Preprints of IFIP Working Conference

on MODELLING OF ENVIRONMENTAL SYSTEMS

Tokyo, Japan.
April 26–28, 1976

Edited by
Tosiyasu L. Kunii
Yoichi Kaya

IFIP

Sponsored by
IFIP TC7 on Modelling and Optimization
A.V. Balakrishnan, Chairman
IFIP WG7.1 on Modelling and Simulation
W. J. Karplus, Chairman

Cosponsored by
Japan Society for Simulation Technology

In Corporation with
Information Processing Society of Japan
Institute of Electronics and Communication Engineers of Japan
Institute of Electrical Engineers of Japan
Japan Society of Mechanical Engineers

ted by
Japan, Ltd.

ON THE MODELING AND REPRESENTATION OF ABSTRACTIONS IN SIMULATION LANGUAGES*

Carlos J. LUCENA, Heitor M. QUINTELLA and Daniel SCHWABE

Departamento de Informática, Pontifícia Universidade Católica,
Rio de Janeiro, Brasil

The data abstractions required by simulation models can be handled at a higher level than the ones implied by current discrete simulation languages. This paper describes a language design approach which supports the modeling of abstract data types and a semi-automatic procedure for the selection of an efficient base machine on which the simulation program will perform.

1. INTRODUCTION

In this paper we are interested in programming languages aimed at modeling discrete systems which involve the interaction in time of a number of interrelated processes.

Processes can, for instance, be modeled by a sequence of discrete events, each of which is assumed to occur instantaneously in the time scale of the system being simulated. In an event oriented simulation, a program deals with the scheduling of events that operate on information structures [1] called entities. Events may change data values of entities, create and delete entities.

A considerable amount of work in the area of computer simulation has been based on extended FORTRAN-like and Algol-like programming languages such as SIMSCRIPT [3] and SIMULA-67 [2]. The design of the above languages was based on an attempt to make available a number of software engineering tools to non-specialists in computer science. In fact, besides the basic requirements of simulation languages for event (and/or activity) handling capabilities, the referred languages have provided the user with some powerful mechanisms to model the abstractions that appear in the realm of computer simulation (lists in [3] and classes in [2]).

SIMSCRIPT and SIMULA differ basically as a consequence of the differences in the structure of function modules in FORTRAN and ALGOL, respectively. Both languages provide mechanisms for creation and deletion of entities and for the representation of the successive stages of processes being simulated as they pass through the system. In SIMSCRIPT a simulation program can be thought of as a program executed by an interpreter (scheduler) whose basic function modules are events (the sequencing being performed by a scheduling algorithm). In SIMULA the concept of activity is introduced. An activity is partitioned into a number of active phases which correspond to events that occur instantaneously in system time. The phases are in turn separated by inactive phases, during which system time may elapse.

In the case of an event oriented simulation, events can be interrelated through a wide variety of access mechanisms. We will call an event data space plus the set of operations defined over this space a simulation language abstract data type (SLADT). Events, in turn, operate upon entities. Entities are structured objects that form a simulation language base machine (SLBM).

SIMSCRIPT provides one standard SLADT to model ordered sets of events. Through the SIMULA's class mechanism a user is capable of defining arbitrary SLADTs. Nevertheless, the language being based on ALGOL, a number of programming details are left to be handled by the user (particularly the problems of scope of variables). The SLBMs that implement entities are in both cases specified by the user through the restricted repertoire of data definition facilities provided by both languages (SIMSCRIPT at least provides for free and allocate mechanisms).

We propose a language design that supports the following features:

i. SLADTs are defined by the user, through an extension of the concept of class, called the cluster mechanism [7]. Algebras of events can be arbitrarily proposed by the user who is able to program in terms of the operations applicable to objects of the type defined by the algebra, completely disregarding the access mechanism that interconnect the events.

---

*This research was supported by the Brazilian Government Agency FINEP under contract no 244/CT.

ii. The SLBM that represents an event is transparent to the user. Moreover, the user needs only to know the basic general structure of the entities that occur in simulation programming since the most adequate representation of the basic structure may be picked up for him by an experienced programmer or an automatic programming system. The basic general structure of entities will be called a chain structure [4,8].

Our proposed language design is presently implemented as an extension of PL/I [6].

2. Event Structuring

The major features of the language design, which we claim presents engineering improvements over the SIMSCRIPT/SIMULA-type of language, will be presented here through a very simple example.

Suppose we want to encode in our programming language the following simulation flowchart (figure 1)

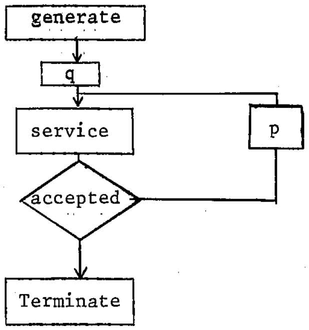


Figure 1. Simple simulation example

This is a model of a very simple single process manufacturing system with a well defined standard of quality for the end product. When they first arrive at the system, events queue in the usual manner to be processed (serviced). After service if they pass a quality control test they leave the system otherwise they are recycled into the system in a LIFO discipline with priority given to newly arrived events.

In the example q stands for a queue and p for a stack of events. An event at the top of p gets to be serviced only when q is empty. At the highest programming level (specification level) the user defines the algebras of events or SLADTs in terms of which he will encode his algorithm. A possible choice is the following:

a. (EVENT, {generate, is-end-of-service, update-time, time, Terminate, service})

The informal meaning of the operations are:

generate: generates an event (arrival);
is-end-of-service: verifies if the event is an end of service;
update-time: changes the event's arrival time (ready for activation);
time: consults the event's arrival time;
Terminate: terminates an event;
service: consults the event's service time;

b. (CFE, {insert, next, first})
CFE stands for calendar of future events and the operations over CFE have the usual meaning (insertion takes place according to arrival time).

c. (QUEUE_OF_EVENTS,{in, out, empty})

d. (STACK_OF_EVENTS, {push, pop, top, empty})

Given the above SLADTs, the flowchart in figure 1 can be coded as follows in our PL/I extension.

```
SIMULATION: PROC OPTIONS (MAIN);
 DCL  EVENT   ABSTRACT_TYPE;
 DCL  CFE     ABSTRACT_TYPE;
                        /*Calendar of Future Events*/
 DCL  QUEUE_OF_EVENTS ABSTRACT_TYPE;
 DCL  STACK_OF_EVENTS ABSTRACT_TYPE;

 DCL   T   EVENT();    /*Current event*/
 DCL   R   EVENT();    /*auxiliary variable*/
 DCL  TTABLE CEF();
 DCL   Q   QUEUE_OF_EVENTS();
 DCL   P   STACK_OF_EVENTS();
 DCL   STBUSY  BIT(1);  /*Indicates station's status*/
 DCL   (CLOCK,LATEST_ARRIVAL,TSIM) BIN FIXED;
 CALL  INITIALIZE         /*Initializes global*/
                                     /*variables*/
 EVENT@GENERATE(T,CLOCK)        /*1st event*/
 DO  WHILE(CLOCK <=  TSIM);     /*TSIM=Simulation*/
                                         /*time*/
   LATEST_ARRIVAL = LATEST_ARRIVAL + IATIME  /*IATIME
                    /*is a function  that randomly*/
                   /*generates inter_arrivaltimes*/

   EVENT@GENERATE(R,LATEST_ARRIVAL); /*generate the*/
                                    /*next arrival*/
   CFE@INSERT(TTABLE,.R);
   IF EVENT@IS_END_OF_SERVICE(T)
      THEN DO;          /*An end of service ocurred*/
           STBUSY='0'B;  /*set the station free for*/
                                           /*use*/
           IF REJECTED      /*REJECTED is a function*/
                                  /*that randomly*/
                           /*rejects events so that*/
                            /*they will be recycled*/
             THEN DO;
                  EVENT@UPDATE_TIME(T,CLOCK);
                  STACK@PUSH(P,.T);/*Places rejected*/
                                 /*event on stack*/
                  END:
             ELSE DO;
                /*gather statistics on service time*/
                  EVENT@TERMINATE(T);
                  END;
```

```
          /*GET the next eventy*/
          IF QUEUE@EMPTY(Q)
            THEN DO;
                 IF ¬STACK@EMPTY(P)      /*QUEUE is*/
                                     /*empty, so if*/
                   THEN DO;          /*stack is not*/
                                            /*empty*/
                        T=STACK@TOP(P)   /*service*/
                                /*the event at the*/
                                /*top of the stack*/
                        STACK@POP(P)
                        /*gather statistics on the*/
                       /*waiting time in the stack*/
                        END;
                   ELSE  DO;        /*the station is*/
                              /*idle for some time*/
                         T=CFE@NEXT(TTABLE, CLOCK);
                            /*gather statistics on*/
                               /*station idle time*/
                         END;
            ELSE DO;          /*queue is not empty, so*/
                             /*service the first event*/
                 T=QUEUE@OUT(Q);
                /*gather statistics on the waiting*/
                               /*time in the queue*/
                 EVENT@UPDATE_TIME(T,CLOCK);
                 END;
          END;                /*event end of service*/

   ELSE  DO ;                 /*An arrival has ocurred*/
          IF STBUSY      /*If the station is busy,*/
            THEN  QUEUE@IN(Q,.T)    /*then put the*/
                               /*event in the queue*/
            ELSE  DO;         /*else service the event*/
                  EVENT@SERVICE(T);
                  CEF@INSERT(TTABLE,.T) /*generate*/
                               /*an end of service*/
                  END;
          T = CFE@NEXT(TTABLE,CLOCK);      /*get the*/
                                      /*next event*/
          END;
 END;     /*WHILE*/
 /* Print results  */
END SIMULATION;
```

The notation CFE@INSERT(TTABLE,.R) is used to indicate that the value of R, which has an abstract type (EVENT), is to be inserted in TTABLE which is of type CFE.

After the problem has been encoded at this very high level, the user is required to make explicit the access paths that structure the abstractions about events. While doing that he is required to make no assumption about the SLBM other than the fact that it is the implementation of a chain structure.

3. The Simulation Language Base Machine (SLBM).

Most simulation languages operate upon a base machine which is said to have a chain structure. The chain structure concept can be formalized as follows.

Let $\mathcal{E}$ be a set. We will call s the totally ordered set $E \subseteq \mathcal{E}$ which stands for the state of a chain structure. We will denote $E(s)$ the "underlying subset" of s. $\bar{E}(s)$ stands for its complement.

Let S be a set of states. We will call m, for modifier, an application of S over S. M is a set of modifiers.

A chain structure is a pair (S,M) which follows the axioms below:

a. $\exists \{ \{\emptyset\}, \leq \} \in S$ such that $E(s)=\{\emptyset\}$

b. $I \in M$ (I being the identity application)

c. $s \in S$ there exists a sequence $(m_i)_{i \in N}$, $m_i \in M$, such that:
$s = \ldots m_n \circ m_{n-1} \circ \ldots \circ m_1 (\emptyset)$

d. $\forall (s,m) \varepsilon S \times M$, the orders induced by s and $m(s)$ over $E(s) \cap E(m(s))$ are identical

e. $\forall (s,m) \in S \times M$, $E(s) \cap \bar{E}(m(s))$ and $\bar{E}(S) \cap E(m(S))$ are finite.

A chain structure calls only for two fundamental operations (modifiers) to handle its required transformations. The semantic of these modifiers can be stated formally as follows.

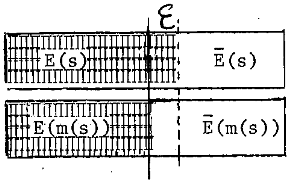
The "underlying subset" of a state s that has been modified by a modifier m is given by the following expression:

$$E(m(s)) = E(s) \circ m^{-}(s) \circ m^{+}(s)$$

where

$$m^{-}(s) = - E(s) \cap \bar{E}(m(s)) \quad \text{if } s \in E(s)$$

$$m^{-}(s) = I \quad \text{if } s \notin E(s)$$



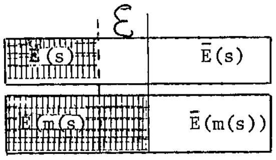
to remove elements
$\bar{E}(s) \cap E(m(s)) = \emptyset$

Fig.2 - illustration of element removal.

and

$$m^{+}(s) = \bar{E}(s) \cap E(m(s)) \quad \text{if } s \in \bar{E}(s)$$

$$m^{+}(s) = I \quad \text{if } s \notin \bar{E}(s)$$



to insert elements
$E(s) \cap \bar{E}(m(s)) = \emptyset$

Fig. 3. illustration of element insertion.

Clearly in this particular formulation it is possible to remove more than one element per modification.

Conveniency dictated that we supplied a larger set of modifiers (although acknowledging the redundancy involved) to operate on representations (Reps) of chain structures. These modifiers are the following:

ADD: adds an element to the rep.
SUB: subtracts an element.
SELECT: selects an element.
INSERT: inserts a new element.
REPLACE: replaces an old element.
LINK: links two sub-structures.
DETACH: detaches two sub-structures.
COPY: generates a copy of the structure.
REMOVE: removes an element.
LENGTH: provides the cardinality of the representation.

A cluster [7] that defines the SLBM or representation level of a chain structure has the following general form:

```
repr: REP (parameter list) USES < template > ;
  [ declaration of global (to the cluster)variables]
CREATE;
  [ create body ]
ENDCREATE;
ADD: PROC (parameter list);
  [ declaration of local variables ]
  [ body of standard operation ADD ]
END ADD;
  :
SUB: PROC...
  :
SELECT: PROC...
  :
END repr;
```

In the above linguistic level the programmer that will build up the user's library of SLBMs can make use of full-PL/I. The symbol < template > stands for the PL/I data types used to implement the representation . The CREATE block initializes the representation. At the access path level of programming, the declaration of any variable as being of type REP causes the activation of the corresponding CREATE block of the representation cluster. In the appendix we display a pair of SLBMs that can be used to ultimately implement a SLADT.

4. Specification of the Access Structure of a Data Abstraction.

The access structure of a SLADT can now be described by an access mechanism that acknowledges both the intended meaning of the abstractions used at the specification level and the fact that programming has to be made to a SLBM that is an implementation of a chain structure. In what follows we illustrate the definition of the access structure of some of the operations defined for the EVENT and CFE data abstractions.

```
EVENT: CLUSTER() ON REP1 IS GENERATE (REP,BIN FIXED),
       IS_END_OF_SERVICE(REP) RETURNS (BIT(1)),
       UPDATE_TIME(REP,BIN FIXED), TIME(REP) RETURNS
       (BIN FIXED), TERMINATE(REP), SERVICE(REP),
       SERVICE_TIME(REP) RETURNS(BIN FIXED);
       TEMPLATE  1  EV BASED (PT_E),
                    2  ARR_TIME   BIN FIXED,
                    2  IS_ARRIVAL BIT(1),
                    2  SERV_TIME  BIN FIXED;

  CREATE;
    DCL E REP1(1);      /*E Stands for the generic*/
                                  /*chain used*/
  END CREATE:      /*which, in this case, has only*/
                                  /*one element*/

  GENERATE: PROC(CLOCK);    /*generate a new event*/
    DCL CLOCK BIN FIXED;
    ALLOCATE EV;
    ARR_TIME = CLOCK;
    IS_ARRIVAL = '1'B;
    SERV_TIME =  SERVICE_TIME; /*SERVICE_TIME is a*/
                             /*function that randomly*/
                        /*generates the service time*/
    REP@REPLACE(E,1,PT_E);        /*assign the newly*/
                               /*created event to the*/
                                    /*representation*/
    RETURN;
END GENERATE;
IS_END_OF_SERVICE: PROC RETURNS BIT(1);/*Test the */
                                /*tipe of the event*/
    PT_E = REP@SELECT(E,1);
    RETURN (¬IS_ARRIVAL);
END  IS_END_OF_SERVICE;
    UPDTE-TIME : PROC...
        :
    END UPDATE_TIME;
        :
    SERVICE : PROC;         /*services an event*/
        PT_E = REP@SELECT(E,1);
        IS_ARRIVAL = '0' B;       /*changes the status*/
                                  /*to end of service*/
        ARR_TIME = ARR_TIME + SERV_TIME; /*updates*/
                                         /*the time*/
    END SERVICE;
END EVENT;

CFE: CLUSTER() ON REP1 IS INSERT(REP,REP), NEXT (REP
     BIN FIXED)  RETURNS(REP), FIRST(REP) RETURNS
     (REP);
     DCL  EVENT  ABSTRACT_TYPE;
     DCL  E1     EVENT();
     CREATE;
      DCL  C  REP1(0);     /*C stands for the chain*/
                                   /*used in the*/
    END CREATE;  /*representation of the calendar*/
     INSERT: PROC(.E);   /*The notation .E implies*/
                  /*in E being of ABSTRACT_TYPE */
      DCL E EVENT();
      E1 = REP@SELECT(C,1);
      DO I = 1 TO REP@LENGTH(C)/*searches the chain*/
        /*for the appropriate place of insertion  */
        WHILE(EVENT@TIME(E1)<EVENT@TIME(E));

           E1 = REP@SELECT(C,I);
      END;
      IF(EVENT@TIME(E1)<EVENT@TIME(E))|
        (EVENT@TIME(E1)=EVENT@TIME(E)&
         ¬EVENT@IS_END_OF_SERVICE(E))
           /*In case two events have same arrival */
          /*time, end of services are placed first*/
        THEN REP@INSERT(C,I-1,'+', E);
        ELSE REP@INSERT(C,I-1,'-', E);
    END INSERT;
    FIRST:PROC RETURNS(REP); /*consults the first*/
         E1=REP@SELECT(C,1);            /*calendar*/
         RETURN(E1);
    END  FIRST;
         NEXT;PROC(CLOCK) RETURNS(REP); /*obtains*/
    DCL CLOCK BIN FIXED;/*the next element in the*/
         E1 = REP@SELECT(C,1); /*calendar,removing*/
         REP@SUB(C,'-') ;                     /*it*/
         CLOCK = EVENT@TIME(E1) ;
         RETURN(E1) ;
         NEXT;
    END NEXT;
   END  CFE;
```

5. Conclusions

Through the multi-level cluster approach a programmer is able to encode his simulation program in two phases: first a very high level statement of the program is provided by using data abstractions involving events, later access path structures are defined for the abstractions which refers to a base machine called a rep. The rep defines the base machine on which the program will operate. The rep definition is transparent to the user and stands for any correct implementation of the chain structure concept (see the appendix). An experienced programmer or an algorithm (as proposed in [9]) can be used in the process of selecting the most efficient rep for a particular simulation program structure. Through the described procedure some usually long and time consuming simulation programs can be made very efficient, their implementation being derived through a synthesis approach that is amenable to systematic correctness checks.

6. References

[1] Wegner, P., "Programming Languages, Information Structures and Machine Organization", McGraw-Hill, 1968.

[2] Dahl, O.J.; Myhrhaug, B.; Nygaard, K., SIMULA 67 Common Base Language, Oslo, Norwegian Computer Centre, 1968.

[3] Dimsdale, B.; Markowitz, H.,"A description of the Simscript Language, IBM Systems Journal, vol. 3 nº 1, 1964.

[4] Ehrmann, R., "Les Languages de Simulation", Dunod, 1971.

[5] Lucena, C.; Schwabe, D.; Berry, D., "Issues in Data Type Construction Facilities", submmited for publication (Technical Report of the Deptº de Informática, Pontifícia Universidade Católica do Rio de Janeiro).

[6] Schwabe, D.; Lucena, C., "Specification and Uniform Reference to Data Structures in PL/I", submitted for publication (Technical Report of the Deptº de Informática, Pontifícia Universidade Católica do Rio de Janeiro).

[7] Liskov, B.; Zilles, S., "Programming with Abstract Data Types", SIGPLAN Symposium on Very High Level Languages, March, 1974.

[8] Quintella H.M., "Fundamentos da Teoria de Cadeias para Uso em Simulação". (Technical Report of the Deptº de Informática, Pontifícia Universidade Católica do Rio de Janeiro) in print and is to appear also in English.

[9] Low, J.R., "Automatic Coding: Choice of Data Structures", Technical Report STAN-CS-74.452, Stanford University, 1974.

Appendix

Two Examples of SLBMs

The programs below implement an array and a linked list representation of a chain structure. Note that the representations can be interchanceably used as a base machine. The base machine is referenced within the cluster that specifies the access mechanism chosen for the data abstraction.

```
 ARRAY: REP USES
   1 A BASED(DUMMY),
     2 UB BIN FIXED,
     2 V (UPB  REFER(A.UB)) PTR;
  DCL UPB BIN FIXED,PT_A PTR ;
  CREATE(LEN) ;
    UPB = LEN ;
    ALLOCATE A ;
    PT_A = DUMMY ;
  ENDCREATE ;
  ADD : PROC(POS,VAL) ;
     DCL POS CHAR(1) , VAL PTR, OLD PTR ;
     OLD,DUMMY = PT_A ;
     UPB = A.UB + 1 ;
     ALLOCATE A ;
     IF POS = '-'
       THEN DO ;
            DO I = 1 TO UPB - 1 ;
              A.V(I+1) = OLD -> A.V(I) ;
            END ;
            A.V(1) = VAL ;
            END ;
       ELSE DO ;
            DO I = 1 TO UPB ;
              A.V(I) = OLD -> A.V(I) ;
            END ;
            A.V(UPB) = VAL ;
            END ;
     FREE OLD -> A ;
     PT_A = DUMMY ;
  END ADD ;
 SUB : PROC(POS) ;
    DCL POS CHAR(1), OLD PTR ;
    OLD,DUMMY = PT_A ;
    UPB = A.UB - 1 ;
    ALLOCATE A ;
    IF POS = '-'
      THEN DO I = 1 TO UPB+1 ;
             A.V(I-1) = OLD -> A.V(I) ;
           END ;
      ELSE DO I = 1 TO UPB ;
             A.V(I) = OLD -> A.V(I) ;
           END ;
    FREE OLD -> A ;
    PT_A = DUMMY ;
 END SUB ;
 SELECT : PROC(POS) RETURNS(PTR) ;
   DCL POS BIN FIXED, OLD PTR ;
   OLD = NULL ;
   IF POS > PT_A -> A,UB THEN RETURN(OLD) ;
   OLD = PT_A->A.V(POS);
   RETURN(OLD) ;
 END SELECT ;
 LENGTH: PROC RETURNS(BIN FIXED) ;
   RETURN(UPB) ;
  END LENGTH ;
END ARRAY ;
```

```
LIST  : REP USES
        1  NODO BASED(PT),
           2  VALOR  PTR,
           2  PROX   POINTER ;
 DCL (HEAD,ULT) POINTER, TAM BIN FIXED ;
 CREATE(PARM) ;
 ALLOCATE NODO SET(PT) ;
 NODO.VALOR = NULL ;
 NODO.PROX  = NULL ;
 ULT        = PT   ;
 HEAD       = PT   ;
 TAM        = 0 ;
ENDCREATE ;
SELECT : PROC(I) RETURNS(PTR) ;
 DCL (I,J) BIN FIXED, (K,M) POINTER ;
 IF I > TAM   THEN RETURN(NULL) ;
 K = HEAD -> NODO.PROX ;
 DO J = 1 TO I WHILE(K¬=NULL) ;
   M = K ;
   K = K -> NODO.PROX ;
 END ;
 IF J ¬= I+1 THEN DO ;
                PUT SKIP LIST('ERROR') ;
                STOP ;
                END ;
           RETURN(M->NODO.VALOR) ;
 END SELECT ;
 ADD : PROC(POS,ELEM) ;
    DCL POS CHAR(1), ELEM PTR, PT POINTER ;
    ALLOCATE NODO SET(PT) ;
    PT -> NODO.VALOR = ELEM ;
    PT -> NODO.PROX  = NULL ;
    IF POS = '+'
      THEN DO ;
           ULT -> NODO.PROX = PT ;
           ULT = PT ;
           END ;
      ELSE IF POS = '-'
          THEN DO ;
               PT->NODO.PROX = HEAD ;
               HEAD = PT ;
               END ;
          ELSE PUT SKIP LIST('ERROR ') ;
     TAM = TAM + 1 ;
     RETURN ;
  END ADD ;
  SUB : PROC(POS) ;
     DCL POS CHAR(1) ; PT POINTER ;

     IF POS = '-'
       THEN DO ;
            PT = HEAD ;
            HEAD = HEAD -> NODO.PROX ;
            FREE PT->NODO ;
            END ;
       ELSE IF POS = '+'
              THEN DO ;
                   PT = HEAD ;
                   DO II = 1 TO TAM - 1 ;
                   PT = PT -> NODO.PROX ;
                   END ;
                   FREE ULT->NODO ;
                   ULT = PT ;
                   END ;
              ELSE PUT SKIP LIST('ERROR ') ;
   TAM = TAM - 1 ;
  END SUB ;
  LENGTH : PROC RETURNS(BIN FIXED) ;
    RETURN(TAM) ;
  END LENGTH ;
END LIST ;
```