SEMANTICS AND IMPLEMENTATION OF PREFIXING AT MANY LEVELS[+]

W.M. Bartol
A. Kreczmar
A.I. Litwiniuk
H. Oktaba

Institute of Informatics
University of Warsaw
00-901 Warsaw, P.K.i N. Poland

Abstract

A generalization of Simula's prefixing of classes is presented. The notion of one-level prefixing is first introduced by means of the example of Simula 67; the semantics of a programming language with prefixing at many levels is then discussed and analyzed. The principles for efficiently implementing programming languages with prefixing of classes at many levels are described. A generalized display mechanism is introduced and the correctness of a display update algorithm is proved. A new data structure for efficient identification of dynamic objects is also presented.

Keywords: block structured programming languages, classes, prefixing, methods of implementation, Simula 67.

[+]This research was supported in part by "Zjednoczenie MERA" of Poland.

1. Introduction

The prefixing of classes is one of the most attractive and powerful mechanisms incorporated into the programming language Simula 67 (cf [4]). This tool allows a programmer to design a program in a structural, abstract way. To present briefly the main ideas of prefixing we start with the notion of a class.

Let us consider the following scheme of class declaration:

```
class A;
   attributes a1,...,an;
   I1;...;Ip; inner; Ip+1;...;Ir
end A;
```

where $a_1,\ldots,a_n$ are attributes (variables or, perhaps, other syntactic units like classes, procedures, functions etc.) and $I_1,\ldots,I_p$, $I_{p+1},\ldots,I_r$ are instructions of the class A. With the help of an object generator ("new A") one can create an object of the class A, i.e. create a frame (activation record) in the memory for attributes $a_1,\ldots,a_n$ and execute the instruction list $I_1,\ldots,I_p,I_{p+1},\ldots,I_r$. When control returns to the object where the expression "new A" has been executed, the frame is not deallocated and a reference to that frame is transmitted as the value of the expression "new A". Hence, a reference to the object may be retained in a reference variable (e.g.X:-new A, where X is a reference variable qualified by the class A).

The attributes of objects are accessible from outside as well as from inside the object. Remote accessing (e.g.$X.a_i$) allows one to use the attributes $a_1,\ldots,a_n$ from outside. Internal access occurs while executing the instructions of the object of A and any unit nested within it or during calls of the class's procedure attributes.

Consider now declaration scheme of a class B:

```
A class B;
     attributes b1,...,bm;
     J1;...;Js; inner; Js+1;...;Jt
end B;
```

Class B is prefixed by A, i.e. B has attributes $a_1,\ldots,a_n,b_1,\ldots,b_m$ and the instruction list $I_1,\ldots,I_p,J_1,\ldots,J_s,J_{s+1},\ldots,J_t,I_{p+1},\ldots,I_r$ and B is called a subclass of A. One can create an object of class B in a similar way as was done for A, i.e. by Y:-new B. Here Y may be a reference variable qualified by class B as well as by class A (for the general rules of this kind of assignment statement see [4]).

The following class C is a subclass of the classes B and A:

```
B class C;
    attributes c1,...,ck;
    K1;...;Ku; inner; Ku+1;...;Kv
end C;
```

and it has the attributes $a_1,\ldots,a_n,b_1,\ldots,b_m,c_1,\ldots,c_k$ and the instruction list $I_1,\ldots,I_p,J_1,\ldots,J_s,K_1,\ldots,K_u,K_{u+1},\ldots,K_v,J_{s+1},\ldots,J_t,I_{p+1},\ldots,I_r$. The sequence of classes A,B,C is called the prefix sequence of the class C. Class C may in turn be used as a prefix of some other class, and so forth, but no class can occur in its own prefix sequence. Hence prefixing has a tree structure.

Blocks may also be prefixed. For instance, a block:

```
A begin
    attributes c1,...,ck;
    K1;...;Ku
  end
```

is prefixed by the class A, i.e. it has the attributes $a_1,\ldots,a_n$, $c_1,\ldots,c_k$ and the instruction list $I_1,\ldots,I_p,K_1,\ldots,K_u,I_{p+1},\ldots,I_r$.

In Simula 67, perhaps because of the method chosen for the original implementation, there is an important restriction on prefixing; namely, a class may be used as a prefix only at the block level at which it has been declared. Before we explain the reasons for this restriction and possible ways of abolishing it, let us look at some examples which illustrate the difficulties arising from this restriction.

Suppose we have a declaration of a class PQ which provides the data structure of a priority queue of integers with maximal capacity defined by an input parametr n:

```
class PQ(n); integer n;
begin
```

```
   integer procedure deletemin;
   ...
   end deletemin;
   procedure insert(x); integer x;
   ...
   end insert;
   ...
end PQ;
```

In the following program:

```
begin
   class PQ(n); integer n;
   ...
   end PQ;
   ...
   begin integer n;
   read(n);
   PQ(n) begin
         ...
         end
   end
end
```

the declaration of PQ is not at the same level as the prefixed block, hence this construction is incorrect in Simula 67.

If the class PQ were translated separately and treated as being declared in the block at level 0, it would never be possible to use this data structure as a prefix in other block except the outermost one.

In Simula 67 this problem has been partially solved, because system classes like SIMSET and SIMULATION may be used at any level. But the user is not able to extend the library of system classes, which still forces him to rewrite the declarations at relevant block levels.

This situation becomes even more cumbersome if we want to make use of two data structures simultaneously and both of them are subclasses of one class. Consider for instance, the data structures A and B using lists as an auxiliary data system. Hence they ought to be subclasses of a class LIST. We have the following declarations:

```
class LIST;
...
```

```
end LIST;
LIST class A;
     ...
     end A;
LIST class B;
     ...
     end B;
```

and now we would like to open two prefixed blocks:

```
A begin
  ...
  B begin
    ...
    end
  end
```

Because of the restriction one must redeclare classes B and LIST at the level where B is used as prefix. Thus, redundancy is unavoidable.

Observe that with the possibility of separate translation and allowing prefixing at many levels we can develop software in a structural way. Any system or user class may be easily extended by the user and attached to the catalog of system classes without the necessity of recompiling already compiled units and without the redundancy of the program text. Moreover, as we showed before, the user is able to make use of arbitrary data structures simultaneously by means of a prefixing mechanism instead of remote accessing (what speeds-up run-time of a program and clarifies its source code).

To conclude, we emphasize that prefixing at many levels is not merely a sophisticated technical problem in programming languages, but an essential step forward in developing an effective software methodology.

The structure of the paper is the following. In section 2 we give an informal insight, illustrated by examples, into some important semantic questions concerning many-level prefixing. Section 3 contains definitions and facts concerning the block structured programming languages, which are well known but necessary. Section 4 contains the formal definition of access to attributes in one-level prefixing (Simula 67). In section 5 we prove that the proposed semantics of the rules for many-level prefixing is correct. Section 6 gives a description of addressing algorithms for many-level prefixing. In particu-

lar, a generalized display mechanism is introduced, a mechanism which realizes an efficient access to attributes. In section 7 we discuss the various strategies of storage management and their impact on the semantics of the proposed construct.

2. Many-level prefixing (informal presentation)

The prefixing in Simula-67 is subject to an important restriction: a class may be used as a prefix only at the syntactic level of its declaration. Hereafter we shall call this prefixing "at one level".

In this paper we consider a Simula-like language, in which there is no such restriction and "many-level" prefixing is possible i.e. a class may be used as a prefix whenever its declaration is visible. To speak about such a language we must be able first to determine its semantics. One might think that prefixing "at many levels" is a trivial generalization of prefixing "at one level", but this is not the case.

The semantics of such a language is not obvious: in particular the rules defining access to object attributes cannot be deduced from the analogous Simula rules.

Consider the following program scheme (we follow Simula syntax):

```
L1: begin
    class A; begin real x;
              .
              .
              .
             end A;
    L2: A begin real y;
        class B; begin
                  .
                  .
                  x:=y;
                  .
                  .
                  end B;
        .
        .
        .
        new B;
```

```
        .
        .
        L3: A begin real y;
            B class C; begin
                       .
                       .
                       .
                       y:=x;
                       .
                       .
                       .
                       end C;
            .
            .
            .
            new C;
            .
            .
            .
            end
    end
end;
```

This program has the following block structure: the class A is declared in the outermost block of the program. It prefixes two blocks (one contained in the other) labelled L2 and L3, respectively. Note that the use of the same prefix for two blocks - one nested in the other - is not allowed in Simula-67.

The first prefixed block contains the declaration of a class B, while the second contains the declaration of a class C prefixed by B.

Let us consider the structure of objects created during the execution of the program. Every object of a prefixed class or block contains all attributes belonging to classes from their prefix sequences. In the above program the first object is created upon entry to the block labelled L1. Denote this object by p1. The second, denoted by p2, is created upon entry to the block labelled L2. This object contains two local real variables: x and y. The execution of the statement new B yields a third object (denoted by p3) corresponding to the class B. As indicated in the program scheme, variables x and y occur in the statements of B. Both variables denote attributes of the

object p2.

Upon entry to the block L3 a new object p4 containing two variables x and y is created. The execution of the statement new C yields a new object p5 (see Fig.1) of the class C.

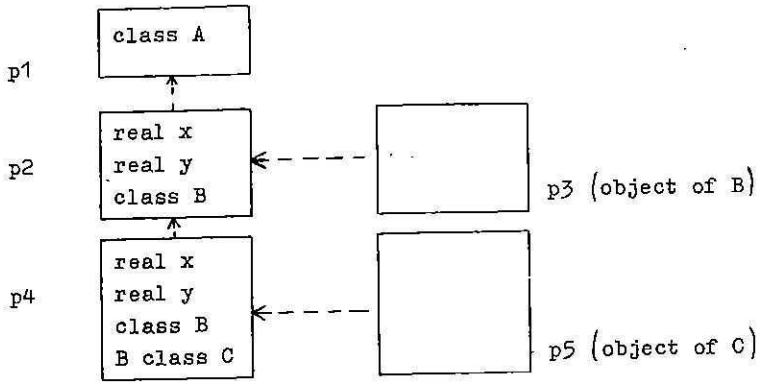


Fig.1.

According to the definition of prefixing the instruction list of C contains the instruction list of B. Therefore we must determine for each occurrence of the variables x and y in the instruction list of C the object from which the appropriate attribute is taken.

Consider first the statement y:=x in the body of C of the object p5. Note that none of the occurrences of the variables x,y is local in C. The object p5 belongs to class C and the nearest block containing the attributes x,y and the declaration of C is the block L3. Hence, both variables denote attributes of the object p4, which represents the block L3.

There are, however, different ways of defining the semantics of the statement x:=y from the class B of the object p5. The semantics of the statement can be based on a purely textual concatenation of the bodies of classes, as in Simula-67. We treat the declaration of class C as if it were concatenated with class B and declared in the block L3. Therefore both variables denote attributes of the object p4.

The semantics of the statement can be also defined in the following way: the syntactic unit to which the variable x is related is the class A, since A is the class in which x is declared; the syntactic unit to which the variable y is related is block L2. During the execution of statements in the object p5 the sequence of objects statically enclosing p5 is the following: p4,p3,p2,p1. In this sequence

p4 is the first object having attributes of the class A. Hence the variable x denotes an attribute of p4. The first object representing block L2 is the object p2, therefore the variable y denotes an attribute of p2.

From the above example it follows that there are some alternative ways of defining the semantics of assignment statement x:=y executed in p5.

In this paper we chose the one described above as the second, and we present its precise and formal definition in Section 5.

Why is this way of defining the semantics preferable?
There are several reasons for this choice. The most important is that we are able to define it in a precise and formal way and we are able to implement it efficiently (cf Section 6).
In the semantics based on a purely textual concatenation we see no way of addressing attributes which would depend only on the place of variable declaration. In particular we are not able to assign a relative displacement (offset) to an identifier occurring in a class statement. Note that an identifier may relate to attributes with different relative displacements depending on the place where a class is used. Compare with the example: in the statement x:=y of B the variable y relates to an attribute of p2 or p4 and these attributes may have different relative displacements. To illustrate the chosen semantics let us consider the program scheme structurally analogous to the example of Section 1.

```
begin
   class LIST;
     begin
     ref (...)head;
     procedure into(...);... head:=...; end;
     .
     .
     .
     end LIST;
    LIST class QUEUE;
     begin
     procedure intoqueue; ... into(...) ...; ... end;
     .
     .
     .
     end QUEUE;
```

```
   LIST class DECK;
     begin
     procedure intodeck; ... into(...) ...; ... end;
     .
     .
     .
     end DECK;
L1:  QUEUE begin
     L2: DECK begin
               .
               .
               .
               S1: intoqueue;
               S2: intodeck;
               end;
          end;
end
```

The above program contains declarations of classes: LIST, QUEUE, DECK. The class LIST describes the general structure of lists and contains the declaration of the variable "head" and the procedure "into", where that variable is used.

The classes QUEUE and DECK use the structure of LIST to describe the structures of queues and decks. In particular, they call the procedure "into" declared in LIST, and they use the variable "head" as its local attribute.

If we want to use both classes: QUEUE and DECK in a program, we may need two blocks prefixed by QUEUE and DECK, respectively. Moreover we wish the procedure "into" called in the body of "intoqueue" to be taken from the object representing the block prefixed by QUEUE; similarly, this procedure when called in the body of "intodeck" is to be taken from the object representing the block prefixed by DECK. Otherwise they should use the same attribute "head", which might destroy completely the proper execution of the program.

Denote the object created upon entry to the outermost block by p1. Objects created upon entries to blocks L1 and L2 will be denoted by p2 and p3, respectively.

The call of the procedure "intoqueue" (statement S1) yields a new object denoted by p4. The procedure "intoqueue" is an attribute of p2, so that the sequence of objects which statically enclose p4 is as follows: p4, p2, p1. The procedure "intoqueue" calls in turn the

procedure "into", which is declared in the class LIST: The first object in the sequence p4, p2, p1 which contains attributes of LIST is p2, thus in our semantics "into" is an attribute of p2 and "head" will be taken from p2. Analogous reasoning shows that the procedure "into" when called in the body of "intodeck" is an attribute of p3.

Thus the discussion shows that such informally presented semantics suits our purposes. In the subsequent sections the precise definition of this semantics and its implementation will be given.

3. Syntactic environment in programming languages without prefixing Static containers

Consider first the case of a programming language with block structure and without prefixing. By a syntactic unit in such a language we shall mean a block or a procedure. Arbitrary syntactic units will be denoted by U, V, W with indices or dashes, if necessary.

From the point of view of its block structure, any program may be treated as a tree T. The root of this tree $R(T)$ is the outermost block and for $U,V \in T$, U is the father of V iff V is declared in U (in definition blocks are treated as declarations in units where they appear). For the sake of simplicity of notation we shall write V decl U when V is declared in U (or alternatively, when U is the father of V in T).

Let decl^+ denote the transitive closure of the relation decl and let decl^* denote the transitive and reflexive closure of decl. So we have, in particular, U decl^* U and U decl^* $R(T)$ for any U.

The level of a node in a tree T is introduced as usual, i.e. $\text{level}(R(T)) = 1$ and $\text{level}(U) = \text{level}(V)+1$ if V decl U.

Any variable and any syntactic unit except a block has a name, called an identifier, introduced at the moment of its declaration. The identifier is then used to represent the variable or the unit in a program. The question of distinction between identifiers and syntactic entities (variables and syntactic units) is essential, because the same identifier may be introduced by different declarations in the program text.

Let id denote an arbitrary identifier. We consider now an occurrence of an identifier id in a statement of a program. Since a declaration associates an identifier with a syntactic entity, for the occurrence of id one must determine a unit U such that a syntactic entity named id is declared in U. For the semantics of a program to be

unambiguous, the correspondence between occurrences of identifiers and syntactic entities should be unique, i.e. only one syntactic entity may be associated with the given occurrence of an identifier id. Let us assume that id occurs in a unit V, i.e. V is the innermost unit containing the considered occurrence of id. In the following definition we make precise what is meant by scope of declarations or visibility rules.

Definition 3.1.
By a static container of the occurrence of an identifier id in a unit V, denoted by SC(id,V), we mean a syntactic unit U such that
(a) id is declared in U,
(b) V decl^* U,
(c) there is no unit U' such that V decl^* U' and U' decl^+ U and id is declared in U' (i.e. U is the innermost unit enclosing V such that id is declared in U).

If SC(id,V) does not exist, i.e. if there is no U such that (a) and (b) hold then of course the program is incorrect. Otherwise we say that the occurrence of id is local in V if V=SC(id,V), and non-local in V if V≠SC(id,V).

Dynamic containers

During a program's execution we can deal at the same time with many objects of the same syntactic unit, hence a computation of any instruction in an object requires identification and access to all the syntactic entities that it uses. In Algol-60 instances of blocks and procedures may be treated as the examples of objects, (in Simula-67 this is augmented with the objects of classes). The collection of objects of a syntactic unit U will be denoted by $|U|$. The objects themselves will be denoted by small latin letters p,q,r with indices, if necessary.

Consider an object $p \in |U|$. If the occurrence of an identifier id is local in a unit U, then the syntactic entity identified by id is situated within the object p. Hence there is no problem either with identification or with access to this syntactic entity. In general, however, for any id such that SC(id,U) exists, we must determine a unique object q such that $q \in |SC(id,U)|$. Then during the execution of the instruction list of U in the object p, the syntactic entity identified by id will be taken from q. Such an object q will be called a dynamic container of id with respect to p, and will be denoted by DC(id,p). Dynamic containers are unequivocally determined by means

of static links.

Upon a unit U is entered an object of this unit is allocated and initialized. It contains some system pointers in addition to declared attributes, for example the dynamic link (DL) which points to the calling object and the static link (SL) pointing to the object which is its syntactic father. We shall write p.SL=q when SL link of the object p points to the object q. (If p.SL is not defined, then we shall write p.SL=none.)

An object q is called the syntactic father of an object p, since q must be the object of a unit V where U is declared, i.e. if p.SL=q, $p \in |U|$, $q \in |V|$, then U decl V.

A sequence $p_k,\ldots,p_1$ of objects is called the SL chain of the object p_k, if p_1. SL=none and p_i. SL=p_{i-1} for i=k,...,2. The SL chain of an object p will be denoted by SL(p).

The SL chains define completely and uniquely syntactic environment of objects. This follows from the well-known results quoted below:

Lemma 3.1.

(a) If $SL(p_k) = p_k,\ldots,p_1$ and $p_i \in |U_i|$ for i=k,...,1, then the sequence $U_k,\ldots,U_1$ is a path from U_k to R(T) in the tree T,

(b) Let $SL(p_k) = p_k,\ldots,p_1$ and $p_k \in |V|$. If SC(id,V) exists, then there is a unique i, $1 \leqslant i \leqslant k$, such that $p_i \in |SC(id,V)|$. □

Lemma 3.1 (b) shows that the SL chain of an object defines completely and uniquely its syntactic environment. All syntactic entities which can be used in V are uniquely situated in $SL(p_k)$. Consequently the dynamic container $DC(id,p_k)$ of the occurrence of id with respect to the object p_k is defined as a unique object p_i belonging to $SL(p_k)$ such that $p_i \in |SC(id,V)|$.

The way SL links are defined during a program's execution induces the semantics of identifiers. The following algorithm determines exactly what should be done with SL links in order to obtain the most natural semantics (cf [7]).

Algorithm 3.1.

We can assume the only one object of the outermost block R(T) may be entered and, of course, for that object SL= none. Consider now the call of a unit U in an object $r_k \in |V|$. If id identifies U, then according to the definition 3.1 U is declared in SC(id,V). The syntactic father of $p \in |U|$ must be the object of the unit SC(id,V), i.e. the

unit where U is declared. Let $SL(r_k) = r_k,\ldots,r_1$. By Lemma 3.1 (b) there is a unique i, $1 \leq i \leq k$, such that $r_i \in |SC(id,V)|$. Then define $p.SL=r_i$, i.e. r_i becomes the syntactic father of p. (cf Fig.2). □

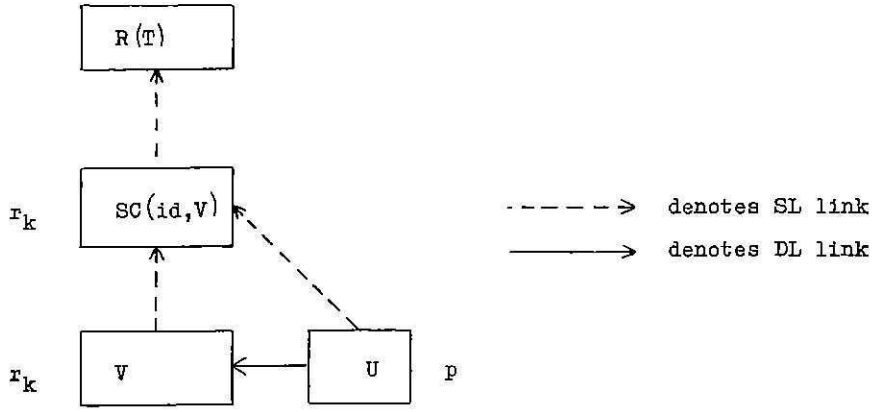


Fig.2.

4. Prefixing at one level

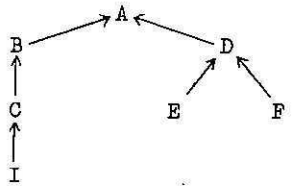
Prefix structure of a program

In this section we shall consider a programming language with block structure and one-level prefixing, i.e. exactly the case of Simula 67.

From the point of view of its prefix structure, any program may be treated as a forest of prefix trees $\{P_i\}$. Each prefix structure of a program is a tree P_i where for $U,V \in P_i$, U is the father of V iff U is the prefix of V and the root of P_i is a unique element of P_i without any prefix. Similarly to the relation decl we introduce the relation pref, i.e. U pref V iff U is the prefix of V.

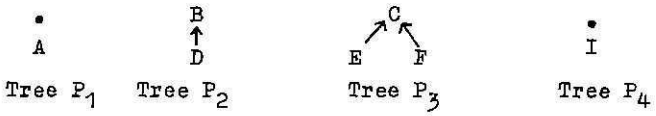
By a prefix sequence of a unit U (denoted by prefseq(U)) we mean a sequence $V_1,\ldots,V_k$ of units such that $V_k = U$, V_1 has no prefix and V_i pref V_{i+1} for $i = 1,\ldots,k-1$. The example of the block and the prefix structures of a program are illustrated in Figure 3.

```
A: begin ref D Z;
   class B; begin ref(C)X1,X2;
     class C; begin
       class I; begin
                  ...
                  end I;
              end C;
         end B;
   B class D; begin
     C class E; begin ref(I)Y1;
                  ...
                  Y1:- new I;
                  ...
                  end E;
     C class F; begin ref(I)Y2;
                  ...
                  Y2:- new I;
                  ...
                  end F;
                ...
                X1:- new E;  X2:- new F;
                ...
                end D;
  Z:- new D;
  end A;
```

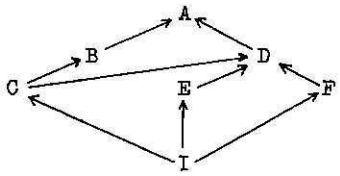


tree T

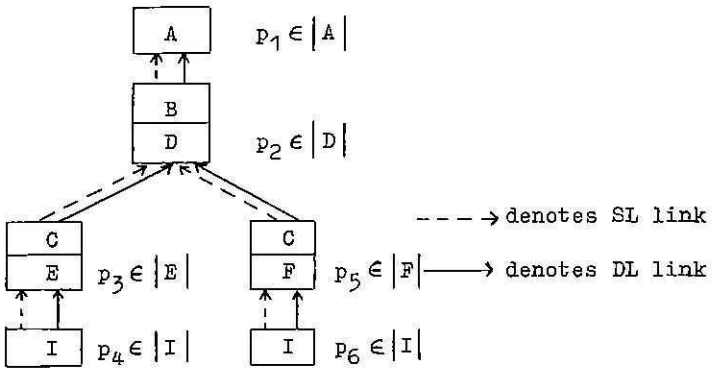
scheme of a block structure



Tree P_1 Tree P_2 Tree P_3 Tree P_4



Graph G



Graphs of SL' and DL's

Fig. 3.

Let $pref^+$ denote the transitive closure of pref and let $pref^*$ denote the transitive and reflexive closure of pref. Then, in particular, U $pref^*$ U for any U, U $pref^*$ V for any U $\in$ prefseq(V) etc.

Note now that if U $pref^*$ V, then an attribute of U is an attribute of V as well. In particular, a syntactic unit W may be an attribute of U and, hence, it will be an attribute of V. Let us denote this extension of the relation decl by attr, i.e. W attr V iff there is a unit U such that W decl U and U $pref^*$ V. While the relation decl always defines a tree, the relation attr need not define a tree. Denote by G the graph determined by the relation attr. Since the relation attr is the extension of decl, the tree T is a subgraph of the graph G.

In Figure 3 the syntactic unit C is the attribute of the syntactic unit D, because C decl B and B pref D. Thus C being the attribute of D may be used as a prefix of the syntactic units E and F. Finally, I decl C and C pref E implies I attr E, similarly I decl C and C pref F implies I attr F.

One-level prefixing is characterized by the following restriction:
(4.1) If U pref V, then level(U) = level(V).
(In words, U may prefix V only if both have the same level in the tree T.) This restriction has many interesting consequences which make the implementation problem almost trivial. First, as an immediate consequence of (4.1) we obtain the following lemma.

Lemma 4.1.

(a) If U attr V, then level(U) = level(V) + 1,
(b) G is a directed acyclic graph with one sink R(T),
(c) Every path in G from U to R(T) has length level(U). □

The definition of a static container for the occurrence of an identifier in a unit is generalized in the following way:

Definition 4.1.

By a static container of the occurrence of an identifier id in a unit V denoted SC(id,V) we mean a syntactic unit U such that id is declared in U and there is a syntactic unit W such that

(a) U pref* W,
(b) V decl* W,
(c) there is no unit W' such that V decl* W' and W' decl+ W and id is the attribute of W' ,
(d) there is no unit W' such that U pref+ W' and W' pref* W and id is declared in W' .

In block structured languages without prefixing we search for the innermost unit W such that id is declared in W and W contains a unit V with the occurrence of id. However, according to the definition of prefixing, the attributes of a prefixing unit are contained in the set of attributes of prefixed unit. This implies that the relation pref is stronger than the relation decl in the following sense: in the process of searching for a static container, we search for it first in the prefix sequence and then in the lower levels of the block structure of a program. Conditions (a)-(c) of definition 4.1 require that we search for the innermost unit W such that id is the attribute of W (U is a unit where the searched syntactic entily is declared). Condition (d) says that U is the nearest prefix of W satisfying the conditions (a)-(c).

We now present an algorithm determining the static container SC(id,V).

Algorithm 4.1.

Start from V. If there is no declaration of id, look for it in pref-seq(V) reading from right to left. If id is not an attribute of V, then take V' such that V decl V' and repeat the above process for V'. If id is not an attribute of V', then take V'' such that V' decl V'' and so on. When the algorithm terminates on the outermost block without finding the required declaration, the static container SC(id,V) does not exist and a program is incorrect. □

Look at Figure 3. We have SC(I,E) = C = SC(I,F), SC(Y1,E) = E, SC(Y2,F) = F, SC(X1,D) = SC(X2,D) = B and SC(D,A) = SC(Z,A) = A.

According to the definition of prefixing, the attributes coming from a prefix sequence are the attributes of a prefixed unit, hence, all of them are local in that unit. Thus we say that the occurrence of an identifier id is local in U if SC(id,U) pref* U, otherwise the occurrence of id is non-local in U.

In the example on Figure 3 all occurrences of identifiers are local.

Dynamic containers

Let prefseq(U_k) = $U_1,\ldots,U_k$ and let us consider an object $p \in |U_k|$. This object consists of layers corresponding to the syntactic units $U_1,\ldots,U_k$. (In Figure 3 p_1 has a layer A, p_2 has layers B,D, p_3 has layers C,E, p_5 has layers C,F, and p_4,p_6 have a layer I.)

Now consider the execution of the instruction lists of units $U_1,\ldots,U_k$. If an identifier id occurs in a unit U_i, $1 \leqslant i \leqslant k$, then for any object $p \in |U_k|$ we must determine a unique object q such that $q \in |V|$ and SC(id,U_i) pref*V. It means that the object q has a layer which corresponds to the static container for the occurrence of id in a unit U_i. The object q will be called a dynamic container of the occurrence of id in a unit U_i with respect to the object p, and will be denoted by DC(id,U_i,p). Dynamic containers will be uniquely determined by means of static links, as before. However, the definition of a syntactic father is more general. In fact, if p.SL = q, $p \in |U|$, $q \in |V|$, then U need not be declared in V. ▲

Look at Figure 3. The object p_2 is created by the instruction Z:-new D, its syntactic father is, of course, the object p_1. In this case D decl A. The object p_3 is created by the instruction X1:- new I and its syntactic father is p_2. In this case E decl D. The object p_4 is created by the instruction Y1:- new I and its syntactic father is evidently p_3. In this case I is not declared in E but in C. Hence the simple rule of Algol 60 does not work. The syntactic father of p_4 is the object p_3 such that I is the attribute of E (not necessarily declared in E). Similarly, the syntactic father of p_5 is p_2, and F decl D, finally the syntactic father of p_6 is p_5, and I attr F.

The example shows the necessity for a more general definition of syntactic father of an object: if $p \in |U|$ and p.SL = q, then q should be an object of a unit V such that U attr V (previously U decl V). The definition of SL chain remains the same as in Section 3. Before we present an algorithm of setting SL links, we prove a lemma analogous to Lemma 3.1 which is of basic importance for the whole con-

struction.

Lemma 4.2.

(a) If $SL(p_k) = p_k,\ldots,p_1$ and $p_i \in |U_i|$ for i=k,...,1, then the sequence $U_k,\ldots,U_1$ is a path from U_k to R(T) in the graph G,

(b) Let $SL(p_k) = p_k,\ldots,p_1$ and $p_i \in |U_i|$ for i=k,...,1. If SC(id,V) exists and V pref* U_k, then there is a unique i, $1 \leq i \leq k$, such that SC(id,V)pref* U_i.

Proof

By the definition of the syntactic fathera, if $p_{i+1} \in |U_{i+1}|$ and $p_i \in |U_i|$, then U_{i+1} attr U_i, for i = k-1,...,1. Hence $U_k,\ldots,U_1$ is a path from U_k to R(T) in the graph G. Thus (a) is proved.

Now by Lemma 4.1 $level(U_i)=i$ for i=k,...,1. Assume that there are two such integers, i, j, $1 \leq i < j \leq k$, that SC(id,V)pref* U_i and SC(id,V)pref* U_j. By the restriction 4.1 $level(SC(id,V)) = level(U_i)$ and $level(SC(id,V)) = level(U_j)$. Hence $level(SC(id,V)) = i = j$, which is impossible.

The proof that such an i exists is given in Section 5 (Lemma 5.3), where the more general case is considered; namely the case of prefixing at many levels. For this reason we do not repeat this proof in a much simpler case and leave it to the next section. □

Now we are able to present an algorithm which is an immediate generalization of the algorithm 3.1.

Algorithm 4.2.

We assume the only one object of the outermost block R(T) may be entered, and for that object SL = none.

Consider now an object $p \in |U|$ created in an object $r_m \in |V_k|$. Let $prefseq(V_k) = V_1,\ldots,V_k$, and let the instruction which creates p occur in a unit V_i, $1 \leq i \leq k$. If id identifies U, then according to the definition 4.1 U is declared in $SC(id,V_i)$. The syntactic father of p should be an object containing $SC(id,V_i)$ as a layer.

Let $SL(r_m) = r_m,\ldots,r_1$. By Lemma 4.2(b) there is a unique j, $1 \leq j \leq m$, such that $SC(id,V_i)$ is the layer of r_j. Then define $p.SL = r_j$. □

Figure 4 shows this general situation. When the statement of a unit V_i with the occurrence of id is being executed, in the SL chain of r_m there is a unique object r_j which may be the syntactic father of p.

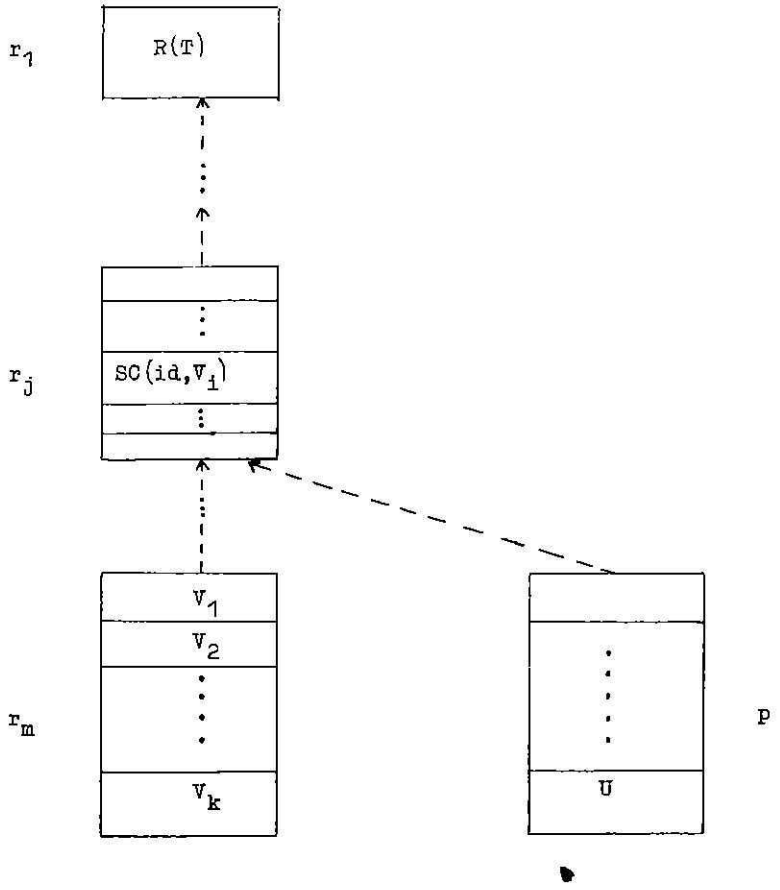


Figure 4

5. Syntactic environment in a programming language with prefixing at many levels.

Existence of a syntactic environment

In this section we shall analyze the situation when Simula's restriction (4.1) is left out. A programming language with block structure and prefixing at many levels, i.e. when (4.1) is not binding, possesses some amazing properties. First we are not able to prove a lemma analogous to Lemma 4.2, where the existence and the uniqu-

ness of the syntactic environment for prefixing at one-level is proved. In particular, the analogon of Lemma 4.2(b) does not hold. However, we can show that for a static container SC(id,V) and V pref* U_k there is at least one i, $1 \leq i \leq k$ such that SC(id,V)pref* U_i, where $SL(p_k)=p_k,\ldots,p_1$, U_1 . Lemma 4.2(b) shows the uniquness of such an i, and thus there is no problem with definition of Simula's semantics. Here the situation is not so clear.

The proof of the existence of such an i, $1 \leq i \leq k$, is given in the following three lemmas. Lemma 5.1 is auxiliary and justifies the implication which is used later in the proof of Lemma 5.2. Lemma 5.2 is crucial for the whole proof. It shows that graph G satisfies the desired property. The proof of this lemma is carried out by double induction, with respect to the length of a path $U_k,\ldots,U_1$ in G, and with respect to the length of the prefix sequence of U_k. At last Lemma 5.3 is a simple corollary of the Lemma 5.2.

Lemma 5.1.

Let the sequence $U_k,\ldots,U_1$ be a path in the graph G from U_k to R(T).
Assumption If V pref* U_k and V decl W, then there exists j, $1 \leq j < k$, such that W pref* U_j.
Conclusion If V pref* U_k and V decl* W, then there exists t, $1 \leq t \leq k$, such that W pref* U_t.

Proof.

First note that the above implication has the following meaning. Assumption says that for any V from prefseq(U_k) and declared in W there is U_j on the path $U_{k-1},\ldots,U_1$ such that W pref* U_j (cf Fig.5.). Conclusion generalizes this property. Namely, for any V from prefseq(U_k) and for any W such that V decl* W there is U_t on the path $U_k,\ldots,U_1$ such that W pref* U_t, (cf Fig.5).

We shall prove the conclusion by induction on the length of path from V to W in the tree T. If V=W and V pref* U_k, then W pref* U_k. Hence t=k in this case.

Now consider units V and W such that V decl+ W. Hence there exists a unit W' such that V decl W' and W' decl* W. If V pref* U_k and V decl W', then it follows from the assumption that there is $1 \leq j < k$ such that W' pref* U_j. Now W' pref* U_j and W' decl* W, where the length of the path from W' to W is less than the length of the path from V to W. Hence by inductive assumption there exists $1 \leq t \leq j$ such that W pref* U_t.

□

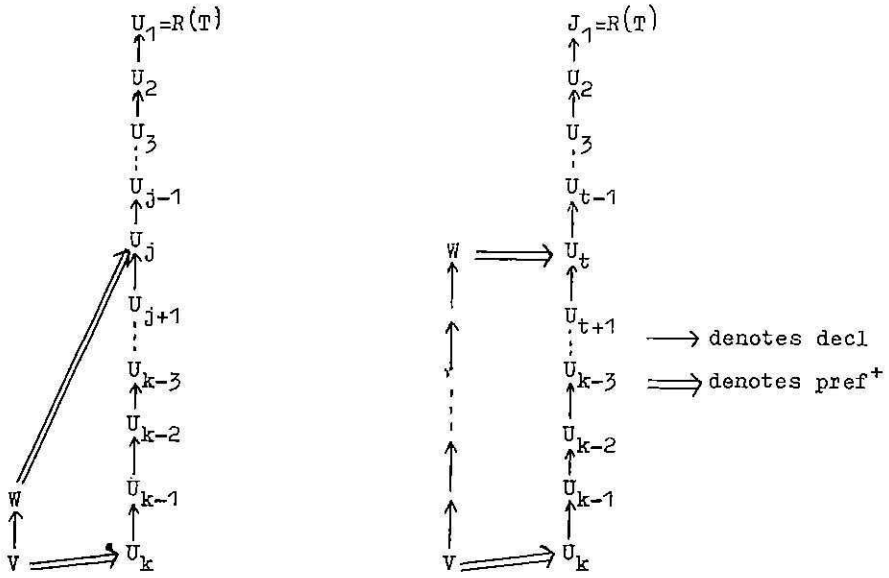


Figure 5

Lemma 5.2.
Let the sequence $U_k,\ldots,U_1$ be a path in the graph G from U_k to $R(T)$. If V pref* U_k and V decl* W, then there exists i, $1 \leqslant i \leqslant k$ such that W pref* U_i.

Proof.

First note that the lemma is simply the conclusion of the previous one. However, it should be proved without the assumption. Since Lemma 5.1 has just been proved, it is sufficient to prove its assumption, i.e.

(5.1) if V decl W and V pref* U_k, then there is j, $1 \leqslant j < k$ such that W pref* U_j.

The proof is carried out by induction on the length of the sequence $U_k,\ldots,U_1$. For k=1, $U_k=R(T)$. Thus V pref* R(T) iff V=R(T) and V decl W for no W.

Assume now that (5.1) holds for all sequences of length less than k, $k \geqslant 2$. For a sequence $U_k,\ldots,U_1$ let V pref* U_k and V decl W. We shall now use induction on the length of the prefix sequence of the unit U_k to prove that W pref* U_j for some $j < k$.

The beginning is simple since for $V=U_k$ we have U_k decl W and consequently W pref* U_{k-1} (by the definition of the relation attr). Assume that (5.1) holds for all prefix sequences of the length less than h. Let V pref* U_k and suppose that the length of the prefix sequence from V to U_k is $h \geq 2$. For some units V', W' we have V pref V' pref* U_k and V' decl W'. The length of the prefix sequence from V' to U_k is h-1. We infer from the inductive assumption that W' pref* U_j for some $j<k$. Now, since V pref V' and V' decl W', the syntactic container SC(id,W') exists, where id identifies V (V occurs in W'). By Definition 4.1 and because V decl W, there is a unit W'' such that W pref* W'' and W' decl* W''. The length of the sequence $U_j,\ldots,U_1$ is less than k and W' pref* U_j so, from the inductive assumption on k, we infer that for a unit $\overline{W}$ such that W' decl $\overline{W}$ there is $m<j$ and $\overline{W}$ pref* U_m.

By Lemma 5.1 if W' decl* $\overline{W}$ and W' pref* U_j there is $1 \leq m \leq j$ such that $\overline{W}$ pref* U_m. Since W' decl* W'' and W' pref* U_j, taking $\overline{W}$ as W'', we obtain W'' pref* U_m. Finally, W pref* W'' and W'' pref* U_m, hence W pref* U_m where $1 \leq m < k$. Thus we have proved (5.1) and the lemma. □

Lemma 5.3.

Let $SL(p_k)=p_k,\ldots,p_1$ and $p_i \in |U_i|$ for $i=1,\ldots,k$. If SC(id,V) exists and V pref* U_k, then there is i, $1 \leq i \leq k$ such that SC(id,V) pref* U_i.

Proof.

From the definition of the SL chain, $U_k,\ldots,U_1$ is a path in the graph G. Since SC(id,V) exists, there is a unit W such that V decl*W and SC(id,V) pref* W. We have V pref* U_k and V decl* W, and by Lemma 5.2 there is i, $1 \leq i \leq k$ such that W pref* U_i. But SC(id,V) pref* W and W pref* U_i implies SC(id,V) pref* U_i. □

Dynamic containers

During the execution of the instruction list of an object $p \in |U|$, we must be able to indicate the dynamic container DC(id,V,p) for any identifier id occurring in any unit V belonging to prefseq(U). To achieve this goal we wish to use the SL chain of the object p, as in Simula 67. Unfortunately, in the case of many-level prefixing the SL chain does not uniquely define the syntactic environment of p, since the same unit may occur more than once as a layer in SL(p). (Lemma 5.3 guarantees a dynamic container belongs to SL chain but not exactly once).

This new complication is well illustrated on Figure 1. The SL chain of the object $p_5 \in |C|$ contains the layer A twice, in the object p_2 and p_4.

Hence it is necessary to introduce a uniform rule for determining dynamic containers. It seems that there are only two concurrent choices. We may take the nearest or the farthest from the given object on its SL chain. However, the second choice is impossible because it contradicts the standart understanding of locality. Consider an occurrence of id local in V and an object $p \in |U|$ containing a layer corresponding to a syntactic unit V. Assume the chain SL(p) contains another object q with a layer corresponding to V. Then, of course, a dynamic container DC(id,V,p) should be the object p, not the object q (for a concrete example see Section 2, where the program with two data structures QUEUE and DECK is considered).

From the above discussion we can infer a new definition of a dynamic container as well as an algorithm which computes SL links.

Definition 5.1.

Let $SL(r)=r_m,\ldots,r_1$ be the SL chain of an object $r \in |V_k|$ and let $prefseq(V_k)=V_1,\ldots,V_k$. Consider an occurrence of an identifier id in a unit V_i. We shall say that r_j is the dynamic container for the occurrence of id in a unit V_i with respect to the object r if r_j is the nearest object to r in SL(r) such that $SC(id,V_i)$ is a layer of r_j.

Algorithm 5.1.

The start is the same as usual. Consider an object $p \in |U|$ created in an object $r \in |V_k|$. Let $prefseq(V_k)=V_1,\ldots,V_k$ and let the instruction which creates p occur in a unit V_i, $1 \leqslant i \leqslant k$. If id identifies U, then according to the definition 4.1., U is declared in $SC(id,V_i)$. Let $SL(r)=r_m,\ldots,r_1$ be the SL chain of r. By lemma 5.3. there is $j, 1 \leqslant j \leqslant m$, such that $SC(id,V_i)$ is the layer of r_j. Let j' be the largest j satisfying this condition i.e. r_j, is the dynamic container of the occurrence of id in the unit V_i with respect to r. Then define $p.SL=r_j$,.

□

6. The addressing algorithm and its correctness.

In this section we shall describe an addressing algorithm for a language with many-level prefixing. The correctness of this algorithm will be proved.

Addressing in Algol and Simula

Let us start with some remarks on an addressing algorithm for the Algol-like language invented by E.Dijkstra ([6], [7]). Let id be a name of a variable v occurring in U and let $SC(id,U)=V$. Then the

variable v is identified by a pair:

$$(\text{level}(V),\ \text{offset}(v))$$

where offset(v) is a relative displacement of v in a memory frame. Note that both quantities level(V) and offset(v) may be computed at compile time. The run-time address of v is evaluated by a simple formula:

$$\text{DISPLAY}[\text{level}(V)] + \text{offset}(v)$$

where DISPLAY is a running system array updated during run-time. When an object $p \in |U|$ is being executed, DISPLAY[i] for $i=\text{level}(U),\ldots,1$ must point to the members of the SL chain of p.

When an object $p \in |U|$ is being generated, it is sufficient to set

$$\text{DISPLAY}[\text{level}(U)] :- p;$$

since for $m < \text{level}(U)$, DISPLAY[m] must be well defined. But when p is reentered the next time (i.e. through DL or goto statement), the following DISPLAY update algorithm is used:

```
X:-p;
for k:-level(U) step -1 until 1 do
begin
  DISPLAY[k]:- X; X:-X.SL;
end
```

For a language with many-level prefixing we postulate that the addressing algorithm is efficient as in the case described above. However, from the discussion given below, it follows that the same method of attributes identification as in Algol-60 (and Simula-67) is not possible.

Let U be an arbitrary unit with prefix sequence $U_1,\ldots,U_n$. It is easy to observe that the prefix sequence has the following property: for every i, $1 \leqslant i \leqslant n$, $\text{level}(U_i) \leqslant \text{level}(U_{i+1})$, where $\text{level}(U_i)$ is determined from the tree T. Due to this property it is not possible to assign one level to all attributes of a given object p since they may be declared in units of different syntactic levels. Hence the local attributes of the object p should be addressed relative to many elements of DISPLAY. (Note that in Simula 67 the equality $\text{level}(U_i) = \text{level}(U_{i+1})$ holds for all U_i belonbing to the prefix sequence of U. Thus, the addressing algorithm is exactly the same as in Algol 60). Consider the following example:

```
B1:begin
    class A;... end A;
    B2:begin
       A class B;... end B;
       ...
       new B;
       ...
       end;
   ...
   end
```

When the object r of class B (generated by new B) is executed, the SL chain of r is described at Fig.6.

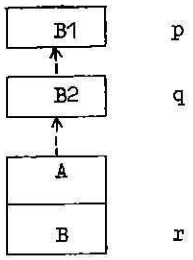


Fig.6.

The Algol-like rule, that DISPLAY[3]=r and DISPLAY[2]=q, is not valid because the attributes of the object r declared in the unit A ought to be addressed with respect to level(A)=2.

In order to avoid these difficulties the assignment of numbers to syntactic units is modified so that levels determined by the program tree T must not be used.

Generalized DISPLAY

To every unit U of a given program we assign a unique number, called a unit number nr(U), determined by any enumeration of tree T. To every id occurring in a unit U we assign a pair of numbers nr(SC(id,U)) and an offset, where the offset is evaluated taking into account all attributes of prefseq(SC(id,U)).

A prefix number sequence pns(U) of a unit U is a sequence

$nr(U_1),\ldots,nr(U_n)$, where $U_1,\ldots,U_n=prefseq(U)$.

The vector DISPLAY is replaced by the vector GDISPLAY, the length of which is equal to the number of vertices of T.

Now we present an algorithm which computes relevant items of GDISPLAY every time an object $p \in |U|$ is entered. Let $SL(p)=p_m,\ldots,p_1$; then the GDISPLAY update algorithm has the form:

Algorithm 6.1.

for k:=1 step 1 until m do
update $GD(p_k)$;

The instruction update $GD(p_k)$ consists of the assignment:

$$GDISPLAY[n_1] :- GDISPLAY[n_2] :- \ldots :- GDISPLAY[n_{d_k}] :- p_k,$$

where $p_k \in |U_k|$ and the prefix number sequence of U_k is $pns(U_k)=n_1,\ldots,n_{d_k}$. □

Observe that for every object $p \in |U|$ the cost of update GD(p) is constant, depending only on the unit U prefix sequence length. The correctness of the GDISPLAY update algorithm can be proved with the help of the following lemma.

Lemma 6.1.
Let $SL(p)=p_m,\ldots,p_1$, where $p_i \in |U_i|$ for i=m,...,1. If id is non-local in V, V pref* U_m and p_j is a dynamic container for id ($p_j=DC(id,V,p)$), then id is non-local in any U_k for k=j+1,...,m.

Proof follows immediately from the definition 5.1 of a dynamic container. □

Theorem 6.1 (correctness of the GDISPLAY update algorithm)
Let $SL(p)=p_m,\ldots,p_1$, where $p_i \in |U_i|$ for i=m,...,1, and assume that the GDISPLAY update algorithm has been executed for an object p. If the occurrence of id is represented by a pair (n,offset) and id occurs in V such that V pref* U_m, then $GDISPLAY[n]=p_j$ ($m \geq j \geq 1$), where $p_j=DC(id,V,p)$.

Proof:

When id is local in U_m, then the dynamic container of id is equal to p and n belongs to $pns(U_m)$. It follows from the algorithm that $GDISPLAY[n]=p_m=p$.

When id is non-local in V, V pref* U_m and $p_j=DC(id,V,p)$, then by Lemma 6.1 for every k=j+1,...,m id is non-local in U_k, hence $nr(SC(id,V))=n$ does not belong to $pns(U_k)$. Since p_j is a dynamic container of id, it follows that SC(id,V) pref* U_j; thus n belongs

to pns(U_j). Therefore, after executing the update algorithm loop "for k:=j"we have GDISPLAY [n] =p_j and by the Lemma 6.1 this value will not be changed for k=j+1,...,m. □

This theorem implies the correctness of the run-time addressing algorithm given by a formula:

GDISPLAY [n] + offset

where the pair (n,offset) represents an attribute in a program.

The following example illustrates the use of the GDISPLAY mechanism. Let us consider the extended scheme of the program given in the previous example.

```
B1[1]: begin
         class A[2]; begin real x[2,m];... end A;
       B2[3]: begin real y[3,n];
                A class B[4]; begin real z[4,k]; ... end B;
                ...
                new B;
                ...
              end;
       ...
       end
```

In this program every unit has a unit number given in brackets and every variable is identified by a pair of numbers: the first is a unit number of the static container of this variable and the second is a displacement in a memory frame. Consider the execution of the statement new B. A new object r of class B is created, the SL chain of r (see Fig.6) consists of the objects r,q (the block B2) and p (the block B1). Before control passes to the object r we must execute the GDISPLAY update algorithm. Fig.7 shows the contents of the vector GDISPLAY after its execution.

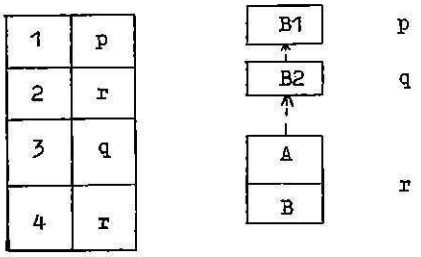


Fig.7.

Note that the attributes x and z of the object r are identified by two different unit numbers. However, due to the GDISPLAY update algorithm, all the elements of the vector GDISPLAY corresponding to the prefix number sequence of the unit B refer to the object r. Thus, the addressing formulas:

GDISPLAY [2] + m
and GDISPLAY [4] + k

compute the addresses of x and z respectively in the frame of the object r.

7. Storage management

In this section we discuss briefly possible strategies of storage management and their influence on the semantics of the language with many-level prefixing. We propose a new approach to the problem and some principles of implementation.

Terminated objects accessibility

Consider first the problem of the accessibility of terminated objects. By a terminated object we mean an object in which control has passed through the final end.

Two different cases occur in Simula 67. A block (or a procedure) object is not accessible after its termination while the termination of a class object does not affect its accessibility. The property that a block object becomes inaccessible after its termination results only from the static properties of the correct program and may be statically checked.

Note another important property of Simula 67. The SL chain of the object being executed contains no terminated objects. It follows from the above properties that the activation record for a block or a procedure may be deleted from a memory as soon as this object is terminated.

The situation is quite different when many-level prefixing is allowed. Consider the following example:

```
L1: begin ref(A)X;
      class A;
        ...
    end A;
```

```
L2: begin integer j;
      A class B;
      begin
        procedure P;
        ...
        j:=j+1;
        ...
      end P;
    end B;
    X:-new B;
  end L2;
  XquaB.P; comment XquaB.P denotes instantaneous qualification which
           changes the qualification of X;
end L1;
```

After the execution of the assignment X:-new B there exist three objects: p of block L1, q of block L2 and r of class B, the latter pointed by X. Recall that this assignment is valid because X is qualified by class A and A prefixes B.

Observe now the instruction XquaB.P after the termination of objects r and q. This instruction denotes a call of the procedure P. The created object of the procedure would have in its SL chain two terminated objects: q and r. Note that P may use the attribute j from the terminated block object q. As we see, Simula's access rules are violated. Therefore the semantics of such a call must be determined. (Is the call of procedure P legal or would it cause a runtime error?) Is the access to j of object q legal or would it cause a run-time error? Two solutions are admissible, each implying a possible storage management strategy (cf [2]).

Retention semantics

The first semantics is called "retention". The object remains accessible as long as at least one user's or system pointer (e.g. SL or DL link) refers to that object. The retention strategy of storage allocation corresponds to the above semantics. This strategy may be accomplished either by the use of reference counters or by garbage collection.

Observe however, that within the retention semantics the concepts of block and procedure become trivial. A procedure would be a kind of a crippled class without a remote access mechanism. A block would only be an abbreviation of an anonymous class declaration and a generation at the same time. In this semantics the call of procedure P

from the example is legal because the objects q and r are accessible.

Deletion semantics

Following the Simula principles we choose the other semantics, which may be called "deletion". It consists in the principle that a non-class object becomes inaccessible after its termination while a class object remains accessible as long as at least one user's or system pointer refers to that object. We regard this semantics proper for two reasons. First, it keeps the distinction between classes and blocks or procedure. Second, it admits the deletion of terminated non-class objects from a memory (but whether terminated non-class objects are actually deallocated immediately after their termination still depends on the implementation).

Since we are aiming at the possibility of deallocating non-class objects, we must provide the following property:

(7.1) The object being executed has no terminated non-class objects in its SL chain.

The implementation we propose makes use of SL links defining the SL chains for objects. These links are additional attributes of objects. We intend to treat system reference variables and user's reference variables uniformly. Hence, an SL link should become inaccessible after non-class object termination. (Observe also that when an object contains in its SL chain a terminated non-class object, it can not become an active object, because the display update algorithm (Algorithm 6.1) would fail in searching through the SL chain. In such a case a syntactic environment of the object would not be recovered even if the object requiring the display updating does not refer to inaccessible attributes).

Recall the statement XquaB.P from the example. The new created instance of P has a terminated non-class object q in its SL chain. The property (7.1) fails in this case.

Referencing mechanism

The new method of referencing must carry the information about the termination of non-class objects. Thus that method should realize the dictionary operations:insert, delete and member on the collection of all accessible objects.

In this paper we are not concerned with the strategy of allocating new frames for objects. Therefore we may omit some details and assume the existence of the function newframe (appetite) yielding an address of a new allocated frame of length appetite. Similarly we assume the existence of a procedure free(X) which releases the frame indicated by an address X.

The operation insert corresponds to the creation of a new object and should be understood as making the new object accessible. Insert does not deal with memory allocation itself.

Operation delete corresponds to the termination of a non-class object, and member yields information whether a reference points to an accessible object.

We will use an auxiliary data structure, an array H, containing references to objects. Roughly speaking, objects will be addressed indirectly through array H. It is obvious that the operation member should be as efficient as possible, for it is the most frequently used. (In our implementation the cost of member is really low: only two machine instructions).

Array H occupies low addresses of core, from 0 to the position pointed by a variable LASTITEM. (Objects may be allocated in high addresses of core). Each item in H is represented by two words, the physical address of an object and an integer called an object number. The algorithms presented below also use a procedure "intolist", a function "deletefrom" and a boolean function "empty", operating on the auxiliary list of released items of H. Because of their obvious meanings, details are omitted. Let the variable LIST be the head of this list.

Now objects are referenced by the so-called virtual addresses defined as pairs (addres in H, object number). The object number will be used for checking whether the object is accessible, while address in H will be the indirect address of the object (if accessible).

For a reference X denote the first and the second component of the virtual address of X by Xadd and Xob. The method of referencing will satisfy the following properties:

(7.2) If X refers to an accessible object, the H[Xadd] contains the physical address of the object,

(7.3) X refers to an accessible object iff Xob=H[Xadd+1] (i.e. iff object numbers are the same in the virtual address of X and the corresponding item of H).

Hence, the algorithm for the member operation is as follows:

```
boolean procedure member (Xadd,Xob,physical address);
name physical address; integer Xadd,Xob,physical address;
begin
  if Xob=H[Xadd+1] then
  begin physical address:=H[Xadd]; member:=true
```

```
  end else member:=false
end member;
```

Consider now delete operation. Following property (7.3) it is sufficient to change the object number in an item of H to guarantee that the subsequent executions of a member concerning this item return value false. All items in H which previously pointed to some objects, subsequently being made inaccessible, are linked together into a list (started by the variable LIST) and may be reused for addressing some new objects.

The algorithm of delete operation is as follows:

```
procedure delete (Xadd,Xob);
integer Xadd,Xob;
begin integer addr;
  if member (Xadd,Xob,addr) then
  begin free (addr); comment a frame in memory may be released;
    H[Xadd+1] :=H[Xadd+1] +1;
    intolist (Xadd,LIST)
  end
end delete;
```

When a new activation record is allocated, a new element must be inserted into H. If the list of released items of H is not empty, one of the previously used elements of H may be reused. Otherwise array H is extended (LASTITEM:=LASTITEM+2).

```
procedure insert (appetite,Xadd,Xob);
name Xadd,Xob; integer appetite,Xadd,Xob;
begin
  if empty(LIST) then
  begin Xadd:=LASTITEM+1; H[Xadd+1]:=0; LASTITEM:=LASTITEM+2
  end else Xadd:=deletefrom(LIST); comment one element has been
  taken from the list of released elements;
     Xob:=H[Xadd+1];
     H[Xadd]:=newframe(appetite)
end insert;
```

Moreover we intend to treat uniformly references to terminated objects of non-classes and the reference to the empty object none. This is easily accomplished by the following initialization:

```
none:=(0,0); H[0]:=H[1]:=1;
```

Hence none does not refer to any accessible object because its object number equals 0 and H[1] equals 1.

Finally, we recall now that the SL chain may be cut off. Therefore the display update algorithm must be modified.

Algorithm 7.1.
Let $SL(p)=p_m,\ldots,p_1$, then the GDISPLAY update algorithm has the form:

```
X:-p;
while X.SL =/= none do X:-X.SL;
if X ∉ |R(T)| then error else
for k:=1 step 1 until m do updateGD(p_k);
```

□

Let us now discuss the cost of the proposed referencing method. Each accessible object needs two extra words for an item in the array H. Each reference variable needs two words for a virtual address. Thus, with respect to a standard method we lose two words for each accessible object and one word for each reference variable. (However, the pair of integers forming a virtual address may sometimes be packed into one machine word; the same may be done for an item of two words in the array H.)

On the other hand, we profit in an essential increase of the total number of different objects which may be used through the program lifetime without garbage collection. This number exceeds by far the capacity of H, though the number of objects accessible at the same time is limited by H. The new strategy has the advantage of the standard one when a program uses many procedures (what is natural and very common). Then the terminated objects of these procedures are deallocated on line and the corresponding space may be immediately reused by the other objects (as in the case of stack-implementable language). Observe that the lack of on-line deallocation of terminated non-class objects was the main snag to efficient implementations of Simula-67. Mereover, by virtue of this indirect addressing (in case of memory segmentation), the memory compactification may be done without traversing a graph of objects and without updating the reference variables. It may be accomplished by removing inaccessible objects and changing the corresponding addresses stored in the array H.

Finally the time-cost of these three operations (delete, insert, member) is as follows. The cost of the operation member is constant and very low. It may be compared with the cost of testing on none in standard implementation. The cost of insert and delete depends on the cost of other operations like newframe, free, intolist, deletefrom, which maintain the frames of inaccessible objects. Apart from the cost resulting from these operations, the cost of delete and insert is constant. These operations may be implemented in many different

methods. However, with the use of good algorithms and data structures (i.e. linear lists, heaps etc.) one can obtain the same time complexity as in the case of standard solutions. Moreover, observe that due to the property (7.1) display may contain physical addresses instead of virtual ones, so an access to the visible attributes is not charged by the cost of member operation.

Programmed deallocation

To end this section, as a consequence of the reference mechanism introduced above, we can propose the new operation to be introduced to the programming language. This new operation is called usually programmed deallocation and may be denoted by kill(X), where X is a reference. The semantics of kill(X) is as follows. If X is a reference to an accessible object, then kill(X) makes this object inaccessible (and in consequence this object may be deallocated). Otherwise kill(X) is equivalent to the empty statement.

We got kill operation as a benefit from the referencing method introduced because of the other reasons. Roughly speaking, kill is realized by the delete operation described previously. Thus, after the execution of kill(X) the object pointed (if any) by X becomes inaccessible. Moreover, any remote access to such a being made inaccessible object will cause a run-time error. The realization of this is possible as a result of the operation member already existing in the set of storage management operations. Here the simple test on none is extended to the test on being accessible (member operation). We showed that the cost of member operation is constant and very low, and may be compared with the cost of the test on none. Thus, with some lost of space and a minimal loss of time we can solve the problem of "dangling reference".

We are confident that a programmer when allowing the use of programmed secure deallocation will be able to perform an efficient storage management by conscious deletions of useless objects. Therefore in most cases the time consuming garbage collection may be omitted.

Acknowledgement

The authors are very grateful to Tomasz Muldner for many useful comments and for careful reading of the manuscript.

References

[1] Bartol W.M. "The definition of the semantics of some instructions of a block structured language with type prefixing.", manuscript, 1980.

[2] Berry D.M., Chirica L., Johnston J.B., Martin D.F. and Sorkin A. "Time required for reference count management in retention block-structured languages." Part 1, Int. J. Comp. and Inf. Sciences, Vol.7, No.1 (March 1978), pp.11-64.

[3] Bobrow D.G., Wegbreit B. "A model and stack implementation of multiple environments", Comm.A.C.M., Vol.16, No.10 (Oct.1973), pp.591-603.

[4] Dahl O-J., Myrhaug B., Nygaard K., "Simula 67 Common Base Language", Norwegian Computing Center 1970.

[5] Dahl O-J., Wang A., "Coroutine sequencing in a block structured environment", B.I.T. Vol.11 (1971), pp.425-449.

[6] Dijkstra E.W., "Recursive programming", Numerische Mathematik 2, Vol.2 (1960), pp.312-318.

[7] Gries D., "Compiler construction for digital computers." New York, Wiley 1971.