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 <u>ref</u> (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 <u>Compiler - pass 3</u>

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, ..., maxblev 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 stacks 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 addres within a data record. For a declared label th run time address is an instruction address (pc For any quantity matching a virtual specification the attribute addr is a relative addres within a prototype. For a label or procedure whose declaration has not been processed, addr 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 <u>inner</u> 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 match
                 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" 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 <u>true</u> 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.