1

]

CESIMO Working Paper IT-9608

GLIDER Reference Manual

Version 4.0 (for MS-DOS operating system)

CESIMO Working Paper IT-9608

GLIDER Reference Manual

Version 4.0 (for MS-DOS operating system)

August, 1996

CESIMO (Simulation and Modeling Research Center) Universidad de los Andes, Mérida, Venezuela
phone +58-74-402879; fax +58-74-402872; email cesimo@ula.ve

Contents

Chapter 1

INTRODUCTION

1.1 Characteristics of the GLIDER Simulation Language

GLIDER[1] is a language for simulation of both continuous and discrete systems. The system to be simulated is seen as a set of subsystems, that exchange information in different ways. The subsystems can store, transform, transmit, create, and eliminate information. GLIDER has the following features to represent systems and information processes:

- Subsystems are represented by **nodes** of a **network**. Nodes may be of different **types** according to their functions.
- Information transformation is described and performed by **program code** associated to each node.
- Information exchange among nodes is done by the action of the code on shared variables or files, and also by passing **messages** from a node to other.
- Information is stored in variables, files or in **lists** of received messages in the nodes.

Two simple examples follow to give a general idea how GLIDER programs look like.

1.1.1 Example of discrete simulation

. Railroad System[2].

```
In a simple railroad system, trains, with a number of coaches equal
to a random number from 16 to 20,  depart from a station each 45
minutes. After 25 minutes of travel, the trains reach their destination.
The program writes the time of arrival and the number of coaches.
For clarity, in all the manual, GLIDER and Pascal reserved words are in
upper case letters; user identifications are in lower cases with normally
the first letter in upper case.

NETWORK
 Depart   (I) Railroad      :: Coaches := UNIFI(16,20); IT := 45;
 Railroad (R) Destination   :: STAY := 25;
 Destination (E)            :: WRITELN('A train arrives at time:',
                                   TIME, 'Its length is', Coaches);
                                PAUSE;
```

[1] A complete set of examples can be found in the GLIDER examples book or in the directory DEMOS of the GLIDER disk.
[2] The GLIDER and PASCAL reserved words are all in capital letters. In the case of user words, only the first letter is capital.

```
INIT TSIM := 1300; ACT(Depart, 0);

DECL MESSAGES Depart(Coaches: INTEGER);
          STATISTICS ALLNODES;
END.
```

The program has the following **sections**:

- Heading section with title and explanations.
- NETWORK section that describes the nodes (subsystems) and the relations among them.
- INIT section that sets the simulation time and the node to be firstly activated at simulation start. It may include other initializations (see Chapter 3).
- DECL section that declares the structure of the messages and the statistics required. It may include declarations of other variables and elements of the program (see Chapter 2).

It follows a description of the NETWORK section and its functions: After each node name follows a letter between parenthesis indicating its type and the **successor** node to which messages can be sent. **Depart** is a node of I (Input) type. When executed, it generates a message and sends it to the successor node **Railroad**. The message in this example represents a train. It has an integer type field called **Coaches**, that indicates the number of coaches. In the code of the node (after the separator ::) this field is set to a value given by a GLIDER function UNIFI that produces a random integer number, in this case between 16 and 20. A new activation of the **Depart** node is scheduled after an Interval Time of 45 time units.

The node **Railroad** is of R (Resource) type. When executed, it processes the message and it stores it temporarily in an internal list (IL). The code sets, by means of the GLIDER instruction STAY, that the message must remain in the node for a time equal to 25. When the message is released. it is sent to the node **Destination**.

The node **Destination** is of E (Exit) type. It takes the message, executes the instruction of the code (it writes, for instance: "A train arrives at 140. Its length is 17") and it deletes the message. See that the value (17) is transported by the message in the variable **Coaches**. After writing, the program stops (instruction PAUSE) and when any key is pressed the simulation continues. New trains are generated in **Depart** each 45 units time, then they pass to **Railroad**, and so on.

The above source program is the input to the compiler system that translates it and runs the simulation. The **simulation time** is not the real processing time. It takes the value at each event. At the beginning is 0, when the first train arrives is 25, when the second train departs is 45, when the second train arrives it is 70, when the third train departs it is 90, etc.

1.1.2 Example of continuous simulation

.

```
Plantation with periodic cuts.
In a plantation the quantity of wood growths according to a
logistic curve whose differential equation is:
                    w' = k * (1 - w / wm) * w
When the quantity is greater than wm, a cutting process starts
which decides the proportion of wood to be extracted according
to a function of the actual price, p(TIME) and the mean price pm.
The time for the cut process is negligible compared with the
rates of growth.

NETWORK
```

```
Growth (C) :: w' := k * (1 - w / wmax) * w;
                      if w >= wm then ACT(Cut, 0);
      GRAPH(0, 50, BLACK; TIME: 7: 0, WHITE; W: 6: 0,Wood, 0, 10000, GREEN;
                               price(TIME): 6: 0, Price, 0, 100, YELLOW);

Cut    (A) :: w := w - MIN(0.3 * wm, MinCut + MAX(0, cc*(price(TIME) - pm)));

INIT TSIM := 50; ACT(Growth, 0);
         k := 0.21; wm := 6000; MinCut := 500; cc := 40.2; pm := 24;
         w := 100; wmax := 9500;
         DT_Growth := 0.125;
         price := 0, 30 / 10, 20 / 20, 15 / 30, 25 / 40, 30 / 50, 42 /
                 55, 45;
DECL VAR k, wm, wmax, MinCut, cc, pm: real; w: CONT;
           GFUNCTIONS price SPLINE (real): real: 8;
END.
```

The type C node **Growth** solves the differential equation. When `w >= wm` the node **Cut**, of A type, is activated. This reduces the mass of wood by a quantity that is 30% of the `wm` at most, and `MinCut` at least. Within these limits it depends on the difference between actual price and average price `pm` of the wood. The variable w must be declared of CONT (continuous) type.

The time function `price` is given by the 7 pairs of values indicated in the INIT section. For the time 0, its value is 30; for the time 10, it is 20; etc. It is declared of real argument, real value and the interpolation is made by a SPLINE algorithm.

A graph is programmed to display, during the run, w and the price as a function of TIME.

In the INIT section, values to the parameters and to the initial value of the variable w are given. The integration interval DT_Growth is also set. All variables are declared in the DECL section.

In GLIDER programs the basic features are implemented in the following way:

- Each node is specified by a name, a node type, and normally a list of successors (nodes to which it can send messages). The **type** of the node must be defined. The type implies specific ways of being activated and processing the messages.
- The code can include:
  - Instructions in Pascal language (Turbo Pascal in the present implementation).
  - Structured GLIDER instructions.
  - Procedures and functions of the GLIDER library.
  - System generated instructions.
- Information exchange among nodes is accomplished by allowing the code of different nodes to share some of the data (global variables, files, fields of the messages). The messages are structured like records in the Pascal language. The user can declare different types of records with different types of fields. They may be used to represent entities traveling from one node to another.
- Information is stored in variables, files and in messages. Messages are stored in two basic lists that are associated with some nodes:
  - **EL** (Entry List) of received messages to be processed by the node. They may be used to represent queues or waiting lines.
  - **IL** (Internal List) where messages can be stored until they are extracted. Usually they are used to represent entities using a facility or remaining in delay lines.

The abbreviations EL and IL will be respectively used for Entry List and Internal List, with the plurals ELs and ILs.

In addition to the global variables and variables local to nodes, that are declared by the user, the system introduces global **field variables** for each field of each message structure declared by the user. The name of these variables is equal to the name of the field. When a message is processed, the values of the fields of the message are transferred to the field variables. So the user's code can use and change these values referring to them by its simple name. When the process of the message is over, the values of the field variables are passed to the fields of the message.

When messages of different structure have some fields with the same name, they correspond to the same field variable. This allows to deal with different entities sharing common properties.

The execution of the code of a node is called the **activation** of the node. At a point during the simulation it may be necessary to schedule the activation of certain nodes for the future. The schedule is kept in a list called **FEL** (Future Event List). Each element in the FEL represents a pending event. It contains a reference to the node to be activated and the time at which the activation must take place. This list is automatically processed by the system, but the user may also handle it through special GLIDER procedures.

The order of activation of the nodes is essential to the simulation. An event (activation fact) begins with the processing of the element of the FEL, that has the minimum time value. The node referred by this element is activated (executed) and this starts the event execution.

The code of the node may cause changes in the values of the variables, execution of procedures and message processing. In particular this execution may cause message exchanges. It can also change conditions that allow other nodes to move messages or change values of variables. All this must be done in these events. To perform all these changes it follows a scanning of all the nodes that could be affected by the execution of the node. These scanned nodes are then activated. The scanning is repeated cyclically until no further movements of messages are possible. The event processing is then ended and the following event from the FEL will be considered. During the execution of an event the state of the system changes and new future events are scheduled. The value of the simulation time do not change during the transformations produced in the event. When a new event is executed the time variable is updated to the time of the event indicated in the FEL. So the simulation process goes on until an end condition specified by the user finishes the simulation run. The user must be aware of the two ways in which a node can be activated: **by an event that refers to the node**, which starts a new event, or **by scanning of the network during an event**.

1.2 Types of nodes

A summary of the characteristics of the predefined node types is shown in the following list. **Any node can be activated by an event that refers to it**. Others (not all) may be also activated during the network scanning process. Some node types have EL and IL, others only EL, others none of them. When activation is attempted by the scanning process, it may be inhibited in certain conditions (EL empty). Otherwise **the code is executed each time that the node is scanned**. The user can avoid these repetitions by means of special instructions. All activation of a node starts a scanning process with the exception of type C nodes. See Chapter 4 for a detailed account of the processing of the different node types.

- **I (Input)** Generates messages that enter the simulation process. It creates messages and sends them to the successor nodes. It has neither EL nor IL. It may only be activated by an event that refers to it. It is never activated again by the scanning of the network.
- **L (Line)** Simulates queue discipline. The messages that enter its EL are passed to its IL where they remain in the order indicated by the code. This order may be FIFO (First In First Out) LIFO (Last In First Out) or any other programmed by the user. It is activated by an event or during the scanning of the network if the EL is not empty.
- **G (Gate)** Stops or allows message flow. It has an EL for the retained messages. It may be activated by an event or during the scanning of the network if the EL is not empty. The instruction STATE, used to change the state of the gate, is only executed if the node is activated explicitly by an event which refers to the node.

- **R (Resource)** Simulates resources used by messages. A real value (called the node capacity) is associated to the Resource type node. The messages in the EL represent entities that demand a certain quantity of that capacity during a certain period of time. **The code has two parts**. One examines the incoming messages, the other processes the outgoing messages. The EL is examined and, for each message, it is checked if the demanded quantity is available. If it is so, the message is moved to the IL, the time of future depart is scheduled in the FEL, and the quantity of resource used for the message is subtracted from the available capacity. If there is not enough capacity, the message remains in the EL. When the depart event is executed, the message with the corresponding time of depart is extracted from the IL and it is sent to the successor node. If there are more than one successor, a copy is sent to each of them. The user can also control this process of message sending using the RELEASE instruction. Each time the node moves a message, the network is scanned. The part of the code that processes the departing message is only activated by a departing event and executed only once in the event process. The part that deals with the incoming messages in the EL may be activated during the scanning of the network, but only if the EL is not empty.

- **D (Decision)** Selects messages from predecessor nodes and sends them to successor nodes. It has an EL for each predecessor. Some messages are taking from the ELs and sent to the successor nodes according to the selection rules indicated in the code. It is activated during the scanning process and only if any of the EL has messages.

- **E (Exit)** Destroys messages. The message in the EL is processed. The code in the node is executed and the message is destroyed. The node is activated during the scanning of the network if the EL is not empty.

- **C (Continuous)** Solves systems of ordinary differential equations of first order. It accepts instructions of the form `<variable>' := <expression>`. The system considers a succession of such instructions as a system of differential equations. When the node is activated, the solving process starts. It may be interrupted and continued when the node is deactivated or activated by special instructions. During the execution the node schedules its own activation at each integration step. This activations does not cause scanning of the network. C type nodes have neither EL nor IL.

- **A (Autonomous)** Executes events at scheduled times. It is only activated by an event that refers to it. It is not activated again during the scanning of the network. It has neither EL nor IL. It can activate itself and other nodes, change variables, and send messages.

- **X (General)** Used to general processes programmed by the user. These nodes may be designed by any other letter different of G, L, I, D, E, R, C, A. The node is activated during scanning processes (this is the main difference with the A type). It has not EL or IL unless this is explicitly required by the user. The management of these lists must be programmed by the user with the help of GLIDER instructions and procedures.

1.3 Nodes with indexes

The nodes (Except those of C or D type) can have a subscript or index that can take values 1, 2, ..., until a maximum value that must be declared by the user in the node heading. Indexed nodes are equivalent to a set of nodes of the same type equal to the declared maximum. These nodes share a common code. They are used to represent sets of similar subsystems.

When activated during the scan of the network, they are activated sequentially. When activated by an event that refers to I and A type nodes, only the node with the index of the event is activated. An index variable INO takes the value of the index of the currently executed node. The user has access to that index in order to be able to control differences when processing the common code. The multiplicity or dimension of the node is declared as arrays in PASCAL, i.e., in the form [1..N].

Example:

```
Window (R) [1..5] Exit1, Exit2 ::  <code>
```

This line declares that the node named Window is of R type, its successor nodes are Exit1 and Exit2. It consists in 5 nodes that share the `<code>`.
The nodes without index are called **simple**, those with index are called **multiple nodes**. The number of nodes of a multiple node is called its **dimension** or **multiplicity**.
The nodes of a multiple R type node can be of different capacity.

1.4 Example 1: Simple Serving System

The problem is to compute the queues and waiting times of people that are to be served one by one in a window. They enter the system at random and the serving times are also random. After being served they left the system. The GLIDER program may be as follows:

```
Simulation of a simple serving system.

Patrons enter the system to be served at a window.
The times between arrivals are taken at random from an exponential
distribution with mean Tba. The patrons are served or form a queue
in front of the window. The serving time is taken from a gaussian
distribution with mean: MeanWait and standard deviation: StaDev.
Once they are served they go to Exit to leave the system.
The simulation is to go until time 1000. Statistics of nodes are
required. For clarity reserved words are written in uppercase letters.

NETWORK
  Entry (I) Window :: IT := EXPO(Tba);
  Window (R) Exit  :: STAY := GAUSS(MeanWait, StaDev);
  Exit ::

INIT TSIM := 1000; ACT(Entry, 0); Tba := 4; MeanWait := 3.8;
         StaDev := 0.8;
DECL VAR Tba, MeanWait, StaDev: REAL;
           STATISTICS Entry, Window, Exit;
END.
```

The program has four sections divided by three separators:
NETWORK, INIT and DECL.

- The initial section is a text that can be used to put the title of the program, author, date, etc. An explanation may be included that can be extended to a full documentation. The only restriction is that no line of the text can start with the word NETWORK because this is the separator to the next section.
- NETWORK section must start with this word as the first in the first line of the section (comments between parenthesis `{ }` or `(* *)` may however be included at any place in which a blank is allowed). This section contains the nodes of the program. The **name** of each node must be at the beginning of the line (except blanks and comments). The heading of a node also includes the **type** of the node (a letter between `()`) and the **successors** nodes. The characters :: terminate the heading, after them the code may follows. The code is written in free format from column 1 to 75. The word INIT at the beginning of a line indicates the end of the NETWORK section. See Chapters 4 and 5 for more details. The **structured instructions** of GLIDER (like `STAY := <expression>`) are described in Chapter 6, the **procedures** (like `ACT`) in Chapter 7, the **functions** (like `EXPO` or `GAUSS`), in Chapter 8. For each user programmed node, the compiler will generate a **procedure** that contains the user code and system provided code to perform the other operations of the node implied by its type. This procedure is executed when the node is activated.

- INIT section contains instructions to give initial values to the variables and parameters for the simulation run. It may contain the specification of the total simulation time by assignation of the value to the system variable TSIM. There are other ways to finish the simulation.

  This section must contain at least an ACT procedure to activate one node of the network to start the simulation. In the example the node `Entry`, indicated in the first argument, will be activated at a time equal to the time of the initialization (that is 0 by default) plus the value of the second argument (also 0 in this example). Thus, it is activated at the beginning of the simulation.

  Values are assigned to the parameters `Tba`, `MeanWait` and `StaDev`. Complete algorithms may be included in this section. In this section may also be assigned: capacities to R nodes, values to user defined functions, frequency table parameters and database tables. See Chapter 3 for details.

- DECL section contains the declaration of some elements of the program introduced by the user. These names must be different of the reserved or system words (see Chapter 9).

  The subsection VAR contains the declarations of the variables. The subsection STATISTICS may contain node names and variable names from which statistics are wanted. In this case only node statistics are requested.

  Other declarations of data corresponding to the base language may be done here (types, constants, records, arrays, sets, files). The procedures and functions declared and programmed by the user must be also included in this section.

  Function defined by pairs of values (a special feature of the GLIDER language), the data base tables, frequency tables and the structure of the messages must be also declared here. See Chapter 2 for details.

Lexicographic remark:

- The text of the model must be written on columns 1 to 75.
- The translator is case insensitive in the DOS version. Window may be written WINDOW.
- Comments may be included within parenthesis `{ }` or `(* *)`
- The names of nodes in the node heading, as Entry and sections as NETWORK, must be the first names in its lines, but can be preceded by comments, otherwise the format is free.

1.4.1 Description of the process

The execution of the program starts with the INIT section. The total simulation time is assigned to the GLIDER variable TSIM. The GLIDER variable TIME, that keeps the value of the time during the simulation, is initialized to 0. The user could put explicitly other assignation (even a negative number).

The GLIDER procedure ACT schedules an activation for the actual time plus the second argument (here also 0, but it may be a real expression) so the first activation will happen at $TIME = 0$ in this example. The other instructions of the INIT section assign values to the parameters. .

Then the execution of the NETWORK section starts. The components of the simulated system: entrance, window and exit are represented by the nodes `Entry`, `Window`, and `Exit` in the NETWORK. The patrons are represented by the messages that each node may send to its successor.

The node `Entry` is of I type. Its successor (node to which messages can be sent) is the node `Window`. In this case we could omit the explicit indication of successor, the GLIDER system takes the following node as successor by default. An I type node (or more exactly the procedure into which the compiler translates it), when activated, generates a message, and sends it to the end of the EL of the successor node. This is done automatically by the GLIDER system. In this node **a next arrival event** (i.e., the next activation of the node Entry) may also be scheduled. This is done by assigning a value to the system variable IT (Interval Time). This value is interpreted as the time that has to pass until the next arrival. In this case, this interval is the value computed by the function EXPO, that generates a pseudorandom value taken from an exponential distribution (see Chapter 8 for further explanations). So the GLIDER schedules a new activation of `Entry`. This activation will occur at a time that is equal to the actual value of TIME plus the value assigned to IT. The user can also control the sending of the message to other nodes by means of a SENDTO instruction. Values to the fields of the message can also be assigned in the code of this node.

The node `Window` simulates a resource, in this case the window where the patrons are served. The capacity is 1 by default (no other capacity was assigned) so that only one message can enter the IL. The messages enter the EL of the node. This EL represents the queue of patrons at the window. The IL represents the patron being served. When the node `Window` is activated **during the scanning process of the network** (and this will happen because the node Entry started such a scanning), the procedure of the node type R, if the IL is empty, searches the EL. If it is empty nothing is further done. If it has messages, the procedure takes the first message to pass it to the IL. If the IL was occupied (it contains a message representing a patron using the resource) the incoming message remains in the EL. When a message is introduced in the IL, that means that a new service starts. Then the GLIDER system schedules an **event of end of service**. That is, a future activation of the node `Window` at a time equal to the actual value of TIME plus the value assigned to the variable STAY. In this case the value is taken at random from a gaussian distribution. That time is also kept in a field of the message as its **exit time**. When that future activation takes place, the node `Window` is again activated, but now **by the event that refers to it**. In this case, the procedure searches the IL for a message having an exit time equal to the time of the event (that is, the actual time). In this simple case only one message may be in the IL. The message is then extracted and sent to the EL of the successor node `Exit`. As this message movement triggers a new scanning of the network, the node Window is now activated by the scanning process with is IL empty. So it searches again to see if there are in the EL other message to be passed to the IL. If it is so, a storage process as the explained above will happen. Otherwise, the `Window` remains idle.

The node `Exit` is of E type (when the type is not declared, the compiler assumes that the type is indicated by the first letter of its name). When it is activated (as a consequence of the scanning of the network started by the extraction event in the node `Window`), the procedure of the node `Exit` removes the message of its EL and destroys it, restoring the freed memory to a free memory pool. During all these operations statistics are recorded of the lists, and the use of the resources.

1.4.2 Trace of the operations

The user can ask, from the Run Interactive Menu, for an execution of the events one by one and a display of some of the effects of the executions. This is useful for a detailed understanding of the process and for debugging purposes.

A partial trace of the simulation of this program is shown below. To follow the trace the following remarks must be regarded:

- Before each event, the status of the Future Event List is displayed.
- The next event is taken from the first element of this list.
- At the beginning notice that the only event is the arrival introduced in the INIT section. The index is 1 (the node is a simple one) an the event parameter is 0.
- The line: `*****************>>>>>>>>>>>>>` indicates the beginning of an event. Its number (order of the node in the NETWORK) and the index IEV is indicated. Then the name of the node firstly activated is shown.
- The procedures executed in the activation are indicated.

In the example the trace displays the content of the FEL. It only has the event scheduled by the ACT instruction in the INIT section. This event is an activation of the node Entry at TIME = 0. The index of the event is 1 and the event parameter is 0.

The EVENT 1, that activates the node Entry at time 0, starts adding other event number 1 (next activation) to the FEL for the future time 0.353 (this number comes from the EXPO function).

A message numbered # 1 that was generated at Entry is added to the EL of the node Window, that previously had length LL equal to 0. Notice that the messages generated in a node contains a sequential number and the name of that node.

Following the scanning of the network the node Window is activated.

The message 1 (generated at Entry) is examined, then extracted from the EL.

An event of type 2 (activation of node Window) is put into the FEL to schedule the end of service at time 4.409.

The extracted message is added to the IL of the node Window.

Three events of type 1 (arrivals) happen at times 0.353, 0.888 and 4.108. These are intercalated in the FEL before the end of service event, scheduled for TIME 4.409.

The generated messages are added to EL_Window and remain there because the IL is occupied.

The next event is of type 2 (activation of Window) at TIME 4.409 scheduled at TIME 0. This event examines the IL of Window seeking for a message with EXIT TIME 4.409. The message is extracted and added to the EL of the Exit node.

Then the EL of Window is examined. The next message (number 2) in the queue (EL_Window) is extracted, its exit time is scheduled at time 7.784 and the message is added to the IL of Window. The following message (number 3) is also examined but as the IL is occupied, the message remains in the EL.

In the same event the node Exit is activated, the message number 2 extracted from its EL and deleted.

The network (excluded the Entry node that is of I type) is scanned again and, as no messages are moved, the next event in the FEL is processed.

The reader can continue the examination of this simulation. In this simple example only two events are at most in the FEL (next arrival and next end of service). In more complex models the FEL can be longer.

The examination of traces is very useful to understand the GLIDER system works. It is also very important for debugging.

```
Future Event List at Time   0.000
NODE        ENTRY     T    0.000  IND     1  EVP       0

EVENT    1  IEV   1  ******************>>>>>>>>>>>>>>          TIME     0.000
NODE ENTRY[1]
ADD TO FEL EV 1 IEV 1 TIME      0.353
ADD MESS.#    1 GEN        ENTRY TO      EL_WINDOW 1  LL     0 TIME     0.000
NODE WINDOW[1]
EXA.MESS.#    1 GEN        ENTRY
EXT.MESS.#    1 GEN        ENTRY FROM    EL_WINDOW 1  LL     1 TIME     0.000
ADD TO FEL EV 2 IEV 1 TIME      4.409
ADD MESS.#    1 GEN        ENTRY TO      IL_WINDOW 1  LL     0 TIME     0.000

Future Event List at Time   0.000
NODE        ENTRY     T    0.353  IND     1  EVP       0
NODE       WINDOW     T    4.409  IND     1  EVP       0

EVENT    1  IEV   1  ******************>>>>>>>>>>>>>>          TIME     0.353
NODE ENTRY[1]
ADD TO FEL EV 1 IEV 1 TIME      0.888
ADD MESS.#    2 GEN        ENTRY TO      EL_WINDOW 1  LL     0 TIME     0.353
NODE WINDOW[1]
EXA.MESS.#    2 GEN        ENTRY
NODE WINDOW[1]
EXA.MESS.#    2 GEN        ENTRY

Future Event List at Time   0.353
NODE        ENTRY     T    0.888  IND     1  EVP       0
NODE       WINDOW     T    4.409  IND     1  EVP       0
```

```
EVENT    1  IEV   1  ******************>>>>>>>>>>>>>>>>         TIME     0.888
NODE ENTRY[1]
ADD TO FEL EV 1 IEV 1 TIME      4.108
ADD MESS.#    3 GEN        ENTRY TO      EL_WINDOW 1  LL     1 TIME     0.888
NODE WINDOW[1]
EXA.MESS.#    2 GEN        ENTRY
NODE WINDOW[1]
EXA.MESS.#    2 GEN        ENTRY

Future Event List at Time   0.888
NODE        ENTRY     T    4.108  IND     1  EVP       0
NODE       WINDOW     T    4.409  IND     1  EVP       0

EVENT    1  IEV   1  ******************>>>>>>>>>>>>>>>>         TIME     4.108
NODE ENTRY[1]
ADD TO FEL EV 1 IEV 1 TIME      7.614
ADD MESS.#    4 GEN        ENTRY TO      EL_WINDOW 1  LL     2 TIME     4.108
NODE WINDOW[1]
EXA.MESS.#    2 GEN        ENTRY
NODE WINDOW[1]
EXA.MESS.#    2 GEN        ENTRY

Future Event List at Time   4.108
NODE       WINDOW     T    4.409  IND     1  EVP       0
NODE        ENTRY     T    7.614  IND     1  EVP       0

EVENT    2  IEV   1  ******************>>>>>>>>>>>>>>>>         TIME     4.409
NODE WINDOW[1]
EXAM.LIST IL_WINDOW 1  SEEK EXIT TIME     4.409
EXT.MESS.#    1 GEN        ENTRY FROM    IL_WINDOW 1  LL     1 TIME     4.409
ADD MESS.#    1 GEN        ENTRY TO        EL_EXIT 1  LL     0 TIME     4.409
EXA.MESS.#    2 GEN        ENTRY
EXT.MESS.#    2 GEN        ENTRY FROM    EL_WINDOW 1  LL     3 TIME     4.409
ADD TO FEL EV 2 IEV 1 TIME      7.784
ADD MESS.#    2 GEN        ENTRY TO      IL_WINDOW 1  LL     0 TIME     4.409
EXA.MESS.#    3 GEN        ENTRY
NODE EXIT[1]
EXT.MESS.#    1 GEN        ENTRY FROM      EL_EXIT 1  LL     1 TIME     4.409
DELETED MESS#     1  GENERATED IN        ENTRY AT     0.000
NODE WINDOW[1]
EXA.MESS.#    3 GEN        ENTRY

Future Event List at Time   4.409
NODE        ENTRY     T    7.614  IND     1  EVP       0
NODE       WINDOW     T    7.784  IND     1  EVP       0

EVENT    1  IEV   1  ******************>>>>>>>>>>>>>>>>         TIME     7.614
NODE ENTRY[1]
ADD TO FEL EV 1 IEV 1 TIME      7.917
ADD MESS.#    5 GEN        ENTRY TO      EL_WINDOW 1  LL     2 TIME     7.614
NODE WINDOW[1]
EXA.MESS.#    3 GEN        ENTRY
NODE WINDOW[1]
EXA.MESS.#    3 GEN        ENTRY

Future Event List at Time   7.614
NODE       WINDOW     T    7.784  IND     1  EVP       0
NODE        ENTRY     T    7.917  IND     1  EVP       0
```

```
EVENT    2  IEV   1  *****************>>>>>>>>>>>>>          TIME     7.784
NODE WINDOW[1]
EXAM.LIST IL_WINDOW 1  SEEK EXIT TIME     7.784
EXT.MESS.#    2 GEN        ENTRY FROM    IL_WINDOW 1  LL     1 TIME     7.784
ADD MESS.#    2 GEN        ENTRY TO        EL_EXIT 1  LL     0 TIME     7.784
EXA.MESS.#    3 GEN        ENTRY
EXT.MESS.#    3 GEN        ENTRY FROM    EL_WINDOW 1  LL     3 TIME     7.784
ADD TO FEL EV 2 IEV 1 TIME     11.858
ADD MESS.#    3 GEN        ENTRY TO      IL_WINDOW 1  LL     0 TIME     7.784
EXA.MESS.#    4 GEN        ENTRY
NODE EXIT[1]
EXT.MESS.#    2 GEN        ENTRY FROM      EL_EXIT 1  LL     1 TIME     7.784
DELETED MESS#     2  GENERATED IN        ENTRY AT     0.353
NODE WINDOW[1]
EXA.MESS.#    4 GEN        ENTRY

Future Event List at Time   7.784
NODE        ENTRY     T    7.917  IND     1  EVP       0
NODE       WINDOW     T   11.858  IND     1  EVP       0

EVENT    1  IEV   1  *****************>>>>>>>>>>>>>          TIME     7.917
NODE ENTRY[1]
ADD TO FEL EV 1 IEV 1 TIME     11.890
ADD MESS.#    6 GEN        ENTRY TO      EL_WINDOW 1  LL     2 TIME     7.917
NODE WINDOW[1]
EXA.MESS.#    4 GEN        ENTRY
NODE WINDOW[1]
EXA.MESS.#    4 GEN        ENTRY

Future Event List at Time   7.917
NODE       WINDOW     T   11.858  IND     1  EVP       0
NODE        ENTRY     T   11.890  IND     1  EVP       0

EVENT    2  IEV   1  *****************>>>>>>>>>>>>>          TIME    11.858
NODE WINDOW[1]
EXAM.LIST IL_WINDOW 1  SEEK EXIT TIME    11.858
EXT.MESS.#    3 GEN        ENTRY FROM    IL_WINDOW 1  LL     1 TIME    11.858
ADD MESS.#    3 GEN        ENTRY TO        EL_EXIT 1  LL     0 TIME    11.858
EXA.MESS.#    4 GEN        ENTRY
EXT.MESS.#    4 GEN        ENTRY FROM    EL_WINDOW 1  LL     3 TIME    11.858
ADD TO FEL EV 2 IEV 1 TIME     16.019
ADD MESS.#    4 GEN        ENTRY TO      IL_WINDOW 1  LL     0 TIME    11.858
EXA.MESS.#    5 GEN        ENTRY
NODE EXIT[1]
EXT.MESS.#    3 GEN        ENTRY FROM      EL_EXIT 1  LL     1 TIME    11.858
DELETED MESS#     3  GENERATED IN        ENTRY AT     0.888
NODE WINDOW[1]
EXA.MESS.#    5 GEN        ENTRY

Future Event List at Time  11.858
NODE        ENTRY     T   11.890  IND     1  EVP       0
NODE       WINDOW     T   16.019  IND     1  EVP       0

EVENT    1  IEV   1  *****************>>>>>>>>>>>>>          TIME    11.890
NODE ENTRY[1]
```

1.4.3 Standard Statistics

When required by the declaration STATISTICS, the GLIDER produces an output with statistics of nodes and variables. This statistics are always displayed at the end of the simulation, but they can be shown at any time during the run by program indication or by an operator interruption that calls the Run Interactive Menu. Statistics for one run of the simulation example follows:

```
Basic Experiment
Time    1000.00 Time Stat.   1000.00  Replication    1  22/ 2/1996 11h 53m 27s
Elapsed time  0h  0m  3.2s
Nod/Ind Ant/Li #Ent Lgth   Max    Mean    Dev   MaxSt   MeanSt    Dev    T.Free
ENTRY
   1 # Gen.     287
WINDOW
   1 EL         287   23    31 19.6351 7.42638 121.549 64.6315 33.9571      0.4
     IL         264    1     1 1.00000     0.0 5.89579 3.78773 0.78256      0.0
EXIT
   1 EL         263    0     1     0.0     0.0     0.0     0.0     0.0   1000.0
  Time in System: Mean  73.7248   Dev.  29.8822   Max.  124.650   Min.  4.40897


Var/Ind                 Mean(t) Dev.(t)    Max.    Min. Mean(v) Dev.(v)  Actual
    U_WINDOW  1         1.00000     0.0 1.00000     0.0 0.50095 0.50000 1.00000
```

The title of the experiment is by default “Basic Experiment”. The user can put other titles by program (see instruction TITLE, 6.27) or from the Menu.

The total simulation time and the last time in which the Statistics were cleared (see procedure CLRSTAT, 7.4) are also shown.

If replications of the experiment are asked for, the number of the replication is given.

The date and time of the beginning of the execution and the computing time spent in the simulation are also recorded.

For the I type nodes the number of generated messages is shown.

For each EL and IL the number of entries, the actual length and the maximum length are displayed. The statistics show the mean length and its deviation, the maximum and mean waiting time of the messages in the list, its deviation and the time that the list remained empty.

For the E type nodes the mean time in the system of the messages destroyed at the node and its deviation are recorded.

In this example only the used time of the resources are displayed, because no other variables in the STATISTICS were requested. The statistics given are the mean and deviation of the variables as a function of time, the maximum and minimum values, the mean and deviation of the values and the actual (final) value.

1.5 Example 2. Port with Three Types of Piers

This is the simulation of a port that has three types of piers to receive different types of ships:

1. ships with general charge,
2. ships that bring charge to be discharged in bulk to silos, and
3. ships that come to be repaired.

The port has an entry channel in which only a ship at a time is allowed to pass. The same channel is used to enter to the piers and to exit, once the operation in the port is finished.

```
Model of a Port.                                (C. Domingo 2/23/96)
Ships arrive at a port with interval times with exponential
distribution with mean Tbarr.
There are 3 types of ships. The type is selected from an empirical
random distribution by means of a function FTyp.
According to the type they go to the piers 1, 2 or 3.
The ships are queued before the entry channel. A ship is allowed to
pass only if the channel is free and the ship has a place in the pier
of its type.
The time spent in passing the channel is a fixed value TChannel.
In the pier the ship remains a time taken from a Gamma distribution
The mean TPier depends on the type. The deviation is 10% of the mean.
Each ship uses only one position in the pier.
To the served ships the type 4 is assigned and they are sent to the
channel to exit the system.
Make a frequency table of the mean time in the system.
Experiment with different serving times in the piers.

NETWORK

 Entrance (I) :: IT := EXPO(Tbarr); ShiTyp := FTyp;

 Control(G)  :: IF (F_Pier[ShiTyp] > 0) and (F_Channel > 0)
                      THEN SENDTO(Channel);

 Channel (R) Pier[ShiTyp], Departure ::
            RELEASE   IF ShiTyp = 4 THEN SENDTO(Exit)
                                ELSE SENDTO(Pier[ShiTyp]);
            STAY := TChannel;

 Pier (R) [1..3] Channel ::
            STAY := GAMMA(TPier[ShiTyp], 0.1 * TPier[ShiTyp]);
            USE := 1; ShiTyp := 4;

 Departure (E) :: TinSys := TIME - GT; TAB(TinSys, TabTSys);

 INIT
  TSIM := 1200; ACT(Entrance, 0);
  Tbarr := 4;                       (*Mean time between arrivals       *)
  TChannel := 0.2;                  (*Time to pass the channel         *)
  FTyp := 1, 50 / 2, 40 / 3, 10;    (*Values of frequency of types     *)
  ASSI Pier[1..3] := (4, 2, 1);     (*Capacities of the piers          *)
  ASSI TPier[1..3] := (20, 12, 28); (*Mean time spent in the piers     *)
  TabTSys := 0, 15, 5;              (*Parameters of the frequency table*)
  TITLE := 'Port. Basic experiment';
  INTI TPier: 7: 2: Time in the Pier ;

 DECL
  VAR Tbarr, TinSys, TChannel: REAL;
      TPier: ARRAY[1..3] OF REAL;
      MESSAGES Entrance(ShiTyp: INTEGER);  (*structure of the messages   *)
      TABLES TinSys: TabTSys;        (*table of frequencies for TinSys   *)
      GFUNCTIONS  FTyp FREQ (INTEGER): REAL: 3;  (*frequency function    *)
      STATISTICS ALLNODES;
```

```
END.
```

1.5.1 Remarks about the program

The ships are represented by messages generated by the I type node `Entrance`. The messages have a field `ShiTyp`. To declare, that the messages must have this field, a declaration is issued in the subsection MESSAGES of the DECL section with reference to the generating node.

The `Entrance` node assign value (1, 2 or 3) to the `ShiTyp` parameter of the generated message. The value is given by the function `FTyp` defined in the GFUNCTIONS declaration. The values of the frequencies are given in the INIT (value 1 frequency 50, value 2 frequency 40, value 3 frequency 10). The function `FTyp` produces a random number 1, 2, or 3 according to those frequencies. It is assigned to the field `ShiTyp` of the message. The messages are sent to the EL of the node `Control`.

In the `Control` node, that represents the control of entry to the port, the GLIDER generated variable `F_Pier[ShiTyp]` holds at each moment the value of the free capacity of the node `Pier[ShiTyp]`. The maximum capacities are assigned in the INIT section: 4 for the `Pier` 1, 3 for the `Pier` 2, 1 for the `Pier 3`. `F_Channel` has the same meaning for the resource `Channel`. In this case it is 1 if the channel is free and 0 if it is engaged. When the `Control` node, that is a type G node, scans the EL of messages and executes the code for each message. Only if it find a message for which is true the condition of the if (there is free capacity in the `Pier` of its type and the `Channel` is free), the SENDTO instruction is executed. This extracts the message from the EL and sends it to the EL of the node `Channel`.

The channel is represented by the resource type node `Channel`. When activated by the scanning of the network, this node takes the message in its EL and passes it to the IL (this will be empty because the Control only allows to pass a message if `F_Channel = 1`, that means that no message is using the resource. The exit from the channel is scheduled for a later time given by the user defined variable `TChannel`. When the message abandons the resource, it is not automatically managed, because there is a RELEASE instruction that takes care of the outgoing message. In the instruction associated to the RELEASE the message is disposed according to its type. If it is 1, 2, or 3 it is send respectively to the Pier[1], Pier[2], or Pier[3]. If it has type 4 (that is the type that will be assigned to the ships abandoning the piers) it is sent to the `Departure` node.

The node `Pier` is a multiple node of dimension 3. Each node represents a different pier; each may receive the corresponding type of ship. The capacity of each node is assigned in the INIT section as said above. When the capacity is different of 1, the user has to define the quantity of resource used by each message. This is done by the instruction USE := 1, i.e., it is assumed that all the ships use the same space, equal to 1 unit of the resource. This would be different with ship of very different size.

The operation time in the pier is taken from a Gamma distribution; for different types of ships the mean value used is different. The values of the means are assigned in the INIT section. Notice that the type of the received ship is put to 4. After the operation it is sent to `Channel` and after passing the channel, the ship of type 4 is sent to Departure.

The departure node is of E type. It takes each message of its EL, executes the code and destroys the message. The code is the computation of the time spent in the system. The GT is a GLIDER defined field of the message that keeps its generation time. As TIME is the value of the actual time, TIME - GT is the total time that the message remained in the system. The procedure TAB put this value in the frequency table `TabTSys`. This table was declared in the DECL section. The parameters of the table are assigned in INIT. In this case the initial value of the table is 0, and there will be 15 intervals of length 5. The table and a histogram is shown when requested by the user.

The instruction INTI in the INIT section allows for interactive change of the values of `TPier` for different experiments.

1.5.2 Standard Statistics

```
Port. Basic experiment
Time     1200.00 Time Stat.   1200.00  Replication    1  23/ 2/1996 21h 20m  7s
```

```
Elapsed time  0h  0m 27.58s
Nod/Ind Ant/Li #Ent Lgth   Max    Mean    Dev   MaxSt   MeanSt    Dev    T.Free
ENTRANCE
   1 # Gen.     291
CONTROL
   1 EL         291    0     9 0.93442 1.26150 101.587 3.80141 11.4577    628.2
CHANNEL
   1 EL         579    0     6 0.03409 0.27533 1.00466 0.07064 0.16318   1176.0
     IL         579    0     1 0.09650 0.29528 0.20000 0.19965 0.01175   1084.2
PIER
   1 EL         155    0     3 0.11652 0.44114 18.2005 0.90207 3.07138   1107.9
     IL         155    2     4 2.53706 1.30822 24.4311 19.3037 3.60884    109.2
   2 EL         100    0     1 .632- 2 0.07922 3.22191 0.07579 0.40935   1192.4
     IL         100    0     2 0.99144 0.79357 15.7321 11.7546 2.26446    383.0
   3 EL          36    0     3 0.53564 0.87107 89.3944 17.8548 25.6300    807.0
     IL          36    1     1 0.83015 0.37550 33.6624 27.9559     0.0    203.8
DEPARTURE
   1 EL         288    0     1     0.0     0.0     0.0     0.0     0.0   1200.0
  Time in System: Mean  25.0310   Dev.  21.6358   Max.  141.692   Min.  9.42319


Var/Ind                  Mean(t) Dev.(t)    Max.    Min. Mean(v) Dev.(v)  Actual
   U_CHANNEL  1          0.09650 0.29528 1.00000     0.0 0.50000 0.50000     0.0
      U_PIER  1          2.53706 1.30822 4.00000     0.0 2.69481 1.07103 2.00000
      U_PIER  2          0.99144 0.79357 2.00000     0.0 1.17000 0.68637     0.0
      U_PIER  3          0.83015 0.37550 1.00000     0.0 0.50704 0.49995 1.00000


Table TABTSYS of variable TINSYS

 Interval         Freq. Rel.Fr.

      ..<  0.00     0   0.000
  0.00..<  5.00     0   0.000
  5.00..< 10.00     2   0.007*
 10.00..< 15.00    82   0.285*********************************************
 15.00..< 20.00    65   0.226***********************************
 20.00..< 25.00    70   0.243**************************************
 25.00..< 30.00    23   0.080************
 30.00..< 35.00    16   0.056********
 35.00..< 40.00     5   0.017**
 40.00..< 45.00     3   0.010*
 45.00..< 50.00     2   0.007*
 50.00..< 55.00     2   0.007*
 55.00..< 60.00     1   0.003
 60.00..< 65.00     1   0.003
 65.00..< 70.00     0   0.000
 70.00..< 75.00     1   0.003
       >= 75.00    15   0.052********
Total             288   1.000
```

1.6 Compilation Details

Although compiling details are transparent to users some information may be useful.

The source program is written in a file. It is read and processed by the GLIDER compiler that translates it to several units of Pascal code.

The program code of the nodes are translated into procedures. The messages are translated into records that may have different structures, according to the fields declared in the MESSAGES declarations. The actual items that are in the ELs and ILs are fixed structure records (for all kinds of messages) called **indicators** of messages. They have pointers to the true messages, and some auxiliary fields: number of the message, original node name and pointer to the following indicator in the list (or NIL if it is the last).

The program so produced by the GLIDER compiler may be then compiled and executed by a Pascal system.

Chapter 2

DECLARATIONS

A GLIDER program uses variables of different types. GLIDER includes definitions for certain types, variables, constants, functions and procedures required for general processing. Others are also system defined to meet special requirements of particular models.

Besides, users may define types, variables, constants, functions and procedures needed to program their models. Other instances of language elements may also be defined: messages, functions given by pairs of values, frequency tables, data base tables, and needed statistics.

All user's variables must be explicitly declared, as mandatory in Pascal.

The names assigned to types, variables, constants, functions and procedures are called **identifiers**. They are strings of at most twelve characters, including letters, numbers and the sign _ . The first character in the identifier must be a letter. GLIDER is case insensitive [1], so Water_Temp and WATER_temp are the same variable.

A declared identifier must not duplicate any system or previously user defined identifier. See Chapter 9 for a list of system defined identifiers.

The declaration section begins with the separator **DECL** as the first word of a line. Identifiers are declared in the following subsections of the DECL section, according to their element class:

1. **TYPE**	types
2. **CONST**	constants
3. **VAR**	variables
4. **NODES**	nodes
5. **MESSAGES**	messages
6. **GFUNCTIONS**	user's functions defined by values
7. **TABLES**	frequency tables
8. **DBTABLES**	tables from a data based system
9. **PROCEDURES**	user's built procedures and functions
10. **STATISTICS**	required from nodes and variables

A declaration consists of the name of the element class (one of the above) followed by identifier declarations, according to the syntax described in the following sections.

2.1 TYPE

A type declaration specifies to the compiler that a given **identifier** or name will designate a type of data that may be referred to in the program, i.e., an abstract set of data values.

[1] Note that the GLIDER UNIX version *is* case sensitive, see GLIDER UNIX Version Manual.

There are types already defined by the GLIDER or Pascal system (REAL, INTEGER, BOOLEAN, CONT, POINTER etc.).

Each variable in a program must be declared indicating its type. With this information the compiler reserve to that variable enough memory space to hold its values, that must belong to the type. This information also could be used by the compiler to validate that only values belonging to the type are assigned at run time.

Type identifiers may be referred to in other type declarations in order to define new types. A trivial case is to define an 'alias' to an existing type, as in the following example: `Decimal = REAL`

The true usefulness of type declaration is not this mere 'alias' to predefined types, but when new structured types have to be defined.

New types may be defined for convenience of clarity and maintenance, otherwise type specifications for variables may be stated directly in variable declarations.

In the following example the new type Arr5 is declared:

```
TYPE Arr5 = ARRAY[1..5] OF REAL;
```

With this new type the programmer can avoid to declare the variables A, B, z, X in this way:

```
VAR  A, B: ARRAY[1..5] OF REAL;.......;
......................
VAR  z: ARRAY[1..5] OF REAL; ... X: ARRAY[1..5] OF REAL;
```

Instead, the programmer only needs to write:

```
VAR  A, B: Arr5;.......;
VAR  z: Arr5;....; X: Arr5;...
```

Note the use of the = sign for new *type declarations* whereas the : sign is used for *type specification* in variable declarations.

New types declarations are mandatory in Pascal when the user needs to pass a structured variable in the parameter list of a function or procedure, as Pascal syntax only accepts type identifiers in parameter list declarations.

The reader may consult Pascal and TurboPascal bibliography for a complete description on types. Restrictions or changes in GLIDER to Pascal syntax and use will be stated when appropriate.

GLIDER includes the following predefined types and type constructors for new type declarations or variable type specifications:

2.1.1 Simple types

The following table shows the set of simple types in GLIDER. Besides the simple Pascal types, GLIDER adds the CONT, RET and FREQ simple types.

REAL	Real numbers
INTEGER	Integer numbers
LONGINT	Long integer numbers
BOOLEAN	Logic values (TRUE or FALSE)
CHAR	ASCII character set
WORD	Non negative integers
BYTE	Non negative integers in the range 0..255
STR80	ASCII character set
TEXT	ASCII character set
CONT	Continuous (for variables with derivatives in differential equations). Must be used in VAR only for a list of contiguous variables. See 4.7.

RET	Retard. To be used in the RETARD procedure. Must be used in VAR only. See 7.17.
FREQ	For component type of arrays to be used as multivariate frequency tables.

The user may also declare as new simple types:

- **Enumerated types**: ordered sets of identifiers.

  The syntax is:

  `<type identifier> = (<list of identifiers>);`

  Example:
  `Size = (Small, Medium, Large, Extralarge);`
  `Medium` is then a constant of type Size. Order relations hold, for instance, `Small < Large` is true, `ord(Large)` has the value 3.

- **Subrange types**: subranges of any ordered type (INTEGER, LONGINT, CHAR, WORD, BYTE, and user's defined enumerated types) The syntax is:
  `<type identifier> = <initial value> .. <final value>`

  Examples:

```
VAR CapLetter: 'A' .. 'Z';
    LargeSizes: (Large .. ExtraLarge);
TYPE    Digits = 0 .. 9;
```

2.1.2 Pointer types

- **POINTER**: Is a general pointer type. Variables of this type may be used as temporal storage for pointer values, i.e., values of addresses of program elements (variables, functions or procedures) in the space of memory locations at run time.

  Example: `VAR Ppa: POINTER;`

- **Typed pointers**: They are specific pointer types. Variables of specific pointer type may hold values of addresses of program elements of the specified type in the space of memory locations at run time.

  The syntax of the declaration is:
  `<pointer type identifier> = ^<type identifier>`
  It defines a new pointer type to element values of the `identifier` type, which may be a simple or structured type.

  Example:

```
Ptra = ^Train;
Train = Record
       Engine: STRING[6];
       Wagon: ARRAY[1 .. 16] of REAL;
       Follows: Ptra;
      END;
```

  If `PTrain: Ptra` is declared in the `VAR` subsection of `DECL`, then the pointer variable `PTrain` is supposed to point to a record with the shown structure. The variable `PTrain^.Wagon[3]` refers to the third array element of the field `Wagon` of the `Train` type record pointed by `PTrain`.

  These user's declared records are seldom used in GLIDER because messages suffices for almost all the uses and they are internally handled by GLIDER instructions and procedures.

Pointer variable value assignment results from dynamic memory allocation (Pascal procedure NEW) or from direct assignment of memory addresses values, as returned by function `ADDR( )`, for instance. Pointer variables must be carefully used.

2.1.3 Structured types

Structured type constructors allow programmers to define as new types complex information structures with components of the same or different types. They are:

- **ARRAY**: To declare as new type a succession of a fixed quantity of components of the same type. The components of a variable of any ARRAY type occupy contiguous memory locations and can be individually referred to by using one or more indexes.

  The syntax of the type declaration is:

  `<type identifier> = ARRAY[ni1 .. nf1] OF <component type>;`

  for unidimensional arrays, or

  `<type identifier> = ARRAY[ni1 .. nf1, ni2 .. nf2, ...] OF <component type>;`

  for multidimensional arrays. Note that GLIDER restricts the array declaration syntax of index types to subranges, whereas Pascal accepts any ordinal type specification.

  The first index can take values from ni1 to nf1, the second from ni2 to nf2, etc. The values of the indexes may be of type INTEGER, LONGINT, BYTE, WORD or enumerated type.

  Examples:

  ```
  TYPE Vec =ARRAY[-5, 5] OF REAL;
  VAR X,Z:Vec; Mat:ARRAY[1..3, 2..3] OF CHAR;
       ShirtPrice:ARRAY[Small. .XLarge, Cotton.. Silk] OF REAL;
  ```

  `Vec` will be a simple array with index values from -5 to 5.
  `Mat` a two index array, the first from 1 to 3, the second from 2 to 3. The successive elements in the memory will be:
  `Mat[1, 2]; Mat[1, 3]; Mat[2, 2]; Mat[2, 3]; Mat[3, 2]; Mat[3, 3];`

  The rightmost indexes change first.
  `ShirtPrice` has, as indexes, values of an enumerated type that have to be defined previously, for example with declarations:

  ```
  Size = (Small, Medium, Large, XLarge);
  Fabric = (Dacron, Cotton, Flax, Silk);
  ```

- **RECORD**: To declare as new type a succession of components or **fields**, each of them of a declared type. Thus, the field components of a record may be of different types whereas array components are of the same type.

  The syntax of the declaration is:

  `<type identifier> = RECORD  <list of field declarations> END;`

  The field declarations of the list are separated by ;

  Examples:

  ```
  PShip: ^Ship;
  Ship = RECORD
        Draft, Displacement: REAL;
        LoadClass: ARRAY[1..5];
        Pfs: PShip;  (*Pointer to the next ship*)
        END;

  TYPE Mo = (January, February, March, April, May,
           June, July, August, September, October,
           November, December);
  ```

```
     DayType = ( 1 .. 31);
   VAR Birthday: RECORD Month: Mo; Day: DayType;
                          Year: WORD; END
       Bounty: Ship; PConvoy: PShip;
```

The `Ship` record type has a pointer type field for building lists of records of this type. In `VAR` a type record variable `Bounty` is defined. The pointer variable `PConvoy` can be used to construct a list of records of type Ship.

The record variable `Birthday` has fields type `Mo` with the values of the months and `Day` whose values range from 1 to 31. The field `Year` is of the predefined type `WORD`, for this reason, it can take values from 1 to 65535.

Fields of a record type variable are referred to by the variable name followed by the character, followed by the field name.

Example:

`Birthday.Year`

refers to the value of the field Year in the variable Birthday.

- **FILE**: To declare as new type unlimited successions of components of a specified type. The system manages the existence of the components of a file type variable in external I/O devices at run time.

  The syntax is:

  `<file type identifier> = FILE OF <type identifier>`

  `<type identifier>` must not be another file type identifier or any structured type identifier with a file type identifier as component type.

  Example:

  ```
  FileShip: FILE OF Ship;
  ```

  The Pascal predefined type `TEXT` is a file of CHAR type for which Pascal has special variable processing.

- **SET**: To declare as new type the *power* set of an ordinal type, called the *base* type. The power set of the base type is the set of all its subsets, including the **empty** set. The cardinality, i.e., the number of elements of the base type, has to be not greater than 256. This restriction excludes as base type the ordinal types INTEGER, LONGINT and WORD.

  Values of a set type are sets that can be operated with the set operators

  `+, -, *`

  Also are defined the logical binary `IN` operator relating a base type value with a `SET` type value and the set *constructor* using the characters [and] as delimiters. The empty set is the expression: [ ].

  The syntax is:

  `<name> = SET OF <ordered type>;`

  Examples:

  ```
  TYPE COLOR = SET OF (Red, Yellow, Green, Brown);
  VAR  Paint, Hue, Dye: COLOR;
       Digit: SET OF 0 .. 9;
  ```

  `Paint` can take as value any of the 16 subsets of the elements indicated in the declaration of its type (including the empty and the total set).
  `Digit` may take as value any subset of the digits. One assignation of value may be :
  `Digit: = [0, 1, 2];`
  In this case the expression `2 IN Digit` Is true. `4 IN Digit` is false.
  `If Paint := [Red, Yellow] and Hue := [Red, Green, Yellow]`
  then `Dye := Hue * Paint + [Brown]` would have the elements `Red` and `Brown`.

2.1.4 String types

To declare as new type a succession of characters of variable length ranging from 0 to a maximum declared. The syntax is:

<type identifier > = STRING[<positive integer number>];

If [<positive integer number>] is omitted in type specification for variable declaration, the maximum length 255 is assumed, i.e., the word STRING alone is a predefined type identifier.

Examples:

```
TYPE Names: STRING[25]
VAR Roll: ARRAY[1..1000] OF Names; Trademark: STRING[10];
LongLine: STRING;
```

Roll is an array of 1000 elements; each of them is a string that may contain up to 25 characters. LongLine may contain up to 255 characters, Trademark may contain up to 10.

2.1.5 Procedural types

PROCEDURE or FUNCTION types are declared by writing the **prototype** or heading pattern for a class of procedures or functions, i.e., the set of procedures or function that have the same pattern for parameter list declaration and that return the same type of value in the case of functions. The variables declared of these type can take as values names of procedures or function that conform with the prototype, i.e., that belong to the specified class. The syntax is:

<type identifier >= PROCEDURE(<parameter list>) or

<type identifier >= FUNCTION(<parameter list>) : <returned type>

Example:

```
TYPE TyProcess = PROCEDURE(Material, Machine: INTEGER);
VAR Pro1, Pro2: TyProcess;
```

The variables Pro1, Pro2, can take the name of any procedure with any name conforming with the declared prototype. See 2.9 for declarations of procedures and functions in GLIDER.

2.2 CONST

Constants are identifiers to which values are assigned at compiler time or at initial run time, as implemented in the TurboPascal extension of Pascal.

Compiler time declared constants cannot be the target of any assignment operation at run time. The declaration syntax for compiler time declared constants is:

<identifier> = <value>; where

<value> may be a constant expression of any type. The declared constant implicitly takes the same type.

Initial run time declared constants have their type explicitly declared and can be the target of assignment operation at run time.

The declaration syntax for initial run time declared constants is:

<identifier> : <type> = <list of values>;

Examples:

```
CONST
Flow: REAL = 30.7; Quantity: INTEGER = 12;
ch: 'Y';
Load: ARRAY[1..5] OF REAL = (3.8, 4.5, 4.2);
Markov: ARRAY[1..3, 1..3] OF REAL = ((0.2, 0.4, 0.2),
                                     (0.7, 0.2, 0.3),
                                     (0.1, 0.4, 0.5));
```

2.3 VAR

Variables declared in the DECL section are global to the model program, i.e., they may be used anywhere in the program.

Variables declared in a node are local to that node and can only be used in the node's code. Their values are not retained after node processing.

Variables declared in the MESSAGES subsection of the DECL section, are **fields** of each generated message and may have different values in different messages. **Field variables** are variables introduced by the GLIDER system, one for each field name in messages, having the name of the field. They are global, and change their values each time a message is processed, taking the values of the corresponding fields of the processed messages. Note that if messages of different structure have fields of the same name they **must be of the same type**, and GLIDER introduces for them only one field variable.

Example:

```
MESSAGES Cans(Diameter, Height, Weight: REAL);
         Packages(Length, Wide, Height, Weight: REAL);
```

The fields Height and Weight are in both record messages. Field variables having these names are system created so different messages may share same processes on these characteristics.

The declaration of variables reserves memory for them.

After the VAR specification one or more declarations of the following type follow:

`<list of variable names> : <type>;`

The `<list of variable names>` consists of one identifier or many identifiers separated by commas.

`<type>` may be a predefined type identifier, a user defined type identifier or a type specification written with the same syntax given for type declarations.

Examples:

```
DECL
TYPE
     VecType = (Small, Large, Truck);
     Color = (Rer_L, Yellow_L, Green_L);
     VecChar = RECORD Size, Weight: REAL; VType: VecType; END;
VAR  VMax, Length, Slope: REAL;
     Ntotal, j, k: INTEGER; Semaf: Color;
     Vehicle: ARRAY[1 .. 30] of VecChar;
     Connections: ARRAY[1 .. 5, 1 .. 8] of BOOLEAN;
     X, Y: CONT;
     InpFreq: ARRAY[1 .. 4, 1 .. 3] OF FREQ;
     TabVeh: TEXT;
```

2.4 NODES

In this section the user may instruct GLIDER to assign IL and/or EL to general type nodes declared in the NETWORK section. The declarations must have the heading NODES. After this one or more declarations follow. The declaration syntax is:

`<node identifier>( | EL | IL | EL,IL | );`

`<node identifier>` is the name of a general type node. Index must not be included in this declaration. According to the arguments the node will have EL or IL or both.

Examples:

```
NODES Synchr(EL); Proc(EL, IL);
```

2.5 MESSAGES

In this section the user may define structures of messages with user's defined fields, besides the default fields (see 9.1). The declarations must have the heading MESSAGES. After this one or more message declarations follow. The declaration syntax is:

`| <node identifier> | <message identifier> | (<list of field declaration>);`

`<node identifier>` is the name of a I type node. Index must not be included in this declaration. The declaration defines the structure of the messages generated at that Input type node.

`<message identifier>` is an identifier that may be used by a CREATE instruction to generate messages with the declared structure. The CREATE instruction may be used in any node, except I type nodes

`<list of field declaration>` is a list of field declarations with the same syntax that the RECORD structure declaration.

Examples:

```
ShipArr (I) :: ShipType := FTypeFreq; IT := EXPO(Tba[ord(ShipType)];
.....................................
TrainContr (A) ::
      ................................
      CREATE(Train) BEGIN
                      WCars := Round(TotalLoad / Capacity);
                      SENDTO(Yard) END;
.....................................
DECL
TYPE St = (Freighter, Tanker, Barge)
MESSAGES  ShipArr(ShipType: St; Load: ARRAY[1 .. 5] OF REAL; Displ: REAL);
          Train(WCars: INTEGER; Capacity: REAL);
```

In the node `ShipArr`, messages representing different types of simulated ships are generated. The characteristics (fields of the message) are: ship type, five quantities indicating different kinds of loads, and displacement. In the node `TrainContr`, according to conditions computed by a complex code, the decision of sending a message (train) to a yard is taken.

2.6 GFUNCTIONS

GLIDER allows to define functions of one variable given by a set of **pairs of values** (points). The first element of the pair is the value of the argument (it belongs to the domain). The second is the corresponding value of the function. The type of the argument, type of the function and the method of interpolation must be declared. This declaration has the heading GFUNCTIONS followed by declarations with syntax:

```
<function identifier> <interpolation method>
(<argument type>) : <function type> : <maximum number of points>;
```

`<interpolation method>` indicates the method to be used to compute values for intermediate values of arguments not given in the set of pairs. The methods may be:

- `DISC`: intermediate values are not allowed. The function is only defined for the values given in the set of pairs.
- `STAIR`: the function value corresponding to an intermediate value of the argument is the corresponding to the more near value of the arguments in the set of values lesser than the given argument, i.e., the function is supposed piecewise constant and continuous from the right.
- `POLYG`: a linear interpolation is used.
- `SPLINE`: a cubic spline interpolation is used.

- FREQ: is not an interpolation method. The value of the function is taken at random among the possible values of the first element of each pair. The second element is interpreted as proportional to the probability of the corresponding first value.

<argument type> and <function type> may conform with the following table:

	argument	*value*
DISC	enumerate or real	enumerate or real
STAIR	real	real
POLYG	real	real
SPLINE	real	real
FREQ	enumerate or real	real (>= 0)

<maximum number of points> is a positive integer that indicates the maximum number of pairs of values that define the function.
In 3.3 the assignation of the set of pairs of values to a function is described.
Example:

```
GFUNCTIONS Temp POLYG (REAL): REAL: 8;
           pH SPLINE(REAL): REAL: 12;
           Price STAIR (REAL): REAL: 9;
           CarTyp FREQ (CarType): REAL: 8;
           Cost DISC (Size): REAL: 5;
```

2.7 TABLES

In order to display statistical results in the form of frequency tables and histograms the names of the tables must be declared. The header of the declaration is TABLES. It must be followed by one or more declarations of the form:
<variable identifier> : <table identifier>;
<variable identifier> is the name of a simple real variable whose values through time are to be tabulated.
<table identifier> is the identifier assigned to the table.
The tabulation of values must be commanded by the user with the TAB procedure (see 7.22).

2.8 DBTABLES

The GLIDER system can import, update, and export files generated by a data base system (DBASE IV in this implementation). A data base file is a succession of records, each with a set of fields. All the records in a file have the same structure. The records usually describe the characteristics of individuals of a set (people, objects, years, cities, etc.) by means of the values of the fields. These are **simple tables**.
DBASE file in order to be suitable for GLIDER processing must have as its first field in its record structure a primary *key* field named with the same file or table name.
Association tables are used when the values of a relation of two (or more) sets have to be described. For each combination set of two (or more) individuals, one of each set, there is an entry in the table that describes, by one or more fields, the characteristics of the relation. In the actual version only complete tables can be imported, i.e., all the possible combinations of records in the associated tables must appear. In an association table of N simple tables the names of these tables must appear as the first N fields in each record, conforming the primary key of the association.
The user must give in a DBTABLES declaration, the names of the tables and the association indicator (i.e., how many simple tables are associated in the declared table).
The declarations have DBTABLES as a heading, followed by one or more declarations:

`<table name> : <association indicator>`

These declarations must be separated by commas and each or some groups of them may be delimited by ( ). The groups are also separated by commas.

`<table name>` is a name of an existing DBASE file optionally preceded with directory path.

`<association indicator>` is a number from 1 to 5. If omitted, 1 is assumed (simple table).

The declarations that are in a group within ( ) are assumed to have the same structure.

Example:

```
DBTABLES (GroupP1, GroupP2), (Goods), (ConsPatt: 2, CPat1: 2);
```

Declares two simple tables: `GroupP1, GroupP2` of the same structure, a simple table `Goods` of different structure and two tables that are association of two tables and have the same structure. The two first fields of `ConsPatt` must have as name the name of the two simple tables that `ConsPatt` associates. Such simple tables must have been declared before.

All the declared tables must exist as DBASE tables. The compiler uses information from these tables. If they are in other directory the table names must include the path. To handle these tables the GLIDER system provides the instructions: LOAD, UNLOAD (see 6.14, 6.29), and UPDATE (see 7.26).

The system declares the following variables:

N_`<table name>` is an integer whose value is the number of records of the table.

NOM_`<name of a simple table>` is a constant array of N_`<name>` elements that contains the values (usually key names) of the first field of the table.

For each table integer constants are declared with the names in the first field and successive values 1, 2, 3 ... N_`<name>` . They are useful in the program to refer to records by name.

The types of the fields may be: REAL, INTEGER, STRING or BOOLEAN.

Example:

```
DBTABLES ( \Pop\Group), ( \Econ\Goods),
           ( \Modat\ConsPatt: 2, \Modat\Consp: 2);
```

The file `Group` in the directory `Pop` may be:

```
Fields     Group      Size     Income    Savings   EmplFrac
Records
 1         Entrep1    20000 4275000.00  870000.00      1.00
 2         Entrep2   300000 1875000.00  322000.00      0.98
 3         Entrep3   850000  975000.00   52000.00      0.97
 4         Worker1   560000  675000.00    5000.00      0.92
 .         ...................................         ....
 9         SelfEmp   130000   52500.00   10000.00      0.72
```

This table contains some characteristics of different population groups.

The system generates the constants `N_Group` (value 9) and the array `NOM_Group`

(values `'Entrep1', 'Entrep2', .. , 'SelfEmp'`).

The constants `Entrep1 = 1, Entrep2 = 2,.., SelfEmp = 9` are also defined.

An array of length 9 is dynamically allocated in memory at run time for each field. The instruction `LOAD` imports the data from the file table. So the instruction:

```
LOAD(Group, Size, Income);
```

loads the record values of the table `Group` into the system created arrays `Size` and `Income`. So, `Income[Worker1] = 675000`. The model program can now use and change the values of these array. The instruction `DBUPDATE` can pass these modified data to the original file or to other file with the same structure. When the values are no longer needed in the simulation they can be discarded by an `UNLOAD` instruction and the memory they used is freed.

If the table `Goods` have the following fields and values:

Fields	Goods	Production	PrizeIndex
Records			
1	Housing	380000	1.10
2	Food	7850000	1.05
3	Clothing	56000	1.20
.	...	...	...
7	Health	22400	1.32
8	Education	837000	1.22

Then the system generates `N_Goods = 8; NOM_Goods = 'Housing', 'Food'`, etc. and the constants: `Housing = 1`, `Food = 2`, etc. This table contains some characteristics of certain goods. The table `ConsPatt` will describe the consumption pattern of the population groups indicating how much each group consumes of each good. One part of this table may be:

Fields	Group	Goods	Consumpt	Priority
Records				
1	Entrep1	Housing	150400	8
2	Entrep1	Food	250100	5
3	Entrep1	Clothing	170000	5
.	...	...	...	.
7	Entrep1	Health	230000	7
8	Entrep1	Education	296000	8
9	Entrep2	Housing	75000	7
10	Entrep2	Food	92000	6
11	Entrep2	Clothing	80600	5
.	...	...	...	.
15	Entrep2	Health	119000	7
16	Entrep2	Education	104000	7
.	...	...	...	.
65	SelfEmp	Housing	2500	5
66	SelfEmp	Food	3060	8
67	SelfEmp	Clothing	1200	6
.	...	...	...	.
71	SelfEmp	Health	1350	6
72	SelfEmp	Education	1095	7

When this table is loaded the system pass the values to an array called `ConsPatt` with two indexes and sizes of 9 rows and 8 columns. So, `ConsPatt[Entrep2, Food] = 92000`.

2.9 PROCEDURES

The user can program functions and procedures as in the base Pascal language[2]. Procedure and function declarations must be placed under the heading PROCEDURES of the DECL section.

In these functions and procedures, instructions of the base Pascal language as well as GLIDER functions can be used. In addition to the instructions STOPSCAN and BEGINSCAN, the following (all but IT, NT, DEACT, DOEVENT, DONODE) **common** GLIDER instructions and procedures, enumerated in Chapters 6 and 7, may be used. They are:

ACT, BLOCK, ASSI, CLRSTAT, DEBLOCK, DBUPDATE, ENDSIMUL, EXTFEL, EXTR, FILE, FREE, GRAPH, INTI, LOAD, MENU, OUTG, PAUSE, PUTFEL, REL, REPORT, SCAN, SORT, STAT, TAB, TITLE, TRACE, TRANS, TSIM, UNLOAD, UNTRACE, UPDATE. USE.

[2]Conforming to TurboPascal version 6.0

However names of nodes, lists, GFUNCTIONS and database tables cannot be used as formal parameters in the present version. This restricts the usefulness of these instructions in procedures.

Type declarations, procedure declarations and function declarations within a function or procedure are not allowed.

Examples:

```
PROCEDURES
 PROCEDURE CountActBar(VAR n: INTEGER);
           BEGIN
              ACT(Bar,0); n := n + 1;
              WRITELN('Bar activated', n, ' times!!!');
           END;

 FUNCTION CubicRoot(z: REAL): REAL;
          BEGIN CubicRoot := EXP(LN(z) / 3 END;

 PROCEDURE FirstTrue(VAR Count: BoolArray; i: INTEGER);
           VAR j: INTEGER;
           BEGIN
             j := 1;
            WHILE (j <= 4) AND NOT Count[j] do j := j + 1;
            IF j = 5 THEN i := 0 ELSE i := j
           END;
```

Calls to these procedures may be:

```
CountActBar(N);
CubicRoot(1728.0);
FirstTrue(CON, k);
(It was defined:  TYPE BoolArray = ARRAY[1 . .4] OF BOOLEAN;)
```

2.10 STATISTICS

The GLIDER system collects statistics of nodes and variables. The nodes and variables for which statistics are desired must be specified by the user. After the heading STATISTICS, a list of node identifiers and simple real variable identifiers may follow.

The variables for which statistics are required, must be initialized in INIT.

If one of the elements of the list is the reserved word ALLNODES, the statistics of all nodes are given.

The statistics are always displayed at the end of the simulation run, but they may also be called by the STAT procedure in the program or by user's interruption during the execution of the program.

Examples:

```
STATISTICS Dock, Crane, TotalWeight;
```

Statistics of the nodes `Dock` and `Crane`, and the variable TotalWeight will be displayed.

```
STATISTICS ALLNODES, TotalCash, Money[1], Money[2],
           NumberTrans;
```

Statistics of all the nodes, and the variables `TotalCash, Money[1], Money[2],` and `NumberTrans` will be displayed.

Chapter 3

INITIALIZATIONS

In the INIT section of a GLIDER program the user puts the instructions to give the particular values to the data to be used in a simulation run. These are assignation for:

1. Initial values to simple variables an arrays.
2. Initial capacity values to R type nodes.
3. Values of functions given by pairs of values.
4. Values of frequency tables parameters.
5. Values from DBASE IV tables to arrays.
6. Initial activation of nodes.
7. Experimental variables

3.1 Initial Values to Simple Variables and Arrays

For simple or indexed variables assignative instructions are used;
Examples:

```
Pressure := 15; Temp := 275.0; R := 8.2056E - 2; n:=4; CH := 'Y';
Volume := n * R * Temp / Pressure;
FOR i := 1 TO 7 DO Concentr[i] := Fconc;
```

`Fconc` may be a user's defined function, a FREQ type function, etc.
For arrays the ASSI instruction (see 6.2) may be used:
Examples:

```
ASSI Conc[1..7] := (0.22, 0.33, 0.42, 0.11, 0.13, 0.17, 0.66);
ASSI OrigDest[1..4, 1..4] := ((0.4, 0.2, 0.3, 0.1),
                              (0.1, 0.6, 0.1, 0.2),
                              (0.2, 0.4, 0.1. 0.3),
                              (0.2, 0.2, 0.5. 0.1));;
```

Reading from keyboard or files may be used:
Examples:

```
WRITE('Relative Humidity '); READLN(RelHum);
i := 1;
WHILE NOT EOF(AbsRateFile) DO
 BEGIN  READLN(AbsRateFile, AbsRate[i], i := i + 1 END;
```

3.2 Initial Capacity Values to R Type Nodes

For simple or indexed R nodes an assignation is used. For multiple nodes an ASSI instruction (see 6.2) may be used. The initialization updates the following system variables:

- M_<node name> that contains the maximum capacity.
- U_<node name> that contains the actual used capacity. It is initialized to 0.
- F_<node name> that contains the actual free capacity. It is initialized to M_<node name>.

The assignation must be also made in any node. However, if a new capacity are assigned to a R node, less than the actual value of U_<node name> the system does not made any warning. Updating of F_<node name> or U_<node name> is not automatically made. The user must change these values (see example below). The IL of the node is not altered.
Examples:

```
Storage := 1E9;
Ocean := MAXREAL;
Pier[5] := 4;
ASSI ParkingLot[1..4] := (75, 34, 34, 88);
```

If at some point in the execution the values are: M_ParkingLot[4] = 88, U_ParkingLot[4] = 70, F_ParkingLot[4] = 18 and it is needed to change the capacity to 100 the following instructions must be put:
M_ParkingLot[4] := 100;
F_ParkingLot[4] := M_ParkingLot[4] - U_ParkingLot[4]; (must be 30)
If the needed change of M_ParkingLot[4] is to 50, it results:
F_ParkingLot[4] = -20; The value of U_ParkingLot[4] remains 70.
The system does not delete the excess messages in the IL. When enough resource is freed during the run, the used and free quantities will recover the normal new maximum (50) and minimum (0) possible values.

3.3 Values to Functions Given by Pairs of Values

These functions are defined in 2.6. To assign values to them a special assignation is used. Its structure is:
<name of the function> := <list of pairs>
The pairs are separated by /. Each pair is:
<value of argument>, <value of function>
Examples:

```
Rain := 0, 23 /31, 24 /59, 23 /81 ,37 /111 ,42 /141 ,48 /171 ,39
       /202 ,37 /232, 35 /263, 33 /293, 30 /329, 25 /365, 23;
Tax := 0, 0 /1000, 5 /3000, 18 /5000, 22 /12000,35;
FVehType := 1, Car /2, Van /3, Truck;
FrVesselType := 1, 342 /2, 87 /3, 21;
```

Assuming that the first is of type POLYG (linear interpolation) the value Rain(72) would be, as the 72 fall in the interval 59 81:

```
Rain[72] = 23 +(37 - 22) / (81 - 59) * (72 - 59) = 31.86
Rain[263] = 33
Rain[387] not defined
```

Assuming that Tax is of STAIR type (piecewise constant) it would be:

```
Tax(514) = 0  Tax(2999) = 5  Tax(11500) = 22  Tax(12000) = 35
```

If VehType is of type DISC, then: VehType[3] = Truck VehType[1.5] not defined.
Assuming that FVesselType is of FREQ type then the probability that its value is 2 would be:
87 / (342 + 87 + 21) = 0.193333

3.4 Values to Frequency Table Parameters

The definition of these tables is given in 2.7. To assign values to the parameters, after a heading TABLES, the following assignation have to be made:

<table name> := <initial value>, <number of intervals>, <wide of the interval>;

<table name> is a name declared as table in DECL.
<initial value> is a real value.
<number of intervals> is an integer value.
<wide of the interval> is a real value.

The system adds lower class for the values below <initial value> and an upper class for the values beyond the last class.
Examples:

```
TABLES  TABTba := 0.0, 8, 5.0; TFrTemp := 32, 12, 5;
```

TABTba is a frequency table for values from 0 to 40 (8 intervals of wide 5).

3.5 Values from DBASE IV Tables to Arrays

The definition and use of these tables are explained in section 2.8. To store the values in the DBASE IV tables into arrays the instruction LOAD (see 6.14) is used. This instruction may be used in the NETWORK section too.

3.6 Initial Activations

Some node must be activated in the INIT section in order to start the simulation run. This is made by an ACT procedure (see 7.1).
Example

```
ACT(Hairdresser, 5);
FOR i := 1 TO 5 DO ACT(Pump[i], UNIF(0.0, 0.2));
```

3.7 Interactive Experiments

Values assigned to variables in the INIT section can be changed interactively during the simulation or between different runs of the same model. To do that, the assignation instructions must be included between the words EXPER and ENDEXP. The variables in this range are called **experimental variables**. The assignation must be of the form:

<identifier> := <value(s)>

<identifier> corresponds to a experimental variable of the type:

- simple variable.
- element of an array.

- name of a type **R** node.
- functions given by pairs of values.
- name of a frequency table.

When the option **read Experiment and continue** is selected in the Menu, the system displays the values of these variables in the form:

`<identifier> = <value(s)>`

and it askes for a file name. This file will contain the assignation of the experimental variables in the defined format. The order and quantity may be different of those of the declaration in the INIT.
Example:

```
EXPER
TSIM := 6000;  A := 5.08;
Fx := 1. 3, 8 / 5, 2 / 8, 1;
Reci := 6.0;
S := 7;
Table2 := 0, 12, 2, 5;
ENDEXP
```

The program will be run with the assigned values of these variables. If the E option in the run interactive menu (see section 10.5) is selected, the system prompts:

ENTER NAME OF DATA FILE

The user must write the name of an input data file (may be CON for the keyboard). The file must begin for a line with a title for the experiment and lines follows of assignation like those in the INIT without the characters / ; or : For instance:

```
Fx = 1.3 8   5 2
     8 1
S := 7   Table2 = 0 12 2
TSIM = 6000
END
```

The system reads the data and assigns the values to the variables until the END is found. The system display the new values that can be corrected or accepted by the user. When accepted, the simulation continues.

Chapter 4

SYSTEM PREDEFINED NODES

The GLIDER system provides different types of nodes that differ in the activation and message processing. They are summarized in the Introduction. This chapter gives a detailed description and examples. In the following we refer to **GLIDER common instructions and procedures** to instructions and procedures that can be used in any type of node. In general are instructions or procedures whose action does not depend on the message being processed. They are:

ACT, BEGINSCAN, BLOCK, ASSI, CLRSTAT, DEACT, DEBLOCK, DBUPDATE, DOEVENT, DONODE, ENDSIMUL, EXTFEL, EXTR, FILE, FREE, GRAPH, INTI, IT, LOAD, MENU, NT, OUTG, PAUSE, PUTFEL, REL, REPORT, SCAN, SORT, STAT, STOPSCAN, TAB, TITLE, TRACE, TRANS, TSIM, UNLOAD, UNTRACE, UPDATE, USE.

BEGINSCAN and STOPSCAN are included here, because they can be used in any nodes as part of the associate instruction of SCAN.

The admissible Pascal and GLIDER instructions, functions and procedures, are listed in chapters 6, 7 and 8.

4.1 I (Input) type nodes

This type of nodes are used to simulate entrance of items to the system. An I type node creates messages and sends them to the successor nodes.
It has neither EL nor IL.

4.1.1 Code

It may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions and the GLIDER instructions SENDTO and COPYMESS.

4.1.2 Activation

It may be activated only by an event that points to the node, and only during the first scanning of the network. If it has an index, only the node with the index of the event is activated.

4.1.3 Function

When activated:

1. If there is an IT instruction, a next activation of the node is scheduled to happen at the time TIME + IT.
2. A message is generated that always contains the default fields:

- GT Generation Time: contains the value of TIME at the generation.
- USE quantity of resource to be used in node R. Initialized to 1 as default value.
- ET to put the time of entrance in a list (EL or IL).
- XT to put the time of future exit from a list (EL or IL).
- NUMBER number of the message generated in the node.
- NODE name of the node.

The user can change the USE by a USE instruction.
NUMBER, NODE and GT remain constant.
ET are changed by the GLIDER system each time the message enters a list.
XT is updated each time the message enters a IL and a departure time from it is scheduled.

3. The user's instructions are processed. These may usually be assignation of values to the message fields and SENDTO instructions to the message to other nodes (one and only one SENDTO is to be executed).
   Assignation of values to the fields are made by assigning values to the corresponding field variables. If there is not a SENDTO instruction in the code, the message is automatically sent to the successor node. If there are more than one successor, a copy is sent to each successor.

4. Before the sending of the message the actual values of the field variables are passed to the message fields.

4.1.4 Relation with other nodes

The node must have successors that can accept messages in their EL, but it must not have predecessors.

4.1.5 Indexes

In a multiple node the activations of the nodes occur independently. In this case, a successor must be explicitly indicated. If the successor has index too, then a SENDTO instruction may indicate the destination node.

4.1.6 Examples

1. `I::`

   This node can only be activated from the INIT section or from another node. It generates a message and sends it to the successor node (by default the following in the program).

2.
```
 MainEntrance (I) Office1, Office2 ::
   IF Number < 40 THEN IT := TRIA(TBAMin, TBAMode, TBAMax);
   ClassOff := FunClass;
   IF ClassOff = Normal THEN SENDTO(Office1) ELSE SENDTO(Office2);
   ------------------------------------------------------------

INIT TIME := 5; TSIM := 2000; ACT(MainEntrance, 0);
    TBAMin := 3; TBAMode := 5; TBAMax := 8;
    FunClass := Normal, 4 / Special, 1;

DECL TYPE Tc = (Normal, Special);
    VAR TBAMin, TBAMode, TBAMax: REAL;
    MESSAGES MainEntrance(ClassOff: Tc; h: REAL);
    GFUNCTIONS FunClass FREQ(Tc): REAL: 2;
```

Messages are generated and sent to `Office1` or `Office2` according to the field `ClassOff`, whose value is given by the function `FunClass`. Up to 40 messages are generated. The time between successive generations are taken from a triangular distribution.

3.
```
Samples (I) [1 .. 7] LabSec[INO] :: IT := EXPO(MeanArrT[INO]);
     BactConc := GAMMA(MBac[INO], 0.2 * MBac[INO]);
     SENDTO(LabSec[INO]);

LabSec (R) [1 .. 7] :: STAY := 0.031 * BactConc;

INIT  TSIM := 7200;
 ASSI  MeanArrT[1..7] := (3.2, 3.6, 3.3, 3.8, 4.0, 3.97, 3.5);
 FOR I := 1 TO 7 DO ACT(Samples[I], EXPO(MeanArrT[I]));
 ASSI MBac[1 .. 7] := (98, 106, 86, 113, 102, 122, 104);

DECL
 VAR MBac: ARRAY[1 .. 7] OF REAL; I: INTEGER;
     MeanArrT: ARRAY[1 .. 7] OF REAL;
 MESSAGES Samples(BactConc: REAL);
```

The messages (samples to be analyzed in a Lab) arrive at random from seven original points and after being assigned a value for `BactConc` by means of a GAMMA function, they are delivered to one of the seven nodes `LabSec` (according to their source).

4.
```
Hangar (I) Quarry[INO], Mine, City :: SENDTO(City);
          COPYMESS(5) IF ICOPY <= 3 THEN SENDTO(Quarry[ICOPY])
                                    ELSE SENDTO(Mine);
```

The produced message (Truck) is sent to `City`. Five copies are made. The 1,2, and 3 are sent respectively to `Quarry[1]`, `Quarry[2]`, and `Quarry[3]`. The other two (4 and 5) are sent to `Mine`.

4.2 L (Line) type nodes

This type of nodes are used to simulate queue disciplines and sorting lines of items. Messages coming to the node EL are passed to its IL, where they are set in the order indicated by the code. This may be FIFO (First In First Out), LIFO (Last In First Out) or any other order programmed by the user.

4.2.1 Code

It may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions, the GLIDER procedures LIFO, FIFO and ORDER, and the GLIDER instructions SENDTO, ASSEMBLE, SYNCHRONIZE and CREATE.

4.2.2 Activation

It is activated by an event that refers to it or during the scanning of the network if the EL is not empty.

4.2.3 Function

1. When the node is activated the EL is scanned and each message is extracted one by one. For each extracted message the fields are transferred to the corresponding field variables and the code is executed. If there are not LIFO, ORDER, or SENDTO instructions in the code the message is put at the end of the IL.

2. If FIFO is executed, the message is put at the end of the IL.

3. If LIFO is executed, the message is put at the beginning of the IL.

4. If ORDER is executed, the message is put in the IL according to the specification of this procedure (See 7.15) that may sort the IL according to the values of a variable.

5. If SENDTO is executed the message is not added to the IL but sent to the list(s) indicated in the SENDTO.

6. Before the transfer the field variable values are passed to the messages fields. Only one of the instructions LIFO, FIFO, ORDER, or SENDTO must be executed if some of them are in the code. The messages in the IL are not extracted automatically in the node. To extract a message from the IL a user's instruction must be executed (in this or other node) or the action of automatic instructions from other nodes (see below) are required.

4.2.4 Relation with other nodes. Identifying of IL with EL

The node must have predecessors that can send messages and may have successors that accept messages. None of them may be other L type nodes.
If the only successor is a node of G, E, R or D type, or the general type with EL, then the IL of the L node is **identified** with the EL of the G, E, D, R or general node. The multiplicity must be the same. These successors will manage automatically the IL of the L type node.

4.2.5 Indexes

A multiple L type node has multiple ELs and ILs. They correspond each other. A message in the j EL is passed to the j IL. The variable INO has the number of the node being executed.

4.2.6 Examples

1.
```
L ::
```
   The incoming messages are stored into the EL_L. When the node is activated they pass to the IL_L in the same order (FIFO discipline)

2.
```
Queues (L) [1..5] Gate[INO], Out::
 IF LL(EL_Gate[INO]) >= 4 THEN SENDTO(Out) ELSE FIFO;
```
   The messages in the ELs are sent to the corresponding ILs (which are the ELs of the `Gate` node). When one of this lists has 4 messages or more, the messages are sent to Out.

3.
```
OrdQueue (L) E ::  IF LL(IL_OrdQueue) > 15
                    THEN SENDTO(E)
                    ELSE IF Priority < 4 THEN FIFO
                                         ELSE ORDER(Priority, D);
```
   If the queue is greater than 15 the message is sent to `E`. Otherwise, if the field Priority is 4 or more it is ordered by descending Priority, otherwise by increasing time of arrival.

4.
```
QFiles (L) :: P := A / Length_a + C; ORDER(P, A);
```
   The messages are ordered by the ascending values (A) of the variable `P` computed from the field `Length_a` and the parameters `A` and `C`.

4.3 G (Gate) type nodes

This type of node retains or allows message flow. The nodes type G have an EL for the retained messages.

4.3.1 Code

It may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions, the GLIDER instructions ASSEMBLE, COPYMESS, CREATE, DEASSEMBLE, SYNCHRONIZE, SENDTO, STATE and the procedures STOPSCAN and BEGINSCAN. If there is a STATE instruction (only one is allowed), it must be the first in the code. Its associated instruction may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, and GLIDER functions. It cannot have SENDTO instructions.

4.3.2 Activation

It is activated by an event or during the scanning of the network if the EL is not empty. The instruction STATE, used to change the state of the gate, is only executed if the node is activated by an event which refers to the node.

4.3.3 Function

1. When the node is activated by an event that refers to the node and there is an instruction STATE, this is first executed. After this execution or when the node is activated because the scanning of the network, the other (non STATE) part of the code may be executed.

2. If the EL is not empty, it is scanned starting from the first message. Each message is examined, the values of the fields are passed to the corresponding field variables and the user code is executed. After this execution, if the message was not extracted, the fields are updated to the values of the field variables (that could have been changed) and the scanning continues.

3. If in the above process a SENDTO instruction is executed, the fields are updated, the message is extracted and sent to the indicated list(s) or successor node(s). **The scanning of the EL continues**.

4. If a STOPSCAN procedure is executed, the scanning is stopped and the process of the node finishes. However, it may be re-started during the same event if the scan of the network activates again the node. STOPSCAN is used, for instance, if the sending of further messages depends on the changes caused in other nodes by the actually sent message.

5. If a BEGINSCAN procedure is executed, the scanning begins again from the first element. This may be useful if the sending of a message change the sending conditions of the already examined messages. Unwise use of this procedure may cause a loop.

6. Note that the sending of message from a G type node must be explicitly ordered by a SENDTO instruction. To send a message a SENDTO must be executed one and only one time for the examined message.

4.3.4 Relation with other nodes

A G type node must have some predecessor and some successor. If the predecessor is a L type node the IL of this is the same of the EL of the G type node. If the G is multiple the predecessor L must have the same multiplicity.

4.3.5 Indexes

A multiple G type node has as many EL as its multiplicity. When it is activated all the nodes are executed sequentially. The variable INO has the number of the node being executed.

4.3.6 Examples

1. `G::`

   When activated, this node scans the EL and nothing is made. From other nodes it is possible to extract and modify the messages of its EL.

2.
```
Inspect (G) Sale, Repair :: IF Defect
                            THEN BEGIN STOPSCAN; SENDTO(Repair) END
                            ELSE SENDTO(Sale);
```

   If the boolean field `Defect` of the message (article) is TRUE, the examination of the list stops and the message is sent to `Repair`. Otherwise, it is sent to `Sale`. In the first case, the scan is stopped (will continue in the next activation during the same event); in the second case, the scan continues. Note that the STOPSCAN must be used only if it is necessary (see next example) because it increases the processing time by repeating the scanning of the network.

3.
```
Semaf (G) Street :: STATE BEGIN Green := NOT Green; IT := 25 END;
                    IF (LL(IL_Street) < 20) AND Green
                    THEN SENDTO(Street); STOPSCAN;
```

   Each 25 units of time the boolean variable `Green` changes its true value. The node allows the messages (cars) to pass if `Green` is true and the IL of the successor node `Street` has less than 20 messages. When a message passes, the scanning of the EL is stopped to allow the change in the IL of `Street`. Without the STOPSCAN more than 20 messages would be sent to `Street`. On the other hand, if the condition is not fulfilled stopping of the scan avoids unnecessary examination of the EL. Eventually, future changes in the IL of `Street` will produce a network scanning an a new activation of `Semaf`.

4.
```
Control (G) Machine[IN0] :: SENDTO(Machine[FREE]);
```

   The message (object to be processed) is sent to the first free `Machine` (with free capacity). If there is no free `Machine`, the message remains in the EL of `Control`.

5.
```
Selection (G) ::
       IF (Typ = Special) AND First
        THEN BEGIN SENDTO(Depot); BEGINSCAN; First := FALSE END
        ELSE IF NOT First THEN SENDTO(Depot);
```

   Among the messages in the EL of `Selection` there is one and only one with the field `Typ = Special`. Only after it pass all, the others may pass. The boolean variable `First` is originally TRUE.

4.4 R (Resource) type nodes

This type of nodes simulates resources used by the entities represented by the messages. A real value, called the node **capacity**, is associated to the node. The messages in the EL represent entities that demand a certain quantity of that capacity during a certain time. The EL is examined and, for each message, it is checked if the demanded quantity is available. If it is so, the message is moved to the IL, the time of future depart is scheduled in the FEL, and the quantity of resource used for the message is subtracted to the available capacity. If there is not enough capacity, the message remains in the EL. When the departing event is executed, the message is removed from the IL and sent to the successor nodes. So, the node has two function an **accepting function** that assigns resource to a message and put it in the IL and a **departing function** that extracts the message of the IL and frees resource.

4.4.1 Code

The type R node may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions, the GLIDER instructions PREEMPTION, RELEASE, STAY, If there is an instruction RELEASE (only one is allowed) it must be the first in the code. Its associated instruction may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures and GLIDER functions, the instructions SENDTO, DEASSEMBLE, EXTR, CREATE, and COPYMESS, and the procedures NOTFREE, BLOCK and DEBLOCK.

4.4.2 Activation

The part of the code that processes the departing message (that may be a system generated code or a user's programmed RELEASE instruction) is only activated by a departing event that refers to the node. This event can be only introduced in the FEL by the node itself when a STAY instruction is executed. The node must not be activated by an ACT instruction.
The part that controls the entrance of the messages to the IL is activated during the scanning of the network when the EL is not empty.

4.4.3 Function

1. If there is no RELEASE instruction, the departing of a message from the IL is automatically made as follows: when the depart event pointing to the node is executed, the message with the time of depart XT equals to the actual value of TIME (i.e., the time of the departing event) is extracted from the IL and sent to the successor nodes. If there are more than one successor node, copies are sent to all of them. The value of the freed and used resource account is updated (see 3 below) The part of the code used by the system to process the departing message is only executed one time in the event and is not repeated during the scanning of the network.

2. The user can control the departing process of the message by means of a RELEASE instruction. This is only executed if the R type node is activated by an event that refers to the node. The associated instruction is executed only one time at the beginning of the event and not repeated during the scanning of the network.
   The execution begins, as in the automatic case, by seeking the first message with an exit time XT equal to the actual TIME. If it is not found, a message error of "empty list" is given. If it is found, the message is extracted, the fields are copied in the field variables and the user's code in the associated instruction is executed. This code must have a SENDTO instruction to send the message to one (or various) successor nodes. Before this sending, the fields of the message are updated at the values of the field variables.

3. After the sending of the message the capacity of the node is updated adding the value of the USE field to the free capacity (variable `F_<node>`) and subtracting the value of USE to the used capacity (variable `U_<node>`). This resource freeing can be inhibited if a NOTFREE procedure appears in the associated instruction of the RELEASE instruction. This is used in applications in which some items abandon the resource, but this requires an additional time to be available again. The freeing is made elsewhere by a FREE procedure at a latter time.

4. The part of the code that pass the messages to the EL (accepting function) is executed when the EL is not empty. It examines the messages in the EL starting from the first. For each examined message the fields are passed to the field variables. and the user's instructions in this part are executed. One of them may be a `STAY := <expression>` instruction. The required amount of resource (USE field) is compared with the available resource (`F_<node>` variable). If USE is greater than `F_<node>`, the message remains in the EL and the scanning of the EL continues. Otherwise, the message is extracted, the event of its future departing is scheduled (if there was an STAY instruction) and the message is added at the end of the IL.
   The capacity of the R node is updated subtracting the value of the USE field of the message to the free capacity (variable `F_<node>`) and adding the value of USE to the used capacity (variable `U_<node>`). The scan of the EL continues to look for other candidates to use the resource. If there is not STAY instruction and one message enters the IL, no departing event is generated. The message will remain in the IL until it is extracted by an instruction REL (see 6.18), that refers to the IL of the node. It extracts the first message without assigned exit time; it updates the quantity of free and used resource

and it sends the message to the desired node.
It is important to remark, that the EL is examined and the user's code executed even if there is not resource available. If the user wants to avoid repetition of all the code or part of it, she or he must program the conditions for that omission.

Example:

```
IF F_Parking > 0 THEN STAY := LOGNORMAL(TPark, DTPark)
```

avoids to compute the STAY if there is no room in Parking.

5. If there is the instruction PREEMPTION (see 6.17), some of the above operations are modified.

6. If the capacity of the node was defined as MAXREAL, it is assumed to be infinite and no update is made in the free or used capacity.

4.4.4 Relation with other nodes

A R type node must have predecessors and successors. If the predecessor is a L type node, the EL is the same IL of the L node. Both nodes must have the same multiplicity.

4.4.5 Indexes

Multiple nodes are processed successively, when the node is activated during the scanning of the network. The INO variable takes the value of the index of the processed node.
If the multiple node is activated by a departing event, only the departing part (system produced or RELEASE) of the node with the index of the event is processed.

4.4.6 Examples

1.
```
Disposal (R) ::
--------------

INIT  Disposal := MAXREAL;
```

The node `Disposal` enters any quantity of messages (compatible with memory available) in its IL . They may be extracted by a REL instruction in other node.

2.
```
Crane (R) Base :: RELEASE BEGIN Transported := TRUE;
                               NOTFREE; SENDTO(Base);
                               END;
                               STAY := GAMMA(Travel_T, Dev_T);
```

If the `Crane` is free (capacity 1 is assumed), a message (box) takes it; it uses the resource for a time took from a GAMMA distribution. When this time is over, the boolean field `Transported` is put to TRUE and the message is sent to `Base` without freeing the resource. This must be freed by a REL instruction.

3.
```
Sea (R) :: STAY := FTravelTime(VesselType);
----------------------------------------

INIT    ...       Sea := MAXREAL;
```

The messages (vessels) enter the resource `Sea` of infinite capacity and remain there for a time given by a function `FTravelTime` that depends on `VesselType`.

4.5 D (Decision) type nodes

This type of node selects messages from predecessor nodes and sends them to successor nodes. **D type nodes have an EL for each predecessor**. Some messages are taking from the ELs and sent to the successor nodes according to the selection rules indicated by the code.

4.5.1 Code

It may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions, the GLIDER instructions SELECT, SENDTO, ASSEMBLE, DEASSEMBLE SYNCHRONIZE, CREATE and COPYMESS.
If the node has more than one predecessor, the SELECT instruction must be used.

4.5.2 Activation

It is activated by an event that refers to it or during the scanning of the network, if some of the ELs is not empty.

4.5.3 Function

1. If there are only one EL, each message, from the first to the last, is extracted from the EL. For each message the values of the fields are passed to the field variables and the code is executed. The code must have a SENDTO instruction for the message. The fields are updated before the message is sent to some list.

2. If there are many predecessors, there is one EL for each. Then a SELECT instruction must be executed, that selects messages from some EL and sent them to other nodes. See 6.22 for details of this instruction.

4.5.4 Relation with other nodes

A type D node must have predecessors and successors.

4.5.5 Indexes

A type D node cannot have indexes.

4.5.6 Examples

1.
```
Queue1 ::
Queue2 ::
Queue3 ::
ReArrange (D) Proc1, Proc2 ::
 SELECT(Queue2, Queue3, Queue1, J) IF Size = 2 THEN SENDTO(Proc1)
                                              ELSE SENDTO(Proc2)
```

   The `ReArrange` node takes J messages from each `Queue` (in the order in which they appear in the SELECT) and according to the value of the field `Size` it sends them to `Proc1` or `Proc2`.

2.
```
 Divide (D) :: IF Good THEN SENDTO(FIRST, Pile)
                       ELSE SENDTO(Garbage);
```

4.6 E (Exit) type nodes

These nodes destroy messages. The messages in the EL are processed. The code in the node is executed and the messages are destroyed. The node is activated during the scanning of the network if the EL is not empty.

4.6.1 Code

It may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions, and the GLIDER instructions SENDTO, CREATE and DEASSEMBLE,

4.6.2 Activation

It is activated by an event that refers to it or during the scanning of the network, if the EL is not empty.

4.6.3 Function

1. The EL is scanned from the beginning to the end.
2. For each examined message the values of the fields pass to the field variables and the code is executed.
3. If there is a SENDTO instruction, the fields of the message are updated and the message is sent to other node. Otherwise, the message is destroyed and the used memory may be re-used.

4.6.4 Relation with other nodes

An E type node must have predecessors and may have successors.

4.6.5 Indexes

If a multiple E type node is activated the nodes are executed successively. The value of the variable INO corresponds to the node being processed.

4.6.6 Examples

1. `E ::`

   The messages arriving at E are destroyed.

2.
```
Exit (E) :: Tis := GT - TIME; Tab(Tis: TablST);
------------------------------------------

INIT
   TablST := 0.0, 12, 5.0;
------------------------------------------

DECL
   TABLES Tis: TablST;
```

   The time that the messages remained in the system is computed by subtracting from the actual TIME the generation time GT. This value is recorded in a frequency table.

4.7 C (Continuous) type of nodes

This nodes solve systems of ordinary differential equations of first order. This nodes do not have EL or IL.

4.7.1 Code

It may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions, the GLIDER instruction CREATE, and the procedures RETARD and METHOD (see 7.18 and 7.13).
It accepts successions of instructions of the form `<variable>' := <expression>`. These may be mixed with other type of instructions. The system considers a series of such instructions of the above type without other type of instructions among them as a system of differential equations. So, various differential systems can be represented in a unique node separated by ordinary instructions. The systems of the same node share a common METHOD and path of integration. In different C type nodes the METHOD and path of integration may be different. In the above instruction `<variable>` is a variable that was declared of CONT (continuous) type. It may have a numerical index. If the variable TIME appears in some equation, it must be represented by this name. If the value is transferred to other variable, then both will differ without knowledge of the user, because **the solving algorithm changes the variable TIME** without updating the user's variable.

4.7.2 Activation

The node is first activated from the INIT section or from other node. Then the solving process starts. During the execution the node activates itself at each integration step. This activations does not produce scanning of the network. They may be interrupted and continued when the node is deactivated or activated by the instructions ACT and DEACT.

4.7.3 Function

1. In a C type node one or more systems of differential equations are solved. The whole code is executed at each integration path and the execution of the events of all the program interleaved with those of the differential equations. On the other hand, the C node can generate events and messages by conditions that may depend on the values computed for the differential equations. Thus, a symbiosis of continuous and discrete simulation is complete.

2. As the self-activations of the node do not produce scanning of the network, all actions in the network that depend on the values computed by the differential equations must be explicitly programmed in the node (see examples below). The C type node can be called to be executed at any time from any other node. The calling instruction is:

   ```
   <name of the C node > (0);
   ```

   When this happens the node solves the systems for the actual value of the time and then the control goes back to the calling code. So an updated value of the computed variables can be used by the calling code. The C type node continues solving the system at the originally prescribed intervals.

4.7.4 Relation with other nodes

The node has not predecessors but it can have successors to which messages may be sent. As was explained above, the node can be called from any node for an instantaneous evaluation of the continuous variables.

4.7.5 Indexes

Type C nodes cannot have index.

4.7.6 Continuous Variables Declaration

The variables that appear as derivatives must be declared of CONT type. All the CONT and RET type variables must be declared together.
Example:

```
C1, C5: CONT; Rr: REAL; H1, H2: CONT;  ExitPro: RET: 3;
```

is incorrect. A correct declaration is:

```
C1, C5: CONT; ExitPro: RET: 3; H1, H2: CONT; ExitPro: RET: 3; Rr: REAL;
```

or may be:

```
C5, H2, C1, H1: CONT; ExitPro: RET: 3; Rr: REAL;
```

note that the order within the list is irrelevant.

4.7.7 Examples

1. Tank Source for TankCar.

```
A tank receives a flow of water given by a function FEntry(TIME)
and has an outflow proportional to the Level of the tank. This is
expressed in the differential equation for the Level. When the
Level exceeds 3m the SendTankCar subsystem is activated. Further
sending is inhibited until the TankCar completes its work. The
SendTankCar subsystem sends the TankCar to Travel. When the Travel
(to the place in which the Tank is) finishes the Level is lowered
in a quantity corresponding to the volume of the TankCar. The
TankCar is eliminated and the SendTankCar system may be activated
again if Level > 3. There is an hourly Inspection that puts Alarm
to TRUE when it finds the Level less than 2. Then the Recovering
process begins by decreasing the rate of outflow until the Level
3.2 is reached.

NETWORK
Tank (C) ::  Level' := KQLevel * FEntry(TIME) - KOLevel * Level;
                IF Alarm OR Recovering THEN KOLevel := 0.0018
                                       ELSE KOLevel := 0.0030;
                IF (Level > 3.0) AND Send
                THEN BEGIN ACT(SendTankCar, 0); Send := FALSE END;

SendTankCar (A) Travel ::
                CREATE(TankCar) BEGIN
                                 VTankCar := UNIF(15.3, 18.0);
                                 SENDTO(Travel);
                                 END;

Inspection (A) ::    Tank(0); (*Update Level*)
                IF Level < 2.0 THEN BEGIN Alarm := TRUE;
                                      Recovering := TRUE
                                     END;
                IF Level > 3.2 THEN Recovering := FALSE;
                ACT(Inspection, 3600);

 Travel (R) ::  RELEASE BEGIN
                      Level := Level - VTankCar * KQLevel;
                      SENDTO(Exit);
                      Send := TRUE;
                      END;
```

```
                     STAY := TravelTime;

Exit ::

Gra (A)   ::  IT := 15;
               GRAPH(0, 86900, BLACK; TIME: 6: 1, WHITE;
                     Level: 6: 0, Alt, 0, 10, GREEN);

INIT  ACT(Tank, 0); ACT(Inspection, 0); ACT(Gra, 0);
          TSIM := 86400;
          TravelTime := 240;
          Travel := MAXREAL;
          DT_Tank := 1.0; Level := 2.0;
          KQLevel := 0.03;    (* 1/M*M   *)
          K0Level := 0.0035;  (* 1/Seg    *)
          FEntry := 0.0, 0.4 / 14000.0, 0.6 / 27000.0, 0.3 /
                         52000.0, 0.2 / 86900.0, 0.4;
          Alarm := FALSE; Recovering := FALSE;
          Send := TRUE;

DECL VAR KQLevel, K0Level, TravelTime: REAL;
                    Level: CONT; Alarm, Send, Recovering: BOOLEAN;
   MESSAGES TankCar(VTankCar: REAL);
   GFUNCTIONS FEntry POLYG (REAL): REAL: 6;
   STATISTICS ALLNODES;

END.
```

2. TEST OF DIFFERENTIAL EQUATIONS

Computed known solutions of some systems of differential equations.

```
NETWORK
C1 :: Y'    := R;           (*Y := SIN(T)*)
      X'[1] := 0.1;         (*X[1] LINEAR SLOPE 0.1*)
      X'[2] := 0.2;         (*X[1] LINEAR SLOPE 0.2*)
      R'    := -Y;          (*R := COS(T)*)
      X'[3] := X[1];        (*X[3] PARABOLA*)
      Z'    := -R;          (*Z := -SEN(T)*)

C2 :: Z'    := 0.15;        (*X[1] LINEAR SLOPE 0.01*)
      X'[1] := -X[3];       (*X[1] COS(T)*)
      X'[3] := X[1];        (*X[3] -SIN(T)*)
      S'    := Z - 2;       (*X[1] PARABOLA*)

Gra1(A) :: IT := 0.0625;
    GRAPH(0, 100, BLACK; TIME: 6: 1, WHITE;
               Y: 6: 2, SEN, -1, 1, RED;
               R: 6: 2, COS, -1, 1, GREEN;
               X[1]: 6: 2, Lp01, -10, 10, YELLOW;
               X[2]: 6: 2, Lp02, -10, 10, MAGENTA;
               X[3]: 6: 2, Parab1, -10, 10, LIGHTGREEN;
               Z: 6: 2, MSen, -2, 2, BLUE);
```

```
Gra2(A) :: IT := 0.0625;
GRAPH(0, 100, BLACK; TIME: 6: 1, WHITE;
      X[1]: 6: 2, Lp01, -10, 10, YELLOW;
      X[3]: 6: 2, Parab1, -10, 10, LIGHTGREEN;
      Z: 6: 2, MSen, -2, 2, BLUE;
      S: 6: 2, Parab2, -10, 10, BROWN);

INIT TSIM := 100; Tc := 0;
      INTI  Tc: 3: Tc (0 FOR C1, 1 FOR C2) ;
      IF Tc = 0 THEN BEGIN ACT(C1, 0); ACT(Gra1, 0) END
                     ELSE BEGIN ACT(C2, 0); ACT(Gra2, 0) END;
      DT_C1 := 0.0625; DT_C2 := 0.0625;
      Y := 0; R := 1; X[1] := 0; X[2] := 0; X[3] := 1; Z := 0; S := 1;

DECL VAR X: ARRAY[1 .. 3] OF CONT; Z, S, Y, R: CONT; Tc: INTEGER;

END.
```

4.8 A (Autonomous) type nodes

These nodes execute their code at scheduled times. They have neither EL nor IL. They may activate themselves and other nodes, change variables and send messages.

4.8.1 Code

It may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions, and the GLIDER instructions CREATE and SENDTO. The common instruction UPDATE can only be used in the associated instruction of CREATE or SENDTO.

4.8.2 Activation

An A type node is only activated by an event that refers to it. It is not activated again during the scanning of the network. If the node is multiple, only the node with index value equal to the index of the event is activated.

4.8.3 Function

When the node is activated the code is executed.

4.8.4 Relation with other nodes

It have not predecessors but may have successors if messages are created in it and sent to other nodes.

4.8.5 Indexes

A multiple type A node is a set of independent nodes that can be activated independently.

4.8.6 Examples

1. `Print (A) :: IT := 365; Annual;`

   Each 365 unit times the procedure Annual is called (for instance to print a report of the state of the model). The node must be activated for the first time from other node or from the INIT section.

2.
```
GraphProPri (A) ::   IT := 0.125;
                     GRAPH(0, 50, BLACK, TIME: 6: 1, WHITE;
                           Prod: 7: 1, Production, 0, 1000, RED;
                           Price: 7: 1, PriceInd, 0, 2, Yellow);
```

Each interval time of 0.125 units (8 times per year) points with the new values of `Prod` and `Price` are added to the time graphic and united by a line to the previous point. The node must be activated for the first time from other nodes or from the INIT section.

3.
```
NewStr (A) NewSource ::  IT := 0.5;
                         If CondStCh THEN
                                     BEGIN
                                       ACT(NewSource, 0);
                                       CREATE(MessSource)
                                       SENDTO(NewSource);
                                       DEACT(Inst3);
                                       ACT(Proc4, 12);
                                       STAT;
                                       CLRSTAT;
                                     END;
```

Each 0.5 units of time the boolean function `CondStCh` (that may compute conditions for a structural change) is evaluated. If it is TRUE, the nodes `NewSource` and, with a delay, `Proc4` are activated. The `Inst3` is deactivated. A message is sent to the newly activated node. The statistics are displayed and then initialized for a new statistical gathering.

4.9 General type nodes

Used to general processes programmed by the user. A node of this type may be designed by any other letter different of G, L, I, D, E, R, C, A. The node is activated during scanning processes (this is the main difference with the A type). It has not EL or IL unless this is explicitly required by the user in a NODES declaration. The management of these lists must be programmed by the user with the help of GLIDER instructions and procedures.

4.9.1 Code

It may contain Pascal instructions, functions and procedures, GLIDER common instructions and procedures, GLIDER functions, and the GLIDER instruction SENDTO, CREATE, DEASSEMBLE, ASSEMBLE and SYNCHRONIZE,

4.9.2 Activation

It is activated by an event that refers to the node or during the scanning of the network (this is the difference from an A type node).

4.9.3 Function

The activation consists in the execution of the code. All management of messages must be explicitly indicated in the user's code by means of Pascal or GLIDER instructions and procedures.

4.9.4 Relation with other nodes

Predecessors and successors are only required if the node must receive and send messages.

4.9.5 Indexes

When it is activated, all the nodes are executed. The variable INO has the number of the node being executed.

Chapter 5

NETWORK

The NETWORK section of a GLIDER program describes the relationships among the subsystems and the code that simulates their behavior. In what follows the general rules to write this section and the way in which it is processed are described.

5.1 Defining a node

The node has two parts, a heading part and a code part. The structure is:

```
<heading> :: <code>
```

5.1.1 Heading

This part declares the name, the type , the successors and the local variables. The heading has the following structure:

```
<name> / <type> / <index range> / <list of successors>
/ <LABEL: <list of labels>;> / <VAR <declarations of local variables>;> /
```

Items between // may be omitted

- `<name>` is an **identifier** (see Chapter 2) that indicates the **name** of the node. It is also the declaration of the node.
- `<type>` is a letter that designs the **type** of the node (See Chapter 1 and Chapter 4). If omitted the first letter of the name is assumed to describe the type.
- `<index range>` is a range indicator of the form: `[1 .. <positive integer>]`. If omitted the node is assumed simple.
- `<list of successors>` is a list of node names. They are the nodes to which messages may be sent. Multiple nodes must have an index. If the list is omitted and the node needs a successor, the system tries to see if the following node is a feasible successor. If so, it will be the successor. Otherwise, an error condition occurs.
- `<LABEL : <list of labels>;>`. If labels are used in the code, they must be declared here; the list is formed by numbers or identifiers separated by commas. The labels are local to the node and all must be used.
- `<VAR <list of declarations of local variables>;>`. This is to declare **local variables** as in the VAR part of the section DECL (see 2.3).

Examples:

```
Ingoing ::
Crane (R) [1..3] Belt[INO] VAR Vel: real; MaxWeight: ARRAY OF REAL; ::
Control (G) Street LABEL: 1, EndCont ::
```

5.1.2 Code

Node code may be empty. In this case, no separator ; is required after ::. If it is not empty, it may be a succession of combinations of Pascal and GLIDER elements. For instance:

- Simple Pascal instructions.

  Example:

  ```
  Results (A) :: TotCash := TotNum * Price; Quant := Quant + 1;
  ```

- Compound and structured Pascal instructions.

  Example:

  ```
  FindMax (A) :: M := 0;
                 FOR J := 1 TO NTot DO IF Bag[J] > M THEN M := Bag[J];
                 WRITELN(' The most heavy weights ', M);
  ```

- Compound and structured Pascal instructions including GLIDER instructions.

  Example:

  ```
  ConTower (D) :: IF International THEN SENDTO(Runway_I)
                                    ELSE SENDTO(Runway_N);
  ```

- GLIDER instructions.

  Example:

  ```
  Robot1 (X) Robot2 :: ASSEMBLE(ELIM, 2, EQUFIELD(L)) SENDTO(Robot2);
  ```

- GLIDER procedures.

  Example:

  ```
  Q1 (L) :: ORDER(Temp, A);
  ```

- Compound and structured Pascal instructions, including GLIDER procedures and instructions.

  Example:

  ```
  ActDec (A) :: IF Danger THEN ACT(Proc_Emer) ELSE ACT(Proc_Norm);
  ```

- A succession of the above items.

Restrictions in the use and order of the instructions in the code are explained in Chapter 4.

5.2 Processing the network

The processing of the network starts when a node is activated by an event. The procedure in the node is executed. After this a **scanning of the network** starts. The successive nodes are examined one by one in the programming order, if they are of G, L, D, E, R, or general type. The nodes of types I, A and C are skipped in the scanning. If the EL is non empty, the code is executed. When the last node in the program is scanned, the scanning continues with the first one, until the node activated at the beginning of the event is reached. If during this process a movement of messages occurred, then the scanning is repeated again. When in a cycle no message is moved, the event finishes and the next event in the FEL will be processed. This scanning makes possible to take into account the consequences that the original event produced in other nodes. Even if the original node do not move messages, the scan is necessary because it can modify restrictions that maintain certain nodes inactive. Although in most practical cases the programmer may be unaware of this process, in complicated situations may be important to consider the activation process. Some processes that would be simultaneous may fail to do so in certain cases.

Example: If in a point of time three variables must change simultaneously, for instance `X -> Z -> U` and the changes are in different nodes:

```
A :: Z := X + 6;
B ::
C :: X := X + 1;
D :: U := Z + 3;
E ::
```

then if the event starts in node C and no message is moved in the event, the scanning will not produce the change of U due to the change of X. This may be solved by changing the order of the nodes or by a procedure ACT(A, 0) in the node D. The user must decide from the nature of the problem what chain of modification makes sense and what are the adequate activations.

Problems of this type may occur also in ordinary programming, but they are more easily avoided because the order of execution is more explicit for the programmer. Another consequence of the scanning of the network is that some instruction may be unnecessarily executed many times. This may produce waste of time or erroneous results.

Example:

```
Gate (G) :: IF (Class = Good) OR (LL(EL_Gate) > 3) SENDTO(Market) END;
            Insp := Insp + 1;
```

The EL of `Gate` is examined and if the condition is fulfilled the messages are sent to Market. In `Insp` it is supposed to take into account the number of inspected messages. However, as the EL may not be exhausted in one activation of the node (up to three not Good items may remain) in a further activation in the same event the remaining messages are inspected again given a false value of the inspected number.

The user can avoid the effects of the repeated execution of some or all the instructions of a node using the instructions DONODE and DOEVENT (see 6.7 and 6.8). Example:

```
Gate (G) :: IF (Class = Good) OR (LL(EL_Gate) > 3) SENDTO(Market);
            DOEVENT Insp := Insp + 1;
```

In this case the instruction `Insp := Insp + 1` is executed only the first time the node is activated.

The type C nodes activate themselves automatically each integration interval. For example, if a message must be sent when Level reaches or exceeds 123.0, the program

```
Tank (C) ::   Level' := Inflow - K * Level;
              IF Level = 123.0 THEN CREATE(Mess) SENDTO(Center);
```

may fail if Level do not reach exactly that value. If we put:

```
Tank (C) ::   Level' := Inflow - K * Level;
              IF Level >= 123.0 THEN CREATE(Mess) SENDTO(Center);
```

after the value is reached, a message is sent during each integration interval. One solution may be:

```
Tank (C) ::   Level' := Inflow - K * Level;
              IF (MesNotSent and (Level > 123. 0) THEN
              CREATE(Mess) BEGIN SENDTO(Control);
                            MesNotSent = FALSE
                           END;
```

MesNotSent is a boolean variable originally TRUE. When the condition is fulfilled for the first time a message is sent and MesNotSent becomes FALSE, so, no more messages are sent.

Chapter 6

INSTRUCTIONS

GLIDER instructions are structured instructions characterized by a name, a list of parameters, that may be variable in length, and in many cases an associated simple or structured instruction. This last may contain certain GLIDER instructions and procedures. In this sense they are like structured instructions of general purpose languages. GLIDER instructions execute complex procedures and are translated into many basic instructions and procedure calls. They are used to create and copy messages, to manage the movements of messages, to examine and change the entry and internal lists of the nodes, to write or read files, to make graphics, etc.

In this chapter, the following conventions and definitions are used:

- names of the elements of the language are put within < >. Example: <variable> means the name of a variable.

- elements that can be omitted are within / /.

- if an element of the language is between | it means that there are other alternatives for the element. The other alternatives are also between | and only one of them must be used. Example:
  | LAST | FIRST |
  in a syntactical description, means that one of the two: the word LAST, or the word FIRST, must appear.

- <integer value> is an integer variable taking only positive values or a positive integer.
  Examples: 3, 334, J (J was defined as INTEGER, LONGINT, WORD or BYTE and must have a positive value).

- <node> means the name of a node. If the node is multiple the name must include the index within [ ].
  Examples: Machine, Carrier[3], Take are examples of node names.They must have been declared as nodes in the program (see chapters 4 and 5). The node Carrier must have been declared as a multiple node.

- <list of <elements> > means a list of the items indicated by elements separated by commas.
  Examples:
  <list of <integer values> > may be a list as: J, 3 ,350, h.
  <list of <nodes> > may be: Machine, Carrier[j], Inspect. where Machine, Carrier and Inspect were defined as nodes elsewhere. Carrier was defined as a multiple node.

- the lists of messages are denoted in the program by :
  | EL | IL|_<node> / (<name of predecessor node>)/
  The node name of a multiple node must include the index between [ ].
  If the node is of D type and has many predecessors and, for that reason, it has many EL, the name of the node must include the name of the predecessor node.
  Example: EL_Machine, IL_Carrier[2], EL_Take(Queue3) are examples of list names.

The first one is the entry list of the node Machine. The second one is the internal list of the node Carrier. EL_Take(Queue3) is the EL of the node Take, that has many ELs. It indicates the name of the EL for the messages that come from the node Queue3.

- `<list>` in the syntactical description represents the name of an IL (internal list) or EL (entry list) of messages of a node.

- `<associated instruction>` is a `<Pascal instruction>` simple or structured, that is included as part of a GLIDER instruction. The associated instruction may include GLIDER functions and procedures and the GLIDER instructions that are allowed in each case. It may also be a unique GLIDER instruction or a list of GLIDER instructions and procedures between the separators **BEGIN ... END**.
Example:

```
SCAN(EL_Machine) BEGIN j := UNIF(1, 4);
IF a < 5 THEN SENDTO(Carrier[j])
END;
```

SCAN is a GLIDER instruction (see 6.21) used to scan a list of messages (in this case the EL of the node Machine). The associated instruction between BEGIN ... END is executed for each scanned message. The variable j of the message is set to the value given by the random GLIDER function UNIF. If the value of a is less than 5 the message is taken out of the EL and sent to the EL of the node `Carrier[j]`. Here, the associate instruction is a Pascal compound instruction (between BEGIN ... END) which contains an assignative instruction and an IF instruction containing a GLIDER instruction SENDTO.

- `<field>` is the name of a simple or indexed variable that was declared as a field of a message.

- `<PASCAL instruction>` is a PASCAL instruction structured or simple.

- `<simple PASCAL instruction>` is an assignative instruction:
`<variable> := <expression>`; or a call to a procedure or a input or output instruction or a GOTO instruction.

- `<structured PASCAL instruction>` is a PASCAL instruction of the type IF, CASE, WHILE, FOR, REPEAT or a compound instruction (set of instructions delimited by **BEGIN ... END**). Where a simple PASCAL instruction is allowable, it is possible also to put a GLIDER instruction or a GLIDER procedure.
Example:

```
WHILE <logical expression> DO
 BEGIN
  <GLIDER instruction>;
  IF <logical expression> THEN <GLIDER instruction>
                           ELSE <GLIDER procedure>;
 END;
```

There are restrictions, that depend on the type of node and of the GLIDER instruction, in which the GLIDER instruction or procedure is used.
Example: In the associated instruction of a GLIDER instruction SCAN it is not allowed to use another SCAN instruction. Within an associated instruction, GLIDER instructions with associated instructions are, in general, forbidden, except in the case of the associated instruction of a RELEASE and STATE instructions.

- `<nest of constants>` is a set of simple INTEGER, REAL, CHAR or STRING constants in a nest as it is used in PASCAL to give values to indexed constants: The whole set is included within ( ); its elements are constants or successions corresponding to the successive values of the first index. Each succession is included within ( ). Each ( ) is formed by constants or successions corresponding to the successive values of the second index, etc. Successions and values are separated by commas.
Example: the array M of 3 indexes of dimension 2x3x3 formed by the two matrices of 3x3:

```
4 5 0     1 0 1
7 6 2     3 3 8
1 4 3     9 6 9
```

in which the first index, that indicates the matrix, the second the line and the third the column, is represented by:

```
(((4, 5, 0), (7, 6, 2), (1, 4, 3) ) , ( (1, 0, 1), (3, 3, 8), (9, 6, 9)))
```

The element M[2, 2, 3] has the value 8.

- `<assignative instruction>, <expression>, <real expression>, <integer expression>, <logical expression>, <real variable>, <integer variable>, <boolean variable>, <character variable>`

  are defined as in PASCAL.

- GLIDER functions may be used in the same way as PASCAL functions.

  Some of the instructions and procedures (see Chapters 6 and 7) may be used in any node. They are called common instructions and procedures. They are:

  ACT, BEGINSCAN, BLOCK, ASSI, CLRSTAT, DEACT, DEBLOCK, DBUPDATE, DOEVENT, DONODE, ENDSIMUL, EXTFEL, EXTR, FILE, FREE, GRAPH, INTI, IT, LOAD, MENU, NT, OUTG, PAUSE, PUTFEL, REL, REPORT, SCAN, SORT, STAT,STOPSCAN, TAB, TITLE, TRACE, TRANS, TSIM, UNLOAD, UNTRACE, UPDATE, USE.

  BEGINSCAN and STOPSCAN are included here, because they can be used in any nodes as part of the associate instruction of SCAN.

- `<structured GLIDER instructions>` are the following (a brief comment about the function is indicated) :

1. ASSEMBLE assembles various messages in a representative message.
2. ASSI assigns values to arrays and multiple nodes.
3. COPY makes copies of a message.
4. CREATE creates a message.
5. DBUPDATE updates a DBASE table.
6. DEASSEMBLE disassembles messages put together by an ASSEMBLE.
7. DOEVENT makes the associated instruction executed only one time each event.
8. DONODE makes the associated instruction executed only one time and only if the event starts in the node.
9. EXTR extracts a message from a list.
10. FILE writes values in a file.
11. GRAPH draws graphics during a run.
12. INTI allows interactive changes of data values.
13. IT schedules a new activation of the node after an Interval Time ahead the actual time.
14. LOAD imports the content of a DBASE IV table to an array.
15. NT schedules a new activation of the node at a future time.
16. OUTG writes variable and array values with titles.
17. PREEMPTION allows a message displace another message from the IL of a R type node and to put itself in that IL.

18. REL makes a message to go out of the IL of a R type node.
19. RELEASE manages the message released in a R type node.
20. REPORT allows to write an output report in a file.
21. SCAN examines a list of messages allowing changes and extractions.
22. SELECT selects messages from the different EL of a D type node.
23. SENDTO sends messages in process to lists.
24. STATE allows to change status variables and control activations of a G type node.
25. STAY indicates the time of staying of a message in the IL of a R type node and gives that value to a variable STAY.
26. SYNCHRONIZE retains a certain number of messages in the EL to release them together at the same time.
27. TSIM sets the duration of the simulation run and gives that value to a variable TSIM.
28. TITLE allows to put titles to the experiments.
29. UNLOAD releases the memory of an array used to import a DBASE IV table.
30. USE allows to define the quantity of resource to be used by a message in a R type node and puts this value in a variable USE.

In the following the syntax and function of each one of these instructions are described. A sign ; is put after each instruction. This separator may be omitted before an END or ELSE separator.

6.1 ASSEMBLE assembles messages in a representative message

Syntax:

```
ASSEMBLE( | ELIM | FIRST | NEW |, |<integer value> | ALL|, |<logic expression>
          | EQUFIELD(<field>) | ) <associated instruction>;
```

The ASSEMBLE instruction is used to join a set of messages that came to a node, maybe at different times, and to represent the set by only one message, named representant. The **representant** may be the first assembled or a new message. **Represented messages** may be deleted or maintained in a list for further disassembling. The associated instruction must contain a SENDTO instruction to dispose of the representant.
The instruction is suited to simulate loading and transportation of things, using the representant as the transporter.

ASSEMBLE can be used in a node with an EL that has the ability to send messages (nodes of G, L, D, and general type with EL).

When this instruction is executed, the EL is scanned from the beginning. Each message (comparing message) is compared with those (compared messages) that follow it. The first comparing message is the first one in the EL. The values of the fields of the comparing message are in the field variables. Those of the compared message are put in the O_<variables>. The comparison consists in the evaluation of the <logic expression> that may include field variables of the comparing messages, O_<variables> of the compared message, and any other type of variables and constants. If the expression is TRUE both, the comparing and compared messages, are **candidates** to be assembled. When all messages are compared, the second message in the EL is taken as the new comparing message, which is compared with those that follow it. The process is repeated so that all the possible comparisons are made. If at any point of the process a number of candidates reach the value expressed by <integer value>, the assemble is successful. Then, the assembled messages are taken out of the list and the <associated instruction> is executed. The whole process is then repeated to see if another successful assemble group of the required size is possible with the remaining messages.
If the parameter ALL is used, there is not limit for the assembled set.

If the `<logical expression>` is reduced to the constant TRUE, the messages are unconditionally assembled up to the specified number (or all the messages if ALL is used).
If there is not successful assemble at all, nothing is done and the `<associated instruction>` is not executed.
If the function `EQUFIELD(<field>)` is used instead of the `<logical condition>`, the messages, which have the same values on that field, are assembled.
In all cases, it is assumed that all the messages in the EL have the fields that are used by the assembling conditions.
Which message is the representant and the fate of the assembled messages depends on the first parameter:

- ELIM indicates that the representant is the first of the assembled group and the remaining of the group are eliminated.
- FIRST indicates that the representant is the first of the assembled group and the rest are put in an assembled list pointed by the representant. This allows the retrieval of them by a DEASSEMBLE instruction. If the representant has yet an assembled list from a previous assemble process, the new assembled messages are added to this list.
- NEW indicates that a new message is created as representant of the assembled group. The values of its fields are copied from the first of the group, but they can be changed in the `<associated instruction>`. This new message points to the assembled list so that it may be later retrieved through a DEASSEMBLE instruction.

The `<associated instruction>`, executed for each successful assemble, may have instructions to change the fields of the representant (assigning values to field variables) and must have a SENDTO instruction to dispose of the representant. Note that this `<associated instruction>` may change the values of variables in the `<logical condition>` and in the `<integer value>` so changing the assemble conditions for the next group or for a new execution of the ASSEMBLE instruction. Assembled messages cannot be changed. If there was some successful assemble the variable SYNC is put to TRUE, otherwise it is put to FALSE.
See the instruction SYNCHRONIZE (6.26).
Examples:

1.
```
Unite_Parts :: ASSEMBLE(ELIM, 3, EQUFIELD(Typed))
               BEGIN Status := Finished, SENDTO(Paint) END;
```

Groups of three messages with equal values in the field `Typed` are sought for. When a group is found, the first message is marked as `Finished` in its field `Status` and sent to the EL of the node Paint. The other two messages of the group are deleted. The node `Unite_Parts` is of the common type.

2.
```
FormGroups (G) :: ASSEMBLE(FIRST, N,(Age > 10) AND (O_Age > 10))
                  BEGIN Leader := TRUE; SENDTO(Tour) END;
                  IF NOT SYNC AND (N > 2) THEN N := N - 1;
```

The process attempts to form groups of size `N` (whose value was defined elsewhere) of messages with the value of the field `Age > 10`. When a group is found, the first one is marked putting to TRUE its field `Leader` and sent to the node `Tour`, keeping a point to the assembled messages. All the possible groups of this size are formed by the ASSEMBLE instruction. When the instruction is finished, as the node is of G type, the process is repeated if the EL is not empty starting again for the first message of the EL. When an attempt to form groups is not successful, the variable SYNC is put to FALSE by the ASSEMBLE instruction and the size of the sought group is reduced by one. The process is repeated until groups of size two are sought for.

6.2 ASSI assigns values to arrays and multiple nodes

Syntax:

```
ASSI<| <variable> | <node> |  /range/ := <nest of constants>;
```

This instruction is used to assign values to arrays of any number of indexes or capacity to multiple nodes.
`<variable>` is a name of an indexed variable;
`<node>` a name of a indexed node;
`/range/` is a range of indexes within the range defined for the node or variable.

ASSI can be used in any type of node.

Examples:

1. Let Alpha be an array of CHAR of one index. The instruction:

   ```
   ASSI Alpha[2,4] := ('a', 'r', 's');
   ```

   assigns the values 'a' to Alpha[2], 'r' to Alpha[3], 's' to Alpha[4].
   If r was declared r: ARRAY[1..3, 2..4, 1..2] OF INTEGER then the instruction:

   ```
   ASSI r[1..2, 2..4, 1..2] := ( ( (3, 3), (5, 6), (8, 9) ),
                                 ( (6, 7), (0, 1), (9, 2) ) )
   ```

   assigns the values:

   ```
   r[1, 2, 1] = 3;   r[1, 2, 2] = 3;
   r[1, 3, 1] = 5;   r[1, 3, 2] = 6;
   r[1, 4, 1] = 8;   r[1, 4, 2] = 9;

   r[2, 2, 1] = 6;   r[2, 2, 2] = 7;
   r[2, 3, 1] = 0;   r[2, 3, 2] = 1;
   r[2, 4, 1] = 9;   r[2, 4, 2] = 2;
   ```

   the instruction does not assign values to the elements:

   ```
   r[3, 2, 1]  r[3, 2, 2]  r[3, 3, 1]  r[3, 3, 2]  r[3, 4, 1]  r[3, 4, 2]
   ```

2. The following instruction assigns the paths of 3 clients in a supermarket:

   ```
   ASSI Paths[1..3, 1..5] :=
                   {Stage 1  Stage 2    Stage 3       Stage 4     Stage 5}
   ( ( { Client 1} 'Entry', 'Fruits',  'Bread',       'Beverage', 'Exit'),
     ( { Client 2} 'Entry', 'Bread' ,  'Canned_Food', 'Exit',     '    '),
     ( { Client 3} 'Entry', 'Pastry',  'Bread,        'Fruits',   'Exit') );
   ```

   The value assigned to Path[2, 3] is 'Canned_Food' as the third stage of the second client. Texts between {} are comments to explain the data.

6.3 COPYMESS makes copies of a message

Syntax:

```
COPYMESS(<integer value>) <associated instruction>;
```

This instruction produces a number of copies of the message that is being processed, equal to the value of `<integer value>` (that must be greater than zero) and for each copy executes the `<associated instruction>`. All fields of the original are copied.

The `<associated instruction>` must have a SENDTO instruction to dispose of the copy. Before it, instructions can be used to change the values of the fields. To do this the names of the `O_<fields>` must be used.

The original message must be processed through instructions outside the `<associated instruction>` before or after the COPYMESS. If it is not sent to a node, it will be lost.

The global control variable ICOPY takes the value of the generated copy. Its value is not recorded in the copied message. If the programmer wants to keep it, a field must be defined in the message for this purpose and the value of ICOPY must be passed to each copy in the `<associated instruction>`.

COPYMESS can be used only in G, I, D nodes and in the RELEASE instruction of an R node.

Examples:

1.
```
COPYMESS(N_Offices) BEGIN Nc := ICOPY + 1; SENDTO(Office[nc]) END;
SENDTO(Office[1])
```
   A number of copies equal to the actual value of `N_Offices` are generated and sent (with an indication of its number plus one) to the nodes `Office[2]`, `Office[3]`, etc. The original is sent to `Office[1]`.

2. The following code may be used to initialize a simulation of a queue system with the system not empty. Six messages are sent at TIME with -10 value to the node `Bridge` with a negative generation time.

```
Entrance(I) Bridge :: IT := EXPO(Tell); SENDTO(Bridge);
FillSys (I) :: GT := -10; TIME := GT; SENDTO(Bridge);
               COPYMESS(5)
               BEGIN GT := -Ant[ICOPY]; TIME := GT; SENDTO(Bridge) END;
Bridge (R)  :: STAY := 1;
---------------------
INIT  ACT(FillSys, 0); ASSI Ant[1..5] := (9, 7, 3, 2, 0);
```

   The node `FillSys` creates a virtual past before TIME = 0 in which it sends six messages to `Bridge`. It finishes its action when he puts TIME to zero (`Ant[5]`).

6.4 CREATE creates a message

Syntax:

```
CREATE( | <message name> | <node> | ) <associated instruction>;
```

This instruction is used to create messages in nodes that are not of I type.

It creates a message with the structure defined in the declaration MESSAGES with the name indicated in `<message name>`.

If `<node>` is used, this may be a name of a I type node and the structure of the created message will be that of the messages that can be created by that I type node. It may also be a general type node for which a message structure has been defined.

After one message is created, the `<associated instruction>` is executed. The `<associated instruction>` must have a SENDTO instruction to send the generated message to a list.

In the `<associated instruction>` values may be given to the fields of the created instruction by means of the names of the `<field>` variables (not the `O_<fields>`). This must be done before the message is sent to the EL of a node.

This instruction can be used in nodes of G, L, D, E, R, A, C, or general type.

Examples:

1.
```
Death (E) Heaven :: N := N + 1;
                    CREATE(Soul)
                    BEGIN Nc := N; SENDTO(Heaven) END;
```

```
------------------
INIT N := 0;
------------------
DECL VAR N: INTEGER;   MESSAGES Soul(Nc: INTEGER);
```

For each message that reach the node `Death` and it is destroyed a new message with the structure defined in the `MESSAGE` declaration `Soul` is created and sent to `Heaven`. The number of creation is counted in the global variable N and stored in the field Nc of the message. As the destroyed and generated messages may have different structures, the process may be useful to simulate the change of the structure of a message.

2.
```
Prod_Order (I) :: IT := EXPO(Tb_Orders); Recep_Time := TIME;
Control (R) File_Order, Ship_Order ::
               RELEASE BEGIN
                        SENDTO(Ship_Order);
                        CREATE(Prod_Order)
                         BEGIN
                          Control_Time := TIME;
                          SENDTO(File_Order)
                         END
                       END;
               STAY := Insp_Time * (1 + UNIF(-0.1, 0.1));
-------------------------------------------
INIT ACT(Prod_Order, 0); TSIM := 100; Tb_Orders := 4; Insp_Time := 2;
DECL
 VAR Insp_Time, Tb_Orders: REAL;
 MESSAGES Prod_Order(Quantity:  INTEGER; Name: STRING[20];
                     Recep_Time, Control_Time, Ship_Time: REAL);
```

Messages (orders) produced by `Prod_Order` are sent to `Control` to be inspected and then sent to `Ship_Order`. For each order sent to `Ship_Order` a message of the same structure is created an sent to `File_Order` with the value of the control time in the field `Control_Time`.

6.5 DBUPDATE updates a DBASE IV table

Syntax:

```
DBUPDATE(<source table>, <destination table>, <list of fields>);
```

This instruction is used to update a DBASE IV table (`<destination table>`) with the values of fields that were previously loaded in an array from another table (`<source table>`). Both tables must have the same structure. Eventually they may be the same table.
Most frequently DBUPDATE is used in this case: fields of a table are imported, by means of a LOAD instruction, into arrays in the running program, these arrays are changed by the simulation process. Then the new values may be returned, by means of the DBUPDATE instruction, to the same table or to another table that may be void or filled with any number of records. The `<list of fields>` indicates what fields have to be updated into the `<destination table>`.
After the execution of the DBUPDATE, the `<destination table>` remains with the same number of records than the `<source table>`. See declaration DBTABLE (2.8) and instruction LOAD (6.14).

UPDATE can be used in any type of node.

Example:

```
LOAD(Student1, Name, Identif, Age, Year, Score);
---------------------------------------
```

```
FOR J := 1 TO N_Student1 DO Student1.Age[J] := Student1.Age[J]+1;
---------------------------------------
DBUPDATE(Student1, Student2, Age, Year, Score);
---------------------------------------
DECL
 DBTABLES (\MainFiles\Student1, \NewFiles\Student2);
```

The fields `Age`, `Year`, `Score` of the DBASE table `Student1` are charged in the running program through the LOAD instruction into the arrays `Student1.Name`, `Student1.Identif`, etc. They may be changed by the program. An example of the change of age is shown. After the changes are made, the values are written by means of the DBUPDATE instruction into the corresponding fields of the table `Student2`. The other fields of the table `Student2` remain unchanged. The variables `N_Student1`, `Student1.Age` and similar variables for the other fields are introduced automatically by the system with the information taken from the declared DBASE IV tables. They may be managed by the programmer as arrays with index corresponding to the record number. Note that the two files may be in different directories. See example in section 2.8.

6.6 DEASSEMBLE disassembles assembled messages

Syntax:

```
DEASSEMBLE <associated instruction>;
```

This instruction is used to extract messages from an assemble list generated by an ASSEMBLE instruction. When it is processed, it is supposed that a message is being processed. Then it is checked if the message has a pointer to an assemble list. If there is not that list, nothing is done. If there is a pointer to a list, the list is scanned from the beginning to the end and, for each message, the `<associated instruction>` is executed. The fields of the message are in the `O_<fields>`. If the `<associated instruction>` executes a SENDTO, the `O_<fields>` are passed to the message which is extracted from the assemble list and it is sent to the indicated node. If no sending is occured, the `O_<fields>`, that may have been changed, are passed to the message and this remains in the assemble list. The representant message (see instruction ASSEMBLE, 6.1) may be disposed before or after the execution of the DEASSEMBLE instruction.

DEASSEMBLE may be used in nodes of G, D, E ,and of general type and in the RELEASE instruction of an R node.

Examples:

1.
```
Travel (R) :: RELEASE
                      DEASSEMBLE SENDTO(Depot[Typ]); SENDTO(Parking);
                      STAY := Travel_Time(Weather);
```
After remaining in the IL of the node `Travel` for a time that is a function of the variable `Weather`, the representants are released and their assembled lists are disassembled. Messages of the assembled list (transported items) are sent to the different nodes `Depot` according the value of the field `Typ`; the representant (transporter) is sent to the node `Parking`.

2.
```
Bus_Stop Exit_Pas (E) ::
                  DEASSEMBLE
                   BEGIN By_Bus := TRUE; N_Arr := N + 1;
                    IF By_Air THEN SENDTO(Airport)
                              ELSE SENDTO(RailRoad)
                   END;
```
Messages (passengers) of the assembled list of the first message (bus) are marked in the field `By_Air` to TRUE and a number of arrival is put to each (N was initialized to 0). Those with the field `By_Air` are sent to the node Airport, the others are sent to `Railroad`. As the node is of E type, the original message is destroyed (See example 13, GLIDER examples book).

6.7 DOEVENT permits instruction execution only once in the event

Syntax:

```
DOEVENT <associated instruction>;
```

DOEVENT is used to avoid repeated executions of an instruction or a set of instructions caused by repeated scanning of the network.
When this instruction is executed, the `<associated instruction>` is only executed in the first scanning of the nodes during an event.
In a multiple node the effect occurs for all the nodes. See instruction DONODE (6.8).
This instruction can be used in any type of node.
Example:

```
    MainGate (G) ::
          DOEVENT N := 0;
           IF Apt AND (IL(EL_Work) < 10)
            THEN BEGIN SENDTO(Work); N := N + 1; STOPSCAN END;
```

Messages in the EL of `MainGate` with the field `A = TRUE` are sent to `Work`, one in each revision of the node (because of the STOPSCAN procedure. see 7.21). This is made so to check the length of the `IL_Work`. The number of messages sent in the event is registered in the variable `N`. The DOEVENT avoids repeated initializations to 0 in the successive sending.

6.8 DONODE prevents instruction execution during network scanning

Syntax:

```
DONODE <associated instruction>;
```

DONODE is used when an instruction or a set of instructions must be executed only in the event that activates the node, where the instruction appears, and only in the first activation of the node. When the node with this instruction is activated as the first in the execution of an event, the `<associated instruction>` is executed. Execution is skipped in the activations produced by scanning of the network. In a multiple node the effect occurs for all the nodes.
See instruction DOEVENT (6.7)
This instruction can be used in any type of node.
Examples:

1.
```
Road_Entr (I)     ::  IT := EXPO(1.5);
Traffic_Contr (G) ::  DONODE BEGIN SENDTO(Road); IT := 20 END;
```

   Messages (cars) leaving the node `Road_Entr` are stopped at `Traffic_Control`. Each 20 time units the accumulated messages are sent to `Road`. It is assumed that the messages are not recycled to `Traffic_Contr` in the event that activates that node.

2.
```
Road_Entr (I)     :: IT := EXPO(1.5);
Traffic_Contr (G) :: STATE BEGIN N := 0; IT := 20 END;
                           DONODE
                           IF N < 3
                           THEN BEGIN N := N + 1; SENDTO(Road) END
                           ELSE STOPSCAN;
```

Messages (cars) leaving the node `Road_Entr` are stopped at `Traffic_Control`. Each 20 time units the node is activated and the instruction associated to STATE is executed. This instruction puts `N` to 0 and schedules a new activation 20 units of time later. It is executed only one time regardless the messages in the EL. The DONODE is then executed for the messages accumulated in the EL of `Traffic_Control`. The associated instruction sends up to three messages to `Road`.

6.9 EXTR extracts a message from a list

Syntax:

```
EXTR(<list name>, <pointer>) <associate instruction>;
```

It is used to extract a message from a list of messages.
`<list name>` indicates the list.
`<pointer>` is a variable declared of POINTER type or may be FIRST or LAST.
The pointed message (or the first or last) is extracted, the values of its fields are passed to the `O_<variables>` and the `<associate instruction>` is executed. After this, the values in the `O_<variables>` are passed to the fields of the messages. The `<associate instruction>` must have a SENDTO instruction to send the message to a list.
This instruction can be used in any type of node and in the associated instruction of a RELEASE.
Example:

```
Process1 (R) Process2, ProcFile2 ::
          RELEASE BEGIN SENDTO(Process2);
          SCAN(IL_ProcFile1, PFile)
                    IF NumP1 = O_NumP1 THEN STOPSCAN;
          EXTR(IL_PartFile, PFile) SENDTO(ProcFile2)
          END;
          STAY := UNIF(32, 36);
```

When a message (object to be processed) leaves the node `Process1` is sent to `Process2`. Then a IL (used to simulate a file) is scanned until a message is found with the same value in the field `NumP1` as the leaving message. The found message, that is now pointed by the POINTER `PFile`, is extracted and sent to the node `ProcFile2`.

6.10 FILE writes values in a text file

Syntax:

```
FILE(<file name>, <list of expression /format/ >);
```

It is used to write a line of data in a text file. The user may also use the instructions of the Pascal language to write files. FILE instruction relieves the user from declare, assign, open, and close the files.
`<file name>` is a valid name of file. If the file exists, it is overwritten by the new data. If it does not, then it is created and opened at the beginning of the simulation and closed at the end of the simulation.
`<list of expression /format/ >` is a list of expressions as the lists in WRITE or WRITELN in Pascal. Expressions must not be broken in different lines.
Generated `text` files have the following structure:

- A heading of strings each with the expressions in the list (one each line). The format is omitted.
- A symbol # 1 to indicate the end of the heading.

- The values recorded, one line each execution of a **FILE** instruction.
- If there are replications then, for the second, third, etc., replication, the list of expressions is omitted, and instead of # 1, the number of the replication: # 2, # 3, etc. precedes the data of each replication. Thus, only one file is made with all the data generated by the replications.

The system introduces an extra blank space between contiguous values. If many **FILE** instructions write in the same file, the order of the lines in the file is the order of the executions.

If the file is to be used with the Graphic option, the list would consist of simple real variables without format.

This instruction can be used in any type of node.

Example:

```
Aa ::  FOR I := 1 TO 3 DO
       FILE(Arxm, I + Azz: 8: 2, Ch1, I: 2, B[I]: 5: 1, Azz);

INIT ACT(Aa, 0);
     Azz := 99; Ch1 := 'J'; ASSI B[1..3] := (19, 20, 21);
```

The file Arxm, if written only for one execution of node Aa, will contain:

```
I + Azz
Ch1
I
B[I]
Azz
\#
100.00 J  1  19.0  9.9000000000E+01
101.00 J  2  20.0  9.9000000000E+01
102.00 J  3  21.0  9.9000000000E+01
```

See example 5 (**GLIDER** examples book) for a file to be used with the Graphic option.

6.11 GRAPH draws graphics during a run

Syntax:

```
GRAPH(<initial time>, <final time>, <background color> / <,title> / ;
     / TIME: <format>, <color> / ; <list of variable descriptors>);
```

It is used to display graphics during the simulation run. When executed for the first time, it constructs the axis, reference lines, numbers on the axis, displays the names given to the variables and scales and marks the initial values of the variables to be graphed. In the following executions it adds a new point to the curves of the variables being graphed and unites the points to the the already displayed curves.

`<initial time>` and `<final time>` are two real numbers indicating the simulation time in which the graphic starts and ends respectively.

`<background color>` is the color that is to be assigned to the background.

`<title>` is a string of characters (without ' or ;). If it is used, it will appear as a title in the graphic. It may be omitted.

`TIME: <format>, <color>` is only used when the independent variable will be the **TIME** of the simulation.

`<format>` is a Pascal format to write the values of the **TIME** along the horizontal axis.

`<color>` is the color of the horizontal axis and the values on it.

`<list of variable descriptors>` is a list of items describing the parameters to represent the variables to be depicted.

If `TIME: <format>, <color>` is omitted, the variable of the first descriptor is taken as independent variable and its values are on the horizontal axis. Variable descriptors are separated by ;

`<list of variable descriptors>` consists of the following elements, separated by commas. They describe the variables to be graphed :

- `<real expression>`, that expresses the value to be graphed; it may be a single variable or constant.
- `<format>` gives the format for the values on the axis; it consists of:
  `<number of places>: <number of decimals>`,
- `<name of the graphed values>`, a name that will appear in the graph. It is a string of characters without blanks, apostrophes or commas.
- `<minimum value>`, a real number with the minimum value expected.
- `<maximum value>`, a real number with the maximum value expected.
- `<color for the curve>`; color must be anyone of the following list:

BLUE	GREEN	CYAN	RED	MAGENTA
BROWN	LIGHTGRAY	DARKGREY	LIGHTBLUE	LIGHTGREEN
LIGHTCYAN	LIGHTRED	LIGHTMAGENTA	YELLOW	WHITE

For the background color, the BLACK may also be used.

Descriptors are separated by ;.

Up to 14 variables may be represented in a graph. GRAPH may be used in any node but it must be executed only one time during each activation. If there are many GRAPH instructions, only one must be executed in a simulation run. When TIME is used as independent variable and a screen is full, the display is stopped half a second, the displayed graph is erased and the display continues in a new screen. The user can use the key PAUSE to stop the graphic and then to re-start the simulation with any key. It is possible to control the horizontal scale by changing the interval of the successive activations of the node that contains the GRAPH instruction. The more frequent the activations the more extended is the graph.

The horizontal axis of the screen is divided in 10 parts. If the total length of the axis is for example 531 pixels (the number of horizontal pixels in the total screen assumed 640 minus 17% for a free space at both sides) each part has 83 pixels. At each new call and execution of the instruction GRAPH the new point is advanced one pixel so it takes 83 time units to advance a part. If successive calls are made each 0.125 time units, the part represents the part of the curve generated in 83 * 0.125 = 6.64 units of simulation time. If the calls are made each 0.0625 time units, the same part of the graph represents the curve generated in only 3.32 units of time. The represented curve appears then stretched twice in the horizontal direction.

This instruction can be used in nodes of any type.

Examples:

1.
```
Aa ::     IT := Dta; Y := SIN(2 * PI / 2.0 * TIME);
Ag ::     IT := Dtg;
             GRAPH(0, 20, BLACK; TIME: 7: 1, WHITE;
               Y: 7: 1, Sinus, -2, 2, LIGHTBLUE);
INIT TSIM := 20; ACT(Aa, 0); ACT(Ag, 0);
         Dtg := 0.00753125;
         Dta := 0.0150625;
DECL VAR Dta, Dtg, Y: REAL;
```

A sinusoidal curve is displayed. Note that each calculated point is displayed two times, because Dtg is half of Dta. As a complete wave takes 2.0 units of time, in a screen as the described above each horizontal division is 53 * 0.00753125 = 0.39915 and in the 10 divisions (3.9915) enter almost 2 waves.

2.
```
Entrance      (I)   ::  IT := EXPO(TBA); ACT(Gra, 0);
Teller        (R)   ::  STAY := EXPO(Tat, 2);
Exit          (E)   ::  ACT(Gra, 0);
Gra           (A)   ::  GRAPH(0, 18000, BLACK; TIME: 6: 0, WHITE;
                        LL(EL_Taq): 6: 0, Queue, 0, 100, GREEN);
```

The program is a simulation of a simple bank teller and a graphic of the queue is made. Each time a client enters or leaves the Teller. the Gra node is activated and the length of the queue (EL of Teller) is put into the graph.

See examples 14, 15, 16, 17, 18, 19 (GLIDER examples book).

6.12 INTI allows interactive change of data

Syntax:

```
INTI <variable>: <format>: <title>;
INTI <list of GFUNCTIONS>;
```

This instruction allows the interactive modification of initialized data.

<variable> is the name of a simple or indexed (with only one index) arithmetical variable.

If the variable has an index the name must be followed by:

[<lowest value> .. <highest value>] that indicates the range of the array to be changed.

<format> consists of two integers <number of places>: <number of decimals>. This define the format of the values presented to the user.

<title> is a string of characters without ; or apostrophes that contains a description of the variable.

This instruction may be used in the INIT and NETWORK but never within GLIDER instructions or I/O instructions.

Examples:

1.
```
INIT
-------------------
Alfa := 1.058;
-------------------
INTI Alfa: 9: 2: Scattering angle ;
```

When this instruction is executed, Alfa must have an assigned value. Let it be 1.058 As it appears in the INIT section (see chapter 3), it will appear before the simulation in the screen:

```
Scattering angle     1.06
C to continue, other to modify
```

If C (or c) is pressed, nothing is done and the execution continues. If another key is pressed, it appears:

```
Scattering angle
```

The user must write the new value and press enter. For example, if 1.2 is written, then the following appears:

```
Scattering angle     1.20
C to continue, other to modify
```

The user may change this value again. When C or c is pressed, the variable Alfa takes the last assigned value.

2.
```
ASSI Temp[1..5] := (899.221, 700.037, 988.111, 655.15, 408.599);
INTI Temp[1..5]: 8: 2: Furnaces Temperatures;
```

When INTI is executed, it appears in the screen:

```
Furnaces Temperatures
     1       2       3       4       5
899.22  700.04  988.11  655.20   408.60

C to continue, other to modify
```

If the user press C or c, it appears:

```
Modify number
```

The user must indicate the number (1 to 5) to be modified. The system asks for the new value. The system rewrites the array modified and asks for new modifications until C or c is pressed.

See examples 1, 6, 8, 9, 10, 13, 14, 16, 18, 19 (GLIDER examples book).

3. The second form of this instruction allows the interactive modification of functions defined by the user as GFUNCTIONS. The mouse must be installed to use this option.
See example 21 (GLIDER examples book).

Example:

```
INIT
---------------------------------------------
 Poli := 0, 20 / 35, 50 / 45, 70 / 60, 45 / 75, 42 / 100, 50;
 Spli1 := 0, 20 / 45, 80 / 100, 30;
 Spli2 := 0, 1 / 5, 10 / 10, 40 / 20, 20 / 40, 30 / 70, 50 / 100, 60;
 Spli3 := 0, 10 / 20, 40 / 30, 70 / 40, 50 / 70, 30 / 80, 50 / 100, 60;
 Esca := 0, 20 / 35, 40 / 45, 60 / 60, 45 / 90, 42 / 100, 50;
 FrecR := 1, 2.5 / 2.3, 4 / 3.2, 3.1;
 FrecA := 2, 6.5 / 5, 2.3 / 7, 1.1 /9, 2.7;
 DiscoN := 2, 6.5 / 5, 2.3 / 7, 1.1 / 9,2. 7;
 DiscoA := Aa, 5 / Bb, 3 / Cc, 4 / Dd, 7;
 DiscoR := 4.1, 2.2 / 5.3, 4.5 / 6.1, 4.4 / 7.7, 5.5;
 -----------------------------------------------
 INTI FrecR, Poli, Esca, Spli1, Spli2, Spli3, FrecA,
      DiscoN, DiscoA, DiscoR;
 -----------------------------------------------
 TYPE Letters = (Aa, Bb, Cc, Dd);
 VAR Let: Letters;
 GFUNCTIONS Poli POLYG (REAL): REAL: 7;
         Spli1 SPLINE(REAL): REAL: 5;
         Spli2 SPLINE(REAL): REAL: 10;
         Spli3 SPLINE(REAL): REAL: 10;
         Esca STAIR (REAL): REAL: 7;
         FrecR FREQ (REAL): REAL: 8;
         FrecA FREQ (INTEGER): REAL: 8;
         DiscoN DISC (INTEGER): REAL: 11;
         DiscoA DISC (Letters): REAL: 11;
         DiscoR DISC (REAL): REAL: 10;
```

When the INTI is executed, a graph of `Poli` appears on the screen. If the user press 2 in the mouse, the system continues displaying the graph of `Spli1`. The XY points indicated in the program are shown as small circles and the curve is plotted. When a graph is shown and the user press 1 in the mouse with the cursor in a point, the following cases may occur:

a) The cursor is on a point of the function. For instance in the `Spli1` the cursor is in X = 45 Y = 80 (or within the circle at this point). Then this point is deleted and a new graphic without it appears.

b) The cursor is very near in X to one of the points but with a different ordinate (Y). For instance in X = 45.003 Y = 63.5 in the example `Spli1`. Then the old near X remains but the old Y is substituted by the new one. In the example the point (45, 80) is replaced by (45, 63.5). The new graph is displayed.

c) The cursor is clearly away from all XY defined points. Then if the X is REAL, a new point is introduced. If the X is scalar or INTEGER the Y is assigned to the nearest X; the Y is changed but no new X is introduced. For instance, in the case of the `DiscoA` function, if the cursor is in a position between Aa and Bb but near of the Bb and the Y is 7.2, then the point (Bb, 3) is replaced by the (Bb, 7.2). The new graph is displayed.

The user can introduce many changes one by one, but the total number of points cannot exceed the maximum declared. In the example, only up to 2 points can be added to `Spli1`. When 2 is pressed to change to the next graphic, the user has the option of storing the new values in a file.

6.13 IT schedules next activation of the node

Syntax:

```
IT := <real expression>;
```

This instruction is used to activate the node in which it appears after an interval time given by the current value of `<real expression>`. When executed, the value of `<real expression>` is assigned to the variable IT and an event is put in the future event list (FEL) with:

- The node number equal to the number in the NETWORK of the node being executed (this node number is not used by the programmer, who refers to the nodes by its names).
- An index equal to the current value of INO (1 for single nodes 1, 2, 3, .. for successive executions of a multiple node).
- Even parameter equal to the current value of EVP.
- Time equal to the current value of TIME + `<real expression>`, that is to say TIME + IT.

When the node is multiple, the instruction may be executed many times. In each execution an event is entered in the FEL. The successive values of the index are 1, 2, 3, .. etc.

See instruction NT (6.15) and procedure ACT (7.1).

The instruction can be used in any node, except of C type.

Examples:

1.
```
Ent (I) Lathe[INO] :: IT := UNIF(1.00, 1.03); SENDTO(Lathe[MIN]);
Lathe (R) [1..4] ::
```

   Messages (parts) are generated each time the node `Ent` is activated. This happens at intervals taken at random from a uniform distribution between 1.00 and 1.03. The messages go to the less crowded of the four nodes `Lathe`.

2.
```
Signal (I) :: IF TIME < 1999
                 THEN IT := 1999 - TIME
                 ELSE WRITELN('This is the last signal');
```

   If the node is activated before the TIME = 1999, then an activation is scheduled for the time 1999 and then the message `This is the last signal` is written.

3.
```
Gate_n1 (G) [1..3] ::
  STATE BEGIN OPEN := NOT OPEN; IT := Interval[INO]; END;
  IF OPEN THEN SENDTO(Deposit[INO]);
-----------------------------------------
INIT
  ASSI Interval[1..3] = (17, 13, 19);
```

Incoming messages are retained each in its gate while the gate is closed and allowed to passed to `Deposit` when the gate is open. The gates are alternatively closed and open. The intervals of each state `OPEN` and `NOT OPEN` are equal for a gate but are different for the different gates. For the first gate they are 17, for the second 13, for the third 19.

See examples 1, 6, 10, 13, 18 (GLIDER examples book).

6.14 LOAD imports a DBASE IV table to an array

Syntax:

```
LOAD (<table>, <list of fields>);
```

This instruction is used to load values in a DBASE IV table into arrays that can be managed for the GLIDER program. The fields of the `<table>` indicated in the `<list of fields>` are stored in a series of arrays named `<table> . <name of field>` that were generated by the system by means of the information of the DBTABLE declaration.
This instruction can be used in nodes of any type.
See declaration DBTABLE (2.8), instruction LOAD (6.14) and examples in section 2.8.

6.15 NT schedules a new activation of the node to a future time

Syntax:

```
NT := <real expression>;
```

This instruction is used to activate the node in which it appears at a future time, given by the current value of `<real expression>`.
When executed, an event is put in the future event list (FEL) with:

- The node number equal to the number of the node being executed.
- An index equal to the current value of INO (1 for single nodes 1, 2, 3 .. for successive executions of a multiple node).
- Even parameter equal to the current value of EVP.
- Time equal to the current value of `<real expression>`.

When the node is multiple, the instruction may be executed many times. In each execution an event is entered in the FEL.
The instruction can be used in any node, except of C type.
See instruction IT (6.13) and procedure ACT (7.1).
Example:

```
Signal (I) :: IF TIME < 1999
              THEN NT := 1999
              ELSE WRITELN ('This is the last signal');
```

If the node is activated before the TIME =1999, then an activation is scheduled for the time 1999 and then the message `This is the last signal` is written.

6.16 OUTG writes variable and array values with titles

Syntax:

```
OUTG <variable>: <file>: <format>: <name>;
```

It is used to write values of a single or indexed variable (with only one index) preceded by a written name. The output is always made for the screen.

- `<variable>` is the name of an INTEGER or REAL variable. If it is indexed, the `<range>` must follow the `<name>`.
- `<file>` is a file variable corresponding to a text file that the user must define, open and close. If `<:file>` is omitted then the output is displayed in the screen only.
- `<format>` is a Pascal format for the data.
- `<name>` is a string of characters without ' or ;. It will appear in the written output.

This instruction can be used in nodes of any type.
See instruction REPORT (6.20).
Example:

```
Lock  (R) :: STAY := 3.2;
-------------------------------------------
Results (A) :: Ttcs := MSTL(EL_Lock);
   OUTG Ttcs: WaitFile: 6: 2: Mean waiting at lock entrance ;
   PAUSE; CLOSE(WaitFile); ENDSIMUL;
-------------------------------------------
INIT
-------------------------------------------
   ACT(Results, 40);
   ASSIGN(WaitFile, 'Waitf.Pas'); REWRITE(WaitFile);
-------------------------------------------
DECL VAR Ttcs: REAL; WaitFile: TEXT;
```

If the mean permanence of messages (ships) queue at the entrance (EL) of the node `Lock` was 7.177, then, when this instruction is executed, the following is written in the screen and in the file `Waitf.Pas`:

```
Mean waiting at lock entrance     7.18
```

The procedure PAUSE stops the process to allow to read the output.

```
OUTG Pass_Number[1..6]: 5: Number of passengers at each bus stop;
```

If the content of the array `Pass_Number` is 2, 4, 12, 7, 5, 16, then the following is written in the screen:

```
Number of passengers at each bus stop
    1    2    3    4    5    6

    2    4   12    7    5   16
```

See examples 1, 6, 18 (GLIDER examples book).

6.17 PREEMPTION allows a message to preempt another that is using the resource

Syntax:

```
PREEMPTION(<logical expression> / ,REMA / ) <associated instruction>;
```

This instruction is used to remove a message that is occupying a resource represented by a R type node of size 1 with a STAY instruction. It may be used only in a R type node of capacity 1, but not in the RELEASE instruction. It is executed when some message is in the EL and it is examined. When executed, the IL is checked. If it is void, the message enters the IL and nothing more is done. If it is not free, then the fields of the message in the IL are passed to the corresponding `O_<fields>` and the `<logical expression>` is evaluated. If it is FALSE, the examined message remains in the EL and nothing more is done. If it is TRUE, then the occupying message is removed from the IL and the new message occupies the resource (enters the IL). When this preemption process happens, the `<associated instruction>` is executed. The preempted message is disposed by a SENDTO instruction that must be in the `<associated instruction>`. This instruction may change the fields of the preempted message before the SENDTO is executed. If the option REMA is present, the preempted message must have a REAL field called REMA that the user must define and initialize to zero. Then the remaining time (time that the preempted message would yet remain in the IL) is stored into the field REMA. When this message regains the resource with its REMA field different from zero, then the time that has to remain in the IL is taken from the REMA instead of using the one indicated by the instruction STAY. This value might have previously been changed in the `<associated instruction>`. The REMA field is automatically put to zero when the message regains the resource.

The `<logical expression>` may have global variables, local variables, field variables of the examined messages and `O_<fields>` of the occupant message. Note that if the removed message has a field REMA and, instead of regaining the node from which it was removed, it enters another R node, the time to remain there would be taken from the REMA and not from the STAY of this node, unless the REMA were explicitly put to zero.

PREEMPTION can only be used in the non RELEASE part of the code of an R type node of capacity 1 and before the STAY instruction.

Example:

```
Messages (patients) that arrive at Hospital have a Disease
 type and a Status of seriousness. The Critical are put at the
 beginning of the queue for Diagnostic. In Diagnostic a Critical
 can remove a non Critical from attention. The removed is sent
 at the beginning of the queue. Its future time of attention is the
 remaining time increased 20%. After the Diagnostic the messages
 are sent to one node of the multiple node Treatment according to
 their Disease.
 Note the declaration and initialization of the field REMA.
 One defect of the program: if one Critical is in the IL and two
 other Critical arrive, then they will be put in the queue (EL) in
 reverse order. See instruction SCAN (6.21) for a solution.
 Note that it is also unrealistic that the variables Status
 and Disease were uncorrelated+. See RAND function (8.25) for a solution.

 NETWORK
   --------------------------------------------------
   Patient   (I) :: IT := EXPO(5.5); Disease := Fdisease;
                    Status := Fstatus; REMA := 0;

   Hospital  (G) :: IF Status = Critical THEN SENDTO(FIRST, Diagnostic)
                                      ELSE SENDTO(Diagnostic);
```

```
Diagnostic (R) Diagnostic, Treatment[INO] ::

    RELEASE BEGIN Disn := Ord(Disease); SENDTO(Treatment[Dis]) END;
    PREEMPTION((Status = Critical) AND (O_Status < Critical), REMA)
       BEGIN REMA := REMA * 1.2; SENDTO(FIRST, Diagnostic) END;
    STAY := GAMMA(MDiag, DDiag);
-----------------------------------------------------
INIT
Fdisease := Hurt,15 / Infectous,20 / Shock,25 / Hearth,30 / Internal,20;
Fstatus := Non_Serious, 30 / Medium, 40 / Serious, 30 / Critical, 20;
-----------------------------------------------------
DECL TYPE St = (Non_Serious, Medium, Serious,  Critical);
          Dis = (Hurt, Infectious, Shock, Heart, Internal);
     MESSAGES Patient(Disease: Dis; Status: St; REMA: REAL);
     GFUNCTIONS
      Fdisease FREQ (Dis): REAL: 5;
      Fstatus  FREQ (St): REAL: 4;
```

See example 8 (GLIDER examples book).

6.18 REL takes a message out of the IL of an R type node

Syntax:

```
REL(<internal list of node R>, <logical expression>) <associated instruction>;
```

This instruction is used to extract a message from the IL of a R type node.

When executed, a search starts at the IL of the node, looking for the first message for which `<logical expression>` is true. If there is not such a message nothing is done. If it is found, it is extracted from the IL and the `<associated instruction>` is executed. If a time of exit is scheduled for a normal release of the resource, the corresponding event of the FEL is removed to eliminate the scheduling.

If there is extraction of message, the boolean variable **EXTREL** is put to TRUE, otherwise, it remains FALSE. The values in the fields of the extracted messages are passed to the corresponding `O_<variables>`.

The `<associated instruction>` must have a SENDTO instruction to dispose of the extracted instruction. Before the SENDTO is executed, instructions can be used to change the values of the fields. To do this, the names of the `O_<fields>` must be used.

This instruction can be used in nodes of any type.

Example:

```
NETWORK
 Entry (I) :: IT := EXPO(3); Urgency := Round(TRIA(0, 5,7));
              WRITELN('Enter # ', Number: 3,' Urgency ', Urgency: 2);
              PAUSE;

 Ward  (R) :: USE := 1;

 Call  (A) Physician ::
              REL(IL_Ward, O_Urgency + 0.2 * (TIME - O_Et) > 5)
                SENDTO(Physician);
              IF NOT EXTREL AND (U_Ward > 0) AND (LL(EL_Physician) < 4)
               THEN EXTR(IL_Ward,FIRST) SENDTO(Physician);
              IT := 10;
```

```
    Physician (R) :: STAY := TRIA(4, 6, 9);

    Exit (E) :: WRITELN('Exit  # ', Number: 3, ' Urgency ', Urgency: 2);
                PAUSE;


   INIT TSIM := 200; Urgency := 3; ACT(Entry, 0);  ACT(Call, 0);


   DECL MESSAGES Entry(Urgency: INTEGER);
```

Messages (patients) are admitted into the `Ward` if there is room enough. Each 10 minutes `Call` is activated. This extracts the first message for which the value of the field `Urgency` plus a multiple of the waiting time is greater than 5. The extracted message is sent to the node `Physician`. If no message fulfills this condition the first one in the queue is extracted and sent.

See example 10 (GLIDER examples book).

6.19 RELEASE manages the message released in a R type node

Syntax:

```
    RELEASE / ALL / <associated instruction>;
```

This instruction is used to manage the messages that abandon the IL of a R type node, when the scheduled time is over. It is used when the default disposition of the message is not desired. The user may program a different procedure in the `<associated instruction>`. This may includes changes in the fields, conditional sending to different nodes, and a NOTFREE instruction to inhibit the freeing of the used resource. It may also contain common GLIDER instructions and procedures, but their associated instructions must not contain GLIDER instructions with associated instructions.

The instruction, when it is used, must be the first in the code of the node. It is executed only if the node is activated by an event. The execution is not repeated during the scanning of the network.

When the instruction is executed, a scanning of the IL starts, looking for a message with the exit time of the IL (that is recorded in the field XT). If such a message is not found, an error condition occurs. If it is found, its fields are passed to the corresponding field variables and the `<associated instruction>` is executed. This instruction must execute a SENDTO instruction to dispose of the message.

In the process (unless NOTFREE is used) a quantity of resource is freed that is equal to the value of the USE parameter of the message. So, the variable `F_<node>` is increased by USE and `U_<node>` is decreased by USE. If the capacity of the resource is the constant MAXREAL, the resource is considered of infinite capacity and no change in those variables is made.

If the ALL parameter is used, the search for other messages with the same XT continues until all of them are processed.

This instruction can be used in R type nodes.

Example:

```
        Physician (R) ::
         RELEASE BEGIN Revised := TRUE; SENDTO(Treatment[Disease]) END;
         STAY:=TRIA(4, 6, 9);
```

Messages (patients) that leave the node `Physician` after completing its scheduled time of attention (given at the entrance by the instruction STAY) are tagged putting TRUE its field `Revised`. Then they are sent to one node of the multiple node `Treatment` according the value of their field `Disease`.

See examples 9 and 13 (GLIDER examples book).

6.20 REPORT allows to write an output report in files

Syntax:

```
REPORT <code> ENDREPORT;
```

This instruction is used to show and file user's built reports of partial or final results of a simulation run. All the instructions between the delimiters REPORT and ENDREPORT are executed. Pascal instructions (WRITE, WRITELN, etc.) and common GLIDER instructions (in particular OUTG) can be included.

Before writing, the GLIDER system asks for the name of the file to write in. CON is used to write in the screen. LPT1 to the line printer. Instructions that write some output will do that in the files indicated by the user just before the writing. After finishing, the system asks if another writing is required. The same writing may be repeated in different files (for instance first in the screen and after in a disk or printer).

If it is desired that some included writing instructions use other file than that indicated for the whole report, the file variable OUTF must be used in those instructions. The user have to assign a file name to this variable in the INIT section (see chapter 3).

This instruction can be used in nodes of any type.

See example 18 (GLIDER examples book).

6.21 SCAN scans and processes a list of messages

Syntax:

```
SCAN(<list name>, / <pointer> /) <associated instruction>;
```

This instruction is used to examine lists of messages to point, modify and extract messages.

`<list name>` is a name of an IL or EL. It may be of any node.

`<pointer>` is a variable declared as POINTER.

When executed, it scans the list indicated. For each scanned message the fields are passed to the `O_<fields>` and the `<associated instruction>` is executed. Then the `O_<fields>` are passed to the fields of the message. The scanning may be suspended when a STOPSCAN instruction is executed. Otherwise, it continues until the end of the list is reached. If a `<pointer>` is included, it will contain a point to the last processed message. A system generated pointer points to the previous message in the list. This allows the user to manage the pointed message.

If during the execution of the `<associated instruction>` a SENDTO instruction is executed the message in process is extracted and sent to the place indicated by the SENDTO. The scanning continues.

This instruction can be used in nodes of any type.

Examples:

1.
```
   Heating (A) Process :: IT := 10;
     SCAN(EL_Line)
      BEGIN
       O_Temp := O_Temp + UNIF(1.2, 1.3);
       IF O_Temp > 15.5 THEN SENDTO(Process);
      END;
```

   When `Heating` is activated (each 10 units of time) the EL of a node `Line` is scanned. A quantity (random value between 1.2 and 1.3) is added to the value of the field `Temp` of the message. If one message surpasses the value 15.3, it is extracted and sent to `Process`.

2.
```
Depot (R) ::

TakeHighest (A) Group ::
```

```
H := 0; PHigh := NIL;
SCAN(IL_Depot, Pt)
  IF O_Height > H THEN
  BEGIN PHigh := Pt; H := O_Height END;
SCAN(EL_Group, Pg) IF O_X = 3 THEN STOPSCAN;
IF PHigh <> NIL THEN EXTR(IL_Depot, PHigh) SENDTO(Pg, A, Group);
```

When TakeHighest is activated, the message of the IL of Depot with highest value in the field Height is found and sent to the EL of Group and put after the first message with field X equal to 3.

6.22 SELECT selects messages from ELs of a D type node

Syntax:

```
SELECT(| <list of nodes> | <multiple node> <range> |, <integer value>)
       <associated instruction>;
```

It is used to select messages from ELs of a type D node corresponding to its predecessors.

`<list of nodes>` is a list of predecessor nodes of the D type node.

`<multiple node > <range>` is the name of an antecessor multiple node with a range within the range of the node indexes.

`<integer value>` is an integer positive constant or an integer type variable with positive values.

When the instruction is executed, the ELs of the node are examined in the order in which they appear in the list. If the predecessor is a multiple node, the successive indexed nodes indicated in the `<range>` are examined. A quantity of messages equal to `<integer value>` (or less if there are not enough in the EL) are extracted. For each extracted message the fields are passed to the `O_<fields>` and the `<associated instruction>` is executed. This must have a SENDTO instruction to send the message to any list. If the SENDTO is not executed, the message remains in its EL.

The selected maximum can be known and modified for the next selection using the MAXSEL system variable. If the predecessor is a multiple node, the index of the EL actually examined is indicated in INO. MAXSEL and INO may be used in the `<associated instruction>`

The instruction only can be used in type D nodes and if and only if there are more than one predecessor or the predecessor is an indexed node.

Examples:

1.
```
Proda (I) Sel ::  IT := Tproa + UNIF(0.2, 0.4);
Prodb (I) Sel ::  IT := Tprob + UNIF(0.1, 0.5);
Prodc (I) Sel ::  IT := Tproc + UNIF(0.2, 0.6);
Sel (D) Assemb ::
         I := MINI(LL(EL_Sel(Proda)),
         MINI(LL(EL_Sel(Prodb)), LL(EL_Sel(Prodc))));
         IF I >= 2 THEN SELECT(Proda, Prodb, Prodc:  I) SENDTO(Assemb);
Assemb (E) ::
```

   Messages (parts to be assembled) are produced at `Proda`, `Prodb` and `Prodc` and sent to 3 queues that are the ELs of Sel. When the minimum quantity in the queues is 2, then 2 messages are selected from each queue and sent to `Assemb`.

2.
```
Parts of five different types are produced at Pr with different
processing times. They are sent to a Selection subsystem.
When the production of each kind accumulates the required quantities,
given by the function FQuantReq, (for some types more than that may
be produced) the required quantities are selected and sent to Assembly,
This simulation is a part of a bigger model. Its purpose is to see
```

```
how many assembles can be made and how many unmatched parts remain.

NETWORK
 Pr (I) [1..5] Selection ::
    IT := UNIF(Tmin[INO], Tmax[INO]); PartType := INO; SENDTO(Selection);

 Selection (D) :: {Evaluate extraction condition: }   Extract := TRUE;
    FOR INO := 1 TO 5 DO IF LL(EL_Selection(PR[INO])) < FQuantReq(INO)
                             THEN Extract := FALSE;
                             MAXSEL := FQuantReq(1);
                             IF Extract THEN
                             SELECT(PR[1..5]: MAXSEL)
                   BEGIN MAXSEL := FQuantReq(INO); SENDTO(Assembly);
                   WRITELN(MAXSEL,' Selected Type ',INO {or PartType} );
                  END;

 Assembly (E) :: WRITELN(TIME: 8: 3, ' ***Out From PartType ', PartType);
                PAUSE;

INIT
 TSIM := 50; FOR INO := 1 TO 5 DO ACT(PR[INO], UNIF(Tmin[INO], Tmax[INO]));
 TITLE := 'Test Select I';
 ASSI Tmin[1..5] := (2.8, 3.5, 3.6, 3.9, 1.5);
 ASSI Tmax[1..5] := (4.8, 7.2, 7.6, 8.4, 3.5);
 FQuantReq := 1, 2 / 2, 1 / 3, 1 / 4, 1 / 5, 2;

DECL  VAR Extract: BOOLEAN; Tmin, Tmax: ARRAY[1..5] OF REAL;
            MESSAGES Pr(PartType: INTEGER);
            GFUNCTIONS FQuantReq DISC (INTEGER): INTEGER: 5;
            STATISTICS ALLNODES;
END.
```

6.23 SENDTO sends messages in process to lists

Syntax:

```
SENDTO(< | <pointer>, | A | B |>| FIRST | LAST |>, <destination list(s)>);
```

This instruction is used to send messages to one or more lists of messages.
<destination list> is a list of names of nodes or lists (they may be of the form EL_<node> or IL_<node>)
Message **in process** is sent to all these destinations (if the list has more than one element, copies are sent at each destination). If one destination is indicated by a node, it is assumed that it must be sent to the EL of the node.
The place in the destination list in which the message is put, depends on the first parameter:

- LAST the message will be appended at the end of the list.
- FIRST the message will be put at the beginning of the list.
- <pointer>, B the message is put before the pointed message.
- <pointer>, A the message is put after the pointed message.
- If the first parameter is omitted, LAST is assumed.

When SENDTO is executed, the values in the field variables are passed to the fields of the message and this is put in the indicated place.

If this instruction is within an `<associated instruction>` of a GLIDER instruction, then the values in the `O_<fields>` are passed to the message (except for the CREATE in which the field variables are used). In the case the SENDTO is in an `<associated instruction>`, the sent message is:

- In ASSEMBLE: the representant of the assembled messages.
- In COPYMESS: each copy made of the original message.
- In CREATE: the created message.
- In DEASSEMBLE: each message that is taken off the assembled list.
- In EXTR: the extracted message.
- In PREEMPTION: the removed (preempted) message.
- In REL: the extracted from the IL.
- In RELEASE: the extracted from the IL.
- In SCAN: the actually examined message.
- In SELECT: each selected message.
- In SYNCHRONIZE: each synchronized message.

SENDTO can be used in type G, L, I, D, E, A, and general type nodes, and in the `<associated instruction>` of the GLIDER instructions indicated above, such as in the RELEASE instruction.

The nodes referred to in the SENDTO instruction must be explicitly indicated in the list of successors, unless there are only one successor and it is implicitly defined as such by being the following in the program.

If the successor is a multiple node, the user have three alternatives to send a message to its ELs:

- To indicate the desired index in the node specified in the SENDTO.
- To use the reserved word MIN as index. In this case, the message is sent to the first node for which the sum: `<Length of the EL> + <quantity of used resource>` is the minimum. That means the less engaged node of the array.
- To use the reserved word FREE as index. In this case, the message is sent to the EL of the first node that has the EL empty. If all the ELs have messages, the message is not sent. This option only may be used in the code of a type G node. Not sent messages remain in the EL of the node.

Examples:

1.
```
Mail (D) :: SENDTO(Director, Secretary, File);
```
   Copies of the message received at `Mail` are sent to the nodes `Director`, `Secretary`, and `File`.

2.
```
Discrim (G) Queue :: IF Class = Friend THEN SENDTO(FIRST, Queue)
                                       ELSE SENDTO(LAST, Queue);
```

3.
```
Parkman (G) Parkinglot[INO] :: SENDTO(Parkinglot[MIN]);
Parkinglot (R) Street :: STAY := GAMMA(Tpark, DevTpark);
```

4.
```
Ii               :: IT := 1; Pr := UNIFI(1, 4);

OrdQueue (G) LINE :: IF LineGate = Closed
                     THEN
                           BEGIN
                           SCAN(EL_Line, Poin) IF Pr > O_Pr
                           THEN STOPSCAN;
                           IF Poin < > NIL THEN SENDTO(Poin, B, Line)
                           ELSE SENDTO(LAST,Line);
                           END
                           ELSE SENDTO(Line);

Line (G)          :: STATE BEGIN
                           IF LineGate = Closed THEN LineGate := Open
                           ELSE LineGate := Closed;
                           IT := 8
                           END;
                           IF LineGate = Open THEN SENDTO(Process);

INIT TSIM := 100; ACT(Ii, 0); LineGate := Open; ACT(Line, 8);

DECL TYPE Opcl = (Closed, Open);
           VAR LineGate: Opcl; Poin: POINTER;
           MESSAGES Ii(Pr: INTEGER);

END.
```

Messages, with priorities from 1 to 4 in the field `Pr`, arrive at `OrdQueue` and they are passed to `Line` if the variable `LineGate` has the value `Open`. In this case, they continue to `Process`. `LineGate` is `Open` and `Closed` each 8 time units. If `LineGate` is closed, then when a message reaches `OrdQueue`, a scan process of the EL of `Line` starts. If a message in the EL is found with priority less than the incoming message, then this is put before of that in the EL. So, the EL is always ordered in decreasing order of priority. In this system the entities that arrive when the gate is open are allowed to pass, when they have to wait, they are sorted according to their priority. This type of sorting may be more easily done using a type L node.

See examples 2, 6, 7, 8, 10, 11, 12, 13, 18 (GLIDER examples book).

6.24 STATE controls activations and state of G type node

Syntax:

```
STATE <associated instruction>;
```

It is used to change the state in a type G node. The state is specified by the user using variables of any type, and giving them values by means of any code.

It may be used only in type G nodes. If it is used must be the first instruction in the code.

The `<associated instruction>` is only executed if the event refers to the node. No execution is made during the scanning of the network.

The instruction is executed even if the EL of the node is empty.

If the node is a multiple one, the instruction is activated and the value of the index IEV of the event element is passed to the variable INO. Thus, the programmer can control the execution for the different nodes.

The `<associated instruction>` cannot have SENDTO instructions but can use Pascal and common GLIDER instructions.

Examples:

1.
```
Portal (G) :: STATE IF p = open
                        THEN BEGIN p := closed; IT := 10 END
                        ELSE BEGIN p := open;   IT := 20 END;
                    IF p = open THEN SENDTO(Castle) ELSE STOPSCAN;
```

Messages (Knights) that arrived at Portal only pass to Castle if the value of p is open. Due to the STOPSCAN only one message of the EL (the first one) is considered if p is closed. The STATE instruction works in an independent way. The variable p is put closed by 10 units of time, and open by 20 units of time alternatively. To start this process the node Portal must be firstly activated from other part of the program.

The part of the code outside the <associated instruction> of the STATE (the second IF instruction) is executed each time the node Portal is activated during the scanning of the network if the EL is not empty.

2.
```
RandomSend (G) ::
                 STATE BEGIN j := 1; u := UNIF(0, 1);
                         WHILE tran[i,j] < u DO i := i + 1; j := i;
                         IT := perm[i]
                         END;
                 SENDTO(Destination[j]);
```

tran is a matrix with accumulated values of a probability distribution in each column. It can, for instance, be defined by:

```
ASSI tran[1..3, 1..3] :=  (( 0.1, 0.4, 0.2),
                            ( 0.7, 0.9, 0.6),
                            ( 1.0, 1.0, 1.0));
```

This corresponds to the transition probability matrix T:

$$T = \begin{matrix} 0.1 & 0.4 & 0.2 \\ 0.6 & 0.5 & 0.4 \\ 0.3 & 0.1 & 0.4 \end{matrix}$$

If it is $j = 2$, the probability of transition to 1 is 0.4, to 2 is 0.5, to 3 is 0.1. See that:
Tran[1, 2] := 0.4; Tran[2, 2] := 0.4 + 0.5; Tran[3, 2] := 0.4 + 0.5 + 0.1; When a message in the EL of the node is processed and a initial value was assigned to j (that is the state of the RandomSend node), then a random value between 0 and 1 is assigned to u. The WHILE cycle searches from top to bottom in the column j, for the last row for which u < Tran[i, j]. So, the probability of a certain final value of i is the value of T[i,j]. That i value is assigned to j. The next change of state is made after a time given in the Perm array. Messages arriving during this time interval are sent to Destination[j]. So, the destination varies according to states that change at random with the transition probabilities given by the matrix T.

6.25 STAY schedules exit time from a R type node

Syntax:

```
STAY := <real expression>;
```

It is used to assign value to the time that a processed message will remains in the IL of a R type node.

It is only used in a R type node.

When executed a message enters the IL, an event is scheduled in the future to extract this message. More exactly, an event element is introduced in the FEL that refers to the R type node, to be executed at a time

TIME + STAY (with the assigned value for the STAY). The index of the event is taken from the variable INO and the event parameter will be the actual value of the variable IEV. If the node contains a PREEMPTION instruction with a REMA parameter, the message must have a REMA field. The system checks the value of this field. If its value is not equal to zero, the time of execution of the event will be TIME + REMA. See PREEMPTION instruction (6.17).

See examples 1, 2, 3, 4, 5, 6, 7, 8 (with PREEMPTION and REMA), 9, 11, 12, 13, 17, 18 (GLIDER examples book).

6.26 SYNCHRONIZE retains messages in the EL to release them together

Syntax:

```
SYNCHRONIZE(<integer value> | ALL | , | <logic expression> |
             EQUFIELD(<field>)  | ) <associated instruction>;
```

SYNCHRONIZE is used to retain messages that come to a node, maybe at different times, and to release together a set of them at the same time (synchronized) when certain conditions are met. The associated instruction must contain a SENDTO instruction to send the synchronized set.

The instruction is suited to simulate synchronization processes in assembling lines, and gathering items for batch processing.

The instruction must be used in a node with an EL that has the ability to send messages (nodes of G, L, D, and general type with EL).

When this instruction is executed the EL is scanned from the beginning. Each message (comparing message) is compared with those (compared messages) that follow it. The first comparing message is the first one in the EL. The values of the fields of the comparing message are in the field variables. Those of the compared message are put in the O_<variables>. The comparison consists in the evaluation of the <logic expression>, that may include field variables of the comparing messages, O_<variables> of the compared message and any other type of variables and constants. If the expression is TRUE, both, the comparing and compared messages, are **candidates** to be synchronized. When all messages are compared, the second message in the EL is taken as the new comparing message which is compared with those that follow it. The process is repeated so that all the possible comparisons are made. If at any point of the process a number of candidates reach the value expressed by <integer value>, then synchronization is successful. Then, the synchronized messages are taken out of the list one by one and for each synchronized message the <associated instruction> is executed. The whole process is then repeated to see if another successful synchronization group of the required size is possible with the remaining messages.

If the parameter ALL is used, there is not limit for the synchronized set.

If the <logical expression> is reduced to the constant TRUE, the messages are inconditionally synchronized up to the specified number (or all the messages if ALL is used).

If there is not successful synchronization at all, nothing is done and the <associated instruction> is not executed.

If the function EQUFIELD(<field>) is used instead of the <logical condition>, messages with the same values on that field are synchronized.

In all cases, it is assumed that all the messages in the EL have the fields that are used by the synchronizing conditions.

The <associated instruction> executed for each successful synchronization, may have instructions to change the fields of the messages (assigning values to field variables) and must have a SENDTO instruction to dispose of the message that represents them. Note that this <associated instruction> may change the values of variables in the <logical condition> and in the <integer value>, so changing the synchronization conditions for the next group or for a new execution of the SYNCHRONIZE instruction. If there was some successful synchronization, the variable SYNC is put to TRUE, otherwise it is put to FALSE.

See the instruction ASSEMBLE (6.1). See example 11 (GLIDER examples book).

6.27 TITLE puts title to an experiment

Syntax:

```
TITLE := <string constant>;
```

It is used to put a title heading to the tables of the standard statistics.

`<string constant>` is a string constant that may have up to 80 characters.

The variable TITLE takes the value of this constant and can be used in any writing instruction. When the standard statistics are written, this title is written at the beginning. If no value is assigned to TITLE, its default value is 'BASIC EXPERIMENT' .

This instruction can be used in nodes of any type.

Example:

```
T := -272.5;
TITLE := 'Experiment with Low Temperature: T < -270 ';
```

6.28 TSIM sets duration to the simulation run

Syntax:

```
TSIM := <real expression>;
```

It is used to set the simulation duration. The GLIDER system variable TSIM is set to the value of `<real variable>` and the end of the simulation is scheduled for a future time equal to TSIM. The end of the simulation can also be done using the ENDSIMUL instruction. If none of these instructions is found, an error condition arises. If many ENDSIMUL instructions are present, the first one executed will end the simulation. If many TSIM instructions are executed, the last executed defines the simulation time.

This instruction can be used in nodes of any type and in the INIT section.

6.29 UNLOAD releases memory used to import DBASE IV table

Syntax:

```
UNLOAD( <DBASE IV table name>, <list of table fields>);
```

It is used to eliminate from the dynamic memory the data (array) corresponding to a DBASE IV table. The freed space may be used by the running program.

`<DBASE IV table name>` is the name of a table declared in DBTABLES declaration.

`<list of table fields>` is a list of fields of the table that correspond to the arrays to be deleted.

This instruction can be used in nodes of any type.

See examples in section 2.8.

6.30 USE defines the quantity of resource to be used

Syntax:

```
 USE := <real expression>);
```

It is used to define the quantity of resource to be used for a message.

The instruction assigns the value of `<real expression>` to the field USE of the actually processed message. When the message is processed by a R type node, the message will try to use that quantity of resource. The value of the USE field can be changed any number of times during a simulation run.

This instruction can be used in nodes of any type.

Example:

```
Warehouse (R) :: IF BoxSize = Large THEN USE := 4 ELSE USE := 1.5;
```

Chapter 7

PROCEDURES

GLIDER procedures are:

1. ACT schedules future activation of a node.
2. BEGINSCAN repeats the scanning of an EL.
3. BLOCK blocks future release events of node.
4. CLRSTAT initializes the gathering of statistics.
5. DEACT deactivates a node.
6. DEBLOCK suppresses blocking of future release events of node.
7. ENDSIMUL ends the simulation at the end of the actual event.
8. EXTFEL extracts an event from the FEL.
9. FIFO puts the message at the end of the IL of a L type node.
10. FREE frees a resource.
11. LIFO puts the message at the beginning of the IL of a L type node.
12. MENU calls the Run Interactive Menu.
13. METHOD defines the method of integration in a C type node.
14. NOTFREE inhibits the release of a resource.
15. ORDER puts the message in the IL of a L type node in a given order.
16. PAUSE stops the execution.
17. PUTFEL adds an event to the FEL.
18. RETARD produces a delayed function from a given function of time.
19. SORT sorts a list of messages.
20. STAT displays the statistics.
21. STOPSCAN stops the scan of a list of messages.
22. TAB adds a value to a frequency table.
23. TRACE starts the tracing.

24. TRANS sends an element from a list to another list.
25. UNTRACE stops the tracing.
26. UPDATE updates field variables or fields of a message.

This chapter notation is the same of the chapter 6.

7.1 ACT schedules a future activation of a node

Syntax:

```
ACT(<node name>, <real expression>);
```

It is used to schedule a node activation in a future time.
`<node name>` is the name of the node to be activated.
`<real expression>` is a real expression whose value indicates the time from now until the future activation.
When this instruction is executed, a future event is put in the FEL. The event node is the indicated by `<node name>` . The time of the event is `TIME+<real expression>`. Here TIME is the system variable that contains the value of the actual simulation time.
The index and the event parameter of the event are the actual values of the variables INO and IEV respectively.
This procedure can be used in nodes of any type.
Examples:

```
ACT(SubwayDepart, 0);  ACT(Controller, 7);
FOR i := 1 TO 8 DO ACT(Arrivals[i], EXPO(TMBarr[i]);
```

The components of the multiple node `Arrivals` are activated for future times taken from an exponential distribution with mean values depending on the node.

7.2 BEGINSCAN repeats the scanning of a list

Syntax:

```
BEGINSCAN;
```

It is used to re-start the scanning of the list being processed, usually after the list has been altered. When this instruction is executed the scanning of the list starts again from the beginning.
The user should be careful in using this procedure, lest a loop may happen. If the BEGINSCAN is executed continually during the scan of a non vanishing list, a loop will result.
This procedure can be used in nodes of G type and in the associated instruction of SCAN .
Example:

```
Ent (I) :: COPYMESS(10) BEGIN Age := UNIFI(1,100); SENDTO(CheckAge) END;
              SENDTO(CheckAge);
CheckAge (G) Egress :: A := 0;
              SCAN(EL_CheckAge,Points)
              IF O_Age > A THEN BEGIN PointOld := Points; A := O_Age END;
              TRANS(EL_CheckAge, PointOld, A, EL_Egress, LAST);
              BEGINSCAN;
------------------------------------------------------------
INIT  ACT(Ent, 0);
```

```
------------------------------------------------------------
DECL VAR A: INTEGER;
         Points, PointOld: POINTER;
         MESSAGES  Ent(Age: INTEGER);
```

Messages (persons) that arrive to `CheckAge` are sent to `Egress` by decreasing order of the field value `Age`.

7.3 BLOCK blocks future events of release at a type R node

Syntax:

```
BLOCK(<node name>);
```

It stops the creation of events in the node. Those events that would activate the node and are already in the FEL are put in **suspended state** (their SU parameters are set TRUE) so that they are not executed although their execution time are less than TIME. If the node to block is already blocked nothing is done. When the event state is changed again to **active state** (see DEBLOCK, 7.6) the delayed events are executed at the actual time.

This procedure can be used in nodes of any type.

Example:

```
Arrivals (I) :: IT := Farr(TIME);
                      IF LL(EL_Entrance) > 8 THEN BLOCK(Arrivals);

Entrance :: RELEASE BEGIN SENDTO(Parking)
                        LL(EL_Reception) < 5 THEN DEBLOCK(Arrivals);
                    END;
            STAY := TicketTime;
```

Messages (cars) are generated at irregular times given by the function `Farr(TIME)`. When a queue of 9 is formed at the parking `Entrance` the generation is stopped (the scheduled next arrival is suspended) until the queue is less than 5. See DEBLOCK procedure (7.6) for another example.

7.4 CLRSTAT re-starts the gathering of statistics

Syntax:

```
CLRSTAT;
```

It is used to reset to zero the variables in which the statistics are accumulated. Thus the statistics are counted since the last time in which a procedure of this type is executed. That time is recorded in the standard statistics.

This procedure can be used in nodes of any type.

Examples:

```
 PassTrans (A) ::
    IT := 1000;
    IF ABS(Qma - MEDL(EL_Tell)) / (0.5 * (Qma + MEDL(EL_Tell))) < 0.03
              THEN BEGIN Stable := TRUE; CLRSTAT; BLOCK(PassTrans)
                   END
              ELSE BEGIN  Qma := MEDL(EL_Tell); CLRSTAT;
                   END;
-------------------------------------------------
INIT TSIM := 24000; ACT(PassTrans, 200); Qma := 0; Stable := FALSE;
```

At intervals of 1000 time units, the queue `EL_Tell` is examined. The previous mean length `Qma` is compared with the mean length in the last 1000 time units. This is given by the GLIDER function MEDL. If the difference, compared with the mean of the two values, is less than 3%, the queue is considered stable, a new count of statistics begin, and the node is blocked. Otherwise, the last value of the mean is saved in `Qma`, and a new statistical count is initialized for the next 1000 time units. Statistics at the end of the run are counted since the beginning of the stable situation. This procedure tries to avoid the influence on statistics of the transient produced when starting with a void queue.

7.5 DEACT deactivates a node

Syntax:

```
DEACT(<node name>);
```

It is used to deactivate a node, i.e. to suppress the future event that would turn out it active. A typical use is to stop the integration in a C type of node that solve differential equations.
This procedure can be used in nodes of any type.
Examples:

```
Growth (c) :: Diameter' := KGrowth * (1 - Diameter / DMax) * Diameter;
                        If Diameter > 50 then DEACT(Growth);
                        CREATE(CutMessage) SENDTO(Saw);
```

While the node `Growth` is active, it computes the change in time according to the shown differential equation (logistic growth). When the variable `Diameter` exceeds 50, the process of integration is stopped and a message is created and sent to `Saw`.

```
   Warning (I) :: IT := Week; SENDTO(Debtor);
                  IF Number > Enough THEN
                  BEGIN DEACT(Warning); ACT(Procedure3, 0) END;
```

The node `Warning` produces messages and sends them to `Debtor`. When the NUMBER of the messages exceeds the value `Enough`, the sending is stopped and the node `Procedure3` is activated.

7.6 DEBLOCK suppresses blocking of node

Syntax:

```
DEBLOCK(<node name>);
```

It is used to finish the block state of a node (inhibition of sending messages). Its action consists in suppressing the suspended state of the next event activation of the node and to execute the event at the actual time. If the node were not blocked nothing is done.
This procedure can be used in nodes of any type.
Example:

```
Inc ::     IT := EXPO(1.0); SENDTO(FirstProc);
             IF LL(EL_FirstProc) >= 4 THEN BLOCK(Inc);
FirstProc (R) :: RELEASE
                        BEGIN SENDTO(SecProc);
                        IF LL(EL_SecProc) >= 5 THEN BLOCK(FirstProc);
                        END;
                        STAY := EXPO(5, 2);
```

```
                         IF LL(EL_FirstProc) < 4 THEN DEBLOCK(Inc);
SecProc (R) ::  STAY := 1.8;
                         IF LL(EL_SecProc) < 3 THEN DEBLOCK(FirstProc);
Exit ::
INIT TSIM := 25.0; ACT(INC, 0);
                    FirstProc := 7;
                    IF (LL(EL_FirstProc) > 4 THEN DEBLOCK(FirstProc);
```

Messages (objects) arrive the at `Inc`. They must undergo a process in parallel (7 at most) and then a sequential process. The queues before these processes have a limited capacity and the movement of the messages must be blocked upstream when this quantity is reached. Arrivals at `Inc` are stopped when the queue at `FirstProc` (capacity 7) is equal to 4. After certain time in `FirstProc` messages pass to `SecProc` to be processed one by one. The queue in `SecProc` is limited to 5. When it contains 5 messages, the `SecProc` in `FirstProc` is stopped. See that the node `Inc` can block itself after it has sent the message to `FirstProc`, checking the following queue. The node `FirstProc` also can block itself after sending the message to `SecProc`. Each node can deblock the former when it scan its EL to extract (if possible) one message.

7.7 ENDSIMUL ends the simulation at the end of the actual event

Syntax:

```
ENDSIMUL;
```

It is used to stop the simulation run. When it is executed, the actual event execution is completed and the simulation run is terminated.

This procedure can be used in nodes of any type.

Examples:

```
 Pop := Pop + 1; IF Pop = 5041 THEN ENDSIMUL;
  WRITELN('Polis population is ', Pop); PAUSE;
```

When this code is executed, the written message is displayed and the operation pauses until a key is pressed. If, when the code is executed, the value of `Pop` is 5040, after this pause, the simulation run finishes.

```
Result (A) :: WRITELN('Maximum Obtained ', MaxProfit: 8: 2);
                    WRITELN('Press C to continue; any other to finish ');
                    Ch := READKEY;
                    IF (Ch = 'c') OR (Ch = 'C')
                    THEN BEGIN ACT(StartSearch, 0); ACT(Result, 14400) END
                    ELSE ENDSIMUL;
```

When the node `Result` is activated, the value of `MaxProfit` is displayed. The message

```
 Press C to continue; any other to finish
```

appears, and the computer stops waiting for the pressing of a key. If this is C or c the `StartSearch` node is activated and the simulation continues. The node `Result` is scheduled for a new activation 14400 units of time afterwards. If other key is pressed the simulation ends.

7.8 EXTFEL extracts an event from the FEL

Syntax:

```
EXTFEL(<node name>, <time>, <index>, <event parameter>);
```

It is used to extract an event from the Future Event List. Parameters of the instruction are the elements (fields) of the event to be extracted.
`<node name>`, is the name of the node which the event points to. It must be explicitly indicated.
`<time>`, is a real expression whose value is the time of the event. If it is negative, it is not taken into account.
`<index>`, is the index of the node. If it is 0, it is not taken into account.
`<parameter>` is the event parameter. If it is 0, it is not taken into account.
This procedure can be used in nodes of any type.
The first event (if any) found with the specified values is destroyed and used memory released.
Example:

```
Avenue (R) Exit ::
---------------------------------
Accident (A) :: EXTFEL(Avenue, TimeExit, 0, 0);
```

The first event, that schedules the exit of the IL of `Avenue` that has an scheduled exit time equals to `TimeExit`, is eliminated.

7.9 FIFO puts the message at the end of the IL of a L type node

Syntax:

```
FIFO;
```

It is used to add a message to the end of the IL of a L type node. The first message in the EL of the L type node is passed to the end of the IL. The IL is formed in the order of arrival (FIFO order). As this order is the same that the default order in which the EL is processed, the FIFO procedure can be omitted unless other order specifications are also used in the same node.
It may be used only in L type nodes.
Example:

```
LinePile :: IF Box THEN LIFO ELSE FIFO
```

Messages, that have the field `Box` equal to TRUE, are added at the beginning of the IL; the others at the end (see FIFO procedure, 7.9). When they are extracted from the IL in the order of this list those with `Box = TRUE` are extracted in LIFO order (that who came last is extracted first) as in a pile of boxes. After these are the messages with `Box = FALSE` ordered as they arrived.

7.10 FREE frees a resource

Syntax:

```
FREE(<node name>);
```

It is used to release a resource from the R type node that was left by the message without releasing the resource at the leaving time (see NOTFREE procedure, 7.14). The quantity of resource freed from the indicated node is equal to the **actual value** of the USE variable. The procedure may be executed in any node when processing any message or no message at all. However, in most usual applications, it is executed while processing the message that left the resource without releasing the used capacity. In many applications the message has to do other things, or await certain time, before the abandoned resource become available.
This procedure can be used in nodes of any type.
Example:

```
Crane (R) Return, ProcLoad, Command::
    RELEASE
    IF HeapSize > 0 THEN BEGIN NOTFREE; SENDTO(Return, ProcLoad) END
                               ELSE SENDTO(Command, ProcLoad);
    IF F_Crane >= 1 THEN
                         BEGIN
                              STAY := GAMMA(MTCrane, DevTCrane);
                              LoadCrane := MINI(HeapSize, UNIFI(2, 3));
                              HeapSize := HeapSize - LoadCrane;
                         END;
 Return (R) Crane::
     RELEASE BEGIN FREE(Crane); SENDTO(FIRST, Crane) END;
     STAY := ReturnTime + TRIA(1, 2, 4);
```

A crane have to remove a heap of packages that may be transported in groups that have, at random with the same probability, 2 or 3 packages. The time to handle and transport the load is taken from a GAMMA distribution. When the heap is reduced to zero, the process finishes and the crane is available to transport another heap when that is commanded. If the remaining heap is not zero, the crane have to return to continue the removal. The crane is not available during this period. Continuation of the process has priority over any other new process of heap transportation.
Crane operation is simulated by a node `Crane` of capacity 1. It is activated by a message that represents the order to operate. This messages can queue in the EL of `Crane`. The message has a field `HeapSize` with the size of the heap to be removed and a field `LoadCrane` to keep the transported quantity. When the message is processed the `LoadCrane` is computed (all the heap or 2 or 3) and the time delay in loading and transport is taken into account. When this process finishes the message is sent to `ProcLoad` to process the actual load. If the heap is not empty, a copy is sent to a node `Return` to account for the delay to return near the heap. Otherwise, the copy is sent to `Command` that decides about new command messages to remove heaps. In the `Return` node the message is delayed a time for the return and then sent to the `Crane` again, before other commands, to continue the heap removal.
Notice the condition `IF F_Crane = 0` to avoid execution of instructions without transfer of the message to the IL.

7.11 LIFO puts the message at the beginning of the IL of a L type node

Syntax:

```
LIFO;
```

It is used to pass a message from the EL to the beginning of the IL of a L type node. The first message in the EL of the L type node is passed to the beginning of the IL. The IL is formed in the order reverse to that of arrival (LIFO order).

It may be used only in L type nodes.

Example:

```
Cellar (L) :: FIFO;
```

Messages coming to `Cellar` are put in the EL in order reverse to the arrival order (LIFO order).

7.12 MENU calls the Run Interactive Menu

.

Syntax:

```
MENU;
```

It is used to call the Run Interactive Menu (see 10.5).
This procedure can be used in nodes of any type.

7.13 METHOD sets the method of integration in a C type node

Syntax:

```
METHOD(| <RKF> | <EUL> | <RK4> |);
```

It is used in a type C node to set the integration method of the differential equations in the node. If RKF is used, the method will be a Runge Kutta method of fourth order with a fifth order calculation to estimate the error (Runge Kutta Fehlberg). The integration path is changed according to the error. If RK4 is used, the method will be a Runge Kutta of fourth order. If EUL is used, the method will be the simple Euler method of the first order. RKF is the more exact and slower. EUL is the less exact and faster.
EABS and EREL are used with RKF method of integration to change the integration step when the error in one step exceeds, respectively in absolute or relative value, the values of these variables. Default options are EABS = 0.00001 EREL = 0.0001.
This procedure may be used only in C type nodes. If used, it must be put at the beginning of the code.
Example:

```
Growth  (C) ::  METHOD(RK4);
                        Size' := K_Growth * (1 - Size / MaxSize) * Size;
                        Quantity' := MaxQuant - K_Incr * Quantity;
                        Biomass := SpGr * Size * Quantity;

HeatEffect (C) ::  METHOD(EUL);
                        K_Growth' := K_Growth_Max - Ctg(Temp) * K_Growth;
                        K_Incr' := K_Incr_Max - Cti(Temp) * K_Incr;
```

It is supposed that `K_Growth` and `K_Incr` growth very slowly with `Temp` so that a less exact but more fast integration method may be used.

7.14 NOTFREE inhibits the release of a resource

Syntax:

```
NOTFREE;
```

It is used to suppress the release of capacity by the messages going out of the IL of a R type node.
This procedure may be used only in the RELEASE instruction of R type nodes.
Example:

```
Lock (R) Ocean ::
   RELEASE BEGIN NOTFREE; SENDTO(Ocean, Fill) END;
   STAY := T_Enter + T_Ebbing + T_Leaving;
Fill (R) Elim :: RELEASE BEGIN FREE(Lock); SENDTO(DestrMess) END;
   STAY := T_Filling;
```

Messages (vessels) queue up at a lock to go down to the ocean level. The time spent in the dock is for the entrance maneuvers, lowering the level of the water, and going out. The dock is again freed for use after the dock attains again the upper level. This is simulating by not freeing the lock when the message goes out and by sending a copy to the node Fill. This delays the message by the filling time and then the resource is freed. See FREE procedure (section 7.10).

7.15 ORDER puts the message in the IL of a L node in a given order

Syntax:

```
ORDER(<real variable>, | A | D|);
```

It is used to put the IL of a L type node in a certain order. Messages arriving to the EL are passed to the IL and put:

- If the second argument is A, before the first in the IL that has greater the value of `<real variable>`.
- If the second argument is D, before the first with lower value of `<real variable>`.
- If none is found, the message is put at the end of the IL.

With the argument A an ascending order is obtained, with the D a descending order.
It may be used only in L type nodes.
Examples:

```
OrdLine (L) :: if Prior >= 5 then order(Prior, D) else FIFO;
```

Messages with the value in the field `Prior >= 5` are ordered in the IL by descending order of `Prior`. The others are put in the order of arrival at the end of the IL.

```
LinBatch (L)  :: ORDER(ProcTime, A);  k := 10;
OrdBatch (G) :: IF (LL(IL_LinBatch) >= k) and (F_Process > 0)
                    THEN BEGIN SENDTO(Process); k := k - 1 END;
Process (R)    :: STAY := ProcTime;
INIT TSIM := 200; ACT(Arrival, 0); Process := 10;
```

Messages (parts) that need different times to be processed at `Process` (field `ProcTime`) must be ordered in increasing order of the process time to optimize the processing. This is done by batches of 10. The node `LinBatch` produces this ordering. When the arriving messages make a queue of 10 and `Process` is idle, the queue is sent, in the attained order, to the node `Process`. See SORT procedure (section 7.19).

7.16 PAUSE stops the execution

Syntax:

```
PAUSE;
```

It is used to stop the execution of a simulation run. The user can restarts the simulation by pressing any key.
This procedure can be used in nodes of any type.

7.17 PUTFEL adds an event to the Future Event List

Syntax:

```
PUTFEL(|A | B|, <node name>, <time>, <index>, <parameter>);
```

It is used to put an event in the FEL. The parameters of the instruction are the elements (fields) of the event to be aggregated.

- `<node name>` is the name of the node which the event points to.
- `<time>` is a real expression whose value is the time of the event
- `<index>` is the index of the node. It must be an integer value.
- `<parameter>` is the event parameter, an integer value from 0 to 255.

The element is aggregated to the FEL keeping the ascending order of time. If there is one with exactly the same value put in `<time>` it is put After (A) or Before (B) of it according to the first parameter. See EXTFEL procedure (section 7.8).
This procedure can be used in nodes of any type.

7.18 RETARD produces a delayed function from a given function of time

Syntax:

```
RETARD(<order>, <delayed output>, <delay>, <input>);
```

It is used to generate values delayed and smoothed of an time dependent variable.
`<order>` is a number from 1 to 30 that indicates the order of the retard.
`<delayed output>` is a variable that was declared of RET type.
`<delay>` is a real parameter.
`<input>` is a real expression.
The procedure may be used only in a type C node, so that, when the node is active, it is processed repeatedly at small intervals of time. As the `<input>` variable or expression takes successive values, the values of the `<delayed output>` in each repetition tend to approach, with certain `<delay>`, those of the `<input>`. The form of approach depends on the `<order>` of the RETARD. For low order the output is the input smoothed. For higher orders the output is more and more like the input but retarded in time by the quantity `<delay>`.
The output variable must be declared in a declaration in the form:
`<variable name>: RET: <order>`.
An order n delay is calculated for a set of n differential equations.
This procedure may be only used in a type C node.
Example:

```
Spp (C) ::
  Demand' := K * Demand + GovDemand(TIME);
  RETARD(4, Supply, Ret_Months, Percep * Demand);
```

The values of `Demand` are produced by the differential equation. The `Supply` tends to adjust to the `Demand` affected by a perception factor of the `Demand`. This factor may depend on other socioeconomic variables.

7.19 SORT sorts a list of messages

Syntax:

```
SORT(<list name>, <field>,| A | D|);
```

It is used to order an EL or IL by the ascending (A) or descending (D) values of o field of the messages.
This procedure can be used in nodes of any type.
Example:

```
OrdBatch (G) :: IF (LL(EL_OrdBatch) >= 10) and (F_Process > 0)
                THEN BEGIN SORT(EL_OrdBatch, ProcTime, A);
                           SCAN(EL_OrdBatch) SENDTO(Process);
                     END;
Process (R) :: STAY := ProcTime;
INIT TSIM := 200; ACT(Arrival, 0); Process := 10;
DECL MESSAGES Arrival(ProcTime: REAL);
```

Messages (parts), that need different times to be processed at `Process` (field `ProcTime`), must be ordered in increasing order of the process time to optimize the processing. This have to be done by batches of 10 parts. The node `OrdBatch` produces this ordering. When the arriving messages make a queue of 10 and `Process` is idle, the queue is sorted in ascending order of `ProcTime` and it is send, in the attained order, to the node `Process`.

7.20 STAT displays the statistics

Syntax:

```
STAT;
```

It is used to display the standard statistics during the simulation run.
This procedure can be used in nodes of any type.
Examples:

```
 WriteStat (a) :: IT := 100;
```

Statistics are display each 100 units of time. The node `WriteStat` must be activated from outside the first time.

```
Rep (G) :: IF Reports and (EL_Memory > 20) then STAT; Sendto(DEC);
```

Each time a message with the field `Reports` equal to TRUE arrive and the queue in node `Memory` is greater than 20, the statistics are shown.

7.21 STOPSCAN stops the scanning of a list of messages

Syntax:

```
STOPSCAN;
```

It is used to stop the scanning of a list (EL or IL). When used in a SCAN instruction the list is that indicated in the instruction. If it is used in a type G node outside a GLIDER instruction the list is the EL of the node.
This procedure can be used in nodes of G type, and within a SCAN instruction, in any type of node permitting SCAN.
Example:

```
Test STOPSCAN Example 1
NETWORK
  Tour_Gate (G) Ballon_Gate ::
                 TotWeight := TotWeight + Weight;
                 IF (TotWeight <= 930) AND NOT Complete
                   THEN BEGIN  SENDTO(Balloon_Gate); END
                   ELSE BEGIN STOPSCAN;
                          Complete := TRUE; TotWeight := 0;
```

```
                     END;
      Balloon_Gate (G) :: IF Complete and Present
               THEN   BEGIN SENDTO(Tour);
                      IF LL(EL_Ballon_Gate) = 0 THEN
                      BEGIN NBatch := NBatch + 1; Complete := FALSE;
                         Present := FALSE; TotWeight := 0;
                      END;
                 END;
      Tour (R) ::
                RELEASE Present := TRUE;
                STAY := UNIF(50, 63);
      Exit (E) ::

      INIT ACT(Entrance, 0); NBatch := 0; TotWeight := 0;
           Present := TRUE; Complete := FALSE; Tour := MAXREAL;

      DECL VAR  NBatch: INTEGER; TotWeight: REAL;
                Present, Complete: BOOLEAN;
           MESSAGES Entrance(Weight: REAL);
           STATISTICS  ALLNODES;
```

Messages (passengers for a balloon) are weighted at `Tour_Gate`. While the total weight is less than 930 and `Complete` was FALSE they are sent to `Balloon_Gate`. When one incoming message cause the weight to exceed 930, the scanning of the EL is stopped, `Complete` is put to TRUE and `TotWeight` to zero. No more passengers can pass the `Tour_Gate`. Then `Balloon_Gate` can operate if the balloon is present (`Present = TRUE`). The messages in the EL are sent to `Tour`. `Complete` and `Present` are put to FALSE and the batch is counted. `TotWeight` is set to zero to permit the gathering of a new batch of passengers. The `Tour` takes between 50 and 63 minutes. When it finishes the passengers went to `Outside` and `Present` becomes TRUE.

7.22 TAB adds values to a frequency table

Syntax:

```
TAB(<real variable>, <table name>);
```

It is used to count a value for a frequency table; `<table name>` must have been declared in a TABLES declaration (see 2.7). Parameters of the tables must be initialized in the INIT section (see 3.4). The actual value of the `<real variable>` is classified to define to which frequency class of the table it belongs and one is added to the count of this class. The frequency classes and a histogram are shown when a display of the standard statistics is produced. This is always done at the end of the simulation, but also can be done interactively at any moment of the run or by the program using the STAT instruction. See 2.10 and 10.7 for declaration and description of the standard statistics.

This procedure can be used in nodes of any type.

Example

```
      Ent_People (I) :: IT := EXPO(32.4); L_Queue := LL(EL_Couter);
                              TAB(L_Queue, TabQue);
   ----------------------------------------------------------
      Exit (e) :: L_Queue := LL(EL_Counter); TAB(L_Queue, TabQue);
   ----------------------------------------------------------
      INIT
        TabQueue := 0, 10, 4;
   ----------------------------------------------------------
```

```
DECL VAR L_Queue: REAL;
  TABLES L_Queue: TabQueue;
```

Messages (people) enter the system and queue at the node Counter (not shown). They exit at node Exit. When these events happen, the queue changes. The length of the queue is passed to a real variable L_Queue and is registered at the frequency table TabQueue. This table has 10 classes (intervals) of length 4 and starts at the value 0.

7.23 TRACE starts the tracing

Syntax:

```
TRACE;
```

It is used to put the trace mode that displays a detailed account of the events process, messages movements, FEL evolution, etc., useful for debugging. It must not be executed if the system is displaying a graphic.

This procedure can be used in nodes of any type. It has not effect if used in the INIT.

Example:

```
IF EL_Shop > 10 THEN TRACE ELSE IF EL_Shop <= 5 THEN UNTRACE;
```

If the queue in Shop is greater than 10 the trace starts, when the queue decreases and becomes 5 or less the trace is stopped.

7.24 TRANS transfers a message

Syntax:

```
TRANS(<origin list>, | FIRST | LAST| <origin pointer> |, | A | B |,
    <destination list>, | FIRST | LAST | <destination pointer> |);
```

It is used to transfer a message from one position in a list to another position in another or the same list.

- <origin list> is the name of the list from which the message is taken.
- FIRST | LAST | <origin pointer> indicates the position of the message to be sent.
- <destination list> is the name of the list to which the message has to be sent.
- | A | B | is only used if a <destination pointer> follows. It is used to indicate if the message has to be put after or before the position indicated by the pointer.
- FIRST | LAST | <destination pointer> indicates the position to which the message has to be sent.

This procedure can be used in nodes of any type.

Example:

```
SCAN(IL_Depot, P_Item) IF Cost > 500 THEN STOPSCAN;
TRANS(IL_Depot, P_Item, B, NewProc, P_Rear);
```

After a scan in IL of node Dept the first message with Cost > 500 is pointed. Then it is transferred to the EL of node NewProc before the element pointed by P_Rear.

7.25 UNTRACE stops the tracing

Syntax:

```
UNTRACE;
```

It is used to stop the trace mode that displays a detailed account of the events process, messages movements, FEL evolution, etc., useful for debugging.

This procedure can be used in nodes of any type. It has not effect if used in the INIT.

It must not be executed if the system is displaying a graphic.

Example:

```
IF TIME > 2000 THEN UNTRACE;
```

Tracing mode is suppressed after the time 2000.

7.26 UPDATE updates messages fields or field variables

Syntax:

```
UPDATE( | MESS  | O_MESS | VAR | O_VAR | );
```

It is used to pass the values of fields from a message in process to the corresponding field variables or, alternatively the actual values of the field variables to the corresponding fields of the message in process. The argument indicates what has to be updated.

This procedure can be used only in nodes that process messages code of I, G, R, L, D, E type, general nodes and associate instructions of GLIDER instructions that processes messages.

Example:

```
Inspect (G) Continue :: IF pH > 4 THEN SENDTO(Continue)
                                     ELSE BEGIN Acid := TRUE;
                                     UPDATE(MESS);
                                     END;
```

While the EL of `Inspect` is searching, the messages with `pH > 4` are sent to the EL of the `Continue` node. The others are left in the EL of `Inspect` after putting their `Acid` field to TRUE. Without the UPDATE the field `Acid` is not changed.

Chapter 8

FUNCTIONS

The GLIDER provides a set of functions that give values of characteristics of the lists, some simple mathematical functions (besides those provided by the Pascal language) and some random functions that generate random variates. These functions can be used in any place in the INIT and NETWORK sections in which a variable of the same type is allowed. Here is a list of these functions an the returned values.

1. LL number of messages of a list.
2. MAXL maximum number of messages of a list.
3. MINL minimum number of messages of a list.
4. MEDL mean length of a list.
5. DMEDL deviation of the mean length of a list.
6. MSTL mean time of staying time in a list.
7. DMSTL deviation of the mean time of staying time in a list.
8. TFREE time that the list was free.
9. ENTR number of entries in a list.
10. MAX maximum value of a pair of real expressions.
11. MAXI maximum value of a pair of integer expressions.
12. MIN minimum value of a pair of real expressions.
13. MINI minimum value of a pair of integer expressions.
14. MODUL rest from dividing two real expressions.
15. BER random value from a Bernoulli distribution.
16. BETA random value from a Beta distribution.
17. BIN random value from a Binomial distribution.
18. ERLG random value from an Erlang distribution.
19. EXPO random value from an Exponential distribution.
20. GAMMA random value from a Gamma distribution.
21. GAUSS random positive value from a Normal distribution.

22. LOGNORM random value from a Lognormal distribution.
23. NORM random value from a Normal distribution.
24. POISSON random value from a Poisson distribution.
25. RAND random value from a Multivariate distribution.
26. TRIA random value from a Triangular distribution.
27. UNIF random value from a real Uniform distribution.
28. UNIFI random value from an integer Uniform distribution.
29. WEIBULL random value from a Weibull distribution.

8.1 LL number of messages of a list

Syntax:

```
LL(<list name>);
```

Returns the value (INTEGER) of the length of the list (EL or IL) indicated in `<list name>`. Example:

```
IF LL(Clerk1) <= 2 * LL(Clerk2) THEN SENDTO(Clerk1)
                                ELSE SENDTO(Clerk2);
```

8.2 MAXL maximum number of messages of a list

Syntax:

```
MAXL(<list name>);
```

Returns the value (INTEGER) of the maximum value attained for the list (EL or IL) indicated in `<list name>`.
Example:

```
WRITELN('The maximum in the third queue was ', MAXL(EL_Dec(Gat2));
```

8.3 MINL minimum number of messages of a list

Syntax:

```
MINL(<list name>);
```

Returns the value (INTEGER) of the minimum value attained for the list (EL or IL) indicated in `<list name>`.
Example:

```
MaxiFreeSpa[i] := ParkSpa[i] - MIN(IL_Parking[i]);
```

8.4 MEDL mean length of a list

Syntax:

```
MEDL(<list name>);
```

Returns the value (REAL) of the mean of the length as a function of time for the list (EL or IL) indicated in `<list name>`.
Example:

```
IF MEDL(EL_Mac[1]) > 1.5 * MEDL(EL_Mac[2])
THEN T_Mac[1] := 1.5 * M_Mac[1];
```

8.5 DMEDL deviation of the mean length of a list

Syntax:

```
DMEDL(<list name>);
```

Returns the value (REAL) of the standard deviation of the mean of the length as a function of time for the list (EL or IL) indicated in `<list name>`.
Example:

```
WRITELN('Coefficient of variation for queue ',
         DMEDL(EL_Office) / MEDL(EL_Office): 7: 3);
```

8.6 MSTL mean waiting time in a list

Syntax:

```
MSTL(<list name>);
```

Returns the value (REAL) of the mean of the waiting time of the messages that left the list (EL or IL) indicated in `<list name>`.
Example:

```
IF MSTL(EL_Park[1]) < MSTL(Park_[2]) THEN SENDTO(Park[1])
                                     ELSE SENDTO(Park[2]);
```

8.7 DMSTL deviation of mean waiting time in a list

Syntax:

```
DMSTL(<list name>);
```

Returns the value (REAL) of the standard deviation of the mean of the waiting time of the messages that left the list (EL or IL) indicated in `<list name>`.
Example:

```
IF DMSTL(EL_Door_A) > DMSTL(Door_B) THEN Ch := 'A'
                                    ELSE Ch := 'B';
WRITELN('Queue in door ',Ch,' had greater fluctuation');
```

8.8 TFREE time that the list was free

Syntax:

```
TFREE(<list name>);
```

Returns the value (REAL) of total time in which the list (EL or IL) indicated in `<list name>` was void.
Example:

```
Access (I) A,B :: IT := UNIF(5, 7); IF BER(Prob_A) THEN SENDTO(A)
                                                   ELSE SENDTO(B);
 Prob_A := (1 + TFREE(EL_A)) / ((1 + TFREE(EL_A)) * (1 + TFREE(EL_A)));
```

8.9 ENTR number of entries in a list

Syntax:

```
ENTR(<list name>);
```

Returns the value (INTEGER) of the number of entries in the list (EL or IL) indicated in `<list name>`.
Example:

```
IF ENTR(EL_Theater) > 120 THEN BLOCK(Th_Arrival);
```

8.10 MAX maximum value of a pair of real expressions

Syntax:

```
MAX(<real expression>, <real expression>);
```

Returns the value of that `<real expression>` that is greater.
Example:

```
n:=6; m := MAX(7, MAX(8, n)) * 2;
```

The value assigned to m is 8;

8.11 MAXI maximum value of a pair of integer expressions

Syntax:

```
MAXI(<integer expression>, <integer expression>);
```

Returns the value of that `<integer expression>` that is greater.

8.12 MIN minimum value of a pair of real expressions

Syntax:

```
MIN(<real expression>, <real expression>);
```

Returns the value of that `<real expression>` that is lower.

8.13 MINI minimum value of a pair of integer expressions

Syntax:

```
MINI(<integer expression>, <integer expression>);
```

Returns the value of that `<integer expression>` that is lower.
Example:

```
 Me := MAXI(MINI(X1, X2), MAXI(MINI(X2, X3), MINI(X1, X3)));
 WRITELN('Medium is ',Me);
```

8.14 MODUL rest from dividing two real expressions

Syntax:

```
MODUL(<real expression>, <real expression>);
```

Returns the remainder that results from dividing the value of the first `<real expression>` by the second `<real expression>`, when the division is performed until the units in the quotient.

Example: MODUL(804, 364.25) is computed by dividing 804 / 364.25 = 2
MODUL(804, 364.25) is computed by: 804 - 2 * 364.25 = 75.5; (assuming the year equal to 364.25 days, 804 days are 2 years plus 75.5 days)

8.15 BER random value from a Bernoulli distribution

Syntax:

```
BER(<real expression>, / <stream> /);
```

Returns the BOOLEAN value TRUE with a probability equal to the value of `<real expression>` and FALSE with a probability of `1 - <real expression>`. The probabilities are computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

Example:

```
IF BER(0.6) THEN SENDTO(LargeDep) ELSE SENDTO(SmallDep);
```

At random 60 % of the times the messages are sent to `LargeDep`. 40 % to `SmallDep`.

8.16 BETA random value from a Beta distribution

Syntax:

```
BETA(<real expression>, <real expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is a BETA function with first parameter equal to the first `<real expression>` and second parameter equal to the second `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.17 BIN random value from a Binomial distribution

Syntax:

```
BIN(<integer expression>, <integer expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is a binomial function with first parameter equal to the first `<integer expression>` and second parameter equal to the second `<integer expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.18 ERLG random value from an Erlang distribution

Syntax:

```
ERLG(<real expression>, <integer expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is an Erlang function. This is the sum of a number equal to `<integer expression>` of exponential functions with mean equal to `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.19 EXPO random value from an Exponential distribution

Syntax:

```
EXPO(<real expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is an exponential function with mean equal to `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.20 GAMMA random value from a Gamma distribution

Syntax:

```
GAMMA(<real expression>, <real expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is a GAMMA function with mean equal to the first `<real expression>` and deviation equal to the second `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.21 GAUSS random positive value from a Normal distribution.

Syntax:

```
GAUSS(<real expression>, <real expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is a normal function in which only the positive values are considered. The mean is equal to the first `<real expression>` and the standard deviation equal to the second `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.22 LOGNORM random value from a Lognormal distribution

Syntax:

```
LOGNORM(<real expression>, <real expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is a lognormal function with mean equal to the first `<real expression>` and deviation equal to the second `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted 1, is assumed.

8.23 NORM random value from a Normal distribution

Syntax:

```
NORM(<real expression>, <real expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is a normal function with mean equal to the first `<real expression>` and deviation equal to the second `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed. The algorithm may produce negative values.

8.24 POISSON random value from a Poisson distribution

Syntax:

```
POISSON(<real expression> / <,stream> /);
```

Returns an INTEGER value of a random variable whose probability function is a Poisson function with mean equal to `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.25 RAND random value from a multivariate distribution

Syntax:

```
RAND(<array>) / <,stream> /);
```

This is in reality a procedure. It is used to generate random variates taken from multivariate random functions. The n-dimensional random functions are given by an n-dimensional `<array>` whose values are an n-dimensional frequency table. It must be declared of FREQ type and initialized in the INIT section.
The values of the variables must be declared in a two dimensional arrays (one array for each variable). In the first row are the successive values of the first variable, in the second row the successive values of the second variable, etc. The number of rows is then equal to the number of variables. The number of columns must be equal to the number of values of the variable that has the maximum number of values. The name of this array must be VAL_`<array>`. When the procedure is evaluated, a random value is computed for each variable according to the frequency table given in `<array>` and these values are stored in an array of name RAND_`<array>` to be used in the program. The values are computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.
Example 1:

```
  Test RAND procedure for multivariate random values.
  This example generates random numbers from a distribution
  given by a three variable frequency table given by an array Mul of
  three indexes. The multivariant frequency table is:

                15 18                   15 18
               ------                  ------
        2.0   |  1  3          2.0   |  3  2
   0.1  5.0   |  4  3     0.4  5.0   |  2  1
       10.0   |  3  1         10.0   |  3  6

   So, Mul[0.1,5.0,15]=4 is the number of cases in which the values
   of the variables were (0.1, 5.0, 15);
```

```
    Mul[0.4,10.0,18]=6
 The sum of all values is 32. The first matrix sum up 15, the
 second one, 17.

 The RAND algorithm selects one vector, for example (0.1, 5.0, 15)
 with a probability according to the frequency table.
 Each value is selected according to its marginal probability
 distribution conditional to the previous values.
 It is decided between 0.1 and 0.4 according its
 marginal probabilities: 15/32 for 0.1 and 17/32 for 0.4.
 If the selected is, for instance 0.1, the second value is selected
 according to the marginal probabilities: (3+1)/17 for the
 value 2.0,(4+3)/17 for the value 5.0, (3+1)/17 for the value 10.0.
 If the selected is, for instance 5.0, the third value is selected
 according to the probabilities 4/7 for 15.0 and 3/7 for 18.0.
 See that the probability of the value (0.1, 5.0, 15) is
 then 17/32*7/17*4/7=4/32, as may be see directly.
 The following program computes 32000 random numbers with the
 frequencies in the array Mul, counts the different results
 and put them in a table Zzz similar to Mul. The values in Zzz
 must be approximately those in Mul.
NETWORK
 A:: IT := 1;    RAND(Mul,2);
 IF RANDV_Mul[1] = 0.1   THEN I := 1 ELSE I := 2;
 IF RANDV_Mul[2] = 2.0   THEN J := 1 ELSE
 IF RANDV_Mul[2] = 5.0   THEN J := 2 ELSE J := 3;
 IF RANDV_Mul[3] = 15.0  THEN K := 1 ELSE K := 2;
 Zzz[I, J, K] := Zzz[I, J, K] + 1;

 Result (A):: WRITELN;
  FOR I := 1 TO 2 DO
   BEGIN  WRITELN;
    FOR J := 1 TO 3 DO
     BEGIN WRITELN;
      FOR K := 1 TO 2 DO WRITE(Zzz[I, J, K] / 1000: 7: 4);
     END
   END;

  PAUSE; ENDSIMUL;

INIT
 ACT(A, 0);
 ASSI Mul[1..2, 1..3, 1..2] :=
       (  ( ( 1, 3 ),( 4, 3 ),( 3, 1 ) ) ,
          ( ( 3, 2 ),( 2, 1 ),( 3, 6 ) )  ) ;
 ASSI VAL_Mul[1..3, 1..3]:=( (0.1, 0.4, 0.0), (2, 5, 10), (15, 18, 0) ) ;
 ASSI Zzz[1..2, 1..3, 1..2] :=
      (  ( ( 0, 0 ), ( 0, 0 ), ( 0, 0 ) ) ,
         ( ( 0, 0 ), ( 0, 0 ), ( 0, 0 ) )  ) ;

 ACT(Result, 32000);

DECL  VAR Mul: ARRAY[1..2, 1..3, 1..2] OF FREQ;
          Zzz: ARRAY[1..2, 1..3, 1..2] OF REAL;
```

```
           VAL_Mul: ARRAY[1..3, 1..3] OF REAL;
           I, J, K: INTEGER;
 END.
```

Example 2:

```
  Logs are processed by a set of machines. The processing time
  was adjusted by regression to a linear function of the Length L
  and the Diameter D of the logs. Logs arrive each 10 m. with random
  sizes. A frequency table aggregated in 3 diameters (32, 36, 40
  inches) and 4 lengths (15, 25, 35, 45 feet), obtained by sampling,
  gives the quantity in each class.

  NETWORK
  Arrivals (I)  :: IT := 10; RAND(Fld);
                   L := RANDV_Fld[1]; D := RANDV_Fld[2];
  Machines (R)  :: STAY := 8.5 + 0.32 * L + 0.21 * D;
  Dep (E)       ::

  INIT
   ACT(Arrivals, 0); TSIM := 2000; Machines := 3;
   (*Multivariate frequency array  D: rows L: columns:    *)
   ASSI Fld[1..3, 1..4] :=
                                {  Length feet   }
                                { 15  25  35  45 }
                       {32}  ( ( 123, 108, 47, 12 ),
    {Diameter inches}  {36}    (  89, 104, 99, 43 ),
                       {40}    (  22, 87, 106, 172) ) ;

   (*values of the variables (columns) :               *)
   ASSI VAL_Fld[1..2, 1..4]:=
              ( (32, 36, 40, 0), (15, 25, 35, 45)) ;

  DECL  VAR_Fld: ARRAY[1..3, 1..4] OF FREQ;
        VAL_Fld: ARRAY[1..2, 1..4] OF REAL;
        MESSAGES Arrivals(L, D: REAL);
        STATISTICS ALLNODES;
 END.
```

8.26 TRIA random value from a Triangular distribution

Syntax:

```
    TRIA(<real expression>, <real expression>, <real expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is a triangular function with minimum equal to the first `<real expression>`, mode equal to the second `<real expression>`, and maximum equal to the third `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.27 UNIF random value from a real Uniform distribution

Syntax:

```
UNIF(<real expression>, <real expression> / <,stream> /);
```

Returns a REAL value of a random variable whose probability function is a continuous uniform function with minimum equal to the first `<real expression>` and maximum equal to the second `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

8.28 UNIFI random value from an integer Uniform distribution

Syntax:

```
UNIFI(<real expression>, <real expression> / <,stream> /);
```

Returns an INTEGER value of a random variable whose probability function is a discrete uniform function with minimum equal to the first `<real expression>` and maximum equal to the second `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted 1, is assumed.

8.29 WEIBULL random value from a Weibull distribution

Syntax:

```
WEIBULL(<real expression>, <real expression> / <,stream> /);
```

Returns a value of a random variable whose probability function is a function with first parameter equal to the first `<real expression>` and second parameter equal to the second `<real expression>`, computed with random numbers taken from the stream indicated by `<stream>`. This is an integer from 1 to 10. If omitted, 1 is assumed.

Chapter 9

RESERVED WORDS

The GLIDER system defines a set of identifiers as reserved words. They cannot be defined by the user.

9.1 Message variables

- **GT** real; generation time of the message being processed.
- **O_GT** real; generation time of the message being processed.
- **ET** real; time of entry of the message to the actual list.
- **O_ET** real; time of entry of the alternative message to the actual list.
- **XT** real; time of exit of the message from the actual list.
- **O_XT** real; time of exit of the message from the actual list.
- **USE** real; quantity of resource to be used by the message.
- **O_USE** real; quantity of resource to be used by the alternative message.
- **NUMBER** integer; generation number, in its origin node, of the message being processed.
- **NODE** string; name of the origin node of the message being processed.
- **O_NUMBER** integer; generation number, in its origin node, of the alternative message being processed.
- **O_NODE** string; name of the origin node of the alternative message being processed.
- **REMA** real; its value is the remaining time of a preempted message; although it must be declared by the user in a MESSAGES declaration, it is managed by the instruction PREEMPTION.

9.2 Event variables

- **IEV** integer; index of the event.
- **EVP** byte; parameter of the event.
- **SU** boolean: parameter to suspend the event.

9.3 Indexed node variable

- **INO** integer; index of the node being executed; also index of the EL from which the messages are selected from a type D node with multiple predecessors.

9.4 Control and state variables

- **ICOPY** integer; number of the copy produced by a COPYMESS instruction.
- **EXTREL** boolean; TRUE if there was extraction of messages by an instruction REL.
- **SYNC** boolean; TRUE if there was assembling or synchronization of messages by ASSEMBLE or SYNCHRONIZE instructions.
- **SENT** boolean; TRUE if a SENDTO was executed.
- **RND**[< *integer* >] real; index from 1 to 10; is the actual value of the random number between 0 and 1 of the stream given by the index.
- **RN**[< *integer* >] real; index from 1 to 10, is the actual value of the random number (between 1 and the module 21474836647) of the stream given by the index. Its initial value is the seed of the stream. This value can be changed by the user.

9.5 Node dependent variables

In the following `<node name>` includes the index between [ ], if the node is multiple.

- **IL**`<node name>` designs the internal list of the node.
- **EL**`<node name>` designs the entry list of the node. If the node is of D type with several predecessors it must include the reference to the predecessor between ().
- **U**`<node name>` real; its value is the actual quantity of used resource of the R node.
- **F**`<node name>` real; its value is the actual quantity of free resource of the R node.
- **M**`<node name>` real; its value is the total quantity of resource of the R node.
- **DT**`<node name>` real; its value is the integration step that will be used in the indicated C type node. The RKF method may change it during the integration process.

9.6 Variables depending on user's variables

- **O_**`<variable>` where `<variable>` is a field variable. It keeps the value of the alternative field.
- **RANDV_**`<frequency array name>` keeps the vector of the random values taken from a random multivariate frequency table.
- **VAL_**`<frequency array name>` must be given by the user; it must contain the possible values of the random variables corresponding to a multivariate frequency table.
- **N_**`<DBASE table name>` is generated when DBASE tables are used; it contains the number of records in the table.
- **NOM_**`<DBASE table name>` is an array generated when DBASE tables are used. It contains the names of the first field of the records in the table.
- **name in fields** are constants generated when DBASE tables are used. They have the names of the first fields in the table and their values are 1, 2, 3, etc. (see example in 2.8)

9.7 Variables that may be initialized by the user

- **IT** real; its value is the Interval Time until the next activation of the node being executed.
- **NT** real; its value is the Next Time of activation of the node being executed.
- **STAY** real; its value is the scheduled permanence of the message being processed in the IL of a node type R.
- **TIME** real; its value is the actual simulation time.
- **TITLE** string of 80 characters. It contains the title of the experiment.
- **TSIM** real; its value is the total simulation time.

9.8 Classes of declarations

CONST	DBTABLES	FACTORS	GFUNCTIONS
MESSAGES	NODES	PROCEDURES	RESPONSES
STATISTICS	TABLES	TYPE	VAR

The use of RESPONSES and FACTORS (for the design of experiments) will be introduced in a future version of this manual.

9.9 GLIDER or Pascal predefined types

ARRAY	BOOLEAN	BYTE	CHAR
CONT	DOUBLE	FILE	FUNCTION
FREQ	INTEGER	LONGINT	POINTER
PROCEDURE	REAL	RECORD	RET
SET	STR80	STRING	TEXT
WORD			

9.10 Pascal separators

BEGIN	CASE	DO	DOWNTO
ELSE	END	END.	FOR
GOTO	IF	IN	NIL
OF	REPEAT	THEN	TO
UNTIL	WHILE		

9.11 GLIDER types of functions (GFUNCTIONS)

DISC	FREQ	POLYG	SPLINE
STAIR			

9.12 GLIDER separators

NETWORK	INIT	DECL	END.
MODULE	EXPER	ENDEXP	

MODULE is used in the NETWORK section to divide the simulation program in units. This is very convenient in case of very large programs. The use of NETWORK, INIT, DECL and END. is discussed in the previous chapters (see Chapters 1, 2, 5, etc.)

9.13 Operators

AND	DIV	MOD	NOT
OR	XOR		

9.14 GLIDER instructions and procedures

ACT	ASSEMBLE	ASSI	BEGINSCAN
BLOCK	CLRSTAT	COPYMESS	CREATE
DBUPDATE	DEACT	DEASSEMBLE	DEBLOCK
DOEVENT	DONODE	ENDSIMUL	EXTFEL
EXTR	FIFO	FILE	FREE
GRAPH	INTI	IT	LIFO
LOAD	MENU	METHOD	NOTFREE
NT	ORDER	OUTG	PAUSE
PREEMPTION	PUTFEL	REL	RELEASE
REPORT	RETARD	SCAN	SELECT
SENDTO	SORT	STAT	STATE
STAY	STOPSCAN	SYNCHRONIZE	TAB
TITLE	TRACE	TRANS	TSIM
UNLOAD	UNTRACE	UPDATE	USE

9.15 Pascal procedures

APPEND	ASSIGN	BLOCKREAD	BLOCKWRITE
CLOSE	CLRSCR	DELAY	DISPOSE
FILEPOS	FILESIZE	GETDATE	GETTIME
GOTOXY	MAX	MIN	NEW
OPEN	READ	READLN	RELEASE
RESET	REWRITE	SEEK	SOUND
STR	TRUNCATE	VAL	WRITE
WRITELN			

9.16 GLIDER functions

BER	BETA	BIN	DMEDL
DMSTL	ENTR	ERLG	EXPO
GAMMA	GAUSS	LL	LOGNORM
MAX	MAXI	MAXL	MEDL
MIN	MINI	MINL	MODUL
MSTL	NORM	POISSON	RAND
TFREE	TRIA	UNIF	UNIFI
WEIBULL			

9.17 Pascal functions

ABS	ARCTAN	COPY	COS
EOF	EXP	FILESIZE	INT
KEYPRESSED	LENGTH	LN	NOSOUND
OFS	ORD	RANDOM	READKEY

ROUND	SEG	SIN	SQRT
TRUNC	WHEREX	WHEREY	

9.18 Colors

BLACK	BLUE	BROWN	CYAN
DARKGRAY	GREEN	LIGHTBLUE	LIGHTCYAN
LIGHTGRAY	LIGHTGREEN	LIGHTMAGENTA	LIGHTRED
MAGENTA	READ	RED	WHITE
YELLOW			

9.19 Pascal constants

HEAPORG	HEAPPTR	HEAPEND

9.20 Other reserved words

- **A** or **B** used in SENDTO, TRANS, and PUTFEL to indicate if the message is put before (B) or after (A) the pointed message (in the case of SENDTO or TRANS) or the event with the same value of TIME (in the case of PUTFEL). Note that there are not reserved words outside these instructions.
- **A** or **D** used in the procedures ORDER and SORT to indicate the increasing (A) or decreasing (D) order. They are not reserved word outside these procedures.
- **ALL** used in ASSEMBLE or SYNCHRONIZE to indicate that all the elements of the list must be assembled or synchronized.
- **ALLNODES** used in STATISTICS to ask for statistics of all the nodes.
- **EABS** absolute error for RKF method of integration.
- **ELIM** in the ASSEMBLE instruction indicates that all the assembled messages but the representative will be eliminated.
- **EQUFIELD** in the ASSEMBLE or SYNCHRONIZE to group messages with equal value in a field.
- **EREL** relative error for RKF method of integration.
- **EUL** indicates Euler method of integration.
- **FIRST** in an ASSEMBLE instruction it indicates that the first assembled message will represent the group.
- **FIRST** or **LAST** in SENDTO and TRANS to indicate if the sent or transferred message is put at the beginning or ending of the list. These words are also used in pointers as indicated in section 2.1.2.
- **FREE** in SENDTO to indicate the first free in a multiple node.
- **LABEL** used in the heading of a node (to declare labels for the node).
- **MESS** used in UPDATE to update the fields of the message being processed.
- **MIN** in SENDTO to indicate the less occupied in a multiple node.
- **MAXSEL** in SELECT to indicate the maximum of messages to be selected in the next selection attempt.

- **NEW** in an ASSEMBLE instruction it indicates that a new message will represent the group.
- **NREP** number of the replication when the Replications option is used (its use will be discussed in a future version of the GLIDER manual).
- **O_MESS** used in UPDATE to update the fields of the alternative message being processed.
- **OUTF** alternative file variable in REPORT instruction.
- **O_VAR** used in UPDATE to update the fields variables of the alternative message being processed.
- **RKF** indicates Runge-Kutta-Fehlberg method of integration.
- **RK4** indicates fourth order Runge-Kutta method of integration.
- **TRUE** used in the instruction ASSEMBLE instead of a logical expression.
- **VAR** used in UPDATE to update the field variables of the message being processed.
- **VAR** used in the heading of a node (to declare local variables).

9.21 Types of Node

The following letters, that indicate the types of node, are not reserved words; see chapter 5; however, they are included here.

- G
- L
- I
- D
- E
- R
- C
- A

9.22 Constants

- **MAXREAL** 1.7E38
- **MAXINTEGER** 32767
- **MAXLONGINT** 2147483647
- **MAXWORD** 65536
- **PI** 3.14159627

Chapter 10

OPERATION

The GLIDER system operates in the MS-DOS operating system.

10.1 GLIDER system

The Operation of the GLIDER system needs at least the following files to function in its elementary form:

UG8.EXE	GLIDER compiler
TURBO.TPL	Turbo Pascal line compiler v-6
PP.PAS	GLIDER procedures
FUNG.TPU	GLIDER functions
INPOUT.TPU	I/O unit
GDINAR.TPU	Random variates from multivariate distributions
GGRAPHIC.TPU	GLIDER graphic generation
RATON.TPU	Mouse managing unit
OBJECTS.TPU	Mouse management
DRIVERS.TPU	Mouse driver
GRAPH.TPU	Turbo Pascal graphic unit
TPC.EXE	Turbo Pascal procedures
GL.BAT	System monitor
TURBO.EXE	Turbo Pascal compiler (if the editor of this program is used)
EGAVGA.BGI	Graphic unit

The elementary use requires to have the source files in the same directory as the GLIDER files and its presentation is not very sophisticate, but may be convenient during the development of large programs because of memory saving in the compilation.

An environment may be used without those restrictions. To use the environment the following units have to be added:

GLIDERIN.EXE	Environment unit
GLEDIT.EXE	Editor unit
TVGLIDER.BAT	System monitor
TVGL.BAT	System monitor

In this case TURBO.EXE and GL.BAT are not needed.

To use graphic input the following units have to be added

GLIDERIN.EXE	Environment unit
GLEDIT.EXE	Editor unit
GLIDER.GRA	Graphic data

TVGLIDER.EXE	System monitor
TVGL.BAT	System monitor
GLISIMB.EXE	Symbols generator

The files TURBO.EXE, TURBO.TPL, GRAPH.TPL, EGA.BGI, and TPC.EXE are files from the package **Turbo Pascal V-6 of Borland Corp.** *and are not provided with the GLIDER system. The user may contact with authorized dealers to get them*

10.2 Source program

The source program must be in a file with extension **GLD** in ASCII code. It is a text file with lines of 75 characters at most, from column 1 to 75.

10.3 Operation without environment

The source file must be in the directory of the GLIDER system. The process of compiling and execution is called with the command:
$> GL < filename >$
The extension GLD must be omitted. The GLIDER compiler produces a Pascal program that is then compiled by the Turbo Pascal compiler TPL.EXE. The executable program remains in the file $< filename.EXE >$. Then the execution phase begin. That starts with the Initial Menu.

10.4 Initial menu

This menu appears at the **beginning** of the execution. It has the following options:

I	Initialize and run
T	set Title of the experiment
R	make Replications
G	Graphic
A	set Additional time
Q	Quit

- **I** to initialize the system variables; it executes the INIT section and it calls the Run Interactive Menu.
- **T** allows to introduce a title up to 80 characters that will appear in the statistical output.
- **R** allows to make replications of the run without re-initializing the seeds of the random numbers.
- **G** to display a graphic for the results filed with the FILE instruction during a simulation run.
- **A** to run an experiment with extra time **after a run**.
- **Q** to abandon the run

After this menu the INIT section is executed and the Run Interactive Menu is called.

10.5 Run interactive menu

It appears after the execution of the INIT section and just before the beginning of the simulation. During the simulation it may be called by pressing the M type.
It has the following options:

C Continue
S initialize Statistics
E read Experiment
W Write statistics
F display Future event list
D Display or skip statistics after each replication
T Trace
G Graphic
Q Quit

- **C** to continue the operation or run.
- **S** to initialize Statistics and continue the run.
- **E** to shown the Experimental variables; these may be changed by the user.
- **W** to write the standard statistics at the actual state of the run.
- **F** to display Future Event List indicating Node, Time, Index and Event Parameter.
- **D** to allow to decide if the standard statistics will be displayed or skipped after each replication.
- **T** to put the tracing mode.
- **G** to allow to display a graphic for the results filed with the FILE instruction during the simulation up to the actual time.
- **Q** to finish the simulation.

10.6 Tracing

In the Trace mode the program is executed step by step. It may be very useful for inspection and debugging. In each event the movements of the messages are reported in detail and the FEL is displayed. The steps are advanced pressing any key different of T (stop tracing) and M (call Menu). Tracing may be very useful for inspection and debugging.

Tracing may be called from the Run Interactive Menu, from the program (TRACE procedure, see 7.23) or by pressing the T key during the running. If it is called from the Menu the system asks if user want to keep the output shown in the screen in a file for detailed examination. Tracing is terminated by pressing the T key several times or M to call the Run Interactive Menu. See an example in the Chapter 1.

10.7 Statistics

The standard statistics of the simulation are requested by the STATISTICS declaration (see 2.10) in which nodes and variables for which statistics are wanted are indicated. Statistics are always displayed at the end of the simulation run. They may also be displayed when a STAT instruction (see 7.20) is executed or by request from the Run Interactive Menu. The statistics presented are:

- The title of the experiment; by default "Basic Experiment". The user can put other titles from the program (see instruction TITLE, section 6.27) or from the Execution Menu.
- The total time of simulation and the last time in which the statistics were initialized (see procedure CLRSTAT, section 7.4).
- The number of the replication.
- The date and time of the beginning of the execution.

- The real time spent in the simulation run.
- For the I type nodes the number of generated messages.
- For each EL and IL the statistics are:
  - the number of entries, in the list.
  - the actual length.
  - the maximum length attained.
  - the mean length and its deviation.
  - the maximum waiting time that one message had to wait in the list.
  - the mean waiting for all the messages that **were** in the list.
  - its deviation.
  - the time that the list remained empty.
- For the E types nodes the mean time in the system of the messages destroyed in the node and its deviation.
- For the used time of the resources and the requested variables the statistics given are:
  - the mean (t) and deviation of the variables as a function of time.
    Example: if a variable take the values:
    2 at time 0; 5 at time 4; 4 at time 5; 7 at time 8; and remain 7 to the end time, that is 12; then the mean is:
    $[(4-0)*2+(5-4)*5+(8-5)*4+(12-8)*7]/(12-0) = 4.4166667$
  - the maximum and minimum values.
  - the mean (v) and deviation of the values attained for the variable.
    Example: In the above example this mean is:
    $[2+5+4+7]/4 := 4.5$
  - the actual (final) value.

After the statistics the Final Menu is displayed

10.8 Final Menu

R	Repeat statistics
C	Continue
M	Menu
F	Frequency tables
G	Graphic

- **R** The system asks for a file to put the statistics. It may be CON to use the screen again, LPT1 for the printer or any file indicated by the user.
- **C** Continue to the next menu that will be the **Initial Menu** again to continue the simulation with additional time or to repeat the run with possible modifications. From this it is also possible to quit the simulation.
- **M** Call the Run Interactive Menu.
- **F** To display frequency tables.
- **G** to display a graphic for the data filed with the FILE instruction during a simulation run.

When the option C is selected the Initial Menu is called again.

Chapter 11

DEVELOPMENT OF GLIDER

11.1 The GLIDER group

The development of the language started in 1985 from a proposal made by Renato Del Canto, who participated in the early meetings with Carlos Domingo and Marisela Hernández. All them were working at the IEAC (Institute for Applied Statistics and Computing of the Economy Faculty at Los Andes University, Mérida, Venezuela). The general outlines were designed by C.Domingo and M. Hernández during 1985-1986, and useful suggestions were made by Cristina Zoltán from Simón Bolívar University. Since 1986, a formal group, constitued by Carlos Domingo, Marta Sananes, Giorgio Tonella, José G. Silva and Herbert Hoeger, developed important features of the language and advised the implementation of the compiler, made by C.Domingo and the graphic input programmed by M.Sananes. G. Tonella, from the CESIMO (Modeling and Simulation Research Centre of the Engineering Faculty), was responsable for the development of applications that suggested new developments and improvements. He also contributed with general key ideas linking the structure of the language with the general theory of simulation. Important suggestions, developments and applications were made in graduate and undergraduate thesis, supervised by member of the GLIDER group (such as Carlos Domingo, Herbert Hoeger, Marta Sananes, Giorgio Tonella):

Omar Velandia (1990), Oscar Nuñez (1990), Cruz Mario Acosta (1991)[1], Gerardo Rojas (1991), Henry Reinoza (1992)[16], Maria Dávila (1992), Mayba Uzcátegui (1992)[24], Segundo Quiróz (M.Sc. 1992)[15], Liliana Capacho (1992), Dominique Tineo (1992), Hungría Berbesí (1993), Tania Zambrano (M.Sc. 1993)[26], Mauricio Jeréz (M.Sc. 1993), Esperanza Díaz (1993)[3], Kay Tucci (1993)[23], Francisco Palm (1994)[14], Gilberto González (1994), Terán Osvaldo (M.Sc. 1994)[19], Ricardo Guanieri (M.Sc. 1994), Margarita Molina (M.Sc. 1995).

And with the research training projects of the following students:

Kay Tucci (1993), Tania Jiménez (1994)[13], Gilberto González (1994), Rafael Tineo (1996), Dante Conti (1996). Marisol Benitez (1996).

Financial support were obtained from the C.D.C.H.T. (Comittee for Scientific, Humanistic and Technological Development) of the Andes University, Mérida, Venezuela for the period 1992-1994 (project E-122-92) and 1996-1998 (project I-524-95-AA).

11.2 How to get the GLIDER

The software for using the GLIDER and a User's Manual in Spanish or in English is available from: Carlos Domingo, IEAC, FACES, Universidad de los Andes, Mérida, Venezuela. email: carlosd@ula.ve or from: Giorgio Tonella, CESIMO, Facultad de Ingeniería, Universidad de los Andes, Mérida, Venezuela. email: cesimo@ula.ve.

GLIDER is available in two versions: a commercial and a student version, for MS-DOS or Unix operating systems.

The software does not include Turbo Pascal V.6. Some unit of this system are required to use the GLIDER. This system must be obtained from Borland International or authorized dealers.

11.3 Differences of Version 4.0 (July 1996) with the previous versions

11.3.1 Introduction

Only the main differences are listed below. The first prototypes developed during 1985-1987 had very few commands (such as STAY, STOP, TSIM, TIME, FIFO, LIFO, STAT, USE, PREE (for Preemption), ACT, ORDER, ASSEMBLE, SYNC). They included the 6 basic nodes (G, L, I, D, E, R) and the following commands for the lists: EXAM, EXTRACT, MOVE, ACREF, FIN, SORT. None of these were included in the later versions. In addition, the code of nodes were divided depending of the code functions (for example in the following parts: Assign, Next, File, Revi, Use, etc.). The heading of node was completly different from the present version.

11.3.2 Version alfa (1988) and beta (1989)

The alfa and beta versions are more similar to the present version. The alfa version was including the instructions POINT, EXAM, EXTR, MOVE to use records. It includes other commands that disappears in the version 1 such as NOSUG, EXAM, MOVE. The beta version included (in addition to the alfa version) the instructions EXTR, GRAPH, SCAN, CLRSTAT, DEASSEMBLE, TRACE, UNTRACE, DONODE, DOEVENT, RELEASE, STATE, TAB and most of the basic GLIDER functions. It included also node with subindex (multiple node)

11.3.3 Version 1 (February 1991)

The differences with the previous beta version were the inclusion of:

- Node type C with the METHOD Euler and Runge Kutta Fehlberg.
- Instruction GRAPH
- Instruction ASSI
- Instruction FILE.
- Instruction INTI.
- Instruction OUTG.
- Procedure DEACT.
- Important changes in the instructions ASSEMBLE, SCAN, SELECT, SYNCHRONIZE.
- Elimination of the instructions MOVE, SPLIT, COUNT, SENDTOF.
- Instruction COPY.
- Functions MAX, MAXI, MIN, MINI.
- Random functions BETA, BER, GAMMA, LOGNORM, POISSON, TRIA, WEIBULL.
- The GLIDER environment (without the graphic facility).
- The syntactical editor.

11.3.4 Version 1.1 (May 1992)

It included:

- Node type A.
- METHOD Runge Kutta (RK4).
- LABEL and VAR as local variable of a node.
- Functions GAMMA, BETA, LOGNORM.
- Interpolation with SPLINE.
- Function MODUL.
- Graphic facility to develop the model.

11.3.5 Version 2.0 (May 1993) and Version 2.1 (November 1993)

It included:

- Change of the instruction NODE to the instruction MESSAGES.
- The instruction TRANS.
- The instructions REPORT ... ENDREPORT.
- Change in ASSEMBLE (EQUFIELDS).
- The instruction FILE.
- A new editor.

11.3.6 Version 3.0 (April 1994) and 3.1 (September 1994)

It included:

- Possibility to use MODULE to divide the network in units.
- Possibility to connect with a data base (DBASE IV, instructions LOAD, UNLOAD, DBUPDATE).
- Procedure RETARD.
- Procedures BLOCK and DEBLOCK.
- Procedures BEGINSCAN and MENU.
- Elimination of the instructions GO, GOALL, STOP, STOPALL.
- Change in the instruction SENDTO.
- Elimination of the procedure AGRE.
- Elimination of resources with size 0 and introduction of resources with size MAXREAL.
- New version of the manual in Spanish.
- New default statistics.
- First short version of the manual in English.
- Change of instruction COPY to COPYMESS.
- A new version of the GLIDER environment.

11.3.7 Version 4.0 (July 1996)

It includes:

- A complete new version of the manual (in English).
- A correction of some bugs and a complete review of all the commands with minor changes.
- A version for the Unix operating system.
- Elimination of GPOINTER declaration.

Bibliography

[1] Acosta Castillo, C. A. 1991. "Editor Sintáctico del Lenguaje de Simulación GLIDER, Documento de Diseño". Trabajo especial de grado. (*Syntactic Editor for GLIDER Simulation Language. Thesis*) Escuela de Ingeniería de Sistemas, Universidad de los Andes, Mérida, Venezuela. Tutor: G. Tonella.

[2] Acosta Castillo, C. A. 1991. "Ambiente Integrado del GLIDER: Manual del Usuario, Versión 1.0". (*Integrated Environment of GLIDER: User's Manual, Version 1*) CESIMO Technical Report IT-9101, Universidad de los Andes, Mérida, Venezuela. March 1991.

[3] Díaz E. 1993. "Modelo en GLIDER para el Central Azucarero de Río Turbio". Trabajo especial de grado. (*GLIDER Model of Sugar Mill at Rio Turbio. Thesis*) Escuela de Ingeniería de Sistemas, Universidad de los Andes, Mérida, Venezuela. Tutor: C. Domingo.

[4] Domingo, C. 1988. "GLIDER, A Network Oriented Simulation Language for Continuous and Discrete Event Simulation". Intl. Conf. on Math. Models., Madras, India. (Aug. 11-14).

[5] Domingo, C. y Hernández. M. 1985. "Ideas Básicas del Lenguaje GLIDER". (*Basic Ideas of GLIDER Language*) Mimeografiado, IEAC, Universidad de los Andes. Octubre de 1985. Mérida, Venezuela. pp. 37.

[6] Domingo, C. y Hernández, M. 1988 "GLIDER, A Network Oriented Simulation Language", 2nd Workshop on Computer Performance Evaluation, Milan. (May 30 - Jun 1).

[7] Domingo, C., M. Hernández, M. Sananes, J. Silva y G. Tonella. 1989. "GLIDER, A New Simulation Language", Intl. Congress of New Technology for the Development of Software and Supercomputers, Caracas, Venezuela. (Nov. 29- Dic. 1).

[8] Domingo C., G. Tonella, H. Hoeger, M. Hernández, M. Sananes y J. Silva. 1993. "Use of Object Oriented Programming Ideas in a New Simulation Language", in Schoen (ed.) Proceedings of Summer Computer Simulation Conference, Boston, 19-22 de Julio de 1993, pp 137-142.

[9] Domingo C., Quiroz S., Terán O.,1994. Sistema Estadístico para el Lenguaje de Simulación GLIDER. *(Statistical System for Simulation Language GLIDER)*, Meeting of Venezuelan Society of Biometric and Statistics, Caracas Julio 1994.

[10] Domingo C., Fargier M.E., Mora J.,Rojas A., Tonella G. 1994. "Modelo de la Economía de Venezuela basado en Conceptos Microeconómicos" *(Model of Venezuelan Economy based on Microeconomic Concepts)*. XII Latin America Meeting o Econometric Society, August 1994.

[11] Domingo C. 1994. "Modelo de Simulación de Flujo de Crudos y Puerto Petrolero usando el Lenguaje GLIDER" . (*Simulation Model of Oil Flow and Shipment using GLIDER Language*). IEAC, Universidad de los Andes, Mérida, Venezuela.

[12] Hernández, C. 1993. "Enseñanza Programada para el GLIDER". Trabajo Especial de Grado. (*Programmed Learning of GLIDER Language. Thesis*), Escuela de Ing. de Sistemas, Universidad de los Andes. Mérida, Venezuela. Tutor: G. Tonella.

[13] Jiménez, T. 1994. "Ambiente Gráfico para el Lenguaje de Simulación GLIDER, sobre el Sistema X-Windows". *(The Graphical Environment for the GLIDER Language, using the X-Windows System)*. Pasantía de Investigación, Escuela de Ing. de Sistemas, Universidad de los Andes, Tutor: M. Sananes.

[14] Palm, F. 1993. "Prototipo de Lenguaje GLIDER para Manejar Problemas de Valores Iniciales en Derivadas Parciales". Trabajo especial de grado. (*GLIDER Language Prototype to handle Initial Value Problems in Partial Differential Equations. Thesis*) Escuela de Ingeniería de Sistemas, Universidad de los Andes, Mérida, Venezuela. Tutors: G. Tonella and C.Domingo.

[15] Quiróz, S. 1992. "Diseños Experimentales en Simulación". Tesis de Maestría, (*Experimental Designs in Simulation. M.S. Thesis in Statistics*) IEAC, Universidad de los Andes, Mérida, Venezuela. Julio de 1992. Tutor: C. Domingo.

[16] Reinoza, M. 1992. "Análisis de Lenguajes de Simulación para Control y su Posible Adaptación al Lenguaje GLIDER". Trabajo especial de grado. (*Analysis of Control Simulation Languages and Adaptation to GLIDER Language. Thesis*) Escuela de Ingeniería de Sistemas, Universidad de los Andes, Mérida, Venezuela. Tutor: G. Tonella.

[17] Sananes, M. 1992. "Editor Gráfico del Lenguaje de Simulación GLIDER: Documento de Diseño". (*Graphic Editor of GLIDER Simulation Language. Design Report*) Publicación Cesimo IT-9202, Cesimo and IEAC, Universidad de los Andes, Mérida, Venezuela. Febrero de 1992.

[18] Sananes, M. 1992. "Instructivo del Uso del Editor Gráfico del Lenguaje de Simulación GLIDER". Cesimo and IEAC, (*Manual for Using of Graphic Editor of the GLIDER Simulation Language*) Universidad de los Andes, Mérida, Venezuela. Febrero de 1992.

[19] Terán O. 1994. "Simulación de Cambios Estructurales en Lenguaje GLIDER" . Tesis de Maestría en Estadística Aplicada. (*Simulation of Structural Changes in GLIDER Language. M.S. Thesis in Applied Statistics)*, IEAC, Universidad de los Andes, Mérida, Venezuela. Tutor C.Domingo.

[20] Tonella G. y C. Domingo. 1990. "GLIDER, a New Simulation Language" in K.G. Nock (ed.) Proceedings de UKSC Conference on Computer Simulation 1990, Brighton 5-7 de Septiembre de 1990. UKSC Publications, Burgess Hill, England. p 69-75.

[21] Tonella, G.; Acevedo M.; Sananes M.; Domingo C.; H. Hoeger; M. Ablan. "Simulation of Ecosystems with GLIDER: a Discrete-Continuous Simulation Language", in M. H. Hamza (ed.) Applied Modeling and Simulation 1994 Conference, Lugano Switzerland, June 20-22. Acta Press, Anaheim, California, pp. 20-23.

[22] Tonella, G.; Domingo, C.; Sananes, M.; Tucci, K. "El Lenguaje GLIDER y la Computación Orientada hacia Objeto". *(The GLIDER Language and the OO Programming)*, Asovac XLIII Convención Anual 14-19 de Noviembre de 1993, Mérida, Venezuela.

[23] Tucci, K. 1993. "GLIDER Orientado hacia Objetos en C++". Trabajo especial de grado. (*Object Oriented GLIDER Language in C++. Thesis*) Escuela de Ingeniería de Sistemas, Universidad de los Andes, Mérida, Venezuela. Tutor: G. Tonella.

[24] Uzcátegui, M. 1992. "Mecanismos de Manejo de Materiales en GLIDER Basados en SIMAN y SLAM". Trabajo especial de grado. (*Material Handling Mechanisms in GLIDER based on SIMAN and SLAM. Thesis*) Escuela de Ingeniería de Sistemas, Universidad de los Andes, Mérida, Venezuela. Tutor: G. Tonella.

[25] Velandia, O. 1990. "Simulación Discreta Orientada al Proceso Usando GLIDER". Trabajo especial de grado. (*Process Oriented Discrete Simulation using GLIDER. Thesis*) Escuela de Ingeniería de Sistemas, Universidad de los Andes, Mérida, Venezuela. Tutor: G. Tonella.

[26] Zambrano, T. 1992. "Modelo Preliminar de Simulación del Crecimiento en Area Basal para Plantaciones de Teca" Trabajo de Maestría en Ciencias Forestales. (*Preliminar Simulation Model of Basal Area Growth of Teak Plantations. M.S. Thesis in Forestry Science*) Universidad de los Andes, Mérida, Venezuela. Tutor: C. Domingo.