSET , b)	(q_a a , EQUAL , ITS b)
(a , SUPERSET , q_b b)	(ITS a , EQUAL , q_b b)
(a , NOT SUBSET , b)	(Some a , NOT EQUAL , ALL b)
(a , NOT SUPERSET , b)	(ALL a , NOT EQUAL , SOME b)
(a , OCCUR)	(SOME a , EQUAL , SOME a)

Similar conventions are used for assertion fragments.

2.5 A formal grammar for simple statements.

```
STATEMENT    ::=  { DECLARATION
                  | FRELATION
                  | SENTENCE
                  | ORDER      } ;

DECLARATION  ::=  TYPE NAME | DECLARATION , NAME

FRELATION    ::=  ( [QUANTIFIER] NAME , [REVERSE] [NOT]
                  RNAME , [QUANTIFIER] NAME ) |
                  ( NAME , OCCUR )

SENTENCE     ::=  TYPE NAME SRELATION | SENTENCE
                  SRELATION

SRELATION    ::=  ( [QUANTIFIER ,] [REVERSE] [NOT] RNAME ,
                    [QUANTIFIER] NAME ) | (OCCUR )

QUANTIFIER   ::=  { ALL
                  | SOME
                  | ITS
                  | DEF
                  | THAT     }

TYPE         ::=  { CONSTANT
                  | VARIABLE
                  | DUMMY
                  | SINGLEVARIABLE
                  | SINGLEDUMMY
                  | ENDOFDEF  }

ORDER        ::=  $NAME
```

3. Questions.

3.1 Simple questions.

These are expressed exactly as assertions but are preceded by the rese ved word "QUESTION".

Example: QUESTION (A, R1, B);

3.2 Compound questions.

Several assertions can be joined by the connectives "OR" and "AND" to form compound statements. Both these connectives should not be used in the same compound question.

Example: QUESTION (R1, RR, R2); OR (A, R3, B); OR (C, R5, D);

3.3 Open questions.

The basic form of open questions is the reserved word "WHICH", followed by a description.

Example: WHICH CONSTANT X (R1, A) (R2, B);

This causes a question answering routine to start retrieving all nodes X satisfying the description.

Other forms of open questions not yet implemented in the system are similar and correspond to questions containing such words as "WHEN", "WHERE" "WHY" etc.

These questions initiate, first a retrieval of a hopefully unique node X and then a looking for other nodes connected to X with certain relations such as spatial, time or cause relations, which are somehow specified in the question.

4. Compound statements.

There are a few possibilities to join simple statements and form compound statements. This is only possible on top level i.e. you can not join compound statements in the same way.

4.1 Temporary data.

Sometimes in connection with questions, it is desirable to have temporary data that are present in the data base only while the question is answered and then removed.

This is achieved by enclosing the statements between the reserved words "TEMP" and "ENDTEMP". "TEMP" causes the assimilator to save pointers to the present top of the data base, and "ENDTEMP" causes it to reset these pointers and zeroize all references made to the temporary data from the permanent data.

A typical question like

" Is copper a heavy metal ? "

could be translated into the following PROPLAN statements:

```
TEMP
     SINGLEVARIABLE HEAVYMETAL (DEF, PRED, HEAVY)
                               (DEF, SUBSET, METAL*S);
     QUESTION  (COPPER, SUBSET, HEAVYMETAL);
ENDTEMP
```

Thus we introduce teporarily the defined set of heavy metals, and ask whether copper belongs to this set.

4.2. IF statements.

These are useful when the device (or person) creating the input data wants to give alternatives depending on what is already stored in the data base. They have two forms:

(1) IF string1 THEN string2 CLOSE

(2) IF string1 THEN string2 ELSE string3 CLOSE

string1 should be a question, simple or compound, optionally preceded by temporary data.

string2 and string3 should be strings of simple statements to be assimilated.

The statement is treated in the following way:

The question is answered (in a 2-valued logic, that is all answers except "yes" are regarded as "no".)

If the answer is "yes", then string2 is assimilated, otherwise string3 (if there is such a string).

5. Description of data fields used in an IBM 360 implementation.

The specifications in this section are applicable to all byte-oriented computers. The data base consists of two main parts:

(1) The external names area.

This consists of variable length fields containing essentially the external name of a node, and a pointer to this node in the network.

(2) The network.

This is organized in fields of eight bytes each. There are three types of fields:

(1) origin fields
(2) property-string fields
(3) property-list fields

The first byte in a field contains (among others) information to indicate what kind of field it is.

Each field consists of a number of sub-fields, each of which consists of one or more bits. These sub-fields will here be denoted by capital letters.

The address of a field is defined as the address of the first byte in the field.

5.1 Storage of external names.

The first time a node is presented to the system, it creates an origin cell in the network, and a reference field connecting this cell with the external name. All the reference fields are linked together to form an alphabetic binary tree.

Optionally, the name of the node could be followed by "spelling information" separated by an underline character. This consists of a string of letters, which is stored only to support the output translation procedures. In later references to the node only the name should be used.

One reference field contains the following information:

n_1 number of bytes in the name

n_2 number of bytes in the spelling information

p_0 pointer to the origin field in the network

p1 upward pointer in the binary tree

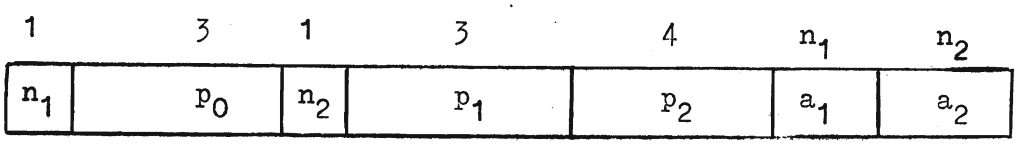
p_2 downward pointer in the binary tree

a_1 external name of the node

a_2 spelling information

p_0 is a relative address in the network area, p_1 and p_2 are relative addresses in the area of reference fields.

The structure of a reference field is:



The text at the right end of the field is filled out with blanks to make the size of the field a multiple of four bytes.

Example:

The name NAME1_SPELLINFO
would result in

$n_1 = 5$
$n_2 = 9$ and the text

NAME1SPELLINFObb
a_1 a_2

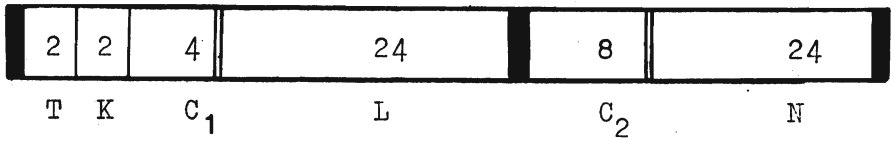
A name may contain letters, digits and the characters

* . + - .

5.2 Origin fields.

The purpose of an origin field is to represent a node in the network, and to be a key to all information about the node. The address of the origin field is used inside the computer as an internal name for the node. Thus (apart from the complication with DUMMY symbols) there should be an 1-1 correspondence between declared identifiers and origin fields.

The structure of an origin field is:

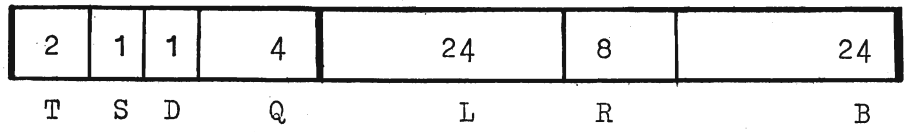


where the sub-fields are used as follows:

subfield name	number of bits	purpose	codes
T	2	indicate that this is an origin field	00
K	2	indicate what kind of node	constant = 00 variable = 01 dummy = 10
C_1	4	open	
L	24	give the first item on the property-list (an address)	
C_2	8	in case this is a dummy node: an internal code for z, otherwise open	
N	24	pointer to the external name field of the node as described in 4.1	

5.3 Property-list fields.

A property-list field has the following layout:



If an origin cell a is to be assigned a property Gb, represented

s,d,q,r b (in the notation used in section 1.2)

then the subfields with these names are used.

The subfield T contains the code 01 (to indicate a property list field) and the subfield L contains a pointer to the next property of a.

A detailed description of the use of the field Q is given in section 5.6.

5.4 Property-string fields.

A property-string field for the relation aGb, has exactly the same layout as the corresponding property-list field, but the contents of the T and L fields differ.

The subfield T contains the code 10, to indicate a property-string field.

The subfield L now contains a pointer to the next property of the node b.

Every property-string field in fact also belongs to the property-list of the node pointed to by its B-field.

5.5 Relationship between property-list and property-string fields.

Each origin field is connected with properties in two ways:

(1) starting in the L sub-field of the origin field, we have a chained list of property-list fields. Each field on the property-list expresses one property of the node.

(2) in the positions immediately after the origin field, we have a sequence of property-string fields. The string is terminated by the occurence of a property-list field or an origin field.

It is convenient to describe the contents of these fields in terms of the notation used in memo 6. We assume, therefor, that input relations are converted to that form by the conventions in section 2.2 and 2.3 of this memo.

An arc aGb between two nodes in the property structure can be represented in either of two ways:

(1) using two property-list fields: one field on a's property-list, which assigns the property Gb to a, and one field on b's property-list, which assigns the property [Rev(G)] a to b.

(2) using one property-string field, which is on the property-string of a and on the property-list of b, and which points to b. This is the property-string field for aGb. (Alternatively, a property-string field for b [Rev(G)] a may be selected.

Thus the property-string representation only requires half as much space as the property-list representation. However,

property-strings can only be used if space is available immediately above the origin field for the node. This condition is satisfied if memory is never released in the data base (no garbage collection), and properties are assigned to a node immediately after it has been created. This is the case at least when we use description statements in the data language. Thus we make the following convention:

> Assertions in the data language are to be translated into pairs of property-list fields; assertion fragments into property-string fields.

As a program runs down the property-list of a node a, it will encounter both property-list and property-string fields. In the latter case, it may be preferable to copy the contents of the field to an auxiliary space, and to reduce it there to what it would have looked like, if it had been a property-list field.

The following operations are needed:

(1) change the contents of the T sub-field from 10 to 01;

(2) go backwards (by decreasing addresses in steps of 8) from the property-string field, until you find an origin field. Store its address in the sub-field B;

(3) complement D;

(4) apply the Rev operation to the Q field. This is done by bit manipulations as specified in the next section.

5.6 Internal representation of quantifiers.

The left and right quantifiers are transformed into one single fourbit quantifier Q , as shown in the table below.

Q consists of two different parts, the first three bits denoted by Q_{123} , and the rightmost bit Q_d . Q_d is nonzero only for pairs containing at least one DEF quantifier in some place, and thus serves as a marking of the definition properties of variables.

Q_{123} are so chosen as to facilitate the operations of reversion and negation. Reversion is done by interchanging the two leftmost bits of Q (except the pair Ad , Da where the third bit is complemented).

For the quantifiers not containing DEF or THAT there exists a negation -Q which is obtained simply by complementing Q_{123}.

The following notation is used in the heading of the table:

Q_1 left quantifier

Q_r right quantifier

q mnemonic code for total quantifier

Q_{123} first three bits of Q

Q_d rightmost bit of Q (definition marker)

Q total quantifier

$\overline{Q}$ reverse quantifier

-Q negation of quantifier

Q_1	Q_r	q	Q_{123}	Q_d	Q	$\overline{Q}$	-Q
ALL	ALL	Aa	000		0000	0000	1110
ALL	SOME	Ae	010		0100	1000	1010
ALL	DEF	Ad	000	1	0001	0011	
ALL	ITS	At	011		0110	1010	1000
SOME	ALL	Ea	100		1000	0100	0110
SOME	SOME	Ee	111		1110	1110	0000
THAT	DEF	Bd	100	1	1001	0101	
DEF	ALL	Da	001	1	0011	0001	
DEF	THAT	Db	010	1	0101	1001	
DEF	DEF	Dd	110	1	1101	1101	
DEF	ITS	Dt	011	1	0111	1011	
ITS	ALL	Ta	101		1010	0110	0100
ITS	DEF	Td	101	1	1011	0111	
THAT	THAT	Bb	001		0010	0010	

6.1 A short description of the method used.

The process of assimilation is performed in two separate. routines:

SEMANTICS which takes as input one single symbol. For each symbol it performs an appropriate sequence of actions (which is "the semantic meaning" of the symbol). The totality of these actions eventually creates the data base.

ASSIM which makes the parsing of the PROPLAN text. This parsing contains the following steps:

Step 1 Preprocessing of simple statements and handling of compound statements. Symbols are taken from left to right, and if part of a simple statement put into a statement buffer. First they are compared to the table of terminal symbols and found to be either:

(a) A terminal symbol used in constructing compound statements. This only causes a call of SEMANTICS.

(b) Another terminal symbol. Its index in the terminal symbol table is put into the statement buffer. If it is a ";" then the statement is complete and the parsing (step 2) is started.

(c) A string not found among the terminal symbols. Then it must be an external name, either of a node or a relation. It is determined from the context in the statement, whether it is a relation name (RNAME) or a node in an assertion (NAME) or a node in a declaration (DNAME), and the corresponding index (=internal name) is put into the statement buffer.

Step 2 The statement buffer now contains exactly one simple statement in the form of a string of indicesin the table of terminal symbols. This is now parsed using mainly the Wirth-Weber method.

The precedence functions are obtained by using a program "SYNPROC" written by Nicklaus Wirth. This program takes as input the productions of the grammar, and as output it produces the same productions and the precedence functions in the form of data declaration statements in PL360. These can be put directly into the syntax analysis program.

For every symbol, terminal or non-terminal, that appears during the parse, the routine SEMANTICS is called and thus the data base is created.

6.2 Specification of terminal symbols and relation symbols.

Before presenting cards with PROPLAN text to the assimilator, tables containing the terminal symbols and relation symbols must be built up. These tables are specified on cards in the following way:

First we have a steering card with the directive "%ASSPAR", starting in column 1. This indicates that what follows is ASSimilator PARameters. Each set of parameters is headed by a steering card with an * in col. 1. The data cards contain strings of characters separated by spaces. Each string is taken as a symbol until a card with " * " or " % " in column 1 is found.

*SYMBOLS

These are the basic symbols (or reserved words) used in the PROPLAN language as described in sections 2-4. They must always be presented in the same order but symbols can be replaced by other symbols having the same meaning.

*RELATIONS

The set $\{R\}$ of relations symbols. The first five of them must always denote the same relations as

DISJOINT OVERLAP SUBSET SUPERSET EQUAL

in the sense of section 2.4. The rest of them can be chosen freely.

*TRANSITIVE

The set of relations $\{R_T\} \subset \{R\}$ which are transitive.

*RSYMMETRIC

The set of relations which are symmetric. $\{R_S\} \subset \{R\}$

*REVERSIONS

Pairs of symbols (a_i , b_i) $(i = 1,2, \ldots.)$
where $a_i \in \{R\}$, $b_i \notin \{R\}$ and $b_i \equiv$ REVERSE a_i.

Thus we can here introduce new relations that are interpreted by the assimilator as the reversion of some relations introduced under *RELATIONS.

6.3 Control parameters and switches.

The program contains parameters and switches, that have to be set at the beginning of a run, and which can in in some cases be changed during the run. Most of them control the deduction procedures and are to be described in a later report.

There are a few more steering cards similar to those in the previous section.

*PARAMETERS

After this card follows a number of cards with integers, which are converted to binary half-words and taken as initial values of certain steering parameters.

***NEWPARAMETERS-nn**

Sets new values to the parameters beginning with the nn: th.

*SWITCHES-vvvvvvvvvvvvvvvv

After the minus-sign (col 11) follows a string of 16 zeroes and ones. A corresponding set of logical bytes in the memory are set true or false.

A single switch among the first nine can also be set on or off directly in the PROPLAN text by writing:

$SWITCHn; or $NOSWITCHn; where n is the number of the switch. This type of statement ($ keyword ;) are called orders and are used to perform simple actions in the SEMANTICS routine. Sometimes an order is followed by a string of text, that is to be interpreted in some special way.

A few examples of orders:

$CRITIQUE;

The subsequent assertions are put as questions to the deduction procedures, before they are added to the data base. If contradictory, they are ignored and an error message is given.

$UNCRITIQUE;

Assimilation is performed without checking for contradiction.

$NEGQUESTION; ($NONEGQUESTION;)

In questionanswering the negation of the question is also answered (not answered).

$PROPERTIES;

followed by a number of node names terminated by the string " FINE " , gives a listing of the property structures of these nodes.

$QPRINT;

after a question gives a listing of all the subproblems that were produced in answering this question.

There are several other similar report generating and dump facilities in the form of orders. These have been of great value in debugging and developing the programs.

6.4 Error messages.

There are a few error messages indicating incorrect or incomplete PROPLAN input. Also the deduction procedures (which are normally called during the assimilation) can produce error messages, but these are to be described in a later paper. Mostly they indicate overflow in some table or stack.

The messages are all of the form:

********SQAP ERROR MESSAGE: S
where S is a 32-character string.

During the process of assimilation S could be:

(a) PROPLAN SYNTAX ERROR

The next line contains 64 characters of text beginning with the statement containing the error.

(b) UNDEFINED NODE A

Where A is a name of a node. This name has been used in an assertion or description before it has been declared.

(c) UNDEFINED RELATION R

Where R is a relation name not contained in the table of relation names.

(d) DUMMY FAIL A

Where A is a name of a DUMMY variable. The program

has failed to eliminate this node from the data base.

(e) CONTRADICTION

An assertion is written on the line following this message. This assertion has been proved to contradict the data base. (That is, the negation of the assertion has been proved from the previous data base.) The assertion is ignored.