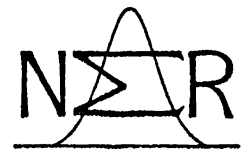
NORSK REGNESENTRAL / NORWEGIAN COMPUTING CENTER

Forskningsvn. 1 B, Blindern, Oslo 3, Norway, telefon (02) 46 69 30



Publication No. S 47
Revision March 1973

SIMULA™ IMPLEMENTATION GUIDE ©

BY

OLE-JOHAN DAHL AND BJØRN MYHRHAUG

0. INTRODUCTION

The "SIMULA implementation guide" did, as the name indicates, serve as a guide for teams implementing the SIMULA Common Base.

This guide was not intended to answer all questions that can come up during implementation, but to serve as a source of background material for the implementors.

The present revision is mainly a reprint of the 1969 edition, except for correction of some misprints and rearrangement of the material. Some sections (e.g. on implementation of the sequencing set) have been dropped, since the techniques are now well known.

References are frequently made to the "Common Base" meaning "SIMULA 67 Common Base Language" by O.J. Dahl, B. Myhrhaug and K. Nygaard, published by the Norwegian Computing Center, May 1968, revised 1971.

The guide is divided into two parts:

Part 1: the compiler
Part 2: the run-time system

The techniques described are only proposals and must be regarded as such. Even if the scheme outlined in this guide was used, one should expect that modifications were made to suit the particular computer.

1.0 INTRODUCTION - COMPILER

The following documentation outlines some of the main features of a compiler for the SIMULA Common Base. The compiler has been planned according to the design principles of the Gier Algol compiler (see P. Naur et al: various papers in Nordisk Tidsskrift for Informasjonsbehandling, BIT).

This documentation is mainly concerned with the implementation of the non-Algol features of the SIMULA language.

The compiler is described in terms of four passes. The functions of the last pass may alternatively be effected through scatter read fix-ups at load time, or through indirect addressing at run time.

On smaller computers it may be convenient to split up the third and main pass into more than one.

Formal descriptions of compile time data and associated algorithms are given in SIMULA with the following features added to the Common Base:

A generalized type declaration is introduced, similar to the class declaration.

A declared type is itself a type declarator. In this document, that has been indicated by using the type identifier underlined.

```
type quantity ; begin ..... end ;
quantity array loc [1:nloc] ;
ref (quantity) x ; x :- loc[i] ;
```

x will now point into the middle of the array loc.

Descriptions of partially compiled source language text are given in Backus Normal Form (BNF).

1.1 Operation of pass 1

The main functions of this pass are:

- lexicographic analysis
- syntactic analysis
- transformation of source program to an intermediate language

A possible hash algorithm for identifiers is: add together consecutive bit sequences of length k of the binary representation of the given character string, take the result modulo 2**k as hash index. Approximate partial sums can be obtained by wordwise multiplications by a factor 2**o + 2**k + 2**2k + ... for a 6-bit character machine k=12 will give immediate lookup if all identifiers are 2 characters or less.

A hash table of identifiers and the corresponding lookup algorithm may be as follows:

```
ref (item) array hash [0:2**k-1] ;
class item (charstring) ; value charstring ; text charstring ;
begin ref (item) next ; ref (quantity) sem ; integer blev, qualev ;
end item ;

procedure lookup (s,id,old) ; name id, old ;
          text s ; ref (item) id ; Boolean old ;
```

```
begin integer i ;
  i := hasher (s) ; old := false ;
  if hash [i] == none then id :- hash [i] :- new item (s)
  else begin id :- hash [i] ;
    L:  if s = id.charstring then old := true
    else if id.next == none then id :- id.next :- new item (s)
    else begin id :- id.next ; go to L end
  end
end lookup ;

integer procedure hasher (s) ; text s ;
begin hash algorithm end hasher ;
```

The item attributes sem, blev and qualev are used in pass 3 to define the semantic contents of an identifier at any time.

The table is initialized to contain all system defined procedures and classes, and can be generalized to accommodate external symbols and constants as well as identifiers. The internal representation of any such item is an internal code followed by a ref (item) value (output in reverse order).

In the output of pass 1 blocks of the different kinds have the following formats.

1. prefixed block:
   <block prefix> prefbegin <declaration list> declend
   <statement list> prefend

2. sub-block:
   begin <declaration list> declend <statement list> blockend

3. procedure body:
   procbegin <declaration list> declend <statement list> procen

If the external procedure body is an unlabelled block, the <declaration list> and <statement list> are those of the external body. Otherwise the declaration list is empty and the statement list is the external body.

4. class body:
classbegin <declaration list> declend <statement list> classend

where <declaration list> and <statement list> are defined as for a procedure body.

5. connection block:
connbegin blockbegin declend <statement> blockend connend

where statement is the external connection block. (The sub-block is ignored in pass 3 if its LQL is empty, see the following sections.)

1.2 Compiler - pass 2

The program text is scanned backwards. The main purpose of pass 2 is to assemble a list of all quantities local to a block in the head of the block, for each block in the program except class bodies. The list includes all local labels and information about the attributes of local and the formal parameters of local procedures.

In the output of pass 2 a prefixed block has the following format:

pref <block prefix> prefbegin <local quantity list>
declbegin <reduced declaration list> declend <statement list>
prefend

and a sub-block has the format:

blockbegin <local quantity list> declbegin <reduced declaration list> declend <statement list> blockend

A <reduced declaration list> is a sequence of reduced declarations, possibly empty:

<reduced declaration> ::= <array declaration>|<switch declaration>
<reduced procedure declaration>|<reduced class declaration>

<reduced procedure declaration> ::= procedure <proc.id.> procbegi
<local quantity list> declbegin <reduced declaration list>
declend <statement list> procend

<reduced class declaration> ::= class <class id.> classbegin
<reduced declaration list> declend <statement list> classend

A <local quantity list>, LQL, is defined as a collection of records containing a main record of class brec. Furthermore any record referenced from a record in an LQL belongs itself to the LQL. No record will belong to more than one LQL.

```
class brec (pref,nvirt,npar,nloc); ref (item) pref; integer
nvirt,npar,nloc; begin quant array loc [1:nloc]; end brec;

class quant; begin ref (item) ident,qualid; integer type,
      kind,categ,dim; Boolean last; ref (brec) descr; end quant;
```

The attributes have the following meaning:

pref: Prefix identifier of a class declaration.

nvirt: The number of virtual quantities (of the main part of a class decl.).

npar: The number of parameters and virtual quantities.

nloc: The total number of local quantities.

loc: A vector of quant records, one for each local quantity in the order virtual quantities, parameters, declared quantities.

ident: The declared or specified identifier.

qualid: The qualifying class identifier of a quantity of type ref.

type: Notype, real, integer, boolean, text, label, ref, character.

kind: Simple, array, proc, class.

categ: Declared, virtual, value parameter, name parameter, parameter by reference.

dim: The number of dimensions of an array.

last: True for a declared quantity which is the last one of a type declaration or an array segment.

descr: Reference to a brec record describing the attributes of a class and the formal parameters of a declared procedure.

Notice that the quant attribute descr excludes the attributes last and dim.

Switches can be treated as quantities of kind proc and type label.

LQLs can be built up by means of two segmented auxiliary stacks, Q and L. The Q-stack contains entries of class quant. Whenever an identifier is processed as part of its declaration or specification, an appropriate quant record is added to the stack. When a new block end of any sort or a procedure heading is encountered, another Q-stack segment is started.

The L-stack contains entries of class brec. When a class declaration or procedure heading is finished, a brec record is formed and added to the L-stack. Its loc array is a copy of the last segment of the Q-stack. Rearranged as described above (virtuals, parameters, locals) that segment is removed and a quant record for the declared procedure or class is added to the Q-stack.

When any new block, except a class body, is encountered another L-stack segment is started. When the block is finished, the course of actions is as above, except that nothing is added to the Q-stack. The generated brec record is the main record of the LQL of the block. The LQL itself is the last segment of the L-stack. That segment is removed and transferred to the output file.

In the output file, a ref (brec) value is conveniently represented by a record ordinal within the LQL. The records should be output in the natural LIFO order (last in, first out).

Then pass 3 will read each record before the one containing the corresponding descr attribute is read. The reference value of the latter can therefore be found by direct lookup in a table built up during the input of the LQL.

1.3 Compiler - pass 3

Pass 3 is the main compiler pass, which performs the actual translation into machine instructions. The output consists of the following types of information:

1. Machine instructions (or abstract representations).

2. Control information for pass 4 to update machine instructions output earlier in pass 3.

3. Prototypes containing block information relevant to the runtime system for the administration of storage, the run time checking of parameters (when necessary), the interpretation of virtual quantities and the implementation of the subclass concept.

Pass 3 maintains the following counters relevant to the output:

1. pc: The program instruction counter.

2. tc: The text item counter.

3. pt: The prototype space coordinate.

The hash table of identifiers (and constants) generated in pass 1 can be simplified by omitting the array hash. For formal convenience the item class is redefined with no consequences for the internal representation of item records.

```
class item (charstring,sem,blev,qualev, next) ;
text charstring ; ref (quantity) sem ; integer blev, qualev ;
ref (item) next ;;
```

Initially all records are cleared (they are assumed to occupy a known continuous area). Then a system block prefix is entered, whose LQL (see below) describes all system defined procedures and classes. The item attributes next will in the following be used to represent a redeclaration stack of item records for each identifier. The item record referenced by the internal representation of an identifier will at any time display the current semantic contents of the identifier.

1.3.1 Blocks

The following quantities describe the blocks enclosing the current point of compilation:

```
ref (brecord) array display [0:maxblev];
Boolean array refable, con[0:maxblev];
integer bl ;
```

Where maxblev is the maximum block level, and bl is the current block level. display [i] , i = 1, ..., maxble is a reference to the record describing the enclosing block at level i, refable [i] is true if the block is or connects a referenceable object and con [i] is true for a connection block. For a prefixed block, a sub-block or a procedure body the entry in display refers to the main record of the associated LQL.

An LQL in pass 3 is a collection of records of the classes brecord and quantity, which are extensions and modifications of the classes brec and quant of pass 2. The collection of LQLs of enclosing blocks forms a stack. The item records of redeclaration stack may easily be incorporated into the same stack, and the same is true for the entries in use of the above arrays

The class quantity has been extended by the following attributes:

addr: Run time address, normally the relative addre within a data record. For a declared label t run time address is an instruction address (p For any quantity matching a virtual specification the attribute addr is a relative addre within a prototype. For a label or procedure whose declaration has not been processed, add is used to contain the text coordinate tc of the last compiled forward reference, which points to the next one a.s.o.

def: Relevant for a declared label or procedure. It has the value true if the declaration has been processed. As long as def is false addr may contain the text coordinate (tc) at which the quantity was last referenced.

qual: A reference to the brecord describing the class which qualifies a quantity of type ref. The attribute qual and qualid may occupy the same storage position.

encl: A reference to the enclosing brecord.

dispq: Procedure to display this quantity into the hash table.

undispq: Procedure to remove this quantity from the hash table.

setqual: Procedure for assigning values to the attributes qual and qualev.

```
class quantity ;
   begin ref (item) ident,qualid; ref (brecord) descr,encl,qual;
   integer type,kind,categ,dim,addr; Boolean last,def,locqual;

   procedure dispq (bl); integer bl;
      inspect ident when item do
         begin if sem =/= none then
               begin if blev = bl then
                     begin if encl == sem.encl then
                           error ("redeclaration");
                     end;
                     next :- new item ("",sem,blev,qualev,next);
               end;
               sem :- this quantity; blev := bl;
               if type = ref then qualev := if locqual then
                  bl else qual.deq.ident.blev;
          end dispq;
```

```
   procedure undispq;
      inspect ident when item do
         if next =/= none then
         begin sem :- next.sem;
               blev := next.blev;
               next :- next.next;
         end else
          begin sem :- none;
               blev := 0;
         end undispq;

   procedure setqual (bl); integer bl;
      inspect qualid do
         begin if sem == none then error ("unknown qualifier");
               if sem.kind ≠ class then
                     error ("qualifier not class");
               qual :- sem.descr;
               locqual := blev = bl;
         end setqual;

end quantity;
```

The class brecord has the following attributes in addition to those of the class brec:

deq: A reference to the quantity record representing the declaration of a class block or a procedure block.

prec: A reference to record describing the prefix class of a class block or a prefixed block, or the one describing the formal parameters of a procedure. The attributes pref and prec may occupy the same storage location.

virtrec: A reference to a record describing the virtual quantities at this or any lower prefix level.

pad: Runtime prototype address (relative).

reclg: Length of runtime data record.

plev: Prefix level.

decbeg: Instruction address of the first array declaration of this block head or, if none, equal to statbeg.

statbeg: Instruction address of the first statement, or if none, equal to finbeg.

finbeg: Instruction address of the first statement following the symbol inner or, if none, the instruction address of the final end.

contclass: True for a block containing local class declarations.

seen: Set to true when allocation has started (see allocate).

taken: Set to true when prefix information has been collected during allocation (see allocate).

virtuals: The attributes of a virtuals record have the following meaning:

totvirt: The number of virtual quantities (accumulated).

actq: Each component is a reference to the quantity record which represents the matching quantity, if any.

actad: The run time address of the matching quantity, except for a procedure.

allocate: Procedure for defining the values of the following brecord attributes:

outer,prec,virtrec (and all attributes of the virtuals record), pad,reclg,plev,contclass, and the quantity attributes addr,qual and qualev of the relevant components of the array loc. The quantity attribute categ is set to virtual for a quantity matching a virtual one. The hash table mechanism is used to discover quantities matching virtual specifications and to define the attributes prec and qual. The procedure is recursive, its net result is to allocate its own record, those of the prefix sequence not already allocated, and those of local classes.

dispc: Procedure to display the local class identifiers into the hash table, including those local to the prefix sequence.

undispc: Procedure to remove all local classes from the hash table.

incl: Procedure to determine whether a given class is included in this one.

displ: Procedure to display all local quantities in the hash table, including those local to the prefix sequence.

undispl: Procedure to remove all local quantities from the hash table.

```
class brecord(pref,nvirt,npar,nloc);
   ref(item)pref; integer nvirt,npar,nloc;
   begin ref(brecord)prec; ref(virtuals)virtrec; ref(quantity)deq;
      integer pad,reclg,plev,decbeg,statbeg,finbeg;
      Boolean contclass,taken,seen; quantity array loc[1:nloc];

      class virtuals(totvirt); integer totvirt;
         begin ref(quantity)array actq[1:totvirt];
      integer array actad[1:totvirt];
      end virtuals;

   procedure allocate(bl); integer bl;
      begin integer i,k,r,v;
        seen := true;
        comment establish prefix sequence;
        if pref==none then begin v:=plev:=0; r:=rechead;contclass:=
                                                        false end
        else inspect pref do
        if sem==none then error ("unknown prefix")
        else if sem.kind≠class then error ("prefix not class")
        else if blev≠bl then error ("class declarations at
                                     different block levels")
        else begin prec :- sem.descr;
           if not prec.seen then prec.allocate (bl)
            else if not prec.taken then error ("prefix loop");
            v := prec.virtrec.totvirt; r := prec.reclg;
      contclass:=prec.contclass; plev:=prec.plev+1
   end prefix sequence and initial conditions established;
   taken:=true; virtrec:-new virtuals(v+nvirt);
   pad:=pt; pt:=pt+prohead+v+nvirt;
   if deq=/=none and deq.kind=proc then pt:=pt+prec.npar;
   comment the prototype of a procedure should contain the
      parameter descriptors although they belong at compile
      time to a prefix brecord;
   for i:=nvirt+1 step 1 until nloc do inspect loc i do
   if kind≠proc and kind≠class and type≠label then
      begin addr:=r; r:=r+1; if type=text then r:=r+1;
```

```
   comment a text descriptor is assumed two words long;
   if(kind=array or type=ref or type=text) and last then
                                                pt:=pt+1
end evaluation of attribute addresses and prototype length;
reclg:=r; dispc(bl);
comment all local classes must be in display when allocating
   each of them and in order to check the qualifications
   of virtual refs;
inspect virtrec do
   begin for i:=1 step 1 until v do
      begin actq[i]:-prec.virtrec.actq[i];
         actad[i]:=prec.virtrec.actad[i];actq[i].dispq(bl)
      end take over and display of old virtuals;
      for i:=v+1 step 1 until totvirt do inspect loc[i-v]do
   begin actq[i]:- this quantity;actad[i]:=addr:=i;def:=true;
      if ident.blev=bl then error("conflicting virtual
                                          specification");
      if type=ref then setqual(bl); dispq(bl)
   end initialisation and display of new virtuals;
for i:=nvirt+1 step 1 until nloc do inspect loc[i]do
   begin if type=ref then setqual(bl);
      if kind=class then
        begin descr.deq:-this quantity; contclass:=true;
           if not descr.taken then descr.allocate(bl+1)
        end class case;
      if kind=proc then inspect descr do
        begin seen:=taken:=true;
           for k:=1 step 1 until npar do inspect loc[k]do
        begin addr:=k-1+rechead;if type=ref then setqual
              (k+1); end;
           end proc case, there is no separate prototype
                          for a proc heading;
        if ident.blev=bl and ident.sem.categ=virtual then
           begin if kind≠ident.sem.kind or (type≠ident.sem.typ
              and ident.semtype≠universal)then error("no matcl
            else if type=ref and ident.sem.type=ref and
            not ident.sem.qual.incl(qual) then error
                                    ("not subordinate");
```

```
                    k:=ident.sem.addr; if k>v then actq[k].ident:-
                                                          none;
                    actq[k]:- this quantity;actad[k]:= addr;
                    addr:=k; categ:=virtual; def:=true;
                end virtual case
            end scanning of locals;
          for i:=1 step 1 until totvirt do actq[i].undispq
        end binding of virtuals;
      undispc
    end allocate;

procedure dispc(bl); integer bl;
begin integer i;
    if prec =/=none then prec.dispc(bl)
    for i:= npar+1 step 1 until nloc do inspect loc[i]do
        if kind = class then dispq(bl) end dispc;

procedure undispc;
if contclass then begin integer i;
   for i := npar+1 step 1 until nloc do inspect loc[i]do
       if kind = class then undispq;
   if prec =/= none then prec.undispc end undispc;

Boolean procedure incl(x); ref (brecord) x;
if x.plev < plev then incl := false else
L: if x.plev = plev then incl := x = this brecord else
   begin x :- x.prec; go to L; end incl;

   procedure displ(bl); integer bl;
   begin integer i;
      if prec =/= none then prec.displ(bl);
      for i := 1 step 1 until nloc do loc[i].dispq(bl)end displ;

   procedure undispl; begin integer i;
   for i := nloc step -1 until 1 do loc[i].undispq;
   if prec =/= none then prec.undispl end undispl;
end brecord;
```

The following procedure is used whenever a block head is encountered.

```
procedure enter (x,r,c); ref (brecord) x; Boolean r,c;
begin bl := bl+1; display[bl] :- x; refable[bl] := r;
                                con[bl] := c; x.displ[bl] end enter;
```

The course of actions on entering a block head depends on the type of block. It is assumed that the LQL of a prefixed block, or a procedure body has been read in and established as a list structure.

```
ref (brecord) mr,cr; ref (item) ci,pi;
ref (quantity) pr;
```

1. Prefixed block:

```
mr.pref :- ci; mr.allocate (bl+1);
enter(mr,false,false)
```

Where "mr" is a reference to the main record of the LQL, and "ci" is the class identifier of the block prefix.

2. Sub-block:

```
if mr.nloc ≠ 0 then
  begin mr.allocate (bl+1); enter (mr,false,false)end
```

3. Procedure:

```
pr:-pi.sem;mr.deq:-pr;mr.prec:-pr.descr;pr.def:=true;
if pr.addr≠0 then fixup(pr.addr,pt); pr.descr.pad:=pt;
mr.allocate(bl+1); enter(mr,false,false)
```

Where "pi" is the procedure identifier, and "fixup" error("no an outputs control information for pass 4 to insert the correct prototype address into all forward references to this procedure.

4. Class:

enter (ci.sem.descr,true,false)

Where "ci" is the class identifier.

5. Connection block:

enter (cr,true,true)

Where "cr" is a reference to the record describing the class associated with the connection block. For a connection block 1, cr = ci.sem.descr, where "ci" is the class identifier of the connection clause. For a connection block 2, "cr" is the qualification of the preceding reference expression.

The following procedure describes the course of actions upon completion of any block, except that a blockend is ignored if con(bl) is true.

```
procedure leave; begin if not con(bl) then outprot;
   display (bl).undispl; display (bl) :- none; bl := bl-1 end leave;
```

The assignment of none to display(bl) implies that the LQL stack can be reduced by one level if refable(bl) is false.

The procedure "outprot" constructs and outputs the prototype of the completed block. The procedure is not described formally, since many details of a prototype will be machine dependent.

A prototype should contain at least the follcwing information: (The compile time equivalent is indicated for each item.)

1. The prototype length (possibly defined implicitly by a terminal item).

2. A reference to the prototype of the prefix class, if any (prec.pad).

3. The type of a procedure (deq.type), and qualification (dec.qual) if type is ref.

4. The block level (bl).

5. The data record length (reclg).

6. Prefixed block bit (prec =/= none and deq == none).

7. Object bit (refable[bl]).

8. Local classes bit (contclass).

9. The number of virtual quantities (virtrec.totvirt).

10. The number of parameters of a procedure (prec.npar).

11. The instruction address of the first array declaration (decbeg).

12. The instruction address of the first statement (statbeg).

13. The runtime address of the matching quantity for each virtual quantity (normally virtrec.actad[i]; for a procedure virtrec.actq[i].descr.pad, i=1,2,...,virtrec.totvirt)

14. A complete description of each formal parameter of a procedure (prec.loc[i] (type,kind,categ,qual,addr), i=1,2,...,prec.npar).

15. For each type ref declaration or array segment or text quantity, the type, kind, the number of declared quantities, and the relative record address of the first one are given.

The same information is given for each ref or array parameter to a class. (In this case the actual parameter correspondence can always be checked at compile time, and the parameter values can be stored in a generated object by the calling sequence itself). Entries of this type are generated by the following algorithm

```
begin integer i,n; n := 1;
   for i := nvirt+1 step 1 until nloc do
   inspect refto loc[i]do
           if last then begin
             if kind = simple and (type = ref or type = text)
                                    or kind = array then
             entry15(type,kind,n.(if categ≠virtual then addr
                    else virtrec.actad[addr])-n+1);
             n := 1 end
           else n := n+1;
end;
```

where "entry 15" outputs an entry of type 15 in the required format. It is assumed that the quantity attribute "last" is true for any procedure, class, switch or label and for each formal parameter.

16. The prefix level (plev)

The following information may be useful as part of a prototype, but is not essential: the brecord attribute "finbeg", the identifier of a class or procedure block, a line number coordinate in the source text. The first piece of information is required by the runtime system described in the second part of this documentation.

The prototype address (relative to the beginning of the runtime prototype area) is defined at allocation time, see "allocate". The address (pad) should be included in the output text as information to pass 4 since the prototypes are output in an order different from that of the allocation.

2.0 INTRODUCTION - RUNTIME SYSTEM

Formal description of runtime system routines are given in a SIMULA 67 like language.

The difference of the language used for description, as compared to ordinary SIMULA 67, is as follows:

1. label variables and the denotes operation for labels have been introduced with an obvious meaning. Labels as in SIMULA will be considered to be label constants.

2. The basic symbol exit has been introduced, meaning the place where a procedure was called. exit is a designational expression.

3. Arithmetic and comparison operations on ref expressions have been included with an obvious meaning:

   1. The expressions "ref + integer", "ref - integer", "integer + ref", "integer - ref", treat the reference as an integer, the address designated by the ref quantity.
      The result is an integer.

   2. The assignment

      ref variable :- integer expression;

      causes the value of the integer expression to be stored in the ref variable as an address.

   3. Value relations between ref expressions are defined by adding the integer 0 (zero) to both operands, applying rule 1 above, and use the conventional rules for relations between arithmetic expressions.

4. The notation refto X is used to obtain a reference to a quantity X. It is a reference expression.

5. The notation val X is used to refer to the value of a quantity referred to by X.

6. The notation ref(program) has been used for reference to memory cells (program points).

The term "textual link" ("static link") means a pointer from the driver of a block instance to the driver of the block instance textually enclosing the former. An example is: the textual link of a procedure P is to the driver of the block instance B where the procedure is declared.

The term "dynamic link" means a pointer from the driver of a block instance to the driver of the block instance dynamically enclosing the former. An example is: the dynamic link of a procedure P is to the driver of the block instance B where the procedure is called.

The abbreviation B.I. will be used for "block instance".

2.1 Driver technique

Since a terminated B.I. seldom requires the system information that was necessary prior to the termination, it is natural to try to find a method so that this information may be discarded when the B.I. terminates.

One such method is the driver technique. The system information in the B.I. itself is reduced to a minimum, the rest is stored outside the object in a "notice".

There are two kinds of notices: "drivers" containing system information relevant to any kind of B.I., and "eventnotices" containing sequencing information for an active or suspended process.

All notices are assumed to be of the same size.

2.2 Storage organization

When the driver technique is used, data core storage (assumed to be continuous) is dynamically divided into two distinct parts called POOL 1 and POOL 2. Notices are allocated from POOL 2, all other storage from POOL 1.

Storage is allocated from POOL 1 simply by moving a pointer towards POOL 2. POOL 2 uses a similar technique only it has in addition an available storage list of the last in - first out type. On this list, notices are kept chained when not in use.

The head of this list is a ref (notice) nothead declared in the runtime system.

In the following it is assumed that POOL 1 is from the "lower end" of available storage while POOL 2 is from the "upper end".

Four ref variables declared in the runtime system serve as boundary pointers for the two pools of storage: POOL1FIRST, POOL1LAST, POOL2BOTTOM, POOL2TOP.

Two of these are fixed when execution of the SIMULA program is initiated: POOL1FIRST points to the first word of available storage, POOL2BOTTOM points to the last word of available storage where a notice may start.

The two other ref variables, POOL1LAST and POOL2TOP determine the boundaries of POOL 1 and POOL 2 respectively.

POOL2TOP points to the first word of the uppermost notice (whether available or not). POOL1LAST points to the first word following POOL 1.

A storage collapse condition exists whenever a storage assignment would cause POOL2TOP to become less than or equal to POOL1LAST + 1. (At least storage for one integer must be available for the store collapse between POOL 1 and POOL 2.)

Fig. 1 indicates the usual situation during program execution (proportions of POOL 1 and POOL 2 are distorted).

During program execution, the ref (driver) variable CD ("current driver") in the runtime system will always point to the dynamically innermost driver in the current operating object.

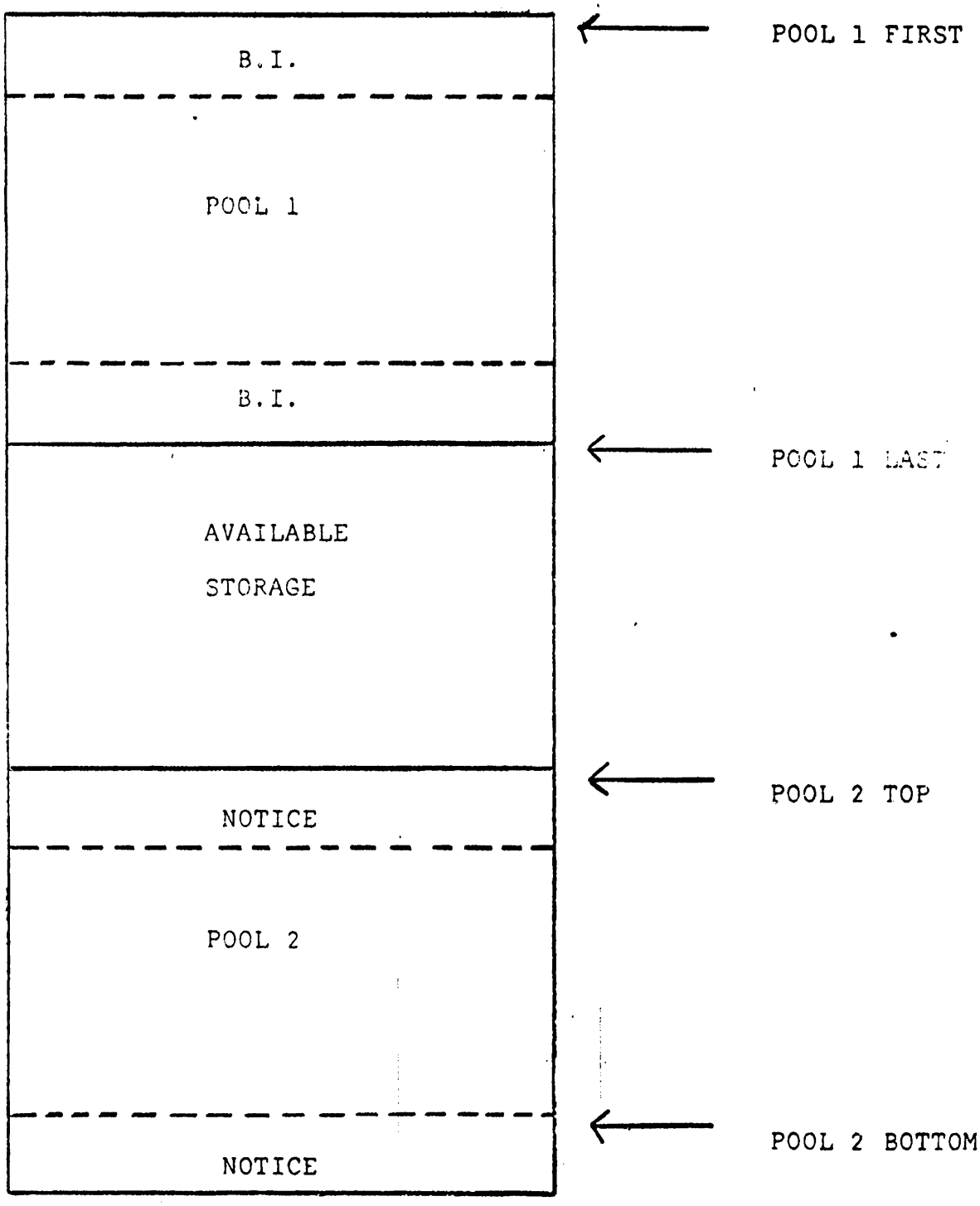


Fig. 1. Core Storage Layout during program execution.

2.3 The classes "object" and "prototype"

A logical unit of information in POOL 1 will be called a "block instance". Examples of data blocks are: instance of a subblock, procedure instance or class instance (without their local arrays), an accumulator stack or an array.

A block instance is an instance of a subclass of the (fictive) class "object". For each family of data blocks (different instances of the same block constitute one family) there exists one compiler generated "prototype" describing that family. An object contains a reference to the corresponding prototype, called the "prototype pointer" (abbreviated PP).

Arrays and accumulator stacks are considered two special families. They have no prototype. The relevant information is contained within the objects.

Special values of the prototype pointer PP are used to indicate arrays and accumulator stacks.

The use of each variable declared in class prototype is described below:

Type	Name	Usage
integer	lg	Total length (including that of data for possible prefixes) of a B.I. of this family. This includes necessary space for description of each array declared within the object.
integer	nvirt	Number of virtuals in B.I.'s of this family.

Type	Name	Usage
integer	nrp	Number of parameters for B.I.'s of this family.
integer	nrl	Number of local pointers in this family.
integer	level	Static block level of the declaration of this family.
integer	vtype	For procedure, type of result.
ref(program)	statements	Address of first statement in body.
ref(program)	inretur	Return address from statement inner; in this class. none if not prototype for a class.
ref(program)	declare	In-line coding to perform declarations local to the object. For a class, or a prefixed block, this in-line coding returns to the run-time system. In all other cases, the in-line coding will continue with the first statement.
integer array	relad	Relative address in data object for parameters and local pointers.
Boolean array	valu	true if and only if value parameter.
ref array	progaddr	Either: program address (for virtual labels and switches) or pointer to procedure prototype (for virtual procedures).

Type	Name	Usage
Boolean	pb	true if and only if this family is a prefixed block.
Boolean	ob	true if and only if this family is a class or a prefixed block.
Boolean	local classes	true if and only if this family has local class declarations.
ref (prototype) array	prefix	Pointers to prototypes for the prefix hierarchy of this family. Prefix (0) is the outermost prefix.
ref (program)	endblk	Address of statement after end of a prefixed block.
integer	plev	Prefix level (i.e. the number of prefixes in the prefix chain of this block).

As mentioned earlier there are two kinds of notices: drivers and eventnotices.

The ref variable "notc" is only used during store collapse to indicate where a notice has been moved. In this case the actual contents of the notice is not used. Thus "notc" may, in actual implementation, occupy the same part of a notice as for instance "obj".

"evp" and "acs" are mutually exclusive, and may thus use the same space in the driver.

In an actual implementation, the pointers to POOL 2 may be made relative by assuming a maximum size of POOL 2. This also applies to pointers from POOL 1 to POOL 2, for instance "MDP".

The use of each variable declared in class notice and its subclasses driver and eventnotice are described below.

Type	Name	To	Usage
ref (object)	obj	POOL1	Points to the B.I. belonging to this notice. The value none is illegal for "obj" except when the driver has not been completed. The store collapse must be able to handle the case "obj == none" even for master drivers.
Boolean	referenced	n.a.	false always except when in store collapse where it is set true if notice is referenceable.
ref (notice)	notc	POOL2	Used solely during store collapse. When a notice has been moved, "notc" in the old notice will indicate where the new notice has been put. Used for update of pointers to POOL 2.
real	time	n.a.	Evtime for the process referenced by obj of this eventnotice.
ref (eventnotice)	BL,LL,RL	POOL 2	See the section on organization of the sequencing set.
ref	pex	program	Program address descriptor: for procedures, thunks and attached objects: return point (exit). For detached objects and prefixed blocks: program point where object or prefixed block is to resume operation.

Type	Name	To	Usage
ref (driver)	cdrp	POOL 2	Pointer to the driver of the B.I. statically outside the connected B.I. (May be dropped if compiler remembers the block level of the class declaration).
ref (object)	acs	POOL 1	Pointer to accumulator stack which was saved when this driver was created. The value none indicates no accumulator stack.
ref (driver	drp	POOL 2	Pointer to the driver of the B.I. statically outside the B.I. belonging to this driver.
ref (driver)	drex	POOL 2	Pointer to the driver of the B.I. dynamically outside the B.I. belonging to this driver, except for prefixed blocks and detached objects when it is a pointer to the driver of the dynamically innermost B.I.
ref (driver)	drch	POOL 2	Used in store collapse to chain drivers for later processing.
Boolean	con	n.a.	true if and only if the driver is a connector or a thunk driver acting for a connector.
Boolean	rp	n.a.	true if and only if this is the driver of a detached object or a prefixed block.

Boolean	pb	n.a.	true if and only if this is the driver of a prefixed block.
Boolean	dot	n.a.	true if and only if this is the driver of a procedure called by remote referencing (dot-notation).
Boolean	md	n.a.	true if and only if this driver is a master driver.
Boolean	ob		true if and only if this is the driver of an attached or a detached object or a prefixed block.
ref (eventnotice)	evp	POOL 2	Pointer to the eventnotice of the process belonging to this driver. The value is none if "obj" does not point to a suspended or active process.
integer	level	n.a.	Apparent block level of this driver.

Below is a summary of the contents of drivers for different block states. Note that the contents of "obj" and "drp" are not shown since their contents are the same for all drivers.

"drch" and "notc" are also omitted since they are only used during store collapse.

For the Boolean variables, f is used for false and t for true.

	pex	drex	evp	acs	con	pb	rp	md	ob
Subblock	none	equal to drp	none	none	f	f	f	t	f
Procedure or attached object	exit	pointer to driver of dyn.outer block	none	pointer to acc. stk.(3)	f	f	f	t	(7)
Thunk (5)	exit	pointer to driver of block where thunk is called.	none	pointer to acc. stk.(3)	(5)	f	f	f	f
Prefixed block	react. point. (1)	pointer to innermost block (4)	none	none	f	t	t	t	t
Detached class body	react. point (1)	pointer to innermost block (4)	none (2)	none	f	f	t	t	t
Connector (6)	none	pointer to driver of the dyn. outer blk.	none	none	t	f	f	f	f

(1) Conceptually none if operating B.I. (Cfr. Common Base section 8).

(2) Pointer to eventnotice if active or suspended process.

(3) none is a possible value meaning "no accumulator stack".

(4) Disregarded for innermost operating B.I.

(5) "con" and "cdrp" as driver of calling B.I.

(6) "cdrp" is equal to "drp" of the connected B.I.

(7) false for procedure, true for attached B.I.

2.4 Display

For reasons of efficiency, the static chain may be duplicated in a vector of fixed locations (registers or core storage) called a display.

The display may be explicit as a vector or implicit by the static links (drp) and B.I. pointers (obj) in the drivers.

We may, when using driver technique, talk of two different displays:

1. DISPLAY pointing to B.I.'s

2. DDISPLAY pointing to drivers

Obviously it is possible to establish either of these from the information in CD.

If DISPLAY is kept in index registers, the number of possible static levels in a program may be limited. This number should however never be less than 8.

DDISPLAY may be partially present or completely absent in some implementations.

DISPLAY and DDISPLAY must be updated whenever the value of CD is changed, except in the case of exit from a subblock, connection block or a prefixed block.

2.5 Actual prototype implementation

Information in the prototypes is heavily used at runtime. This is most obvious in the store collapse, where the prototype must be scanned once for every referenceable B.I. Even when a B.I. it is not referenceable, the length (lg) of the data universe must be found in the prototype.

Prototype information is also used when a block, prefixed block, procedure or class B.I. is created.

Since, in other cases, the access to prototype information usually would be indirect (through the prototype pointer, PP, in the object), part of the prototype information may be duplicated in the drivers.

The following should be noted about the information kept in the prototype in the formal description:

lg — Required item. Without it, the length of a B.I. could not be found at runtime. To avoid accumulating length of all prefixed at runtime, this length is the total length including data for the prefixes.

ob,pb — Required.

nvirt — Required in one form or another. Virtual descriptors must be located at fixed positions within the prototype relative to PP. Thus, nvirt is required to skip these descriptors when scanning the prototype during store collapse.

nrp — Required for procedures to compare the actual number of parameters to the number given in the procedure declaration. This information is used when a formal or virtual procedure is called.

nrl — Redundant if a special end prototype signal is introduced in the prototype.

level — Required.

statements — Required for classes and prefixed blocks. May be omitted for subblocks and procedures if the compiler generates a jump from end of declarations to first statement.

inretur — Required if the compiler does not generate the end of a subclass as an in-line return following inner; of the prefix.

endblk — Required for prefixed blocks.

declare — Required for procedures, classes and prefixed blocks. May be omitted for subblocks if the declarations immediately follow call on BB.

prefix — Required for classes and prefixed blocks.
Prefix is used to get fast transition from declaration code in the prefix to declaration code in the main part, as well as fast transition from the declaration code in the main part to the statements in the outermost prefix.

relad
kind
type
valu
} Required as part of the individual descriptors.

virtloc
progaddr
} Required as part of virtual descriptors.

local classes — Required for driver deallocation purposes.

type Required for procedures. For a ref procedure it could be a pointer to the prototype of the class qualifying the result.

plev Required.

2.6 Wholesale deallocation

The present section describes a possible technique for wholesale deallocation in SIMULA 67 Common Base program. The formal description of the runtime system does not include a description of the allocation scheme and the store collapse required for this technique.

The natural dividing lines between parts of a SIMULA 67 program are the prefixed blocks. If a wholesale deallocation technique should be used, the fact that no non-local references may refer to anything local, is of great importance. This property is inherent in prefixed blocks, sub-blocks and procedures. In the present approach, only prefixed blocks and sub-blocks will be considered.

The approach described makes wholesale deallocation of data objects possible. Drivers are assumed to be deallocated separately.

POOL 1 is assumed to contain a list of descriptions of the prefixed blocks and sub-blocks in the system at each instant. This is a dynamic description where a declared block may be represented more than once due to its use within an object or due to recursive use of procedure. Since the size of this list varies dynamically, it must be possible to expand and contract this list.

The rest of POOL 1 is divided in as many parts as there are prefixed blocks and sub-blocks in the system at any time.

The compacting garbage collector is called whenever an area overflows into that of another block. Only the area that overflowed is compacted. Possibly, some part of the available area for the block preceding this one in core is stolen.

If area compacting does not yield a sufficiently good result, the entire store is compacted and new areas assigned.

Upon generation of a class B.I., its storage, including that for local arrays, is taken from the area assigned to the block where the declaration of the class is found. The local storage for a procedure may go into that of the block surrounding the call.

If a denotes operation is available for arrays (this is not the case within the Common Base), arrays could be put into a separate area. The array area would be compacted only when the entire store was compacted.

When a prefixed block or sub-block is left, the entire area assigned to that block is made available simply by removing its descriptor from the descriptor table. Drivers may be deallocated by the sequential scan of the area.

3. TYPES

3.1 Arithmetic types

The SIMULA 67 Common Base has two basic arithmetic types:

integer and real

3.2 The type Boolean

A variable of type Boolean may only assume truth values, i.e. true or false.

The internal representation and packing of Boolean quantities may be decided by the implementor. It is recommended that Boolean arrays be packed to ensure storage efficiency.

3.3 The type ref

A variable of type ref may denote ("be a name on", "refer to") an instance of a class declaration.

The qualification of a ref variable limits the range of values that this variable may assume. If the qualification of a ref variable is C, this variable may only denote instances of the class C or a subclass of C. An exception is the anonymous subclass of C formed by using C as a block prefix. The rules for the use of the local reference "this" prohibits generation of references to a prefixed block.

3.4 The type character

The type character is introduced to handle one-character alphanumeric information.

3.5 The type text

The type text is introduced to handle sequences of alphanumeric information.

Each declared text has a built-in sequential facility that make its use as an input and output buffer fairly easy.

The type text is the only type within the Common Base which has local procedures.

A procedure of type text is permitted both on the left and on the right hand side of an assignment statement.

3.6 Arrays

It is understood that arrays of any of the six types mentioned may be formed. It is recommended that Boolean and character arrays are packed.

3.7 Type procedures

Each of the six types may be used to form type procedures.

3.8 Initial values

The SIMULA 67 Common Base requires that the following initial values are given to local variables when a block, procedure or class is entered:

integer	zero
real	zero
Boolean	false
ref	none
character	implementation defined
text	notext

The same initial value shall also be associated with type arrays and the value associated with the name of a type procedure.

4. SUB-BLOCKS

An anonymous sub-block is represented in the compiled program by (1) a prototype and (2) the coding related to the block, enclosed by a begin block (BB) and an end block (EBL) "subroutine" call.

Storage for data local to the block (the B.I.) and a driver for the B.I. are created when the block is entered, i.e. the PSC encounters the PP call.

The update display mechanism is simplified, in that DISPLAY is correct for all levels except that of the block itself.

When DISPLAY has been updated, the declarations and statements of the block are performed in the indicated sequence. There is no need to return to the runtime system between the last declaration and the first statement.

It should be noted that "statements" is unnecessary if end of declaration code for a sub-block contains a jump to the first statement.

It is not possible to establish references to a sub-block or its interior.

```
procedure BB(p); ref (prototype) p;
  begin
     CD :- new driver(new object(p),CD,none,CD,none,true,
                                                    p.level);
     CD.obj.MDP :- CD;
     DISPLAY [p.level] :- CD.obj;
     DDISPLAY [p.level] :- CD;
     go to p. declare;
  end BB;
```

```
procedure EBL;
  begin ref (driver) x;
    x :- CD.drp;
    deletenotice (CD);
    CD :- x;
  end EBL;
```

5. PROCEDURE DECLARATION

A procedure deviates from a block in that (1) it has a name and (2) may be referred to at several different places in the program, and (3) that parameters may be given to it when invoked. A procedure shares the property of a block that it is impossible to establish a reference to it or to its interior.

A procedure declaration found in a block heading serves to define the procedure within that block. The declaration is not executed at block entry. A procedure call serves to invoke a procedure, i.e. create data storage and transmit actual parameters. When the end of the procedure body is reached, control returns following the call on the procedure.

In order to refer to the actual parameters supplied to the procedure when invoked, a series of formal parameters is given in the procedure declaration. The actual and formal parameters are matched by their position in the parameter lists when the procedure is invoked.

The formal parameters must be specified. The actual parameter must be compatible with the formal parameter.

A parameter to a procedure may be specified as being one of the six types as well as label, switch, procedure, type procedure and type array.

An additional specification, name or value, may be given. If neither of these are present, a default specification will be assumed dependent upon the mandatory specification mentioned above.

If the declarator procedure is preceded by a type, this indicates that a value of the given type should be associated with the procedure name. Its initial value should be as previously defined for local variables.

If the name of a procedure is used on its own within the procedure body on the left hand side of an assignment or denotes operator, this indicates an operation upon the value associated with the procedure. If it is given on the right hand side of an assignment or denotes operator it will be a recursive call of the procedure unless it is on its own and another assignment or denotes operator is found on its right hand side in the same statement.

As for sub-blocks, it is unnecessary to treat declarations and statements within the body as separate items, as a jump from end of declaration code to statements may be made unconditionally.

The end of the procedure body is indicated by a call on the end procedure (EPR) subroutine.

There are several alternatives open to the implementor for organization of the reference to parameters specified as being called by name.

SIMULA has adopted the scheme from Algol 60 that the location of an actual parameter or subscripted variable on the left hand side of an assignment or denotes operator should be evaluated prior to evaluation of the right hand side.

It would thus be natural to have three subroutines dealing with parameter evaluation: CPL (calculate parameter location), SP (store parameter) and LP (load parameter).

The actual and the formal parameter should be compatible. This means for instance a real variable in a procedure call may correspond to a formal parameter of type integer and vice versa. Several other cases exist which should be properly handled by the runtime system.

```
universal procedure EPR (val);
   begin ref (driver) x; ref (program) out;
     x :- CD.drex;
     out :- CD.pex;
     restore (CD.acs);
     if CD.dot then deletenotice (CD.drp);
     deletenotice (CD);
     CD :- x;
     EPR := val;
     update display;
     go to out;
   end EPR;
```

Comment: "val" is the value to be returned by the procedure from which we are currently returning. Since the type is immaterial to the description of EPR, "val" has not been specified. EPR is called a "universal" procedure, because its value depends on where it was called.

6. CLASS DECLARATION

A class declaration defines the class associated with a class identifier. Instances of a the class may be created dynamically. The formal parameters, virtual quantities and local quantities are called the "attributes" of the class. The statements within the class are called the "operation rule" of the class.

The end of the operation rule of a class is indicated by a call on the end class body (ECB) subroutine which will return following inner in the prefix (if any).

```
ref (object) procedure ECB(p); ref (prototype) p;
   begin ref (program) out; ref (driver) x;
      procedure delete;
         begin x.rp := false;
           if x.obj. PP.local classes then
           begin x.drex :- x.drp; x.pex :- none; x.acs :- none; end
           else begin x.obj.MDP :- none; deletenotice (x) end
         end delete;
      if p.plev ≠ 0 then go to p.prefix [p.plev-1].inretur;
      x :- CD;
      if CD.rp and not CD.pb then
      begin CD :- CD.drp; delete; go to L2;
      L1:   CD :- CD.drp;
      L2:   while not CD.rp do CD :- CD.drex;
            if not CD.pb then go to L1;
            x :- CD.drp;
            while not X.rp do x :- x.drex;
            x.drex :- CD;
      L3:   if CD.pex =/= none then go to L4;
            CD :- CD.drex;
            go to L3;
      L4:   out :- CD.pex;
      end else
```

```
    begin
        if x.pb then
        begin CD :- x.drp;
           out :- x.obj.PP.endblk;
           deletenotice (x);
           go to ud
        end else
        begin out :- x.pex;
           CD :- CD.drex;
           ECB :- x.obj;
           restore (x.acs);
           delete
        end
      end;
      update display;
      ud:   go to out;
end ECB;
```

If this class has a prefix, return after the statement inner; in the prefix.

The compiler may determine this. In that case, return is compiled directly into the class body and ECB is not entered.

If the object has local class declarations, the master driver is made special by setting "pex" to none and the dynamic link (drex) equal to the static link (drp).

Note that "md" must not be changed in this case since the driver must remain a master driver.

If the object has no local class declarations, the driver is deleted (put on the available storage chain for POOL 2) by a call on "deletenotice".

The accumulator stack is restored. The function value of this procedure is a pointer to the B.I., except for a prefixed block or a detached object when it is none.

DISPLAY is updated.

6.1 Subclasses

In the declaration of a class D, the class may be prefixed by another class C. This defines D as a subclass of C. C may itself have a prefix. The sequence of prefixes A, ..., C is called the prefix chain of D. The prefix chain for a class C may not include a subclass of C.

The prefix to a class C is limited to be either a system defined class or a class declared within the same block as the declaration of C.

Declarations and operation rules of a class and its prefix are concatenated according to the rules given in the Common Base.

At runtime, the fact that the class D has the class C as prefix may be indicated by a pointer, "prefix", from the prototype of D to the prototype of C.

Since the declarations D1,.....,Dn and statements S1, ...,Sn in the prefix sequence P1,.....,Pn-1 of the class Pn by definition should be executed in the order first declarations, then statements, it is advantageous to have a pointer from the prototype of Pn to the prototype of each of the classes in the prefix hierarchy.

A class Pj may have a "split body", which explicitly indicates that if the object is of a class Pn which is a subclass of Pj, the concatenated operation rule of Pj+1, ...,Pn should be executed prior to continuing the operation within Pj. A split body of a class Pj is

indicated by the occurrence of the statement inner; in the declaration of Pj. A class declaration with no inner; statement should be regarded as having an implicit inner; at the end of the operation rule.

When Pj+1 has Pj as prefix, the return point at the end of the operation rule of Pj+1 should be after the inner; statement of Pj. It is worth giving the compiler some intelligence at this point. At the end of the operation rule of the Pj+1, an unconditional jump could be compiled to the statement following inner; in the innermost of the classes P1,.....,Pj which has a split body. If neither of these have a split body, a call on (a slightly simplified) ECB subroutine is compiled which in fact indicates the end of the concatenated operation rule.

The formal definition here is written assuming no such compiler intelligence.

A call on the call inner (CINNER) subroutine represents the explicit or implicit inner statement.

```
procedure CINNER(lev); integer lev;
        begin ref (prototype) p;
           p :- CD.obj.PP.prefix[lev+1];
           if p =/= none then
           go to p.statements
end CINNER;
```

6.2 Parameters

The parameters given when an instance of a class is generated are matched against the formal parameters as described for procedures. The formal parameters must be specified, and permissible specifications are type and type arrays. The type of an actual parameter must be compatible to the type specified for the formal parameter.

Parameters of types integer, real, Boolean and character are by definition called by value, while ref and text parameters are called by "copy". Arrays are called by "copy description". The option exists for the user to specify text and arrays as being called by value.

It is suggested that parameters to an instance of a class are stored within the class instance by in-line coding which is a part of the calling sequence.

6.3 Virtual quantities

A virtual quantity V specified in a class Pj in the prefix sequence Pl,.....,Pn will be replaced by the innermost declaration of a matching quantity V of the sequence Pj, ...,Pn. Suppose that this is in Pk. Then the replacement is valid also for Pj,.....,Pk-1. This is implemented by having, in the prototype for the class Pn, one item for each specified virtual quantity in the classes Pl,.....,Pn. These are in fixed positions within the prototype, so that the virtual quantity valid in a specific instance may be found when its index and the class of the B.I. (i.e. its prototype) is known.

In the Common Base, the following specifiers are accepted for virtual quantities:

label, switch, procedure and <type> procedure

A virtual quantity in an instance of a class or a prefixed block will be called "closed" if a declaration matching the specification exists within the object. A virtual which is not closed will be called "open".

It should be noted that a reference to a virtual quantity always must be through the prototype pointer of the actual object.

Subroutines associated with virtual quantities are:

CCVP	Call connected virtual procedure
CDVP	call dot virtual procedure
CVP	call virtual procedure
ENTVIRT	enter virtual procedure
GVL	go to virtual label
CVS	calculate virtual switch

These are discussed in the sections on expressions and labels and switches respectively.

7. EXPRESSIONS

7.1 Procedure call

When a procedure is called, we must distinguish between four kinds of procedures: non-formal-non-virtual, virtual, formal and external. Their characteristics are as follows:

For a non-formal-non-virtual procedure, the compiler may check that the number of actual parameters is correct and that the actual parameters are compatible with the formal parameters. Value parameters may be transmitted by compiler generated (in-line) coding.

For a virtual procedure, these checks must be made at runtime. The compiler does not know which parameters are called by value. In fact, the compiler does not even know that a matching virtual procedure exists. The access to the procedure must be through the prototype of the object where the procedure is local.

For a formal procedure, much of the same checking must be done as for virtual procedures.

For an external procedure, the checking of ref parameters may be simplified as compared to formal procedures, since ref parameters to an external procedure may only be qualified by a system class.*)

In the formal description, a ref (driver) parameter has been used instead of a display index because it is easier to understand. The use of a display index would be simpler in an implementation.

We have, when calling a non-formal (non-virtual or virtual) procedure, P, three different cases:

*) This point is currently being clarified by the SIMULA Standards Group.

1. P is normal, i.e. its name is statically visible without connection. In this case, only one driver is required. The static link (drp) of this driver is found in DDISPLAY(bl) at the level of the block where P is declared.

2. P is connected, i.e. its name has become visible through a connection. Two drivers are required. The first driver is an ordinary procedure driver with a static link to the other extra driver. For the second driver con is true, pex and drex are none, obj is found in DISPLAY (blc) where blc is the level of the connection block where the object in which P is declared is connected, and drp is found in DDISPLAY(blc).cdrp where blc is defined as above, or alternatively as DISPLAY[bld-1].MDP where bld is block level of class declaration.

3. P is remote, i.e. accessed by remote referencing, and the class of the element expression, **X**, preceding the dot preceding P is C (P is thus local to C). **X** must be supplied by the compiler as a parameter to the runtime system. Two drivers are required: an ordinary procedure driver with a static link (drp) to the extra driver. The extra driver has a static link (drp) found in ddisplay (blc) where blc is the apparent block level of the class C, con is true, pex and drex are none and obj is **X**.

7.1.1 Non-formal non-virtual procedure

A procedure call P(A1,.....,An) is assumed to create the following in-line coding in the compiled program:

1. A call on a call procedure runtime routine to generate the date storage for the procedure, insert PP, create necessary drivers and prepare to accept value parameters and name parameter descriptors through in-line coding.

2. In-line coding to compute and store dynamic parameter descriptors and value parameters.

During the value parameter evaluation, the procedure itself has two drivers.

The first is an ordinary procedure driver, the second is a thunk driver which is deleted when all value parameters have been computed and stored.

If the number of value parameters is zero, the second driver is not created.

The contents of the thunk driver are as follows:

obj	pointer to B.I. B where the procedure is called
drp	equal to drp of B
pex	none
drex	pointer to the procedure driver
acs	none
md	false

con and cdrp are inherited from the driver of B

When all parameters and descriptors have been evaluated, the second driver is discarded and display is updated to point to the static environment of the procedure.

The compiler generated coding for the procedure is entered. In this coding, the local declarations are performed and the procedure body entered.

The parameter "acs" is used to indicate that an accumulator stack may be present. The corresponding actual parameter will be either none or a generating expression. This does not imply that the accumulator stack need be saved with in-line coding. In an implementation, acs will probably be a description of the accumulator stack, and the save function is performed by the runtime system.

```
procedure ENTER;
     begin ref (driver) y;
        y :- CD;
        CD :- CD.drex;
        CD.pex :- exit;
        deletenotice (y);
        update display;
        go to CD.obj.PP.prefix[0]. declare;
end ENTER;
```

Note: procedure prototype have prefix [0] == prototype of procedure.

A call on the subroutine "ENTER" is compiled at the end of the in-line coding for parameter transmission to procedures and classes.

```
procedure ENTPROC;
     begin ref (driver) y;
     CD.obj.MDP :- CD;
     if CD.obj.PP.nrp = 0 then begin update display;
                                  go to CD.obj.PP.declare
                                end;
     y:- CD.drex;
     CD :- new driver (y.obj,y.drp,none,CD,none,false,
                                                    y.level);
     CD.con := y.con; CD.cdrp :- y.cdrp;
     DDISPLAY[y.level] :- CD;
     go to CD.drex.pex;
end ENTPROC;
```

If the procedure has no parameters, "ENTPROC" will cause the in-line coding for declarations local to the procedure to be entered. This in-line coding continues with the first statement of the body.

```
procedure CPR (p,acs);
     ref (prototype) p; ref (object) acs;
     begin
     comment call procedure;
        CD :- new driver (new object(p),DDISPLAY[p.level-1],
                                        exit,CD,acs,true,p.level);
        ENTPROC;
     end CPR;
```

A driver for the procedure is created and the procedure is entered using "entproc" (described above).

```
procedure CCP (p,c,acs);
     ref (prototype) p; ref (object) acs;
     ref (driver) c;
     begin ref (driver) x;
     comment call connected procedure;
         x :- new driver(c.obj,c.cdrp,none,none,none,false,
              p.level-1);
         x.con := true;
         CD :- new driver(new object(p),x,exit,CD,acs,true,
                                                 p.level);
         CD.dot := true; ENTPROC;
     end CCP;
```

CCP is used when a procedure belonging to a connected object is called. It creates a substitute driver for the connected object, the driver for the procedure and the procedure universe, and enters the procedure using "ENTPROC" (described above).

```
procedure CDP (p,c,slc,acs);
     ref (prototype) p; ref (object) c,acs;
     ref (driver)slc;
     begin ref (driver) x;
     comment call dot procedure;
```

```
        x :- new driver (c,slc,none,none,none,false,p.level-1);
        x.con := true;
        CD :- new driver (new object(p),x,exit,CD,acs,true,
                                                    p.level);
        CD.dot := true;
        ENTPROC;
    end CDP;
```

A substitute driver is created for the object where the procedure is declared.

A driver for the procedure is created with a static link to this substitute driver. Dot of the procedure driver is true to indicate that both dynamic and static links must be followed by the store collapse, and that the substitute driver should be deleted at procedure exit.

The procedure is entered through ENTPROC.

7.1.2 Virtual procedure

The number of parameters and their types are not checked by the compiler for a call on a virtual procedure. The checking is left to the runtime system subroutines.

A virtual procedure, V, when called, may be of either of the cases mentioned for non-formal - non-virtual procedures.

A call on a virtual procedure V(A1,.....,An), is assumed to create the following in-line coding in the compiled program:

1. A call on a "call virtual procedure" runtime routine to:

    1. Generate the data storage for the procedure.
    2. Insert PP.
    3. Create necessary drivers.
    4. Check parameter numbers and types.
    5. Compute value parameters.
    6. Insert name parameter descriptors.
    7. Enter the procedure (starting with local declarations).

2. The number of parameters

3. Static parameter descriptor (spd) for each parameter.

```
procedure entvirt (p,dr,dot,acs);
   ref (prototype) p; ref (driver) dr;
   ref (object) acs; Boolean dot;
   begin check number of parameters etc;
      CD :- new driver (new object(p),dr,exit,CD,acs,true,
                                                  p.level);
      CD.dot := dot;
      CD.obj.MDP :- CD;
      store descriptors for name parameters;
      store value parameters;
      update display;
      go to p.declare
   end entvirt;

procedure CVP (cl,index,acs);
   integer index;
   ref (object) acs; ref (driver) cl;
   begin ref (prototype) p,q;
      p :- cl.obj.PP;
      q :- p.progaddr (index) qua prototype;
      if q == none then error ("cvp",1);
      entvirt (q,cl,false,acs)
   end CVP;

procedure CCVP(c,index,acs);
   integer index;
   ref (object) acs; ref (driver) c;
   begin ref (prototype) p,q;
       p :- c.obj.PP;
       q :- p.progaddr (index) qua prototype;
       if q == none then error ("ccvp",1);
       c :- new driver (c.obj,c.cdrp,none,none,none,
            false, p.level);
       entvirt (q,c,true,acs)
   end CCVP;
```

```
procedure CDVP(c,index,slc,acs);
   integer index;
   ref (object) c,acs; ref (driver) slc;
   begin ref (prototype) p;
      p :-  c.PP.progaddr(index) qua prototype;
      if p == none then error ("CDVP",1);
      slc :- new driver (c,slc,none,none,none,
                         false,p.level-1);
      slc.con := true;
      entvirt (p,slc,true,acs);
   end CDVP;
```

7.1.3 Formal procedures

A formal procedure may only be normal, not remote or connected.

The dynamic parameter descriptor for an actual procedure or <type> procedure corresponding to a formal parameter specified as procedure or <type> procedure must contain the following information:

1. Pointer to the prototype for the procedure.
2. Pointer to driver of object where procedure is declared.

The number of parameters and their type given in the call will be matched against the parameter requirements of the procedure parameters.

Type differences will be tolerated only for value parameters with type compatible with the declared type.

7.2 Arithmetic and Boolean expressions

These follow the rules as defined in ALGOL 60 with the exception of <factor>↑<term> which is of type real.

7.3 Ref_expressions

7.3.1 Validity

The qualification of a reference may be divided into two parts: a static qualification (the class) and a dynamic qualification (the object instance where the class is local).

The dynamic qualification is not in general known at compile time. The need to check the dynamic qualification would be a heavy burden on the runtime system in the general case.

The implementation is greatly simplified by the following restrictions in the Common Base:

1. Quantities declared within a class containing local class declarations may not be accessed by dot-notation.

2. Synonomous classes whose apparent block levels are different, are assumed by the compiler to have different dynamic qualifications.

These two restrictions imply that the dynamic qualification may be completely checked at compile time, except in the case of ref parameters to a formal or virtual procedure. In these exceptional cases, the dynamic qualification must be checked at procedure entry time.

The check is performed by comparing BL prior to entry of the procedure with BLF, where BL is the apparent block level of the qualification of the actual parameter and BLF the one for the formal parameter.

In addition, the qualification compatibility check normally performed at compile time must be performed at runtime when a formal or virtual procedure is called. The check ensures that one of the qualifying classes includes the other.

It follows that the static qualification as well as the associated apparent block level for each actual and formal parameter must be available to the runtime system at the time of entering a formal or virtual procedure.

The preceding paragraphs also apply for ref array parameters.

In the following four cases, class membership of a reference value must be compared and checked against the static qualification of a variable:

1. Explicit reference assignment, case 2.

2. Reference parameter transmission to non-formal, non-virtual procedures in default mode, case 2.

3. Name parameter to the left of denotes. This check is required even in case 1, since the qualification of the actual parameter may be a subclass of the qualification of the formal one. Since the qualification of the actual parameter is not known at compile time, this check belongs in a system soubroutine which interprets the store operation (SFP).

4. Name parameter in reference expression. A check similar to that of 3 above must be performed in the subroutine which interprets the load operation (LFP) because the formal qualification may be a subclass of the actual one.

7.3.2 Generating reference

The generating reference new C(A1,....,An) is assumed to create the following in-line coding in the compiled program:

1. A call on a "begin class" procedure to generate the C object, create the necessary drivers and prepare to accept parameters as in-line coding.

2. In-line coding to compute parameters and store the values in the C object.

3. A call on the subroutine ENTER which will enter the coding for declarations in the outermost prefix.

During the parameter evaluation, the object has two drivers:

The first one is the ordinary driver for an attached object, the second is a thunk driver which is deleted when all parameters have been computed and stored.

If the number of parameters is zero, the second driver need not be created.

The contents of the thunk driver are as follows:

obj	pointer to block B where generating expression is found.
drp	equal to static link (drp) of block b.
pex	none.
drex	pointer to object driver.
acs	none.
md	false

When all parameters have been evaluated, the second driver is discarded and display is updated. The declarations local to the class are then performed starting with the outermost prefix.

The program of the outermost prefix is then entered.

The driver for the object is given DISPLAY (BL).mdp as static link (drp) where BL is either the level of the block containing the declaration of the class C (normal case) or the level of the connection of the class D where the class C is declared.

Note:

Since the class or block where C is declared must have local class declarations (C is one of these), it must always have a master driver.

The procedures in the formal description relating to generating references are:

BC	begin class
BCR	begin class return
ECB	end class body

In addition, the procedures "detach", "resume" and "attach" may be considered related to generating references.

ECB has been previously described.

```
procedure BC (x,slx,acs);
   ref (prototype) x; ref (object) slx;
   ref (object) acs;
   begin ref (driver) y; ref (prototype) q;
      comment begin class;
      y :- new driver (new object (x),slx.MDP,exit,CD,acs,
                                                  true,x.level);
      y.ob := true; y.obj.MDP :- y;
      if x.nrp ≠ 0 then
         begin y :- new driver (CD.obj,CD.drp,none,y,none,
                                                  false,CD.level);
            y.con := CD.con;
            y.cdrp :- CD.cdrp;
            CD :- y;
            DDISPLAY[CD.level] :- CD;
            go to exit
         end else CD :- y;
            update display;
            go to CD.obj.PP.prefix[0].declare;
      end BC;
```

ENTER has been described previously (section 7.2.1).

BC is entered to create data storage for the instance of the class declaration, make necessary drivers, transfer parameters and enter local declarations of the outermost prefix.

The parameters have been checked by the compiler, and they are stored into the class body by in-line coding in the compiled program. A temporary (thunk) driver is used during this evaluation. Return to the runtime system after the parameter evaluation is by the procedure ENTER.

DISPLAY is updated, and the declaration coding in outermost prefix is entered.

The coding for declarations in each prefix will end by a call on BCR, for a prefixed block BPBR.

```
procedure BCR (q); integer q;
   begin ref (prototype) x,y;
      comment begin class return;
      x :- cd.obj.PP;
      y :- x.prefix[q+1];
      if y =/= none then go to y.declare;
      go to x.prefix[0].statements;
end BCR;
```

The subroutine BCR is used to locate the next class where declarations should be performed. When declarations in the innermost class have been processed, the outermost prefix is located and the first statement entered.

DISPLAY is updated prior to entry of the declaration part.

Return from declarations in a prefixed block is through BPBR.

7.3.3 Instantaneous qualification

An instantaneous qualification (X qua C) will always result in a runtime check. It is verified that X is of class C or a subclass of C.

The case of X being none is handled by compiler generated coding as usual.

The subroutine CIQ (check instantaneous qualification) is used to check the validity of the instantaneous qualification.

The subroutine either gives an errormessage or acts as a ref(C) procedure whose value is X.

```
ref procedure CIQ(x,c);
   ref (object) x; ref (prototype) c;
      begin ref (prototype) d;
         d :- x.PP;
         if d.plev < c.plev then error ("ciq",1);
         if d.prefix[c.plev] =/= c then error ("ciq",2);
         CIQ :- x
   end CIQ;
```

Possible error: the object X is not of class C or a subclass of C.

The subroutine will conceptually follow the prefix chain in the prototypes starting on the prototype for the class of X. If the prototype for the class C is not found during this scan, a runtime error condition exists.

The procedure value is a reference to the block instance.

7.4 Character expressions

Character expressions are in the language only as character constants, character variables and character procedures.

The character procedure "char" is system defined.

The character procedure "getchar" is defined local to the type text.

7.5 Text expressions

Text expressions are in the language as text constants, text variables and text procedures.

The text procedures "copy" and "blanks" are system defined.

The text procedures, "sub", "main" and "strip" are attributes of any text.

8. ACTUAL PARAMETERS

For <type> quantities, the call by value causes an assignment to be made to a local variable. In the same way, a call by default (text only) will cause a denotes operation to be performed to a local variable.

For classes, the parameter checking and type conversion are done by the compiler which will also generate the necessary in-line coding to store the values into the class instance.

For procedures, with the exception of external, formal and virtual ones, the value parameters are checked, type converted and stored within the procedure instance by compiler generated in-line coding.

Name parameters have been partly checked by the compiler in that all errors that do not depend on actual values at runtime are detected.

For external, formal and virtual procedures, the parameter correspondence cannot be checked at compile time. Thus the checking must be done by the appropriate runtime system routines.

For external, formal and virtual procedures, all parameters have a static parameter descriptor (spd).

The contents of an spd are as follows:

All spd's contain type, kind and spd-type information.

Depending on the spd-type information, the rest of the spd contains:

1. A constant address.
   Actual parameter is a <type> constant (real, integer, Boolean, character, text or ref).

2. A block level and a relative data address.
   Actual parameter is a simple variable (except label) or a formal parameter.

3. A block level and a switch (or label) address.
   Actual parameter is a switch (or label) name.

4. A block level and a prototype pointer.
   Actual parameter is a procedure name.

5. A thunk address.
   Actual parameter is a subscripted variable, remote variable or an expression.

Any reference parameter spd must contain or give access to the qualification of the actual parameter, i.e. the identification (prototype pointer) of the qualifying class and its apparent block level.

The appropriate runtime subroutine will, for name parameters, convert an spd to a dynamic parameter descriptor (dpd).

For value and default parameters, a dpd is not formed, but the parameter is evaluated and the result stored within the procedure instance. The exceptions to this rule are switches, labels and procedures which get a dpd and are handled (from the compiler's point of view) as name parameters.

The contents of the different possible dpd are given below using the following notation:

tha	thunk address
dp	driver pointer
ra	relative address
oa	object address
swa	switch address
la	label address
pp	prototype pointer
ca	constant address

Possible dpds are:

<type> constant:	ca
simple <type> variable:	ra,oa or pp,dp
<type> subscripted variable or <type> remote variable or <type> expression:	tha,dp
switch name or default	swa,dp
designational expression:	tha,dp
label name, value label:	la,dp
procedure name or default	pp,dp
array default	no dpd. A copy of array descriptor.

Any reference dpd must contain or give access to the identification (prototype pointer) of the class qualifying the actual parameter.

9. ASSIGNMENT

The assignment operation is defined for all types except ref. For integer, real, Boolean and character it has an obvious meaning. For text, the assignment operation is defined as a transfer of the character contents of one text to another.

10. DENOTES

The denotes operation is defined for ref and text only.

For ref, it has an obvious meaning. The operation must be checked for validity, possibly at runtime. (Cf. section on reference expressions.)

For text, the denotes operation establishes the left hand variable as a reference to the text designated by the right hand side.

Coding for the denotes operation is in-line.

11. RELATIONS

Result of all relations is Boolean.

For integer, real, character and text, the relations defined in ALGOL 60 may be used.

Comparison of ref expressions may be done using the operators == and =/=.

The above mentioned relations (except possibly for texts) are handled by in-line coding.

The operators is and in may be used to test the class membership of objects.

X is C has the value true if and only if the value X refers to an object belonging to the class C, while X in C has the value true if and only if the value of X refers to an object belonging to a class included in C. Both relations are false if X has the value none.

The is relation may be handled by in-line coding while the in relation is represented by a call on the check in (cin) subroutine.

```
Boolean procedure CIN(x,c);
   ref (object) x; ref (prototype) c;
         if x =/= none then
            begin if x.PP.plev ≥ c.plev then
                  CIN := x.PP.prefix [c.plev] == c
            end;
```

The compiler may give an errormessage if the static qualification of x and c disagree.

Comparison of two texts x and y are defined for all the six usual relational operators. The following rules are defined for the comparison:

1. Two empty texts are always equal.

2. Non-empty texts are equal if and only if they are of the same length and are instances of the same character sequence.

3. An empty text value is less than any non-empty text value.

4. If text values T and U are both non-empty then T is less than U if

   a) U is longer than T and has T as an initial subtext,

   b) the ith character of T ranks lower than the ith character of U where i ($i \geq 1$) is the lowest character position in which T differs from U.

5. The comparison is performed character by character from left to right.

6. The collating sequence is implementation defined.

12. FOR-STATEMENTS

All for-statements are handled by compiler generated coding.

13. REMOTE IDENTIFIERS

The syntax and semantics of remote references are defined in the Common Base definition. The runtime system is equipped with subroutines to handle calls of procedures referenced in this fashion.

14. CONNECTION

Cfr. Common Base definition, section 6.2.

The driver connecting an object will be called a connector driver.

The connector driver must contain a pointer to the driver of the block statically enclosing the connected object. cdrp is used to hold this pointer. This information is required when a procedure in the connected object is called.

The actual connection is performed by the subroutine connect. cdrp is found in DISPLAY[bl].MDP where bl is the level of the block containing the declaration of C, or if C is connected, i.e. local to a connected class D, the level is that of the block connecting D.

A connection has no prototype.

The end of a connection is through EBL (end block).

```
procedure CONNECT (p,bl);
   ref (object) p; integer bl;
   begin
      CD :- new driver (p,CD,none,CD,none,false,CD.level+1);
      CD.con := true;
      CD.cdrp :- DISPLAY[bl].MDP;
      DDISPLAY[CD.level] :- CD;
      DISPLAY[CD.level] :- p
   end CONNECT;
```

Note:

bl is required. Since p need not have a master driver, bl cannot be found in the prototype because the class containing p may be terminated.

15. BLOCK PREFIXING

The compiler generated coding for a prefixed block is:

1. A call on the begin prefixed block (BPB) subroutine.

2. In-line coding to evaluate the parameters and store them into the block instance.

3. A call on the end prefixed block parameters (EPBPAR) subroutine to indicate end of the parameter evaluation.

4. In-line coding for declarations within the block.

5. A call on the begin prefixed block return (BPBR) subroutine to indicate the end of the declarations within the block.

6. In-line coding for statements within the block.

7. A call on the end prefixed block (EPB) subroutine to indicate the end of the prefixed block.

A prefixed block is assumed to have the exit from the block indicated in the prototype.

The static link from this driver is to the block B statically enclosing the prefixed block P.

The reactivation point (pex,drex) for a prefixed block is initially none. It is set by a resume statement or a call on the store collapse.

The procedures in the formal description associated with a prefixed block are:

```
BPB       begin prefixed block
BPBR      begin prefixed block return
EPB       end prefixed block
EPBPAR    end prefixed block parameters

procedure BPB (x); ref (prototype) x;
   begin ref (driver) a,y,z;
      comment begin prefixed block;
      z :- new driver (new object(x),CD,none,none,none,true,
                                                      x.level);
      z.rp := z.ob := z.pb := true;
      z.obj.MDP :- z;
      a :- CD;
      while not a.rp do a :- a.drex;
      a.pex :- none;
      a.drex :- z;
      if x.nrp ≠ 0 then
         begin
            y :- new driver(CD.obj,CD.drp,none,z,none,false,
                                                   CD.level);
            y.con := CD.con;
            y.cdrp :- CD.cdrp;
            CD :- y;
            go to exit
         end else CD :- z;
         DISPLAY[x.level] :- CD.obj;
         DDISPLAY[x.level] :- CD;
      go to x.prefix[0].declare
end BPB;

procedure BPBR;
   go to CD.obj.PP.prefix[0]. statements;

procedure EPB;
   go to CD.obj.PP.prefix[CD.obj.PP.plev-1].inretur;
```

```
procedure EPBPAR;
   begin ref (driver) y;
      comment end prefixed block parameters;
      y :- CD;
      CD :- CD.drex;
      deletenotice (y);
      DISPLAY[CD.level] :- CD.obj;
      DDISPLAY[CD.level] :- CD;
      go to CD.obj.PP.prefix[0].declare
end EPBPAR;
```

16. SEQUENCING STATEMENTS

16.1 Detach

```
ref (object) procedure detach;
    begin ref (driver) x,y; ref (program) out;
       x :- CD;
       if x.rp then
       begin
          while not x.pb do
          begin x :- x.drp;
                while not x.rp do x :- x.drex;
          end;
          CD.pex :- exit;
          CD.drex :- CD;
          y :- x;
          x :- x.drp;
          while not x.rp do x :- x.drex;
          x.drex :- y;
          x.pex :- none;
          while y.pex == none do y :- y.drex;
          out :- y.pex;
          CD :- y.drex;
       end else
       begin
          out :- CD.pex;
          y :- CD.drex;
          CD.pex :- exit;
          CD.drex :- CD;
          CD.rp := true;
          detach :- CD.obj;
          restore (CD.acs);
          CD.acs :- none;
          CD :- y;
       end;
       update display;
       go to out
end detach;
```

If x is a prefixed block, the statement detach is a dummy statement. If x is an attached object, x is given (pexit, CD) as reactivation point and we return to the exit of x. The function value is in this case x.

If x is a detached object, the associated prefixed block is located. The detach statement is equivalent to a resume of this prefixed block.

16.2 Resume

```
procedure resume (x);  ref (object) x;
   begin
      ref (driver) y,z;
      Boolean b;
      if x =/= none then
      begin
         z :- x.MDP;
         if z == none then error ("resume",1);
         if not z.rp then error ("resume",2);
         y :- CD;
         while not y.rp then y :- y.drex;
         y.drex :- CD;
         y.pex :- exit;
         CD :- z;
         while CD.pex == none do CD :- CD.drex;
         exit :- CD.pex;
         CD :- CD.drex;
         y :- z;
   L:    b := y.pb;
         y :- y.drp;
         while not y.rp do y :- y.drex;
         if not b then go to L;
         y.drex :- z;
         update display;
         go to exit
   end
   else error ("resume",3)
end resume
```

Possible errors:

1. x is terminated.

2. x is not detached.

3. x is none.

Starting on current driver and following dynamic links (drex), locate the first detached object or prefixed block.

There must be at least one, since the program, by definition, is a prefixed block. (Cfr. Common Base definition, section 8).

The reactivation point is inserted in the driver of this object.

The driver of x becomes the current driver.

Starting on the driver of x, follow downward links (drex) until an object with pex not equal to none is found. This brings us down to the current detached object. This may step through more than one quasi-parallel system. Current driver may be changed during this prosess.

Note:
Pex of the master driver for an operating detached object must be none if there exists an inner quasi-parallel system for this object. Drex must point to the master driver of the active object of this quasi-parallel system.

The reactivation point pex,drex of the driver referenced by CD is fetched to (exit,CD). This brings us down to the innermost block of the operating object.

Locate the nearest detached object in the outer quasi-parallel system and modify drex in the master driver of this object to indicate that x is now the operating object of the inner quasi-parallel system.

Display is updated.

The object referenced by CD is then entered.

16.3 Call

The procedure "call" is not a part of the Common Base, but is a natural part of a SIMULA 67 Common Base implementation. "call" is formally a procedure with one unqualified parameter.

Let the actual parameter of a call on attach be a reference to a detached object y, which is a component of a quasi-parallel system s. Since y cannot be referenced from outside s, there must be a component x of s which is operating. The call on "call" has the following effects:

1. The OSC of s and LSC of x enters y at the current position of its LSC.

2. y becomes attached to x.

Informally a call on "call" means that the reactivation point of y is replaced by the exit to the call on attach and the control enters y at the position of its reactivation point (which may be in some inner block or even in some inner quasi-parallel system).

```
procedure call (x);
   ref (object) x;
   begin ref (driver) a,y,z; ref (program) next;
      if x =/= none then
         begin z :- x.MDP;
            if z == none then error ("call",1);
            if not z.rp then error ("call",2);
            y :- z;
            while y.pex == none do y :- y.drex;
            next :- y.pex;
            z.pex :- exit;
            a :- y.drex;
            z.drex :- CD;
            z.rp := false;
            CD :- a;
            update display;
            go to next
         end else error ("call",3);
      end call;
```

Note: This definition of call is tentative, since the problem is currently being studied by a Technical Committee under the SIMULA Standards Group.

17. LABELS AND SWITCHES

17.1 go to-statements

A program point (label) and a switch is uniquely defined by the following items:

Ordinary (label or switch):
Apparent block level and program address.

Virtual (label or switch):
Apparent block level and virtual index.

Actual parameter (label or switch):
Driver pointer and program address. (This is called a dynamic label).

Go to an ordinary label (except for local labels) and go to a formal label will be treated as equivalent since replacing the apparent block level BL by DDISPLAY (BL) for an ordinary label will give a case that may be handled by the formal go to procedure.

Thus only two routines in the runtime system will handle go to as a label:

GL	(go to label)
GVL	(go to virtual label)

The subroutine CONDDEL (which will determine whether a driver shall be deleted or not and perform the deletion) is used by GL.

For an actual parameter which is not an identifier, a thunk is created.

A designational expression is evaluated by TFL (take formal label).

For a switch, a switch calculation routine SWC is assumed to calculate a dynamic label (dp,pa) and enter the go to subroutine. This routine is not described here.

```
procedure conddel (x); ref (driver) x;
   begin
   if x.md then
      begin if not x.obj.PP.local classes then
         begin if x.dot then deletenotice (x.drp);
            deletenotice (x); x.obj.MDP :- none; end
      else begin x.drex :- x.drp; x.pex :- none; x.acs :- none;
           end
      end
      else if x.dot then begin deletenotice (x.drp);
            deletenotice (x) end
      else deletenotice (x);
   end conddel;

procedure GVL (bl,index); integer bl,index;
   begin ref (program) k;
      k :- DISPLAY (bl).PP.progaddr (index) qua program;
      if k == none then error ("GVL",1);
      GL (DISPLAY (bl),k)
   end GVL;

procedure GL(b,m); ref (object) b; ref (program) m;
   begin ref (driver) d; Boolean legal;
         while CD.obj =/= b or not CD.md do
         begin if CD.rp then
               begin d :- CD.drp;
                     if d == none then error ("GL",1);
                     legal := CD.pb;
               end else d:- CD.drex;
               conddel (CD);
               CD :- d;
         end;
         if not legal then error ("GL",2);
         go to m;
end GL;
```

18. RANDOM DRAWING AND DATA ANALYSIS

18.1 Random drawing

18.1.1 Generation of random numbers

The basic method for the generation of random numbers is described in the sequel. This is only one of several possible methods. The results of random number generators using this method at NCC have been satisfactory.

The technique described below obtains a basic drawing from the uniform distribution in the interval 0 to 1.

A basic drawing will replace the value of a specified integer variable, u, by a new value according to the following algorithm:

u(i+1) := remainder ((u(i)*5**(2*p+1))//2**n),

where u(i) is the i-th value of u.

It can be proved that, if u(o) is a positive odd integer, the same is true for all u(i), and the sequence u(o), u(1),u(2)..... is cyclic with the period 2**(n-2). (The last two bits of u remain constant, while the other n-2 take on all posible combinations). For the UNIVAC 1107/ 1108 implemention, we have n=35. P is chosen equal to 6.

The real numbers v(i)=u*2**(-n) are fractions in the range 0,1. The sequence v(1),v(2),..... is called a stream of pseudo-random numbers, and v(i), (i=1,2,....) is the result of the i-th basic drawing in the stream u. A stream is completely determined by the initial value u(o) of the corresponding integer variable. Nevertheless it is a fairly good approximation to a sequence of truly random drawings.

19. SYSTEM CLASSES

19.0 Permissible limitations

Either of the following limitations may be imposed on the use of system classes:

a. A system class may be used as prefix on one single block level only.

b. There will be as many incarnated copies of a system class as the number of different block levels where the system class has been used as prefix.

Limitation a implies that the dynamic validity of a parameter which is a system class may be completely checked at compile time for ordinary procedures.

The following restrictions may be imposed on the use of system classes as prefixes (p = permitted, n = not permitted):

	block	class	
SIMULATION	p	p	
process	n	p	
MAIN PROGRAM	n	n	not user accessible
SIMSET	p	p	
linkage	n	p	
link	n	p	
head	n	p	
BASICIO	p	p	
FILE	n	n	not user accessible
infile	(p)	(p)	
outfile	(p)	(p)	
directfile	(p)	(p)	
printfile	(p)	(p)	

19.1 Implementation of the class "SIMSET

The class SIMSET and its associated procedures are assumed to be coded in assembly language.

1. For computers with a word size permitting two references in one word, the links used for set inclusion, SUC and PRED, would be conveniently put into the first word following the (MDP,PP) word in the object. The store collapse must know about this special form of packing, as both links must be followed and updated.

2. The validity of the parameter to a procedure declared within the class SIMSET must be checked by the compiler.

3. The procedures in the class SIMSET must be called by the user by remote referencing. The ref expression pointing to the linkage, link or head object must be considered as an implicit parameter to the runtime subroutines.

   A set should be implemented as an off-line item.

   The head must be an ordinary object with an MDP and a PP. The links SUC and PRED may be packed into one word provided that the compiler, the procedure parameter mechanism and the store collapse take this into account. An actual prototype need not be provided for the class head if the runtime system and the compiler is suitably written.

   Prototypes for the classes linkage, link and head and for the procedures suc, pred, out, follow, precede, into, first, last, empty, cardinal and clear are not required if the compiler and the runtime system reflect this fact.

19.2 Implementation of the class "SIMULATION"

The main differences between various implementations of the class SIMULATION would probably be found in the organisation of the sequencing set (SQS).

Three logical alternative approaches are:

1. Linear list (ordered on time)
2. Unbalanced tree
3. Balanced tree

The reader should refer to relevant literature on list processing for details.

20. SEPARATE COMPILATION *)

According to the Common Base definition, separate compilation of procedures and classes may be introduced in a Common Base implementation.

When a program using a separately compiled class is compiled, all identifiers local to the class must be known by the compiler. This may be solved by making a separately compiled class consisting of two parts: a name table (with type indication and relative address) and the object code.

It is permitted to restrict the use of a separately, compiled class (and also a system class) in the program by enforcing the rule that an external declaration of a class C may only appear on one single block level within the program.

This does not prohibit the use of an external declaration for the same class in two blocks with disjoint scope if these are on the same static level.

21. Store_collapse

21.0 Main_flow_of_store_collapse

The store collapse consists of six phases:

1. Start on CD, locate all referenceable objects.
   MDP of referenceable objects will at the end of this phase point to itself.

2. Scan POOL 1 sequentially.
   Compute addresses of objects in POOL 1 after move (in phase 5) and store in MDP.
   Chain first block of each available area to the next used block to speed up later phases.

*) The problems of Separate Compilation are currently being studied by the SIMULA Standards Group.

3. Move POOL 2 by minimal moves.
   Update pointers from POOL 2 to POOL 1.

4. Scan POOL 1 sequentially.
   Update all pointers in POOL 1.

5. Move POOL 1. Zero MDP.

6. Scan POOL 2 sequentially.
   Update POOL 2 pointers to POOL 2.
   Insert new MDPs.

21.1 Phase 1

During phase 1, an object chain is used to save objects in which pointers have not yet been followed. This object chain is, first-in last-out. Each object, using MDP, points to its predecessor. The last object is pointed to by a ref variable object declared in the runtime system.

Whenever a reference to an object is found, the procedure chain is used. This procedure will put the object on object chain, provided:

1. It is not in the object chain.

2. It has not been fully processed before.

This may be determined by MDP of the process, which during phase 1 may be in one of three states, as follows:

1. MDP points to the object itself:

   The master driver (if any) of this object has been marked as referenced and all pointers in the object have been followed.

2. MDP points to another object or MDP = none and objch or another object points to this object:

   The master driver (if any) of this object has been processed, but the pointers in the object itself have not yet been followed.

3. MDP points to a driver or MDP = none and neither objch nor another object points to this object:

   It has not yet been found that this object is referenceable.

When all pointers in an object have been processed, the next object is removed from the object chain, its MDP set to point to itself and then all pointers local to this object are processed.

21.1.1 Asgn

Asgn is a utility procedure used by the procedures first-pointer and nextpointer to set up the non-local variables kin, typ, pa and va declared local to storecollapse, and to increment the pointer index pnr.

21.1.2 Firstpointer, nextpointer

Firstpointer and nextpointer are procedures which locate pointers in an object.

A call on the procedure firstpointer has two parameters:

1. A ref to the object where pointers should be located.

2. A label to which control will be transferred when no more pointers are found in the object.

   If there are no pointers, the procedure will at once go to the label parameter.

If there is at least one pointer, firstpointer will return with the address of the first pointer as function value. It will also note the parameters for future use in the variables pobj and exit declared local to storecollapse.

Each successive call on nextpointer will then return as function value the address of the next pointer.

The index of the current pointer is recorded in the integer pnr local to storecollapse.

When no more pointers are left in the object, nextpointer will go to the label given to firstpointer as a parameter (now found in exit local to storecollapse).

21.1.3 Chain

Chain is used to insert objects on the object chain if they are not already on this chain or have been previously processed.

21.1.4 Map

The procedure map will put objects on the object chain and mark their drivers as referenced. In some cases, drivers will be inserted on the driver chain.

Map may be used to map an entire dynamic structure, starting in the innermost quasi-parallel system of the structure, or to map a static structure, such as the drivers and objects for terminated objects.

A step by step description of map is given below:

Step 1:

If the driver has been processed before, exit.

Step 2:

If drex of this driver is none, we are going to process a static structure. In this case continue from step 13.

Step 3:

Locate the nearest detached object or prefixed block.

Step 4:

Locate operating object of innermost quasi-parallel system.

Step 5:

Mark this driver as referenced.

Step 6:

If this is a connector driver and the extra static link (cdrp) is none, this driver is included in the driver chain for later processing of cdrp. Continue on step 9.

Step 7:

If dot is true, i.e. the driver has been created when a procedure was called by remote referencing, the driver pointed to by the static link (drp) is put on the driver chain for later processing. Continue on step 9.

Step 8:

Put the accumulator stack on the object chain for later processing.

Step 9:

Put the object on the object chain for later processing.

Step 10:

If this was not the driver of a detached object or a prefixed block, follow the dynamic link (drex) once and proceed from step 5.

Step 11:

If this was the driver of a prefixed block, follow the static link once and proceed from step 5. (A prefixed block has no dynamic link.)

Step 12:

Follow static link once and proceed from step 1. (A detached object has no dynamic link).

Step 13:

Mark this driver as referenced and put it on the driver chain for later processing. Follow static links once and proceed from step 1.

21.1.5 Maptree

The procedure maptree is used to mark all eventnotices and drivers of processes they refer to as referenced, and put processes on object chain when necessary.

The algorithm supposes that the sequencing set is organized as a binary tree (cfr. section 20.3). Furthermore, it assumes that a left branch which is empty implies that the right branch for this node is also empty.

There exist several possible algorithms. The method chosen here requires a fixed amount of working storage to handle sequencing sets of any size.

The algorithm starts from the uppermost node of the tree. This is pointed to by the ref (EVENTNOTICE) variable high in class SIMULATION. At this point, all EVENTNOTICES in this sequencing set have referenced false:

1. Proceed to the right until an EVENTNOTICE is found which has either no right brancn or is previously marked as referenced.

2. Proceed to the left until an EVENTNOTICE is found which has no left branch.

3. Put the process referenced by this EVENTNOTICE on the object chain (using chain), and mark this EVENTNOTICE as referenced.

4. Follow the backward link from this EVENTNOTICE.

5. Put the process referenced by this EVENTNOTICE on the object chain (using chain) and mark this EVENTNOTICE as referenced. All EVENTNOTICES to the right of this event-notice must have been marked previously.

6. Follow the backward link from this EVENTNOTICE, if there is none, the whole sequencing set has been marked, if not proceed from 1.

It is possible to use a similar algorithm to the one described here starting on low.

21.1.6 Chain 2

Chain 2 is used to insert drivers on the driver chain drchn for later processing.

21.2 Phase 2

Phase 2 is a sequential scan of POOL 1.

During this scan:

1. Address after move of each referenceable object is computed and stored in MDP of the object for fast lookup when updating pointers in phase 3 and phase 4.

2. Adjacent available blocks are combined into one block by making PP none and MDP a pointer to the next referenceable block.

21.3 Phase 3

Phase 3 is a minimal move of POOL 2 and update of all pointers from POOL 2 to POOL 1.

When the contents of a notice is moved, an indication where it has been moved is put into the old notice for updating purposes.

The algorithm is as follows:

1. Start from the bottom of POOL 2.

2. Locate a notice which is not referenced.
Update POOL 1 pointers in all referenced notices found during this scan. If we have come to the top of POOL 2, we are finished.

3. Start on top of POOL 2 and decrement the size of POOL 2 until a referenced notice is found.
If we during this reach the notice found in 2, we are finished.

4. Move contents of notice found in 2 to available space found in 3.
Update pointers to POOL 1.
Record where the notice has been moved.

5. Decrement POOL2 size by one notice and increment the other pointer by one notice.
If the pointer is not outside POOL 2, proceed from 2.

6. Perform some housekeeping to figure out the two separate cases when the pointers meet.

At the end of phase 3, it is poosible to compute the amount of available storage and check that this covers the amount required. This is not done in the algorithm shown here. The test has been postponed to after phase 6.

21.3.1 Move

The procedure move has three arguments:

1. Address to move from, x.

2. Address to move to, y.

3. Length of area to be moved, i.

The continuous area of storage (x,x+i-1) is moved to (y,y+i-1).

21.3.2 Updl

updl will update pointers from POOL 2 to POOL 1:

1. Pointer to the object, obj.

2. If notice is a driver, pointer to the accumulator stack, acs.

21.4 Phase_4

Phase 4 is the updating phase for POOL 1. It is a sequential scan of POOL 1, using the shortcuts established by phase 2 combining adjacent available areas.

All pointers are updated. For POOL 1 pointers, the new value is found in MDP of the referenced object.

For POOL 2 pointers, the new value is recorded in the old notice if it has been moved. Only POOL 2 pointers pointing to the area between POOL 2 top when the store-collapse was entered and POOL 2 top after phase 3 should be updated; as only these have a new address after move.

Note:

Since the outermost block of a program must be resident from the point of view of the runtime system, the area starting at POOL 1 first is always referenceable.

21.5 Phase 5

Phase 5 is a move of objects of POOL 1.

POOL 1 is scanned sequentially using mfa.

mta contains the address where the next referenceable object should be moved.

MDP is set to none.

The actual move is performed only if mfa is not equal to mta.

A continuous referenceable area is moved by one call on the move procedure (cfr. section 23.4.1).

21.6 Phase 6

Phase 6 is a sequential scan of POOL 2 from starting at POOL 2 bottom and some housekeeping to prepare for exit from the store collapse.

Pointers to POOL 2 in the notices are updated using upd2.

CD is updated.

The available storage list is cleared.

If the required storage exceeds available storage, an error condition is raised.

21.6.1 Upd2

upd2 will update a pointer to POOL 2.

If the value of the pointer is less than the POOL2TOP determined in phase 3 (i.e. outside POOL 2), the new pointer value will replace the old value.

22. FORMAL DESCRIPTION OF SIMULA COMMON BASE RUNTIME SYSTEM

```
begin ref (driver) CD;
      ref (notice) nothead,POOL2TOP,POOL2BOTTOM;
      ref (object) POOL1FIRST,POOL1LAST;
      ref (driver) array DDISPLAY[1 : maxlevel];
      ref (object) array DISPLAY [1 : maxlevel];

      comment the integer maxlevel is assumed to be
      defined when the RTS is written. The variables
      and arrays above are "non-local" to the runtime
      routines.

      comment "driver" and "eventnotice" are sub-
      classes of a class called "notice".

      class notice (obj); ref (object) obj;
            begin Boolean referenced; ref (notice) notc;
            end notice;

      notice class eventnotice (time); real time;
             begin ref (eventnotice) BL,RL,LL;
             end eventnotice;

      notice class driver (drp,pex,drex,acs,md,level);
             ref (program) pex; ref (driver) drex,drp;
             ref (object) acs;
             Boolean md; integer level;
             begin Boolean con,rp,pb,dot,ob;
                   ref (eventnotice) evp;
                   ref (driver) cdrp,drch;
             end driver;

      comment all block instances are a subclass of
      "object". This includes arrays and stored accumu-
      lator stacks which have special values of PP.
```

```
class object (PP); ref (prototype) PP;
      begin ref (driver) MDP; end object;

comment  a prototype is a description of a family of
block instances.  One prototype is generated by the
compiler for each procedure, class, subblock and pre-
fixed block in the program;

class prototype (lg,nvirt,nrp,plev,nrl,level);
      integer lg,nvirt,nrp,nrl,level,plev;
      begin ref (program) statements,inretur,endblk,declare;
            ref (prototype) array prefix[0:plev+1];
            integer array relad,kind,type[nvirt+1 :
                              nvirt+nrp+nrl];
            Boolean array valu[nvirt+1 : nvirt+nrp];
            ref array progaddr[1 : nvirt];
            Boolean pb,ob,local classes;
            integer vtype;
      end prototype;

comment  update display is used to update DISPLAY and
DDISPLAY to reflect the textual situation defined by CD;

procedure update display;
   begin integer i;
      DDISPLAY [CD.level] :- CD;
      for i := CD.level-1 step -1 until 1 do
          DDISPLAY [i] :- DDISPLAY [i+1].drp;
      for i := 1 step 1 until CD.level do
         DISPLAY [i] :- DDISPLAY [i].obj;
   end update display;

comment  deletenotice is used to put a driver or event-
notice in the list  of available notices;

procedure deletenotice (x); ref (notice) x;
   begin x.notc :- nothead; nothead :- x; end;
```

```
comment  SUBBLOCKS
         BB   - begin subblock
         EBL  - end subblock;

procedure BB(p); ref (prototype) p;
   begin
      CD :- new driver (new object (p),CD,none,CD,none,true,
      p.level);
      CD.obj.MDP :- CD;
      DISPLAY [p.level] :- CD.obj;
      DDISPLAY [p.level] :- CD;
      go to p.declare
   end BB;

procedure EBL;
   begin ref (driver) x;
      x :- CD.drp;
      deletenotice (CD);
      CD :- x;
    end EBL;
```

```
comment  PROCEDURES;

comment  procedure end;

universal procedure EPR (val);
   begin ref (driver) x; ref (program) out;
      x :- CD.drex;
      out :- CD.pex;
      restore (CD.acs);
      if CD.dot then deletenotice (CD.drp);
      deletenotice (CD);
      CD :- x;
      EPR := val;
      update display;
      go to out;
   end EPR;

comment  utility routine to enter declaration code
or in-line coding for parameter transmission;

procedure ENTPROC;
   begin ref (driver) y;
      CD.obj.MDP :- CD;
      if CD.obj.PP.nrp = 0 then
         begin update display;
            go to CD.obj.PP.declare end;
      y :- CD.drex;
      CD :- new driver (y.obj,y.drp,none,CD,none,
                  false,y.level);
      CD.con := y.con;
      CD.cdrp :- y.cdrp;
      DDISPLAY[y.level] :- CD;
      go to CD.drex.pex;
   end ENTPROC;
```

comment utility routine to enter declaration code of procedure or class after parameter transmission.

Note: It is assumed that prefix [0] of the prototype for a procedure points to the procedure itself;

```
procedure ENTER;
   begin ref (driver) y;
      y :- CD;
      CD :- CD.drex;
      CD.pex :- exit;
      deletenotice (y);
      update display;
      go to CD.obj.PP.prefix [0].declare;
   end ENTER;
```

```
comment  NON-FORMAL, NON-VIRTUAL PROCEDURES;

comment  call on normal procedure (local or non-local);

procedure CPR (p,acs);
   ref (prototype) p; ref (object) acs;
   begin
      CD :- new driver (new object (p), DDISPLAY [p.level-1],
          exit,CD,acs,true,p.level);
      ENTPROC;
   end CPR;

comment  call on connected procedure (cfr. CONNECTION);

procedure CCP(p,c,acs);
   ref (prototype) p; ref(object) acs;
   ref (driver) c;
   begin ref (driver) x;
      x :- new driver (c.obj,c.cdrp,none,none,none,false,
          p.level-1);
      x.con := true;
      CD :- new driver (new object(p),x,exit,CD,acs,
          true,p.level);
      CD.dot := true;
      ENTPROC
   end CCP;
```

```
comment  call on remote procedure;

procedure CDP (p,c,slc,acs);
   ref (prototype) p; ref (object) c,acs;
   ref (driver) slc;
   begin ref (driver) x;
      x :- new driver (c,slc,none,none,none,false,p.level-1);
      x.con := true;
      CD :- new driver (new object(p),x,exit,CD,acs,true,
          p.level);
      CD.dot := true;
      ENTPROC;
   end CDP;
```

```
comment  VIRTUAL PROCEDURES;

comment  ENTVIRT utility procedure for virtuals corresponding
to ENTPROC.  The routine must
1.  check actual vs. formal number of parameters
2.  check type/kind of parameters
3.  perform calls by value;

procedure entvirt (p,dr,dot,acs);
   ref (prototype) p; ref (driver) dr;
   ref (object) acs; Boolean dot;
   begin
      CD  :- new driver (new object(p),dr,exit,CD,acs,true,
          p.level);
      CD.dot := dot;
      CD.obj.MDP :- CD;
      store descriptors for name parameters;
      store value parameters;
      update display;
      go to p.declare
   end entvirt;

comment  call on normal virtual procedure;

procedure CVP (cl,index,acs);
   integer index;
   ref (object) acs; ref (driver) cl;
   begin ref (prototype) p,q;
      p :- cl.obj.PP;
      q :- p.progaddr(index) qua prototype;
      if q == none then error ("cvp",1);
      entvirt (q,cl,false,acs)
   end CDVP;
```

```
comment  call on connected virtual procedure;

procedure CCVP (c,index,acs);
   integer index;
   ref (object) acs; ref (driver) c;
   begin ref (prototype) p,q;
      p :- c.obj.PP;
      q :- p.progaddr(index) qua prototype;
      if q == none then error ("CCVP",1);
      c :- new driver (c.obj,c.cdrp,none,none,none,
         false,p.level);
      c.con := true;
      entvirt (q,c,true,acs)
   end CCVP;

procedure CDVP(c,index,slc,acs);
    integer index;
    ref (object) c,acs; ref (driver) slc;
    begin ref (prototype) p;
       p :- c.PP.progaddr (index) qua prototype;
       if p == none then error ("CDVP",1);
       slc :- new driver (c,slc,none,none,none,
           false,p.level-1);
       slc.con := true;
       entvirt (p,slc,true,acs)
    end CDVP;
```

```
comment  CLASSES;

comment  GENERATING REFERENCE;

comment  begin class, enters declaration code if no
parameters, otherwise continue with in-line parameter
evaluation and return is later through ENTER.

procedure BC (x,slx,acs);
   ref (prototype) x; ref (object) slx;
   ref (object) acs;
   begin ref (driver) y; ref (prototype) q;
      y :- new driver (new object(x),slx.MDP,exit,CD,acs,
          true,x.level);
      y.ob := true; y.obj.MDP :- y;
      if x.nrp ≠ 0 then
         begin y :- new driver (CD.obj,CD.drp,none,y,none,
                  false,CD.level);
               y.con := CD.con;
               y.cdrp :- CD.cdrp;
               CD :- y;
               DDISPLAY[CD.level] :- CD;
               go to exit
         end else CD :- y;
      update display;
      go to CD.obj.PP.prefix[0].declare
   end BC;

comment  begin class return, signifies the end of declaration
code in a class.

procedure BCR(q); integer q;
   begin ref (prototype) x,y;
      x :- CD.obj.PP;
      y :- x.prefix[q+1];
      if  y =/= none then go to y.declare;
      go to x.prefix[0].statements
   end BCR;
```

```
comment  procedure corresponding to the statement inner;

procedure CINNER (lev); integer lev;
   begin ref (prototype) p;
      p :- CD.obj.PP.prefix[lev+1];
      if p =/= none then go to p.statements
   end CINNER;

comment  end class body;

ref (object) procedure ECB(p); ref (prototype) p;
   begin ref (program) out; ref (driver) x;
      procedure delete;
         begin x.rp := false;
            if x.obj.PP.local classes then
            begin
                  x.drex :- x.drp;
                  x.pex :- none;
                  x.acs :- none;
            end
            else begin x.obj.MDP :- none;
                       deletenotice(x)
                 end
         end delete;
             if p.plev ≠ 0 then go to p.prefix [p.plev-1]
                                                 .inretur;
             x :- CD;
             if CD.rp and not CD.pb then
             begin CD :- CD.drp; delete; go to L2;
             L1:   CD :- CD.drp;
             L2:   while not CD.rp do CD :- CD.drex;
                   if not CD.pb then go to L1;
                   x :- CD.drp;
                   while not x.rp do x :- x.drex;
                   x.drex :- CD;
             L3:   if CD.pex =/= none then go to L4;
                   CD :- CD.drex;
                   go to L3;
             L4:   out :- CD.pex;
             end else
```

```
    begin
        if x.pb then
        begin CD :- x.drp;
              out :- x.obj.PP.endblk;
              deletenotice (x);
              go to ud
        end else
        begin out :- x.pex;
              CD :- CD.drex;
              ECB :- x.obj;
              restore (x.acs);
              delete
        end
    end;
    update display;
    ud : go to out
end ECB;
```

```
comment  PREFIXED BLOCK;

comment  begin prefixed block, enter declarations or
evaluate parameters.  If latter case, return through EPBPAR;

procedure BPB (x); ref (prototype) x;
   begin ref (driver) a,y,z;
      z :- new driver (new object(x),CD,none,none,none,true,
          x.level)
      z.rp := z.ob := z.pb := true;
      z.obj.MDP :- z;
      a :- CD;
      while not a.rp do a :- a.drex;
      a.pex :- none;
      a.drex :- z;
      if x.nrp ≠ 0 then
         begin
            y :- new driver (CD.obj,CD.drp,none,z,none,false,
                CD.level);
            y.con := CD.con;
            y.cdrp :- CD.cdrp;
            CD :- y;
            go to exit;
         end else CD :- z;
      DISPLAY [x.level]:- CD.obj;
      DDISPLAY [x.level]:- CD;
      go to x.prefix[0].declare
    end BPB;

comment  end prefixed block parameters;

procedure  EPBPAR;
   begin ref (driver) y;
       y :- CD;
       CD :- CD.drex;
       deletenotice(y);
       DISPLAY [CD.level] :- CD.obj;
       DDISPLAY [CD.level] :- CD;
       go to CD.obj.PP.prefix[0].declare
   end EPBPAR;
```

```
comment  begin prefixed block return, end declaration
code in prefixed block;

procedure  BPBR;
   go to CD.obj.PP.prefix[0].statements;

comment  end prefixed block;

procedure EPB;
   go to CD.obj.PP.prefix[CD.obj.PP.plev-1].inretur;
```

```
comment  QUASI-PARALLEL SEQUENCING;

comment  resume;

procedure resume (x);
   ref (object) x;
   begin
      ref (driver) y,z;
      Boolean b;
      if x =/= none then
      begin
         z :- x.MDP;
         if z == none then error ("resume",1);
         if not z.rp then error ("resume",2);
         y :- CD;
         while not y.rp do y :- y.drex;
         y.drex :- CD;
         y.pex :- exit;
         CD :- z;
         while CD.pex == none do CD :- CD.drex;
         exit :- CD.pex;
         CD :- CD.drex;
         y :- z;
```

```
L:      b := y.pb;
        y :- y.drp;
        while not y.rp do y :- y.drex;
        if not b then go to L;
        y.drex :- z;
        update display;
        go to exit
    end
    else error ("resume",3);
end resume;
```

```
comment detach;

ref (object) procedure detach;
   begin ref (driver) x,y; ref (program) out;
      x :- CD;
      if x.rp then
      begin while not x.pb do
                begin x :- x.drp;
                      while not x.rp do x :- x.drex;
                end;
                CD.pex :- exit;
                CD.drex :- CD;
                y :- x;
                x :- x.drp;
                while not x.rp do x :- x.drex;
                x.drex :- y;
                x.pex :- none;
                while y.pex == none do y :- y.drex;
                out :- y.pex;
                CD :- y.drex;
      end else
      begin
          out :- CD.pex;
          y :- CD.drex;
          CD.pex :- exit;
          CD.drex :- CD;
          CD.rp := true;
          detach :- CD.obj;
          restore (CD.acs);
          CD.acs :- none;
          CD :- y;
       end;
       update display;
       go to out
    end detach;
```

```
comment  call (not part of Common Base);

procedure call (x);
   ref (object) x;
      begin ref (driver) a,y,z; ref (program) next;
         if x =/= none then
            begin z :- x.MDP;
               if z == none then error ("call",1);
               if not z.rp then error ("call",2);
               y :- z;
               while y.pex == none do y :- y.drex;
               next :- y.pex;
               z.pex :- exit;
               a :- y.drex;
               z.drex :- CD;
               z.rp := false;
               CD :- a;
               update display;
               go to next
            end else error ("call",3)
      end call;
```

```
comment  PARAMETER EVALUATION (THUNK);

comment  call (enter) thunk;

procedure CTH(tha,thu,acs);
   ref (program) tha;
   ref (object) acs; ref (driver) thu;
   begin
      CD :- new driver (thu.obj,thu.drp,exit,CD,acs,false,
          thu.level);
      CD.con := thu.con;
      CD.cdrp :- thu.cdrp;
      update display;
      go to tha
   end CTH;

comment  end thunk;

ref procedure ETH(addr); ref addr;
   ETH :- epr(addr);
```

```
comment  CONNECTION;

procedure CONNECT (p,bl);
   ref (object) p; integer bl;
   begin
      CD :- new driver (p,CD,none,CD,none,false,CD.level+1);
      CD.con := true;
      CD.cdrp :- DISPLAY[bl].MDP;
      DDISPLAY [CD.level] :- CD;
      DISPLAY [CD.level] :- p
   end CONNECT;

comment   RELATIONS;

comment  check in;

Boolean procedure CIN(x,c);
   ref (object) x; ref (prototype) c;
   if x =/= none then
      begin if x.PP.plev ≥ c.plev then
               CIN := x.PP.prefix[c.plev] == c
      end;
```

```
comment  INSTANTANEOUS QUALIFICATION (QUA);

ref procedure CIQ (x,c);
   ref (object) x; ref (prototype) c;
      begin ref (prototype) d;
         d :- x.PP;
         if d.plev < c.plev then error ("CIQ",1);
         if d.prefix [c.plev] =/= c then error ("CIQ",2);
         CIQ  :- x
      end CIQ;
```

```
comment  GO TO STATEMENTS;

comment  utility procedure conddel;

procedure conddel (x); ref (driver) x;
   begin
      if x.md then
         begin if not x.obj.PP.local classes then
            begin if x.dot then deletenotice (x.drp);
            deletenotice (x); x.obj.MDP :- none; end
            else begin x.drex :- x.drp; x.pex :- none;
            x.acs :- none end
         end
         else if x.dot then
         begin deletenotice (x.drp);
               deletenotice (x)
         end else deletenotice (x);
   end conddel;

comment  go to normal label;

procedure GL (b,m); ref (object) b; ref (program) m;
   begin ref (driver) d; Boolean legal;
         while CD.obj =/= b or not CD.md do
         begin if CD.rp then
               begin d :- CD.drp;
                     if d == none then error ("GL",1);
                     legal := CD.pb;
               end else d :- CD.drex;
               conddel (CD);
               CD :- d;
         end;
         if not legal then error ("GL",2);
         go to m;
   end GL;
```

```
comment  go to virtual label;

procedure GVL (bl,index); integer bl,index;
   begin ref (program) k;
      k :- DISPLAY [bl].PP.progaddr (index) qua program;
      if k = none then error ("GVL",1);
      GL (DISPLAY [bl],k);
   end GVL;
```

```
procedure storecollapse(req);integer req;
   begin
      integer kin,typ,pnr;
      Boolean pa,va;
      ref (object) objch,x,y,na,mta,mfa,pobj;
      ref (driver) drchn,drv;
      ref (program) pexit;
      ref (notice) d,lm;
      ref z;

procedure chain (x); ref (object) x;
  if x =/= none then
      begin
         if (x.MDP==none or x.MDP is driver) then
            begin x.MDP :- objch; objch :- x end
      end chain;
```

```
procedure chain2(y); ref (driver) y;
   begin y.referenced := true;
   y.drch :- drchn; drchn :- y end chain2;

procedure map (x); ref (driver) x;
   if x =/= none then
      begin
      L0:  if not x.referenced then
           begin
              if x.drex == none then  go to L6;
              while not x.rp do x :- x.drex;
              while x.pex == none do x :- x.drex;
      L3:  x :- x.drex;
      L4:  x.referenced := true;
           if x.con then
           begin
           if x.cdrp =/= none then begin chain2(x);
           go to L5; end
           end else if x.dot then chain2 (x.drp);
           chain (x.acs);
           chain (x.obj);
      L5:  if not x.rp then go to L3;
           if x.pb then begin x :- x.drp;
           go to L4 end;
           x :- x.drp; go to L0;
      L6:  chain (x.obj);
           x :- x.drp;
           go to L0
      end
  end map;

procedure upd1(n); ref (notice) n;
   begin procedure upd(z); name z; ref (object) z;
      if z =/= none then z :- z.mdp;
      upd (n.obj);
      inspect n when driver do upd (acs);
   end upd1;
```

```
procedure upd2(n); name n; ref (notice) n;
   if n =/= none then
      begin
         if n < POOL2TOP then n :- n.notc
      end upd2;

procedure maptree(e); ref (eventnotice) e;
   begin if e == none then go to ret;
   L:   map(e.obj.MDP);
        e.referenced := true;
        if e.RL =/= none then
           begin e :- e.RL; go to L end;
        if e.LL =/= none then
           begin e :- e.LL; go to L end;

   M:   e :- e.BL;
        if e =/= none then
           begin if e.LL.referenced
                 then go to M
                 else begin e :- e.LL; go to L end;
           end;
   ret: end maptree;

ref (ref) procedure asgn(z,i); ref (prototype) z; integer i;
   inspect z when prototype do
      begin kin := kind[i]; typ := type[i];
      på := par[i]; va := valu[i];
               asgn :- pobj+relad[i]; i := i+1 end asgn;

ref (ref) procedure firstpointer(x,L); value L; ref (object) x;
label L;
   begin ref (prototype) z;
      z :- x.PP;
      if z.nrpointers = 0 then go to L;
      pnr := 1; pobj :- x; pexit :- L;
      firstpointer :- asgn (z,pnr)
   end firstpointer;
```

```
      ref (ref) procedure nextpointer;
         begin ref (prototype) z;
            z :- pobj.PP;
            if pnr > z.nrpointers then go to pexit;
            nextpointer :- asgn (z,pnr)
         end nextpointer;

      procedure move (x,y,i); ref x,y; integer i;;

      comment  pass 1;

      x :- CD;
      while not x.rp do x :- x.drex;
      x.pex :- exit;
      x.drex :- CD;
      POOL1LAST.val :- none;
      map (CD);

L:
      if objch =/= none then
         begin x :- objch; objch :- x.MDP; x.MDP :- x;
               z :- firstpointer (x,L).val;
         r:    inspect z when object do
                  begin if MDP == none then chain (z)
                     else if MDP is driver then map(MDP)
                  end
               when driver do map (this driver)
               when eventnotice do maptree (this eventnotice);
               z :- nextpointer.val;
               go to r
         end;
rep2:
      if drchn =/= none then
         begin drv :- drchn; drchn :- drv.drch;
            inspect drv when driver do
            inspect obj when object do
            begin if MDP == none then chain (obj)
            else if MDP is driver then map (MDP);
         end;
```

```
            if cdrp =/= none then map (cdrp); drv.referenced := true
         end;
         go to rep2
      end else if objch =/= none then go to L;

      comment  pass 2;

      y :- none;
      na :- x :- POOL1FIRST;
nextblock:
      if x.MDP =/= x then
          begin if y == none then
              begin y :- x; x.PP :- none end
          end
      else begin x.MDP :- na;
                 na :- na+x.PP.lg;
                 if y =/= none then begin y.MDP :- x; y :- none end
          end;
      x :- x+x.PP.lg;
      if x.PP =/= none then go to nextblock;
      if y =/= none then y.MDP :- none else x.MDP :- none;

      comment pass  3;

      d :- POOL2BOTTOM;
used:
      if d.referenced then
         begin updl(d);
            d.referenced := false;
            if d == POOL2TOP then go to pass3end;
            d :- d-drlg;
            go to used
         end;
```

```
unused:
     if not POOL2TOP.referenced then
        begin
             POOL2TOP :- POOL2TOP+drlg;
             if POOL2TOP == d then begin POOL2TOP :- POOL2TOP-drlg;
             go to pass3end end else go to
             unused
         end;

     move (POOL2TOP,d,drlg);
     POOL2TOP.notc :- d;
     d.referenced := false;
     updl(d);
     lm :- d;
     POOL2TOP :- POOL2TOP+drlg;
     d :- d-drlg;
     if d > POOL2TOP then go to used else
        if not (d == POOL2TOP and d.referenced) then POOL2TOP :-
        lm else
             begin updl(d); d.referenced := false end;
pass3end:

     comment  pass 4;

     x :- POOL1FIRST;
upnext:
     y :- firstpointer (x,m);
update:
     z :- y.val;
     inspect z when notice do
        if z  < POOL2TOP then y.val :- notc
     when object do y.val :- MDP;
     y :- nextpointer;
     go to update;

m:
     x :- x+x.PP.lg;
     if x.PP  =/= none then go to upnext;
     x :- x.MDP;
     if x =/= none then go to upnext;
```

```
        comment  pass 5;

        mta :- x :- POOL1FIRST;
        mfa :- x;
        go to entpass5;
   clear:
        x.MDP :- none;
        x :- x+x.PP.lg;
entpass5: if x.PP =/= none then go to clear;
        if mfa =/= mta then move (mfa,mta,x-mfa);
        mta :- mta+x-mfa;
        x :- x.MDP;
        mfa :- x;
        if x =/= none then go to clear:
        POOL1LAST :- mta;

        comment pass 6;

        d :- POOL2BOTTOM
    nextnotice:
        inspect d when eventnotice do
           begin upd2(BL); upd2(LL); upd2(RL) end
        when driver do
           begin
               upd2(drex); upd2(drp); upd2(evp);
               upd2(cdrp);
               if md then obj.MDP :- this driver
           end;
        if d =/= POOL2TOP then begin d :- d-drlg; go to nextnotice
        end;
        upd2(CD);
        update display;
        nothead :- none;
        if req > POOL2TOP-POOL1LAST then
                error ("storecollapse",1);
      end storecollapse
    end runtime system
```