ELSE — The proper transfers are set following the first statement or expression of a conditional.

{go to ___} → π ; Set address in C(α[a]) to next target instruction address ; place current target instruction address in α[a].

THEN — This sets a previously unspecified transfer

Set address in C(α[a]) to next larger instruction address

CC — This serves merely to adjust the temporary counter in conditional expressions.

if α[a] ≠ 'η[h]' then STV ; a := a−1 ; h := h−1

CC1

h := h+1 ; a := a+1 ; α[a] := 'η[h]'

Practical Considerations

Specifications for other parts of the translator may be obtained using the same principles that were used in connection with the expressions. We close with some comments on the implementation of these for a given machine.

An internal representation may be chosen for the ALGOL delimiters and "identifiers" which simplifies coding. This means that the precedence level of operations and relations is apparent in the representation and yet that they may be considered alike as incoming symbols. Type designations must be included in the representation of the "identifiers." States must be given internal representations which permit the operations to be easily incorporated as part of the states in the cases E2 and E3.

The coding for the building blocks is for the most part self-evident. In our discussion the consideration of type has been omitted. This can be taken into account in the building blocks EXU and EXB, and needs be treated nowhere else. Provision must be made in the two blocks for writing the target equivalent for each binary operation and relation. It follows that they will in general be much larger pieces of code than indicated in our summary.

The matrix represents essentially a switching program, which may be implemented in several ways. The matrix itself, complete with error stops and without the OTHERWISE feature may be put into memory and the switching may be accomplished by a double jump. This is fast, but perhaps wasteful of memory space. Another approach lists the permissible pairs with the addresses of the corresponding codes in a table. A table look-up operation then does the switching. A final possibility is to write the program for each column separately. The controlling state then determines which of these programs to enter.

It must be emphasized that in the approach outlined here, the scan for expressions is not considered isolated from the scan for the rest of the ALGOL program. The recursive nature of ALGOL is such that this cannot be readily done. Unlike in some other algebraic systems, it is not desirable to have separate programs to handle different kinds of statements. In following the approach outlined, not only for the expressions which have merely served as an example of techniques, but for the processing of the entire language, the recursive nature of the language is automatically reflected in an equally recursive translation process.

Use of Magnetic Tape for Data Storage in the ORACLE-ALGOL Translator

H. Bottenbruch

Oak Ridge National Laboratory, Tennessee

The ALGOL 60 language was designed to simplify formulation of computation processes to be executed on electronic computers. Emphasis in the design of the language was put on ease of formulation rather than efficient use of the facilities of a computer. Where in many cases a translation program can produce an efficient machine program from an ALGOL description of a computation process, there are certain areas which a translator cannot handle efficiently. In such cases, in order to insure efficient use of computers, information has to be added to an ALGOL program to allow a translation program to produce efficient programs. One of these cases is use of auxiliary storage, which is particularly important for a machine like the ORACLE with its limited internal memory (2000 words) and its powerful tapes.

This paper describes how to use magnetic tape for data storage in the ORACLE-ALGOL Translator. Every attempt was made to keep the additional information which has to be supplied to the translator if tapes are used as small as possible. It was an important consideration to keep this additional information separated from the ALGOL program so that the operational meaning of a program is not

changed when this information is ignored. Furthermore, it was attempted to keep up with the high standard of notational convenience set forth in ALGOL 60. Compromises were necessary, however, to secure efficiency in the target program.

We will use tapes only for storage of arrays, not for simple variables. Those arrays which are to be stored on tapes are declared so. The translator inserts the necessary instructions for transmitting elements of the tape arrays between high speed storage and tapes. There is one subroutine (called tape-positioner) which *at object time* decides whether a transmission is necessary or not, and which executes it if this is necessary.

1. Brief Description of the ORACLE Tapes

There are four tapes, each having a capacity of approximately 250,000 words (2^{18}). Information is stored in blocks of 128 words. After execution of a tape order the read-write head of a tape stops in the gap between two successive blocks. The following tape operations are used in this paper:

(a) Read forward: Read the block on tape t which is to the *right* of the read-write head into locations A, A+1, · · · , A+127. The leftmost word of the book will go into location A, the rightmost word will go into location A+127.

(b) Read backward: Read the block on tape t which is to the *left* of the read-write head of tape t into locations A+127, A+126, · · ·, A. The leftmost word of the block will go into location A+127, the rightmost word will go into location A.

(c) Write forward: Write information from location A to A+127 into the block immediately to the *right* of the read-write head of tape t. The word in location A goes into the leftmost word of the block, etc.

(d) Write backward: Write information from location A+127 to A into the block immediately to the *left* of the read-write head of tape t. The word from location A+127 goes into the leftmost word of the block, etc.

(e) Hunt forward: Move the read-write head of tape t, b blocks to the right.

(f) Hunt backward: Move the read-write head of tape t, b blocks to the left.

In the ORACLE, operations 5 and 6 go on in parallel to other operations not involving the moving tape. Execution of of operations 1 to 4 takes approximately 50 milliseconds. (In comparison: Floating point arithmetic (programmed) takes 3 milliseconds per operation.) The blocks can be used in any desired order, and information at the end of the tape is not lost if a block at the beginning of the tape is used in an operation. Two machine instructions are needed to execute these operations.

In reading and writing information, any number of words can be transferred. We do not make use of this facility.

2. Specifications

2.1. A new declaration is added to the language which declares that one or more arrays are to be stored on one of the four tapes during execution of the program. The declaration begins with the new delimiter **tape**, is followed by a number i ($0 \leqq i \leqq 3$), and then a normal ALGOL array declaration follows.

2.2. Only one declaration may be given for each tape.

2.3. These declarations have to be at the very beginning of each program.

2.4. The arrays will be stored on the designated tape in the order in which they are written in the tape array declaration. The first element of each array is always stored in the first location of a (physical) block on the tape.

2.5. The elements of tape arrays are stored in lexicographical order; that is, a[4, i, k] is stored after a[3, ℓ, m] regardless of the values i, k, ℓ, m. And a[3, 4, k] is stored before a[3, 5, m] regardless of the values of k and m, etc. There are no gaps between the elements of one array.

2.6. The elements of a tape array can be used inside any program in the same way as the elements of any non-tape array.

2.7. In order to assure efficiency in using the tapes, the following new statements are added to the ORACLE-ALGOL language:

tapestate i := **read only**
tapestate i := **write only**
tapestate i := **read write**

} i = 0, 1, 2, 3

The effect of the first statement is that information will never be written from high-speed storage onto tape i after that statement is given, until statement two or three is given for that same tape. Similar remarks hold for the second and third statements.

2.8. During execution of the target program one block of each tape which is declared will be "represented" in high-speed storage. If an element of a tape array is referred to, a subroutine first determines whether that block of the tape on which this element lies is represented in high speed storage. If so, the required operation is executed without any tape motion.

If it is not represented, the following action will take place depending on the state of the tape:

(a) In the *read only* state the block in which the referenced element lies is called into high speed storage, and a certain location which keeps track of the tape standing is changed to the new position.

(b) In the *write only* state, the block just represented in high speed storage is written on tape, then a hunt order is given to the block in which the referenced element lies. The location mentioned in (a) is changed accordingly.

(c) In the *read write* state, the block just represented in high speed storage is written on tape, and then the block in which the referenced element lies is read into high speed storage.

In order to use the tape arrays efficiently, the elements stored on tape must be used sequentially. The subroutine which transmits information between tapes and high speed storage uses the backward or forward read and write order, whichever is fastest. It automatically changes the orders for computing the address of a tape array element depending on the way (backward or forward) in which the tape information is recorded in high speed storage.

2.9. Using the tapestate *write only* is not foolproof. It is impossible to make this foolproof without treating it the same and inefficient way as the *read write* state. Here is a discussion of a possible blunder using the *write only* state.

If a tape is in *write only* state, and not all the elements of the (physical) block B1 represented in high speed storage are changed before an element in a different block B2 is referred to, all the storage locations of B1 on the tape will nevertheless be changed to whatever the information in high speed storage was before the first element in B1 was called for.

This means, for instance, that it is impossible, in the *write only* state, to change all elements of an array except the first one, or that it is impossible to change only every second element, etc.

If a computation process requires such things, the *read write* state must be used.

Since a new block is started with each array, no errors of the above mentioned type can occur in case all elements of an array are changed (or if the elements which are not changed are no longer needed).

If, in matrix work, the elements of certain rows undergo changes, other rows being unaffected, use of the inefficient *read write* state can be avoided by giving an array-declaration where the number of columns is a multiple of 128. This assures that a new physical block is started for each row.

3. Implementation

3.1. A numerical address will be associated with each location on each of the tapes. The addresses used for tape 0 range from 2^{18} to $2^{19}-1$, those for tape one range from 2^{19} to $2^{19}+2^{18}-1$, etc. This means that the 19th, 20th and 21st bit of the binary representation of an address determine the tape number.

A tape declaration associates one of the above mentioned numerical tape addresses to each element of each array in accordance with 2.4 and 2.5.

If, for instance, a tape declaration runs as follows:

tape 0 **array** a, b[1 : 4], c[2 : 3, 4 : 7];

the following addresses will be associated with the elements of these arrays:

Element	*Address*	*Element*	*Address*
a[1]	2^{18}	c[2, 4]	$2^{18}+256$
a[2]	$2^{18}+1$	c[2, 5]	$2^{18}+257$
a[3]	$2^{18}+2$	c[2, 7]	$2^{18}+259$
a[4]	$2^{18}+3$	⋮	⋮
b[1]	$2^{18}+128$	c[3, 4]	$2^{18}+260$
⋮	⋮	⋮	⋮
b[4]	$2^{18}+131$	c[3, 7]	$2^{18}+263$

3.2. One part of the implementation is the construction of a program which scans the tape array declarations and builds up a table which for each array contains the information:

1. First location of the array.
2. Control information determined by the subscript bounds of the declaration.

The work to be done here is very similar to the work which has to be done for normal arrays, the only difference being the way in which the initial address for the tape array is computed.

3.3. Non-tape arrays are handled in a particularly simple way in the present ORACLE-ALGOL Translator, although in some cases it is not very efficient. Whenever an element of an array (subscripted variable) appears in a program, the translator creates a call for a subroutine (called "address calculation") which computes the address of that element and places it into a special location. We will call this location ADD. This is done regardless of where and in which connection the subscripted variable is used. After this subroutine call the translator inserts instructions to use the address. The same scheme can be used for tape arrays. The tape-address of a subscripted variable is first computed by the routine "address calculation." For tape arrays, however, another subroutine (the "tape positioner," see below) is called after execution of routine "address-calculation." On exit from the "tape positioner," the required tape location is represented in high speed storage, and its associated high speed storage address is in location ADD. This address can be used in the same way as the address of an ordinary subscripted variable.

3.4. The major part of the implementation is the construction of the subroutine "tape positioner." This subroutine, when supplied with a tape address in location ADD, controls the necessary tape motions and transmissions, and replaces the tape address in ADD by a high speed storage address.

Following is a "quasi" ALGOL procedure which does this. The procedure uses the following quantities:

1. An array FRTL (one element for each type). FRTL[t] contains the first location of tape t which is represented in high speed storage.
2. An array FHS (one element for each tape). FHS[t] contains the first high speed storage location used by tape t.
3. An array tapestate (one element for each tape). Tapestate [t] contains, in a certain code, the present state of tape t (read only, write only, read write).
4. An array Di[t] (one element for each type). Di[t] contains, in certain code, whether tape t is represented in forward or backward mode in high speed storage.

The quantities 1, 2, 3, and 4 are global quantities to the procedure tape positioner in the sense of ALGOL 60.

The only parameter to the procedure is the referenced address (a tape address) given in ADD.

The procedure needs the local quantities t, δ, and d. Here, t denotes the tape number involved, δ is the difference between the required address and the first address of tape t which is represented in high speed storage, and d is the difference in blocks between these two addresses.

The tape motions and transfers to be carried out depend on the tapestate. They are written down in the program only for the *read write* state. The actions to be carried out

for the other two states form a subset of those for the read write state. The proper subsets are given by yes-no information.

In the following quasi-ALGOL program, the words "read write," "forward" and "backward" have been used to represent the numbers by which these words are represented inside the machine. The statements which handle tape motions are self-explanatory. The high speed storage locations which are involved in transmitting information are not explicitly given in the read and write statements. The first of these locations is contained in FRTL[t].

The subroutine assumes that on entering it the read-write head of each tape will be at the beginning or end of the block represented in high speed storage depending on the tapestate and the direction, according to the following table. The subroutine will leave the head in a position determined by the same table.

tapestate [t]	Di[t] = forward	Di[t] = backward
Read only	end	beginning
Write only	beginning	end
Read write	beginning	end

A quasi-ALGOL program for the "Tape positioner":

```
    t := ADD ÷ 2 ↑ 18;
L: delta := ADD − FRTL[t];
   if ¬ (0 ≦ delta ∧ delta < 128) then begin comment if this
     condition is fulfilled, the required tape address is not repre-
     sented in high speed storage;
     d := ent (delta/128);
   if tapestate [t] = read write then begin
                                                    Read only   Write only
   if Di[t] = forward then begin                      No          Yes
                        Write on tape t
                        forward;                      No          Yes
                        d := d − 1 end                Yes         Yes
                      else Write on tape t backward;  No          Yes
   if d < 0           then begin                      Yes         Yes
                        Di[t] := backward;            Yes         Yes
                        Hunt backward − (d + 1)       Yes         Yes
                        on tape t;
                        Read tape t backward;         Yes         No
                        Hunt forward 1 on tape        No          No
                        t end
                      else begin
                        Di[t] := forward;             Yes         Yes
                        Hunt forward d on tape t;     Yes         Yes
                        Read tape t forward;          Yes         No
                        Hunt backward 1 on tape t     No          No
                        end end;
   FRTL[t] := 128 × ent (ADD/128); go to L end;
   if Di[t] = forward then ADD := FHS[t] + delta else
                           ADD := FHS[t] + 127 − delta;
```

3.5. The statements mentioned in 2.7 will be translated into a call of a subroutine which changes the tape states. In addition to changing the variable tapestate [t], certain tape motions must be done by this subroutine and, if the state is changed from *read write* or *write only* to *read only*, the information in high speed storage must be written on tape.

More exactly, the following tape actions must be taken:

Old State	New State	Action to be taken Di[t] = forward	Di[t] = backward
Read only	Read write	Hunt backward 1 on tape t	Hunt forward 1 on tape t
Read only	Write only	Hunt backward 1 on tape t	Hunt forward 1 on tape t
Read write	Read only	Write forward on tape t	Write backward on tape t
Read write	Write only	None	None
Write only	Read only	Write forward on tape t	Write backward on tape t
Write only	Read write	None	None

4. Example for Using Tape Arrays

A sequence of measurements is punched on a paper tape. Each measurement consists of a series of numbers m_i ($i = 1, 2, \cdots, n$) which lie between 1 and 10. The number zero is punched after each series to indicate the end of that measurement. n may differ from one measurement to another and is less than 100000. The number -1 is punched at the very end of the tape instead of zero. It is required, for each measurement, to punch the numbers $(m_i - m)^2/p$ where

$$p = \frac{1}{n}\sum_{i=1}^{n}(m_i - m)^2;$$

m is the mean value of the m_i for each measurement. The numbers $(m_i - m)^2/p$ may, within each measurement, be punched in any order desired. The control numbers zero and -1 are inserted as in the original tape.

The formula

$$p = \frac{1}{n}\cdot\sum m_i^2 - m^2$$

is used to compute p, saving one pass through the numbers.

```
tape 1  array a[1 : 10^5]; integer i, n; real s, t, c, m, p;
        tapestate 1 := write only
M:      s := 0; t := 0;
        for i := 1 step 1 until 10^5 do begin n := i;
            read c; if (c = 0) ∨ (c = −1) then go to L;
            a[i] := c; s := s + c; t := t + c↑2 end
L:      n := n − 1;
        m := s/n; p := t/n − m↑2;
        tapestate 1 := read only
        for i := n step −1 until 1 do punch (a[i] − m)↑2/p;
        punch c;
        if (c = −1) then stop else go to M;   [See Footnote 1.]
```

Using the tapestate *write only* does not cause the trouble mentioned in 2.9. The tape will first be written in the forward mode and will then be read in the backward mode. So if B blocks are used by one measurement the tape will move 2B blocks, with the tape in rewound position after each measurement is processed.

[1] The program is written in ORACLE-ALGOL, with the extensions set forth in this paper.

Using the tapes in an uncritical way might well result in a total tape movement of 12B blocks, namely if

(a) the *read write* state is used throughout,

(b) the trick for computing p is not used,

(c) the elements of each array are always used in the order from 1 to n.

If no provision would be made internally for the forward and backward mode, the best possible program would result in a total tape movement of 4B blocks.

If the *read write* state would be used throughout and no provision would be made internally for the forward and backward mode, the best possible program would result in total tape movement of 6B blocks.

5. Extensions of the Techniques to Storage of Programs on Tapes

A program which is too big to fit into the memory at one time must be divided into segments. Segmentation in the ORACLE-ALGOL translator will be done in the same way as is done in ORBIT [1], because the ORBIT routines can be used with practically no change. If we started from scratch, we would do segmentation in the following way:

A new delimiter **segment** is added to the language. A statement may be declared a "tape-segment" by preceding it with the string:

tape i **segment** $\cdots$, where i = 0, 1, 2, 3.

A segment is a statement, and it is automatically a block, even if no declarations are given for this statement. A segment may itself be part of another segment, which may even have a different tape number. If a statement S is declared a **tape** i **segment**, it will normally be stored on tape i, and will be transmitted into high speed storage only if control is transferred to that segment S. The storage space for S will be made available for storage of other segments after control has left segment S. If S is a block with own variables, these are stored on tape i along with the program for S, and they are transmitted to high speed storage if control is transferred to S. These variables must be written on tape after control has left S. A subsegment SS of S is called into high speed storage only if control is transferred to SS. In this case, S remains in high speed storage together with its own variables.

Storage space on tape and in high speed storage can be allocated to the segments at translation time, but this allocation is much more complicated than it is in the normal case. The translator has to construct a table which contains, for all segments, the storage space allocated to them. A **go to** statement to a label of a segment must be translated into a call of subroutine which transmits the segment into high speed storage and then jumps to the first order of that segment. The above mentioned table is not required at execution time.

The idea can be extended to procedures. A procedure which is declared a **tape** i **procedure** is normally stored on tape i and is transmitted to high speed storage if and only if it is called. Procedures which are local to a segment S and which are not declared tape procedures are considered part of S, and are transmitted along with the code for S whenever control is transferred to S.

ACKNOWLEDGMENT

This paper is the result of discussion which the writer had with A. A. Grau, A. C. Downing, and G. J. Atta on the subject of tape files.

REFERENCE

1. GRAU, A. A. ET AL: ORACLE Binary Interal Translator (ORBIT) Oak Ridge National Laboratory, Sept. 1959, Central Files Number 59-9-20.

The CLIP Translator

Donald Englund and Ellen Clark

System Development Corporation, Santa Monica, California

Introduction

CLIP, a compiler and language designed for information processing, is the joint research project of E. Book, H. Bratman, Ellen Clark, D. Englund, H. Isbitz, H. Manelowitz and E. Myers of the System Development Corporation. A CLIP compiler for the IBM 709 computer has been written. As a test of the adequacy of the language, the CLIP compiler was written in its own language and has successfully reproduced itself.

The compiler is divided into two parts, a generator and translator. This paper will concern itself primarily with the translator.

CLIP Language

The CLIP language is based on ALGOL but has the following additional data declarations which were found to be needed in information processing:

1. TABLE declarations specify the subscripted variables or items which make up the table. The size and form (Boolean, alpha-numeric, integer, signed integer) of each item are declared, and provision is made for packing items into parts of a machine word. Initial data may be supplied if desired.
2. STRING declarations define a contiguous set of alphanumeric characters. Initial data may be given. Operations are permitted on any contiguous subset of a string.
3. ORIGIN declarations permit the programmer to specify the