The University of Wisconsin
Computer Sciences Department
1210 West Dayton Street
Madison, Wisconsin 53706
Received: April 26, 1976
TR 272

META-SYMBOLIC SIMULATION SYSTEM (MESSY)
USER MANUAL

by
Matthew A. Appelbaum
with
FORWARD: The History of MESSY
by
Sheldon Klein

META-SYMBOLIC SIMULATION SYSTEM (MESSY)
USER MANUAL

by
Matthew A. Appelbaum
with
FORWARD: The History of MESSY
by
Sheldon Klein

Computer Sciences Department
Linguistics Department
University of Wisconsin
1210 W. Dayton St.
Madison, Wisconsin 53706

Abstract

USER MANUAL for the Meta-symbolic Simulation System that includes a behavioral simulation programming language that models, generates and manipulates events in the notation of a semantic network that changes through time, and a generalized, semantics-to-surface structure generation mechanism that can describe changes in the semantic universe in the syntax of any natural language for which grammar is supplied. The system can handle generative semantic grammars in a variety of theoretical frameworks, and is especially suited for modelling text grammars, including 'frames', 'scripts', and 'scenarios'.

While Sheldon Klein is responsible for the basic design of the system, it is the result of the efforts of more than twenty students, over a five year period.

FORWARD: The History of MESSY

by

Sheldon Klein

While I am responsible for the basic design of the Meta-symbolic Simulation System, it is the result of the efforts of more than twenty students over a five year period. A product of the classroom, it was produced in response to my specifications. What I got was sometimes less and usually more than I asked.

Our original motivation was to develop a simulation system to work in conjunction with our language learning programs to provide a basis for modelling language change in speech communities as a function of social structure and historical events (Klein 1966, 1967, 1974; Klein, Kuppin, Meives 1969, Klein & Rozencvejg 1975). Toward this goal we developed a meta-symbolic simulation system that includes a behavioral simulation programming language that models, generates and manipulates events in the notation of a semantic network that changes through time, and a generalized, semantics-to-surface structure generation mechanism that can describe changes in the semantic universe in the syntax of any natural language for which a grammar is supplied. The system can handle generative semantic grammars in a variety of theoretical frameworks.

One of the most significant features of the system is that the semantic deep structure of the non-verbal, behavioral rules may be represented in the same network notation as the semantics for

natural language grammars, and thereby provide non-verbal context for linguistic rules. This feature also permits experimentation with a natural language meta-compiling capability. The semantic network may be used to generate texts in the simulation language itself, and these may be compiled as new behavioral simulation rules during the course of a simulation. These rules may be rules that, themselves, describe the process of deriving new rules. Because of the common notation, non-verbal, behavioral rules can be derived from natural language conversational inputs using inference techniques analagous with those for inferring natural language, generative semantic grammars (Klein et al 1968; Klein & Kuppin 1970; Klein 1973). The total system has the power of at least the second order predicate calculus, and will make it possible to formulate text grammars with arbitrarily abstract semantic components, together with rules for their logical quantification. Our use of the term 'text grammar' extends to any kind of general, formal statement of rules for generating structured behavior, and includes the concepts of 'frames', 'scripts', and 'scenarios' (Klein 1975).

Our first major experiment with the system was a text grammar model that generated 2100 word murder mystery stories in less than 19 seconds each, complete with calculation of the plot as well as the surface syntax (Klein et al 1973). The speed of this generation is 100 to 1000 times faster than other existing programs using transformational grammars just for sentences in isolation.

The goals we set for the system are heroic, and demand heroic test cases. The approach we adopted was to model major, significant portions of fields unrelated to our own, and to allow the impact of our models for the independent field validate our own claims. The idea of using folklore models was suggested to me at an interdisciplinary conference on text grammars (held in Bielefeld, Germany in 1974, sponsored by Janós Petőfi and Teun A van Dijk) by Dimitri Segal, Elli Köngäs Maranda, Heda Jason & Michel de Virville. We were also stimulated by Ed Kahn, who reviewed the murder mystery model in his Ph.D. dissertation (Kahn 1973). Kahn's remarks there, and in a published debate (Kahn & Klein 1974) suggested that our system was incapable of handling the logical quantification complexities of Fairytale and Myth models of Propp and Lévi-Strauss. As a consequence, we automated Propp's first order predicate calculus text grammar for Russian fairytales, originally published in 1928 (Propp 1968, English translation). Our model generated several hundred Russian fairytales at an average speed of 128 words a second, including plot computations and specification of deep structure as well as surface syntax (Klein et al 1974, Klein et al 1976).

We soon perceived that the Myth models of Levi-Strauss presented a challenge for control of semantics and symbolism undreamt of in ordinary linguistic studies. We produced a simple model for five myths from The Raw and the Cooked (Lévi-Strauss 1969) (Klein et al 1974), and a more elaborate one (Klein et al 1976). Our

current efforts are directed toward automatic inference of text-grammars using meta-compiling techniques, and our truly heroic test case for this is the automation of the analytic techniques used by Lévi-Strauss, himself. Toward this goal, a portion of the MESSY research group will spend academic year 1976-77 in Paris, working with Lévi-Strauss. Our mailing address for the period September 1976-June 1977 is:

Professor Sheldon Klein
Laboratoire d'Anthropology Sociale
L'ECOLE DES HAUTES ÉTUDES EN SCIENCES SOCIALES
11 Place Marcelin-Berthelot
75231 PARIS CEDEX 05
FRANCE

At this point I would like to list the contributors to the Meta-symbolic Simulation system:

John D. Oakley was the first major contributor who, together with David J. Suurballe and Robert A. Ziesemer, produced the first version of the system, operational on a Burroughs 5500 computer, and written in Burroughs extended ALGOL (Klein et al 1971).

Steven L. Converse moved the system from the B5500 computer to the Univac 1108, converting from ALGOL to FORTRAN V. It is fair to say that, without his efforts, the project would have died at the time the University of Wisconsin lost the B5500 computer. His contributions also include the look-ahead construction and the secondary triple device.

David F. Balsiger is the main figure connected with the natural language generation component. Mark Foster, with some assistance from Robin Lao, made contributions to the grammar and dictionary structure.

John F. Aeschlimann, has, at this point, made the largest contribution to the structure of the Meta-symbolic Simulation System. A few of his contributions include predicate operations, dynamic modification of lexical lists, subscripted classes and reality value codings.

Matthew A. Appelbaum is, of couse (with the assistance of John F. Aeschlimann) the author of the USER Manual. His other contributions include subroutines (in combination with Scott J. Kamin and S. David Kalish) as well as contributions to the general revision and clean-up of the latest version of the system. Appelbaum's primary contribution remains his coding of the Propp model for Russian fairytales, our major test case (assisted by S. David Kalish).

With regard to test cases, Lynne A. Price was the major creator of the design of our Lévi-Strauss model and John F. Aeschlimann worte the code. Some assistance was provided by Elizabeth J. Curtis.

Claudine Court and Joel Smith produced the Murder Mystery program.

For all the test cases, David F. Balsiger and Mark Foster were primarily responsible for the associated semantics-to-surface-structure grammars.

Other contributors include Ying-Da Lee, who added classes of relations, Abbi-Jane Lau, who added the capability of using simulation language classes as semantic distinctive features with dyanmic modification capability, Donald Gordon Wilson for expansion of operations on subscripted classes, Babak Chubak and Carlos A. Cortez for unifying our treatment of relations, James Weiner, Anita Siu-Man Lo, Priscilla M. Lu, Sally Siu-Lie Chan and Mark E. Sanders for miscellaneous contributions, and James E. Moore and Barry T. Rowe for a component for facilitating the input and update of grammars and dictionaries.

The system is currently implemented in FORTRAN V for the Univac 1110 with an interpreter. The code, including that for the generator, occupies 30K words of core storage, and the data structures may occupy from 10K to 50K words.

References

Klein, S. 1966 Historical Change in Language Using Monte Carlo Techniques. Mechanical Translation, Vol. 9, Nos. 3 & 4, September and December, 1966.

Klein, S. 1967 Current Research in the Computer Simulation of Historical Change in Language. Actes du X^{e} Congrès International des Linguistes Bucharest 1967, Vol. IV, (published 1970) Academy of the Socialist Republic of Roumania, Bucharest.

Klein, S. 1973 Automatic Inference of Semantic Deep Structure Rules in Generative Semantic Grammars. UWCS Tech Rept. 180. Also in Computational and Mathematical Linguistics: Proceedings of the 1973 International Conference on Computational Linguistics, Pisa, A. Zampolli, Editor, Olschki, Florence, 1976.

Klein, S. 1974 Computer Simulation of Language Contact Models. Towards Tomorrow's Linguistics Shuy & Bailey, Editors, Georgetown University Press, Washington, D.C.

Klein, S. 1975 Meta-compiling Text Grammars as a Model for Human Behavior. In Theoretical Issues in Natural Language Processing: An Interdisciplinary Workshop In Computational Linguistics, Psychology, Linguistics, Artificial Intelligence, MIT, Cambridge, Mass. June 1975. Schank & Nash-Webber, editors, Association for Computational Linguistics, Arlington.

Klein, S., J. F. Aeschlimann, D. F. Balsiger, S. L. Converse, C. Court, M. Foster, R. Lao, J. D. Oakley & J. Smith, 1973. AUTOMATIC NOVEL WRITING: A Status Report, UWCS Tech Rept. 186. Presented at Internal Conference on Computers in the Humanities, Minneapolis, 1973. Also in press (abridged) in The Role of Non-Automised and Automised Grammars in Text Processing Systems, Burghardt & Hölker, de Gruyter, Berlin & New York.

Klein, S., J. F. Aeschlimann, M. A. Appelbaum, D. F. Balsiger, E. J. Curtis, M. Foster, S. D. Kalish, S. J. Kamin, Y-D Lee, L. A. Price, & D. F. Salsieder. 1974 Modelling Propp and Lévi-Strauss in a Meta-symbolic Simulation System. UWCS Tech Rept. 226. Also in press in Patterns in Oral Literature, Jason & Segal, editors, as a retroactive contribution to this volume of the 1973 World Conference of Anthropological and Ethnological Sciences, Chicago.

Klein, S., J. F. Aeschlimann, M. A. Appelbaum, D. F. Balsiger, E. J. Curtis, M. Foster, S. D. Kalish, S. J. Kamin, Y-D Lee, & L. A. Price 1976. Simulation d'Hypotheses Emises par Propp & Levi-Strauss en Utilisant un Systeme de Simulation Meta-symbolique. Informatique et Sciences Humaines Spring 1976. (A revised and expanded version of Klein et al (1974), in French translation, and with new, expanded Lévi-Strauss model and revised Propp model.)

Klein, S., W. Fabens, R. Herriot, W. Kathke, M. A. Kuppin, & A. Towster 1968. The AUTOLING System. UWCS Tech. Rept. 43.

Klein, S. & M. A. Kuppin 1970. An Interactive, Heuristic Program for Learning Transformational Grammars. Computer Studies in the Humanities and Verbal Behavior Vol. 3, No. 3.

Klein, S., M. A. Kuppin & K. Meives 1969 Monte Carlo Simulation of Language Change in Tikopia and Maori. Preprints of Papers Presented at 1969 International Conference on Computational Linguistics, Stockholm, KVAL, Stockholm.

Klein, S., J. D. Oakley, D. A. Suurballe & R. A. Ziesemer 1971. A Program for Generating Reports on the Status & History of Stochastically Modifiable Semantic Models of Arbitrary Universes. UWCS Tech Rept. 142. Also in Statistical Methods in Linguistics 8, 1972.

Klein, S. & V. Rozencvejg 1974. A Computer Model for the Ontogeny of Pidgin and Creole Languages. UWCS Tech Rept. 238. Presented at the 1975 International Conference on Pidgins and Creoles, Hawaii, January 1975.

Lévi-Strauss, C. 1969, The Raw and the Cooked. (English translation) Harper & Row, New York.

Propp, V. 1968. Morphology of the Folktale (English translation) 2nd Edition, University of Texas Press, Austin.

1. Introduction

This manual will contain a syntactic and semantic description of the simulation language component of the meta-symbolic simulation system. When necessary, information about the internal system mechanism and data structures will be given, but this will generally be avoided. Also absent from this manual, except in a few special instances, will be any description of the linguistic generative component of the simulation system; this component is used to take the output from the simulation system and, using a grammar and dictionary, produce sentences in a chosen natural language.

As this is a simulation system, the modelled universe changes as a function of time. A report on the current state of the universe is printed out at each time change, this report being called the change stack.

Internally, the universe is modelled as a semantic network (also referred to as the network) which is composed of semantic triples (also called triples) which represent the semantic deep structure. The modelled universe also contains some information, such as regarding class membership, which is not in the semantic network.

The major component of the language is the group, which comprises any number of rules. A group is executed at intervals according to an internal clock, and may also be temporarily or permanently turned off (disabled) or activated (enabled). Groups may also be called as subroutines. The rules in a group are evaluated sequentially, but the sequence may be dynamically changed.

A rule controls changes in the universe. It is made up of two parts: an action list, which manipulates the universe--the network, class membership lists, and sequencing information (and which may include a look-ahead into the future); and a series of conditions (subrules) which test the universe and set probability parameters according to the outcome of the tests, causing execution or skipping of the associated action list depending on the cumulative probability as compared to a system generated random number.

The network is changed and tested with semantic triples, which are assertions about the universe. A triple is composed of an ordered arrangement of two or three atoms, atoms being either nodes (semantic objects) or relations. A triple is generally of the form: subject node relation object node. Triples are asserted and negated in the network by rule actions, and have creation and deletion times, user set reality values, and hypotheticality values associated with them.

Atoms--nodes and relations--are abstract semantic entities whose meanings are defined by the supplied data base. They may be linked to a lexical dictionary which contains roots in the vocabulary of the chosen surface language, and to lexical triples, which are triples not included in the semantic network but useful for encoding multi-word strings as single entities. Nodes may also point to other triples in the network, functioning as complex predicate nodes, thus allowing lists of triples.

Classes of nodes and relations are provided for. Subscripted classes permit class union and quantification. Loops are used to step through the elements of a class or the triples of a complex predicate node.

A program running on the simulation system goes through two phases. During compilation time the program is translated into an internal form, with syntax checks being made. If the program compiles without error, it is then executed.

This manual will attempt to give a thorough description of the language, including rules for its use and examples. It will be assumed that the reader has a basic idea of simulation techniques, as well as some understanding of the uses of this particular system. Not all cases of every language construct have been checked or described, especially those which, although perhaps legal, would most likely never be used. Also, the simulation language is constantly changing and evolving, although the changes are usually in the direction of adding new abilities and not changing present ones.

As this manual is intended both for learning the language and for reference purposes, the description of each construct is complete. Therefore, when reading for initial learning purposes the more complicated aspects can safely be ignored.

In particular, the following chapters and sections can be omitted until the basic language has been learned: 3.1.4, 3.2, 4.2, 4.2.1, 5.2, 7., all parts of 12.5 (but should be looked at on the second pass), 13. (a chapter giving a powerful addition to the basic language), 15., 17., 19., 20., 21., 23.4, many of the details of 25., all parts of 26.2, 26.3.2, the SECondary functions in 27., 27.8, 28., 29., 30.2, 30.3.

A modified BNF is used to present the syntax of the various constructs:

[] = optional
$\{\}^*$ = zero or more repetitions
$\{\}^+$ = one or more repetitions
$\{\}^{\text{lower limit-upper limit}}$ = from lower limit to upper limit number of repetitions
(An explanation of the number of repetitions may be used instead.)

When appropriate, an English description is included in the BNF. Finally, syntactic entities are placed inside the brackets, < and >, in syntax descriptions, but the notation is not always carried into the text when the meaning is clear.

2. General Form of a Program.

An outline of a typical program is given below, along with a description of each of the major components. This sample structure is not mandatory, as each of the components may be repeated and the order changed. However, the given form is typical as it is easiest to use and understand.

2.1 Outline of a program.

```
$LIMITS command, giving range of simulated time and several system options.
$NODES; followed by a list of all nodes and their associated lexical
        dictionary  elements (and information for the generative
        mechanism).
$RELATIONS; followed by a list of all relations, similar to the list of
            nodes.
$CLASSES; followed by a list of all classes and subscripted classes,
          detailing any initial members they may contain.
$EXCLUSIONS; followed by exclusion lists, allowing automatic deletion
             of contradictory triples.
$NETWORK; followed by actions which initialize the semantic network.
             { $GROUP command, giving timing and initial status for this named group.
             { $RULE followed by rule actions and subrule conditions for this rule.
             {  .
any number   {  .
of sets of   { $RULE
             {  .
             {  .
             { $ENDGROUP;
$END; signalling the physical end of the program and always the last
     command.
```

The sections preceding $GROUP are used to define items and initialize system variables and the simulated universe. The information in these sections is acted on when reached, and is stored for use by the simulation program. The program which actually controls the simulation is contained in the sets of groups, and is first compiled into an internal form and then executed.

2.2 Rules For Constructing Names.

Nodes, relations, classes, variables, and groups have names, and rules, switches, and loops may have them. Names must start with a letter, followed by letters and/or digits to any length; however, as only the first eight characters are saved internally, RULENAME1 and RULENAME2 would be taken as the same name by the system. With some exceptions, all names must be unique. There are also a number of reserved words which are defined by the system and may not be used (see below). The exceptions to the uniqueness rule for names are:

1. Groups and class names may also be atoms. If they are not explicitly defined as atoms they will automatically be defined as nodes. A single name could be an atom, a class, and a group.
2. Loop variables names and subrule variable names may have been previously used as the same type of variable name as long as the previous use is no longer in existence.
3. Labels--names of rules, loops, and switches--are known only in the group in which they are defined and are known only as labels. Therefore, they can be used for a different purpose in the same group and for any purpose in other groups.

2.2.1 Reserved words.

These names may not be used for any user-defined purpose. Their meanings are defined by the system and they are described in appropriate places in the manual.

ABS	EVER	NEQP	TIMER
AND	EVERSEC	NOT	TO
CLOCK	FLOAT	NUM	VAL
DEL	FROM	NUMP	VALSEC
DELSEC	GE	OR	
DUR	GT	PICK	
DURSEC	LE	RANDOM	
ENTIER	LT	RV	
EQ	MOD	RVAL	
EQL	NE	RVALSEC	
EQLP	NEQ	SEC	

2.3 Card Format.

Programs written for the system can be punched in free format, with specific rules given below. For readability, however, it is recommended that some standard format be used.

1. Only columns 1-72 of a card will be processed. Columns 73-80 are ignored but will be printed out with the program listing.
2. All names, reserved words and other system functions, numbers, and durations must be preceded and followed by at least one delimiter. Delimiters include blanks, which may be freely used subject to the rules in (3) below, and special symbols such as commas, semicolons, colons, parentheses, equal signs, etc., which are used only where required.
3. Any number of blank spaces may appear wherever one space is legal. Spaces are legal everywhere except: within names and reserved words, within numbers and durations, anywhere in an option field, and anywhere in a trace field.
4. Program lines may continue onto successive cards, the break from one card to the next being legal at any place where a blank space would be allowed. There is an assumed blank space between column 72 of one card and column 1 of the card following it.
5. Comments may be inserted anywhere in the program by placing a per cent sign ('%') in column one of a card; anything on the rest of that card (all 79 remaining columns) will be printed out with the program listing but will be ignored by the system. Note that comments must appear on their own cards, and that to continue a comment over successive cards requires percent signs on each of the cards.

2.4 Sample Program.

The sample program below is quite simple and is shown only to give an idea as to how a program might look. This program only demonstrates some of the many capabilities of the language. Numerous comments are included to explain the program.

```
%THIS IS AN UPDATED VERSION OF A SMALL PIECE OF THE ORIGINAL MURDER
%MYSTERY PROGRAM.  ONLY TWO GROUPS ARE SHOWN, AND THE ONLY ATOMS
%AND CLASSES DEFINED ARE THOSE WHICH ARE USED IN THOSE GROUPS.
%THE REVISIONS IN THE PROGRAM ARE DUE TO CHANGES IN THE SYNTAX OF
%THE LANGUAGE, CHANGES TO SOME POORLY WRITTEN CODE, AND DELETION
%OF A FEW SECTIONS WHICH WOULDN'T ADD TO THIS EXAMPLE.
%
%THE STARTING AND ENDING TIMES OF THE SIMULATION ARE GIVEN.
$LIMITS    START=19W3D10H, END=20W2D14H;
%
%ALL OF THE NODES (SUBJECTS AND OBJECTS) ARE LISTED.   THE NUMBERS ARE
%USED BY THE GENERATIVE MECHANISM.  THE QUOTED STRINGS REPRESENT THE
%'MEANING(S)' OF EACH NODE.
$NODES;
AFFAIR 2 = 'AFFAIR';
DETAILS 1 = 'DETAIL';
DRHUME 0 = 'DR. HUME' 'DR. BARTHOLOMEW HUME' 'HUME';
JAMES 0 = 'JAMES';
JOHNBUX 0 = 'JOHN' 'JOHN BUXLEY';
LADYBUX 0 = 'LADY BUXLEY';
LADYJANE 0 = 'JANE' 'LADY JANE';
LORDED 0 = 'LORD EDWARD' 'EDWARD';
LOVER 0 = 'LOVE';
MARION 0 = 'MARION';
MOVIE 0 = 'MOVIE';
MOVIG 0 = ;
PARK 0 = 'PARK';
PASSION 0 = 'PASSION';
RUNINTOG 0 = ;
TELEPHONE 0 = 'TELEPHONE';
TENNISCOURT 2 = 'TENNIS COURT';
THEY 0 = 'THEY';
%
%ALL RELATIONS ARE LISTED.  THE NUMBERS AND QUOTED STRINGS ARE AS FOR
%NODES.  RELATIONS WITHOUT LEXICAL LISTS WILL LATER BE GIVEN 'MEANING'
%BY USING A LEXICAL TRIPLE ACTION.  A RELATION WITH NUMBERS IN ITS
%LEXICAL LIST WILL HAVE ITS LEXICAL ITEM PICKED ACCORDING TO THE
%VALUE OF THE TRIPLE IN WHICH THE RELATION APPEARS, THUS PROVIDING
%GREATER CONTROL.
$RELATIONS;
AFFECTION 3 0 = /'HATE'/-2.5/'DISLIKE'/-0.5/'LIKE'/2.5/'LOVE'/;
BLACKMAIL 3 0 = 'BLACKMAIL';
CARESS 3 0 = 'CARESS';
FLIRT 3 0 = 'FLIRT';
FLIRTWITH 3 0 = ;
FOLLOW 3 0 = 'FOLLOW';
GOSSIP 3 0 = 'GOSSIP';
GOZZIP 3 0 = ;
INVITE 3 0 = 'INVITE';
IS 3 0 = 'BE';
```

```
LEAVE 3 0 = 'LEAVE';
POS 5 0 = 'BE';
RUN 3 0 = 'RUN';
RUNINTO 3 0 = ;
SEE 3 0 = 'SEE';
SMILE 3 0 = 'SMILE';
TALK 3 0 = 'TALK';
TALKWITH 3 0 = ;
%
%ADJECTIVE RELATIONS
%
ATTRACTIVE 2 0 = /'UGLY'/-0.5/'PRETTY'/1.5/'BEAUTIFUL'/;
JEALOUS 2 0 = /'NOT JEALOUS'/0.5/'JEALOUS'/;
MARRIED 2 0 = 'MARRY';
WEALTH 2 0 = /'IMPOVERISHED'/-2.5/'POOR'/0.5/'WELL TO DO'/
                   2.5/'RICH'/;
%
%PREPOSITION RELATIONS
%
BY 4 0 = 'BY';
IN 4 0 = 'IN';
INTO 4 0 = 'INTO';
NEAR 4 0 = 'NEAR';
WITH 4 0 = 'WITH';
%
%CLASSES CONTAIN SETS OF RELATED ATOMS.  IF A LIST OF ATOMS IS GIVEN,
%THEY ARE THE INITIAL ELEMENTS OF THE CLASS.  A CLASS MAY BE
%SUBSCRIPTED, AS IS THE SPOUSE CLASS IN ORDER TO PROVIDE A SIMPLE WAY
%OF FINDING THE SPOUSE OF ANY PERSON.  CLASSES CAN HAVE THEIR ELEMENTS
%CHANGED DURING THE SIMULATION.
$CLASSES;
DETECT = DRHUME;
FEMALE = LADYBUX LADYJANE MARION;
LOSER = ;
MALE = DRHUME JOHNBUX JAMES LORDED;
PLASE = PARK MOVIE TENNISCOURT;
RENDM = ;
SPOUSE(JAMES) = MARION;
SPOUSE(LADYJANE) = LORDED;
SPOUSE(LORDED) = LADYJANE;
SPOUSE(MARION) = JAMES;
%
%THIS SECTION INITIALIZES THE SIMULATED UNIVERSE.  THE *LEXTRP ACTION
%IS USED TO ENCODE A SINGLE ATOM AS A MULTI-WORD CONSTRUCT.  THE *SET
%ACTION GIVES INITIAL VALUES TO SEVERAL ASSERTIONS.  FOR EXAMPLE,
%DR. HUME IS GIVEN A WEALTH OF -2, WHICH, ACCORDING TO THE DEFINITION
%OF THE RELATION WEALTH, MEANS THAT HE IS 'POOR.'
$NETWORK;
                   *LEXTRP (GOSSIP WITH) TO GOZZIP,
                   *LEXTRP (RUN INTO) TO RUNINTO,
                    *LEXTRP (TALK WITH) TO TALKWITH,
                    *SET (DRHUME WEALTH) = -2,
                    *SET (LADYJANE WEALTH) = 3,
                    *SET (LADYBUX ATTRACTIVE) = -2,
                    *SET (LORDED WEALTH) = 3,
                    *SET (LORDED AFFECTION LADYJANE) = 1,
                    *SET (LADYJANE AFFECTION LORDED) = 1,
                    *SET (LADYJANE ATTRACTIVE) = 1;
%
%                   TWO PERSONS HAVE AN AFFAIR.  DEPENDING ON WHO
%                   SEES THEM, IT CAN GIVE RISE TO BLACKMAIL,
%                   GOSSIP, OR JEALOUSY.
%
%THIS GROUP WILL BE ACTIVATED BY THE GROUP WHICH FOLLOWS.  A GROUP
%SPECIFIES A SET OF RULES WHICH TEST AND CHANGE THE UNIVERSE AND
%WHICH ALL SEEM TO OCCUR AT THE SAME INSTANT OF TIME.
%
$GROUP MOVIG:       1ØM/OFF;
%THE GROUP IS DEACTIVATED SO IT WON'T AUTOMATICALLY REPEAT IN TEN
%MINUTES.
$RULE:              *DISABLE MOVIG;
%THE CLASS RENDM ALREADY HAS AS ELEMENTS A MALE AND A FEMALE WHO ARE
%GOING TO HAVE AN AFFAIR.  THE EFFECT OF THE NEXT FOUR COMMANDS IS TO
%ASSIGN ONE OF THOSE PEOPLE TO THE VARIABLE X AND THE OTHER TO THE
%VARIABLE Y, LEAVING THE CLASS RENDM WITH BOTH ELEMENTS.
$LOOP:              X.PICK(RENDM);
$RULE:              *REMOVE X FROM RENDM;
$LOOP:              Y.PICK(RENDM);
$RULE:              *ADD X TO RENDM;
%FOUR THINGS ARE ASSERTED: X IS WITH Y IN SOME RANDOMLY PICKED PLACE,
%X IS NEAR Y, THEY CARESS, AND THEY ARE LOVERS.
$RULE:              *SEC(X WITH Y)(Y IN PICK(PLASE)),
                    (X NEAR Y),
                    *SEC(Y CARESS X)(CARESS WITH PASSION),
                    *SEC(Y IS LOVER)(LOVER POS X);
%THE EFFECT OF THE LOOP AND SWITCH IS TO PICK A FEMALE WHO IS DIFFERENT
%FROM THE FEMALE HAVING THE AFFAIR.  THIS NEW PERSON WILL SEE THE
%AFFAIR, AND IS ASSIGNED TO THE VARIABLE Z.
$LOOP L1:           Z.PICK(FEMALE);
$SWITCH:            T(L1);
1Ø,-1Ø:             (Z EQL RENDM);
%THE ACTIONS IN THIS RULE WILL BE EXECUTED ONLY IF SOME CONDITIONS ARE
%MET.  IF THEY ARE MET, THEN Z WILL BLACKMAIL Y, Z THEREFORE GETTING
%WEALTHIER AND Y GETTING POORER.
$RULE:              T(R1)
                    *SEC(Z SEE AFFAIR)(Z FOLLOW THEY),
                    (Z BLACKMAIL Y),
                    *SET (Y WEALTH) =- 1,
                    *SET (Z WEALTH) =+ 1;
%THESE ARE THE FOUR CONDITIONS CONTROLLING THE ABOVE RULE.
```

```
%IF Z IS THE DETECTIVE OR THE SPOUSE OF EITHER PERSON HAVING THE AFFAIR,
%Z WON'T BLACKMAIL Y.
-1Ø,Ø:              (Z EQL DETECT) OR (Z EQL SPOUSE(RENDM));
%Y MUST BE MARRIED.
.8,-1Ø:             (Y MARRIED);
%Z CAN'T ALREADY BE BLACKMAILING Y.
-1Ø,Ø,C:            (Z BLACKMAIL Y);
%Z IS MUCH LIKLIER TO BLACKMAIL Y IF Z NEEDS THE MONEY.
.5,-.2:             VAL(Z WEALTH) LT 1;
%IF Z IS THE SPOUSE OF ONE OF THE PEOPLE HAVING THE AFFAIR, Z WILL GET
%VERY JEALOUS.
$RULE:              T(R1)
                    (Z SEE AFFAIR),
                    *SET (Z JEALOUS) =+ 2;
1Ø,-1Ø:             (Z EQL SPOUSE(RENDM));
%IF Z DOESN'T BLACKMAIL ANYONE OR GET JEALOUS, Z MAY STILL GOSSIP
%ABOUT THE AFFAIR.  Z WILL PROBABLY TALK WITH THE SPOUSE OF X IF X
%IS MARRIED AND IF Z IS NOT BLACKMAILING X.
$RULE:              (Z SEE AFFAIR),
                    *SEC(Z GOZZIP SPOUSE(X))(GOZZIP BY TELEPHONE)
                         (GOZZIP WITH DETAILS);
Ø,-1Ø:              (X MARRIED);
-1Ø,.8,C:           (Z BLACKMAIL X);
%THESE ENDLOOPS ARE REQUIRED TO END THE THREE LOOPS USED ABOVE.
$ENDLOOP;
$ENDLOOP;
$ENDLOOP;
%THE CLASS RENDM IS ERASED, SINCE THE TWO PEOPLE IN THE CLASS HAVE BEEN
%TAKEN CARE OF.
$RULE R1:           *ERASE RENDM;
%THIS ENDS THE GROUP.  THE NEXT SCHEDULED GROUP IS NOW EXECUTED.
$ENDGROUP;
%
%
%
%                   GUY MEETS A GIRL.  THEY TALK, POSSIBLE AFFAIR.
%
%THIS GROUP IS ACTIVATED FROM A GROUP NOT SHOWN IN THIS EXAMPLE.  IF
%CERTAIN CONDITIONS ARE MET, IT MAY ACTIVATE THE ABOVE GROUP.
%
$GROUP RUNINTOG:    1D/OFF;
%ONE AT A TIME, M GETS AN ELEMENT OF THE CLASS OF MALES.  IT GETS A NEW
%ELEMENT WHEN THE ENDLOOP OF THIS LOOP IS REACHED.  A LOOP ALLOWS YOU
%TO EXECUTE THE SAME SECTION OF CODE WITH EACH OF THE MEMBERS OF A
%CLASS.
$LOOP:              M.MALE;
%IF ANY FEMALE HAS ALREADY FLIRTED WITH M, OR IF M LIKES ANY FEMALE,
%THEN THE NEXT M IS CHOSEN.
$SWITCH:            T($NEXT M);
1Ø,Ø:               (FEMALE FLIRTWITH M);
```

```
10,-10:             VAL(M AFFECTION FEMALE) GT 2;
%THE FEMALES ARE LOOPED THROUGH ALSO, WITH W REPRESENTING THE CHOSEN
%FEMALE.
$LOOP:              W.FEMALE;
%IF M IS W'S HUSBAND, OR IF ONE OF THEM IS BLACKMAILING THE OTHER, THE
%NEXT W IS CHOSEN FOR THIS M.
$SWITCH:            T($NEXT W);
10,-10:             (M EQL SPOUSE(W)) OR (M BLACKMAIL W) OR
                                   (W BLACKMAIL M);
%AN ACCEPTABLE M AND W HAVE NOW BEEN FOUND.
%IF W IS REASONABLY ATTRACTIVE, THERE IS A 45% CHANCE THAT THIS RULE
%WILL BE EXECUTED; OTHERWISE THERE IS ONLY A 5% CHANCE. IF THE RULE IS
%EXECUTED, THEN M WILL RUN INTO W SOMEWHERE, THEY'LL TALK AND W WILL
%FLIRT WITH M.  IF THIS DOESN'T HAPPEN, THE NEXT W WILL BE CHOSEN.
$RULE:              F($NEXT W)
                    (W IN PICK(PLASE)),
                    (M RUNINTO W),
                    (M TALKWITH W),
                    (W FLIRTWITH M);
.4,0:               VAL(W ATTRACTIVE) GT 0;
:                   .05;
%IF M AND W HAVE FLIRTED, THEN IF W DOESN'T REALLY LIKE ANY OTHER MALE,
%THEN: M AND W'S MUTUAL AFFECTION WILL GROW, THE ABOVE GROUP DESCRIBING
%THEIR AFFAIR WILL BE ACTIVATED, W'S SPOUSE WILL BE CONSIDERED A LOSER,
%AND M AND W WILL BE PUT IN THE CLASS RENDM FOR USE BY THE ABOVE GROUP,
%AND THIS GROUP WILL BE EXITED FROM.
$RULE:              T($ENDGROUP)
                    (M INVITE W),
                    *SET (M AFFECTION W) =+ 1,
                    *SET (W AFFECTION M) =+ 1,
                    *ENABLE MOVIG,
                    *ADD SPOUSE(W) TO LOSER,
                    *MOVE M TO RENDM,
                    *ADD W TO RENDM;
-10,10,C:           VAL(W AFFECTION MALE) GT 2;
%IF M AND W DID FLIRT, BUT THEY ARE NOT GOING TO HAVE AN AFFAIR, THEN
%M JUST SMILES AND LEAVES, AND THE GROUP IS EXITED FROM.
$RULE:              ($ENDGROUP)
                    *NEG(M RUNINTO W),
                    (M SMILE), (M LEAVE W);
$ENDLOOP;
$ENDLOOP;
%THIS ENDS THE GROUP.  AT MOST ONE MALE AND ONE FEMALE HAVE MET, AND
%UNDER CERTAIN CONDITIONS THEY WILL HAVE AN AFFAIR.
$ENDGROUP;
%THIS IS ONLY A PROGRAM SEGMENT; A COMPLETE PROGRAM WOULD END WITH $END.
```

3. Simulated Time and System Options.

Simulated time is kept by an internal clock, which always contains the number of basic time units, ordinarily minutes, which have elapsed since the beginning of the simulation. The simulation may start and end at any desired simulated times, and the time units may be changed from the regular ones. There are options for seeding the random number generators, listing the program, and saving a compiled program.

3.1 $LIMITS.

The $LIMITS command is generally the first command in the program, as it usually initializes the clock. If no $LIMITS command is present, all options are set to their defaults and a warning is given. There may be more than one $LIMITS command in a program, but as the information in this command is used only during compilation, it makes little sense (and could cause problems) to use multiple $LIMITS commands, except in one special case (see 31.).

The syntax of the $LIMITS command is:

```
$LIMITS [SAVE,] [RESTORE,] [NOLIST,] [JUMBLE=<unsigned integer>,|CLOCK,]
      [START=<time>,] [END=<time>] ;
```

All items in the $LIMITS command are optional, and any or all may appear, but only in the given order. There should be no comma just before the semicolon.

3.1.1 Time Limits.

The START and END times initialize the clock and end the simulation. The clock is initially set to the START time, which will generally be zero time units. However, as events may have occurred before the simulation begins (for example, John has loved Mary for three weeks), the START time may be set by the programmer (see 8.). If no START time is explicitly given, it defaults to zero and the simulation begins with no basic time units elapsed.

The END time specifies when the simulation will terminate. All groups scheduled for execution on or before the END time will be executed, with the simulation ending immediately after the last such group has finished. Note that any groups scheduled for exactly the END time will be executed. The simulation may end before the clock reaches the END time if no further groups are scheduled or if a *END action is executed (see 18.). If no END time is specified, it defaults to 99999999 basic units of time. If the END time is less than or equal to the START time, only groups scheduled for the START time are executed.

<time> is specified in time units which are initially weeks, days, hours, and minutes, with minutes being the basic unit of time. These units may be changed (see 3.2). Each time unit is denoted by its first letter, ordinarily W for weeks, D for days, etc. Each letter which is to be used is preceded by an unsigned integer, and combined with any other number-letter pairs into a single string, with no internal blanks. For example, 3H10M would be converted into 190 minutes, and 2W19H27M into 21327 minutes. There is no limit to the number preceding any unit, so 72M and 34H9M are valid. The time units may be in any order in a <time> specification.

3.1.2 Seeding The Random Number Generators.

There are two random number generators in the system, one for the PICK function (see 11.2) and the other for subrules (see 24.). Ordinarily, whenever a program is run, each random number generator begins with the same number as it always does, and produces the same sequence of numbers. This allows you to rerun a program and get the same results, which is very important for debugging. However, in order to be able to run a program and get different results, the random number generators can be reseeded. Notice that even if you do reseed the random number generators you can still rerun any particular execution of a program by using the same seed values.

JUMBLE=<unsigned integer> uses the given integer to reseed both random number generators. CLOCK uses the computer's internal clock to reseed both random number generators. If CLOCK is used, the seed value is printed out so, by using JUMBLE, the run can be duplicated. CLOCK is especially useful when running the same program numerous times, as the seed number need not be changed for the different executions of the program.

The random number generators may also be changed during the execution of the program (see 21.).

3.1.3 Program Listing.

If NOLIST is present, then from this point on the program will not be listed, and only the change stack will be printed. If the $LIMITS command with NOLIST is the first card, it will be the only card printed. If NOLIST is not present, the program and associated information regarding the initial members of classes will be listed. In large programs, this listing can be quite large and expensive and can generally be avoided (see 31.).

3.1.4 Saving a Compiled Program.

SAVE and RESTORE allow repeated execution of a program without recompilation. Since compilation usually takes considerably more time than execution, this option can save a significant amount of time and money. It is accomplished by compiling the program once, for the first execution, and saving the compiled version on a temporary file. Therefore, for the first execution of the program you include SAVE in the $LIMITS command, with the program following as usual. However, each successive execution of the program requires only the $LIMITS command, with RESTORE in it. Also possible is adding additional groups which can be compiled and added to the saved version; this is a very powerful feature for incremental program construction.

In order to get different results from each execution, the JUMBLE or CLOCK option should be used. Use NOLIST to prevent producing multiple copies of your program. For more details, and several examples, see chapter 30.

There is also an action *SAVESYS, which can save the system at any point during execution. See 30.3 for details.

3.1.5 $LIMITS examples.

a. $LIMITS START=0M, END=3D5M;
   The program will be listed, the usual random number sequences used, and the simulation started at zero minutes and terminated at 3 days and 5 minutes.
b. $LIMITS NOLIST, CLOCK, END = 5W2D10M;
   The program won't be listed, the random number generator will be seeded by the computer's internal clock (with the seed value printed out), and the simulation will begin at zero minutes and end at the specified END time.
c. $LIMITS JUMBLE = 37, START = 2D5M, END = 6W3D4H;
   The program will be listed, the random number generators reseeded using the number 37, and the simulation will have the speicfied START and END times.

3.2 $TIME

The units of time used are initially weeks (W), days (D), hours (H), and minutes (M). You may decide to use different units of time, however, and the $TIME command allows you to specify any units you wish. One unit must be the basic unit of time, with all other units defined in terms of the basic unit. One time unit may specify the number of basic time units in a simulated 'day' (your 'day' need not be called 'day'); this is used internally to produce a value for the dayclock, which is the number of time units elapsed modulo the number of units in the 'day.' There may be up to ten different time units.

The $TIME command should occur before any time units are used. Multiple $TIME commands could cause serious problems, as one set of units might be in effect while compiling several groups while another set of units might be in effect while compiling other groups.

The syntax of the $TIME command is:

$TIME {<unit name> = <number of basic time units>[*]}$^{1-10}$;

<unit name>may be any name, up to twelve characters. The names are important only for their first letters, which are used in durations. Make sure that no two units have the same first letters.

<number of basic time units>is an unsigned integer. The first unit must have a 1 in this position; it becomes the basic unit of time. The other units may be any multiples of the basic unit of time.

* may be present for one unit and specifies the number of basic time units in one 'day,' which is used in setting the system's dayclock. If no unit has an asterisk in its definition, then the dayclock will be the current time modulo 1440. See 27.1.2 for an explanation of the dayclock.

Note that at least one unit must be present and there may be up to ten units specified. Also, in durations only the first letter of the unit name is used.

Examples:

a. $TIME MIN=1 HOUR=60 DAY=1440 WEEK=10080;
This is the initial defintion of the time units and will be in force unless changed by a user's $TIME command.

b. $TIME JOE = 1 PETE = 42 MIKE=12345 STEVE=84* BOB=987;
Five time units are speicified, with JOE being the basic time unit, PETE being 42 multiples of JOE and STEVE being the time in one 'day.' A duration written with these units in effect might be: 4B3S2J.

4. Nodes.

Nodes are atoms which will be used as subjects and objects, and thus are usually nouns and sometimes pronouns. The names these nodes will have in the program must be listed, along with their 'meanings.' A node has no inherent meaning, but can be given one by linking it to the lexical dictionary. This dictionary will not be discussed here, but the linkage methods will be. These links are established in the lexical expression list, which will be a major topic of this chapter. However, the system may be run without a dictionary or lexical expression list, as will be described.

4.1 $NODES.

The $NODES; command must precede the list of nodes. All nodes used in the program must be listed before they are used. Therefore, all nodes are generally listed together. The nodes need not be listed in alphabetical order, but for increased readability, nodes should be listed one per line and in order.

Class names and group names are the only exceptions to the rule that all nodes must be listed. Both class names (with a special marker) and group names may be used as nodes even if not explicitly declared to be nodes. However, if either type of name is to have a link to the lexical dictionary, the name should be included in the node list. See chapter 6 for more information about classes and chapter 9 for more information about groups.

Following the $NODES; command is a list of nodes, each of the form:
<nodename><grammar number> = [<lexical expression list>];

<nodename>is an internal name for the node and can be any legal name (see 2.2). While these names are completely arbitrary, they should naturally represent their meanings, with JOHN being used to stand for the person whose name is John, instead of using the perfectly legal node name X123Y. These internal node names will be used when the change stack is printed.

<grammar number> is an unsigned integer which will be used by the generative mechanism. It must be present, even if the generator isn't being used. In examples in this manual, it will always be zero.

<lexical expression list> is optional and will be described in the next section. Nodes can also be linked to be dictionary by program actions (see 15.).

4.2 Lexical Expression Lists.

A node may be given a meaning by associating a lexical expression list with the node. This list points to items in an already constructed lexical dictionary. If the generative mechanism is not being used, a dictionary is not needed and nodes need not have lexical expression lists. (Any such lists will just be ignored, which is useful during debugging.) Except for this section and several others where lexical items are discussed, it will be assumed that the generative component will not be used, and output will be in the form of a change stack with no associated natural language output.

The syntax of a lexical expression list is:
<lexical expression list>::={{'<dictionary entry>'}*{(<lexical triple>)}*}$^+$
This syntax is rather confusing; simply put, a lexical expression list is composed of any number of <dictionary entry>s in quotes and <lexical triple>s inside parentheses, in any order. Several nodes may include the same item(s) on their lexical expression lists. If more than one item is on a lexical expression list for a node, the generative mechanism will randomly pick one item. Note that only the internal node names are printed in the change stack.

A<dictionary entry>is any item that is in the dictionary. It is usually a word, but may be any character string, and is placed inside quotes. A <dictionary entry> may thus be a series of words, such as 'MR. SMITH', but in this case a grammatical transformation may not be possible; in most cases lexical triples (see below and 15.) are better than quoted strings consisting of more than one word.

4.2.1 Lexical Triples.

A lexical triple looks like a semantic triple (assertions about the universe, see 12.) but is not included in the semantic network. Lexical triples are useful for encoding multi-word strings as single entities. In many cases, a multi-word construct could be defined either as a multi-word quoted string or as a lexical triple, or it may sometimes be possible to just use single atoms and connect them in the program by how they are used in semantic triples. Which method is best depends upon the specific case and the grammar being used, and will become more evident with experience.

This section will concentrate on defining lexical triples in the node list, but this will be seen to be somewhat unwieldy. It is suggested that for most uses, lexical triples be defined by a program action (see 15.).

The syntax of a lexical triple is:
(<atom position><atom postion>[<atom name>])
where <atom position>::=<atom name>|-

The triple is composed of two or three items, each item being either an <atom name> or a dash (a minus sign on a keypunch). An <atom name> is a node name or a relation name. (It may be a class name acting as an atom, if the class has been defined already. See 6.).

The first, and the third if used, items must be node names and the middle item a relation (see 5.). For example, the node BALLOFTHREAD might have its meaning the lexical triple BALL OF THREAD, with the nodes BALL and THREAD and the relation OF having the obvious character strings as their meanings. A dash is used in one of the first two positions to indicate an empty position, as in the relation CALLUP being defined by the lexical triple (- CALL UP). This is necessary because of the rule given above regarding the atomic types of the three items. An item in a lexical triple may itself be defined by a lexical triple; an example of this is in the next section.

Finally, a problem would occur if all the atom names used in a lexical triple had to be defined before their use. This is because nodes and relations are usually declared in two complete lists, one for each type, and a lexical triple in the node list will generally include a relation, whose definition had not yet been reached. Therefore, it is legal to use atom names which have not yet been defined, providing that they are later defined as the correct type of atom.

Again, it is advisable to declare lexical triples by using a program action (see 15.), since in that case there are no restricting rules on the lexical triple.

4.3 Examples.

The examples will be shown as part of a single node list, with comments inserted for explanation.

```
%List of nodes.
$NODES;
%One of the three meanings will be randomly chosen by the generative mechanism.
BUG 0 = 'BUG' 'MOSQUITO' 'FLY';
DAUGHTER 0 = 'DAUGHTER';
%No lexical expression list is needed when the system is run without the
%generative mechanism.  If the generator is used, no natural language
%output will appear for this node unless it is assigned a meaning in the program
%by means of the lexical triple action.
FRIEND 0 =;
HIM 0 = 'HIM';
JOHN 0 = 'JOHN' 'MR. SMITH';
LADY 0 = 'LADY';
%One of two lexical triples or a dictionary entry will be randomly chosen.
%The node LADY has already been defined, but the other items in the lexical
%triples have not.  The grammar will take care of changing the word order.
OLDLADY 0 = (LADY OLD) 'LADY' (WOMAN ELDERLY);
%OLDMAN will be given a meaning by a lexical triple assigned to it by an
%action in the program.
```

```
OLDMAN 0 = ;
ROBERT 0 = 'ROBERT';
%Here, a lexical triple uses a node defined by another lexical triple.  How
%this will finally turn out in the natural language depends upon the grammar
%being used.
SWEETOLDLADY 0 = (OLDLADY SWEET);
%This could have been defined as a lexical triple instead.
TENNISCOURT 0 = 'TENNIS COURT';
WOMAN 0 = 'WOMAN';
```

5. Relations.

Relations are atoms which will be used as verbs, adjectives, adverbs, prepositions, etc. The basic concepts governing relations are the same as those for nodes (see 4.). Relations are listed with their 'meanings' described by associated lexical expression lists.

5.1 $RELATIONS.

The $RELATIONS; command must precede the list of relations. The list is similar to the node list except that relations have some additional information associated with them. One other difference in the list is that since class names and group names will automatically be nodes if not declared otherwise, they should be in the relation list if they are to be used as relations. (See chapter 6 for more information about classes and chpater 9 for more information about groups.) Finally, the lexical expression list for a relation may be like that for a node or may include numeric information (see 5.2).

The form of each relation in the relation list is:

<relation name>[<relation type>]<grammar number><grammar number>
    = [<lexical expression list>];

<relation name> is an internal name for the relation and can be any legal name (see 2.2).

<relation name> is T for transitive (see below) or is not used.

<grammar number> is as for nodes (see 4.1). Two numbers must be present, even if the generative mechanism isn't being used.

<lexical expression list> is optional and is as for nodes. See 4.2 and 4.2.1 as the discussion there is also applicable to relations, as are the lexical expressions used in the examples in 4.3. <lexical expression lists>s may have numeric information associated with them, as will be described in 5.2.

A transitive (T) relation is one which is specified as acting in a logically transitivite manner. For example, if ON is a transitive relation, then if (BOX1 ON BOX2) were in the network and (BOX2 ON BOX3) were added to the network, (BOX1 ON BOX3) would return a value of True in a subrule test. Also, transitive relations have automatic object exclusions (see 7.2).

5.2 Numeric Lexical Expression Lists.

It is sometimes useful for a lexical item to be chosen based upon the value of the triple in which an atom appears--this gives you greater control over lexical output than that gotten from the otherwise random selection of a lexical item. For example, the relation AFFECTION can be given a lexical list so that if (JOHN AFFECTION MARY) = 3 the word "adore" might be used for AFFECTION, while if (JOHN AFFECTION MARY) = -3 the word "despise" might be used. In between values might yield less extreme terms.

Numeric lexical expression lists may be attached to any relation. They differ from regular expression lists in two main ways: the numeric lists are divided into disjoint parts by the use of bounds, and the numeric lists may not include lexical triples.

The syntax of numeric lexical expression lists is:

[<lower bound>] {/[<lexical expression list>]/[<inclusive upper bound>]}$^+$

<lower bound> and <inclusive upper bound> are numbers, real or integer, optionally signed. They break the numeric lexical expression list into disjoint parts.

<lexical expression list> is as for nodes (see 4.2 and 4.3), except that here lexical triples are not allowed. Each list must be inside slashes.

During execution, if the generative mechanism is being used and a triple whose relation has a numeric lexical expression list is to be output, the following occurs: the current value of the triple is checked, and only those lexical items (there may be any number of them) whose left bound is less than and whose right bound is greater than or equal to the current value are candidates for the lexical item to be printed (the actual item is then randomly chosen).

An example will help:

IQ 0 0 = /'IDIOTIC'/75/'STUPID' 'DUMB'/99.9/'SMART'/127/'BRILLIANT'/;

If the IQ for some subject is $\leq$ 75, 'idiotic' will be chosen; if it is >75 and $\leq$99.9, either 'stupid' or 'dumb' will be chosen; if IQ is >127, then 'brilliant will be used.

If there is a <lower bound> it is not inclusive, but if there is no such bound, then the leftmost <lexical expression list> will be used for any value which is less than or equal to its upper (left) bound. If there is no final <inclusive upper bound>, then the last <lexical expression list> will be used for any value which is greater than its lower (left) bound. There may be no items between bounds, in which case a value between those bounds would yield no lexical output, as could also be the case if the first <lower bound> or final <inclusive upper bound> were inside the current value of a triple.

5.3 Examples.

The examples will be shown as part of a single relation list, with comments inserted for explanation.

```
%List of relations.
$RELATIONS;
AFFECTION 0 0 = /'LOATHE' /-2.5/'DISLIKE' /-0.5/'LIKE'
   /2.5/'LOVE' 'ADORE'/;
%One of these two meanings will be randomly chosen by the generative mechanism.
ASK 0 0 = 'ASK' 'REQUEST';
%UP must be a node for this to be allowed.  It is easier to use the lexical
%triple action.
CALLUP 0 0 = (-CALL UP) 'TELEPHONE';
%The next two relations both have the same meanings.  The same relation
%name could have been used, but different ones were chosen for easier
%readability of triples.  The grammar will take care of the transformations.
FIGHT 0 0 = 'FIGHT';
FIGHTING 0 0 = 'FIGHT';
FOR 0 0 = 'FOR';
HAVE 0 0 = 'HAVE';
%Values between -4 and -2 or between 3 and 5 cause lexical output,
%but other values don't
LIKE 0 0 = -4/'HATE' 'DESPISE'/-2//3/'LOVE'/5
%This relation will be used for internal counting purposes and so is
% given no lexical expression list.
```

```
NUMBER 0 0 = ;
ON T 0 0 = 'ON';
PREPARED 0 0 = 'PREPARE';
%This will get a lexical triple assigned to it during the program, as both
%PREPARED and FOR are relations and a lexical triple in a lexical expression
%list can't have two relations in it.  It is spelled without the 'E' to
%differentiate it from the relation PREPARED.

PREPARDFOR 0 0 = 'READIED';
SEE 0 0 = 'SEE' 'RECOGNIZE';
TALL 0 0 = 'TALL';
WEALTH 0 0 = /'IMPOVERISHED'/-3/'POOR'/0/'WELL TO DO'
    /2.85/'RICH' 'WEALTHY'/;
YOUNG 0 0 = 'YOUNG';
```

6. Classes.

A class is a set of related atoms. As a set, there may be no duplicates (duplicates will be ignored) and the only ordering of the elements is based on the order in which the elements were originally declared in the node or relation list, which isn't very useful. There are no arrays in the language.

The elements in a class must all be of the same type and must have been previously defined, but there can be classes of either type: classes of nodes or classes of relations.

Group names may be in classes, sometimes useful for sequencing through groups. However, the names must be already defined, so the group names should be in the node or relation list. (It is possible to define the group name by putting the program statements of the group physically before the use of the name in a class list, but this is not recommended.) Remember that if a class of group names is to be used for sequencing, the group names will have to be defined in the correct, undoubtedly non-alphabetical order in the node or relation list.

Elements can be dynamically added to and removed from classes during the simulation (see 14.). Loops are available to go through the elements of a class, one at a time (see 23.). An element may be randomly picked from a class (see 11.2), and there are tests for class membership, determination of subsets, and checks for the inequality of classes (see 26.3.1), as well as a function giving the number of elements in a class (see 27.3).

6.1 Class names.

A class name is automatically made a node if it has not been previously defined. In this case, the class can only contain nodes as elements. In order for a class to contain relations, the class name _must_ be defined as a relation _before_ the class is defined; the class may then contain only relations.

Even if a class is to contain nodes, its name may be put on the node list in order to assign a lexical expression list to it, although it would always be possible to dynamically assign lexical triples to the class name since it would become a node name in either case,

Since class names can be used either to represent the elements in the class or to represent the atom having that class name, a marker (/) is used before the class name in order to signify the second usage. So, if PEOPLE were a class containing nodes (people's names), PEOPLE would refer to the elements in the class, and /PEOPLE to the node PEOPLE.

6.2 Subscripted Classes.

A class name may be subscripted. However, as there are no arrays, the subscripts must be atoms, and may not be numbers. A subscript may be any defined node or relation, regardless of the type of class it is being used with.

A subscripted class may have any number of different subscripts, with each class-subscript pair acting as a separate class, uniquely defined by the class name and the particular subscript. However, since the class name remains the same over all the different subscripts, the important class manipulation capability of class union is gained.

Subscripted classes allow easy class union because a subscripted class need not have only a single atom as its subscript (except when defining a particular subscripted class). The subscript may be a second class, which effectively yields the union of the classes whose name is the subscripted class and whose subscripts are the elements of the second class. To clarify: if PEOPLE is a class composed of the nodes JOHN and DAVE, and FRIENDS(JOHN) is a subscripted class composed of TOM and DICK, and FRIENDS(DAVE) a subscripted class composed of TOM and MARY, then FRIENDS(PEOPLE) would be the union of FRIENDS(JOHN) with FRIENDS (DAVE), or TOM, DICK, and MARY.

A subscript may also be a subscripted class, adding to the class union capability. For more details, see 11.2.

Once a class is defined as being subscripted, any number of subscripts may be dynamically assigned during the program. This actually permits dynamic class generation. A class may be defined as subscripted without giving it a specific subscript at definition time, as the subscript(s) is (are) to be dynamically determined during the simulation.

Subscripted class names may also be uses to represent the atom having that class name by preceding the name of the subscripted class with a slash. In this case, no subscript is allowed. So, /FRIENDS would refer to the node FRIENDS in the above example.

6.3 $CLASSES.

The $CLASSES; command must precede the list of classes. All classes which are to be used inthe program must be listed before they are used.

The definition of a class simply contains the class name, a subscript if desired, and an optional list of initial elements. The syntax of an element in the class list is:

<class name>[([<subscript>])] = [<initial member list>];

<class name> is any legal name (see 2.2). Unless otherwise defined, it is automatically made a node. A subscripted class name may be repeated in the list: if the subscripts differ, then different class are defined and possibly initialized; if a unique class-subscript is listed more than once, all of the initial members are unioned together to form the initial member list for that subscripted class.

<subscript> may be any already defined atom, in which case a specific subscripted class is defined. If that class with that subscript is not going to be initialized, there is no point in listing it with that subscript, as the subscript may be used in the program as long as the class is defined as subscripted. A class may be defined as being subscripted by either giving it at least one specific subscript, or, if no initialization of the class is wanted for any particular subscript, the class may be defined with the null subscript: a pair of parentheses with <u>no</u> intervening blanks (in this case an initial member list has no meaning).

<initial member list> is optional and defines the elements in the class before the simulation starts. The elements can simply be already defined atoms, or they may be any already defined classes or union of classes, or any combination of these things in any order.

If a class name or union of classes is on the intial member list, then the element of the class, or elements resulting from the union, are placed in the class being initialized. In order to put a class name acting as an atom on the initial member list, the class name must be preceded by a slash, as described in 6.1.

A more precise definition of intial member list is:
{<general atom field>}$^+$
which allows for any number of <node name>s, <relation name>s, or /<class name>s, and for <general class reference>s, which allow for classes and unions of classes. <general atom field> is fully defined in 11.2, with the exceptions that subrule variable names and loop variable names are not allowed in the initial member list as they could have no possible meanings.

6.4 Examples.

The examples will be shown as part of a single class list.

```
%List of classes.
$CLASSES;
%Class is defined as subscripted, with subscripts to be dynamically defined.
ENEMIES() = ;
FEMALES = ANN MARY SUE;
%This class will be used as an index into the REL subscripted class, useful
%for quantification.
INDEX = I1 I2 I3;
MALES = JOHN PETE;
MARRIED = JOHN ANN PETE MARY;
%PEOPLE receives five initial elements.
PEOPLE = MALES FEMALES;
%REL contains elements which are relations, so
%REL must have already been defined as a relation.  The type of the
%subscript--nodes in this case--is arbitrary.
REL(I1) = ABDUCT;
REL(I2) = EXPEL;
REL(I3) = MURDER;


%The groups will be ordered as they were defined in the node list (they
%must already be defined).
SEQUENCE = GROUP1 GROUP2 THIRDGROUP LASTGROUP;
SPOUSE(ANN) = JOHN;
```

```
SPOUSE(JOHN) = ANN;
%This expands to JOHN and ANN, and could have been written that way.
FRIENDS(PETE) = JOHN SPOUSE(JOHN);
%This expands first to SPOUSE of each of the four married people.  As
%explained in 11.2, the elements of these four classes will be unioned together.
%If a particular class-subscript pair doesn't exist, such as SPOUSE(MARY), it
%is treated as having no elements.  Therefore, FRIENDS(MARY) has as
%initial elements JOHN and ANN.
FRIENDS(MARY) = SPOUSE(MARRIED);
%RELATIVES contains 3 nodes, one of which is a node which has been defined
%as a class (when the class was defined above, it was automatically made
%a node if it wasn't already one), and so must be preceded by a slash to
%differentiate it from the class SPOUSE.  If RELATIVES had been defined
%before the class SPOUSE, and SPOUSE had been defined at a node, the
%slash would not have been needed.
RELATIVES = CHILD /SPOUSE PARENT;
%Class will be used in the program to temporarily hold nodes.
TEMP = ;
```

7. Exclusions.

Exclusions define concepts which are mutually exclusive, and therefore cannot be allowed to exist simultaneously. Mutually exclusive items are listed together, and should a triple containing one of these items be asserted, an existing triple containing one of the excluded items (in the proper context) would be negated in the network. This ability provides a convenient way of keeping inconsitent information out of the network.

The exclusions section is introduced with the $EXCLUSIONS; command. Following this command are lists of excluded items. Each list is of one of three types, relation, subject, or object exclusion, each of which is described below.

7.1 Relation exclusions.

Certain relations are mutually exclusive when used in the same subject-object context. For example, if (JOHN AT HOME) is in the network and (JOHN LEAVES HOME) is added, then the first triple should be negated. If AT and LEAVES are listed as relation exclusions, this negation will be done automatically.

Any number of relations can be declared to be mutually exclusive by putting them in a single relation exclusion list. The relations may not be transitive.

When one relation in the list is asserted in a triple using a particular subject-object pair (or particular subject for relations not having objects), then if another relation in the list already exists in the network in a triple using the same subject-object pair, this existing triple will be negated. The negation will appear on the change stack.

For example, if AT and LEAVES are listed as being mutually exclusive, (DAVE AT SCHOOL) will be negated if (DAVE LEAVES SCHOOL) is asserted. But (DAVE AT SCHOOL) naturally will be untouched if either (JOHN LEAVES SCHOOL) or (DAVE LEAVES HOME) is asserted.

As another example, if ASLEEP and AWAKE are listed as relation exclusions, (JOHN AWAKE) will negate (JOHN ASLEEP), but will leave (MARY ASLEEP) untouched. However, (JOHN AWAKE TODAY) would not affect (JOHN ASLEEP).

Relation exclusion lists are of the form:

R: {<general relation field>}$^{2 \text{ or more}}$

<general relation field> is described in 11.2, with the exceptions that subrule variable names and loop variable names have no use here.

The examples given above would be written:

```
R:  AT LEAVES;
R:  ASLEEP AWAKE;
```

7.2 Subject and Object Exclusions.

In subject exclusions, a relation is given along with a list of subjects. Two items from the list of subjects will not be allowed to coexist in triples having the specified relation and same (possibly null) object. For example, a subject exclusion might be used with a superlative relation, such as tallest. If the subject list for this relation is all people, then when one person is specified as tallest, if another person were already tallest this fact would be negated. So, (JOHN TALLEST) would negate (TOM TALLEST) and (JOHN TALLEST PERSON) would negate (TOM TALLEST PERSON), but, for example, (JOHN TALLEST) would not affect (TOM TALLEST PERSON).

Object exclusions require lists of objects for a specified relation. No one subject will then be allowed to be in the specified relation with more than one item in the object list. For example, the object exclusion may associate the relation IS with the day of the week, so if (DAY IS TUESDAY) is asserted, the triple asserting any previous day in the same manner would be negated. However, if (TODAY IS THURSDAY) were in the network, it would not be affected.

For both subject and object exclusions, as for relation exclusions, any automatically negated triple will be shown in the change stack.

An object exclusion is automatically set up for a transitive relation, with <u>all</u> objects being assumed as mutually exclusive.

Subject exclusions take the form:

```
S: <relation name> = {<general node field>}+;
```

where <relation name> cannot be a transitive relation.

Object exclusions take the form:

```
O: <relation name> = {<general   node field>}+;
```

where <relation name> cannot be a transitive relation.

In both cases <general node field> is described in 11.2, omitting loop and subrule variable names.

The examples given above could be written:

```
S: TALLEST = PEOPLE;
O: IS = DAYSOFWEEK;
```

7.3 Examples.

A sample exclusion section will be shown. The letters to the left refer to the discussion below the program section, and are not part of the language.

```
     %List of exclusions.
     $EXCLUSIONS;
a.   R: AT LEAVES;
b.   R: SUNNY RAINY WINDY;
c.   R: IN LEAVES;
d.   S: TALLEST = PEOPLE;
e.   S: STRONGEST = MALES;
f.   O: IS = WEEKDAYS SATURDAY SUNDAY;
g.   O: IN = LOCATIONS;
h.   O: IS = PRESIDENT;
```

Notes on above exclusion lists:

a. This is discussed in 8.1.

b. (WEATHER SUNNY) will cause (WEATHER RAINY) to be negated.

c. (JOHN LEAVES ROOM) negates (JOHN IN ROOM). This exclusion might be combined with (a), listing all three relations, but then (JOHN IN SCHOOL) would negate (JOHN AT SCHOOL), so care should be taken.

d. This is discussed in 7.2. PEOPLE is assumed to be a class of nodes naming people.

e. Only one male will be the strongest.

f. This is discussed in 7.2. WEEKDAYS is assumed to be a class of five nodes, being the names of the five weekdays.

g. LOCATIONS is a class of various locations, such as ROOM1, ROOM2, etc. Another way of preventing someone or something from being IN two places at one time is by declaring IN to be transitive, causing an automatic object exclusion for all objects. This exclusion complements (c) above, since now (JOHN IN ROOM) will be negated if either (JOHN LEAVES ROOM) or (JOHN IN ROOM2) is asserted.

h. This allows only one person to be the president.

8. Initializing the Semantic Network.

This section is placed here because this construct would probably appear in this position in a program. However, in order to understand the description given below, you will need to know quite a bit more about the system than has been presented in the preceding sections.

The semantic network, which contains the state of the universe represented by semantic triples, may be initialized before the execution of the simulation. This is accomplished by using the same actions as allowed in the executable program, with these actions taking place immediately. Furthermore, you can specify how long the various semantic triples should seem to have existed before the simulation begins.

As almost any program action is legal in the initialization section, more than just initializing the semantic network can be accomplished, with several specific suggestions given below.

8.1 $NETWORK.

The initialization section is specified by the $NETWORK; command. This is followed by any number of action lists, each list terminating with a semi-colon and each list composed of any number of actions separated with commas. Actions are described beginning in chapter 11, and other than some exceptions to be noted, actions in the initialization section are identical to actions which are part of rules.

Unlike rules, however, the initialization can only contain actions, which are then performed immediately. There may be no subrules (testing of the network) or loops.

Any or all of the action lists (each terminated by a semi-colon) may have durations associated with them. This is of the form:
FOR<time>:<action list>;
where <time> is described in 3.1.1 and <action list> beginning in 10. The <time> specifies the length of time before the start of the simulation that the listed actions are to seem to have taken place. Semantic triples have times associated with them, and their durations in the network can be tested.

For example, if John has loved Mary for three hours and ten minutes before the simulation begins, this might be used;
FOR 3H10M: (JOHN LOVE MARY);

If FOR OM is specified, the associated triples will act as if they were set at the starting time of the $LIMITS command. If no <time> is specified the associated triples will act as if they were set at time zero. If a <time> is given, the $LIMITS command should have already been read in, and the starting time on it should be greater than the <time> now given in order to avoid meaningless negative times.

To summarize, the syntax of the initialization section is:
$NETWORK: {[FOR<time>:]<action list>;}$^+$
Triples set during initialization are <u>not</u> printed on the change stack.

8.2 Actions Used for Initialization.

More than just the semantic network can be initialized, and the following list will give some information on each of the classes of actions that could occur during initialization.

a. Actions affecting the semantic network (12. and 13.). Any action is valid, and a triple's duration in the network can be set as explained above.

b. Actions affecting classes (14.). Any action is valid. A possible use is defining a class in terms of another class where to do so in the class list would cause unalphabetizing of the list. As loops aren't allowed, most class initialization would be done in the class list instead of here. As times are not associated with classes, a <time> specification will be ignored.

c. Actions affecting lexical items (15.). This is a major use of the network section, as lexical triples are easier to set up using the lexical triple action than describing them in lexical expression lists attached to atom definitions. As times are not associated with lexical triples, a <time> specification will be ignored.

d. Actions affecting the scheduling of groups (16.). A group may be enabled to take place at a specified time. However, be sure to use a FOR OM phrase before enable actions in order to set the timer correctly. Disabling a group would make little sense.

e. Actions affecting subroutines (17.). Calling a subroutine from the network section is illegal.

f. Miscellaneous actions. Ending the simulation (18.) makes no sense. Setting the RV (19.) is legal. Hypotheticality values (20.) could be included in any triples which are set. The random number generators (21.) could be reseeded.

g. Tracing actions (32.3). The tracing controls can be conveniently set in the network section. However, even if tracing is activated, tracing will not actually begin until the simulation program begins to execute, and no tracing of the initialization actions is possible.

8.3 Examples.

The examples will be shown as part of a single initialization section. All atoms and classes used are assumed to have been previously defined.

```
%Initialization section.
$NETWORK;
*LEXTRP (CALL UP) TO CALLUP,
*LEXTRP (SHOWN TO) TO SHOWNTO,
*LEXTRP (PREPARED FOR) TO PREPARDFOR,
*SET RV = 2;
%$LIMITS command should have already had a start time greater than 2H27M.
FOR 2H27M:(JOHN LOVE MARY);
          *SET(JOHN IQ) = 125,
          (FRIENDS(MALES)IN LIVINGROOM);
%Could have simply put Allan in the initial list of elements for the class MALES.
*MOVE ALLAN TO MALES,
%Tracing won't start until simulation begins execution.
*TSET ABCDE = 1000;
%Group is scheduled for 10 minutes past the beginning of the simulation.
FOR OM: *ENABLE GROUP3 IN 10M;
```

9. Groups.

At this point, most programs will have included the information discussed in the previous six sections. The $LIMITS command has set certain system variables, the nodes and relations are defined and given meaning, the classes are defined and initialized, exclusions are defined, and the simulated universe is initialized.

This initial information is processed as it is reached, and the universe and system are initialized. Now comes the program which describes the process to be simulated, and this section is first processed by a compiler which translates it into an internal form, and then executed.

The simulation program is divided into units called groups. The group is the major component of the language, and is composed of any number of rules, loops, and switches. Rules contain actions, which cause changes in the universe, and subrules, which interrogate the universe, and are described beginning in chapter 10.

9.1 Scheduling and Executing Groups.

A group--but not a single rule--can be executed at user-specified intervals according to the internal simulated-time clock. This clock advances in steps of one basic time unit, and at each step a check is made for groups scheduled for execution.

Groups are considered active if they have been enabled (see 16.1) or initially declared as being ON. They may be made inactive by being disabled (see 16.2), although they may be reactivated (enabled again). An active group has associated with it the time it is next scheduled for execution. The system keeps a queue of all active groups and their associated next-execution times.

At each advance of the clock, the queue is checked, and any group scheduled for the current time is executed. When all such groups have been executed, the clock advances again. If more than one group is scheduled for a particular time, they are executed in the order in which they were scheduled. If at any time there are no active groups--the queue is empty--the simulation stops, even if the end time has not been reached.

Groups may also be called as subroutines, in which case they are immediately executed (see 17.).

As the above description suggests, groups are used to describe the actions to be taken at a specific instant of time. All actions in a group seem to take place at the same time, and the group must be exited from in order for time to advance. Certain groups of actions are repetitive, and provision is made for specifying the interval at which a group is to be repeated.

Rules in groups are normally executed sequentially, but the sequence may be dynamically changed. Each group can be entered only at its beginning, either by its having been scheduled for the current time or by its having been called as a subroutine. A group is exited from only when its end is reached, although a rule may transfer to the end of a group (subroutines are an exception to this rule).

9.2 $GROUP and $ENDGROUP.

A group is defined by a $GROUP statement, after which follows any number of rules, loops, and switches. Each group is terminated by an $ENDGROUP statement.

The $GROUP statement has the following syntax:

```
$GROUP [,<option field>]<group name>:<time>/<initial state>;
```

<group name> is the name of this group (see 2.2). If it has not already been defined, it will be made a node automatically. Groups may be referenced only after they have been defined, the definition occurring either in an atom list or here at a group definition.

<time> is defined in 3.1.1. This specifies the interval at which the group is to be executed, assuming the group remains active. For example, if <time> is 24H, the group will be executed once each simulated day, as long as it is active. If the group is disabled and then re-enabled, the sequencing interval is reestablished with a

starting point of the first execution time of the reactivated group.

<initial state> is either ON or OFF. If it is ON, then the group is initially enabled and will be executed at the starting time of the simulation. If more than one group is initially ON, they will be executed in the order of their appearance in the program. At least one group must initially be ON, unless a group is enabled in the $NETWORK section (see 8.).

<option field> is optional, and must be preceded by a comma and have no internal blanks if it exists. This field is a sequence of the option characters S, C, and O, which may appear in any combination and in any order.

S signifies that the group is synchronous, meaning it will be executed only on even intervals of the specified <time> interval of the group. If the group is enabled to run at a time which is not an even multiple of the <time> interval, the next even multiple will be used. For example, if the <time> for a group is specified as 2H, the group will always be executed at even numbered hours, even if an enable action specifies that it run at some other time.

C and O are options which have meaning in subrules. C is described in 26.1.2 and O in 28.3. If one of these options appears in the group's <option field>, then it is in force for all subrules in the group, unless overridden by an <option field> at a lower (rule, switch, loop, or subrule) level.

Each group must end with an $ENDGROUP; statement. This statement must be reached (except in subroutines) for a group to be exited from. It can be flowed into from an immediately previous statement, or it can be transfered to (see 10.3). Leaving a group does not disable it, as only a disable action (see 16.2) can do that. When the group is left, if it is still active (it did not disable itself) it will be rescheduled for <time> units of time past the current time.

9.3 Examples.

Sample $GROUP statements might be:

```
$GROUP GROUP1: 3H10M/OFF;
$GROUP DINNER: 24H/OFF;
$GROUP, OC MAIN:  OM/ON; A <time> interval of OM is all right as long
   as the group disables itself.  It is often useful to have an
   initialization group turned ON, with this group scheduling other
   groups and turning itself off so it will never be repeated.
```

A complete group would look like this:

```
$GROUP  G7:  28M/OFF;
$RULE:(JOHN LOVE MARY);
 .
 .
$RULE:
 .
 .
$LOOP:
 .
 .
$ENDLOOP;
 .
 .
$ENDGROUP;
```

10. Rules.

It is in a rule that information is changed in the semantic network, class membership is altered, groups are enabled and disabled, etc. This all occurs in the action list of a rule, which is made up of any number of actions. Each action will be described separately in subsequent sections.

A rule may also have a subrule list, where the current state of the simulated universe may be tested and subrule actions may take place. The subrule test may set probability parameters which will determine whether the actions in the rule are executed. Subrules are described beginning in chapter 24.

There is also a provision for looking into the future. A different type of rule is used for this purpose, and is described in chapter 29. The disucssion of rules other than in that chapter will assume the look-ahead capability is not being used.

Given the above description, it can be seen that rules are used to unite associated actions and tests. Any number of rules may appear in a group. Only groups may be scheduled for execution, and rules inside a group are normally executed sequentially. This order may be dynamically changed, however, by use of branch fields in the rule.

10.1 Control of Execution

When a rule is executed, testing of the universe may cause the action list to be skipped with a certian probability. This allows past actions to affect what will happen later, in either a deterministic or random manner.

If a rule does not have a subrule (test) list, then the rule evaluates to True and its action list is executed, one action at a time in sequential order. Any unconditional or true branch (the same in this case) will be taken after the execution of the last action.

If a rule does have a subrule list, then the subrules are evaluated and a cumulative total of subrule probabilities is produced. (See 24. for a further description of subrule probabilities.) A random number between 0 and 1 is then generated, and if this number is less than or equal to the cumulative probability total, the rule evaluates to True, the action list is executed, and the true branch is taken. If the random number is greater than the cumulative probability, then the rule evaluates to False, the action list is not executed, and the false branch is taken.

If the action list is executed, all actions will be finished before going to the next command (except in the case of subroutines).

10.2 $RULE.

A rule is defined with the $RULE statement. This construct looks as follows:

```
$RULE  [,<option field>][<rule name>] : [<branch part>][<action list>];
        [<subrule list>]
```

<option field> is optional, and must be preceded by a comma and have no internal blanks if it exists. This field may be one or both of the option characters C and O, in any order. C and O are options having meaning in subrules. C is described in 26.1.2 and O in 28.3. If one of these options appears in the <option field> of a rule, then it is in force for all subrules in the rule, unless overrridden by an <option field> attached to a particular subrule.

<rule name> is optional, and names the rule. It is necessary if you want to transfer to this rule from another place in the group. Rule names are known only as labels and only in the group they are defined in, and so may be reused (see 2.2).

<branch part> is optional, and indicates which named rule (or switch or loop) in the group is to be executed next. It is necessary only if you want to jump out of the normal sequential execution of rules. The transfer is done only after the rule is finished, with the subrules evaluated and action list executed if necessary.

The <branch part> is further defined as:

(<branch field>) | one or both of T (<branch field>) and F(<branch field>) in any order

A <branch field> in parentheses means an unconditional transfer, and is done regardless of how the rule evaluates. A <branch field> preceded by a T is taken if the rule evaluates to True, and one preceded by an F is taken if the rule evaluates to False. There may be both True and False branches in a rule.

A <branch field> is defined as:

<statement name>|$ENDGROUP | $NEXT <loop variable>

If it is a <statement name> it must refer to a named rule, loop, or switch in the same group. A rule may branch to itself, but it shouldn't do so unconditionally, as that would cause an infinite loop. Also, you can't branch to a rule inside a loop from outside of that loop (see 23.).

A reference can be made to a <statement name> in the group even if that statement is physically after the statement doing the referencing. During compile time, when the end of the group is reached, a check is made to see if all such forward references have been resolved.

If the <branch field> is $ENDGROUP, it means to branch to the end of the group, thereby leaving the group.

If the rule is inside a loop, you may branch to the end of the loop by using $NEXT followed by the name of the loop variable used in the loop. The <loop variable> must be present, and allows you to go to the end of a particular loop if the rule is inside nested loops. Loops are described in chapter 23.

Examples of rules having branch parts are:

```
$RULE: T(RULENAM1) <action list>;
$RULE  RULENAM1:  T(RULENAM1) F(RULENAM2) <action list>;
$RULE,0: (RULENAM3) <action list>;
```

<action list> is optional, and is a list of the actions to take place if the rule evaluates to True. If the <action list> is missing, this rule is actually a switch (see 22.).

An <action list> is further defined as:

<action> {,<action>}*

and is thus any number of <action>s separated by commas. Actions are described beginning in chapter 11.

<subrule list> is optional, and composed of any number of subrules which test the simulated universe. If a <subrule list> isn't present, the rule always evaluates to True. Subrules are described beginning in chapter 24.

10.3 Examples.

```
$GROUP  WAKEUP: 1D/OFF:
%Loop through each of the people, P representing one person at a time.
$LOOP: P.PEOPLE;
%This is always executed.
RULE: (ALARM RINGS),
      (P LOOKSAT CLOCK),
      (P SILENCES ALARM);
%Half of the time P gets up and half of the time P feels sick.
$RULE:  F(SICK)
        (P GETS UP),
        (P EATS BREAKFAST);
:.5;
%P got up, but may still be tired.
$RULE: T(SICK)
       *SEC (P FEELS TIRED)(TIRED STILL);
.7,.2: CLOCK LT 5H30M;
%P goes to work if over 20.  Once one person goes to work, leave the loop
%by going to the end of the group.
$RULE WORK: T($ENDGROUP)
      (P GOESTO WORK);
10,-10: VAL(P AGE) GT 20;
%P goes to school, and the next person is chosen.
$RULE: ($NEXT P)
       (P GOESTO SCHOOL);
%P feels sick and stays home.  The next person is then chosen.
RULE SICK: (P FEELS SICK),
      (P FALLS ASLEEP);
$ENDLOOP;
$ENDGROUP;
```

Note that this language (like FORTRAN) is not block structured, leading to numerous branches. The number of branches should be kept to a minimum as an aid in reading the program.

11. Introduction to Actions.

This chapter will introduce the construct known as an action. It will also describe the operand types used in actions. The different actions will be described in subsequent sections.

11.1 General Description.

Actions may appear in the action lists of rules and in subrule action lists. All changes to the simulated universe are effected by using actions. These include changes to the semantic network, to class membership information, and to lexical lists, scheduling and calling of groups, and setting certain user and system variables.

An action list is a list of actions separated by commas. With one exception, all actions are introduced by an asterisk followed by the name of the action. Most actions deal with similar operands, and the various types of operands are described below.

11.2 Operands.

An operand is composed of atoms, class, and variable names in some prescribed format. Regardless of the type of operand, at execution time when the operand is evaluated it will finally result in a set of zero of more basic items--references to specific entities such as atoms and classes.

In the definitions, the word 'atom' is used to avoid duplication of definitions using 'node' and 'relation.' Whenever 'atom' appears, it may be replaced throughout the definition by the appropriate specific atom type.

The operand types will be given in order of complexity, with the simplest type being given first.

```
1.  <single valued atom field>::=<atom name>|<loop variable name>
                                 |/<class name>
                                 | PICK(<general class reference>)
```

This operand always evaluates to a single item. <loop variable name>s are described in section 23.1. A slash before a <class name> causes it to be taken as the atom of the same name (see 6.1). Group names are allowed here as all groups are also defined as atoms (see 9.2).

The PICK function is a built-in system function which randomly picks one element from its argument. If the argument has no elements (for example, the class is empty), then the PICK function returns the empty class as its value. The PICK function is extremely useful for inserting randomness into a program

```
2.  <specific class reference>::=<unsubscripted class name>
          |<subscripted class name>(<single valued atom field>)
```

This always results in the name of a specific class or specific class-subscript pair. This type of operand may appear in either the source or destination field of a class manipulation action. In the source field, the contents of the class is desired; in the destination field the class itself is acted upon by having its contents changed.

If a subscripted class is the destination field of an action, then the subscript can be already defined for this class, or not yet defined in which case it is now so defined. This permits dynamic definition of subscripted classes. If a subscripted class is in the source field of an action, then if the subscript does not exist for this class, the operand evaluates to null, or the empty class.

```
3.  <general class reference>::=<specific class reference>
                                |<subrule variable name>
          |<subscripted class name> (<general atom field>)
```

<subrule variable name>s are described in chapter 28.

When a <general class reference> includes one or more subscripted classes, quite complicated combinations are allowed. The subscript of a subscripted class may evaluate to more than one subscript name, in which case the union of all the class-subscript pairs is taken. Notice that the syntax allows subscripts to be subscripted classes, to any level, in which case a union is taken at each level, thus providing the set of subscripts for the next, outer level.

When a subscripted class is used as a destination field of a class action (see 14.), the set of subscripts is determined and then each subscript is used with the outermost class. In effect, the action is performed once for each subscript. If a subscript is not yet defined for the class it is now so defined.

4. <general atom field>::=<single valued atom field>
   |<general class reference>

This is the most general type of operand, and is used when a set of basic items is required.

11.3 Examples.

Examples of each type of operand will be given. In all cases, it is assumed that all atoms and classes have been previously defined, subscripted classes being defined as such. The names used will have the obvious definitions and meanings.

1. <single valued atom field>:
   JOHN   a node
   LOVE   a relation
   /PEOPLE   a class acting as an atom
   P   a loop variable
   PICK(PEOPLE)   one randomly chosen person
2. <specific class reference>: When used in a destination field, this always refers to the specified class, and not to its elements. When used in a source field, it refers to the items in the class, as described with the examples.
   PEOPLE   a list of all people
   FRIENDS(JOHN)   a list of all of John's friends
   WIFE(JOHN)   the wife of John
   WIVES   a list of all wives; a different class than WIFE
   SEXES(/PEOPLE)   the <u>node</u> PEOPLE acts as a subscript, and a list of people's sexes is given
3. <specific class reference>: may be any of (2) above, or
   FRIENDS(PEOPLE)   the friends of all the people; if a person has no friends he is ignored
   FRIENDS(PICK(PEOPLE))   the friends of a randomly chosen person
   WIFE(FRIENDS(JOHN))   the wives of all the friends of John; if one of John's friends is single, he is ignored
   WIFE(FRIENDS(PEOPLE))   the wives of all of the friends of all of the people
   WIFE(PICK(FRIENDS(PEOPLE)))   eventually, the wife of a randomly chosen friend of some person
   WIFE(FRIENDS(PICK(PEOPLE)))   different from the preceding example, since now you get the wives of the friends of a randomly chosen person

This can go on to any level, with the next example an indication of how complicated things can get:

FRIENDS(ENEMIES(PICK(WIFE(FRIENDS(PICK(PEOPLE))))))

In the above examples, the reference is assumed to be in a source field. If the reference were in a destination field, the action would be performed on the class-subscript pairs given by the outermost class and the calculated set of subscripts.

4. <general atom field>: can be any of the examples given in this section

2. Actions Affecting the Semantic Network.

The actions described in this section can cause changes in the emantic network. The semantic network can also be affected by ıse of the reality (RV) and hypothetical (H) values, which are described n chapter 19 and 20, and by predicate nodes (see 13.).

2.1 The Semantic Network and Triples.

The semantic network contains the assertions made about the simulated niverse. The network is composed of semantic triples, which epresent the semantic deep structure of the assertions.

Each semantic triple is an ordered arrangement of atoms, having he form: subject relation object. Subject and object are both nodes, ith the object always being optional.

Associated with each triple is information regarding:

. its times of creation and deletion, allowing you to determine the duration of a triple in the network, as well as the time elapsed since a triple has been negated (declared to be false). See 27. for ways to access these times.
. its numeric value (see 12.4)
. its secondary triples, if any (see 12.5)
. its reality value (see 19.)
. its hypotheticality value (see 20.)

12.2 Asserting Triples.

Semantic triples are asserted in the network simply by putting the triple in an action list. The action is of the form: <triple> where <triple>::=(<general node field><general relation field>[<general node field>]) The first field represents the subject, the second the relation, and the third, if present, the object. The fields are described in 11.2.

<triple> may also be (<triple variable name>) as described in 23.4, which is used when looping through triples on a predicate list.

The assertion of a triple is very powerful, as any of the items in the triple may be a <general atom field> (see 11.2), as long as the list resulting from the evaluation of the field contains only atom names of the proper type. If any item in the triple does evaluate to a list, all possible combinations of triples are added to the network. If any item in the triple evaluates to an empty list, then no triple is added to the network.

If a triple that already exists is added, it is not put in the network as each unique triple occurs only once in the network. However, the other items associated with the triple (time, RV, and H) are updated.

All triples added to the network or updated are printed in the change stack, along with the simulated time of the assertion, and value if appropriate. Any triple added may be followed with triples representing the expansions of any predicate nodes (see 13.2).

In the examples of triples given below, assume the following:
the class MEN = JOHN FRANK,
the class WOMEN = JANE MARY LINDA,
the relation class GOODRELATIONS = LIKE FRIENDOF,
the relation class HEALTHY = TALL THIN STRONG,
and the PROFESSORS class is empty.

```
$RULE: (MEN LIKE TOM);  puts (JOHN LIKE TOM) and (FRANK LIKE TOM)
  into the network.
$RULE: (MEN GOODRELATIONS TOM);  puts the above two triples
  plus (JOHN FRIENDOF TOM) and (FRANK FRIENDOF TOM) into the network.
```

```
$RULE: (MEN GOODRELATIONS WOMEN);  puts 12 triples into the network.
$RULE: (MEN HEALTH);  puts 6 triples into the network.
$RULE: (PROFESSORS GOODRELATIONS WOMEN);  puts no triples into the
  network, as one class is empty.
$RULE: (/MEN GOODRELATIONS /WOMEN);   puts only 2 triples into the
  network, as the subjects and object are class names acting as nodes.
```

The items in the triple may be more complicated, as demonstrated in 11.2, with the following triple action being legal:

```
$RULE: (FRIENDS(PICK(PEOPLE)) GOODRELATIONS WIFE(FRIENDS(JOHN)));
```

12.3 Negating Triples (*NEG)

Semantic triples may also be negated (marked as no longer being an assertion), using an action similar to the one for inserting triples. (Triples may be negated automatically by using exclusions, see 7.)

The syntax of this action is:

*NEG <triple>

where <triple> is described in 12.2.

Negating a triple does not physically remove it from the network. The triple is marked as having been negated but several functions (see 27.) can still reference the triple. If a negated triple is on a predicate list (see 13.), a loop through that predicate list (see 23.4) will include that triple, and if a comparison of triples on predicate lists is made (see 26.3.2) a negated triple will be found on its list. Negated triple are also treated as ordinary triples in all of the predicate node actions (see 13.)

This action is not the same as asserting, for example, that John is not fat. That assertion would have to be made by using something like JOHN NOTFAT or by using a secondary triple to modify FAT. Note that NOT is a reserved word and cannot be used as an atom name.

Again, the items are of the form <general atom field>, so using the example in 12.2 above, *NEG(MEN GOODRELATIONS TOM) negates four triples in the network, and four actions are printed in the change stack. If you try to negate a triple which is not in the network, that triple is ignored.

The change stack lists all negated triples, preceding the triple with the word 'NOT' to signify negation.

12.4 Setting and Modifying Numeric Values (*SET).

All triples have numeric values associated with them. These values may be used for any purposes, being especially useful in conjunction with numeric lexical expression lists (see 5.2). Whenever a triple is asserted it is given an initial value of zero.

The *SET action lets you set a triple's value or modify that value. If a triple whose value is to be set or changed is not currently asserted, it will be so asserted, in which case the assertion time and RV(see 19.) and H value (see 20.) of the triple will be set. If the triple is currently asserted, only its numeric value will change.

This action has the syntax:

*SET <triple> <assignop> <value>

<triple> specifies a triple or set of triples in the same way as a triple or set of triples is specified in an assertion (see 12.2). Each specified triple has its numeric value changed.

<value> is either a <triple> or a number. If it is a <triple> which specifies more than one triple, the sum of the values of the specified triples is used. If a specified triple is not in the network, it is skipped. The rules for determining whether a triple is in the network are the same as for subrule sentences, and are described in 26.1.1 and 26.1.2 (except here the C option would have to be in effect for the rule which contains the *SET action).

<assignop>::= =|=+|=-|=*|=/

= is the usual assignment operator and assigns the value to each of the specified triples. The others modify the values of the specified triples by performing the specified arithmetic operation on their original values with the given <value> and then assigning the results back into the original triples. For example,

<triple> =+ <value> really means <triple> = <triple> + <value>

for each specified triple. <assignop>s which have two symbols can have no space between the symbols.

The triples whose values are being changed are printed in the change stack.

The examples assume the same atoms and classes as the examples in 12.2.

```
$RULE:  *SET(MEN LIKE TOM) = 4;
$RULE:  *SET(MEN GOODRELATIONS MARY) = 12.35;
$RULE:  *SET(MEN HEALTHY) = -9.3;
$RULE:  *SET(JOHN LIKE MARY) =+ 8.2; adds 8.2 to the value of the triple
$RULE:  *SET(JOHN LIKE WOMEN) =* 2; for each of the three specified
           triples, multiples its value by 2
$RULE:  *SET(JOHN LIKE MARY) = (JOHN LIKE JANE);
$RULE:  *SET(JOHN LIKE MEN) =+ (JOHN LIKE WOMEN);  the value of (JOHN
        LIKE WOMEN) is calculated by adding the values of the triples
        it specifies; this value is then added to each of the triples
        specified by (JOHN LIKE MEN)
```

12.5 Secondary Triples.

The semantic triples discussed so far allow only subject, relation, and object. In order to modify any or all of these items, secondary triples are introduced. If it is not clear from context, the triples previously disucssed will be called primary triples in order to distinguish them from secondary triples. Otherwise, primary triples will simply be called triples, with secondary triples always being given their full title.

As secondary triples are used only to modify primary triples, they are transparent to the semantic network and are accessible only through their associated primary triples. This means that, in a test, a secondary triple will not be found unless the primary triple is also given (see 27.4).

Secondary triples look like primary triples, except for the added ability of having either a node or a relation in the subject position. Thus, secondary triples are <triple>s (see 12.2), with the first field being any type of atom--a node or any type of relation.

Secondary triples also have the same extra information associated with them as have primary triples (see 12.1).

12.5.1 Assertion (*SEC).

A secondary triple is added as a modifier of a primary triple by using the *SEC action, whose syntax is:

*SEC <primary triple> {<secondary triple>}$^+$

Notice that this allows any number of secondary triples to be added as modifiers of the one primary triple.

<primary triple> is a <triple> of the form given in 12.2. If the primary triple is already in the network, then its associated items are updated. If it is not in the network, it is put in. In fact, all rules are as in 12.2 for addition of just a primary triple.

<secondary triple> may appear any number of time, as a primary triple may be modified by any number of secondaries. A secondary triple is a <triple> of the form given in 12.2, except that the subject may be any type of atom.

If the secondary for that primary is already in the network, then the secondary's associated items are updated. If it is not already pointed to by the primary, then it is put in the network, but will only be accessable through the primary.

As with assertions of primary triples, all of the items in any triple in *SEC are of the form <general atom field>. If any item does evaluate to a list of atoms (of the proper type), then combinations of primary and secondary triples are asserted in the following manner: each <secondary triple> is paired with the <primary triple>, and ALL combinations of items for each pair of triples are asserted; the secondary triples interact only with the primary, and not with each other. If any item yields an empty list, the triple containing that item will be ignored--if it is the primary triple, no action at all will occur.

This procedure does not always yield the desired result, as demonstrated in the examples below. This problem can be solved by using loops (see 23.).

The change stack lists all asserted or updated primary-secondary triple pairs. This is done by listing, for each primary (there may be only one), the primary tiple followed by all of its secondary triples. If any of the nodes are predicate nodes, things get more complicated (see 13.2).

For example, assume the atoms used below have the obvious meanings, and these classes have been defined:

```
MEN = JOHN FRANK;
WOMEN = JANE MARY;
GOODRELATIONS = LIKE FRIENDOF;
$RULE:  *SEC(JOHN LOVE MARY)(LOVE WITH PASSION):  adds one primary
        and one secondary triple
$RULE:  *SEC(JOHN LOVE MARY)(LOVE WITH PASSION)(PASSION
        GREAT);  adds one primary and two secondaries
$RULE:  *SEC(JOHN LOVE MARY)(JOHN HANDSOME)(MARY PRETTY);
$RULE:  *SEC(MEN MEET JOHN)(MEET AT CLUB);
$RULE:  *SEC(JOHN AFFECTION MARY)(MARY PRETTY);
$RULE:  *SEC(MEN LOVE MARY)(MARY PRETTY);  puts 2 pairs of primary
        and secondary triples into the network.
$RULE:  *SEC(JOHN LOVE WOMEN)(WOMEN PRETTY);  puts 4 pairs into the
        network, two of which, (JOHN LOVE JANE)(MARY PRETTY) and
        (JOHN LOVE MARY)(JANE PRETTY) are probably not desired.
        This can be solved by looping (see 23.1) through the women.
$RULE:  *SEC(JOHN LOVE/WOMEN)(/WOMEN PRETTY); adds only one primary-
        secondary pair, using the node WOMEN
$RULE:  *SEC(MEN LOVE WOMEN)(WOMEN PRETTY);  adds 8 pairs, not all
        of them useful
$RULE:  *SEC(JOHN GOODRELATIONS MARY)(MARY PRETTY) (JOHN HANDSOME);
        adds two secondaries for each of two primaries
$RULE:  *SEC(MEN LOVE MARY)(MARY PRETTY)(LOVE WITH PASSION) (PASSION
        GREAT)(MEN HANDSOME); adds five secondaries for each of two
        primaries, but the (MEN HANDSOME) triple causes problems by
        mismatching MEN in two cases, a problem solvable by using loops
        (see 23.)  Also, the order of the secondaries might have to
        be changed, according to the grammar in use, if natural
        language output is desired.
```

12.5.2 Negation (*NEGSEC).

Secondary triples are negated using the *NEGSEC command, which is the secondary triple version of the *NEG command.

The *NEGSEC command is of the form:

*NEGSEC <primary triple> {<secondary triple>}$^+$

<primary triple> is a <triple> (see 12.2). If a specified triple does not exist, then the triple is skipped. If it does exist, it is left unchanged in the network.

<secondary triple> is like <primary triple> above, except that the subject field may be any type of atom. If this secondary exists for the given primary, then the secondary is negated in the network. If it doesn't exist for the given primary, nothing happens.

Each of the items in the triples is of the form <general atom field>, with the rules for evaluation being the same as those for *SEC (see 12.5.1), causing primary-secondary pairs to be generated. The change stack lists negated triples in the same way as in an *SEC, but all listed triples are preceded by the word 'NOT'. (Note that the primary triple will be preceded by 'NOT' but will never be negated or affected in any way.)

If a primary triple is negated by a *NEG command, all of its secondaries are marked as negated, as, of course, is the primary. A reassertion of the primary does not reassert the secondaries.

Negated secondary triples can still be referenced by several functions; see chapter 27. for details.

For examples of this command, see examples of the *SEC command in the previous section. With both commands the same triples are specified, but with *NEGSEC the primary is never affected, and if a triple is not currently asserted in the network it will be skipped.

12.5.3 Setting and Modifying Numeric Values (*SETSEC).

All secondary triples have numeric values associated with them, in the same way that primary triples have numeric values. The value of a secondary triple can be used for any purpose, being especially useful in conjunction with numeric lexical expression lists (see 5.2). Whenever a secondary is asserted, it is given an initial value of zero.

*SETSEC is the secondary counterpart of *SET and uses the same rules, except that both a primary and a secondary field must be given. Each secondary will have its value set or modified. If a specified primary or secondary is not currently asserted, it will be so asserted, and it will have its associated items (time, RV, and H value) set. The numeric value of a primary is not changed. A currently asserted primary is completely unaffected; a currently asserted secondary will have only a numeric change.

Syntax:

*SETSEC <primary triple><secondary triple><assignop><value>

<primary triple> and <secondary triple> are as in *SEC (see 12.5.1), and for each primary-secondary combination the secondary has its value changed.

<assignop> and <value> are described in the *SET action (see 12.4).

Examples:

```
$RULE:  *SETSEC(JOHN LOVE MARY)(MARY DESCRIPTION) =+ 2.3,
        *SETSEC(MEN LOVE MARY)(MEN AGE) = 25,
        *SETSEC(/MEN IQ)(IQ TOTAL) = (MEN IQ), this sets the
           value of a single secondary triple to the sum of the men's
           IQs.
        *SETSEC(MEN GOODRELATIONS WOMEN)(WOMEN BEAUTY) =* (SAVED BEAUTY);
           this last action may assert and/or give values to some
           undesirable triples, as described in similar examples in 12.5.1
```

13. Actions Affecting Complex Predicate Nodes.

A number of actions are available for manipulating complex predicate nodes (also called predicate nodes). These actions will be described after predicate nodes are explained.

13.1 Complex Predicate Nodes.

Nodes (but not relations) may point to other triples in the network. In this case, the nodes are said to be complex predicate nodes or, simply, predicate nodes. A predicate node may point to any number of triples, but all triples must be primary triples, although they in turn may point to secondary triples. A predicate node acts as a list of triples, with duplicate triples not being checked for.

As the triples on a predicate list are primary triples, they exist only once in the network. Therefore, if a triple is accessed independently of the predicate node mechanism (for example, a triple is simply asserted or negated, or used in a *SET action), any changes made to the triple will be made to the only instance of that triple; if that triple is on a predicate list, the changes will be seen on the list.

However, it is important to note that a negated triple will still be on the predicate list, but will be marked as being negated. Therefore, when a predicate node is specified in an action described below, a loop (see 23.4), or a subrule (see 26.3.2), any negated triples on the predicate list will be used like any other triples on the list.

One possible function of a predicate node is as a discourse variable. For example, the node THATJ may point to triples which describe 'what John knows' so that the triple (JOHN KNOWS THATJ) will refer to all that John knows.

There are commands which manipulate the triples of a predicate node, as well as commands which search for triples having a specified node (see 13.5) or search for predicate nodes having specific triples on their predicate lists (see 13.6). The triples on a predicate node's list can be looped through one at a time (see 23.4) and there are tests for comparing triples (see 26.3.2).

13.2 Predicate Nodes in the Change Stack.

The triples attached to a predicate node are printed on the change stack whenever a triple containing that node is printed, except when the triple is printed due to a negation. Things can get complicated, since in an *SEC command (see 12.5.1), for example, both primary and secondary triples may contain predicate nodes. Furthermore, the triples of a predicate node may point to secondary triples. Finally, a triple of a predicate node may also contain predicate nodes. (Actually, a secondary of a triple of a predicate node may also contain predicate nodes, but that is too much to deal with.)

Before detailing the method of expanding triples on the change stack, secondary triples of predicate node triples will be explained. A predicate node only points to primary triples, but these triples act as any other primary triples and so may point to secondary triples. A problem arises, however, when a secondary triple of a primary triple is automatically printed out, as is the case here, since the primary triple may have many secondaries, not all asserted at the same time. Therefore, when a predicate node triple is printed, it will be followed only by those secondary triples which have the same assertion time as the primary. As primary triples have their assertion times updated when a new secondary is added to them, this rule effectively means that only those secondaries last added to the primary (added at the same time, with no updating of the primary occurring at a later time) will be printed.

So, when a triple is printed on the change stack, except when it has been negated, it is followed by:

1. If the subject is a predicate node, get its first predicate triple. Otherwise, go to step (5).
2. Print the triple.
3. Print, in reverse order, its secondary triples having the same insertion time as the triple itself (see above discussion).

4. Expand this triple by going to step (1). Note that secondary triples of predicate node triples are not expanded.
5. If here, the subject is finished, so check the object. If the object is a predicate node it is expanded in exactly the same way as the subject node. Otherwise, continue.
6. A triple's predicate nodes (if any) have been completely expanded. If this triple is on a predicate node list, then get the next triple on the list and go to step (2).
7. Eventually, the predicates nodes (if any) of the originally printed triple will have been completely expanded. The process is then complete, unless the original triple was a primary triple of a *SEC or *SETSEC command. In that case, each of its secondary triples is treated exactly as the primary was, a secondary being printed and then expanded.

While this procedure is long and complicated, in general triples won't have secondary triples or predicate nodes, or perhaps just a few of them. And preciate node triples won't themselves have predicate nodes. A sample change stack is given in chapter 31.

Finally, it is possible to have a predicate node contain a triple which contains that same predicate node. For example, if THAT is a predicate node, it may contain the triple (JOHN KNOWS THAT), representing the concept "John knows that John knows that ...." To avoid an endless loop, if a predicate node is the same as one already being expanded, a warning is printed and the node skipped.

13.3 Inserting Triples.

Any primary triple may be added to the list of triples of a predicate node. Any node may be a predicate node, and becomes one when a triple is added to its predicate list. A triple so added acts like any other primary triple in the network, and may be modified by secondary triples. It may also contain predicate nodes, even the predicate node whose list it is added to.

If a triple added to a predicate node is already in the network, the triple is left unchanged. If the triple is not in the network, it is now asserted with the important side effect that its hypothetically (H) value (see 20.) is set to one.

Under no circumstances is any information about predicate node actions printed in the change stack, even if a new triple needs to be asserted in the network.

Several of the predicate node actions involve the use of a new kind of operand in the source field. <predicate source field> is defined as follows:

```
<predicate source field>::=|<triple>|<general node field>
```

<triple> is defined in 12.2. If the <triple> is not already in the network, it is added with an H value of 1, but no output appears on the change stack.

<general node field> is as described in 11.2. For each of the nodes, if it is not a predicate node (it currently has no predicate list) it is ignored; if it is a predicate node, its triples are used as the value of the field.

13.3.1 *ADDP.

This command adds the specified triples to the predicate lists of the destination nodes. Its syntax is:

```
*ADDP <predicate source field> TO <general node field>
```

<predicate source field> is described in 13.3. It expands to some number of primary triples.

<general node field> is described in 11.2.

All of the source triples are added to the destination predicate node lists. No output is shown on the change stack. A source triple not already in the network (possible only for a source triple of the form <triple>) is entered with a hypotheticality (H) value of 1.
A source triple already in the network is left unchanged.

Examples:

```
*ADDP (JOHN LOVE MARY) TO THAT2
*ADDP (HERO WORK) TO HEROTASK
*ADDP HEROTASK TO TASKS
*ADDP (JOHN LIKE FRIENDS(MARY)) TO THAT(FRIENDS(JOHN))
```

which might be used to indicate which facts about John his friends are aware of.

13.3.2 *MOVEP.

This action is identical to *ADDP except that the predicate lists of the destination nodes are erased before the source triples are added to them. The syntax of the *MOVEP action is:

*MOVEP <predicate source field> TO <general node field>

with the fields being the same as in *ADDP.

13.3.3 *PUSHP.

This action is the same as *ADDP except that the predicate lists of the destination nodes are treated as stacks. The source triples, in the order they are listed (in the case of a predicate node, this is the order of the triples on its predicate list, which may be meaningless if that list is not being used as a stack), are added one by one to the top of the predicate stack for each destination predicate node. This allows user ordering of the predicate list. A predicate list should not be used as both a list and a stack at the same time since the stack ordering may be destroyed.

The syntax of the *PUSHP action is like that of the *ADDP action:

*PUSHP <predicate source field> TO <general node field>

13.4 Deleting Triples.

Triples can be deleted from the predicate triple lists of predicate nodes. The actions used to accomplish this are similar to the actions for inserting predicate triples. A triple deleted from a predicate list remains otherwise unchanged in the network.

13.4.1 *REMOVEP.

This action removes all occurrences of the source triples from the destination predicate nodes. If a source triple is not on the predicate list of a destination node, no action is taken for that case.

The syntax of this action is:

*REMOVEP <predicate source field> FROM <general node field>

<predicate source field> is described in 13.3.

<general node field> is described in 11.2.

13.4.2 *ERASEP.

This action is of the form:

*ERASEP <general node field>

and simply erases the triples from the predicate node lists of the specified nodes.

13.4.3 *POPP.

If a predicate node's triple list is to be used as a stack, the *PUSHP action should be used to place a triple on the stack (see 13.3.3.) and the *POPP action to remove the top element from the stack and place it in the list of another predicate node for further reference.

The syntax of the *POPP action is:

*POPP <general node field> TO <general node field>

Each source predicate node has the top element of its predicate stack popped off, and all of the triple thus generated are <u>moved</u> to the predicate lists of each of the destination predicate nodes, these destination predicate lists first having been erased. Note that the source field can only include nodes and not triple variables or triples, as only nodes can have predicate stacks attached to them. If a source node is not a predicate node (it currently has no predicate list) it is ignored.

13.5 Searching For Triples (*LOCATE).

This action causes a search of the network to take place. It is sometimes useful to know where a node has been used, and this action finds all triples in which specified nodes have been used as either subject or object.

The syntax of the *LOCATE action is:

*LOCATE <general node field> TO <general node field>

where <general node field> is described in 11.2.

For each source node, the network is searched for triples which have that node in them. Only primary triples are looked at, but all primaries are checked, regardless of their RV or H values. All triples which contain any source node in any position are then added to the predicate lists of each of the destination nodes.

For example, *LOCATE FRIENDS(JOHN) TO /FRIENDS will find all triples referencing a friend of John and place all of these triples on the predicate list of the node FRIENDS.

13.6 Searching For Predicate Nodes (*MOVEPAR).

This action causes a search for predicate nodes, finding those nodes whose predicate lists include any of the specified triples. Its syntax is:

*MOVEPAR <predicate source field> TO <general class reference>

<predicate source field> is described in 13.3.

<general class reference> is described in 11.2 and must specify node classes.

First, the source field generates any number of triples. Then, each and every node defined in the program is checked, and if a node is found whose predicate list contains one or more of the source triples, that node is added to the destination class(es). Note that if a predicate node is used in the source field in order to use its triples, that node will be added to the destination class(es) since it obviously meets the search criteria. The RV and H values of all triples are ignored during the search.

For example, *MOVEPAR (MEN LOVE MARY) TO CLASS2 would find all nodes whose predicate lists contain one or more triples concerning a man loving Mary, and add those nodes to CLASS2.

13.7 Extracting Atoms From Triples (*EXPAND).

This action allows you to break a triple into its component pieces. The atoms in each specified triple are added to the appropriate destination classes.

The *EXPAND action is of the form:

```
*EXPAND <predicate source field> TO <general class reference>
                        <general class reference> [<general class reference>]
```

<predicate source field> is described in 13.3.

<general class reference> is described in 11.2. The first and third classes must be classes of nodes, and the second a class of relations.

The subjects of the source triple are added to the class(es) of the first field and the object to the class(es) of the third field. If a triple has an object but no object field is specified, that object is ignored. If a triple has no object, an object field will remain unchanged for that triple. The relations of the source triples will be added to the second destination field.

Notice that if the <predicate source field> specifies more than one triple, the destination classes get multiple atoms added to them. However, as the atoms in these classes are unordered, you can't determine which atoms were from which triples. It is therefore probably more useful in most circumstances to expand only a single triple (probably a triple variable), although there might be cases where you would want, for example, a class of subjects appearing in the predicates of a complex predicate node.

Examples (assume X is a triple varible):

```
*EXPAND (X) TO SUBJS RELS OBJS
*EXPAND JOHN TO PSUBS(JOHN) PRELS(JOHN)
*EXPAND PICK(PEOPLE) TO CLASS1 CLASS2 CLASS3
```

14. Actions Affecting Classes.

Elements can be added to and removed from classes, and subscripted classes can have additional subscripts defined. Atoms may be extracted from triples and put into classes (see 13.7) and predicate nodes having specified triples can be found and put into classes (see 13.6).

Duplicate class elements are ignored and elements are stored in an internal order. All atoms in a class must be of the same type as the class. See chapter 6 for more information about classes.

In all of the class actions, the destination field is a <general class reference>, which is described in 11.2, and thus may specify a class, or one or more class-subscript pairs. If this is a subscripted class, a new subscript for this class may be formed.

Classes do not have times, RVs, or H values associated with them. Class actions never appear on the change stack.

14.1 Adding Elements.

Two similar actions allow elements to be added to a class. In both cases, source atoms must be of the same type as the destination field.

14.1.1 *ADD.

Syntax: *ADD <general atom field> TO <general class reference>

All members of the source field are added to the current members of the destination class(es).

Examples:

```
*ADD JOHN TO MEN
*ADD FRIENDS(JOHN) TO FRIENDS(MARY)
*ADD FRIENDS(PICK(MEN)) TO TEMP
*ADD JOHN TO FRIENDS(MEN) makes John a friend of each man
*ADD JOHN TO FRIENDS(WIFE(MEN)) make John a friend of each man's wife
```

14.1.2 *MOVE.

Syntax: *MOVE <general atom field> TO <general class reference>

The destination class(es) is (are) first erased, the source atoms then being added to the destination class(es). This is equivalent to erasing the destination class(es) (see 14.2.2) and then adding the source elements to it (them).

Examples:

```
*MOVE FRIENDS(JOHN) TO FRIENDS(MARY)
*MOVE PICK(PEOPLE) TO ENEMIES(PICK(MEN))
```

14.2 Removing Elements.

Specific elements can be removed from a class or a class can simply be erased.

14.2.1 *REMOVE.

Syntax: *REMOVE <general atom field> FROM <general class reference>

All members of the source field are removed from the destination class(es). If a member of the source field is not in a destination class, no action is taken regarding that member with that class. The source field members must be of the same type as the destination field.

14.2.2 *ERASE.

Syntax: *ERASE <general class reference>

The specified class(es) is (are) erased, leaving it (them) with no member elements.

15. Actions Affecting Lexical Lists.

Lexical triples can be added to the lexical lists of atoms by using the two lexical actions. Lexical lists and lexical triples are described in sections 4.2 and 4.2.1. While the lexical actions cannot directly add dictionary entries to the lexical list of an atom, triples of any form may be so added. For this reason, it is strongly suggested that lexical triples be added to atoms by using the lexical actions, rather than the lexical expression lists of atoms. Usually, these lexical actions will take place in the network section (see 8.) of a program.

Any atom, <u>except</u> a relation with a numeric lexical expression list (see 5.2) may have a lexical list added which gives 'meaning' to that atom. As group names and class names are also atoms, class names being atoms when preceded by a special marker (see 6.1), they may have lexical lists added unless they are of the above excepted type.

While dictionary entries cannot be directly added to a lexical list, lexical triples can be added, effectively adding the lexical lists (including dictionary entries) of the atoms of the lexical triples. Thus, the lexical actions allow you to directly add only triples of semantic entities (atoms) to a lexical list, but the surface language meanings associated with those semantic entities, as defined in the atom lists (or since implicitly added by a lexical action), are used in natural language output.

It is important to note that lexical lists of any form are completely disregarded if the generative mechanism is not being used, in which case only a change stack, using internal atom names, will appear.

15.1 Adding Lexical Triples.

Two actions exist for adding to an atom's lexical list, but at present there is no way to remove any atoms from a lexcial list--both actions leave the original list intact. The two actions differ in that the first adds specified triples while the second adds the lexical triples of other atoms.

15.1.1 *LEXTRP.

Syntax: *LEXTRP <arbitrary lexical triple> TO <general atom field>

<arbitrary lexical triple> is a <triple> (see 12.2) composed of one, two, or three <general atom field>s (see 11.2), with no restrictions on the type of each atom. This allows much more freedom than given in lexical expression lists. Even if a triple has only one item, this action is still very different from *LEXADD, as will be described below.

<general atom field> is described in 11.2 Relations with numeric lexical expression lists are not allowed.

All of the source triples are added to the lexical lists of each of the destination atoms.

Examples:

```
*LEXTRP (CALL UP) TO CALLUP
*LEXTRP (JOHN FAT) TO JOHN
*LEXTRP (EVIL LOOKING CREATURE) TO VILLAINS
*LEXTRP (MARY ADJECTIVES(GOOD)) TO MARY
```

15.1.2 *LEXADD.

Syntax: *LEXADD <general atom field> TO <general atom field>

<general atom field> is described in 11.2.

For each source atom, its lexical triples, if any, are added to the lexical lists of each of the destination atoms. If a source atom has a dictionary entry on its lexical list, that entry is not added to the lists of the destination atoms.

For example, assume that the $RELATIONS list includes:
GOODLOOKS 0 0 = 'TALL' 'HANDSOME';
and the action *LEXTRP (GOOD LOOKING) TO GOODLOOKS
has been executed. Then, if *LEXADD GOODLOOKS TO ADJECTIVES(GOOD) is executed, the lexical lists of the atoms representing a certain class of adjectives will have the triple (GOOD LOOKING) added to them, but the dictionary entries of GOODLOOKS('TALL' and 'HANDSOME') will not be so added.

A problem could arise if a source atom is the same as a destination atom, as this could cause an infinite loop to occur. Therefore, in this case the source atom is skipped.

Finally, there is an important difference between *LEXTRP of a triple having a single atom and *LEXADD of that same atom. In the *LEXTRP case, the lexical triple consisting of that atom will be added to the destination atoms. In the *LEXADD case, the lexical triples on the lexical list of that atom will be added to the destination atoms.

16. Actions Affecting the Scheduling of Groups.

In order for a group to be executed, it must be enabled and scheduled for execution. Once a group is enabled, it will be executed repeatedly according to the time interval specified with the group definition. In order to deactivate a group, it may be disabled. A group may be enabled and disabled any number of times. More details are given in section 9.1.

16.1 Scheduling Execution (*ENABLE).

Syntax: *ENABLE <general atom field> [IN <time>]

<general atom field> is described in 11.2. The field must be already defined, so if it is a single group name, that name must be an atom name (see 9.2). If it is an atom name, that name must eventually be defined as a group, else a compile-time error will be given. During execution of the program, if a source element is not the name of a group, that element will be skipped.

<time> is described in 3.1.1.

This action puts the specified groups on the queue of scheduled groups and marks the groups as enabled. If no <time> is given, then each group is scheduled for the current time (in the order in which the elements are stored in the field). If a <time> is given, then each group is scheduled for the current time +<time>(or possibly later for a synchronous group). If other groups are scheduled for the time at which a new group is now scheduled, the new group will be executed after the original ones. If a newly enabled group is already scheduled, only its earlier scheduled time (i.e., the time which will cause the soonest execution of the group) will be kept, as only one schedule entry per group is maintained on the queue. Enabling a group does not mean that it is immediately executed; in order to immediately execute a group, call it as a subroutine (see 17.).

Examples:

```
*ENABLE MAIN IN 2D4M
*ENABLE GROUPS
```

16.2 Cancelling Execution (*DISABLE).

Syntax: *DISABLE <general atom field>

<general atom field> is described in 11.2 and has the same additional explanation as in *ENABLE (see 16.1).

Each specified group is disabled by marking the group as being disabled and immediately removing the group from the queue of scheduled groups. If a group is already disabled, nothing happens to it. Note that no <time> specification is allowed.

A group may include an action which would disable itself, which is useful if the group is to be executed only once for each time it is enabled.

Examples:

```
*DISABLE MAIN
*DISABLE GROUPS
```

17. Actions Affecting Subroutine.

Two actions allow groups to be used as subroutines. These actions will be described after the concept of subroutines is introducted.

17.1 Subroutines.

Any group may be called from any other group (including itself) as a subroutine, in which case the called group is immediately executed, but not enabled. After execution of the called group, the calling group is resumed.

As in other programming languages, this is convenient if there are rules common to several groups; these rules may be put into one group and called from the groups needing those rules. Also, subroutines can be used to split a group into separate, smaller groups according to function, for a more readable program.

Unlike most other programming languages, no parameter passing is provided for. The called group has access to the entire data base, and if the calling group wants to pass information to the called group then it must first put this information into the data base, probably using a temporary class which it could examine and erase upon return of the called group.

A called group may itself call a group, and so on to any level. Each called group returns, upon completion, to the group which called it. A group may call itself recursively, also to any level. Care should be taken to avoid an endless circle of subroutine calls.

A group called as a subroutine need not terminate only at its $ENDGROUP statement but may terminate anywhere in an action list.

17.2 Calling Subroutines (*CALL).

Syntax: *CALL <general atom field>

<general atom field> is described in 11.2, and has the same additional explanation as in *ENABLE (see 16.1).

The action calls each of the groups in the specified field. The groups are executed one at a time, in the order in which they are stored in the field. When a called group returns, the next group in the field is called. When the last called group returns, the calling group continues at the action (or statement) following the *CALL action.

A *CALL action is not allowed in a subrule action list. A *CALL action does not affect the status of a group in any way; the group remains enabled or disabled as before the call, and the group, if scheduled, remains scheduled for the same time.

Examples:

```
*CALL GROUP6
*CALL GROUPSEQUENCE
```

17.3 Returning From Subroutines (*RETURN).

Syntax: *RETURN

When a group is called as a subroutine, it may return to the calling group before it reaches its $ENDGROUP statement (see 9.2). *RETURN may appear anywhere in an action list. If it is executed in a group that is not acting as a subroutine (the group has been scheduled, not called), then the action will be ignored. However, if the group has been called as a subroutine, then when *RETURN is executed the called group will immediately terminate (the rest of the action list will be ignored) and control will return to the calling group. If a called group executes no *RETURN action, then it will terminate at its $ENDGROUP statement. There may be any number of *RETURN actions in a group.

18. Action For Terminating the Simulation (*END).

Normally, the simulation is terminated when either no further groups are scheduled, or the simulated time exceeds the specified end time (see 3.1). However, the *END action may be used, in which case the simulation will terminate immediately after all groups scheduled for the current time are executed. This effectively sets the end time of the simulation to the time at which the *END action is executed. The syntax of this action is simply: *END

19. Actions Affecting the RV.

A reality value (RV) is attached to every triple asserted in the network (whether primary or secondary) including those asserted automatically in a predicate triple action (see 13.3) or a numeric modification (see 12.4 and 12.5.3). This value is printed on the change stack.

The RV might be useful in giving a truth value to a triple, or in setting up separate universes for different contexts. It has no inherent meaning, and can be used for any purpose or simply ignored. The RV of a triple can be tested in subrules (see 27.8).

RV is a global variable which is initially zero. The RV can be changed at any time by the programmer, but is never automatically changed. The RV attached to triple is the current RV, only one current RV being allowed. RVs can be saved as values of triples and then reset, however. The RV is a positive integer with a maximum value of 4095; if a real value is assigned to the RV, the value will be truncated.

19.1 Setting the RV (*SET).

The RV is set or modified in the same way that a primary triple has its numeric value set or modified, using the *SET action. The syntax here is:

```
*SET  RV <assignop><value>
```

where RV is a reserved word and the other fields are described in 12.4 and have the same meanings here.

Examples:

```
*SET RV = 5
*SET RV =+ 2.3
*SET RV =* (RV OLD)
```

20. Actions Affecting the Hypotheticality (H) Value.

A hypotheticality (H) value is attached to <u>every</u> triple asserted in the network (whether primary or secondary) including those asserted automatically in a predicate triple action (see 13.3) or a numeric modification (see 12.4 and 12.5.3). This value is printed on the change stack with the triple.

Unlike the RV, the H value does have a meaning to the system. The H value of a triple can either be 0, meaning the triple is always found in the network, or 1, meaning the triple is hypothetical. In a subrule test, a triple having an H value of 1 will not be found to be in the network unless the test specifically allows the triple to be hypothetical (see 26.1). The H value is ignored, however, for negations of triples and in all constructs dealing with predicate triples (see 13.,26.3.2, and 23.4).

<u>Every</u> triple asserted in the network is given an H value of 0 except in two cases:

1. if a 1 is placed in front of the subject element of the triple (or in front of a triple variable), as in:

```
    $RULE:  (1 JOHN LOVE MARY),
            *SEC(1 MEN LOVE SUE)(SUE PRETTY),
            *SEC(JOHN LOVE MARY)(1 MARY PRETTY),
            *SEC(1 JOHN LIKE FRIENDS(MARY))(1 JOHN HANDSOME),
            *SET(JOHN IQ) = 125, where if this triple needs to
                be asserted, it will have an H value of 0
            *SET(1 WOMEN IQ) = 130, where if any triple needs to
                be asserted, it will have an H value of 1
            *SETSEC(1 MEN LOVE MARY)(1 MARY BEAUTY) =+ 2, where
                any triple which needs to be asserted will have
                an H value of 1
            (1 X);  where X is a triple variable which is being
                asserted
```

2. if the triple is not in the network and is asserted due to its being specified in a predicate action which puts the triple on a predicate list (see 13.3).

21. Actions Affecting the Random Number Generators (*JUMBLE).

The random number generators (see 3.1.2) can be reseeded during the execution of the program. While this action will not yield different results for different executions of the program (unless you actually change the seed value and recompile, of course), it may be useful if, for example, you want a second execution of a program to produce the same results only up to a certain point.

The syntax of this action is:

```
*JUMBLE [one or both of P and S, in either order] = <number>
```

If the field following *JUMBLE contains a P, then the random number generator used for the PICK function (see 11.2) is reseeded; if it contains an S, then the random number generator used for subrules (see 24.) is reseeded; if it contains both letters (with no internal blanks) or is absent, both random number generators are reseeded. <number> is a constant, and it used to reseed the random number generators.

Examples:

```
*JUMBLE P = 17
*JUMBLE SP = 22
*JUMBLE = 4793
```

22. Switches.

A switch statement is the same as a rule statement (see 10.) except that a switch may not have an action list. A switch statement can have branching specifications and a subrule list, and so is used only for branching purposes. A rule statement with no action list may be used in exactly the same manner as a switch, but a switch statement makes the branching function clearer. Its syntax is:

```
$SWITCH [,<option field>] [<switch name>] : [<branch part>] ; [<subrule list>]
```

All fields in the statement are the same as in the rule statement (see 10.2). A switch statement without a subrule list is actually an unconditional branch command.

Examples:

```
$SWITCH,C  SW1: T(RULE3);
10,-10:         (JOHN LOVE MARY);
$SWITCH:        T(RULE2)F(LOOP4);
.7,.3:          (JOHN LOVE MARY);
0,-.2:          (JOHN LOVE TOM);
```

23. Loops

Loops allow you to examine the contents of a set of items by producing the elements of the set one at a time. A loop variable is used to represent the current element. Loops may be used for both classes and the predicate lists of complex predicate nodes. A loop is delimited by $LOOP and $ENDLOOP statements, with any number of rule, switch, and other loop statements allowed in this range.

23.1 Loop Variables.

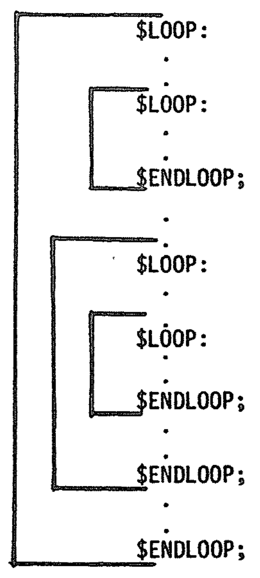
A loop variable is defined for each loop. For each pass through the loop, this variable is set to the next element in the set being examined. A loop varible name is considered to be a <single valued atom field> if its associated set is a class, as the loop variable then points to a single atom at any given time (see 11.2). The atomic type of the variable is the same as the type of its class. Loop variables may also be associated with semantic triples when looping through the predicate lists of complex predicate nodes (see 13.1), in which case they are considered to be <triple variable>s.

A loop variable is known only within the range of its defining loop. A loop variable name may be used again, but only as a loop variable name, and only if the previous use of the name is no longer in effect.

A loop variable is given an initial value at the beginning of the loop, when the $LOOP statement defining it is executed. The variable gets a new value each time the $ENDLOOP corresponding to its $LOOP statement is executed. If there is no new value for the variable to get, the loop if finished and control goes to the statement following the $ENDLOOP.

23.2 Nesting of Loops.

A loop may contain any number of loops inside it. Each loop must have a $LOOP and $ENDLOOP statement, and an inner loop must be completely within its containing loop. For example, if a loop contains one other loop, the first (inner) $ENDLOOP is associated with the inner (more recent) $LOOP, and the second $ENDLOOP with the outer $LOOP. An example of multiple nesting is shown below:



```
$LOOP:
  .
$LOOP:
  .
  .
$ENDLOOP;
  .
  .
$LOOP:
  .
$LOOP:
  .
  .
$ENDLOOP;
  .
  .
$ENDLOOP;
  .
  .
$ENDLOOP;
```

There are also rules regarding the transfering of control from and to statements inside of loops. These rules are identical to those found in most programming languages.

A branch to a statement inside the range of a loop, which includes its $ENDLOOP statement but not its $LOOP statement, is <u>not</u> allowed from

a statement outside of the range of that loop. However, a statement inside the range of a loop can freely transfer control to a statement outside of the range of that loop, although the loop variable will then no longer be defined.

23.3 $LOOP for atom lists.

In this form of a loop, a class of atoms is specified, with the loop variable representing one atom at a time. The general form of this statement is:

```
$LOOP [,<option field>] [<loop name>]  :  <loop variable name>  .
                                          <general class reference>;
```

<option field> is as for rules (see 10.2). These options are in effect for all subrules within the range of the loop, unless overridden by an option field within the loop.

<loop name> is necessary if you want to transfer to this loop from another place in the group (see 2.2).

<loop variable name> is described in 23.1.

<general class reference> is described in 11.2. The elements in this field are determined at entry to the loop (when the $LOOP statement is executed) and saved. This means that if the value of this field is changed by actions inside the loop, there will be no effect on the order or number of loop passes made. If the <general class reference> evaluates to the null set, the loop is skipped, with control transfering to the statement following the $ENDLOOP statement. See 6. for details on the order of elements in classes.

Examples:

$LOOP LOOPNM1: X.PEOPLE;
$LOOP: YLOOP . FRIENDS(ENEMIES(PICK(PEOPLE)));
$LOOP: P . PICK(PEOPLE); Note that this last example assigns one randomly chosen element of the class PEOPLE to the loop variable P; the same effect could be accomplished without using a loop, using instead an unconditionally executed rule of the form:
$RULE: *MOVE PICK(PEOPLE) TO Q;

23.4 $LOOP for Predicate Lists.

In this form of a loop, a set of nodes is specified and the predicate lists (see 13.1) attached to the nodes are looped through, with the loop variable--designated as a triple variable in this instance--representing one predicate triple at a time. The general form of this statement is:

```
$LOOP [,<option field>] [<loop name>] : <triple variable name> *
                                        <general node field>;
```

<option list>, <loop name>, and <triple variable name> (which is actually a <loop variable name>) are the same as for the other form of a loop (see 23.3).

<general node field> is described in 11.2. For each node, its predicate list is determined, and a resultant list of all of the triples of all of the predicate nodes is saved at entry to the loop. The order of the triples is determined by the order of the nodes in the field and the order of the triples on each node's predicate list (which may be a stack). See 6. for details about order in classes and 13.1 and 13.3 for details about order in predicate lists. If the resultant list of triples is empty, the loop is skipped, with control transferring to the statement following the $ENDLOOP statement. Triples which have been negated but not explicitly removed (such as with a *NEG) from a predicate list are still on the list and therefore will be used in a loop.

A triple variable can be used as a <triple> (see 12.2) in any action. A triple variable can also be used as a <sentence> in a subrule, and can thus be tested in numerous ways. When so used, the triple is identical to a <simple sentence> (see 26.1) and can be used independently or as an argument to one of several subrule functions.

Examples:

$LOOP LOOPNM3: Z*PEOPLE;
$LOOP: Q*FRIENDS(ENEMIES(PICK(PEOPLE)));

23.5 $ENDLOOP.

Every loop must be terminated with an $ENDLOOP statement, as described in 23.1. This statement is simply of the form: $ENDLOOP;

When an $ENDLOOP statement is executed, the loop variable defined by the associated $LOOP statement is set to the next value and control is transfered to the statement following the $LOOP statement. If the loop is finished--the loop variable has taken on all possible values--control is transfered to the statement following the $ENDLOOP statement and the loop variable is no longer defined.

An $ENDLOOP statement may be flowed into if it follows a statement which has just been executed. Also, while an $ENDLOOP statement may not have a name, it can be branched to (but only from a statement within the loop) by using the $NEXT feature of a branch field, as described in 10.2.

24. Introduction to Subrules.

The subrule is the last major component of the language yet to be described. Because it is such a broad topic, it will be covered in this and four subsequent chapters. Initially, it is possible to skip the more detailed information dealing with subrule expressions, complex sentences, and subrule variables without losing the ability to perform almost all desired functions.

24.1 Description of Subrules.

Subrules are used to test the state of the simulated universe. The tests cover all aspects of the simulation: the semantic network, class membership, the simulated time, the RV and H values of a triple, etc. Tests can be quite simple, or very complex with several things being tested at once and with variables being defined as a result of the test.

Rules and switches are the only components of the language which may have subrules, with the subrules being used to determine whether or not a rule's action list will be executed and which branch will be taken (see 10.1).

The subrule list of a rule (see 10.2) or switch (see 22.) is defined as:

<subrule list>::= {<subrule>;}$^+$

meaning that a rule or switch can have any number of subrules, each terminated by a semicolon.

During execution, when a rule or switch is reached the subrule list associated with that statement is examined. (If the statement has no subrules, it always evaluates to True.) One at a time, according to their physical order in the program, the subrules are evaluated. Each subrule yields a numeric probability value, either directly as the value of the test or indirectly from the true and false probability values associated with a subrule which returns a logical (True or False) result. If this numeric result is positive, it adds to the likelihood that the statement will evaluate to True, and if it is negative it lessens the likelihood of a True result. A value of zero would have no effect, and might be used, for example, if a subrule's being False had no importance.

As each subrule is evaluated, its numeric result is added to the results of the preceding subrules, a running cumulative probability being kept. After all subrules in the subrule list have been evaluated, the subrule random number generator is called and a random number is generated. If this random number, which is between 0 and 1, is less than or equal to the computed cumulative probability then the rule or switch is considered to be True, the rule's action list is executed, and the true branch taken. Otherwise, the statement is considered to be False and the false branch is immediately taken.

By manipulating the probability values associated with a logical subrule, or value generated by a numeric subrule, the execution of an action list and the path through a program can be made random with any desired degree of control, deterministic control also being possible. Since the random number generated is always between 0 and 1, a cumulative probability less than or equal to 0 will always cause a False statement evaluation, and a cumulative probability greater than or equal to 1 will always cause a True statement evaluation.

Furthermore, the following device exists for deterministic control. If at <u>any</u> time during the evaluation of the subrule list the cumulative probability becomes equal to 10 or more, or equal to -10 or less, no further subrules in the list will be evaluated. Instead, if the cumulative probability is greater than or equal to 10 the statement is True, the other condition yielding a False statement. This means, for example, that while a cumulative probability can temporarily become greater than one only to change value later, it cannot be changed once it becomes greater than 10 since no further subrules will be examined. This device is useful in specifying necessary and sufficient conditions. For example, a -10 associated with the False result of a logical subrule

means that the tested condition is necessary; a 10 associated with the True result means that the tested condition is sufficient.

Finally, a subrule may have two other components. A subrule may have an action list, this being almost identical to the action list of a rule (see 10.2), and being unconditionally executed after its associated subrule is evaluated. A subrule may also define a subrule variable, which is associated with an item in the test and whose resultant value is only those elements of that item which cause the test to be True (see 28.).

24.2 General Syntax.

As described above, a subrule can either yield a numeric result which is then used as its probability value, or it can be logical subrule and therefore be either True or False, with its probability value coming from associated numeric true and false values.

24.2.1 Logical Subrules.

A logical subrule is one in which the test of the universe returns a value of either True or False. Most tests return such values, as most subrule ask questions about the current composition of the universe, such as whether a certain triple has been asserted or if a certain element is in a particular set.

In order to translate this logical result into a numeric one, a subrule of this form has two numeric probability values associated with it. A True result causes its associated <true probability value> to be added to the cumulative probability and a False result causes the same to be done with its <false probability value>.

The general form of a logical subrule is:

```
<true probability value>, <false probability value> [,<option field>]:
    <subrule expression> [:<subrule action list>]
```

<true probability value> is an optionally signed, real or integral constant. It is added to the cumulative probability if the <subrule expression> is True. It is generally a number between 0 and 1, but may be larger or negative, with 10 and -10 having special meanings (see 24.1).

<false probability value> is the same as the <true probability value> above, except that it is added to the cumulative probability if the <subrule expression> is False.

<option field>, if present, overrides any options already in effect due to a group (see 9.2), rule (see 10.2), switch (see 22.), or loop (see 23.3 and 23.4). It affects only the subrule to which it is attached. The field may include one or both of the option characters C and O, in any order. C is described in 26.1.2 and O in 28.3.

<subrule expression> is the actual test of the universe. In a logical subrule, this expression must return a value of either True or False. <subrule expression>s are described beginning in chapter 25.

<subrule action list> is defined as: <action> {,<action>}*. The actions in this list are unconditionally executed after the subrule is evaluated. Generally, subrule actions are used to save information about subrule variables (see 28.). Any action that could be used in a rule's action list (see 11.-21. and 32.3) can be used here, except the *CALL subroutine action.

Examples:

```
-10,-.3,C:(JOHN LOVE MARY);
.3,-10:    (JOHN GOODRELATIONS MARY);
-.6,0,C:   (FRIENDS(JOHN) HATE FRIENDS(MARY));
.3,-7:     (MARY GOODRELATIONS JOHN);
```

24.2.2 Numeric subrules.

A numeric subrule is one in which the test of the universe returns a numeric value. The value may come from a triple, the simulated-time clock, or may be the duration of a triple in the network, among other possibilities. Functions which yield these numeric results are described in chapter 27. Arithmetic operations can be performed on the value in order to scale it for use as a probability value.

The general form of a numeric subrule is:

```
[,<option field>] : <subrule expression> [:<subrule action list>]
```

<option field> and <subrule action list> are as in logical subrules (see 24.2.1).

<subrule expression>, for a numeric subrule, must return a numeric value. This value may be stored in the simulated universe, it may be the result of some computation, or it may simply be a constant. In any case, the returned value is added to the cumulative probability. <subrule expression>s are described beginning in chapter 25.

Examples:

```
:   VAL(JOHN IQ)/100;
,C: (TIMER -DUR(JOHN LOVE MARY))/24H;
:   VAL(MARY BEAUTY);
```

25. Subrule Expressions.

The various components of subrule expressions will be described in this and the next two chapters. Basically, a subrule expression evaluates to either a numeric or logical value, which is then used in a subrule as described in the previous chapter. This value may come from something as simple as a numeric constant or from a complicated test of the universe.

The basic components of a subrule expression which are used to query the state of the simulated universe are <sentence>s and <subrule function>s. <sentence>s are used to test the semantic network and class membership, and are described in chapter 26. <subrule function>s return information associated with triples and information about the simulated time, and are described in chapter 27.

This chapter will deal with methods of combining tests of the universe, which is accomplished by using a number of numeric and logical operators. While this chapter can be understood without knowing what a basic test of the universe looks like, it may be useful to first read at least the section on <simple sentence>s in 26.1 and look at some of the available functions in 27.

25.1 Syntax.

The syntax of a subrule expression given below will only go as far as the <subrule operand> level, with each operand being described separately later. It will be concerned only with the combining of these operands, and the syntax is simply defining the following standard conventions: an expression in parentheses is evaluated before that expression is used in any operation, and a unary operator has precedence over a binary operator.

Not included in this BNF, for reasons of clarity, are two important rules: the operands of any operator must be of the correct type for that operator, and there is a hierarchy among the binary operators.

The syntax of a subrule expression, with the above constraints, is:

```
<subrule expression>::=<subrule factor>|<subrule expression><binary op>
                                                  <subrule expression>
<subrule factor>::=<subrule subfactor>|<unary op><subrule factor>
<subrule subfactor>::=<subrule operand>|(<subrule expression>)
<subrule operand>::=<sentence>|<subrule function>|<constant>
```

<binary op> is an operator taking two arguments and is described in the next section, 25.2.

<unary op> is an operator taking one argument and is described in 25.3.

<sentence> tests the semantic network and is described in chapter 26.

<subrule function> retrieves information from the simulated universe and is described in chapter 27.

<constant> is a numeric constant. It may be real or integral, and may be signed. A <constant> may be a <time> specification, as defined in 3.1.1, in which case <time> is converted to an integer.

A <constant> may be the entire subrule expression, allowing subrules of the simplest form. For example, :.3; is a subrule which adds .3 to the cumulative probability. This could be useful if some action list is to be executed with a constant probability value, as in:

```
$RULE:(JOHN LOVE MARY)
:.27;
```

Naturally, a <constant> used in this manner should be between 0 and 1.

A <constant> may also be used in a larger subrule expression, as shown in the syntax and demonstrated below.

```
$RULE:(JOHN LOVE MARY)
%The VAL function returns the value of a triple.
.5,0: VAL(JOHN AFFECTION MARY) GT 3;
%The DUR function returns the length of time that a triple has been in the network
.4,-.1: DUR(JOHN KNOW MARY) GE 2D6H;
%The closer the ages the more likely it is that
%the rule will be True.  ABS gives the absolute value of its argument.
:       1 - ABS(VAL(JOHN AGE) - VAL(MARY AGE))/20;
%An extra .2 is always added.
:.2;
```

25.2 Binary Operators.

Binary operators are used to combine subrule expressions, as defined in the syntax of 25.1. Each <binary op> takes two arguments and results in a single value. Although not specified in the syntax, the arguments must be of the correct type, numeric or logical. The <binary op>s are split into three categories, based on the type of arguments they take and the type of values they return.

1. Arithmetic binary operators.

The operands must be numeric. The result is also numeric. There are seven such operators: the usual arithmetic operators, + (addition), -(subtraction), * (multiplication), / (division), ** (exponentiation), and an operator which is generally a function in other programming languages, MOD.

All of these operators will take real or integral values as operands, but in all cases (except exponentiation to an integral power) the operands are converted to real values before the operation takes place. This means that division of integers will not result in a loss of the fractional part of the result, as is the case in FORTRAN.

If a negative number is raised to a real power or if zero is raised to the zero power, or if division by zero is attempted, an error message will be printed and a result of zero will be used.

Where Z=X MOD Y, the MOD operator returns the smallest possible positive value Z such that (X-Z)/Y = n where n is an integer. This means that the result is the smallest positive number that must be subtracted from the first argument in order to make it exactly divisible by the second argument. For example, 9 MOD 4 returns 1, and -9 MOD 4 returns -3.

2. Relational binary operators.

The operands must be numeric. The result is logical: True or False. There are the usual six operators of this type, five of which may be written in either mathematical notation (if two characters are required there can be

no space between the characters) or by using an abbreviation of the operator, as in FORTRAN.

The six operators are: EQ or =, returning True if the two operands are equal; NE (no equivalent mathematical notation), returning True if the two operands are not equal; LT or <, returning True if the first operand is less than the second; LE or <=, returning True if the first operand is less than or equal to the second; GT or >, returning True if the first operand is greater than the second; and GE or >=, returning True if the first operand is greater than or equal to the second.

For example, if (JOHN AFFECTION MARY) = 2.5 and the current CLOCK time is 3H10M, then:
VAL(JOHN AFFECTION MARY) GT 2.5 is False, and
CLOCK <= 5H22M is True.

3. Logical binary operators.

The operands must be logical, and the result is logical. There are two logical operators: AND returns True if both operands are True; OR returns True if either operand is True.

For example, if (JOHN KNOW MARY) is not in the network, and (JOHN AGE) = 25, then:

(JOHN KNOW MARY) AND VAL(JOHN AGE) GT 20 is False, but
(JOHN KNOW MARY) OR VAL(JOHN AGE) GT 20 is True.

It is important to note that, except in expressions in which subrule variables appear (see 28.), the evaluation of expressions with these operators is optimized. This means that once a result can be determined, the rest of the expression is ignored. For example, in the first example above, the first operand of AND is False so the second operand is not evaluated; in the second example, the first operand is False so the second is evaluated.

There is a hierarchy among binary operators which defines the order of operations in a situation such as:
op1 <binary op> op2 <binary op> op3
This hierarchy is the same as that for most programming languages. Pro-

ceeding from highest priority to lowest, it is: **, * and / and MOD, + and -, all relational operators, AND, OR.

Operators of higher priority will be performed before operators of lower priority. If operators of equal priority follow one another, evaluation will proceed from left to right. Parentheses can always be used to change the order of evaluation (or just to clarify the expression), with any expression inside parentheses always being evaluated first.

Examples (the ops are always assumed to be of the correct type):

op1 AND op2 OR op3	op1 and op2 ANDed, result ORed with op3
op1 AND (op2 OR op3)	op2 and op3 ORed, result ANDed with op1
op1 + op2 * op3 LE op4	op2 and op3 multiplied, result added to op1, result compared with op4
op2 LE op1 AND op1 LE op3	tests to see if op2$\leq$op1$\leq$op3; note that op1 must be repeated
op1 + op2 ** op3 / op4	exponentiation, then division, then addition

25.3 Unary operators.

Unary operators are used in subrule expressions in accordance with the syntax in 25.1. Each <unary op> takes one operand. Although not specified in the syntax, the operand must be of the correct type. There are six unary operators:

1. unary +, always optional and requiring and returning a numeric value.
2. unary -, requiring and returning a numeric value.
3. ABS, requiring a numeric operand and returning a numeric result equal to the absolute value of the operand.
4. ENTIER, requiring a numeric operand and returning a numeric result equal to the integral part of the operand. ENTIER -26.28 returns -26.
5. FLOAT, requiring a logical operand and returning a numeric result equal to 1.0 for a True operand and 0.0 for a False operand.
6. NOT, requiring a logical operand and returning the opposite logical value.

A unary operator is always used on its operand before any binary operator is used on that operand. For example, in

NOT op1 AND op2   NOT is applied first, and in

-op1**op2   -op1 is raised to the op2 power.

This rule may be effectively overridden by using parentheses, so in

NOT(op1 AND op2)   the AND is performed first.

If unary operators are adjacent to one another, the evaluation proceeds from innermost (closest to operand) operator to outermost.

FLOAT NOT op1   NOT is applied first

NOT FLOAT op1   is illegal, since NOT requires a logical operand

ABS ENTIER -26.28 = ENTIER ABS -26.28 = ABS ENTIER (-26.28)
  = ABS ENTIER -(26.28) = ... = -26.

26. Subrule Sentences.

Subrule sentences are used to query the semantic network in order to determine if triples are in the network or test the intersection of classes. A sentence may stand alone in a subrule, be connected to other items by using subrule operators (see 25.), or be an argument to one of a variety of subrule functions (see 27.) which test various aspects of the triples specified by the sentence.

All syntactic forms, descriptions, and examples in this chapter will ignore the use of subrule variables, which are described in chapter 28.

A sentence can be one of three types:

<sentence> ::= <simple sentence>|<complex sentence>|<class sentence>

<simple sentence>s specify triples and are described in 26.1.

<complex sentence>s are more powerful versions of <simple sentence>s and are described in 26.2.

<class sentence>s deal with the intersection of classes and are described in 26.3.

26.1 Simple Sentences.

A <simple sentence> specifies a semantic triple or group of triples in the same way as a <triple> is specified in an action asserting triples (see 12.2).

The appearance of a <simple sentence> in a subrule expression causes the specified triples to be looked up in the network.

A <simple sentence> must specify only primary triples, except in certain functions in which a <simple sentence> may specify secondary triples (see 27., in particular those functions having the letters SEC as part of their names).

One form of a <simple sentence> is:

(<general atom field><general relation field> [<general atom field>])

<general atom field> is described in 11.2. It must be a <general node field> when a primary triple is being specified. The object field is optional.

<general relation field> is described in 11.2.

A single triple is specified if all of the field are atoms. If any field returns a set of items, all possible combinations of triples are constructed, as described in 12.2. Each triple is searched for in the network, according to the rules given in the next two sections.

A <simple sentence> may also be: (<triple variable name>)
where the variable (see 23.4) takes on the value of a triple and thus acts as a <simple sentence>, in whose place it may always be used.
As the triple is a specific network triple, the sentence only specifies one triple in this case.

26.1.1 Rules for Testing the Network.

For each triple specified in a subrule sentence, the network is searched. If the triple is not in the network as it has never been asserted, or having been asserted has since been negated and not reasserted, then a value of False is returned.

However, even if the triple is in the network, it may seem to not be there and yield the same result as if it had not been there at all. There are three conditions which would cause this to happen.

1. If the triple was asserted in the network at the same simulated time at which the test is occurring (specifically, if the assertion time of the triple equals the current time), then the triple is considered not to be in the network. This rule can be overridden with the subrule option C (see 26.1.2).
2. Secondary triples are transparent to the network and a triple used only as a secondary triple will not be found. Secondary triples can be tested by using the SEC function (see 27.4).
3. If a triple has a hypotheticality (H) value (see 20.) of 1, it is not considered to be in the network unless a 1 precedes the triple in the sentence. If a 1 does precede the triple, then the triple will be found regardless of its H value. For example, (1 JOHN LOVES MARY) will be True if (JOHN LOVES MARY) is in the network with an H value of either 0 or 1, while the sentence (JOHN LOVES MARY) will be True only if the triple is in the network with an H value of 0. The digit 1 may precede a triple variable if it is being used as a sentence.

26.1.2 Subrule Option C.

The subrule option C can be used to override rule (1) in the above section. This option may appear in an <option field> in a group (see 9.2), rule (see 10.2), switch (see 22.), loop (see 23.3 and 23.4) or subrule (see 24.2). If this option is in effect for a particular subrule, those triples whose assertion times are the same as the current time will be found in the network. This option is useful if, for example,

a subrule in a group tests a triple which may have been asserted into the network by that same group. This option does not affect classes as they have no times associated with them.

26.1.3 Evaluation.

At this point, it has been described how a subrule sentence generates triples which are then searched for in the network. The value returned by a subrule sentence is now determined.

A logical value is always returned. The value is True if any of the specified triples are in the network, and False if none of them are in the network. If there are no subrule variables (see 28.) in the sentence, then as soon as any triple is found to be in the network, no further triples are examined and a value of True is returned. If you want to see if all of the specified triples are True, you must either use a complex sentence (see 26.2), or subrule variables (see examples in 28.4).

26.1.4 Examples.

For examples of simple sentences assume the following are class definitions: MEN = JOHN FRED; WOMEN = MARY JANE; GOODRELATIONS = LIKE LOVE;

In all cases, the determination of whether a triple is found in the network follows the rules of section 26.1.1 and section 26.1.2. The numeric probability values are described in chapter 24.

```
$RULE:   *SEC(EXAMPLE DEMONSTRATING LIST)(LIST SUBRULE);
.3,-.5:   (JOHN LOVE MARY);   tests for one triple
10,0:    (MEN LIKE JANE);   if either man likes Jane, the action list is
                            immediately executed
.4,-10,C:(MEN LOVE WOMEN);   if none of the four specified triples is in
                             the network (including triples inserted at
                             the current time), then the action list is skipped
.6,-.4:  (1 WOMEN GOODRELATIONS MEN);   eight triples are searched for,
                             their H values being disregarded
.2,0:    (/MEN LIKE MARY);   one triple is searched for (see 6.1)
.3,-.1:  (MEN FAT);   two triples are searched for
-.24,.368: (FRIENDS(ENEMIES(PICK(MEN))) GOODRELATIONS
                ENEMIES(FRIENDS(PICK(SPOUSE(WOMEN)))));
:.25;
```

In the second subrule above, (MEN LIKE JANE), if either man likes Jane a True value is returned. If you want to see if all men like Jane, and you know the elements in MEN, then you could use (JOHN LIKE JANE) AND (FRED LIKE JANE). However, you generally don't know what the elements of MEN will be during execution, so this method won't work. In general, a complex sentence (see 26.2) or subrule variables (see examples in 28.4) will solve the problem.

Finally, using the operators described in chapter 25, the following subrule list is possible. The functions NUM, VAL, DUR, and TIMER are described in chapter 27.

```
$RULE:    *SEC(JOHN LOVE MARY)(LOVE FINALLY);
.2,-10:   ((JOHN LIKE MARY) OR (MARY LIKE JOHN)) AND NOT (JOHN HATE MARY);
.4,-.3:   VAL(JOHN AGE) GE VAL(MARY AGE) + 3;
.2,-.1:   (JOHN AFFECTION MARY) AND (MARY AFFECTION JOHN);
.2,-.15:  ABS(VAL(JOHN IQ) - VAL(MARY IQ)) LE 25 AND
             (VAL(JOHN IQ) + VAL(MARY IQ))/2 GE VAL(PEOPLE IQ)/NUM(PEOPLE);
-.3,.2:   (1 JOHN BADRELATIONS FRIENDS(MARY)) OR
             (1 FRIENDS(MARY) BADRELATIONS JOHN);
,C:       (3*VAL(JOHN NUMBERDATES MARY) - 2*VAL(JOHN NUMBERDATES WOMEN))/
             VAL(JOHN NUMBERDATES WOMEN);
:         - FLOAT(VAL(MARY NUMBERDATES FRED) GT 3) * VAL(MARY AFFECTION
             FRED)/3;
:         DUR(JOHN KNOW MARY)/(VAL(JOHN AGE) * 1D * 365.25);
10,0:     VAL(SIMULATION ENDTIME) - TIMER LE 1H20M;
:.15;
```

26.2 Complex Sentences.

The complex sentence is an extension of the simple sentence (see 26.1). A complex sentence differs from a simple sentence only in that the complex sentence may have an expression for the relation field. This <relation field expression> is composed of relations, functions and constants joined by operators, in the same way that subrule expressions are composed of sentences, functions, and constants joined by operators (see 25.1). Complex sentences have many uses, with the main ones being discussed below in 26.2.3.

26.2.1 Syntax.

The syntax of a complex sentence is appropriately complex:

```
<complex sentence>::= (<general atom field> <relation field expression>
                              [<general atom field>])
<relation field expression>::=<relation field factor>|<relation field expression>
                                   <binary op> <relation field expression>
<relation field factor>::= <relation field subfactor>|
                                   <unary op> <relation field expression>
<relation field subfactor>::= <relation field operand>|(<relation field expression>)
<relation field operand>::= <single valued relation field>|<subrule relation
                                     function>|<constant>
```

The first rule says that a <complex sentence> is identical in form to a <simple sentence> (see 26.1), except for the relation field.

Rules two, three, and four describe the <relation field expression>, which is shown to be equivalent to a <subrule expression> (see 25.1), with the <relation field operand> of a <relation field expression> being used in place of the <subrule expression>'s <subrule operand>. See 25.2 and 25.3 for information concerning the use of operators in joining operands and for the definitions of <binary op> and <unary op>.

The <relation field operand> is the basic item that actually appears in the relation field, either alone or connected to other <relation field operand>s by various operators. Since this is the relation field of a

sentence, a <relation field operand> can naturally be a relation, but only of the form <single valued relation field> (see 11.2). This allows single relation names and loop variables, but does not allow classes of relations (this means that every <simple sentence> is not a <complex sentence> since <simple sentence>s allow classes of relations).

A <relation field operand> may also be a <subrule relation function>. These functions are described in 26.2.4 and are similar to the <subrule function>s of <subrule expression>s. An operand may also be a <constant>, as described in 25.1, except that here a <constant> may not appear by itself.

26.2.2 Evaluation.

Complex sentences are evaluated in much the same way as simple sentences (see 26.1.3), and the determination of whether a specified triple is in the network follows the rules given in 26.1.1 and 26.1.2.

The key difference in evaluation is that the <relation field expression> determines the type of a complex sentence, whereas simple sentences always return logical results. However, the appearance of a numeric value in the expression (as would be the case when using the VAL function or a duration function; see 26.2.4) does not assure a numeric complex sentence, as in (JOHN VAL(AGE)GT 20) and (JOHN VAL(AFFECTION)LT 2.5 MARY), which are both logical complex sentences. Of course, (JOHN VAL(AFFECTION)* 2 - 3 MARY) is a numeric complex sentence.

It is very important to note that if more than one triple is specified by a logical complex sentence (its subject or object, or both, fields are classes), the entire relation expression is evaluated for <u>each</u> subject-object pair. For example, if MEN = JOHN FRED, then (MEN LIKE AND NOT LOVE MARY) checks to see if (JOHN LIKE MARY) is in the network and if (JOHN LOVE MARY) is not; if both of these conditions are met then the sentence returns a value of True. Otherwise, the same two conditions are then checked for the subject FRED.

When multiple triples are specified in a numeric complex sentence, the value of the entire relation expression is determined for each subject-object pair, with these values then being added together for the final result.

Note that it is probably wrong to put, in one relation expression, a relation which takes an object and one which doesn't, since the object will be used in all tested triples.

26.2.3 Uses.

A list of reasons for using complex sentences is given below. In some cases, complex sentences provide the only means of getting a desired result, while in other cases complex sentences are alternatives to other, possibly not as convenient, methods.

1. In a simple sentence, a True value is returned if <u>any</u> of the specified triples are found in the network.

a. If the subject or the object is a class, in order to determine if <u>all</u> of the specified triples are in the network you can reverse the test condition and see if any of the triples are not in the network. For example,

(MEN LOVE JANE) is True if any man loves Jane and False if none love her,
NOT(MEN LOVE JANE) is True if no man loves Jane and False if any man
loves here, but
(MEN NOT(LOVE)JANE) is True if any man doesn't love Jane and False
if <u>all</u> men do love her.

The same test can be constructed with subrule variables (see examples in 28.4).

b. If the relation is a class, method (a) won't work since the relation field of a complex sentence may not have any classes of relations. Naturally, if you know all of the elements of the class, you can AND simple sentences together, but this knowledge is unlikely. The best solution is to use subrule variables (see examples in 28.4).

c. If two fields are classes, things get complicated. For example,
(MEN LOVE WOMEN) is True if any man loves any woman.
(MEN NOT(LOVE)WOMEN) is False if each man loves every woman.
However, to test if every man loves at least one woman, or if every woman is loved by at least one man, subrule variables are usually necessary (see examples in 28.4). Finally, how do you test for the existence of at least one man who loves every woman? In this last case, and for other similar tests, and when all three fields are classes, the only solution seems to be to use loops and ask the question for each member of the class, reducing the test to where it has fewer classes in it.

2. In a simple sentence, a value can be returned if a function (see 27.) is used, the value being the sum of the values of the specified triples. When a complex sentence returns a numeric result, the same rule applies. For example (see 26.2.4 for subrule relation functions),
(MEN VAL(SALARY) - VAL(TAXES)) gives the same results as
VAL(MEN SALARY) - VAL(MEN TAXES).

However, a simple sentence in a numeric function, combined with an operator and another value to produce a logical result, allows you to determine only if the sum is of the desired value. Thus, VAL(MEN SALARY) GT 20000 is True if the sum of the men's salaries is greater than 20000. So, to determine if at least one man's salary is greater than 20000, a simple sentence won't work, but you can use the complex sentence (MEN VAL(SALARY) GT 20000). If you want to determine if all of the men's salaries satisfy the condition, reverse the test and use
(MEN VAL(SALARY)LE 20000), which if False if they all earn over 20000. (As in (1a) above, this last result could also be achieved by using the test (MEN NOT(VAL(SALARY)GT 20000)), but the first method is obviously clearer.)

This method applies to all functions which take a sentence as an argument and return a numeric result. If this result is to be tested for each triple, instead of testing the sum of the results from the triples, the subrule version of the function might be used (see 26.2.4 and 27.).

3. In a sentence in which subrule variables are defined, a complex sentence may be necessary in order to insure that the subrule variables will get the proper values (see 28.).

4. Two simple sentences ORed together in a subrule expression, such as (JOHN LIKE MARY) OR (JOHN LOVE MARY), could be combined into one simple sentence by, for example, placing LIKE and LOVE into the relation class GOODRELATIONS and then using (JOHN GOODRELATIONS MARY). However, it may not be convenient to place them in such a class, in which case the complex sentence (JOHN LIKE OR LOVE MARY) is more readable than the

two separate sentences. Remember, however, that classes of relations are not allowed in complex sentences.

5. The subrule expression (MEN LIKE MARY) AND (MEN DATE MARY) is True if any man likes Mary and any man dates Mary, the two men not necessarily being the same. However, the complex sentence
(MEN LIKE AND DATE MARY) is True only if at least one particular man both likes and dates Mary. This problem can also be solved by using subrule variables (see examples in 28.4), which will work even if the relations are classes.

6. The only way to ask certain questions may be to use some combination of simple sentences, complex sentences, subrule variables, and loops.

26.2.4 Subrule Relation Functions.

The subrule relation functions allow retrieval of certain information associated with triples, information about the simulated time, and class size and membership. They are very similar in effect to the subrule functions (see chapter 27) of the same names, where they are more fully explained.

A complex sentence with a subrule relation function is evaluated in the usual way (see 26.2.2), with the entire relation field expression being evaluated for <u>each</u> subject-object pair. Subrule relation functions are most useful in those cases when subrule functions do not allow certain questions to be asked; see 26.2.3 for examples, putting appropriate subrule relation functions in the given relation fields.

Five of the subrule relation functions are identical to their subrule function counterparts. These functions may not appear alone in the relation field as they do not represent relations but only return numeric values. These functions are evaluated without regard to the subject and object fields of a sentence in which they appear. They are used in conjunction with other items in the relation field, with relational binary operators (see 25.2) generally being used. The five functions are: CLOCK (see 27.1 for more details), TIMER (see 27.1), RANDOM (see 27.2), NUM (see 27.3), NUMP (see 27.3).

For example, (JOHN DUR(LOVE) GT TIMER/2 WOMEN) would be True if John loved any women for more than half of the current simulated time. A subrule relation function is necessary here, since DUR(JOHN LOVE WOMEN) GT TIMER asks a different question.

(JOHN VAL(AFFECTION) GT RANDOM WOMEN) is True if John's affection for any woman is greater than a randomly generated number (a new number is generated for each subject-object pair). Again, VAL(JOHN AFFECTION WOMEN) GT RANDOM asks a different question.

It is not easy to find reasonable uses for the NUM and NUMP subrule relation functions. However, if NUMLIKE is a relation which is used in triples whose values are the number of people liked by the triples'

subjects, then (MEN VAL(NUMLIKE) GT NUM(PEOPLE)/2) is True if any man likes more than half of the people.

The next five subrule relations functions do use the subject and object fields of the sentences they appear in and can stand alone in a relation field since they specify relation names and thus allow complete triples to be formed.

There are two important differences between these functions and their subrule function counterparts: the subrule relation function has only a relation as an argument whereas the subrule function has a sentence as its argument, and the argument for a subrule relation function may only be a <single valued relation field> (see 11.2) while a subrule function can have a relation class in the relation field of its sentence.

The five functions are: DUR (see 27.5 for more details), DEL (see 27.6), EVER (see 27.7), RVAL (see 27.8), and VAL (see 27.9).

For example, (JOHN DUR(LIKE) GT .5 * DUR(KNOW) MARY) is True if John has liked Mary for more than half of the time he has known her. Two simple sentences could be used to give the same result, although simple sentences could not ask the question (JOHN DUR(LIKE) GT .5 * DUR(KNOW) WOMEN) since here the relation expression is evaluated for each pairing of John and one woman. Note that if the relation expression evaluates to a numeric result the values are added, so (JOHN VAL(LIKE) WOMEN) is the same as VAL(JOHN LIKE WOMEN).

Other examples of the usefulness of subrule relation functions are:

```
(MEN EVER(LIKE) AND EVER(DATE) FRIENDS(MARY))
(MEN VAL(LOVE) GT RANDOM * 2.3 MARY)
(MEN DUR(LOVE) GT TIMER/2 AND DEL(ARGUEWITH) GE 2H4M
     AND (RVAL(LOVE) EQ 5 OR VAL(AFFECTION) GE (RANDOM -.5)*2)
     AND WANTMARRY MARY)
```

Finally, it is possible to use <atom class function>s (EQL and NEQ, see 26.3.1) and <predicate class function>s (EQLP and NEQP, see 26.3.2) in the relation field expressions of complex sentences. If one of these functions appears alone in a relation field, the sentence is a class sentence (see 26.3) and no relation is specified. In a complex sentence, these functions can appear only in expressions in which a relation is somewhere specified.

Due to the way complex sentences are evaluated, these functions have somewhat different effects here than they have in class sentences; this is because here the entire relation field expression is evaluated for each subject-object pair. A subject and object are EQL if they are the same atom, and are otherwise NEQ. Two nodes are EQLP if any triple on one node's predicate list is identical to any triple on the other node's predicate list, and are otherwise NEQP.

For example, (PEOPLE HATE AND EQL PEOPLE) is True only if some person hates himself. (PEOPLE LOVE AND NEQ PEOPLE) is True only if some person loves someone other than himself. (JOHN LOVE AND EQLP MARY) is True if John loves Mary and at least one predicate is common to the predicate lists of the nodes JOHN and MARY. While the last result could be accomplished with simple sentences, (MEN LOVE AND EQLP MARY) requires a complex sentences (or a loop or subrule variables) to ask if at least one man meets the criteria set for John in the preceding example.

26.2.5 Examples.

```
$RULE:    (JOHN LOVE MARY);
.3,-10:   (JOHN LOVE AND NOT (HATE OR DISLIKE) MARY);
      :   (JOHN VAL(AFFECTION)/3 MARY);
.4,-.2:   (JOHN DUR(LIKE) GT 3W4D OR DUR(LOVE) GT 2W5D10M MARY);
-.3,0:    (JOHN NOT LIKE FRIENDS(MARY));
.15,-.2:  (1 JOHN VAL(AFFECTION) LT 2 OTHERWOMEN);
-.2,.05:  (OTHERMEN DUR(DATE) GT 2W MARY);
-.3,.2:   (MARY LOVE AND EVER(MARRY) OTHERMEN);
```

The first three subrules could be changed to a combination of simple sentences, but the last four are easiest to ask with complex sentences although loops or subrule variables could be used instead. Loops or subrule variables would have to be used, for example, to see if John's affection toward Mary is more than his affection toward any other woman.

26.3 Class Sentences.

These sentences are used to test for class intersections, either for classes of atoms or for predicate lists of complex predicate nodes. The description of the sentence depends upon the type of class being tested, where

<class sentence>::= <atom class sentence>|<predicate class sentence>

The class functions described in this section can also be used, with somewhat different results, in the relation expression of complex sentences (see 26.2.4).

Class sentences always return logical values.

26.3.1 Atom Class Sentences.

An <atom class sentence> compares the atoms in two classes, using the EQL and NEQ functions.

```
<atom class sentence>::= (<general atom field><atom class function>
                                              <general atom field>)
```

<general atom field> is described in 11.2. If the two such required fields are of different atom types they can never have elements in common.

<atom class function>::= EQL|NEQ

EQL tests for the non-empty intersection of the specified fields. All possible pairs of elements in the two fields are compared, any as soon as any element in one field is found to be the same as an element in the other field, evaluation stops and TRUE is returned. If the two fields have no elements in common, FALSE is returned.

NEQ is a rather strange functions and is not often useful. NEQ returns FALSE <u>only</u> if the two specified fields are identical <u>and</u> each field has only one element in it. If either field has more than one element, then NEQ always returns TRUE; this is because it will be True that at least one element from one field is different from at least one element in the other field.

Examples:

(JOHN EQL FRIENDS(MARY))	True if John is a friend of Mary
(PEOPLE EQL FRIENDS(MARY))	True if any person is a friend of Mary
(PICK(FRIENDS(MARY)) EQL JOHN)	True if the single randomly picked friend of Mary is John
(JOHN NEQ HERO)	if HERO is a class having one element then this is True if John is not that element and False if JOHN is that element; if HERO has more than one element then this sentence is True.

26.3.2 Predicate Class Sentences.

A <predicate class sentence> compares the predicates in the predicate lists of the specified nodes, using the EQLP and NEQP functions. Predicate nodes are described in chapter 13.

```
<predicate class sentence>::= (<predicate list operand><predicate class
                                 function><predicate list operand>)
```

<predicate list operand>::= <general node field>|<triple variable>
<general node field> is described in 11.2. If a set of nodes is specified, all of the predicates of all of the nodes are considered.
<triple variable> is described in 23.4.

<predicate class function>::= EQLP|NEQP

EQLP tests for the non-empty intersection of the specified predicate fields. As soon as any predicate in one field is found to be the same as a predicate in the other field, evaluation is terminated and True returned. If the two fields have no predicates in common, False is returned.

NEQP is tricky and should be used carefully. If a field specifies a set of nodes, the nodes are examined one at a time, independently of each other; if both fields are sets of nodes, all combinations of nodes are tested. The result is True if any predicate list of one field (the list may have one member in the case of a <triple variable>, or it may be the predicate list of one of the specified nodes) has an empty intersection with any predicate list of the other field.

Examples:

(JOHN EQLP MARY)	True if John and Mary have one predicate in common
(JOHN EQLP PEOPLE)	True if one of John's predicates is also on the predicate list of any person; obviously true if John is in the class PEOPLE
(JOHN EQLP PRED)	if PRED is a triple variable, True if PRED is on John's predicate list.
(JOHN NEQP WOMEN)	True if the predicate lists of John and any women have an empty intersection.

27. Subrule Functions.

Subrule functions are used to retrieve certain information associated with triples, information about the simulated time, and class size. These functions are built in to the system and are used as subrule operands (see 25.1). Many of these functions can also be used as relation field operands in complex sentences, in which case they generally apply to different arguments (see 26., especially 26.2.4).

The two functions which return information about the current simulated time, TIMER and CLOCK, take no arguments. The two functions giving the number of elements in a class or predicate list, NUM and NUMP, take a single argument. The other functions give information about triples and thus each takes either a primary triple for an argument, or for a second form of the function (having SEC in its name), a primary triple followed by a secondary triple as arguments.

It is important to note which functions return numeric results and which return logical results, as this affects the type of operators (see 25.2 and 25.3) a function can be an operand of, or if the function stands alone in the subrule the type of subrule (see 24.2) it is and thus the requirement as to probability values.

Except for one function all of the functions which take triples as arguments allow only simple sentences (see 26.1) or triple variables (which act as simple sentences, see 23.4). If the argument acts as a primary triple, the sentence must have the form of a primary triple; if it acts as a secondary triple, the sentence must have the form of a secondary triple (see 12.5). The exception is the function SEC, which tests to see if a secondary triple exists in the network and which can have a complex sentence (see 26.2) for its primary sentence.

In all cases the determination of whether a specified triple is in the network follows the rules of 26.1.1 and 26.1.2, and a 1 may precede a sentence in order to find hypothetical triples (see 20.).

27.1 Simulated Time.

There are two functions which take no arguments and which return information about the current simulated time. The time returned is converted into an integer equal to a number of basic time units, so that 2W19H27M is returned as 21327 (see 3.2 for changing the units of time. Therefore, the number returned can be used with the various numeric operators (see 25.2 and 25.3).

27.1.1 TIMER.

The TIMER function returns the current simulated time, such as 3H10M which is returned as 190 for the ordinary units of time. The subrule expression DUR(JOHN LOVE MARY) GT TIMER/2 is True if John has loved Mary for more than half of the current time; assuming the simulation started at time zero, TIMER/2 is actually half of the elapsed simulation time.

27.1.2 CLOCK.

The CLOCK function returns the time of day by calculating the current time modulo 1440 (or modulo the number of basic time units in your "day"; see 3.2). The subrule expression CLOCK LE 5H30M is True if it is earlier than 5:30 in the morning, and CLOCK GT 14H10M is True if it is later than 2:10 in the afternoon.

27.2 Random Number (RANDOM).

The RANDOM function takes no arguments and returns a randomly generated number between 0 and 1. (It uses the same random number generator as used in subrules; therefore, the insertion and execution of this function will cause results from the rest of the program to differ from results gotten before this function reference was added.) So, VAL(JOHN IQ) GT 1.5 * RANDOM will be True or False depending on the generated random number, as will also be the case in the subrule expression NUM(PEOPLE)/5 LE RANDOM + 2.

27.3 Size of Class or List (NUM and NUMP).

These functions each take one argument and return a number equal to the number of elements in the given class or the number of triples on the predicate list (see 13.1) of a complex predicate node. See 11.2 for a description of the argument types.

A call to the NUM function is of the form:

NUM(<general class reference>)

The value returned is the number of elements specified by the argument, which may be a specific class or may specify a union of class. For example, if currently PEOPLE = JOHN MARY FRED, then NUM(PEOPLE) = 3 and NUM(FRIENDS(PEOPLE)) = the total number of (different) friends of John, Mary, and Fred.

A call to the NUMP function is of the form:

NUMP(<general node field>)

The value returned is the number of triples on all of the predicate lists of the specified nodes. A specified node may have no predicates. No check is made for duplicates, so the same triple may be counted more than once. The subrule expression NUMP(JOHN) EQ NUMP(MARY) is True if both nodes have the same number of predicates. NUMP(PEOPLE)/2 LE 6 first calculates the number of predicates contained on the predicate lists of all of the nodes in the class of people.

27.4 Secondary Triples in Network (SEC).

The SEC function causes the specified secondary triples of the specified primary triples to be looked up in the network. It is of the form:

SEC <simple or complex sentence><simple sentence>

The first sentence specifies the primary triple(s) and can either have a single relation (see 26.1) or a relation expression (see 26.2). The second sentence specifies the secondary triple(s) and must be a simple sentence, although as a secondary triple it may have a relation in the first position.

The SEC function looks up triples in the same way that the *SEC action (see 12.5.1) asserts them. For each primary triple specified (more than one if one or more fields are class references) all of the specified secondaries are searched for to see if they are pointed to by the primary (see 26.1.1 for an explanation of how the primary triple is searched for). This process stops and returns True as soon as any secondary is found on a primary's list of secondaries.

Both the primary and the secondary triple may be a triple variable (see 23.4). Either one (or both triples) may have an H value of 1 specified, as described in 26.1.1, item 3.

Some examples of the SEC function, using simple primary sentences are:

SEC(JOHN LOVE MARY)(MARY PRETTY)   looks up one secondary

SEC(MEN LOVE MARY)(1 MARY PRETTY)   True if one man loves a Mary who is, perhaps hypothetically, pretty

SEC(MEN LOVE WOMEN)(WOMEN PRETTY)   True if any man loves any woman who is pretty. As with *SEC, some meaningless tests are made; for example, (JOHN LOVE MARY)(SUE PRETTY) would be searched for.

SEC(MEN GOODRELATIONS MARY)(MARY PRETTY)

SEC(MEN MEET JOHN)(MEET AT GAME)

The problem in the third example above (as well as the more important problem with *SEC) due to looking up <u>all</u> possible combinations of primary and secondary triples could be solved by using loops and putting a loop variable in the place of a class reference.

Finally, the primary triple in the SEC function may be a complex sentence. In this case, the evaluation depends upon the item in question in the relation expression, where:

1. If an item in the relation expression is a relation, the specified secondary triple(s) are examined for the primary having that relation. The value returned from looking up the secondary(ies) is the value of the relation in the relation expression.
2. If an item in the relation expression is a call to a subrule relation function (see 26.2.4), the secondary triple is ignored and the value returned by the function is based <u>only</u> on the primary triple.

Some examples of this complicated construction are:

```
SEC(JOHN LOVE OR LIKE MARY)(MARY PRETTY)     True if either primary
        triple has the given secondary
SEC(MEN LOVE OR DUR(KNOW) GT 5H AND LIKE MARY)(MARY PRETTY)
        the DUR function looks only at the triple (JOHN KNOW MARY)
SEC(MEN VAL(LIKE) GT RANDOM MARY)(MARY PRETTY)
        the secondary triple is ignored in this case, following rule
        (2) above; obviously, the function shouldn't have been used,
        as no stand-alone relation appears in the primary sentence.
```

27.5 Duration In Network (DUR and DURSEC).

These functions look up the specified triple(s) and determine the length of time since it was (they were) last asserted in the netowrk. (Various actions update assertion times; see the sections on the particular actions.) DUR is used for durations of primary triples and DURSEC for durations of secondary triples. In both cases, only simple sentences (see 26.1) or triple variables (see 23.4) can be used as arguments.

The rules for finding primary triples in the network are given in 26.1.1. DURSEC looks for secondaries in the same way that the *SEC action (see 12.5.1) asserts them. For each primary triple specified, all of the specified secondaries are searched for to see if they are pointed to by the primary. In both functions, an H value may be specified in a sentence (see 26.1.1, item 3).

The value returned is treated as a number. If a specified triple is not currently asserted in the network it is skipped. If more than one specified triple is found, the value returned is the <u>sum</u> of the durations of the triples.

The syntax of these functions is:

```
DUR<simple sentence>
DURSEC<simple sentence><simple sentence>
```

Examples:

```
DUR(JOHN LOVE MARY)
DUR(1 MEN GOODRELATIONS FRIENDS(JOE))
DURSEC(JOHN LOVE MARY)(MARY PRETTY)
DURSEC(MEN RELS(POSITIVE) MARY)(MARY PRETTY)
```

27.6 Duration Since Deletion From Network (DEL and DELSEC).

These functions look up the specified triple(s) and determine the length of time that has passed since it was (they were) last negated in the network. The value returned is numeric. If multiple triples are specified, the sum of the times is returned. See DUR (27.5) for more details of the method of evaluation.

Syntax:

```
DEL<simple sentence>
DELSEC<simple sentence><simple sentence>
```

Examples:

```
DEL(JOHN LOVE MARY)
DEL(FRIENDS(JOHN) KNOW FRIENDS(MARY))
DELSEC(1 JOHN LOVE MARY)(1 MARY PRETTY)
DELSEC(MEN LOVE FRIENDS(MARY))(FRIENDS(MARY) PRETTY) but watch out
            for extra, probably meaningless, tests as described in
            the examples of SEC (see 27.4)
```

27.7 Existence in Network at Any Time (EVER and EVERSEC).

These functions look up the specified triple(s) and determine if it was (they were) ever asserted in the network. The value returned is True if any one of the specified triples was ever asserted into the network, even if that triple has since been negated. Triples are looked up for EVER and EVERSEC in the same way as for DUR and DURSEC (see 27.5), but a logical value is turned.

Syntax:

```
EVER<simple sentence>
EVERSEC<simple sentence><simple sentence>
```

Examples:

```
EVER(JOHN LOVE FRIENDS(MARY))     True if John loves, or once loved,
                                  a friend of Mary's
EVERSEC(FRIENDS(JOHN) LOVE MARY)(MARY PRETTY)
```

27.8 RV (RVAL and RVALSEC).

The RV (reality value) is described in chapter 19. These functions look up the specified triple(s) and determine its (their) RV. If multiple triples are specified the value returned is the sum of the RV's. If a specified triple is not found, it is skipped. Triples are searched for in the same way as in DUR and DURSEC (see 27.5).

Syntax:

```
RVAL <simple sentence>
RVALSEC <simple sentence><simple sentence>
```

Examples:

```
RVAL (FRIENDS(JOHN) LOVE FRIENDS(MARY))
RVALSEC (JOHN LOVE MARY)(MARY PRETTY)
RVALSEC (MEN GOODRELATIONS MARY)(MARY IQ)
```

27.9 Numeric Value of a Triple (VAL and VALSEC).

These functions look up the specified triple(s) and determine its (their) numeric value(s). The value returned is numeric. If multiple triples are specified, the sum of the value is returned. VAL and VALSEC are evaluated in the same way as DEL and DELSEC (see 27.5). See 12.4 and 12.5.3 for rules concerning the numeric values of triples.

Syntax:

```
VAL<simple sentence>
VALSEC<simple sentence><simple sentence>
```

Examples:

```
VAL(JOHN IQ)
VAL(1 MEN AFFECTION WOMEN)
VALSEC(JOHN LOVE MARY)(MARY BEAUTY)
VALSEC(MEN KNOW WOMEN)(WOMEN IQ)     where extra, probably meaningless,
          tests will be made, as described in the examples of SEC (see 27.4)
```

Since the VAL function is used quite often, and generally with other terms in subrule expressions, some specific examples follow.

Assume the following: (JOHN AFFECTION MARY) = 3.5, (FRED AFFECTION MARY) = 4.2, (JOHN CAREFOR MARY) = 9, (JOHN IQ) = 150, (FRED IQ) = 100, (MARY IQ) = 120 is hypothetical, MEN = JOHN FRED, GOODRELATIONS = AFFECTION CAREFOR LOVE and PEOPLE = MEN MARY JANE. Again, sections 26.1.1 and 26.1.2, and chapter 24 give the pertinent rules. Chapter 25 describes the simple arithmetic operators used to assure reasonable numeric results.

```
$RULE:     *SEC(EXAMPLE DEMONSTRATING LIST)(LIST SUBRULE);
:          VAL(JOHN.AFFECTION MARY)/10;    returns .35 (3.5/10)
,C:        (VAL(MEN AFFECTION MARY) - 7) **2;    returns .49 ((3.5+4.2-7)^2)
:          -1/(VAL(PEOPLE IQ) - 200);    returns -.02 (-1/(150+100-200))
:          VAL(1 PEOPLE IQ)/1000;    returns .37 ((150+100+120)/1000)
:          VAL(MEN GOODRELATIONS PEOPLE)-16;    returns .7 (3.5+4.2+9+tenO's-16)
```

28. Subrule Variables.

Subrule variables are local variables known only within the subrule list (see 24.1) in which they are defined. Although all previous descriptions of subrules have not considered subrule variables, the rules and syntax given in those descriptions remain valid as subrule variables consitute an extension to the power of subrules.

Subrule variables have two main uses:

1. They store those values of a field in a sentence which make that sentence True. These values may be permanently saved by making use of subrule actions.
2. They allow certain questions to be asked which could either not otherwise be asked, or which could be asked in a difficult and less natural way using previous constructs.

Subrule variables may be used in any of the three types of sentences (see 26.). The first time a subrule variable is used in a subrule list it must be defined. Subrule variables can be quite useful, and if the next two sections seem confusing, numerous examples are given in 28.4.

28.1 Definition and Use of Variable.

A subrule variable must be defined as such and given an initial set of elements the first time it is used in a subrule list. A subrule variable can be defined anyplace a <general class reference> can be used in a sentence, namely: 1. the subject, relation, and object fields in a simple sentence (see 26.1), and 2. the subject and object fields in a complex sentence (see 26.2) or a class sentence (see 26.3). A subrule variable can be defined only once in a subrule list.

Once a subrule variable has been defined, it may be used in any sentence in the same subrule list by just using its name anywhere a <general class reference> is allowed. Subrule variables may also appear in subrule actions, again appearing anywhere a <general class reference> is allowed, except where its appearance would cause a change to be made to the value of the subrule variable; specifically, subrule variables cannot appear as the destination field of class actions or the *EXPAND action, or in the *ERASE action.

A subrule variable is defined as follows:

<subrule variable definition>::= <subrule variable name>.<general class reference>

A <subrule variable name> is a variable name (see 2.2) which must not have been previously in this subrule list and which can subsequently be used only in the subrule list in which it is defined. A <subrule variable name> is deleted as a name after the subrule list in which it is used is finished, and therefore may later be used again in another subrule list. <general class reference> is described in 11.2. If this is itself a subrule variable, it is used only for its list of elements and doesn't otherwise act as a subrule variable.

Upon definition, the subrule variable gets as its value the set of elements in the related <general class reference>. From here on, no new elements may be added to the subrule variable; instead, elements are deleted from the variable in the following way: each time the variable is used in a logical primary sentence, either in its initial definition or in later use, the only elements allowed to remain in the subrule variable are those which cause the sentence to be True. A more detailed description of this rule is given in the next section.

28.2 Effects of Use.

1. In a simple sentence (see 26.1), or in a complex sentence yielding a logical result (see 26.2), the only elements allowed to remain in any subrule variable(s) used in the sentence are those which, in at least one context, make the sentence True; any other values in the variable(s) will be deleted.
2. When a subrule variable is in a primary sentence which is an argument to a function (see 27.) which yields a logical result (SEC, EVER, EVERSEC), the variable keeps only those elements which allow the function to be True.
3. Whenever a subrule variable appears in a logical sentence or function (see (2) above), all combinations of elements in that sentence will be tested instead of just testing until the first True is found. This is necessary to correctly update the variable. Also, if that sentence appears in a subrule expression, the sentence will always be evaluated even if its evaluation isn't necessary to determine the outcome of the expression (as could be the case when using an OR in the expression). This last part can be overridden; see the next section.
4. In a numeric complex sentence, or in a sentence used as an argument to a numeric function, subrule variables may be used but their values will not be updated and their presence will not affect the evaluation of the sentence or expression (rule (3) is not followed).
5. Subrule variables can be used in secondary sentences (as arguments to functions) but the variables won't be updated and won't affect the evaluation of the sentence or expression (rule (3) is not followed).
6. For a class sentence there are four cases:
   a. In a sentence using EQL, a subrule variable keeps the elements in the intersection of the two specified fields. If both fields use subrule variables, the variables will be equal after the sentence is evaluated.

b. A subrule variable in an EQLP sentence keeps those nodes which are in an EQLP relationship (have non-empty predicate list intersections) with at least one node of the other field.

c. In a sentence using NEQ, a subrule variable keeps the elements which are difference from at least one node in the other field; that is, those nodes which are NEQ to at least one node in the other field. This is probably not too useful.

d. In a NEQP sentence, a subrule variable keep those nodes which are in a NEQP relationship (have empty predicate list intersections) with at least one node of the other field.

7. If a logical sentence or function (see (2) above) returns False as its value then all subrule variables in the sentence are erased-all of their elements are deleted.

8. Because subrule variables are concerned only with the sentences they appear in and not with the rest of the subrule expression, it is sometimes more useful to use complex sentences instead of simple sentences, effectively moving the expression into the sentence and thus having it affect the subrule variables.

28.3 Subrule Option O.

The subrule option O can be used to override rule (3) in the above section. This option may appear in an <option field> in a group (see 9.2), rule (see 10.2), switch (see 22.), loop (see 23.3 and 23.4), or subrule (see 24.2). If this option is in effect for a particular subrule, then that subrule is evaluated in an optimized way in that the presence of subrule variables does not cause the second part of rule (3) in 28.2 to be followed. In other word, in an expression, a sentence will not be evaluated if its evaluation isn't necessary to determine the outcome of the expression, even if the sentence has subrule variables. This means that some subrule variables in the subrule may not be updated, although the subrule will naturally return the correct result. This option could be used to speed up execution where possible. Naturally, if the subrule doesn't contain any subrule variables, the option has no effect.

28.4 Examples.

For the first set of examples, all of the subrules are assumed to be in a single, long subrule list in order to show the cumulative effects of using subrules. Probability values are not shown, but would be required in an actual program. Also, assume that PEOPLE = JOHN JACK MARY, WOMEN = MARY SUE, MEN = JOHN JACK, HOWLIKES = LIKES LOVES ADORES, (JOHN LIKES MARY), (JOHN LOVES MARY), (JACK LIKES MARY), (JACK LOVES SUE).

a. (P.PEOPLE LIKES MARY);   P retains JOHN and JACK

b. (R. PEOPLE LIKES MARY) AND (R LOVES MARY);   R first retains JOHN and JACK, but is then reduced to just JOHN

c. (S. PEOPLE LIKES AND LOVES MARY);   same result as in (b)

d. (R2. PEOPLE LIKES WOMEN) AND (R2 LOVES WOMEN);   R2 first retains JOHN and JACK, then continues as JOHN and JACK

e. (S2. PEOPLE LIKES AND LOVES WOMEN);   S2 keeps only JOHN; this is <u>not</u> the same result as in (d)

f. (1 T.PEOPLE V.HOWLIKES MARY); T retains JOHN and JACK, and V retains LIKES and LOVES; the 1 is an H value (see 20.)

g. (W.PEOPLE LIKES AND NOT LOVES MARY);   W retains only JACK

h. (X.PEOPLE LIKES MARY) AND NOT (X LOVES MARY);   X retains only JOHN because, unlike in (g), the 'NOT' is outside of the second sentence and doesn't affect the value of X; also note that this subrule returns a value of False, whereas in (g) the subrule returns True.

i. (1 W V MARY);   since W = JACK, V is reduced to LIKES

j. (W V Y.PEOPLE);   W = JACK and V = LIKES, so Y retains only MARY

k. (Y Z.HOWLIKES A.PEOPLE);   Y = MARY, but since she doesn't like anyone in any manner, Y, Z, and A are all erased

l. (B.PEOPLE DUR(LIKES) EQ DUR(LOVES) MARY);   assuming equal durations of liking and loving, B retains JOHN

m. (D.PEOPLE EQL C.MEN);   C keeps JOHN and JACK and D retains those same elements

n. (JACK EQL C) or (V EQL LOVES); C is reduced to just JACK, and since V = LIKES, V is erased; if the subrule option O were in effect, V would be unchanged

o. (F.PEOPLE NEQ MEN) or (C NEQ G.MEN) or (W NEQ JACK); F keeps all three people and its sentence automatically returns True; since C = JACK which is an element of MEN, G loses JACK, leaving C = JACK and G = JOHN; since W = JACK, this sentence is False and W is erased.

p. (H.PEOPLE I.HOWLIKES J.PEOPLE): *MOVE H TO TMPCLS1,

```
                                   *MOVE I TO TMPCLS2,
                                   *MOVE J TO TMPCLS3;
```

H retains JOHN and JACK, I retains LIKES and LOVES, and J retains only MARY; then the contents of these three variables are saved in three classes which can then be used outside of this subrule list, even in the action list of the rule having this subrule list since the subrules are evaluated before the actions take place

In the next set of examples, the numbers on the right refer to section 26.2.3 on complex sentences, and it is shown how subrule variables can be used to ask questions either otherwise requiring complex sentences or otherwise not possible to ask easily.

1a. (A.MEN LOVE JANE) AND NUM(A) EQ NUM(MEN); True if all men love Jane

(JOHN LOVE B.WOMEN) AND NUM(B) EQ NUM(WOMEN); True if all women are loved by John

1b. (JOHN C.GOODRELATIONS JANE) AND NUM(C) EQ NUM(GOODRELATIONS); True if John has all of the good relations with Jane

1c. (D.MEN LOVE WOMEN) AND NUM(D) EQ NUM(MEN); True if every man loves at least one woman

(MEN LOVE E.WOMEN) AND NUM(E) EQ NUM(WOMEN); True if every woman is loved by at least one man

(F.MEN LOVE G.WOMEN) AND NUM(F) EQ NUM(MEN) AND NUM(G) EQ NUM(WOMEN); True if every man and every woman has at least one lover

5. (H.MEN LIKE MARY) AND (H DATE MARY) True if at least one particular man both likes and dates Mary

The next set of examples shows how subrules are updated when used in sentences which are arguments to logical functions.

SEC(A.MEN LOVE MARY)(MARY PRETTY); A retains only those men who like a pretty Mary

SEC(MARY LIKE B.PLACES)(PLACES PRETTY); B retains only those pretty places which are liked by Mary

SEC(MARY SEE JOHN)(SEE IN B); B is used to specify some places, but B is not updated

EVER(C.MEN LOVE D.WOMEN); C retains those men who have ever loved a women, and D retains those women who were ever loved by a man

Finally, two other useful examples of subrule variables are:

(A.CLASS1 EQL CLASS2) AND NUM(A) EQ NUM(CLASS1);

which is True if CLASS1 is a subset of CLASS2, and

(B.CLASS1 EQL CLASS2) AND NUM(B) EQ NUM(CLASS1) AND NUM(B) EQ NUM(CLASS2);

which is True if CLASS1 is identical to CLASS2.

29. The Look-Ahead Rule.

It is possible to temporarily save the state of the simulated universe, look ahead into the future for a specified amount of time, use what happened during the look-ahead to determine later actions, and then return to the original state and continue the actual simulation. While this feature probably won't fit into most programs, it could be used to formulate plans for the future or perhaps in time-travel stories.

Since this construct is not often used, its syntax has been omitted from all other chapters in this manual and is instead described independently here. Therefore, any forms which don't seem to fit into previous descriptions are used only for the look-ahead rule. The look-ahead rule may be used anywhere an ordinary rule can be used.

In brief, the look-ahead rule works by saving the universe (on a temporary file) and then continuing the simulation beginning with the group which is first on the queue of scheduled groups. When the look-ahead terminates, the subrules of the look-ahead rule are evaluated, and specified actions are taken. The original system state is restored, and the simulation continues in the normal way.

29.1 Syntax.

The syntax of the look-ahead rule is:

RULE [,<option field>][<rule name>] : [<branch part>] <pre-actions>;
[<post-actions>]; {<subrule>;}*

It differs in two ways from an ordinary rule (see 10.2):

1. <action list> is replaced by <pre-actions>; [<post-actions>]
2. In a <subrule> (see 24.2), instead of a single optional <subrule action list> being allowed after the <subrule expression>, an optional <subrule action field> is used, where:
   <subrule action field> ::= :{[<look-ahead options>,] <subrule action list>;}*

The syntax of the look-ahead rule must meet the following conditions:

1. The <pre-actions> must somewhere include a *LOOK action. This action is of the form: *LOOK <time> and specifies the duration of the look-ahead. Look-aheads may also be terminated by execution of an *END action (see 18.) or by exhaustion of scheduled groups.
2. The <pre-actions> may somewhere include a *SAVE action. This action is of the form: *SAVE {<class name>}$^+$ where the class names are separated by blanks and subscripted class names are not permitted. The contents of the specified classes are not restored after the look-ahead finishes, and thus keep the values they had during the look-ahead.
3. Both the <pre actions> and the <post-actions> are defined as: <action> {,<action>}* where any action is allowed in either field, except the *LOOK and *SAVE actions discussed above, which can occur only in the <pre-actions>.
4. The <subrule action field> of a subrule can have any number of <subrule action list>s (see 24.2), each list having its own set of <look-ahead options>. The lists are separated by semicolons.

5. The <look-head options> can be one of these three fields:
   a. the single letter I, meaning immediate execution of the <subrule action list>,
   b. the single letter D, meaning delayed execution of the <subrule action list>, or
   c. the combination ID, meaning both immediate and delayed execution of the list.

These options control the time of execution of the subrule actions, where immediate execution means execution in the look-ahead environment and delayed execution means execution in the restored environment.

These options may also be specified in $RULE, $SWITCH, $LOOP, and $GROUP commands, and in the <option field> of a subrule, along with the other possible options. The usual notions of extent apply to the I and D options, where an option field at a lower level overrides one at a higher level.

If no look-ahead option is in effect for a particular <subrule action list> because one wasn't specified in the <look-ahead options> and no surrounding option field specified a look-ahead option, then the D option is assumed.

29.2 Execution.

The syntax rules only make sense when the execution of the look-ahead is explained, so what follows is a detailed step by step description of the execution of the look-ahead rule. These steps are followed whenever a rule which had a *LOOK action in its <pre-actions> is encountered during execution.

1. The current environment (simulated universe) is saved (on a temporarily file).
2. The <pre-actions> are executed. The *LOOK action is naturally skipped, and a *SAVE action, if present, is noted but causes no changes in the universe.
3. The simulation continues for the time specified on the *LOOK action. When the look-ahead begins, control is given to the group which is first on the queue of scheduled groups (it would be the group normally executed after the group having the look-ahead rule finished). This means no rules in the group which initiated the look-ahead (except the look-ahead rule itself) are ever executed during look-ahead. Furthermore, if a group containing a look-ahead rule is encountered during the look-ahead, that entire group is skipped.

   It is very important to note that just because some action is taken during the look-ahead, it does not necessarily follow that the same action will later occur during the actual simulation. In other words, the look-ahead is not a perfect predicter. This is because the random number generators are used during the look-ahead and so generate different numbers during the actual simulation than they generated during the look-ahead. Also, due to the rules in the above paragraph, slightly different parts of the program may be executed in the actual simulation. Of course, if all rules executed during the look-ahead are deterministic, the results will be the same in the actual simulation and the future will be seen perfectly.

4. The look-ahead is terminated when it has run for the time specified in the *LOOK action, due to the execution of an *END action (see 18.), or when there are no longer any groups scheduled. The END time of the actual simulation (see 3.1.1) is ignored during look-ahead.
5. After termination of the look-ahead, the subrules of the look-ahead rule are evaluated in the look-ahead environment. Subrule actions specified for immediate execution are executed in the look-ahead environment. These actions are unconditionally executed when their subrules are reached, so an action can effect a subsequent subrule.
6. The original environment is now restored, destroying all memory of what occurred during the look-ahead, with the exception that classes specified in the *SAVE action keep the values they had in the look-ahead environment.
7. All subrule actions specified for delayed execution are now unconditionally executed in the original environment.
8. If the look-ahead rule evaluates to True (based on the evaluation of its subrules in (5)), the <post-actions> are executed. The <branch part> then has the usual meaning, and the simulation continues in the normal way.

29.3 Example.

Although the look-ahead rule seems quite complicated, it is likely that you would use only some of its features in any particular look-ahead. An example is given below, although it should be noted that it is meant for clarification of the rules of syntax and doesn't necessarily make any sense.

```
$RULE,CI FUTURE: T(RULE3)
                 *LOOK 5D3H10M,                    ⎫
                 *SAVE MARRIED ENGAGED,            ⎬ <pre-actions>
                 *ADD MARY TO FRIEND(JOHN),        ⎪
                 (MARY LIKE JOHN);                 ⎭
                 (MARY MARRY JOHN),                ⎫ <post-actions>
                 *MOVE MARY TO WIFE(JOHN);         ⎭
.2,-.3:  (JOHN LOVE MARY): I, (MARY HAPPY), (JOHN HAPPY);
                           D, *SET (LOOKAHEAD EXECUTED) =+ 1;
0,0   :  (P.PEOPLE HAPPY): D, *MOVE P TO TEMPCLS1;
-1.5,0:  (JOHN KILL MARY);
.4,0  :  NUM(CHILDREN(MARY)) GT 0;
-.3,0 :  EVER(JOHN HIT MARY);
      :  VAL(JOHN SALARY)/50000;
-.5,.1:  EVERSEC (JOHN SEE OTHERWOMEN)(SEE SECRETLY);
      :  .15;
```

30. Using the Simulation System.

This chapter will describe the control cards needed for using the simulation system, the use of SAVE and RESTORE, which allow saving the compilation of a program for future use, and the actions *SAVESYS and *RESTORSYS, which allow saving the state of a simulation for future use. As noted several times, this manual does not include information on the generative mechanism, and so that part of the system will not be considered here.

30.1 Control Cards.

After the usual @RUN card required of all jobs, an @XQT card with the file name that the system currently resides on is used to begin execution of the system. For example, if the executable version of the system were on SYS*A.B, then @XQT SYS*A.B would be used.

This card is followed by your program, which generally begins with a $LIMITS command (see 3.) and always ends with an $END command (see 2.1).

Any number of @XQT-program sets may be included in one job. If the same program is to be executed several times, use the JUMBLE or CLOCK option in the $LIMITS command in order to get differing results (see 3.1.2). A user's program may also be on a file, in which case it is accessed via an @ADD card.

30.2 SAVE and RESTORE Options.

The SAVE option in the $LIMITS command (see 3.1.4) causes the result of the compilation of your program (an internal representation of the program in a form ready for execution) to be saved on a file. This allows further use of that program without recompilation, compilation generally being far more costly than execution for anything other than small programs. The program is saved on a temporary file named 20. and can be used elsewhere in the same job. If you want to keep this file for later use, then 20. should be made the internal USE-name of a catalogued file (see examples in 30.4).

Once the program having the SAVE option has been executed (the SAVE option doesn't stop the execution of a program, it just saves it), the saved compiled version may be executed any number of times by using the RESTORE option in the $LIMITS command. This says that the program can be found on file 20. (which may be the internal USE-name of another file) and should be executed using any other instructions in the $LIMITS command. This means that when executing a program by using the RESTORE option, the $LIMITS and $END commands can constitute the entire program. Note that the $LIMITS command can have its usual information, causing the saved program to produce different results each time it is executed.

As stated above, when using the RESTORE option the entire program can consist of just two commands, in which case the saved program is executed. However, it is often the case that you want to add to a saved program (you can't <u>change</u> it without recompiling it), especially if you are constructing your program in an incremental manner and testing each piece before adding a new one. Therefore, a saved program can be added to by following the $LIMITS command (which has the RESTORE option) with any additions you want made to the saved program.

Any component may be added to a saved program: atoms, classes, exclusions, more $NETWORK actions, and most importantly, new groups. As these new items will be added to the saved program to create one new, larger program, it should be clear that, for example, an added node can't have the same name as an original node or relation.

In order for a new group to be activated, there are several possibilities: it could be turned ON initially; while it could not have been referenced by name in an old action (that would have caused an error in the saved program), an old action could enable or call a class, one of whose members (in anticipation) is the name of the new group; using the $NETWORK section, the name of the new group could be added to a class which is called or enabled by an old action.

Finally, as the result of adding new components to a saved program will be a new program, the compiled version of this program can also be saved. To do this, use both the SAVE and the RESTORE options in the $LIMITS command. Unfortunately, however, this will cause the old saved program to be destroyed since the new saved program will be written onto the same file; until a new version of the system solves this problem, one possible solution (if you still want the old saved program) is to first copy the old saved program onto a second file.

30.3 Saving and Restoring the System During Execution (*SAVESYS and *RESTORSYS).

These actions allow you to save the state of the program during its execution and to restore a saved state during execution. This is a more general and more powerful capability than the SAVE and RESTORE options discussed above, since here it is not only the compiled version that can be saved but a partially run program which can be saved and restored at any time by using program actions.

(These actions are not yet implemented, and the exact details of the actions have not yet been determined.)

30.4 Examples.

1. The simplest case is executing a single program.

```
@RUN...
@XQT   SYS*A.B
$LIMITS   START = 2D, END = 5D;
(program)
$END;
@FIN
```

2. A program can be executed several times, being compiled and saved the first time. The saved version can be used only in this job.

```
@RUN...
@XQT   SYS*A.B
$LIMITS   SAVE, START = 2D, END = 5D;
(program)
$END;
@XQT   SYS*A.B
$LIMITS   RESTORE, CLOCK, START = 2D, END = 5D;
$END;
@XQT   SYS*A.B
$LIMITS   RESTORE, START = 1D, END = 10D;
$END;
@FIN
```

3. A program can be executed, its compiled version being saved permanently.

```
@RUN...
@CAT   YOURFILE.
@ASG,A  YOURFILE.
@SAVE,S   YOURFILE.
@USE   20.,   YOURFILE.
@XQT   SYS*A.B
$LIMITS   SAVE,...
(program)
$END;
@FIN
```

4. After job (3) above is run, the saved program can be executed as often as desired.

```
@RUN...
@ASG,AX   YOURFILE.
@USE 20.,   YOURFILE.
@XQT   SYS*A.B.
$LIMITS   RESTORE,...
$END;
@FIN
```

5. A saved program can be added to and executed. The new program can be saved, which destroys the original saved version.

```
@RUN...
@ASG,AX   MYFILE.
@USE 20.,   MYFILE.
@XQT   SYS*A.B
$LIMITS SAVE, RESTORE,...
(additional program components)
$END;
@FIN
```

31. Output From the System.

If the NOLIST option (see 3.1.3) is not in effect, then all of the output described below will be produced. If the NOLIST option is used, then beginning with the command following the $LIMITS command having NOLIST the user's source program will no longer be listed. If NOLIST is in effect at the end of the program, the information regarding the initial state of the system won't be printed. As this initial information is usually not very useful, you can get a program listing by first using a $LIMITS command without NOLIST and then ending your program with a $LIMITS command having NOLIST (followed by $END as always). Another way of just printing a program listing is by putting the program on a file and listing the file in a separate operation, and then using NOLIST when executing the program.

The program listing is just a copy of your source program, including comment cards. The cards are numbered sequentially, and these card numbers are referred to should an error occur. Interspersed with the listing are messages describing any errors which may have occurred (see 32.1).

Following the program listing, if NOLIST is not in effect, are lists of various items used in the simulation. Most of this information is not too useful as it concerns values of internal system pointers. The items listed, and their useful associated information, are:

1. A description of the time units (see 3.1.1 and 3.2) used: the number of basic time units in a simulated day, followed by the number of basic units in each of the defined units of time.
2. If the generative mechanism is being used, the input grammar and the dictionary list are printed.
3. The node index, which is a list of all nodes (see 4.) which have been defined, either explicitly in a nodes list or implicitly as a class or group name. The column labelled NODGRP will be non-zero if the node is also a group name, and the column labelled NODFLD will be non-zero if the node is also a class name.

4. The relation (see 5.) index, which is similar to the above node index. RTYPE is 1 for a transitive relation and is otherwise 0. RELGRP and RELFLD are non-zero for relations used as group names and class names, respectively.
5. If the generative mechanism is being used, a synonym list is printed.
6. The class list, which lists each class (see 6.) along with its initial elements. If CFLG is 1, the class is subscripted. Each class has a pair of parentheses following its name: if a class is not subscripted there will be nothing inside the parentheses; if a class is subscripted with particular subscripts, the class will be listed with each of its subscripts; if a class is subscripted but given no initial subscripts, an asterisk will appear in the parentheses. The initial elements of each class are listed in the order in which they are stored. This list represents the state of the classes after any initial actions in the $NETWORK section have been executed.
7. If there are any exclusions (see 7.), pointers relating to the exclusion list are printed.

At this point, the NOLIST option has no further effect and the remaining output is always produced.

First comes a count of the number of warnings and errors found during compilation. If there are any errors the program will now be terminated, although a program will run if only warnings were given.

Then, for each moment of simulated time in which some group is active, there is a listing of the change stack - all of the changes made in the semantic network at that time. For each triple in the stack, its assertion time, RV(see 19.), H(see 20.) and numeric values are given. A negated triple is preceded by 'NOT'. Information regarding class manipulation, and triples involved in predicate actions, are not shown in the change stack. See specific actions (chapter 12.) for more details.

Also, to the left of each triple in the change stack there are three numbers (which follow the sequence number of the triple) which explain how the triple got in the stack. SRC stands for source, LEX for lexical level, and PAR for parent.

When there is no expansion of predicate nodes, things are very simple: a primary triple has three zeros; a secondary triple's PAR is the sequence number of its primary triple, and its SRC is 4 to mark it as a secondary.

When predicate nodes are expanded (see 13.2) the numbers are used as follows: PAR is the sequence number of the triple which caused the triple having that PAR value to be printed; SRC is 1 for a triple which came from the expansion of the parent triple's subject node, SRC is 3 for a triple which came from the expansion of the parent triple's object node, and SRC is 5 for a triple which is a secondary triple of a parent which is a predicate triple; and LEX tells how many levels of expansion have occurred to produce the triple - a LEX of 2 means that this triple came from the expansion of a node whose triple itself came from the expansion of a node.

A sample change stack (time, RV, etc., are left off) may clarify things:

	SRC	LEX	PAR		
1	0	0	0	(JOHN LOVE MARY)	primary triple
2	4	0	1	(MARY PRETTY)	secondary of above primary
3	4	0	1	(LOVE WITH PASSION)	another secondary for triple #1
4	0	0	0	(JOHN KNOW THAT2)	primary triple
5	3	1	4	(MARY DATED TOM)	predicate of THAT2
6	5	1	5	(DATED TWICE)	secondary of above predicate
7	3	1	4	(MARY LOVE HIM)	another predicate of THAT2
8	3	1	4	(MARY KNOW THAT3)	a third predicate of THAT2
9	3	2	8	(JOHN LOVE HER)	only predicate of THAT3
10	5	2	8	(LOVE WITH PASSION)	secondary of above predicate
11	3	1	4	(MARY PRETTY)	a fourth predicate of THAT2
12	4	0	4	(KNOW DEFINITELY)	secondary of triple #4

As a further clarification, if this change stack were to come out in English, it might, with an appropriate grammar, come out as: John loves pretty Mary with passion. John knows definitiely that Mary dated Tom twice, that Mary loves him, that Mary knows that John loves her with passion, and that Mary is pretty.

If any traces (see 32.3.1) are active, their output will be shown preceding their associated change stacks. If the generative mechanism is being used, its natural language output is shown for each change stack listing, following that listing.

The end of the simulation, and the cause of its ending (reached the time limit, no more active groups, or reached an *END action; see 3.1.1 and 18.) is noted before execution terminates.

32. Debugging and Tracing.

While the system tries to give helpful hints when an error is found; it does not always succeed. There are basically two types of errors: syntax errors found during compilation, and errors found during execution. Syntax errors are by far the more common and will be discussed first. If a problem is seemingly inexplicable, there are traces which can give complete details about the execution of the simulation.

32.1 Syntax Errors.

Syntax errors are written out with the program listing, usually within a line or two of where the error occurred. The error message will give the card number on which that error occurred, a column on that card, and a message describing the error and giving any other pertinent information. Generally, the message given is enough to find the error; however, the information is not always completely reliable for several reasons:

1. One error may cause the system to think there are other errors, even if there aren't. Sometimes, error messages are given for correct lines that are 5 or more lines past a real error (which has caused an accurate error message). Therefore, if you find an inexplicable error message, especially one which just says "syntax error" and which is not far from another error, this second (and sometimes there are multiple inaccurate messages) error will often disappear if the first, real error is corrected.
2. The card number is generally correct, but it is possible for a previous card to be the real source of the error, as would be the case if a necessary semi-colon were left off.
3. The column number is sometimes not exact, pointing to the item following the incorrect one, pointing to a wrong item, or simply pointing to the end of the line.

4. The messages are generally correct, but the message "syntax error" is often given and is not very helpful. Look for a missing or incorrect delimiter, although the cause could be many other things.
5. If the error seems to make no sense, and was not caused by another error (see (1) above), then it is quite possible that the bug is in the system and not in your program.

32.2 Execution Errors.

These errors occur during the execution of a program and are printed out when they occur. The simplest type is actually a warning, typical warnings being for division by zero, or having a loop while expanding predicate nodes (see 13.2). If the message specifies an error and not just a warning, then although execution may continue the results may be undefined. Execution errors are of two types: an internal array has overflowed, or a syntax error which was undetected during compilation has now caused an impossible situation. In either case, the system program must be examined. If there are more than 50 warnings or more than 10 errors execution will be terminated.

One final note: execution errors are sometimes of the form STOP <number>. This indicates a system error and the program is immediately terminated.

32.3 Actions for Debugging and Tracing.

The five actions described in this section are very useful when trying to determine why a program produced the results that it did. It is often the case that a program will have no error in it as far as the system is concerned, but will still yield incorrect and mystifying output. The execution of the program can be traced, items can be printed, and specified portions of the network can be dumped by using the five actions described below. These actions are used as any other actions, and can thus appear in the <action list> of a rule or subrule.

32.3.1 Tracing (*TSET, *TSTOP, and *TSTART).

A trace is used to describe exactly what is occurring during the execution of a program. There are five different traces, each of which can be turned on and off during the simulation by using the three tracing actions.

Each type of trace is used to be describe one or more specific operations which take place during execution. Each type has a letter associated with it, and the five types are independent of each other, allowing you to trace only those operations which you are interested in.

The five traces, and what they produce messages for, are:

A: new time cycle; time group is scheduled or unscheduled for; entry into group, rule, switch, and loop; values a loop variable will take on

B: execution of all actions; assertion, negation, and setting of a triple, showing its associated values; results of class actions, showing lists of elements involved

C: Cumulative numeric value of subrule list; value of generated random number for subrule list

D: Value of each operand in a subrule

E: Result of network look-up in a subrule, giving status of triple and its associated values

The output produced by a trace is compacted to save space but still is generally easy to read. It gives the trace involved, a message, the appropriate information, and the location of the traced operation in the program (for example, its card number and subrule number, the name of a rule, etc.).

In a large simulation a trace can produce an enormous amount of output, so care should be taken in using traces. Since traces can be turned on and off dynamically, it might be possible to trace just that portion of a simulation that you are interested in, instead of tracing the entire simulation.

The actions which are used to activate and deactivate the traces all have an operand designated as <trace letters>. This consists of any of the five trace letters, in any order, with no embedded spaces. Each letter named is involved in the specified action. The three actions are:

1. *TSET <trace letters> = <line count>

This action initializes the line count of the specified traces. The <line count> specifies the number of times the trace can be used, and is useful in limiting the output produced by a trace. This action must be executed before a trace can be activated, as the line count for each trace is initially zero. In addition, if the <line count> is postive, the trace will be turned on. If the <line count> is negative the trace will initially be off, and the absolute value of the <line count> will be used. A <line count> of zero disables a trace.

Examples:

```
*TSET A = 100    trace A can be used 100 times and is turned on
*TSET BC = -9999   traces B and C can be used 9999 times and are
                   initially off
*TSET ABCDE = 1000   all traces are on and can produce a lot of output
```

2. *TSTOP <trace letters>

This action is used to stop (turn off) the specified traces. The line counts for the traces are not affected.

Examples:

```
*TSTOP B   stops trace B
*TSTOP ABCDE   stops all tracing
```

3. *TSTART <trace letters>

This action is used to start (turn on) the specified traces, and is used after a *TSTOP or an initially negative line count. A trace will start only if it has a non-zero line count.

Examples:

```
*TSTART B
*TSTART ABCDE
```

Finally, a trace action should probably be in a rule by itself in order to assure that it will be executed (a rule with subrules might not have its actions executed) and to allow the trace action to be easily removed when it is no longer needed. The $NETWORK section (see 8.) is often a convenient place for a trace action, although tracing won't start until the simulation actually begins.

32.3.2 *PRINT.

This action is used to print a message or to print the value of a specified field. Its form is:

*PRINT [(<maximum count>)] <print argument>
where
<print argument> ::= <general class reference> | <quoted string>

<maximum count> is optional, and specifies the maximum number of times this particular print action can be executed during the simulation. If not given, the number 32767 is used.

<general class reference> is described in 11.2. The current value of the field is printed, giving its constituent atoms.

<quoted string> is any string enclosed by single quotes (apostrophes), and is simply printed as given. It is useful in constructing your own personalized traces.

Examples:

```
*PRINT (50) 'JUST ENTERED GROUP MAIN'
*PRINT (100) X     (X might be a loop variable)
*PRINT PEOPLE
*PRINT FRIENDS(JOHN)
*PRINT (75) FRIENDS(ENEMIES(PEOPLE))
```

32.3.3 *DUMP.

This action is used to print the specified portion of the semantic network. Its form is:

*DUMP [(<maximum count>)][<general node field>]

The <maximum count> is used as in *PRINT above. If the <general node field> (see 11.2) is present, for each node in the field all of the primary triples in the network which have that node as a subject are printed. If this field is omitted, <u>all</u> nodes are examined, meaning that the entire network will be printed, which could be quite expensive.

Examples:

```
*DUMP JOHN
*DUMP FRIENDS(JOHN)
*DUMP(30) PEOPLE
*DUMP
```

A. Comparison With Older Versions.

Since it began, the system has evolved in a somewhat ad hoc manner, with new constructs being added to fit the requirements of particular programs. The writing of this manual has given us a chance to review the entire system, to tie loose ends together, and provide for an internal consistency that has been lacking. Therefore there has been something of a revolution, and old programs and descriptions are no longer valid.

In order to show how the system has changed, the version used for the Murder Mystery and the Propp and Levi-Strauss models will be considered the base system. The significant changes to this base will be described section by section, although it will be clear that some changes have had profound effects on the entire system, while other changes are either additions or local simplifications.

5. Relations used to be typed according to whether or not they took objects, and whether or not they were numeric. Now only the transitive type remains, which greatly simplifies things. Furthermore, any triple may now have a numeric value, not just those with numeric relations. This change affects the entire system and is probably the most important difference between the old and new versions.

5.2 Any relation can have a numeric lexical expression list, which has a minor syntactic change. Maximum and minimum values for the relation are no longer used.

6. A class of relations can contain any relation, instead of a single type of relation. The type of class had been specified by a number preceding the class name. A class name may now be used as a node.

12.1 All triple have numeric values and times, whereas an old numeric relation was necessary for a value but precluded a time. An object is always optional now. RV's and H values are new.

12.2 The assertion action now requires parentheses around the triple. No numeric information can be specified here as used to be the case, being allowed now only in the new *SET actions.

12.3 *NEG was done by using the keyword NOT in an assertion. Triples are no longer actually removed from the network, but are just marked as negated.

12.4 *SET is new, as numeric modification was done in assertions. The use of <assignop>s, and a <value> which can specify triples instead of just constants, is an important addition.

12.5.1 *SEC was *INSERT.

12.5.2 *NEGSEC was *DELETE. Triple are no longer actually removed from the network.

12.5.3 *SETSEC is new, following from *SET.

13.1 Predicate nodes existed, but only in a rudimentary state, so much of this chapter deals with new features.

13.3.1 *ADDP expands upon the old *DISCADD.

13.3.2, 13.3.3 *MOVEP and *PUSHP are new.

13.4 Only *DISCLEAR existed, which was a simpler version of *ERASEP.

13.5, 13.6, 13.7 *LOCATE, *MOVEPAR, and *EXPAND are new actions.

14. The major change in class actions is that the destination field can be a <general class reference> instead of just a <specific class reference>.

15. Some minor syntax changes were made.

19., 20. The RV and H value are new.

23.4 Looping through predicate lists is new, as are triple variables. This addition affects network actions and subrules, as triple variables can be used as triples and sentences.

26.1, 26.2 There is a major change here due to the delection of numeric relations. In the old version, a sentence having a numeric relation would return a numeric result, with no function being necessary. Also, the object field is always optional now.

26.2.4 There are some name changes, as described below for subrule functions. Multiple elements can now be specified in the subject and object fields when a function is being used, instead of just allowing single elements in all fields.

C. Index to Commands, Actions, Functions, and Syntactic Entities.

26.3.2 EQLP and NEQP are new.

27.3 NUMP is new. NUM was once called LENGTH.

27.4 SEC was VAL, which is confusing since there is a new, completely unrelated VAL function. As with sentences, SEC now always returns a logical value. All of the SECondary functions have had an outer set of parentheses around their arguments dropped.

27.5 DURSEC was DURVAL. In this and the following functions, all fields can now have multiple elements.

27.6, 27.7, 27.8 DEL, EVER, and RVAL are new.

27.9 VAL is new (not to be confused with the old VAL, which is now SEC) and is needed because there are no numeric relations.

29. The look-ahead rule is new.

30.2 Adding new components during a RESTORE is new.

30.3 *SAVESYS and *RESTORSYS are new actions.

B. Possible Additions to the System.

The system will continue to evolve, and these additions are only suggestions for the future. The numbers are the sections where the additions would have their major effects.

6. User-specified ordering of the elements of classes, possibly by using a system defined class of integers and having PREDecessor and SUCCessor functions.

12.4, 12.5.3, 19.1 [value] field of *SET having secondary triples.

13.1 Predicate relations as well as predicate nodes.

13.2 Some parameter which would control the expansion of predicate nodes on the change stack.

14. Intersection, union, NUM, and NUMP actions.

15. Deleting lexical items from atoms.

17. Passing parameters to subroutines.

23. Looping through secondary triples.

27. Getting at all deletion and assertion times, not just the last ones.

28. Triple subrule variables for use in predicate class sentences.

30.2, 30.3 Ability to SAVE a program on a particular file, so that saved version can be RESTORED.

There are also three major additions which would greatly enhance the power of the entire system: separate universes for different people, meta-compiling, and parsing. The latter two of these items are currently being implemented.