RELSIM - AN ON-LINE LANGUAGE FOR DISCRETE SIMULATION IN SOCIAL SCIENCES

Thesis by

Pericles Nicolaides

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1975

(Submitted December 5, 1974)

ACKNOWLEDGEMENTS

I would like to thank Dr. Frederick B. Thompson for his guidance and moral support at all stages of my graduate and thesis work; Dr. Thayer Scudder for his help and friendship, and the much needed non-computer scientist point of view; Dr. Peter Szolovits for his sterling example during the years we shared our office; and Ms. Mary Johnson for her uncomplaining decoding of my handwriting and its transformation into this thesis.

ABSTRACT

With the progress of science, our models of the world or any of its aspects become more and more complex, and therefore less and less susceptible to analytical solution. This is especially true in the field of social sciences, where statistical and stochastic processes are indispensable tools for model examination.

As a result, various simulation languages or packages have been developed to aid in the formulation and testing of such models. This thesis considers the directions that such languages have taken, and introduces a new such language, RELSIM, which attempts to meet present needs of the potential simulation designer. It features a simple structure that is also very flexible, and a timesharing environment, which allows dialogue, gaming and experimentation in the design and the actual simulation run.

A full description of RELSIM is given, with examples illustrating its use, and the implementation of the language on the REL system is also discussed.

TABLE OF CONTENTS

Chapter

I	Introduction	1
II	Description of the Language	13
III	An Inside View	56
	Bibliography	65

Appendix

A.	The RELSIM Syntax	67
B.	Routine Documentation	95

I. INTRODUCTION

One of the most useful methods man has devised for imposing order on his world perception has always been the isolation of appropriate subsets of that perceived world for simplification of observation, linguistic description, and behavior prediction; the appropriate subsets being ones whose interaction with the universe can be considered as occuring in a simple, describable way. These subsets are now commonly called systems.

This method, at first used in a nebulous manner, with the limits of the system intuited rather than described and observations made in a haphazard and incomplete manner (as for example in Aristotle's description of politics), was soon refined in its application on what we term the scientific field. Ptolemy's analysis of the motion of heavenly bodies, although later improved upon, was nevertheless quite precise in defining the system, its parts and their interactions; and its predictive power was quite adequate.

The apotheosis of the compartmentalizing technique occurs of course in modern electronics, where circuits are put together from integrated chips easily described by a few inputs and outputs and their internal behavior. This mirrors the current approach to system description, where an attempt is made to break up the system into simply describable components simply interrelated.

If, then, we regard a system as a set of elements, or subsystems, interacting in a regular way over time, we may proceed

to collect a body of information about these aspects of the system and try to analyze it to predict the system's behavior. This body of information we call our model of the system, and it should be such as to reflect the qualities we are interested in; that is, our model should be a useful simplification of the processes of the system.

The reasons for deriving a model for a system are basically two: Either the system considered is a nonexistent one which we are interested in constructing, in which case a model is necessary to ensure that the behavior of the system will be the desired one before the actual expenses of construction are met; or, the system is an existing one on which physical manipulation and experimentation is impossible or undesirable, in which case experimentation on the model will provide us with an understanding of the system that can be tested, as always, by prediction.

The derivation of a model is essentially made up of two processes: establishing a model structure and supplying the data. By structure we understand the determination of the boundary of the system and the establishment of entities, attributes, and activities in the system, while data provide the values of attributes and define the relationships involved in the activities. Our model of the system can be a physical one, as for example the models of circuits made up of mechanical analogs utilizing springs, masses and shock absorbers, which were used in the past by

electrical engineers; or, more commonly, a mathematical one.

In simple cases, mathematical models are susceptible to analytical solution of the equations governing their behavior. However, in the majority of cases, dynamic systems are too complex for such a description by solvable equations to be possible. With the increasing availability of digital computers, simulation has become the most important method of studying such complex mathematical models of systems.

In simulation we set up a model of the system at an initial point in time, and we follow the changes in the state of the model as they occur with the progress of time. There are two main types of simulation models, dependent upon whether we consider changes in the model to occur continuously or at distinct points in time.

In continuous simulations the system is represented by a set of finite difference equations approximating the differential equations governing smooth change. Thus if $X(t)$ is the vector of the state variables of the system (in which we are interested) at time t, and $Y(t)$ is the vector of variables describing inputs to the system, i. e. the effect of the environment, then the set of equations will be of the form

$$X(t+\Delta t)=f(X(t), X(t), X(t-\Delta t), \ldots, X(0), Y(t))$$

where f is the function specifying the behavior of the system.

In discrete simulation, on the other hand, the state of the system is regarded as changing in discrete steps. Thus the system is seen as consisting of elements which perform defined functions on the entities of the system. These elements, or subsystems, have a finite processing capacity and thus entities may have to be placed in queues pending their processing by each subsystem. In this type of simulation emphasis is placed on examining the capacities of the entire system and the manner in which entities get processed given the structure of the system. Stochastic processes and queuing techniques are the main tools used, and statistics on waiting times are provided to illustrate the performance of the system.

Although many systems may be modeled for either continuous or discrete simulation, the former is more natural in the case of electronic or mechanical feedback systems clearly defined by differential equations, where the latter is useful in traffic or job shop type systems where flow is governed by random variables of a particular density function and bottlenecks imposed by the system structure.

In the first efforts at system simulation, computer programs were designed to simulate a particular model; these programs were written in general purpose languages such as FORTRAN or ALGOL, or in assembler languages. However it soon became obvious that a great number of computations are common to most

models; a number of general purpose simulation languages were designed and on occasion implemented. Such languages enable the user to formulate and program a simulation with less effort and less need for familiarity with conventional programming and techniques of applied mathematics. DYNAMO and CSMP are notable such languages for continuous simulation. There is quite a number of discrete simulation languages in existence; table I. 1 contains a fairly comprehensive list. Of these we will concentrate on four: GPSS and SIMSCRIPT because they are the most often used and most developed languages, while their approaches are quite different; and SOL and SIMULA, which belong to a later generation and contain some interesting new trends.

We will use the following terminology in order to avoid the confusion caused by the different names employed in each of these languages to describe their features.

The fundamental objects of the system being simulated are the _entities_ of the simulation. The set of all entities of a particular type is an entity _class._ The properties of an entity are its _attributes._ We may think of an entity as a data record, with its attributes being the record fields. The data on the system are _system variables._ Entities sharing certain properties may be placed in an entity _list._

The state of the system may be changed by an _event_, which is thought of as an instantaneous occurence, or an _activity_, which is

TABLE I. 1 CURRENT DISCRETE SIMULATION LANGUAGES*

Simulation language	Computer language base	Originating organization
CLP	CORC	Cornell U.
CSL	FORTRAN	IBM U. K.
FORSIM	FORTRAN	MITRE
GASP	FORTRAN	U. S. Steel Corp.
MILITRAN	———	Syst. Res. Gp. for ONR
OPS	———	MIT
QUICKSCRIPT	———	Carnegie-Mellon
SIMPAC	SCAT	SDC
SIMSCRIPT	FORTRAN/Assembler	RAND
SIMTRAN	FORTRAN	MITRE
SIMULA	ALGOL	Norwegian Comp. Center
SOL	ALGOL	Burroughs
UNISIM	Assembler	Bell Labs

an occurrence that takes time. Note that in general an activity may be regarded as two events, marking the activity's beginning and its end.

Activities or events happen when certain conditions in the system are satisfied. One way of ensuring their proper sequencing is the use of a schedule of due occurrences with their times, the system log. Alternatively the main routine of the simulation may

*from Teichroew(14)

check all event or activity routines at each point in simulation time and execute those whose conditions are met.

GPSS is an entity oriented language. There are thirty-six pre-programmed events, and the user can put together a simulation by arranging appropriate events in a block diagram. The entities of the simulation are pre-defined, created in a particular block and they flow through the system as in a network, to be ultimately removed. System variables must also be identified with pre-defined concepts such as facilities, storages and logic switches. The sequencing of events is rigidly determined by the block diagram, and conditions of choice are reduced to alternative paths in that diagram governed by switches with possible random variable resolution. The selection of the next event is determined by interpretation of the diagram to find out what block each entity is due to enter. The gathering of statistics is controlled by certain of the block types, and so is the advancement of the simulation time.

In SIMSCRIPT on the other hand, the user may specify the structure of different types of entities and name them and their attributes. The events are also written by the user, in SIMSCRIPT language, as closed subroutines that are executed at particular instants of time. The sequencing of these events is controlled by the posting of event notices on a log, programmed also by the user to occur when certain conditions are met. This

posting of event notices can occur from within an event routine or through what is called an exogenous event, i.e. directly before the main routine takes control. This main routine, then, keeps track of executable events rather than the flow of entities through the simulation. System variables and lists must all be defined by the user at the beginning of the program. The SIMSCRIPT language includes several commands for I/O, plus a REPORT format which can be called to produce tables of statistics.

SOL was written as a generalization of the characteristics of the two previous languages. Thus the user does not define his types of entities as in SIMSCRIPT, but makes use of the existing concepts in the system such as variables and storages; and the sequencing is of the GPSS type, with the main processor monitoring the state of the system and triggering action when conditions are met. The viewpoint of SOL, however, can be regarded as activity oriented. The user programs activity rather than event routines. These routines may be executed over any amount of simulation time, through the use of a WAIT statement that stops and restarts processing of the activity according to the state of the system. Activities are then usually performed in a pseudo-parallel manner and as a result local variables are an important element in the writing of activity routines, as separate instances of the same activity may be processed in parallel. The syntax of SOL is ALGOL - like, although the commands available are

those specifically suited to, and common in most discrete simulation packages. SOL provides the user with many automatic statistical summaries besides supplying specific I/O commands, as in SIMSCRIPT.

The SOL viewpoint is also used in SIMULA, probably due to an extent to the fact that they are both based on ALGOL. SIMULA, however, represents a more thorough attempt at the implementation of a comprehensive language that can compete with SIMSCRIPT and GPSS. It is a true extension of ALGOL and as such incorporates all the features of a high-level language augmented by the pseudo-parallel processing of activities as in SOL. The objects simulated are again simple variables and arrays rather than user defined entities, while attributes and local variables are attached to activity definitions. Instead of a SOL type main processor, SIMULA emphasizes the concept of an event as a phase of an activity that occurs on a given instant of simulation time, and makes use of a log in which the events to be executed are posted. Another important improvement over SOL is the ability to reference variables local to an activity from a different activity through a CONNECTION statement. Both SOL and SIMULA benefit from the recursive capability of ALGOL.

Having examined the features of existing simulation languages we can see that an on-line environment would greatly augment the power of simulation as a user tool. Initialization of a

simulation model can then be accomplished on an experimental basis, with the user testing out parts of the simulation and modifying his routines. During the actual run the user can stop and examine the simulation, and if appropriate introduce new data; he may indeed redirect the course of the simulation by modifying the parameters, the log of events, the contents of entity classes, or by introducing new event routines; he can reinitialize the entire run, using the results of the interrupted run to refine his parameter values; if inclined, he may design an interactive run where the introduction of parameters from an outside source is expected, and use this for gaming or teaching techniques. However, the only attempt at implementation of an on-line discrete simulation language that we are aware of is the ongoing one at MIT with OPS, reported as an activity oriented language to be implemented on top of PL/1. Kiviat (9), in a recent paper, stresses the lack of such a language and suggests desirable goals.

RELSIM has been designed and implemented on the REL system to satisfy all the above requirements. In the development of the syntax our main concern has been simplicity of structure for the benefit of the user without loss of versatility and power. We believe that discrete simulation is a tool principally suited for the humane and social sciences, where mathematical description of models is usually impossible, and heavy emphasis is placed on the use of stochastic and statistical techniques. The user, then,

is most likely unfamiliar with current computer languages and not interested in acquiring a knowledge of the field.

With this in mind, we designed RELSIM as an event oriented language, believing that the activity orientation of SIMULA and OPS, though aesthetically pleasing to the computer scientist, would be more confusing to the social scientist programmer. Unlike SIMSCRIPT, however, RELSIM allows variables local to events and the passing of parameters from one event to another, thus achieving all the versatility of activity oriented languages. The ability to define different types of entities with their attributes was considered to be a conceptually useful feature and included. Also, to facilitate experimentation we enable the user to delete any kind of object of the simulation, including entities, classes, lists, system variables, the log in part or as a whole, and event routines. RELSIM includes random number generators from all common probability distributions, and also enables the user to define any density function in a simple manner and thereafter obtain random variables governed by that function. The definitional capability of REL-English is also present in RELSIM, so that the user may extend or simplify at will the syntax at his disposal.

Implementation of the language on the REL System proved to be a natural task. The programming of additions to the features of the language is, and we believe will be, very effortless. The user furthermore benefits from the ability to input initialization

parameters in the batch mode and the option of saving multiple runs of the same simulation, so that he may check the state of the system at different points in simulation time, or the effect of different initial conditions. Also, besides regular output commands, RELSIM features a RECORD statement that transforms the results of the simulation into an REL-English data base, to be queried conversationally, with all the power of that data management language at the user's disposal.

The declarative statements and the event routine command statements of RELSIM are simple and almost conversational in structure. They consist of a statement verb, followed by the objects the verb refers to, followed by modifying clauses that can be strung in any order and repeated any number of times. Thus, acquainted with the few verbs and clause prepositions plus the arithmetic expression capability of the language, the user may attempt writing a simulation and teach himself the refinements of the language as he goes along. We shall proceed to describe the syntax of RELSIM.

II. DESCRIPTION OF THE LANGUAGE

In order to describe the RELSIM syntax we will first undertake an exposition of the language's structure, and follow by some clarifying simple examples of its use. For our description we indicate language strings in capitals; metalanguage description by lower case; optional strings in bracket enclosure; alternatives by double brackets with the default parameter underlined.

A. Declaration statements

These statements are used to reserve a location for a system variable, simple or array, initialize a list or an entity class, or specify an event routine. A declarative statement may not be used inside an event routine. In specifying a simple system (entity or numerical) variable, the statement is of the form

DECLARE "name" [{REAL / ENTITY}] [=numerical or entity expression]

where name may be any string beginning with a letter and not ending in a blank. If desirable, the variable is initialized at this point to the current value of the expression used. Numerical and entity expressions will be described later on. If a system array variable is to be initialized, the statement will be

DECLARE "name"(num. expression, num. expression)

for the two dimensional case.

In initializing a list, the declarative statement will be of the

form

DECLARE"name" [{REAL / ENTITY}] {LIST / STACK} [=list expression]

where REAL or ENTITY specifies the nature of the list elements. If LIST is specified the elements are accessed from top down, i. e. a FIFO list is obtained; STACK results in the reverse ordering, i. e. a LIFO list. It can also be initialized with the contents of another list.

When initializing an entity class one can name up to 40 attributes of the class entities and also assign initial values to any number of them. Notice that these initial values are calculated at the time of creation of each entity in the class. The statement is of the form

DECLARE "name" CLASS [WITH"name" [{REAL / ENTITY}] [{LIST / STACK}] [=expression]] [AND "name" [etc.]] etc.

In writing an event routine we also use the declarative format in the following form

DECLARE {"name" / name} EVENT: command statement[;command st] .... [;com. st.].

where any number of command statements, separated by semi-colons, may be used. The name of the event routine is not placed in double quotes if it has already been set in a GENERATE statement of a previously written event routine; for it is often necessary to refer to as yet unwritten event routines, and assign a name to them ahead of time.

B. Numerical and entity expressions

Let us first consider numerical expressions. There are the following elementary numerical forms:

(i) Constants, e.g. 35, 3.27, 2.4305 E09.

(ii) System variables expressed by their declared name, e.g. X, LENGTH, DAY OF BIRTH.

(iii) Numerical attributes of entities. Thus if GEORGE is an entity and INCOME is its attribute, then INCOME OF GEORGE or INCOME(GEORGE) is a valid numerical form.

(iv) Elements of a real list or an array. Thus if PRIME is a list then the following forms are valid: 34TH PRIME or PRIME(34); LAST PRIME or PRIME(L); PRIME(L-34);CURRENT PRIME;NEXT PRIME; PREVIOUS PRIME;RANDOM PRIME. The current list element is the last one placed in the list until set otherwise; NEXT and PREVIOUS refer to this current element above, but can also be used on any element, e.g. NEXT (PRIME(34)), and thus repetitively, e.g. NEXT(NEXT PRIME). If ARGH is a two dimensional array, we can refer to ARGH(2, 3) etc.

(v) CURRENT TIME and SEED, predefined system variables. The latter is used to calculate uniformly distributed random variables in (0, 1) that are used for

stochastic variates.

(vi) The number of elements in a list or entity class. Thus if BOY is a class, we may write NUMBER OF BOY or NUMBER OF BOYS.

These elementary forms may be combined by addition, subtraction, multiplication, division and exponentiation into expressions. Unary plus and minus are also available. Most common arithmetic functions are provided; a list follows:

FUNCTION	FORM
Square root	SQRT
Exponential	EXP
Logarithm (base e)	LN
Logarithm (base 10)	LOG
Sine	SIN
Cosine	COS
Tangent	TAN
Cotangent	COTAN
Factorial	FACT
Integer Part	IP
Fractional Part	FP
Absolute Value	ABS
Sign	SIGN
Maximum (a, b)	MAX(A, B)
Minimum (a, b)	MIN(A, B)
a Mod b	MOD(A, B)

The following random number generators for continuous probability distributions are available:

(i) Uniform distribution over interval (A, B) : UNIFORM(A, B).

(ii) Normal distribution, with density function

$$f(x) = \frac{1}{S\sqrt{2\pi}} e^{-1/2\left(\frac{x-M}{S}\right)^2}$$

where M is the mean and S is the standard deviation: NORMAL (M, S).

(iii) Exponential distribution, with density function f(x)=a*exp(-a*x) where a>0 and x>0: EXPD(a)

(vi) Any distribution with density function f(X): DENSITY(f(X)); e. g. if f(X) is the normal distribution with M=10 and S=3 we could write DENSITY (EXP((-1/2)*(((X-10)/3)**2))/3*SQRT(2*PI)).

Also, the following generators for discrete probability distributions may be used:

(i) Uniform distribution between A and B: RANDOM (A, B).

(ii) Binomial distribution with probability function $f(x)=\binom{n}{x}p^x(1-p)^{n-x}$ with $0\leq p \leq 1$ and $x\geq 0$: BINOMIAL (n, p).

(iii) Poisson distribution with probability function

$f(x)=e^{-\lambda}\frac{\lambda^x}{x!}$ where $\lambda>0$ and $x>0$: POISSON (λ).

(iv) Pascal distribution with probability function

$f(x)=\binom{k+x-1}{x} p^k(1-p)^x$ where $x\geq 0$ (k is an integer denoting the total number of successes out of k+x trials) : PASCAL (k, p).

(v) Any distribution with probability function f(X): FREQUENCY (f(X)) e. g. FREQUENCE (.04 IF X=1, .05 IF X=3, .01 IF X=7, 0 OTHERWISE) (see conditional numerical expressions immediately below). Note that we must have $\sum^{X} f(X)=1$.

A numerical expression may also be dependent upon boolean conditions in the following manner

num. expression IF boolean expression, n. e. IF b. e., ... , n. e. OTHERWISE

The boolean expressions available are discussed in the next section.

We next consider the ways in which entities may be expressed. In an analogous way to numerical forms, we may access them

(i) by their declared name, e. g. GEORGE, BOSTON

(ii) as entity list or class elements. Thus if PERSON is a class or list, the following forms are valid:

34TH BOY or BOY(34); LAST BOY or BOY (L); BOY(L-34); CURRENT BOY; NEXT BOY; LAST BOY; RANDOM BOY; and NEXT (NEXT(CURRENT BOY)) etc., as we have discussed in numerical forms.

(iii) as entity-valued attributes of entities. Thus if MOTHER is an attribute of the entity GEORGE, then MOTHER OF GEORGE or MOTHER(GEORGE) may be used. Note that any level of indirection is permissible, e.g. MOTHER OF MOTHER OF MOTHER OF GEORGE.

C. <u>Boolean expressions</u>

These expressions are utilized in conditional numerical expressions, conditional clauses and scratch list generation (see next section). There are four elementary forms:

(i) (numerical expression) (comparator) (numerical expression) as for example 43>=AGE OF GEORGE, LOG(X)¬=0.

(ii) (entity expression) $\left\{ \begin{matrix} = \\ \neg= \end{matrix} \right\}$ (entity expression), e.g. MARSHA=MOTHER(GEORGE)

(iii) (entity expression) $\left\{ \begin{matrix} \text{IN} \\ \text{IS IN} \end{matrix} \right\}$ $\left\{ \begin{matrix} \text{list} \\ \text{entity class} \end{matrix} \right\}$ or, (numerical expression) $\left\{ \begin{matrix} \text{IN} \\ \text{IS IN} \end{matrix} \right\}$ (list), e.g. GEORGE IS IN ARMY, 127 IS IN PRIME.

(iv) EXISTS $\left\{ \begin{matrix} \text{list} \\ \text{entity class} \end{matrix} \right\}$, e.g. EXISTS BOY.

The comparators allowed are =, >, <, >=, <=, and their negations, ¬ = etc. We may use IS for =, NOT

for ¬ , and IS NOT for ¬ =.

By combining the above forms with logic operators we arrive at complex boolean expressions. The permitted logic operators are NOT or ¬ (unary); AND, OR, XOR (binary). Also,NEITHER (meaning NOT) and NOR (meaning AND NOT) may be used. For example, we may say

AGE OF GEORGE=24 AND MOTHER OF GEORGE=MARSHA

NEITHER AGE OF GEORGE IS 24 NOR MOTHER OF GEORGE IS MARSHA

¬(EXP(A)>=12.7 OR SIN(E)¬=0.5) XOR 127 IN PRIME

D. <u>Scratch lists</u>

Quite often we may wish to access all elements of an entity class or list that meet certain requirements. In RELSIM, we may, by expressing these conditions, form a list with no name to reference it (which gets destroyed at the completion of the sentence). Thus the following forms yield such lists:

(i) (entity class) WITH (boolean expression on its attributes). For example BOY WITH MOTHER= MARSHA AND FATHER=JOHN AND AGE=12.

(ii) $\left\{ \begin{matrix} \text{list} \\ \text{entity class} \end{matrix} \right\}$ logic operator $\left\{ \begin{matrix} \text{list} \\ \text{entity class} \end{matrix} \right\}$; e.g. BOY OR GIRL. We may use IN instead of AND; BOYS IN ARMY means the same thing as BOY AND ARMY.

The above may be combined into expressions of arbitrary complexity. If part or all of the information in such a list should be saved, a declared variable or list may be set accordingly.

E. Event notices

In order to modify the log, we need to be able to make reference to specific event notices. The forms available are the following:

(i) Expression by given name, e. g. GEORGE'S BIRTH, LINE1.

(ii) Expression by location in the log, e. g. 8TH EVENT or EVENT (8); LAST EVENT or EVENT (L); EVENT (L-3); RANDOM EVENT.

(iii) Expression by location in the log of a particular type of event routine. Thus if WEDDING is an event routine, we may use 8TH WEDDING etc. as in (ii) above.

(vi) We may refine (ii) and (iii) above by specifying the simulation time at which the event is to be executed, e. g. RANDOM EVENT AT TIME (num. expression).

F. Command statements

Each command statement may be typed in as a REL sentence and executed immediately, or as part of an event declaration statement. The basic command statements are the following:

(i) CREATE (entity class name) [NAMED "name"]

or,

CREATE (numerical expression) (entity class name)

One or more elements of a named class are created; if desired, a single such created entity may be given a name.

(ii) GENERATE (event routine name) [NAMED "name"]

or,

GENERATE "name" [NAMED "name"]

The simulation log is updated with an instance of the event described by the routine named. In the first instance, the routine has already been declared; in the second, it will be declared later. Clearly, the second form may not be used as a direct sentence, since it cannot be executed before the declaration of the event routine.

(iii) SET variable=expression

SET array=num. expr., ... , num. expr.

The value of a system variable, list or attribute of an entity is updated. In the second form, an array's entries are filled sequentially with the first index being the slowest - changing.

(iv) NAME $\left\{ \begin{matrix} \text{entity} \\ \text{event notice} \end{matrix} \right\}$ "name"

An entity or an event notice is given a name for future reference.

(v)

DELETE { entity | system variable | class | list | array | event routine | event notice }

DELETE LIST list name

DELETE CLASS class name

DELETE ALL { CLASSES | EVENTS | CONSTANTS | ARRAYS | LISTS }

DELETE LOG

These statements delete parts of the simulation as desired and remove their names from the lexicon. Note that DELETE (list) will empty the list of its contents, while DELETE LIST (list) will get rid of the list itself; the same holds for an entity class. DELETE ALL EVENTS gets rid of the routines, while DELETE LOG gets rid of the event notices. DELETE

CONSTANTS gets rid of all the system (numerical or entity) simple variables.

(vi) RESET

This command empties the log, all classes and lists, and resets the simulation time to 0.

(vii)

PUT { entity expr. / num. expr. } [{ { BEFORE / AFTER } { entity expr. / num. expr. } / IN PLACE num. expr. }] IN list

A list is updated by addition of an element. If the optional phrases are not used, the element will be placed as last in a list or first in a stack. IN PLACE, when used, specifies the location in the list the element will occupy; if the number exceeds the elements present in the list, the element is placed last.

(viii) REMOVE list (numerical expression)

REMOVE { entity expr. / num. expr. } FROM list

An element is removed from the list. If there are multiple instances of that element, the first instance is removed. If there is no instance, no action occurs.

(ix)

WRITE { entity / sys. variable / event notice / C"character string" } [, { . . . }] . . . [, { . . . }]

WRITE $\left\{ \begin{array}{l} \text{class} \\ \text{list} \\ \text{array} \\ \text{event routine} \\ \text{LOG} \end{array} \right\}$

In the first form, a line of output may be formed. If the total number of characters exceeds those available in a line, the remaining characters are lost. In the second form a multi-line output results and thus no concatenation of quantities to be written is permissible.

(x) DO ; comm. statement;. . . . ;comm. statement; END

The statements inside the "do loop" are executed as a unit, and any clause attached to the DO sentence applies to all of them. For example,

SET X=0;DO UNTIL X=10; SETX=X+1;WRITE X;END

operates as a conventional FORTRAN do loop.

(xi) SELECT ; (numerical expression)(command statement) ;. . . . ;(n. e.)(s. c.);END

One of the statements in the sequence is selected at random and executed, with a probability of selection according to the number prefacing the statement. If the total of these numbers is less than 1, the remaining probability is assigned to no action; if the total is more than 1, supernumerary statements are ignored,

and possibly the probability of the last statement not ignored is adjusted so that a total of unity is achieved.

(xii) GO

This statement may be used only directly. It results in the transference of control to the main program, and the events posted on the log begin to be executed.

(xiii) PAUSE and CONTINUE

The first statement may not be used directly. When encountered during the execution of an event, it causes control to be returned to the user. If, after various direct commands, the user types CONTINUE (which may be used only directly), the simulation proceeds exactly where it stopped, i. e. with continued execution of the event under way. If, on the other hand, GO is used, the half-finished event gets flushed and the simulation proceeds with the next posted event in the log.

(xiv) EXIT

This statement may be used only directly. It stops processing in the simulation language version and returns the user to the REL command language.

(xv) RECORD SIMULATION

This statement may be used only directly. It causes an REL English data base to be formed out of the results of the simulation. If the name of the simu-

lation language is X, then the version of REL English containing this data base will be XDATA.

(xvi) DEF: name: part of speech

This is the standard REL definitional statement.* It may be used only directly. Any construct of the language may be given a name that can be used thereafter instead of that construct. For example, once we have defined

DEF: C: CONTINUE

we may use this shorthand thereafter. Also, due to this statement's variable capability, we may define

DEF: GAMMA("A", "K"): DENSITY((("A"**"K")*("K"-1))*EXP(-"A"*X)/FACT("K"-1)) and have random variables from a Gamma distribution at our disposal.

WRITE DEF OF name

DELETE DEF OF name

are also available as direct statements to the user.

There are several modifying clauses that may be used with statements (i) - (xi) above. They may be used repeatedly or strung together, unless otherwise specified in their description here.

---

* See Thompson (15)

(i) WITH attribute=expression [AND att. =expr.]. . . . .

[AND att. =expr.]

This clause may only occur as the first modifier of a CREATE statement. It initializes as many attributes of the created entity (ies) as desirable.

(ii) AFTER INTERVAL num. expression

The statement so modified will be executed after the specified interval of simulation time.

(iii) FOR { entity | class | entity list }

In the first instance, the current entity of the class to which the entity referenced belongs gets set to this entity, and the statement modified is executed. In the second instance, the current entity of the class is successively set to each of its elements and the statement modified is executed for each such change. In the third instance the list is broken down into sublists of entities belonging to the same class, and execution occurs as if there were that many successive FOR statements; in each of those, the current entity of the class gets set successively to each element of the sublist, and the statement modified is executed each time. After execution all current entities regain

their former values.

(vi) IF boolean expression

The statement will be executed only if the boolean is true. The only clause that may follow an IF clause is an AFTER clause or another IF clause. When several statements modified by final IF clauses are strung together and followed by the construct

ELSE command statement

then only the first encountered statement with a true boolean will be executed.

(v) FOR INTERVAL num. expression

This clause may only be used as the first modifier of a CREATE, GENERATE, SET or PUT statement. It results in the opposite action (DELETE, SET to previous value, or REMOVE) occuring after the specified simulation interval.

(vi) UNTIL boolean expression

This clause may only be used once in a statement. The statement will be executed over and over until the condition is met. Used with the GO statement, this clause may define the duration of the simulation run.

(vii) AT INTERVALS OF num. expression UNTIL boolean expression

This clause may be used only once in a statement;

and it cannot appear with (vi) above in the same statement. Its result is clear.

G. Local variables and pass parameters

When writing an event routine, we may desire to create and name temporary variables, accessible only to the event being run and deleted after its execution. The definitional statement for local variables has the form

LOCAL name(type)[, name(type)] . . . . . [, name(type)]

and must be the first statement in an event routine; it may only occur once in the routine. The type of variable assigned to each name is given by an abbreviation as follows:

A(number, number)	array
LE	entity list
LR	real list
SE	entity stack
SR	real stack
VE	entity system variable
VR	real system variable

and these are the only types of variables allowable as local.

In order to have communication between events we must have an adequate parameter passing mechanism. When an event generation statement occurs, it may contain any number of parameters to be passed to that event from the event in which the statement occurs, including variables local to this latter event. The basic generation statement format will be

GENERATE {name | "name"} (name[, name] . . . . [, name])

where the variables to be passed are named inside the parentheses after the event routine name.

Corresponding to this statement, the event to be generated must contain the following statement

PASSED name(type)[, name(type)] . . . [name(type)]

which must either be the first statement of the event, or, if a LOCAL statement exists, the second statement. Any local name may be given to the parameters passed, but they must appear in the PASSED statement in the same order in which they were given in the GENERATE statement. Their type must be included and must coincide with that of the variable passed in the GENERATE statement. All of the types available for the LOCAL statement may be used, as well as the following:

AE	attribute, entity valued
ALE	attribute, entity list
AR	attribute, real
ARL	attribute, real list
C	class
EN	entity
EP	event notice (posting)
ER	event routine
R	real number

Thus, if e. g. an event routine named KELP includes the following statement

PASSED X(AE), R(R), Y(EN), Z(LR)

the declaration statement may be

GENERATE KELP(MOTHER, 3. 4, GEORGE, PRIME)

H. Examples

Let us first consider a simple shop simulation, it being an old favorite. Let us say that a machine is turning out parts at the rate of one every 5 minutes. There is a conveyor belt, and three inspectors in a line. Each part takes 2 minutes to reach the first inspector. If he is busy, it moves on to the second inspector taking another 2 minutes, and similarly for the third inspector. If all three are busy, the belt circles back to the first inspector and it takes the tool 6 minutes to come around to him again. The first inspector takes 4 ± 3 minutes to check each tool; the second takes 5 ± 3, and the third takes 5 ± 2 minutes. Each of them rejects 10% of the parts. We would like to know the average transit time of an accepted tool and the percent time that each inspector is busy; and we would like to run the simulation until 1000 parts have been judged acceptable.

The following RELSIM program, which simulates this shop situation, needs no commentary.

```
DECLARE "TOOL" CLASS WITH "CREATION TIME"=CURRENT TIME.
DECLARE "STATION" CLASS WITH "STATUS"=0
                              AND "TOTAL BUSY TIME"=0.
CREATE 3 STATIONS.
DECLARE "TOOL COUNT" REAL=0.
DECLARE "TOTAL TRANSIT TIME" REAL=0.
```

```
DECLARE "TOOL INTRODUCTION" EVENT:
    DO IF TOOL COUNTER<1000;
        CREATE TOOL;
        GENERATE "ENTER STATION"(STATION(1),CURRENT TOOL)
                                                AFTER INTERVAL 2;
        GENERATE TOOL INTRODUCTION AFTER INTERVAL 5;
    END;
    ELSE GENERATE "WRAP UP".

DECLARE ENTER STATION EVENT: PASSED CSTATION(EN),CTOOL(EN);
                             LOCAL BUSY TIME(VR);
    DO IF STATUS(CSTATION)=0;
        SET BUSY TIME=UNIFORM(1,7) IF CSTATION=STATION(1),
                      UNIFORM(1,8) IF CSTATION=STATION(2),
                      UNIFORM(3,7) OTHERWISE;
        SET STATUS(CSTATION)=1 FOR INTERVAL BUSY TIME;
        GENERATE "EXIT STATION"(CTOOL)
                             AFTER INTERVAL BUSY TIME;
        SET TOTAL BUSY TIME(CSTATION)=TOTAL BUSY
                                   TIME(CSTATION)+BUSY
                                                  TIME;
    END;
    ELSE DO;
        GENERATE ENTER STATION(NEXT STATION,CTOOL)
```

```
                                      AFTER INTERVAL 2
                                      IF EXISTS NEXT STATION;
    ELSE GENERATE ENTER STATION(STATION(1),CTOOL)
                                           AFTER INTERVAL 6;
  END.

DECLARE EXIT STATION EVENT: PASSED CTOOL(EN);
    SELECT; 0.9 DO;
                 SET TOOL COUNT=TOOL COUNT+1;
                 SET TOTAL TRANSIT TIME=TOTAL TRANSIT
                                     TIME+CURRENT TIME-
                                     CREATION TIME(CTOOL);
                END;
    END;
    DELETE CTOOL.

DECLARE WRAP-UP EVENT: WRITE C"TRANSIT TIME AVERAGE = ",
                              TOTAL TRANSIT TIME/1000;
  WRITE C"PERCENT BUSY, 1ST STATION = ",
           TOTAL BUSY TIME(STATION(2))/CURRENT TIME)*100;
  WRITE C"PERCENT BUSY, 2ND STATION = ",
           TOTAL BUSY TIME(STATION(1))/CURRENT TIME)*100;
  WRITE C"PERCENT BUSY, 3RD STATION = ",
           TOTAL BUSY TIME(STATION(3))/CURRENT TIME)*100.
```

GENERATE TOOI INTRODUCTION.

GO.

As a second example, let us consider a simulation of the communication of a computer central processing unit with remote terminals. This example is used by Knuth (11) to illustrate the features of SOL.

Let us say that there are four _processor buffer unit pairs_ that handle the input and output between the computer and _site buffers_, where each of the latter controls one or more _terminals_, or typewriters. We will consider what happens at one pair of the processor buffer units in detail, and do a rough simulation of the activity of the other three.

Thus, the processor buffer units we are considering handle three site buffers; site buffer (1) controls three terminals, for which we will use the indices 1, 3, 5; site buffer (2) controls two terminals indexed 2 and 4; and site buffer (3) controls only terminal (6).

As people walk in and attempt to use each terminal they wait in line until that terminal is free. The terminals are far enough from each other that people do not attempt to find another, free, terminal.

Let us say there are three kinds of messages that these people send. Message type A requires 250 msec. of computing and 3 response words from the computer, and it is sent by users

20% of the time; message B needs 300 msec. and 4 response words, and is sent 50% of the time; and message C, sent 30% of the time, takes 400 msec. and 5 response words.

The processor buffer units scan the six terminals sequentially for input; when a positive response is observed at a site buffer, the message is transferred from site to processor buffer and then to the computer; after processing, the appropriate number of words is sent to the site buffer and typed, one word at a time, at the appropriate terminal.

We will comment further on details of the simulation as we write events. Initialization is handled when it is needed rather than all at once in the beginning. Let us first consider the action of each person.

```
DECLARE "PERSON" CLASS WITH "ENTRANCE TIME"=
                                        CURRENT TIME.
DECLARE "TERMINAL" CLASS WITH "QUEUE" ENTITY
      LIST AND "STATUS"=0 AND "MESSAGE" AND "SB"
      AND "TABLE" REAL LIST.
CREATE 6 TERMINALS.
SET SB(TERMINAL(1))=1.
SET SB(TERMINAL(2))=2.
SET SB(TERMINAL(3))=1.
SET SB(TERMINAL(4))=2.
```

```
SET SB(TERMINAL(5))=1.
SET SB(TERMINAL(6))=3.
```

Thus far we have created the terminals and assigned to them, in the attribute SB, the site buffers that control them. Their queues, where persons wishing to use them congregate, are empty. Their status will be denoted as follows: 0 means the terminal is free; 1 means that a message is being typed; 2 that the message has been completed; and 3 that the answer message may be typed. The MESSAGE attribute will be set to 1, 2, or 3, depending on the type of message that the user sends. We also initialized the class of users, with an attribute that will save their entrance time so that we find out the length of their stay in the system. The attribute TABLE of each terminal collects those statistics.

```
DECLARE "USER ENTRANCE" EVENT:  LOCAL X(VR);
   SET X=RANDOM(1,6);
   CREATE PERSON;
   PUT PERSON IN QUEUE OF TERMINAL(X);
   GENERATE USER ENTRANCE AFTER INTERVAL
                                     RANDOM(0,5000);
   GENERATE "MESSAGE TRANSMISSION"(X).
```

This event describes the entrance of a user, it causes another such entrance within 5000 units of simulation time, which here represent milliseconds; and it sets up the next event, the

sending of a message.

```
DECLARE MESSAGE TRANSMISSION EVENT:  PASSED X(VR);
  DO IF STATUS OF TERMINAL(X)=0;
     SET STATUS OF TERMINAL(X)=1;
     SET MESSAGE OF TERMINAL(X)=FREQUENCY(.2
        IF X=1,.5 IF X=2,.3 IF X=3, 0 OTHERWISE);
     DO AFTER INTERVAL RANDOM(6000,8000);
        WRITE C"TERMINAL (", X, C") SENDS
           MESSAGE", MESSAGE OF TERMINAL(X),
           C" AT TIME ", CURRENT TIME;
        SET STATUS OF TERMINAL(X)=2;
     END;
  END.
```

When the terminal is free, the user that is first in line types in one of the three types of messages. It takes him between 6 and 8 seconds to type. When it is completed, we write out a line for our reference and change the terminal status accordingly.

```
DECLARE "ANSWER RECEIVED" EVENT:  PASSED X(VR);
    MAKE 1ST QUEUE OF TERMINAL(X) CURRENT PERSON;
    REMOVE CURRENT PERSON FROM QUEUE OF
                                       TERMINAL(X);
```

```
PUT CURRENT TIME-ENTRANCE TIME OF CURRENT
     PERSON IN TABLE OF TERMINAL(X);
DELETE CURRENT PERSON;
WRITE C"TERMINAL(", X, C") RECEIVES REPLY AT TIME ",
                                   CURRENT TIME;
GENERATE MESSAGE TRANSMISSION(X)
                  IF EXISTS QUEUE OF TERMINAL(X).
```

When status of the terminal is set to free by an event below, this event is also generated, which takes the user out of the system and prepares the terminal for use by the next person in line, if any. We write out another line for reference, showing when the answer was received.

The above three events handle the action at the terminals. We now proceed to simulate the pair of processor buffer units.

```
DECLARE "LINE"=0.
DECLARE "QUEUE OF LINE" LIST.
DECLARE "SCAN" EVENT: PASSED T(VR);
    SET T=T+1; SET T=1 IF T>6;
    PUT T IN QUEUE OF LINE; PUT 0 IN QUEUE OF LINE;
    GENERATE "SEIZE LINE" AFTER INTERVAL 1.
```

The processor buffer unit, having considered terminal T, proceeds to look at terminal T+1 (mod 6). The cyclic scanning process

takes 1 msec. Then an attempt is made to seize the long distance communication line. Such seizure attempts must be handled by queuing when the ling is busy, i.e. LINE=1. This is done by double entries in queue of LINE; the first entry is the number of the terminal we are working on, and the second entry denotes the direction of the message to be transmitted, 0 denoting a message from the terminal and ≠ 0 an answer from the computer.

```
DECLARE SEIZE LINE EVENT:
   DO IF LINE=0;
       SET LINE=1;
       GENERATE "SEIZE BUFFER IN"(1ST QUEUE OF LINE)
                AFTER INTERVAL 5 IF 2ND QUEUE OF LINE=0;
       ELSE GENERATE "SEIZE BUFFER OUT"(1ST QUEUE OF
                LINE, 2ND QUEUE OF LINE) AFTER INTERVAL 5;
     END.
```

Thus the above event controls the access to the line. Now when the processor buffer unit finds the line free, it must then consider if the site buffer of the terminal scanned is free. It takes 5 msec. for a control signal to propagate to the buffer.

```
DECLARE "SITE BUFFER" LIST.
SET SITE BUFFER(1)=0.
SET SITE BUFFER(2)=0.
```

```
SET SITE BUFFER(3)=0.
SET SITE BUFFER(4)=0.
SET SITE BUFFER(5)=0.
SET SITE BUFFER(6)=0.
DECLARE SEIZE BUFFER IN EVENT:  PASSED T(VR);
    REMOVE 2ND QUEUE OF LINE;
    REMOVE 1ST QUEUE OF LINE;
    DO AFTER INTERVAL 80 IF SITE BUFFER(SB OF
                                    TERMINAL(T))=1;
        SET LINE=0;
        GENERATE SEIZE LINE IF EXISTS QUEUE OF LINE;
        GENERATE SCAN(T);
    END;
    ELSE DO;
        SET SITE BUFFER(SB OF TERMINAL(T))=1;
        GENERATE "TEST TERMINAL"(T) AFTER
                                    INTERVAL 15;
    END.
```

If the site buffer is busy, the processor buffer waits 80 more msec., receiving no signal back, and then releases the communication line (which, if there is a queue, will be seized immediately) and proceeds to scan the next terminal. If the site buffer is free, it is seized by the processor buffer. It takes 15 milliseconds to send the number T down the line.

```
DECLARE TEST TERMINAL EVENT:  PASSED T(VR);
    GENERATE "MESSAGE TRANSMISSION" AFTER INTERVAL
        395 IF STATUS OF TERMINAL(T)=2;
    ELSE DO AFTER INTERVAL 65;
        SET LINE=0;
        GENERATE SEIZE LINE IF EXISTS QUEUE OF LINE;
        SET SITE BUFFER(SB OF TERMINAL (T))=0;
        GENERATE SCAN(T);
    END.
```

The site buffer takes 65 milliseconds to determine whether the terminal is ready to transmit. If not, the line and the site buffer are released and the next terminal is scanned. If it is, it takes the site buffer 225 msec. total to get ready to transmit the message, and 170 to send it.

```
DECLARE "COMPUTER"=0.
DECLARE "QUEUE OF COMPUTER" LIST.
DECLARE MESSAGE TRANSMISSION EVENT:
    PASSED T(VR);
    SELECT;
        .02 GENERATE MESSAGE TRANSMISSION AFTER
                                    INTERVAL 190;
        .98 DO;
        PUT T IN QUEUE OF COMPUTER;
        GENERATE "COMPUTATION";
```

```
        DO AFTER INTERVAL 20;
        SET SITE BUFFER(SB OF TERMINAL(T))=0;
        SET LINE=0;
        GENERATE SEIZE LINE IF EXISTS QUEUE OF LINE;
        SET STATUS OF TERMINAL(T)=3;
        END;
    END;
END.
```

There is a 2% probability that an error is detected in the transmission; in this case, a signal is sent asking for retransmission, which takes 20 msec.; and another transmission occurs, taking again 170 msec. When the transmission is correct the line and the site buffer are freed, and the terminal is ready to receive an answer; it takes 20 msec. before those actions are executed. The message is sent to the computer for processing and must wait until the computer is free.

```
DECLARE COMPUTATION EVENT:
    DO IF COMPUTER=0;
       SET COMPUTER=1;
       DO AFTER INTERVAL 250 IF MESSAGE OF TERMINAL
                     (1ST QUEUE OF COMPUTER)=1;
                          300 IF MESSAGE OF TERMINAL
                     (1ST QUEUE OF COMPUTER)=2;
                          450 OTHERWISE;
```

```
        SET COMPUTER=0;
        DO IF 1ST QUEUE OF COMPUTER<=6;
            PUT 1ST QUEUE OF COMPUTER IN QUEUE OF LINE;
            PUT MESSAGE OF TERMINAL(1ST QUEUE OF
              COMPUTER)+2 IN QUEUE OF LINE;
            GENERATE SEIZE LINE AFTER INTERVAL 1;
        END;
        REMOVE  1ST QUEUE OF COMPUTER;
        GENERATE COMPUTATION IF EXISTS QUEUE OF
                                              COMPUTER;
    END.
```

The computer processes the input message for a time interval dependent on its type. Then, if the message came from one of the six terminals we are simulating, an attempt is made to seize the line and send the answer. The terminal number and the number of words in the answer are placed in the line queue. Then the computer considers the next message, if any. The SEIZE LINE event, already written above, generates an attempt to seize the site buffer:

```
DECLARE SEIZE BUFFER OUT EVENT:  PASSED T(VR),
                                          WORDS(VR);
    DO AFTER INTERVAL 80 IF SITE BUFFER(SB OF
```

```
                                    TERMINAL(T))=1;
        SET LINE=0;
    END;
    ELSE DO;
        SET SITE BUFFER(SB OF TERMINAL(T))=1;
        GENERATE "OUTPUT"(T, WORDS) AFTER INTERVAL 155;
    END.
```

We find out if the site buffer is busy, as in the case of input above; if it is, we make another attempt immediately. If we manage to seize the buffer, output is initiated.

```
DECLARE OUTPUT EVENT: PASSED T(VR), WORDS(VR);
    SELECT
        .01 GENERATE OUTPUT(T, WORDS) AFTER INTERVAL
                                                100;
        .98 DO;
           SET LINE=0;
           SET WORDS=WORDS-1;
           REMOVE 2ND QUEUE OF LINE;
           REMOVE 1ST QUEUE OF LINE;
           SET SITE BUFFER(SB OF TERMINAL(T))=0;
                               AFTER INTERVAL 325;
           DO IF WORDS=0;
                  GENERATE SCAN(T);
                  DO AFTER INTERVAL 495;
```

```
                SET STATUS OF TERMINAL(T)=0;
                GENERATE ANSWER RECEIVED(T);
            END;
        END;
        ELSE DO AFTER INTERVAL 495;
            PUT T IN QUEUE OF LINE;
            PUT WORDS IN QUEUE OF LINE;
            GENERATE SEIZE LINE;
        END;
    END;
END.
```

Again 1% of the time there may be a transmission error. Otherwise a word gets typed on the terminal. It takes 325 msec. for the site buffer to send the word to the terminal, and 170 msec. for the terminal to type out the word. The site buffer is released in any case, as has been the line; if the word typed was the last one, the terminal is also released, and a new scan is initiated. Otherwise, we try to seize the line again to send the next word.

Finally, we simulate the other processor buffer units' seizure of the computer by using dummy terminal number 7 for all their activity:

```
CREATE TERMINAL.
DECLARE "OTHER PBU" EVENT:
    SET MESSAGE OF TERMINAL(T)=FREQUENCY(.2 IF X=1,
```

```
            .5 IF X=2,.3 IF X=3,0 OTHERWISE);
    PUT 7 IN LINE OF COMPUTER;
    GENERATE COMPUTATION;
    GENERATE OTHER PBU AFTER INTERVAL
                              RANDOM(3200, 5000).
```

In order to start the simulation we must generate the first events;

```
GENERATE USER ENTRANCE.
GENERATE USER ENTRANCE.
GENERATE USER ENTRANCE.
GENERATE SCAN(0).
GENERATE SCAN(3).
DECLARE "I"=7.
DO UNTIL I=0; SET I=I-1; GENERATE OTHER PBU AFTER
        INTERVAL RANDOM(3200, 5000); END.
GO UNTIL CURRENT TIME=60*60*1000.
```

Thus three users walk in; the pair of processor buffer units begins scanning at the 1st and 4th terminal respectively; and six seizures of the computer by the other processor buffer units are generated. The simulation will run for one hour of the simulated time.

It should be noted that this example was designed to demonstrate the capabilities of an activity-oriented language, yet it can be programmed quite naturally in RELSIM. The programming

would become even simpler if we used the AT INTERVALS-UNTIL statement instead of using queues for the line and the central processor. Thus the process of seizing the line from event routine SCAN would be:

```
    GENERATE "SEIZE BUFFER IN"(T) AFTER INTERVAL 1
                          AT INTERVALS OF 1 UNTIL LINE=0;
```

with event routine SEIZE BUFFER IN modified to

```
DECLARE SEIZE BUFFER IN EVENT:  PASSED T(VR);
    DO IF LINE=0;
        SET LINE=1;
        DO AFTER INTERVAL 85 IF SITE BUFFER(SB OF
                                      TERMINAL(T))=1;
            SET LINE=0;
            GENERATE SCAN(T);
         END;
```

etc., eliminating the event SEIZE LINE. This is more like the type of programming done in simulation languages with no log, where conditions must be checked with every unit of simulation time that goes by. The cost of extra computing outweighs in most cases such simplification of programming.

Finally, let us consider a traffic intersection simulation that is designed to test prospective traffic policemen. This example is modeled after an intersection in downtown Athens where conditions have proven to be unmanageable by traffic signals and by most policemen; the few hardened veterans that can halfway manage it are famous and admired. This is a simplified version.

The intersection is a normal cross. North street has four lanes, which we will call, starting from the west, N1 through N4. The first two are southbound from 7: 00 AM to 4: 30 PM, and northbound during the rest of the day. N4 is open only to buses when N3 is southbound. N1 and N4 are the only lanes which buses may occupy. A car may turn left from N1 or N2. East street has four lanes, all westbound, which we will name E1 through E4 with E1 being the northernmost. Left turns are possible from E3 and E4, and right turns from E1 and, after 4: 30 PM, E2. There are northbound buses on E1 and westbound buses on E2. South street has two lanes, both southbound. No buses are allowed in S1, the easternmost lane. Finally, West street has four lanes named W1 through W4 from the south, two going each way, with buses allowed on W1 and W4, and left turns allowed from W2 after 4: 30. The center of town is to the south-east. Major jams occur at 7: 00 - 9: 00 AM and 4: 30 - 6: 00 PM, with a lesser one at 12: 30 - 2: 00 PM. At all such times there are huge crowds of pedestrians waiting to cross each corner.

For simplicity we will assume that once a crowd starts crossing a street it will occupy it for 3 0 seconds. The policeman is given an assessment of the situation every 30 seconds and is requested to give further directions. The simulation lasts from 7: 00 AM to 6: 00 PM, and may be taken as a test at many sittings; or, parts of it may be considered, e. g. a particular rush hour. The statistics will be kept with the car simulated and used as a REL-English base to determine the traffic policeman's performance.

Details again follow with the programming. In coding where cars come from and where they go to, we use 1 for north, 2 for east, 3 for south and 4 for west.

```
DECLARE "CAR" WITH "TIME IN"=CURRENT TIME AND
        "ORIGIN" AND "OLANE" AND "TIME OUT"=0.
        AND "DIRECTION" AND "DLANE"=0 AND "BUS"=0.
DECLARE "ENTRY" EVENT:
  CREATE CAR WITH DIRECTION=3;
  DO IF CURRENT TIME<9*60*60;
     SET ORIGIN=FREQUENCY(.44 IF X=1,.22 IF X=2,
                .34 IF X=4,0 OTHERWISE;
     SELECT IF ORIGIN=1;
        .4 SET OLANE=3;
        .4 DO; SET OLANE=2; SELECT; 1/8 SET DIRECTION
               =4; END; END;
```

```
        .2 DO; SET OLANE=1; SELECT; 1/4 SET DIRECTION
                                       =4; END; END;
      END;
      SELECT IF ORIGIN=2;
        .25 SET OLANE=4;
        .25 SET OLANE=3;
        .30 DO; SET OLANE=2; SET DIRECTION=4; END;
        .20 DO; SET OLANE=1; SET DIRECTION=4; END;
      END;
      ELSE SELECT;
        .5 SET OLANE=1;
        .5 SET OLANE=2;
      END;
      GENERATE ENTRY AFTER INTERVAL 7.8;
   END;
```

A car is created, and the direction it comes from, the lane it is in, and the direction it will go to, are determined; and the arrival of the next car is generated. If the time is 7: 00 - 9: 00 AM, 44% of the cars come from N, 22% from E and 34% from W; 90% of the cars from N want to go S, and 10% E, while for the cars from E 50% go S and 50% E, and all cars from E go S. Cars arrive every 7.8 seconds. In the same way the event may be completed for the other time intervals in the day that have certain traffic flow characteristics.

Buses are scheduled every minute on each line.

```
DECLARE "BUS ENTRY" EVENT:
    CREATE CAR WITH ORIGIN=1 AND OLANE=1 AND
                DIRECTION=3 AND BUS=1;
    CREATE CAR WITH ORIGIN=2 AND OLANE=1 AND
                DIRECTION=1 AND BUS=1;
    CREATE CAR WITH ORIGIN=2 AND OLANE=2 AND
                DIRECTION=4 AND BUS=1;
    CREATE CAR WITH ORIGIN=4 AND OLANE=1 AND
                DIRECTION=3 AND BUS=1;
    GENERATE BUS ENTRY AFTER INTERVAL 60.
```

At 16: 30 we must empty N3 for the direction change.

```
DO AFTER INTERVAL 16.5*60*60; SET OLANE=2 FOR
    CARS WITH ORIGIN=1 AND OLANE=3; END.
```

The next event controls intersection traffic. CONTROL will be a list with entries signifying the allowable direction coded as origin *10+ destination; i. e. N to S will be entered as 13.

```
DECLARE "CONTROL" LIST.
DECLARE "PRIORITY" ENTITY LIST.
DECLARE "CROSS" EVENT: PASSED D(NU), L(NU);
    LOCAL OUTQUEUE(LE);
    SET OUTQUEUE=CARS WITH DIRECTION=D AND
                DLANE=1;
```

```
DO IF NUMBER OF OUTQUEUE<15
        AND ¬(EXISTS OUTQUEUE WITH TIME
                        OUT>CURRENT TIME)
        AND(EXISTS PRIORITY WITH DIRECTION=D
            AND ORIGIN*10+D IN CONTROL);
    MAKE 1ST PRIORITY WITH DIRECTION=D AND
            ORIGIN=10+D IN CONTROL CURRENT CAR;
    REMOVE CURRENT CAR FROM PRIORITY;
    PUT 2ND CAR WITH ORIGIN=ORIGIN AND OLANE=
            OLANE IN PRIORITY;
    SET TIME OUT=CURRENT TIME+5;
    SET DLANE=L FOR INTERVAL 180;
    SET OLANE=0;
END;
GENERATE CROSS(D, L) AFTER INTERVAL 5.
```

Given a lane (by street and lane numbers) that leads away, this event sends a car to that lane if (a) the lane is not clogged up (b) there is no car already in the intersection heading for that lane (c) there exists a car that wants to go in that direction and has a green light; and if many such exist the car that has been waiting at the intersection the longest is selected.

To start things up we must generate one such event for each lane out:

```
GENERATE CROSS(4, 3).
```

```
GENERATE CROSS(4, 4).
GENERATE CROSS(3, 1).
GENERATE CROSS(3, 2).
GENERATE CROSS(1, 4).
GENERATE CROSS(1, 3) AFTER INTERVAL 4.5*60*60.
```

The next event passes control to the traffic policeman every 30 seconds.

```
DECLARE EVENT "COPE":
    WRITE NUMBER OF CARS WITH ORIGIN=1 AND
                                        OLANE=1;
etc. for all lanes in;
    WRITE NUMBER OF CARS WITH DIRECTION=1 AND
                                        DLANE=4;
etc. for all lanes out;
    WRITE CONTROL;
    DELETE CONTROL;
    GENERATE COPE AFTER INTERVAL 30;
    PAUSE.
```

Thus the man at the console sees the situation at the moment and is invited to put new entries in CONTROL that will allow cars to move in certain directions.

To start the simulation, we must create a few cars in each lane, put the first car of each lane in PRIORITY and generate the first ENTRY, BUS ENTRY, and COPE. The simulation time must

also be set to some time between 7: 00 AM and 6: 00 PM. The GO statement determines the length of the simulation.

III. AN INSIDE VIEW

In implementing RELSIM on the REL system, we have attempted to use the considerable latitude offered to the language writer by that system, and to avoid exceeding its limits. This was a constant temptation, as familiarity with a system implies awareness of all kinds of shortcuts available to the programmer willing to descend to the system level and make slight, unimportant modifications. Certain of the condition routines concerned with event declaration sentences and local variables get quite close to the edge in their treatment of the parsing graph, but in the end retain their virtue.

When the user enters a version of RELSIM for the first time, a context area is set up on a page whose ID is placed in the COMMON region. This contains the simulation time, originally zero; the seed for the random number generator, set to 5^{13}; the page ID's of created pages where system variables, attribute definitions, event definitions and the log will be saved; and various other pointers that should be readily accessible at all times, and will be discussed in their context.

A. Data structures

On declaring a class, an initial page is set up with a header containing pertinent information. The first word contains the page ID of that page; the second one is reserved to contain the ID of the first continuation page; there follow locations containing the

number of attributes defined in the class, the number of existent entities, a pointer to the location where the next created entity will reside, a pointer to the first entity, to the last entity, to the current entity, and a pointer reserved for the execution of a FOR statement on the class or some of its entities (see below); and finally, forty locations reserved for attribute descriptions.

These description pointers function as follows: If on declaration of the class (or in a subsequent entity creation statement) an attribute is declared and set equal to some expression, the parsed tree for that expression gets copied on a page, and a pointer to that location is placed in the class header. This expression will be then evaluated for each event created, at the time of its creation. The pages where these parsed trees are stored sequentially are accessible through the context area and linked.

There follows space for created entities. Each entity is made up of a self ID, a next and a previous entity pointer and the ID of its class, followed by the values of its attributes.

Continuation pages have a simple header with a self ID, the ID of the first page, and the ID of the next page, followed by entity entries.

A list is essentially set up in the same way, except no space for attributes exists in the header or each list element. The elements contain a payload instead of a self ID in their first word; and the header's third word contains a flag signifying whether this

is a FIFO or a LIFO list, rather than the attribute number.

Arrays have a simple header containing a self ID, a next page ID and the dimensions of the array, followed by single word entries.

System variables are placed sequentially in linked pages, available through the context area.

All of the above, when declared, get placed in the lexicon and subsequentially parse to their page ID, with the exception of attributes. These parse to the ID of a page that contains the class ID and the number of the attribute within that class, for all classes that have attributes of that name. Disambiguation then occurs depending on the context of the parsed sentence.

An event routine is parsed, and the parsed tree is placed on pages.

B. The log and the central processor

A notice for each instance of an event that is generated is posted on the log, which is a list structure kept in core during the execution of any sentence that requires its presence (and therefore during the entire simulation run, which is the execution of the sentence GO) and copied out in pages between sentences and on the occurrence of a PAUSE.

The structure of the log is as follows: there is a list element for each time point at which an event is supposed to be executed. These list elements are linked by the third word with ascending

time, and they contain that time in the second word. The first word connects all the event notices at that particular time, posted in the order of their generation.

An event notice is set up in the following manner: the first word, as we have noted, points to the next notice at the same simulation time; the second contains the ID of the parsed tree of the event routine, and the third points, if necessary, to another list element that acts as an information header. Thus, in its first word there may be a lexical ID of where the name of the notice occurs, so that it can be deleted after the event has been completed; the second points to a list of variables associated to the event by a FOR statement (see below); and the third points to a list of the values that the event's pass parameters will assume.

The log's page ID or core address, depending on its whereabouts, is kept in the context area.

The simulation central processor, which is the semantic routine for the GO statement, essentially passes control to the REL semantic processor on a copy of the parsed tree of each event routine,an instance of which becomes the current notice. After each notice is executed in this manner, it is then deleted from the log, and the simulation processor proceeds to find the next available notice.

Before each event is executed, however, the processor has to perform certain preliminary tasks. In the case where the

GENERATE statement which resulted in the current notice had been modified by any FOR clauses, the variables referenced by those clauses are to be found in the event notice; the simulation processor then modifies the copy of the parsed tree so that it is encompassed by a DO FOR statement on those same referenced variables. Again, all passed parameters and local variables reside in a particular page, accessed through the context page, during the execution of each event. The simulation processor must then update this page prior to the execution of each event, using the information on the log.

C. Other implementation aspects

Due to the fact that RELSIM is a fairly high level language, its syntax is very extensive, and the associated semantic routines tackle many diverse problems. A full description of the workings of the language here would be tedious. Appendix A contains the RELSIM syntax in its entirety; and Appendix B contains documentation on all condition routines, semantic routines and utility routines. We will restrict ourselves to discussing here, as illustration of the types of programming the language requires, three features of RELSIM: the DENSITY statement, the FOR clause and the handling of pass parameters and locals.

The DENSITY (f(X)) function operates in the following way: A scale factor c is determined, such that the function given satisfies $c.\, f(x) \leq 1$ in the interval it is defined, say (a, b). Then on request

of a random number of that distribution, we obtain two uniformly distributed random numbers r_1, r_2 in (0, 1) from the standard random number generator utility. We consider whether

$$r_2 \leq c. f(a+(b-a)r_1)$$

and if so, we accept $x=a+(b-a)r_1$. Otherwise another attempt is made.

The major problem here is the determination of the factor c. In a condition routine we make all instances of X in f(X) parse into real numbers containing no payload and flagged to signify their special status. Using copies of this parsed tree by placing a number into each flagged spot and calling the semantic processor, we do a rough search for a maximum value of f(X) in (a, b) using a grid of about 100 points. This produces an initial estimate of c, which is saved in the parsed tree of f(X). The semantic routine performs the test outlined above; and each time it finds that $c. f(a+(b-a)r_1 >1$, it updates the value of c. Thus the distributions become more accurate as more variates are produced.

The implementation of the FOR statement so that it meets the requirements outlined in chapter II, is the following: The semantic routine is a generator, i. e. the semantics of all parts of the parsed tree under the present parse have not been executed, contrary to normal REL procedure. If the modifier is an entity it is placed at the top of a stack referenced by a pointer in the header of the class of which it is a member. Thus it becomes

the current entity of that class; and the entity that was current before FOR modification happened occupies the bottom of the stack. If the modifier is a list, all of its elements are placed in the appropriate stacks in their classes; and the parsed tree is altered to look as if there were as many FOR statements as there were classes modified.

The routine then calls the REL semantic processor on a copy of the parsed tree of the sentence being modified by the FOR statement. On return, it is bumped; with a new current entity the semantic processor is called on another copy of the statement. When the stack of the class contains one element, the routine sets that as current and exits.

Consider what happens when we say

WRITE CURRENT X/CURRENT Y FOR X(1) FOR Y(1) FOR X(2).

The FOR routine is first called on X(2), which displaces the current X and calls the semantic processor on the sentence it modifies, namely

WRITE CURRENT X/CURRENT Y FOR X(1) FOR Y(1)

Thus the first routine called by the semantic processor is again the FOR routine on Y(1), and that, too, is set as current. Next another FOR routine on X(1) is called, X(1) displaces X(2) as the current entity and the sentence

WRITE CURRENT X/CURRENT Y

is executed, with X(1) and Y(1) current. Control returns to the

last mentioned FOR notice, and the stack is bumped so that X(2) is current. Another WRITE is executed, but on return this time the inner FOR notice finds nothing executable in the stack and thus replaces the old current X and destroys the stack. Then the middle FOR is returned in control and performs similarly on Y. Finally control goes to the outside FOR, which finds no X stack; it also, therefore, exits.

In exception to the above, where the basic verb modified is GENERATE, the FOR clauses merely pass the modifying entities or lists to the semantic routine for GENERATE, which then attaches these to the event notice posted on the log. Thus, instead of several identical events being generated, each with different current entities, a single event spanned by a DO FOR is posted, resulting in the same action in a more economical manner.

In order to handle the LOCAL and PASSED variables, a condition routine on the event declaration statement must first build a mini-lexicon of the names included, complete with payloads utilizing the information given about them, and then proceed to search the body of the event for instances of these names and, when found, span them with the appropriate parse. All other parses are destroyed, and control is returned to the system parser for a second try. All variables parse to a page ID on a special page utilized by all events being executed. Local list and array ID's are passed to the semantic routine and included in the

parsed tree of the event routine that is copied in pages, so that on execution of an instance of the event they will get created and initialized before any action occurs, and they will be deleted at the close of the event.

The GENERATE semantics, on the other hand, stores the values of the parameters it passes in the event notice. If a local list or array is to be passed, a new copy of it is made and marked as local for the event generated, so that it will be deleted by it. The central processing routine of the language then retrieves these payloads and stores them in the page where they should be at the time it calls on the event specified by the notice.

The above examples provide, we hope, some feeling for the kinds of tasks faced in the implementation of the language. Of course, implementation is an open-ended process, because as a more complete view of the language is obtained, better ways of doing things and more features that can be included keep occurring to the implementer. We cannot say that RELSIM contains all the features we would like to include in it, but at this point it seems to encompass enough to make it a viable and useful language.

BIBLIOGRAPHY

1. BUXTON, J. N. (Ed.) (1968) Simulation Programming Languages. New Holland Publishing Co., Amsterdam.

2. DAHL, O. J. and NYGAARD, K. (1966) SIMULA - An ALGOL - Based Simulation Language. Comm. ACM 9, 9, 671-678.

3. DOSTERT, B. H. (1971) REL - An Information System for a Dynamic Environment. CIT, Pasadena, Ca.

4. GORDON, G. (1969) System Simulation. Prentice - Hall Inc., Englewood Cliffs, N. J.

5. GREENBERGER, M. (1964) A New Methodology for Computer Simulation. Project MAC, MIT Cambridge, Mass. MAC-TR-13.

6. GREENBERGER, M., JONES, M. M., MORRIS, J. H., Jr., and NESS, D. N. (1966) On - Line Computation and Simulation: The OPS - 3 System. MIT Press, Cambridge, Mass.

7. HEIDORN, G. E. (1972) Natural Language Inputs to a Simulation Programming System. Naval Postgrad. School, Monterey, Ca. NPS-55HD72101A.

8. KIVIAT, P. J. (1969) Digital Computer Simulation: Computer Programming Languages. RAND Corp., Santa Monica, Ca. RM-5883-PR.

9. KIVIAT, P. J. (1974) Requirements for an Interactive Modeling and Simulation System. In Multi - Access Computing: Modern Research and Requirements. P. H. Rosenthal and R. K. Mish (Ed.). Hayden Book Co., Inc. Rochelle Park, N. J.

10. KIVIAT, P. J., VILLANUEVA, R. and MARKOWITZ, H. M. (1968) The SIMSCRIPT II Programming Language. RAND Corp., Santa Monica, Ca. R-460-PR.

11. KNUTH, D. E. and MC NELEY, J. L. (1964) SOL - A Symbolic Language for General Purpose Systems Simulation. Trans. IEEE, 401-414.

12. MC NELEY, J. L. (1967) Simulation Languages. Simulation 9, 2.

13. NAYLOR, T. H., BALINTFY, J. L., BURDICK, D. S. and CHU, K. (1966) Computer Simulation Techniques. Wiley & Sons, Inc., New York, N. Y.

14. TEICHROEW, D. and LUBIN, J. F. (1966) Computer Simulation - Discussion on the Technique and Comparison of Languages. Comm. ACM 9, 10, 723-741.

15. THOMPSON, F. B. (1974) The REL Language Processor. CIT, Pasadena, Ca.

16. THOMPSON, F. B., LOCKEMAN, P. C., DOSTERT, B. H., and DEVERILL, R. S. (1969) REL: A Rapidly Extensible Language System. Proc. 24th ACM Natl. Conf., 399-417.

17. TOCHER, K. D. (1963) The Art of Simulation. D. Van Nostrand Co., Inc., Princeton, N. J.

18. TOCHER, K. D. (1965) Review of Simulation Languages. Operations Research Quarterly. 15, 2, 189-218.

19. YOUNG, K. (1963) A User's Experience with Three Simulation Languages (GPSS, SIMSCRIPT and SIMPAC) System Development Corp., Santa Monica, Ca. TM-1755/000/00.

20. ———— (1966) Proc. of IBM Scientific Computing Symposium: Simulation Models and Gaming. IBM Corp., White Plains, N. Y.

21. ———— (1967) General Purpose Simulation System /360 User's Manual. IBM Corp., White Plains, N. Y. H20-0326-2.

APPENDIX A

```
*****************************************************************
*                                                               *
*                       RELSIM SYNTAX                           *
*                                                               *
*****************************************************************

THE FIRST FEW RULES ESTABLISH THE PARTS OF SPEECH OF
THE LANGUAGE AND THEIR SYNTAX FEATURES. THEY ARE NOT
TO BE APPLIED IN THE PARSING OF SENTENCES.
THE RELSIM PARTS OF SPEECH ARE:
   VR - VARIABLE OF ANY KIND
   OP - FUNCTION OPERATOR (E.G. MAX, MIN ETC.)
   EN - ENTITY
   NU - NUMERICAL EXPRESSION
   AB - ATTRIBUTE
   EC - ENTITY CLASS
   BO - BOOLEAN EXPRESSION
   CO - COMPARATOR (E.G. >, <=, ETC.)
   CJ - CONJUNCTION (E.G. AND, XOR ETC.)
   CN - CONDITIONAL NUMERICAL EXPRESSION
   PB - BOOLEAN EXPRESSION WITH UNRESOLVED ATTRIBUTES
   DE - DEFINITIONAL
   CL - INDIRECT SENTENCE
   VC - VERB CLAUSE
   DP - NUMERICAL EXPR. IN A "FREQUENCY" STATEMENT
   DX - NOT IN USE
   EV - EVENT ROUTINE
   PO - EVENT NOTICE (POSTING)
   NT - EVENT NOTICE NAME
   TA - LOCAL OR PASSED VARIABLE IN EVENT ROUTINE
   TB - VARIABLE TO BE PASSED IN "GENERATE" CLAUSE
   MA - ARRAY
   MN - NUMERICAL SEQUENCE FOR ARRAY DEFINITION
   WL - QUANTITY TO BE WRITTEN OUT

            10  11  12  13  14  15  16  17  18  19  1A
SYN:'SS'=>'VR  OP  EN  NU  AB  EC  BO  CO  CJ  CN  PB'
SEM:SEMRET

            1B  1C  1D  1E  1F  20  21  22  23  24  25
SYN:'SS'=>'DE  CL  VC  DP  DX  EV  PO  NT  TA  TB  MA'
SEM:SEMRET

            26  27
SYN:'SS'=>'MN  WL'
SEM:SEMRET
```

```
SYN: 'SS'=>' * *  NU * *'
CHK: -NAS-NMD-NXP
SEM:SEMRET

SYN:'SS'=>' * *  AB   * *'
      1   2   4   8   1   2   4   8   1   2   4   8
CHK:+FPN+ENT+LIS+SEC+AOF+AOP+ENC+AB2+AB3+AB4+LOS+LAT
SEM:SEMRET

SYN:'SS'=>' * *  VR   * *'
      1   2   4   8   1   2   4   8   1   2   4   8
CHK:+FPN+ENT+LIS+SEC+ELT+VR1+LIN+LIC+LCJ+SCR+LOS+LAT
SEM:SEMRET

SYN:'SS'=>' * *  DE   * *'
      1   2   4   8   1   2   4   8   1   2
CHK:+FPN+ENT+NUL+ATT+NAM+NUK+NUN+NEV+ATC+MAT
SEM:SEMRET

SYN:'SS'=>' * *  VC   * *'
      1   2   4   8   1   2   4   8   1   2   4   8
CHK:+CRE+CSP+TIN+LAP+VEN+VSN+CVC+CON+DOV+VIN+FAE+SEL
SEM:SEMRET

SYN:'SS'=>' * * *  VC   * * *'
      1   2   4   8
CHK:+SCE+GOD+ELS+NLS
SEM:SEMRET

SYN:'SS'=>' * *  DP   * *'
CHK:+DPF+RDP
SEM:SEMRET

SYN:'SS'=>' * *  TA   * *'
      1   2   4   8   1   2   4   8   1   2   4   8
CHK:+REF+VEF+LEF+LRF+MAF+ECF+ENF+NUF+ABR+ABE+ARL+AEL
SEM:SEMRET

SYN:'SS'=>' * * *  TA   * * *'
      1   2
CHK:+EVF+POF
SEM:SEMRET

THE NEXT RULE APPLIES A PRESCAN TO ALL SENTENCES.
BLANKS ARE HANDLED AND NUMBERS COLLECTED.

SYN:  '  '=>'$Z'
CND:  SCRPREN
SEM:
```

THE FOLLOWING RULES HANDLE THE WAYS IN WHICH A
NUMERICAL EXPRESSION MAY BE OBTAINED.

```
SYN:'NU'=>'VR'
CHK:-LIS+FPN
SEM:VTNU

SYN:'NU'=>'VR'
CHK:-FPN-ENT
SEM:VTNUIF

SYN:'NU'=>' N U M B E R   O F  VR'
CHK:+LIS-LAT
SEM:NUQ9

SYN: 'NU'=>'NU +NU'
CHK:,-NAS
SET: 0+NAS
SEM: ADD

SYN: 'NU'=>'NU -NU'
CHK:,-NAS
SET: 0+NAS
SEM: SUBTRACT

SYN: 'NU'=>'NU *NU'
CHK:-NAS,-NAS-NMD
SET: 0+NMD
SEM: MULTIPLY

SYN: 'NU'=>'NU /NU'
CHK:-NAS,-NAS-NMD
SET: 0+NMD
SEM: DIVIDE

SYN: 'NU'=>'NU * *NU'
CHK:-NAS-NMD,-NAS-NMD-NXP
SET: 0+NXP
SEM: SEMPOW

SYN: 'NU'=>' +NU'
CHK: -NAS
CND: SCRKA2
SET: 0+NAS
TNF:1
SEM:SEMRET
```

```
SYN: 'NU'=>' -NU'
CHK: -NAS
CND: SCRKA2
SET: 0+NAS
SEM: UNARYM

SYN: 'NU'=>' (NU )'
TNF:1
SEM:SEMRET

SYN: 'NU'=>' S I N (NU )'
SEM: SEMSIN

SYN: 'NU'=>' C O S (NU )'
SEM: SEMCOS

SYN: 'NU'=>' E X P (NU )'
SEM: SEMEXP

SYN: 'NU'=>' L O G (NU )'
SEM: SEMLOG

SYN: 'NU'=>' S Q R T (NU )'
SEM: SEMSQRT

SYN: 'NU'=>' F P (NU )'
SEM: SEMFP

SYN: 'NU'=>' I P (NU )'
SEM: SEMIP

SYN: 'NU'=>' A B S (NU )'
SEM: SEMABS

SYN: 'NU'=>' S I G N (NU )'
SEM: SEMSIGN

SYN: 'NU'=>' M A X (NU ,NU )'
SEM: SEMMAX

SYN: 'NU'=>' M I N (NU ,NU )'
SEM: SEMMIN

SYN: 'NU'=>' M O D (NU ,NU )'
SEM: SEMMOD

SYN: 'NU'=>' T A N (NU )'
SEM: SEMTAN
```

```
SYN: 'NU'=>' L N (NU )'
SEM: SEMLN

SYN: 'OP'=>' M A X'
SEM:C01

SYN: 'OP'=>' M I N'
SEM:OPA1

SYN:'NU'=>'OP (VR )'
CHK:,+LIS+FPN-LAT
SEM:VRY5

SYN:'NU'=>'OP (AB    F O R   VR )'
CHK:,+FPN,+LIS+ENT-LAT
SEM:VRY6

SYN:'NU'=>'OP (AB )'
CHK:,+FPN-AOP-AOF
SEM:VRY7

SYN:'NU'=>' R A N D O M (NU ,NU )'
SEM:RAND

SYN:'NU'=>' B I N O M I A L (NU ,NU )'
SEM:BIND

SYN:'NU'=>' P A S C A L (NU ,NU )'
SEM:PASD

SYN:'NU'=>' P O I S S O N (NU )'
SEM:POID

SYN:'NU'=>' U N I F O R M (NU ,NU )'
SEM:UNID

SYN:'NU'=>' N O R M A L (NU ,NU )'
SEM:NORD

SYN:'NU'=>' E X P D (NU )'
SEM:EXDI

SYN:'DP'=>'NU    I F    X =NU'
SEM:SEMRET

SYN:'DP'=>' 0    O T H E R W I S E'
SET:0+DPF
SEM:SEMRET
```

```
SYN:'DP'=>'DP  ,   DP'
CHK:-DPF,+DPF
SET:1+DPF
TNF:<1><2,*>
SEM:SEMRET

SYN:'NU'=>' D E N S I T Y ('
CND:CHXT@
SEM:SEMRET

SYN:'NU'=>' @'
CND:CNNCH
SEM:SEMRET

SYN:'NU'=>' D E N S I T Y "NU ""
CND:CNDEN
SEM:NUI1(G)

SYN:'NU'=>' F R E Q U E N C Y (DP )'
SEM:NUI2(G)

SYN:'NU'=>'NU %'
SEM:NUI5
```

THE NEXT FEW RULES HANDLE THE WAYS IN WHICH ENTITIES, SYSTEM VARIABLES AND LIST VARIABLES ARE OBTAINED.

```
SYN:'EN'=>'VR'
CHK:-LIS+ENT
SEM:VTNU

SYN:'VR'=>'VR'
CHK:+LIS+LAT
SET:1-LAT
SEM:VTNU

SYN:'VR'=>'MA (NU ,NU )'
SET:0+EPN
SEM:VRR1

SYN:'VR'=>'AB (EN )'
SET:1-SEC
SEM:NABT

SYN:'VR'=>'AB   O F   EN'
SET:1-SEC
SEM:NABT
```

```
SYN:'VR'=>'AB'
CHK:-AOF-AOP
CND:CNNRE
SET:1-SEC
SEM:VRG1

SYN:'VR'=>'EC'
SET:0+LIS+ENT+SEC
SEM:SEMNCP

SYN:'CE'=>' C U R R E N T   VR'
CHK:+LIS-LAT
SET:1-LIS
SEM:CURC

SYN:'VR'=>'CE'
SET:1+ELT
SEM:VTNU

SYN:'VR'=>' C U R R E N T   T I M E'
SET:0+FPN
SEM:CURT

SYN:'VR'=>' S E E D'
SEM:VRSD

SYN:'VR'=>' R A N D O M   VR'
CHK:+LIS-LAT
SET:1-LIS+ELT
SEM:RANC

SYN:'VR'=>'VR ( L )'
CHK:+LIS-LAT
SET:1-LIS+ELT
SEM:LASC

SYN:'VR'=>' L A S T   VR'
CHK:+LIS-LAT
SET:1-LIS+ELT
SEM:LASC

SYN:'VR'=>'VR (NU )'
CHK:+LIS-LAT,,
SET:1-LIS+ELT
SEM:LETN
```

```
SYN:'VR'=>'NU T H  VR'
CHK:,+LIS-LAT
SET:2-LIS+ELT
SEM:NTHC

SYN:'VR'=>'VR ( L -NU )'
CHK:+LIS-LAT,,
SET:1-LIS+ELT
SEM:LMNC

SYN:'VR'=>' N E X T  VR'
CHK:+LIS-LAT
SET:1-LIS+ELT
SEM:VRA1

SYN:'VR'=>' N E X T  VR'
CHK:-LIS+ELT
SET:1
SEM:VRA2

SYN:'VR'=>' P R E V I O U S  VR'
CHK:+LIS-LAT
SET:1-LIS+ELT
SEM:VRA3

SYN:'VR'=>' P R E V I O U S  VR'
CHK:-LIS+ELT
SET:1
SEM:VRA4

SYN:'VR'=>' A L L  VR'
CHK:+LIS-LAT
SET:1
TNF:1
SEM:SEMRET

THE FOLLOWING RULES BUILD UP BOOLEAN EXPRESSIONS.

SYN:'BO'=>'NUCONU'
SEM:BTST

SYN:'BO'=>'ENCOEN'
CHK:,+ENE,,
SEM:BTST
```

```
SYN:'BO'=>'VRCOVR'
CHK:+LIS-LAT,+ENE,+LIS-LAT
CND:FTEOF
SEM:LTST

SYN:'BO'=>'EN    I  N   VR'
CHK:,+LIS+ENT-LAT
SEM:BCR1

SYN:'BO'=>'NU    I  N   VR'
CHK:,+LIS+FPN-LAT
SEM:BCR1

SYN:'BO'=>' E  X  I  S  T  S   VR'
CHK:+LIS-LAT
SEM:BCX2

SYN:'BO'=>' ¬BO'
CHK:-BOP
SET:1
SEM:BREV

SYN:'BO'=>' N  E  I  T  H  E  R   BO'
CHK:-BOP
SET:1
SEM:BREV

SYN:'BO'=>' (BO )'
TNF:1
SEM:SEMRET

SYN:'BO'=>'BO   CJ   BO'
CHK:,,-BOP
SET:0+BOP
SEM:BPIL

SYN:'CO'=>' I  S'
CND:CNIS
SET:0+ENE
SEM:CO1

SYN:'CO'=>' ='
SET:0+ENE
SEM:CO1

SYN:'CO'=>' <'
SEM:CO2
```

```
SYN:'CC'=>' >'
SEM:C03

SYN:'CO'=>' < ='
SEM:C04

SYN:'CO'=>' > ='
SEM:C05

SYN:'CO'=>' ¬CO'
SET:1
SEM:CORV

SYN:'CO'=>' N O T  CC'
SET:1
SEM:CORV

SYN:'CO'=>' N O T'
CND:CNNO
SET:0+ENE
SEM:C06

SYN:'CO'=>'  CC'
CND:CONBR
SET:1
TNF:1
SEM:SEMRET

SYN:'CC'=>'CO   '
SET:1
TNF:1
SEM:SEMRET

SYN:'CO'=>'    I SCO'
SET:1
TNF:1
SEM:SEMRET

SYN:'   I N   '=>'   I S   I N   '
SEM:

SYN:'CJ'=>' A N D'
SEM:C01

SYN:'CJ'=>' O R'
SEM:C02

SYN:'CJ'=>' X O R'
SEM:C03
```

```
SYN:'CJ'=>' N C R'
SEM:C04
```

THE FOLLOWING RULES HANDLE CONDITIONAL NUMERICAL EXPRESSIONS.

```
SYN:'CN'=>'NU   I F  BC ,  '
SEM:NUC1(G)

SYN:'CN'=>'NU   O T H E R W I S E'
SET:0+COP
TNF:1
SEM:SEMRET

SYN:'CN'=>'CNCN'
CHK:-COP,+COP
SET:2
SEM:CNPU(G)

SYN:'NU'=>'CN'
CHK:+COP
CND:CNCN
TNF:1
SEM:SEMRET
```

THE NEXT FEW RULES BUILD UP BOOLEANS WITH ATTRIBUTES IN THEM WHICH ARE NOT BOUND TO ANY ENTITY AT THIS STAGE. THESE BOOLEANS ARE USED IN MAKING LISTS OF ENTITIES WITH SPECIFIC CHARACTERISTICS.

```
SYN:'PB'=>'ABCONU'
CHK:+FPN+AOF,,,
SEM:BTST

SYN:'PB'=>'ABCOEN'
CHK:+ENT+AOF,+ENE,,
SEM:BTST

SYN:'PB'=>'ABCOVR'
CHK:+LIS+AOF,+ENE,+LIS-LAT
CND:FTFOE
SEM:LTST

SYN:'PB'=>'AB   I N   VR'
CHK:+AOF,+LIS-LAT
CND:BFOE
SEM:BOR1
```

```
SYN:'PB'=>'PB  CJ  PB'
CHK:+PBP,,-PBP
SET:1
SEM:BPIL

SYN:'PB'=>' W I T H  PB'
CHK:-PBP
SET:1+PBP
TNF:1
SEM:SEMRET

SYN:'PB'=>' ¬PB'
CHK:-PBP
SET:1
SEM:BREV

SYN:'PB'=>' N E I T H E R  PB'
CHK:-PBP
SET:1
SEM:BREV

SYN:'PB'=>' (PB )'
SET:1
TNF:1
SEM:SEMRET

THE FOLLOWING RULES COLLECT LEVELS OF UNBOUND
ATTRIBUTES.

SYN:'AB'=>'AB   O F  AB'
CHK:+ENT,+ENT+AOP
SET:2
SEM:FAB1

SYN:'AB'=>'AB   O F  AB'
CHK:-ENT,+ENT+AOP
SET:1+AOP+AOF
SEM:FAB1

SYN:'AB'=>'AB (AB )'
CHK:+ENT,+ENT+AOP
SET:2
SEM:FAB1

SYN:'AB'=>'AB (AB )'
CHK:-ENT,+ENT+AOP
SET:1+AOP+AOF
SEM:FAB1
```

```
SYN:'AB'=>'AB'
CHK:-AOP
CND:ABEF
SET:1+AOP
SEM:FAB2

SYN:'AB'=>'AB'
CHK:-AOF+AOP
CND:ABIF
SET:1+AOF
TNF:1
SEM:SEMRET

THE FOLLOWING RULES ARE USED IN THE CREATION OF
SCRATCH LISTS.

SYN:'VR'=>'EC  PB'
CHK:,+PBP
CND:CNNPB
SET:0+LIS+ENT+SCR
SEM:ECR1(G)

SYN:'EC'=>'EC S'
TNF:1
SEM:SEMRET

SYN:'VR'=>'VR   I N  VR'
CHK:+LIS-LAT-LIN,+LIS-LAT
CND:BFOE
SET:1+LIN-SEC+SCR
SEM:ECR2A

SYN:'VR'=>'VR  CJ   I N  VR'
CHK:+LIS-LAT-LIN-LIC,,+LIS-LAT-LIN
CND:FTFOE
SET:1+LIC-SEC+SCR
SEM:ECR2

SYN:'VR'=>'VR   N O T   I N  VR'
CHK:+LIS-LAT-LIN,+LIS-LAT
CND:BFOE
SET:1+LIN-SEC+SCR
SEM:ECR2B

SYN:'VR'=>'VR   A N D   N O T   I N  VR'
CHK:+LIS-LAT-LIN-LIC,+LIS-LAT-LIN
CND:BFOE
SET:1+LIC-SEC+SCR
SEM:ECR2B
```

```
SYN:'VR'=>'VR   A N D   N O T  VR'
CHK:+LIS-LAT-LIN-LIC-LCJ,+LIS-LAT-LIN-LIC
CND:BFOE
SET:1+LCJ-SEC+SCR
SEM:ECR2B

SYN:'VR'=>'VR  CJ  VR'
CHK:+LIS-LAT-LIN-LIC-LCJ,,+LIS-LAT-LIN-LIC
CND:FTFOE
SET:1+LCJ-SEC+SCR
SEM:ECR2

SYN:'VR'=>' (VR )'
CHK:+LIS
SET:1-LIN-LIC-LCJ
TNF:1
SEM:SEMRET

THE NEXT RULES HANDLE DECLARATION STATEMENTS EXCEPT
FOR EVENT ROUTINES.

SYN:'DE'=>' W I T H   ''
SET:0+ATT+FPN
CND:CNAC
SEM:DER1

SYN:'DE'=>' A N D   ''
SET:0+ATT+FPN
CND:CNAC
SEM:DER1

SYN:'DE'=>' D E C L A R E   ''
SET:0+FPN
CND:CNAC
SEM:DER2

SYN:'DE'=>' N A M E D   ''
SET:0+NAM
CND:CNAC
SEM:SEMRET

SYN:'DE'=>'DE   R E A L'
CHK:-NUK-NUL-NUN-NAM-NEV
SET:1+NUK
TNF:1
SEM:SEMRET
```

```
SYN:'DE'=>'DE   L I S T'
CHK:-NUL-NUN-NAM-NEV
SET:1+NUL
SEM:DER3(G)

SYN:'DE'=>'DE   S T A C K'
CHK:-NUL-NUN-NAM-NEV
SET:1+NUL
SEM:DEP1(G)

SYN:'DE'=>'DE (NU ,NU )'
CHK:-NUK-NUN-NUL-NAM-NEV-MAT,,,
SET:1+MAT
SEM:DEX1(G)

SYN:'DE'=>'DE   E N T I T Y'
SET:1+NUK+ENT-FPN
CHK:-NUK-NUL-NUN-NAM-NEV-MAT
TNF:1
SEM:SEMRET

SYN:'DE'=>'DE   C L A S S'
CHK:-NUK-NUL-NUN-NAM-NEV-MAT
SET:1+NUK+ENC-FPN
SEM:DER4(G)

SYN:'DE'=>'DE =NU'
CHK:+FPN-NUL-NUN-NAM-NEV,,
SET:1+NUN
SEM:DER5(G)

SYN:'DE'=>'DE =EN'
CHK:+ENT-NUL-NUN-NAM-NEV,,
SET:1+NUN
SEM:DER5(G)

SYN:'DE'=>'DE =VR'
CHK:+NUL-NUN-NAM-NEV,+LIS-LAT
CND:BFOE
SET:1+NUN
SEM:DER5(G)

SYN:'DE'=>'DE'
CHK:+ATT-ATC
CND:POSR
SET:1+ATC
TNF:1
SEM:SEMRET
```

```
SYN:'DE'=>'DE  DE'
CHK:+ATT-ATC,+ATT+ATC
SET:2
SEM:SEMRET

SYN:'SS'=>'DE .'
CHK:-ATT-NAM-NEV
CND:CNTS
SEM:SSR1

SYN:'SS'=>'DE  DE .'
CHK:-ATT+ENC,+ATT+ATC
CND:CNTS
SEM:SSR1

THE FOLLOWING RULES HANDLE THE COMMAND STATEMENTS
AND MODIFYING CLAUSES USED IN EVENTS OR DIRECTLY.

SYN:'VC'=>' C R E A T E  EC'
SET:0+CRE+CSP
SEM:VCR1

SYN:'VC'=>' C R E A T E  EC  DE'
CHK:,+NAM
SET:0+CRE+CSP
SEM:VCR4(G)

SYN:'VC'=>'VC  DE'
CHK:+CRE,+ATT+ATC
SET:1
SEM:VCR2(G)

SYN:'VC'=>'VC  PB'
CHK:+CRE,+PBP
CND:CNNPB
SET:1
SEM:VCY1(G)

SYN:'VC'=>'VC  F O R  I N T E R V A L  NU'
CHK:-TIN+CSP-LAP-FAE-VIN-CON,,
SET:1+TIN
SEM:VCR3(G)

SYN:'VC'=>' C R E A T E  NU  EC'
SET:0+CRE+CSP
SEM:VCR5

SYN:'DG'=>' G E N E R A T E  EV'
SEM:SEMRET
```

```
SYN:'DG'=>' G E N E R A T E  "'
CND:CNDVC
SEM:SEMRET

SYN:'VC'=>'DG'
SET:0+GOD
TNF:<1,*>
SEM:VCA1

SYN:'VC'=>'DG (TB )'
CHK:,+TBF
SET:0+GOD
TNF:<1,*><2>
SEM:VCT1(G)

SYN:'VC'=>'VC  DE'
CHK:+GOD-DOV,+NAM
SET:1
SEM:VCA2(G)

SYN:'VC'=>'VC   A F T E R   I N T E R V A L  NU'
CHK:-LAP-VEN-CVC-VSN-ELS-CON,,
SET:1+LAP
SEM:VCA3(G)

SYN:'VC'=>'VC   I F  BO'
CHK:-CON-VEN-CVC-VSN-ELS,,
SET:1+CON
SEM:NUC1(G)

SYN:'VC'=>' E L S E  VC'
CHK:-DOV-SEL-CON
CND:POSR
SET:1+ELS
TNF:1
SEM:SEMRET

SYN:'VC'=>'VC ;  VC'
CHK:+CON,+ELS
SET:1+ELS
SEM:CNPU(G)

SYN:'VC'=>'VC'
CHK:+CON-NLS-DOV-SEL-ELS
CND:NELSR
SET:1+NLS
TNF:1
SEM:SEMRET
```

```
SYN:'VC'=>' S E T  VR =NU'
CHK:+FPN-LIS-SCR,,
SET:0+CSP+SCE
SEM:VCA8

SYN:'VC'=>' S E T  VR =NU'
CHK:-FPN-ENT-LIS,,
SET:0+CSP+SCE
SEM:VCA8A

SYN:'VC'=>' S E T  VR =EN'
CHK:+ENT-LIS-SEC-SCR,,
SET:0+CSP+SCE
SEM:VCA8

SYN:'VC'=>' M A K E  VR  CE'
CHK:-LIS,,
CND:BFOE
SEM:VCA8B

SYN:'VC'=>' S E T  VR =VR'
CHK:+LIS-LAT-SCR,+LIS-LAT
CND:BFOEL
SET:0+CSP+SCE
SEM:VCL1

SYN:'MN'=>'NU'
SEM:SEMRET

SYN:'MN'=>'NU ,MN'
TNF:<1><2,*>
SEM:SEMRET

SYN:'VC'=>' S E T  MA =MN'
SET:0+CSP+SCE
SEM:VCC4(G)

SYN:'VC'=>' D E L E T E  EN'
CND:SIFN
SEM:VCB1

SYN:'VC'=>' D E L E T E  VR'
CHK:-SEC-LIS
CND:SIFN
SEM:VCB3

SYN:'VC'=>' D E L E T E  VR'
CHK:+LIS-SCR-LAT
SEM:VCFA
```

```
SYN:'VC'=>' D E L E T E  VR'
CHK:+LIS+SCR-LAT+ENT
SEM:VCFB

SYN:'VC'=>' D E L E T E  L I S T  VR'
CHK:+LIS-SEC-SCR-LAT
CND:SIFN
SEM:VCF1

SYN:'VC'=>' D E L E T E  C L A S S  EC'
CND:SIFN
SEM:VCF1

SYN:'VC'=>' D E L E T E  EV'
CND:SIFN
SEM:VCB3

SYN:'VC'=>' D E L E T E  PO'
SEM:VCA4(G)

SYN:'VC'=>' D E L E T E  MA'
SEM:VCE4

SYN:'VC'=>' D E L E T E  A L L  C L A S S E S'
SEM:VCF3

SYN:'VC'=>' D E L E T E  A L L  E V E N T S'
SEM:VCF4

SYN:'VC'=>' D E L E T E  A L L  L I S T S'
SEM:VCF5

SYN:'VC'=>' D E L E T E  A L L  C O N S T A N T S'
SEM:VCF7

SYN:'VC'=>' D E L E T E  A L L  A R R A Y S'
SEM:VCG2

SYN:'VC'=>' D E L E T E  L O G'
SEM:VCF8

SYN:'VC'=>' R E S E T'
SEM:VCF9

SYN:'VC'=>' P A U S E'
SEM:VCB4
```

```
SYN:'VC'=>' D O'
SET:0+DOV+GOD
SEM:SEMRET

SYN:'VC'=>'VC   U N T I L   BO'
CHK:-VIN-CVC-ELS-VEN-VSN-CCN,,
SET:1+VIN
SEM:VCH1(G)

SYN:'VC'=>'VC   A T   I N T E R V A L S   O F   NU   U N T
CHK:-VIN-CVC-ELS-VEN-VSN-CON,,,                                 I L   BO'
SET:1+VIN
SEM:VCB6(G)

SYN:'VC'=>'VC   F O R   VR'
CHK:-CVC-ELS-VEN-VSN-CCN,+ENT+LIS-LAT
SET:1
SEM:VCB8(G)

SYN:'VC'=>'VC   F O R   EN'
CHK:-CVC-ELS-VEN-VSN-CON,,
SET:1
SEM:VCI1(G)

SYN:'VC'=>' E N D'
CND:CNCP
SET:0+VEN
SEM:SEMRET

SYN:'VC'=>'VC ;  VC'
CHK:-VEN-DOV-SEL-VSN,+VEN
SET:0+VEN
TNF:<1><2,*>
SEM:SEMRET

SYN:'VC'=>'VC ;  VC'
CHK:+DOV,+VEN
SET:1+CVC-DOV-GOD
SEM:VCC1(G)

SYN:'VC'=>' S E L E C T'
SET:0+SEL
SEM:SEMRET

SYN:'DP'=>'NU  VC'
CHK:,-VEN-DOV-SEL
SEM:SEMRET
```

```
SYN:'DP'=>'DP  ;   VC'
CHK:-DPF,+VEN
SET:0+DPF
TNF:<1>
SEM:SEMRET

SYN:'DP'=>'DP  ;   DP'
CHK:-DPF,+DPF
SET:2
TNF:<1><2,*>
SEM:SEMRET

SYN:'VC'=>'VC  ;   DP'
CHK:+SEL,+DPF
SET:1+CVC-SEL
SEM:VCC3(G)

SYN:'VC'=>' P U T  EN   I N  VR'
CHK:,+LIS+ENT-SEC-LAT-SCR
SET:0+CSP
SEM:VCC5

SYN:'VC'=>' P U T  VR   I N  VR'
CHK:+LIS-LAT,+LIS-SEC-SCR-LAT
CND:BFOE
SET:0+CSP
SEM:VCQ1

SYN:'VC'=>' P U T  NU   I N  VR'
CHK:,+LIS+FPN-SCR-LAT
SET:0+CSP
SEM:VCC5

SYN:'VC'=>' P U T  NU   B E F O R E  VR   I N  VR'
CHK:,-LIS+FPN+ELT,+LIS+FPN-SCR-LAT
SET:0+CSP
SEM:VCD1

SYN:'VC'=>' P U T  EN   B E F O R E  VR   I N  VR'
CHK:,-LIS+ENT+ELT,+LIS+ENT-SEC-SCR-LAT
SET:0+CSP
SEM:VCD1

SYN:'VC'=>' P U T  EN   A F T E R  VR   I N  VR'
CHK:,-LIS+ENT+ELT,+LIS+ENT-SEC-SCR-LAT
SET:0+CSP
SEM:VCD2
```

```
SYN:'VC'=>' P U T  NU   A F T E R   VR   I N  VR'
CHK:,-LIS+FPN+FLT,+LIS+FPN-SCR-LAT
SET:0+CSP
SEM:VCD2

SYN:'VC'=>' P U T  EN   I N   P L A C E  NU   I N  VR'
CHK:,,+LIS+ENT-SEC-SCR-LAT
SET:0+CSP
SEM:VCD3

SYN:'VC'=>' P U T  NU   I N   P L A C E  NU   I N  VR'
CHK:,,+LIS+FPN-SCR-LAT
SET:0+CSP
SEM:VCD3

SYN:'VC'=>' R E M O V E  EN   F R O M  VR'
CHK:,+LIS+ENT-SEC-SCR-LAT
SEM:VCE1

SYN:'VC'=>' R E M O V E  VR   F R O M  VR'
CHK:+LIS-LAT,+LIS-SEC-SCR-LAT
CND:BFOE
SEM:VCQ2

SYN:'VC'=>' R E M O V E  NU   F R O M  VR'
CHK:,+LIS+FPN-SCR-LAT
SEM:VCE1

SYN:'VC'=>' R E M O V E  VR (NU )'
CHK:+LIS-LAT-SEC-SCR,,
SEM:VCE5

SYN:'VC'=>' R E M O V E  VR ( L )'
CHK:+LIS-SEC-SCR-LAT
SEM:VCE6

SYN:'VC'=>' W R I T E  VR'
CHK:+LIS-LAT
SEM:VCZ3

SYN:'VC'=>' W R I T E  MA'
SEM:VCE2

SYN:'VC'=>' W R I T E  PO'
SEM:VCA5(G)

SYN:'VC'=>' W R I T E  WL'
CHK:+WLF+WLD
SEM:VCB7(G)
```

```
SYN:'VC'=>' W R I T E   L O G'
SEM:VCA6

SYN:'VC'=>' W R I T E  EV'
SEM:VCD7

SYN:'VC'=>' N A M E  EN   ''
CND:CNACX
SEM:VCA7

SYN:'VC'=>' N A M E  PC   ''
CND:CNACX
SEM:VCB2(G)
```

THE NEXT FEW RULES HANDLE COMPLETE SENTENCES, DIRECT OR PARTS OF AN EVENT DECLARATION SENTENCE.

```
SYN:'SS'=>'VC .'
CND:CNTS
CHK:-DOV-SEL-VEN-VSN-ELS-CON-NLS
SEM:SSR1

SYN:'SS'=>'VC .'
CND:CNTS
CHK:+CON+NLS
SEM:SSR1

SYN:'SS'=>'VC .'
CND:CNTS
CHK:+CON+ELS
SEM:SSR1

SYN:'CL'=>'VC'
CHK:-DOV-SEL-CON-ELS-VEN-VSN
CND:SCLCFR
TNF:1
SEM:SEMRET

SYN:'CL'=>'VC'
CHK:+CON+ELS
CND:SCLCFR
TNF:1
SEM:SEMRET

SYN:'CL'=>'VC'
CHK:+CON+NLS
CND:SCLCFR
TNF:1
SEM:SEMRET
```

```
SYN:'CL'=>'CL .'
CHK:-CLX
SET:1+CLX
SEM:SEMRET

SYN:'CL'=>'CL ;  CL'
CHK:-CLX,+CLX
SET:2
TNF:<1><2,*>
SEM:SEMRET
```

THE FOLLOWING RULES ARE USED TO DECLARE AN EVENT.

```
SYN:'DV'=>'DE   E V E N T'
CHK:-ATT-NUK-NUL-NUN-NAM-NEV
CND:CNTSEV
SEM:SEMRET

SYN:'DV'=>' D E C L A R E  EV   E V E N T'
CND:PASLOC
SEM:DVA1

SYN:'SS'=>'DV :  CL'
CHK:,+CLX
SEM:SSQ7(G)
```

THE FOLLOWING RULES CONCERN STATEMENTS THAT MAY ONLY BE DIRECT.

```
SYN:   'SS'=>' E X I T'
SEM:EXITSIM

SYN:'SS'=>' C O N T I N U E'
SEM:VCJ1

SYN:'SS'=>' R E C O R D   S I M U L A T I O N'
SEM:VCJ2

SYN:'SS'=>' G O .'
SEM:SSQ1

SYN:'SS'=>' G O   U N T I L  BO'
SEM:SS2

SYN:   'SS'=>' D E F :'
CND:SCRDEF
SEM:SSR1
```

```
SYN:'SS'=>' W R I T E   D E F   O F'
CND:SCRNDEF1
SEM:SEMRET

SYN:'SS'=>' D E L E T E   D E F   O F'
CND:SCRNDEF2
SEM:SEMRET

SYN:'SS'=>' D E L E T E   D E F  NU   O F'
CND:SCRNDEF3
SEM:SEMRET

THE RULES THAT FOLLOW HANDLE THE PASSING OF PARAMETERS
AND THE ASSIGNMENT OF LOCALS.

SYN:'VR'=>'TA'
CHK:+REF
SET:O+FPN-LIS
SEM:VTNU

SYN:'VR'=>'TA'
CHK:+VEF
SET:O+ENT
SEM:VTNU

SYN:'EN'=>'TA'
CHK:+ENF
SEM:VTNU

SYN:'NU'=>'TA'
CHK:+NUF
SEM:VTNU

SYN:'AB'=>'TA'
CHK:+ABR
SET:O+FPN
SEM:VTNU

SYN:'AB'=>'TA'
CHK:+ABE
SET:O+ENT
SEM:VTNU

SYN:'AB'=>'TA'
CHK:+ARL
SET:O+FPN+LIS+LAT
SEM:VTNU
```

```
SYN:'AB'=>'TA'
CHK:+AEL
SET:0+ENT+LIS+LAT
SEM:VTNU

SYN:'EC'=>'TA'
CHK:+ECF
SEM:VTNU

SYN:'EV'=>'TA'
CHK:+EVF
SEM:VTNU

SYN:'PO'=>'TA'
CHK:+POF
SEM:VTNU

SYN:'VR'=>'TA'
CHK:+LRF
SET:0+LIS-LAT+FPN
SEM:VTNU

SYN:'VR'=>'TA'
CHK:+LEF
SET:0+LIS-LAT+ENT
SEM:VTNU
SYN:'MA'=>'TA'
CHK:+MAF
SEM:VTNU

SYN:'TB'=>'VR'
CND:NOAB
CHK:-LIS-ENT
SEM:SEMRET

SYN:'TB'=>'VR'
CHK:+LIS-LAT
CND:NOAB
SEM:SEMRET

SYN:'TB'=>'EN'
CND:NOAB
SEM:SEMRET

SYN:'TB'=>'NU'
CND:NOVRIN
SEM:SEMRET
```

```
SYN:'TB'=>'AB'
SEM:SEMRET

SYN:'TB'=>'EC'
SEM:SEMRET

SYN:'TB'=>'EV'
SEM:SEMRET

SYN:'TB'=>'PO'
SEM:SEMRET

SYN:'TB'=>'TB'
CHK:-TBF
CND:CNPRR
SET:0+TBF
TNF:1
SEM:SEMRET

SYN:'TB'=>'TB ,TB'
CHK:-TBF,+TBF
SET:2
TNF:<1,*><2,*>
SEM:SEMRET
```

THE NEXT FEW RULES HANDLE EXPRESSIONS DENOTING EVENT NOTICES.

```
SYN:'PO'=>'NT'
SEM:POA1

SYN:'PO'=>'EV (NU )'
SEM:POA2

SYN:'PO'=>'EV ( L )'
SEM:POA3

SYN:'PO'=>'EV ( L -NU )'
SEM:POA4

SYN:'PO'=>' R A N D O M  EV'
SEM:POA5

SYN:'PO'=>'NU T H  E V E N T'
SEM:POA7

SYN:'PO'=>' L A S T  E V E N T'
SEM:POA8
```

```
SYN:'PO'=>' E V E N T ( L -NU )'
SEM:POA9

SYN:'PO'=>' R A N D O M   E V E N T'
SEM:POB1

SYN:'PO'=>'PO   A T   T I M E  NU'
CHK:-POT,,
SET:1+POT
SEM:POA6(G)
```

THE LAST FEW RULES HANDLE THE WRITING OUT OF COMPLEX LINES.

```
SYN:'WL'=>'NU'
SET:0+WLD
SEM:SEMRET

SYN:'WL'=>'EN'
SET:0+WLD
SEM:SEMRET

SYN:'WL'=>' C "'
CND:CNACW
SEM:SEMRET

SYN:'WL'=>'WL'
CHK:-WLD
SET:0+WLD
SEM:SEMRET

SYN:'WL'=>'WL'
CHK:-WLF+WLD
CND:CNCN
SET:1+WLF
TNF:1
SEM:SEMRET

SYN:'WL'=>'WL ,  WL'
CHK:+WLF+WLD,-WLF+WLD
SET:1
TNF:<1,*><2,*>
SEM:SEMRET
```

B. ROUTINE DOCUMENTATION

In this appendix we describe the workings of all syntax completion routines, semantic routines and utility routines used by RELSIM. Listings of all these routines are available through Professor Frederick B. Thompson at C. I. T. We shall consider each set of routines alphabetically by their names, so that they can be referenced easily from the syntax rules in Appendix A; and we include flow charts wherever the routine is complex enough to merit it. The name in parentheses next to the routine name refers to the deck in which the routine is to be found.

B. 1. Syntax Completion Routines

B. 1. 1. ABEF (STC1): Determines whether an attribute is part of an unbound boolean. This is done by checking the parsing graph to the right of the attribute, and succeeding if there is a comparator or a right parenthesis immediately following.

B. 1. 2. ABIF (STC1): Given an unbound attribute, this determines whether it is a complete element in a boolean comparison, i. e. there is no level of indirection of the form AB OF AB with the attribute considered being the latter. This is done by checking the parsing graph to the left and making sure that the attribute is preceded by a conjunction or the word WITH, possibly before a left parenthesis.

B. 1. 3. BFOE (STC3): Allows a parse only if the first two parts of speech on the right side of the rule are both either numerical expressions or entity expressions. This is done by checking the relevant features of the parts of speech.

B. 1. 4. BFOEL (STC3): Same as BFOE, but succeeds only if the first part of speech has no constituents. This is because when we apply this routine the first part of speech is a list, but it should not be an entity class.

B. 1. 5. CHXTO@ (STC2): This routine is applied to the function given in a DENSITY statement and changes all instances of the free variable X in the graph to at signs, and the parentheses enclosing the function to the double quotes. All parses to the right are then destroyed and the parse fails. This results in the rule 'NU'=>'@' applying to all instances of the free variable, and subsequently the rule 'NU'=>'DENSITY "NU" ' applying.

B. 1. 6. CNAC (STC1): Collects all characters to the right of a double quote until the next double quote in the parsing graph is reached, and places their PI's as constituent elements of the left hand side part of speech. The parse fails if no second double quote exists, or if the string does not start with a letter.

B. 1. 7. CNACW (STC1): Same as CNAC, but does not check whether the string starts with a letter.

B. 1. 8. CNACX (STC1): Same as CNAC, but, due to the fact that the left hand side part of speech already has a constituent, places

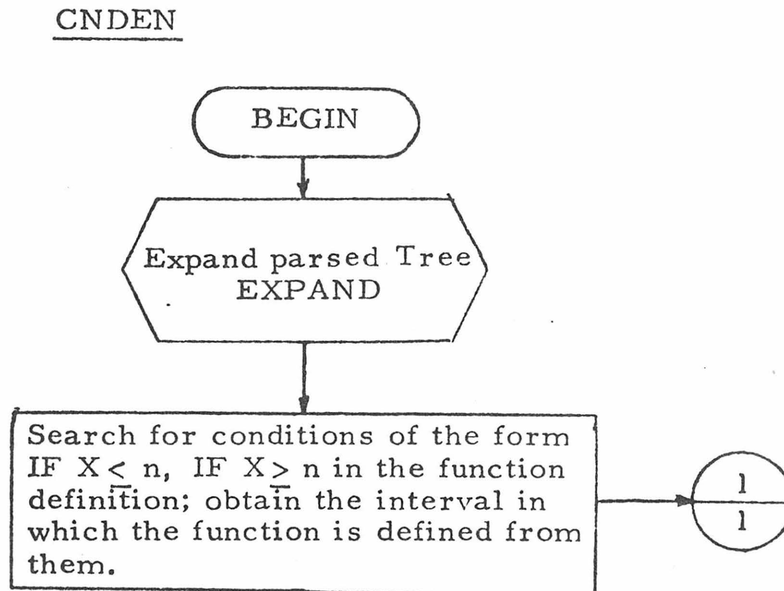
the character PI's under that constituent.

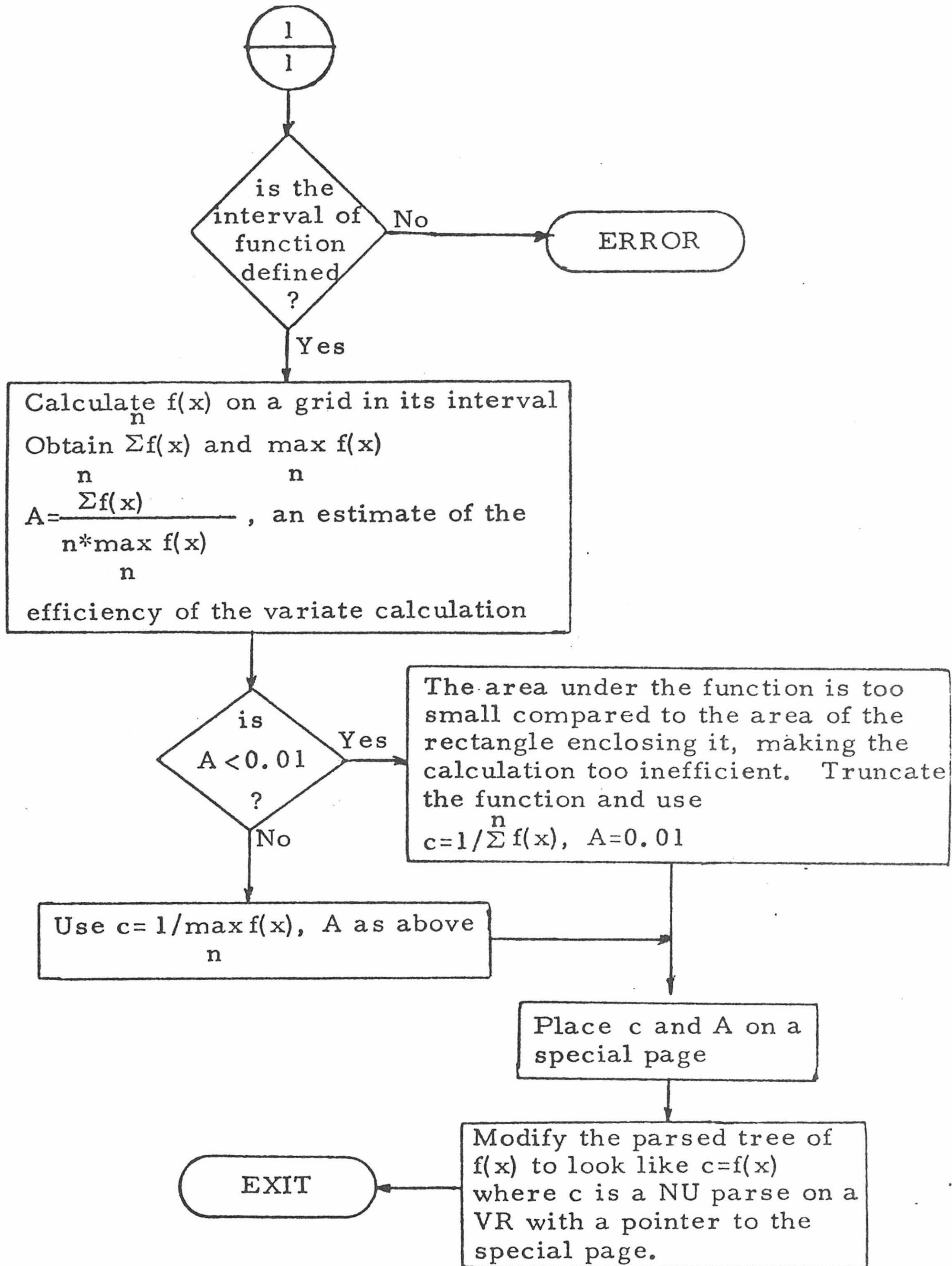
B. 1. 9. CNCN (STC1): Allows a conditional expression to become unconditional if there are no more conditional expressions to the left. This is done by checking to see if there is no comma to the left in the parsing graph.

B. 1. 10. CNCP (STC3): Places a dummy constituent under the left hand part of speech so that a transformation of the form <2, *> may be applied to it.

B. 1. 11. CNDEN (STC2): Examines the function supplied in a DENSITY statement and calculates the first estimate of the scale factor required for the semantic routine of this statement, placing it in a special page and modifying the parsed tree of the function so that multiplication by the variable to be found on that page is performed. A flow chart follows:

CNDEN





1
1
is the interval of function defined ?
No
ERROR
Yes
Calculate f(x) on a grid in its interval
Obtain Σf(x) and max f(x)
n n
A= Σf(x) / n*max f(x) , an estimate of the
efficiency of the variate calculation
is A<0.01 ?
Yes
The area under the function is too small compared to the area of the rectangle enclosing it, making the calculation too inefficient. Truncate the function and use
c=1/Σ f(x), A=0.01
No
Use c= 1/max f(x), A as above
Place c and A on a special page
Modify the parsed tree of f(x) to look like c=f(x) where c is a NU parse on a VR with a pointer to the special page.
EXIT

B. 1. 12. CNDVC (STC1): Collects a string that must be the name of an underlined event and fails if it is already defined or does not start with a letter. If it succeeds, it puts the string in the lexicon with a newly created, blank page as its payload; and it also puts that payload, under an event PI, as a constituent of the left hand part of speech.

B. 1. 13. CNIS (STC1): Allows the word IS to parse as a comparator if there is no other comparator following it in the parsing graph.

B. 1. 14. CNNCH (STC2): When an at sign parses to a number, the payload of that number is set to zero, and a flag is set signifying that this is a free variable.

B. 1. 15. CNNO (STC1): Allows the word NOT to parse as a comparator if there is no comparator following it in the parsing graph.

B. 1. 16. CNNPB (STC1): Allows the parse if there is no further boolean to the right. This is done by checking to see if there is a conjunction to the right.

B. 1. 17. CNNRE (STC1): Determines whether an attribute should be bound to a current entity. This is done by checking that there is no left parenthesis or the word OF following it in the parsing graph.

B. 1. 18. CNPRR (STC2): Allows the parse if there is no right parenthesis to the right.

B. 1. 19. CNRET (STC3): This is the dummy condition routine. It always allows the parse.

B. 1. 20. CNTS (STC1): Allows the parse to occur only if the parse is at the beginning of a sentence. This is done by following the stack and finding if there is a $A to the left.

B. 1. 21. CNTSEV (STC3): First calls the condition routine PASLOC; then collects the name of the event, failing if it is an already defined event name, and placing it in the lexicon otherwise. Thus in the re-parse that has already been caused by PASLOC, all self-references of the event may parse correctly. The event name string is replaced in the parsing graph by an event PI, with a payload, as in the lexicon, of a newly created page where the event parsed tree will reside.

B. 1. 22. CONBR (STC1): Allows the parse if there is a blank to the right of the string being parsed.

B. 1. 23. FTFOE (STC3): Same as BFOE, except that it examines the first and the third part of speech on the right hand side of the rule.

B. 1. 24. GDSP (STC3): Checks the type of command verb and succeeds if it is GENERATE, DO, SET or PUT. This is handled by looking at the feature of the VC PI.

B. 1. 25. MATIR (STC3): Not in use at present; set to CNRET.

B. 1. 26. NELSE (STC3): Examines the VC to the right of the present one, and determines whether it has a feature on signifying that it is an ELSE sentence, or, recursively, that it has a conditional clause and is followed by a VC with the same feature on.

The parse is allowed only if this is not the case.

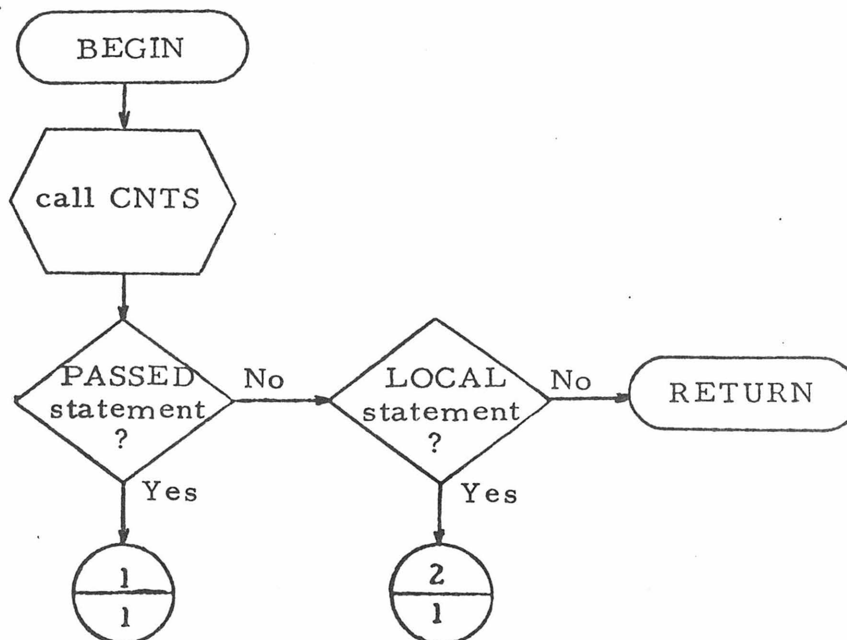
B.1.27. NOAB (STC3): Examines the constituents of the parse, and fails if there is a single attribute as a constituent.

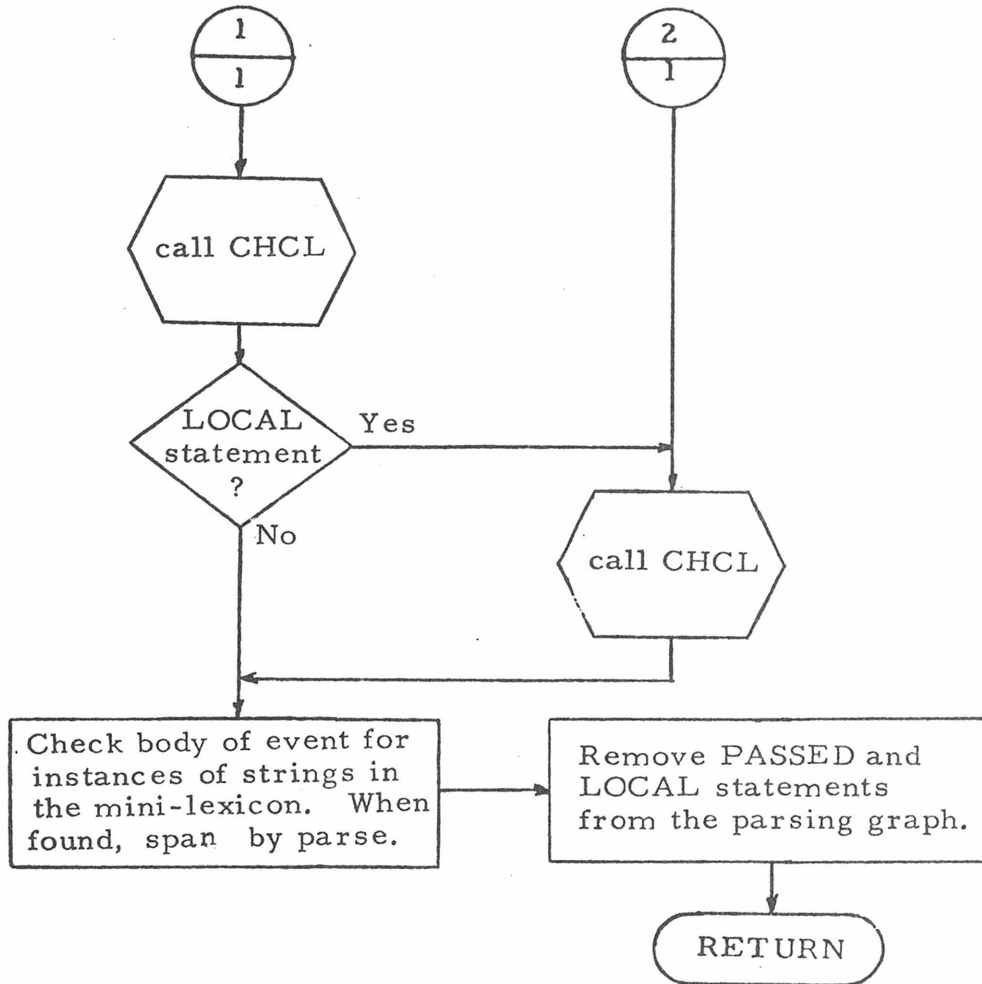
B.1.28. NOVRAB (STC3): Same as NOAB.

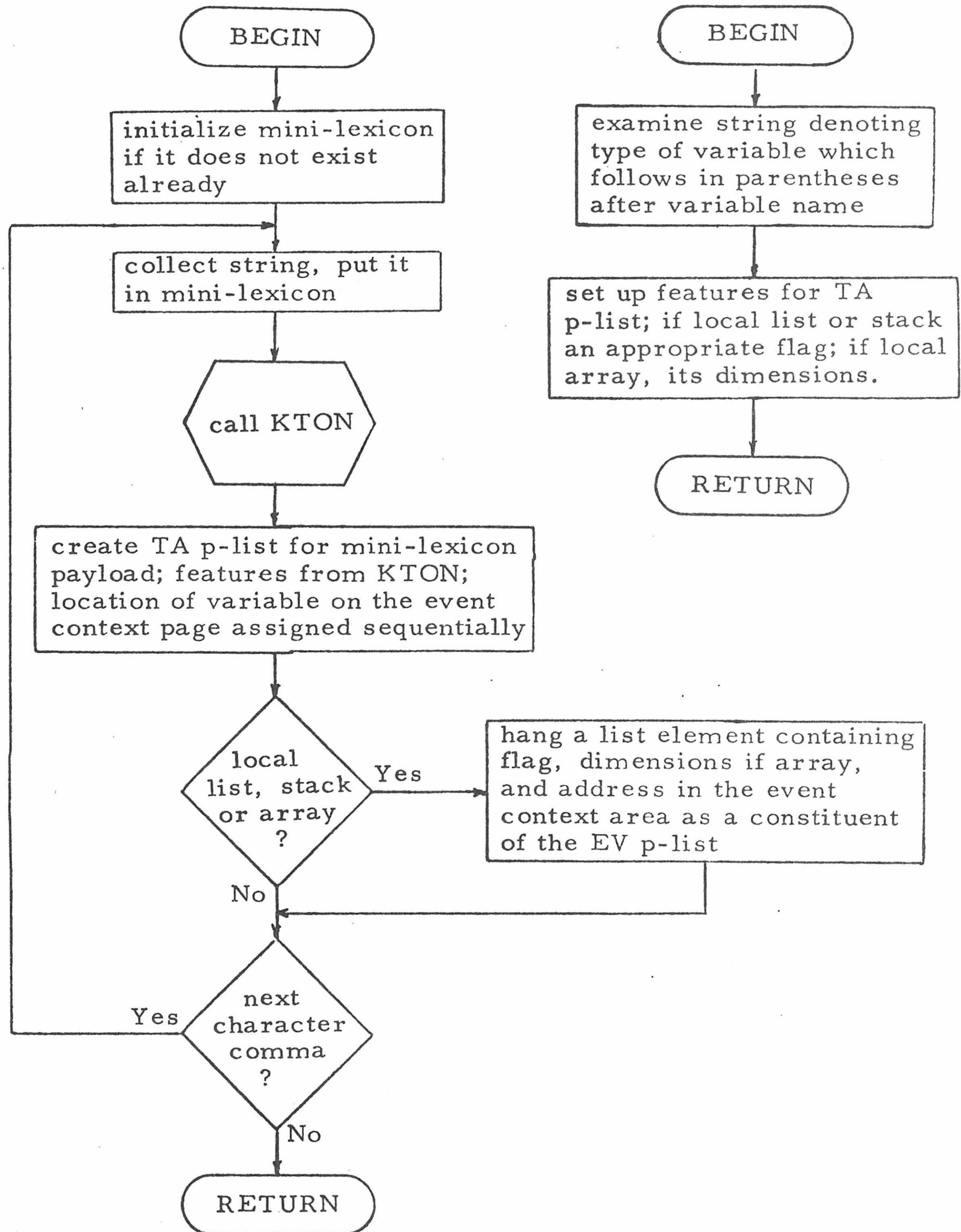
B.1.29. NOVRIN (STC3): Allows the parse if the first constituent of the quantity parsed has no constituents of its own.

B.1.30. PASLOC (STC4): After calling CNTS (and failing if CNTS fails), checks for PASSED and LOCAL statements in the event. If they exist, it collects the strings naming the variables in a mini-lexicon, with payloads of PI's referring to the event context page. Then the body of the event is checked for instances of such strings. When such a string is found, it gets spanned by the appropriate parse. The LOCAL and PASSED statements are then removed from the parsing graph. A flow chart follows:

PASLOC




1
1
2
1
call CHCL
LOCAL statement ?
Yes
No
call CHCL
Check body of event for instances of strings in the mini-lexicon. When found, span by parse.
Remove PASSED and LOCAL statements from the parsing graph.
RETURN


CHCL
BEGIN
initialize mini-lexicon if it does not exist already
collect string, put it in mini-lexicon
call KTON
create TA p-list for mini-lexicon payload; features from KTON; location of variable on the event context page assigned sequentially
local list, stack or array ?
Yes
hang a list element containing flag, dimensions if array, and address in the event context area as a constituent of the EV p-list
No
next character comma ?
Yes
No
RETURN
KTON
BEGIN
examine string denoting type of variable which follows in parentheses after variable name
set up features for TA p-list; if local list or stack an appropriate flag; if local array, its dimensions.
RETURN

B. 1. 31. POSR (STC1): Allows the parse only if there is a period or a semicolon to the right in the parsing graph.

B. 1. 32. SCCL (STC1): Allows the parse only if there is a semicolon or a colon to the left in the parsing graph.

B. 1. 33. SCLCFR (STC1): Allows the parse only if both POSR and SCCL succeed.

B. 1. 34. SCRKA2 (STC2): Succeeds if there is no right parenthesis or digit to the left in the parsing graph.

B. 1. 35. SCRDEF, SCRNDEF1, SCRNDEF2, SCRNDEF3: These are the condition routines that handle the definitional capability. They have been written for REL-English.

B. 1. 36. SCRPREN (SCRPRE): This routine is the sentence prescanner. It takes care of blanks, continuation over lines, and collects all real numbers, replacing the digits by a NU parse in the graph. It also initializes the lexicon and parses all lexical items in the graph. This routine has been written for REL-English. In the RELSIM version a prologue has been added that checks to see if the context area has been created, and, if not, proceeds to create and initialize it. The context area has been described in Chapter III.

B. 1. 37. SIFN (STC2): Collects the character string that is being parsed and hangs it as a constituent list of the PI.

B. 2. Semantic Routines

B. 2. 1. ADD (PSM6): Adds two floating-point numbers.

B. 2. 2. BIND (PSM16): Calculates a random variate with a binomial frequency function. Given N and P, it obtains N numbers uniformly random in (0, 1) and counts the number of these that are smaller than P. This count is the random variate desired.

B. 2. 3. BOR1 (PSM4): Searches a list or class to find whether a given element is in it. Returns a boolean 0 or 1 accordingly.

B. 2. 4. BOX2 (PSM6): Checks whether a list or class is empty, and returns a boolean 0 or 1 accordingly.

B. 2. 5. BPIL (PSM2): Applies a conjunction to two booleans to generate a boolean.

B. 2. 6. BREV (PSM3): Takes the complement of a boolean number.

B. 2. 7. BTST (PSM3): Compares two numerical expressions and generates a boolean depending on whether the condition on their relation is true or false.

B. 2. 8. CNPU (PSM4): Calls SEM on the first conditional expression and returns its numerical expression as an unconditional number if the boolean is true; otherwise calls SEM on the second conditional expression, essentially recursing on itself.

B. 2. 9. CO1 through CO6 (PSM3): These assign a fixed-point value to each comparator or conjunction.

B. 2. 10. CORV (PSM3): This routine reverses the negation flag on a comparator PI.

B.2.11. CURC (PSM12): Returns a pointer to the current entity of the class referenced.

B.2.12. CURT (PSM12): Returns a pointer to the current time location in the context area.

B.2.13. DEP1 (PSM1): Initializes a stack and places its name in the lexicon.

B.2.14. DER1 (PSM3): Puts the name of an attribute in the lexicon and sets up the attribute definition with respect to the proper entity class.

B.2.15. DER2 (PSM1): Initializes a system variable and places its name in the lexicon.

B.2.16. DER3 (PSM1): Initializes a list and places its name in the lexicon.

B.2.17. DER4 (PSM1): Initializes an entity class and places its name in the lexicon.

B.2.18. DER5 (PSM3): Places the parsed tree of an attribute definition on a page and a pointer to that page in the header of the appropriate entity class.

B.2.19. DEXI (PSM5): Initializes a two-dimensional array and places its name in the lexicon.

B.2.20. DIVIDE (PSM6): Divides a floating-point number by another one.

B. 2. 21. DVA1 (PSM16): Bumps the event payload and the possible local list, stack and array references from the constituent EV or DE to the DV list element. A sort of a SEMNOP or TNF: 1 effect.

B. 2. 22. ECR1 (PSM9): Creates a scratch list. Attaches the boolean with free attributes to each member of the entity class and evaluates it each time. Whenever the boolean is true, the entity referred to gets placed in the scratch list.

B. 2. 23. ECR2 (PSM9): Creates a scratch list that is the result of the given conjunction applied to the two lists given.

B. 2. 24. ECR2A (PSM9): Same as ECR2, but instead of a given conjunction, AND is assumed.

B. 2. 25. ECR2B (PSM9): Same as ECR2, but the complement of the list obtained in ECR2 is placed in the scratch list.

B. 2. 26. EXDI (PSM15): Calculates a random variate with an exponential distribution. Given k, a uniformly distributed random number r in (0, 1) is obtained and its natural logarithm is calculated. The variate returned is obtained as -k*ln(r).

B. 2. 27. EXITSIM (PSM12): Calls on the EXITLANG system macro to return to the command language and prints a message to that effect.

B. 2. 28. FAB1 (PSM23): Stacks the attribute page ID's of the constituent attributes in a data list under the attribute they parse to, for later evaluation when the class and entity referred to by the last, free, attribute is determined.

B.2.29. FAB2 PSM19): Starts the stack that FAB1 continues to build up by moving the data element of the constituent attribute under the attribute it parsed to and making its second word a link.

B.2.30. FEVP (PSM18): Frees the pages that contain the second constituent event and then calls SEM on the first constituent.

B.2.31. LASC (PSM2): Given a class, returns the last entity of that class.

B.2.32. LETN (PSM2): Given a class, and a floating-point number n, returns the nth entity in the class.

B.2.33. LISRT (PSM20): Restores the original contents of a list from a copy that was made of them at some previous point. Frees the pages of the copy.

B.2.34. LMNC (PSM2): Given a class and a number n, returns the nth from the bottom entity of the class.

B.2.35. LTST (PSM21): Compares two lists and produces the appropriate boolean value.

B.2.36. MULTIPLY (PSM6): Multiplies two floating-point numbers and returns the result.

B.2.37. NABT (PSM2): Given an attribute and an entity, this routine gets the pointer to the entity page where the payload referred to resides.

B.2.38. NORD (PSM2): Calculates a random variate with a normal distribution. Given a mean m and a deviation s, the sum t of 24 random numbers uniformly distributed in (0, 1) is obtained,

and the desired variate x is calculated as x=s*(sqrt(2)/2)*(t-12)+m.

B.2.39. NTHC (PSM2): Same as LETN except that in the input p-list the number is the first component and the class the second.

B.2.40. NUC1 (PSM1): Given a numerical expression and a boolean expression, evaluates the boolean and, if true, evaluates and returns the number. If false, returns a zero flagged to show it is a failure, rather than a real zero.

B.2.41. NUI1 (PSM12): Given the function p-list from CNDEN, two random numbers r_1, r_2 uniform in (0, 1) are obtained, and a random variate calculated as follows: If $r_1 \leq f(a+(b-a)r_2)$, where (a, b) is the interval of f, we accept $x=a+(b-a)r_2$; otherwise we obtain two new values for r_1, r_2 and try again. If $f(a+(b-a)r_2)>1$ at any time, we modify c (which was calculated in CNDEN), so that $f(a+(b-a)r_2)=1$.

B.2.42. NUI2 (PSM16): Utilizes SELFR to select one of the discrete points over which f(x) is defined and returns that point as the variate desired.

B.2.43. NUQ9 (PSM7): Obtains and returns the total number of elements contained in the class or list that is given.

B.2.44. OPA1 (PSM20): Gives the value X'30000000' to the MIN operator.

B.2.45. PASD (PSM15): Given k and p as floating-point numbers, obtains a random variate by taking the product t of k random numbers uniform in (0, 1) and setting x=t/ln(1-p).

B. 2. 46. POA1 (PSM17): Given the name of an event notice, the notice is located on the log and its address returned.

B. 2. 47. POA2 (PSM17): Given an event routine and a number, n, the nth notice of the routine is located on the log and its address returned.

B. 2. 48. POA3 (PSM17): Given an event routine, the last notice of the routine is located on the log and its address returned.

B. 2. 49. POA4 (PSM17): Given an event routine and a number n, the nth from the last notice of the routine is located on the log and its address returned.

B. 2. 50. POA5 (PSM17): Given an event routine, a notice of the event, randomly selected, is located on the log and its address returned.

B. 2. 51. POA6 (PSM17): Given a p-list of an event notice and a time figure, the search for the notice is limited to log entries under that time.

B. 2. 52. POA7 (PSM17): Given a number n, the nth event notice on the log is located and its address returned.

B. 2. 53. POA8 (PSM24): The last event notice is found on the log and its address returned.

B. 2. 54. POA9 (PSM24): Given a number n, the nth from the last event notice is located on the log and its address returned.

B. 2. 55. POB1 (PSM24): An event notice is selected randomly from the log and its address returned.

B. 2. 56. POID (PSM9): Given k, obtains random numbers in (0, 1) and takes their product t until $t<e^{-k}$. Then returns the number of random numbers in that product as a random variate with Poisson distribution.

B. 2. 57. RANC (PSM2): Given a list or class, one of its elements is selected at random and returned.

B. 2. 58. RAND (PSM15): Given an interval (a, b), an integer random variate uniformly distributed in (a, b) is returned.

B. 2. 59. RSCR (PSM22): Obtains list page ID's from the data p-list of the first constituent, and frees the pages of all these lists.

B. 2. 60. SEMABS (PSM5): Given a number, returns its absolute value.

B. 2. 61. SEMCOS (PSM6): Given a number,considers it to refer to radians and calculates its cosine.

B. 2. 62. SEMCOTAN (PSM6): Given a number in radians, calculates and returns its cotangent.

B. 2. 63. SEMEXP (PSM7): Given a number n, calculates and returns e^n.

B. 2. 64. SEMFP (PSM5): Given a number, returns its fractional part.

B. 2. 65. SEMIP (PSM5): Given a number, returns its integer part.

B. 2. 66. SEMLN (PSM7): Given a positive number, calculates and returns its natural logarithm.

B. 2. 67. SEMLOG (PSM7): Given a positive number, calculates

and returns its base 10 logarithm.

B. 2. 68. SEMMAX (PSM5): Given two numbers, returns the larger one.

B. 2. 69. SEMMIN (PSM5): Given two numbers, returns the smaller one.

B. 2. 70. SEMMOD (PSM5): Given two numbers, n_1 and n_2, returns n_1 mod n_2.

B. 2. 71. SEMNOP (PSM9): Moves the data element of the first component to become the data element of the part of speech parsed.

B. 2. 72. SEMPOW (PSM7): Given two numbers a and b, calculates and returns a**b.

B. 2. 73. SEMRET (PSM3): Attaches a zeroed out data element to the part of speech parsed.

B. 2. 74. SEMSIGN (PSM5): Given a number, returns 1 if the number is positive, 0 if the number is zero and -1 if the number is negative.

B. 2. 75. SEMSIN (PSM6): Given a number in radians, calculates and returns its sine.

B. 2. 76. SEMSQRT (PSM6): Given a number, calculates and returns its square root.

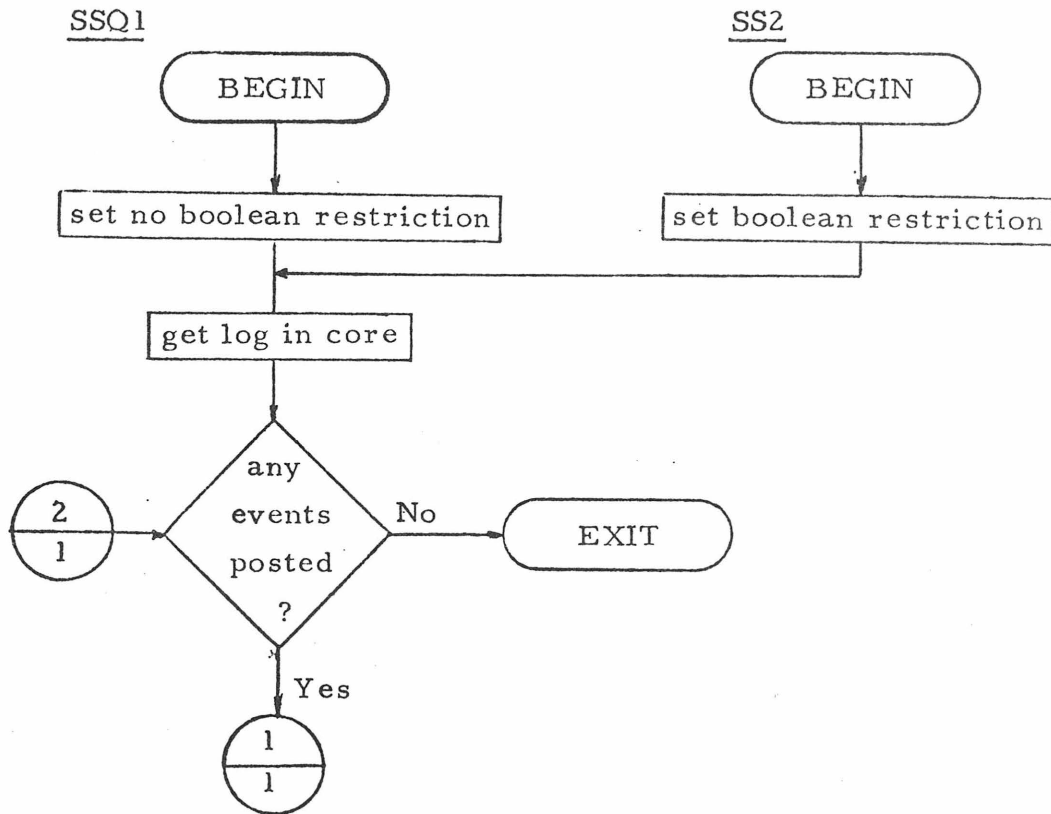
B. 2. 77. SEMTAN (PSM6): Given a number in radians, calculates and returns its tangent.

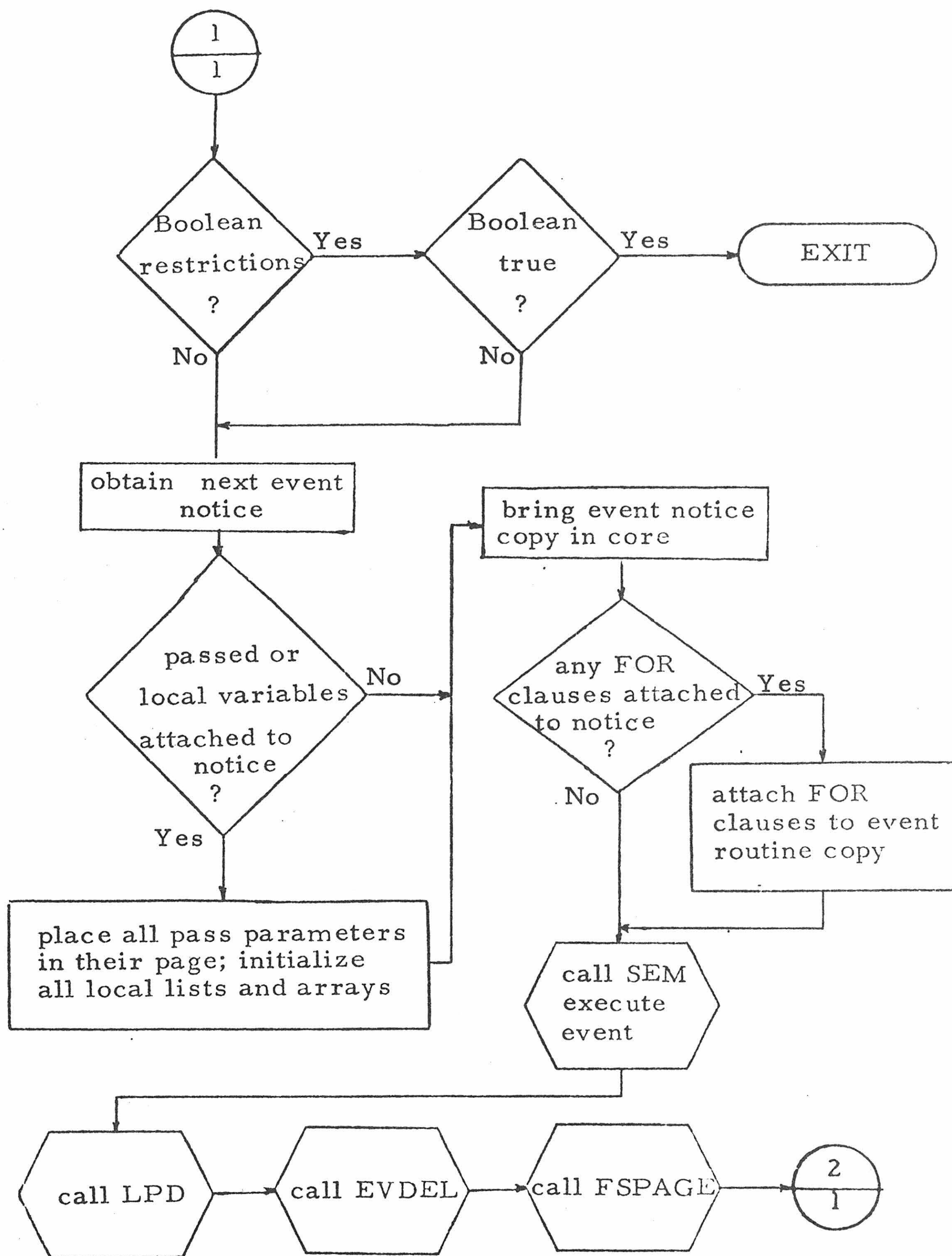
B. 2. 78. SS2 (PSM8): Same as SSQ1, but a boolean is given that will stop the run when it becomes true. See flow diagram below, under SSQ1.

B.2.79. SSJ1 (PSM24): After a PAUSE has stopped a run, this routine recreates the conditions existing before the pause and restarts the run.

B.2.80. SSJ2 (PSM24): Transforms the simulation's data into an REL data base. Entity classes become English classes and attributes form English relations.

B.2.81. SSQ1 (PSM3): Causes a simulation run. Obtains an event notice from the log, brings in the appropriate event routine, and executes it (by calling SEM on a copy of its parsed tree). Deletes the notice and obtains the next one. Stops the run when the log is empty. A flow diagram follows.




1
1
Boolean restrictions ?
Yes
No
Boolean true ?
Yes
No
EXIT
obtain next event notice
passed or local variables attached to notice ?
No
Yes
place all pass parameters in their page; initialize all local lists and arrays
bring event notice copy in core
any FOR clauses attached to notice ?
Yes
No
attach FOR clauses to event routine copy
call SEM execute event
call LPD
call EVDEL
call FSPAGE
2
1

B. 2. 82. SSQ7 (PSM19): Takes the parsed tree of an event routine and stores it on pages after modifying the tree by including an extra clause (with XXX as its semantic routine) that will create all local lists and place their ID's on the event context page.

B. 2. 83. SSR1 (PSM3): If no clause has printed out a message this routine prints out the message 'OK'.

B. 2. 84. SUBTRACT (PSM6): Given two numbers n_1 and n_2, calculates and returns $n_1 - n_2$.

B. 2. 85. TASEM (PSM23): Obtains a displacement and adds it to the top of the event context page to obtain a page ID for a local or parsed variable.

B. 2. 86. UNARYM (PSM6): Given a number returns its complement.

B. 2. 87. UNID (PSM2): Given an interval (a, b) calculates a random variate uniform in that interval. Obtains a random number r uniform in (0, 1) and returns a+(b-a)r.

B. 2. 88. VCA1(PSM15): Given an event routine page ID, generates an event notice for it at the current time.

B. 2. 89. VCA2 (PSM19): Puts a name for an event notice being generated in the lexicon.

B. 2. 90. VCA3 (PSM13): Puts a VC parsed tree out on a page and sets up a notice on the log for it to be executed after an interval as requested.

B. 2. 91. VCA4 (PSM19): Deletes an event notice from the log, given its location on it.

B. 2. 92. VCA5 (PSM24): Writes out an event notice given its location on the log.

B. 2. 93. VCA6 (PSM24): Writes out the contents of the entire log.

B. 2. 94. VCA7 (PSM18): Puts a name assigned to an entity in the lexicon.

B. 2. 95. VCA8 (PSM4): Puts a given payload in a given location on a page.

B. 2. 96. VCA8A (PSM4): Transforms a given floating-point number to integer and puts the result in a given location on a page.

B. 2. 97. VCA8B (PSM16): Puts a given element (given by its address in the list or class) in the current entity location on a page.

B. 2. 98. VCB1 (PSM22): Removes an entity from a class, its name, if any, from the lexicon, and destroys any list or array attributes attached to it.

B. 2. 99. VCB2 (PSM18): Puts a name assigned to an event notice in the lexicon and a pointer to the lexicon entry in the event notice.

B. 2. 100. VCB3 (PSM22): Deletes an entry from the lexicon given a page ID type payload.

B. 2. 101. VCB4 (PSM24): Saves all the list area on pages and copies all scratch pages; then terminates execution of the present sentence (a GO sentence) and returns control to the users.

B. 2. 102. VCB6 (PSM13): Puts a copy of the VC p-list out on a page and creates an event notice for its execution after an interval if the boolean is not met; also executes a copy of the same VC p-list

at the current time.

B.2.103. VCB7 (PSM21): Writes out a line of numbers and characters as given.

B.2.104. VCB8 (PSM14): Evaluates the entity list given and splits it into n lists, each containing exclusively members of one class. Places the ID of each element in the current entity queue of the appropriate class. Modifies the input tree to look like n FOR statements, each with one of the lists as payload. Calls SEM on the VC given.

B.2.105. VCC1 (PSM8): Puts the second component VC, which forms the contents of the DO loop, in the location of the DO VC in the first component, thus applying all modifying clauses of the loop to this VC directly, and calls SEM on the changed first component.

B.2.106. VCC3 (PSM16): Selects one of the VC's in the SELECT loop, using SELFR; places that VC in the location of the SELECT VC in the first component; calls SEM on the changed first component.

B.2.107. VCC4 (PSM20): Given an array and a string of numbers, places the numbers sequentially (last index moving fastest) in the array.

B.2.108. VCC5 (PSM4): Places an element in the last position of a list.

B.2.109. VCD1 (PSM24): Places an element before another element in a list.

B.2.110. VCD2 (PSM24): Places an element after another element

in a list.

B. 2. 111. VCD3 (PSM24): Given a number n and an element, places the element in the nth position in the list, or if there are less than n elements, in the last position.

B. 2. 112. VCD7 (PSM24): Writes out an event routine in the form it was written in as a declaration.

B. 2. 113. VCE1 (PSM10): Removes an element from a list.

B. 2. 114. VCE2 (PSM24): Writes out an array.

B. 2.115. VCE4 (PSM22): Frees all pages of an array and removes its name from the lexicon.

B. 2. 116. VCE5 (PSM10): Given a number n, and a list, removes the nth element of that list.

B. 2. 117. VCE6 (PSM10): Removes the last element of a list.

B. 2. 118. VCF1 (PSM22): Frees the pages of a list and removes its name from the lexicon.

B. 2. 119. VCF3 (PSM11): Frees all pages of all classes, all pages of list or array attributes of their entities, all attribute definition pages, and deletes all attribute, entity and class names from the lexicon.

B. 2. 120. VCF4 (PSM11): Frees all event routine pages and deletes their names from the lexicon.

B. 2. 121. VCF5 (PSM11): Frees all pages of all lists and deletes their names from the lexicon.

B. 2. 122. VCF7 (PSM11): Frees all system variable pages and

deletes their names from the lexicon.

B.2.123. VCF8 (PSM21): Frees all pages of the log and deletes all event notice names from the lexicon.

B.2.124. VCF9 (PSM21): Empties the log, all lists and classes (removing event notice names and entity names from the lexicon and freeing all list and array attribute lists) and resets the simulation time to zero.

B.2.125. VCFA (PSM21): Empties a list of all its elements.

B.2.126. VCFB (PSM21): The elements in an entity scratch list are deleted from their respective classes.

B.2.127. VCG2 (PSM11): Frees the pages of all arrays and removes their names from the lexicon.

B.2.128. VCH1 (PSM19): Evaluates a copy of the given boolean, and if it is false, evaluates the given VC. Then repeats the same action, until the boolean evaluated is true, in which case it returns.

B.2.129. VCI1 (PSM14): Puts the entity given at the top of the current entity stack of the appropriate class, and calls SEM on a copy of the VC given. On return from SEM bumps the current entity stack, and if this action does not empty it, again calls SEM on a copy of the VC.

B.2.130. VCL1 (PSM10): Sets the contents of a given list equal to those of another.

B.2.131. VCQ1 (PSM10): Adds the contents of one list to the contents of another one.

B. 2. 132. VCQ2 (PSM9): Removes the contents of one list from the contents of another one.

B. 2. 133. VCR1 (PSM4): Adds a new entity to an entity class and evaluates all its attributes that are defined.

B. 2. 134. VCR2 (PSM9): Assigns values as given to attributes of a new entity that has just been created by VCR1.

B. 2. 135. VCR3 (PSM13): Posts an event to reverse the action of the VC given after a time interval also given.

B. 2. 136. VCR4 (PSM20): Performs as VCR1 and also adds a name for the new entity in the lexicon.

B. 2. 137. VCR5 (PSM10): Performs the action of VCR1 n times according to a number given.

B. 2. 138. VCT1 (PSM16): Hangs the values of all pass parameters from the event notice generated, making new copies in the case of local arrays or lists being passed.

B. 2. 139. VCY1 (PSM10): Defines new attributes for an entity class prior to creating a new entity of that class as described in VCR1.

B. 2. 140. VCZ3 (PSM15): Writes out the contents of a list or an entity class.

B. 2. 141. VRA1 (PSM24): Obtains the element of a list or class that is next to the current one.

B. 2. 142. VRA2 (PSM24): Obtains the element of a list or class that is next to a given element.

B. 2. 143. VRA3 (PSM24): Obtains the element of a list or class that is previous to the current one.

B.2.144. VRA4 (PSM24): Obtains the element of a list or class that is previous to a given element.

B.2.145. VRG1 (PSM8): Given an unattached attribute, this routine attaches it, if unambiguously possible, to the current entity of the appropriate class, and returns its address.

B.2.146. VRR1 (PSM20): Obtains a pointer to an element of an array, given its indices.

B.2.147. VRSD (PSM18): Obtains a pointer to the location in the context area where the seed for the random number generator is kept.

B.2.148. VRY5 (PSM20): Depending on the operator given, obtains the maximal or minimal element in a numerical list.

B.2.149. VRY6 (PSM20): Depending on the operator given, obtains the maximal or minimal number valued attribute from all entities of a class that are members of a given list.

B.2.150. VRY7 (PSM20): Same as VRY6 but looks at all entities of the class.

B.2.151. VTNU (PSM4): Given the address of a variable returns the payload.

B.2.152. VTNUIF (PSM7): Given the address of a fixed-point variable, transforms the payload to floating-point and returns it.

B.2.153. XXX (PSM23): Creates all lists and arrays to be local for an event notice and places their ID's in the event notice context page.

B. 3. Utilities

B. 3. 1. ABLDL (PSM22): On call, R1 points to an entity in core and R3 contains the class page ID. The utility finds all list attributes of the entity and deletes these lists. R1 and R3 are returned unchanged.

B. 3. 2. CKDF (PSM1): Given a p-list of a part of speech with the characters representing its name as constituents in R1, the ID of the part of speech in the top byte of R2, and its features in the bottom halfword of R3, this utility searches the lexicon and checks if such a name for such a part of speech has already been defined. If so, R2 returns the lexical page ID of that definition; otherwise it remains unchanged. R1 and R3 are always returned unchanged.

B. 3. 3. CLST (PSM20): Given the page ID of a list in R1 this routine copies all the pages of the list (unmodified) onto newly created pages which are linked together by the second word. The page ID of the first page of the copied list is returned to R1.

B. 3. 4. CSTD (PSM20): Given the core address of the top page of an entity class in R1 and the page ID of an entity of that class in R4, this utility places the entity at the top of the current entity stack for the FOR statement.

B. 3. 5. CSTU (PSM14): Given the core address of the top of an entity class in R1, this routine removes an entity from the top of the current entity stack and places it in the current entity location of the class page. R1 is returned unchanged. If the

flag of the entity in the stack is not zero, R0 returns its location in the stack (as a page ID) and R8 returns the page ID of the stack. Otherwise R0 is returned as 0, and R8 is destroyed but carries no information.

B.3.6. DATEC (PSM22): On entry, R2 points to an EC p-list. The routine deletes all attribute definitions for this class and all attribute names from the lexicon that refer to no other class. On return R2 is unchanged.

B.3.7. DEDEF (PSM1): On entry, R1 points to the p-list of a pos with the characters representing its name as constituents; R0 has the ID for the part of speech in the bottom byte; R2 contains the page ID that the part of speech should parse to, and R3 contains the features of the pos in the bottom halfword. This routine puts an entry for the name of this part of speech in the lexicon.

B.3.8. DELLEX (PSM11): Deletes the entire lexicon of the version.

B.3.9. DELNN (PSM11): On entry, R1 points to a p-list of a part of speech. The name string of the part of speech is not included but the lexicon is searched on the basis of pos, flags and payload page ID. The entry is deleted. R0 returns 0 if the entry was not found and 1 if it was deleted successfully. R1 remains unchanged.

B.3.10. DELNNQ (PSM22): Same as DELNN; included in this deck also to ensure efficiency in the running of certain semantic routines that reside in the same deck.

B.3.11. DELPOS (PSM11): On entry, R1 contains the ID of a part of speech in the bottom byte. All entries in the lexicon of that kind of part of speech are deleted. R1 remains unchanged.

B.3.12. DELSL (PSM11): On entry, R0 contains the number of characters in the name of a part of speech; R1 points to the character string; and R2 points to the p-list of the part of speech. The entry referred to is deleted from the lexicon. On exit, R1 and R2 are unchanged, and R0 is 0 if the entry was not found and 1 if it was successfully deleted.

B.3.13. DER7 (PSM19): Requires the p-list of a part of speech with a character string as constituents in R1, its ID in the top byte of R2 and the page ID to be assigned to it in the CXT location of the context area. Puts an entry for the string in the lexicon.

B.3.14. DPG (PSM11): Obtains a displacement in R2 and gets a page ID from the context area location with that displacement. Deletes that page and all pages linked to it by pointer ID's in the first word of each page. Leaves only the last page and puts the ID of its top back in the appropriate word of the context area.

B.3.15. EEXP (PSM7): Requires a floating-point number in FPR-0. Calculates the exponential of the number, and returns it in FPR-0. This utility was written for REL-English.

B.3.16. ENDEL (PSM10): On entry, R3 contains the page ID of a class and R4 that of an entity belonging to it. This routine removes the entity from the class. R3 remains unchanged.

B. 3. 17. EVDEL (PSM8): On entry, R5 points to the top of the log, R3 points to the log header for a specific time and R4 points to an event notice to occur at that time. The notice is deleted and its name, if any, is removed from the lexicon. R4 and R5 are returned unchanged.

B. 3. 18. EVGEN (PSM15): Obtains a pointer to the context area in R2, the page ID for an event routine in R3, and a time figure in FPR-4. Posts a notice for that event under the time given in the log. Returns R2 and R3 and FPR-4 unchanged.

B. 3. 19. EVLVR (PSM13): On call R4 points to a VC p-list. The routine evaluates all numerical and entity variables and lists in the p-list. If there are any scratch lists evaluated, they get copied onto regular pages, which are used as the VR payloads; and a VC p-list is built up with all such VR's as constituents and RSCR as its semantic routine. This VC p-list is returned in R5; if there are no scratch lists evaluated R5 is returned zeroed out. R4 is returned unchanged.

B. 3. 20. EXPI (PSM7): On call FPR-0 and FPR-2 contain two numbers, n_1 and n_2. The routine calculates n_1**n_2 and returns the number in FPR-0.

B. 3. 21. FIND (LEXUTIL): On entry R0 contains the length of a lexical string, R1 points to an address in the lexicon where a search is supposed to start, and R2 points to the lexical string. This utility finds the entry for the string in the lexicon and returns

the address for it in R1. If the search is unsuccessful, R1 is zero on return. R2 returns unchanged.

B. 3. 22. FIPID (PSM12): On entry, R2 contains a page ID. The lexicon is searched for an entry with this page ID as payload. On return, R2 contains the ID of the lexicon page where the entry occurs. If the search fails, R2 is returned zeroed out.

B. 3. 23. FIPSTR (PSM18): On entry R3 contains a page ID. The action is the same as in FIPID, but on return R2 contains the length of the referrent lexical string, and R3 contains the page ID in the lexicon where the string begins. If the search fails, R2 and R3 are returned unchanged.

B. 3. 24. FLA (PSM14): When called from VCI1, R5 contains 1 or -1. The second word of each entity stack entry is modified by having R5 added to it. When called from VCB6, R5 contains X'101', in which case the second word of each entry is OR'd with X'100' and then 1 is added to it. X'100' acts as a flag signifying the entry is a list; the last byte counts the level of recursion of the FOR clause routine.

B. 3. 25. FNEV (PSM12): Given the p-list of a function with free variables (from a DENSITY statement) in R1, and a number in FPR-0, this utility copies the p-list, substitutes the contents of FPR-0 in each instance of the free variable, and calls SEM on the copy. On return, R1 points to the evaluated copy of the p-list and FPR-0 remains unchanged.

B. 3. 26. FPTOIN (PSM2): Receives a floating-point number in FPR-2. Transforms it into a fixed-point number, which is returned in R2. The contents of FPR-2 remains unchanged.

B. 3. 27. LEXUTIL (LEXUTIL): Places a lexical item in the lexicon. On entry, R0 points to a lexical string preceded by one byte containing its length, and R1 to the p-list of the part of speech; on return, R0 and R1 are unchanged.

B. 3. 28. LISLIS (PSM10): Sets the contents of one list to those of another one. On entry, R3 contains the page ID of the list that becomes a copy and R4 that of the list that is copied. On return, the registers are unchanged.

B. 3. 29. LOGE (PSM7): On entry, FPR-0 contains a floating-point number. The routine calculates the natural logarithm of the number and returns it in FPR-0.

B. 3. 30. LOG10 (PSM7): On entry, FPR-0 contains a floating-point number. The routine calculates the base 10 logarithm of the number and returns it in FPR-0.

B. 3. 31. LPD (PSM16): Given the core address of an event notice in R4, this routine frees the pages of all lists and arrays local to the notice, preparatory to the deletion of the notice from the log.

B. 3. 32. NTDEL (PSM11): Deletes an entry from the lexicon given its page ID payload. On entry, R4 contains the page ID; it remains unchanged. On exit R0 is 0 if the deletion attempt failed,

and 1 if it succeeded.

B. 3. 33. PUTELT (PSM4): On entry, R4 contains the page ID of a class or list. R6 is 0 if it is a class and 1 if it is a list; and R5 contains the payload if it is a list. The routine places a new element in the class (without assigning the attribute values) or in the list. On return, R4 and R5 are unchanged, R6 contains the page ID of the new element, and R0 is 1 if a new page had to be created for continuation of the class or list, and 0 otherwise.

B. 3. 34. RANDOM (PSM2): Obtains a seed number from the context area. Multiplies the seed by 5^{13} and takes the result mod 2^{31} as the next integer random number, r. Puts this number back in the context area as a seed for the next one. Then calculates $r/2^{31}$ to obtain a real number uniformly distributed in (0, 1); this is returned in FPR-0. The integer random number in $(0, 2^{31})$ is returned in R2.

B. 3. 35. SELFR (PSM16): On entry R1 points to a p-list associating probabilities to parts of speech. On the basis of these probabilities, one of the parts of speech is selected and returned, with semantics not yet performed on it.

B. 3. 36. SRCG (PSM14): Finds any GENERATE clauses in the structure modified by a FOR clause and hangs the current entity under it. On entry, R2 points to the VC p-list, R3 contains the current entity, and R4 the past current entity. On return they all remain unchanged.

B. 3. 37. UTYKA1 (PSM5): Given a floating-point number in FPR-0, this utility returns an EBCDIC decimal representation of that number in an LDD list structure with the characters in the last two words. R1 points to this list on return.

B. 3. 38. VCC2A (PSM12): Currently not used.

B. 3. 39. VCF3A (PSM11): Does all the freeing of pages for VCF3.

B. 3. 40. VCF5A (PSM11): Does all the freeing of pages for VCF5.

B. 3. 41. VCF7A (PSM11): Does all the freeing of pages for VCF7.

B. 3. 42. VCFAA (PSM21): On entry, R1 points to a list. The list is emptied of all its elements. On return, R1 is unchanged.

B. 3. 43. VCG2A (PSM11): Does all the freeing of pages for VCG2.

B. 3. 44. VCZB (PSM12): On entry, R1 points to an in core entity and R6 to a location in a buffer. A name or description of that entity, in EBCDIC, is placed in the buffer. R1 remains unchanged, and R6 points to the next location in the buffer.

B. 3. 45. VCZC (PSM5): On entry, R1 points to a floating-point number and R6 to a location in a buffer. A decimal representation of that number, in EBCDIC, is placed in the buffer. R1 remains unchanged, and R6 points to the next location in the buffer.