The Compiler of ISSP ALGOL

KENZO INOUE*, HIDECHIKA TAKAHASHI* AND KIMIKO SHIMIZU*

1. *Introduction*

By the ALGOL 60 language [1] one can elegantly describe any algorithm for numerical analysis, but implementing its effective compiler will be led to some difficulties. For example, by a computer not possessing the function of stacking memory, the recursive call of procedures can be implemented only at the waste of the execution time of object programs.

In implementing a compiler, the production of fast object programs is important from the practical point of view, so the syntax of the language has usually restrictions on some points depending on the scale and the instruction system of a computer on which the compiler runs. For this reason, ISSP ALGOL prohibits to use a designational expression and an array identifier as an actual parameter corresponding to any formal parameter called by value, in addition to the ECMA subset restrictions.

As to the processing of arithmetic or Boolean expressions, we do not respect them here. It is only mentioned that Bauer's method [2] was adopted with some modifications. The following sections partially describe a storage allocation scheme on block structures and a processing method of parameters of procedures. ISSP ALGOL compiler was coded in an assembler language but the programs shown in this paper are described in an ALGOL-like language.

2. *The Computer Concerned and the Phase of the Processor*

The computer, FACOM 202, on which ISSP ALGOL runs is a medium size one having the one-address instruction system, the core memory of 8192 words with the word length of 24 bits, and the addition time of 67 μs in the fixed mode.

The processor consists of three blocks of program recorded on a magnetic tape and works under a master control program existing always in the core memory.

1) compiler (5712 steps) 2) loader (1050 steps)

3) standard procedures and administrators (1134 steps)

During translation it needs further one magnetic tape as working use.

3. *Storage Allocation Schemes*

To permit recursive call, all variables and link data have to be stacked into a stack but their locations relative to the stack top at the entry of the innermost

This paper first appeared in Japanese in Joho Shori (the Journal of the Information Processing Society of Japan), Vol. 5, No. 1 (1964), pp. 9–20.

* The Institute for Solid State Physics, University of Tokyo.

block including their declarations are decided in the compiling phase. The address of the stack top has to be dynamically evaluated and stacked to recover the information corresponding to an outer call of the same block. These points are the main part of Dijkstra's and Watt's method [5], [6].

To minimize the amount of the storage space for variables, this allocation scheme is very effective but leading to too slow object programs, in the case of not being allowed recursive call. If the storage space for variables declared in any procedure body is reserved as far as the exit of the block in which this procedure is declared, it is possible to decide the absolute addresses before the run time of the object program at the expense of the amount of the storage space (static allocation).

The address of any dynamic array is evaluated at run time, so its allocation has to be controlled using a run time stack (dynamic allocation). To control dynamically the block structure there is Watt's method which administrates only at the entry of block (including procedure). It is very convenient because of the one point administration, but the number of the innermost block including the activation of the entered one has to be passed to the latter. And, in the case of a function designator given as an actual parameter, this block number is not decided in the compiling phase. Consequently we take a way which divides the controlling into three parts i.e. at block entries, block exits, and labels. Those are essentially the same as Naur's method [4], with the exception of the label administration. The administration through any label requires as a parameter the number of the innermost block including the label. Assuming that any recursive call is prohibited and the program is not segmented, this way simplifies the processing in the compiling phase because the block number is given in advance of the processing.

4. *Programs for a Static Allocation (in the Compiling Phase)*

Suppose that the following nonlocal variables and arrays have been declared.

integer INDEX, LOCAL MAX, MAX, S INDEX, BLOCK NUMBER, CBR, CBN;

comment CBR stands for the current block reference and CBN stands for the current block number;

integer array STACK 1, STACK 2 [1 : N];

Here INDEX is a counter pointing the first free location of the storage space for variables. The maximum value reached by INDEX is recorded to MAX. The identifiers are put into the array, STACK 1 and their corresponding informations i.e. their kinds, types, and the values of INDEX (stand for their addresses) into STACK 2. S INDEX is a pointer pointing the first free element of STACK 1 and STACK 2. The value of S INDEX at the entry of the current block is kept in CBR and used to cancel the storage space for variables declared in the block at the exit. At the entry of any ordinary block it is taken place the following processing.

procedure COMMON BLOCK ENTRY;

begin STACK 1 [S INDEX]:=CBR; STACK 1 [S INDEX+1]:=CBN;
STACK 2 [S INDEX]:=INDEX;
STACK 2 [S INDEX+1]:=LOCAL MAX;
CBR:=S INDEX; CBN:=BLOCK NUMBER;
S INDEX:=S INDEX+2; BLOCK NUMBER:=BLOCK NUMBER+1
end

The declaration of any variable *I* is processed as follows.

ALLOCATION OF VARIABLE:
begin STACK 1 [S INDEX]:=AS INTEGER ('*I*');
STACK 2 [S INDEX]:=INDEX {with kind and type};
INDEX:=INDEX+1; S INDEX:=S INDEX+1;
if LOCAL MAX<INDEX **then** LOCAL MAX:=INDEX
end

The allocation of static array becomes slightly complicated because the information of the mapping has to be reserved, but it is essentially the same as that of a simple variable.

If an identifier appears as an actual parameter of any procedure call, its kind is unknown except the cases that the identifier is already declared or used as a label in advance of the call in the innermost block including this one. In this case the identifier is provisionally processed as a label and the final process is left to the exit of the block.

The processing of a label declaration *I*: is substantially the same as above but that the information marking the declaration is put into one bit in the element of the STACK 2 corresponding to *I* and the following instructions are generated.

DEF *Pi*; CALL LABEL ADM (value of CBN);

Here *Pi* is a symbolic address corresponding to the label *I* and LABEL ADM is an administrator for labels.

At the end of any block, the all identifiers declared in the block are cancelled. With regard to an identifier not declared in the block and processed as a label, searching the same identifier in the region of the STACK 1 corresponding to the innermost block enclosing the block being left, if it is not found out it is merely transferred to the outside block, otherwise there are two cases.

One is the case that the kind of the identifier found outside is also a label and another is not. In the former case, assuming that the symbolic addresses designated to the inside and outside label are *Pi* and *Pj* respectively, since *Pi* and *Pj* must denote the same address, a control instruction to the loader, EQT *Pi*/*Pj*, is generated which serves in the loading phase to replace *Pi* by *Pj*.

In the latter case, the instruction generated for the inside identifier must be replaced by a suitable one to the outside identifier, so a control instruction, REP *Pi*/*OP I*, is generated. In the loading phase this instruction serves to replace, by the instruction *OP I*, all instructions whose address parts include *Pi*.

procedure COMMON BLOCK EXIT;
begin integer E, e, IO, I, i;

```
if LOCAL MAX>MAX then MAX:=LOCAL MAX;
I:=S INDEX-1; S INDEX:=IO:=CBR;
CBR:=STACK 1 [S INDEX]; CBN:=STACK 1 [S INDEX+1];
INDEX:=STACK 2 [S INDEX]; E:=S INDEX-1;
for i:=IO+2 step 1 until I do
if STACK 2 [i]=UNDEFINED LABEL then
  begin for e:=CBR+2 step 1 until E do
        if STACK 1 [i]=STACK 1 [e] then
        begin if STACK 1 [e]=LABEL then
              PRODUCE (EQT Pi/Pj) else
              PRODUCE (REP Pi/OP I); go to REPEAT
        end;
        STACK 1 [S INDEX]:=STACK 1 [i];
        STACK 2 [S INDEX]:=STACK 2 [i];
        S INDEX:=S INDEX+1;
REPEAT: end
end
```

For the declaration of any procedure, the storage space for variables declared in the innermost block including this procedure is not to be overlapped by that to be used inside this procedure, then the next programs are required.

```
PROCEDURE ENTRY:
begin COMMON BLOCK ENTRY; LOCAL MAX:=INDEX end;
PROCEDURE EXIT:
begin COMMON BLOCK EXIT; INDEX:=LOCAL MAX;
      if LOCAL MAX<STACK 2 [S INDEX+1] then
      LOCAL MAX:=STACK 2 [S INDEX+1]
end
```

5. *A Dynamic Storage Allocation Scheme*

If the recursive call of procedures is not allowed, dynamic arrays are only concerned in the dynamic storage allocation. Opposing to the static case, the processing of procedure exits is not different from that of ordinary blocks in this case. Consequently hereafter blocks stand for ordinary blocks and procedures.

Assume the next declarations in run time of any object program.

```
array D STACK [1:N];
integer D INDEX, CBR, CBN;
```

Dynamic arrays are allocated into D STACK. The administrators BLOCK ENTRY and BLOCK EXIT to be executed at the entry and exit of blocks are described below.

```
procedure BLOCK ENTRY (BN); value BN; integer BN;
comment BN is block number;
begin D INDEX:=D INDEX+2; D STACK [D INDEX-2]:=CBR;
      CBR:=D INDEX-2; D STACK [D INDEX-1]:=CBN; CBN:=BN
```

```
end;
procedure BLOCK EXIT;
begin D INDEX:=CBR; CBR:=D STACK [D INDEX];
      CBN:=D STACK [D INDEX+1]
end
```

The administrator for labels is as follows.

```
procedure LABEL ADM(BN); value BN; integer BN;
begin integer I; for I:=CBN while I ≠ BN do BLOCK EXIT
end
```

6. *Parameters of Procedures*

Only the processing for parameters called by name is described. Any procedure declaration is translated into a program segment of the form shown in the following.

P: BLOCK ENTRY (*n'*); PARAMETER TRANSFER (*n*);
PLANT RETURN INTO # OF END; **go to** BEGIN;
*L*1: formal location 1; ....; *Ln*: formal location *n*;
BEGIN: body of procedure *P*; BLOCK EXIT; END: **go to** #;

Each formal parameter respectively corresponds to each formal location into which an instruction corresponding to an actual parameter is transferred by the administrator, PARAMETER TRANSFER, when this procedure has been called. Procedure PLANT RETURN INTO # OF END serves to plant the return address for the procedure call to the go to statement labelled with END.

With regard to any actual parameter, an instruction corresponding to the parameter is generated as a program parameter following the procedure call instruction. If an actual parameter is an expression other than a variable or a label, a program segment to evaluate the expression is generated as a subroutine and its call instruction is used as the program parameter.

The instructions for possible actual parameters are shown in the Table 1.

Table 1
Actual parameter and its corresponding program parameter.

Actual Parameter	Program Parameter
Simple variable, *X*	LOAD *X*
Array indentifier or string, *A*	SET *A* into B-REG
Procedure or switch identifier, *P*	CALL *P*
Label, *L*	GO TO *L*
Formal parameter, *Li*	EXT *Li*
Other expression, *PE*	CALL *PE*

When a formal parameter appears in the procedure body, the instruction EXT *Li* is generated which executes an instruction in the corresponding formal location *Li*.

Let's consider a case that there is a formal paramater in a left part of an assignment statment, $L:=E$. In this case, it does not require the value but the address of the corresponding actual parameter.

We resolved this case by using an administrator which fetches the address of the actual parameter from the corresponding instruction. The form of the object program segment in this case is as follows.

EXT L; ADDR ADM (Pi); PLANT ADDR into ♯ of Pi; E; Pi: STA ♯;

If the actual parameter is a subscripted variable the instruction EXT L causes to call the corresponding subroutine which evaluates its address and value.

This method is not too inefficient to treat formal parameters comparing ordinary variables with the exception of left part variables.

7. *Some Remarks for the Recursive Call*

A speculation which extends monotonously the preceding method is described to simplify the block reference in the case including the recursive call.

Assume instruction form, $OP\ A/B$, to a machine on which object programs run, where OP represents the operation part of instructions, A and B are both the address part giving the actual address of an operand $A+B$.

Assuming that the relative location of any variable is designated to B, and the address of the block reference is designated to A in the indirect mode, any operand can be referred by $A+B$.

Watt has proposed the way that, considering the exit by go to statement, when the block having the static level number j is called within the block having i ($j<i$), all the informations of block references from j to i are stacked.

Because the address of A is peculiar to each block, it is in place of the block number and its content gives the block reference of the corresponding block. So, providing A is an array whose subscript a is a block number, the administration at the entry and exit of the block is very much easily executed as follows.

```
procedure BLOCK ENTRY (a); value a; integer a;
begin INDEX:=INDEX+2; STACK [INDEX-2]:=CURRENT A;
      STACK [INDEX-1]:=A[a]; CURRENT A:=a; A[a]:=INDEX-2
end;
procedure BLOCK EXIT;
begin INDEX:=A [CURRENT A]; A [CURRENT A]:=STACK [INDEX+1];
      CURRENT A:=STACK [INDEX]
end
```

INDEX and STACK have the different meaning from those of the preceding sections.

In the case of leaving a block by any go to statement, it has to be controlled through the go to statement but not the declaration of the corresponding label. If the destination of the go to statement is a formal parameter, it needs as the actual parameter not only the address of the label but the content of $A[a]$ of the block in which a corresponding actual parameter is declared.

References

[1] NAUR, P. (Ed.), Revised Report on the Algorithmic Language ALGOL 60. *Communications of the Association for Computing Machinery, 6,* 1 (1963), 1–17.

[2] SAMELSON, K. AND F. L. BAUER, Sequential Formula Translation. *Communications of the Association for Computing Machinery, 3,* 2 (1960), 76–83.

[3] JENSEN, J., P. MONDRUP AND P. NAUR, A Storage Allocation Scheme for ALGOL 60. *Communications of the Association for Computing Machinery, 4,* 10 (1961), 441–445.

[4] JENSEN, J. AND P. NAUR, An Implementation of ALGOL 60 Procedures. *Nordisk Tidskrift for Informationsbehandling 1,* 1 (1961), 38–47.

[5] DIJKSTRA, E. W., Recursive Programming. *Numerische Mathematik 2,* (1960), 312–318.

[6] WATT, J. M., The Realization of ALGOL Procedures and Designational Expressions. *Computer Journal 5,* 4 (1963), 332–337.