THE BURROUGHS ALGEBRAIC COMPILER

FOR THE 205

PROGRAMMER'S MANUAL

Automatic Programming
Burroughs Corporation
Pasadena, California

Preliminary Edition
Bulletin 3041
February, 1961

Minor Revision of
December 1960 Edition

I. INTRODUCTION

In order to solve a complicated problem on a high-speed digital computer, a large amount of time is usually required for a programmer to translate the specifications of the problem into the rather restricted machine code. Later, if the same problem is to be solved on a different computer, it is necessary to spend an appreciable amount of time re-programming for the new machine.

A lot of the work involved in coding these problems is actually quite mechanical; and since computers thrive on such tasks, it is quite natural that automatic programming systems, "compilers" in particular, would be developed. A compiler program takes in the description of a problem, written in a language which closely resembles ordinary English and mathematical terminology, and translates it into the machine language of a computer.

After many compilers for many computers had been written, it became evident that a single compiler language which would be standard for use with every computer was desirable, so people would not have to learn a new language with each new machine. As a result of a subsequent international conference, "Algol 58" was born. The problem language of the Burroughs Algebraic Compiler for the 205 has been patterned after this international language.

The language of Algol 58 was full of so much generality, however, that compilation on a medium-size computer was found to be quite impossible. (In fact, it was quite difficult to accomplish the translation even on the largest computers available at that time!) Therefore, the full repertoire of Algol 58 could not be put onto the 205. The language of the compiler has, accordingly, been chosen to include as much of Algol 58 as possible, including all of the essential features, and the result is the language to be described here, a language which, although less than Algol 58, is still more powerful than any other language now used on machines of comparable size, and a language which has been found quite easy to learn and to use.

In addition, the language of this programming system has been chosen so that it is completely contained in the language of the Burroughs Algebraic Compiler for the 220 Electronic Data

Processing System; in other words, any program correctly written in this language will work properly if compiled and run on the larger and faster 220.

As an example of Algol language, consider the formula for computing one of the roots of the equation $AX^2+BX+C = 0$:

$$X = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

The chief difference when formulas are translated into Algol form is that everything must be written on one line:

X = (- B + SQRT(B*2-4A·C))/2A.

Here "SQRT" replaces the radical sign, * means taking of a power, and · is a symbol for multiplication.

II. BASIC SYMBOLS

The Alphabet

The first step in formulating a language is to choose the list of symbols that will be used to construct it. This list of symbols is known as the "alphabet" of the language. This Compiler's alphabet consists of the twenty-six capital letters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

the ten digits

0 1 2 3 4 5 6 7 8 9

and ten special symbols

+ - . / * , ; ( ) =

which are used to indicate arithmetic operations and punctuation marks. All of these symbols appear on the standard "Fortran" keypunch, except the semicolon ";" for which the dollar sign "$" is substituted.[1] No other characters except those listed above and blank spaces have any meaning in this language.

Identifiers and Numbers

The preceding symbols are strung together to form two types of basic entities: identifiers and numbers.

An identifier is any string which begins with a letter and is possibly followed by letters and/or digits. For example, "X" and "ALPHA4" and ALGOL205" and

"THISISALONGIDENTIFIERWITHJUST41CHARACTERS"

are identifiers. But "GROSS-PAY" and "4ALPHA" are not identifiers unless they are written "GROSSPAY" and "FOURALPHA".

Several words have a special meaning to the compiler, however, and they may not be used as identifiers. These words (for example, BEGIN, SQRT, END, IF, FOR, AND, UNTIL, ARRAY, SUBROUTINE) are all listed in Appendix A, and they will also appear in the text below as their function is described.

1. On non-Fortran keypunches, the special characters % ¤ # are substituted for ( ) =, respectively.

Although identifiers may extend up to 43 characters in length and the Compiler will still understand, the speed of punching the input and the speed of compilation are reduced when long identifiers are used, so there is no point in making up very long ones unless some ease in programming results.

A number is even simpler to describe: It is just a string consisting of digits only. For example, "0" and "1234567890" and "004" are numbers.

The reader is probably thinking at this point, "Does this guy think he's telling me what a number is?" Well, of course everyone knows what a number is, in the ordinary sense, but the definitions given here are to explain what the words "number" and "identifier" will mean when used later in this manual. Any other words, say even "ngogn" and "fmurg," might have been defined instead of number and identifier and used later on.

III. VARIABLES AND CONSTANTS

Types of Arithmetic

This compiler language features two distinct types of arithmetic operations. The first type deals only with the integers, and no fractional or decimal values are allowed. Such arithmetic is limited to integers of ten digits or less, so the allowable values are

-9999999999,-9999999998,...,-1,0,1,...,9999999998,9999999999.

The result of adding, subtracting, multiplying, dividing, or taking powers of integers is an integer. Notice, however, that division (e.g., -5 divided by 3) and taking powers (e.g., 5 to the -1 power) lead to fractions under ordinary rules; in such cases the fractional part is disregarded, so -5 divided by 3 equals -1; and 5 to the -1 power equals 0. For this reason, integer arithmetic is rarely used for division and taking negative powers; it is most used for calculating subscrips or for keeping track of how many times something is to be done.

The other type of arithmetic has a much broader range; it is known as "floating point" or "real-valued" arithmetic. A floating point quantity r has eight significant figures, and its magnitude can be zero or anywhere in the range

$$10^{-51} \leq |r| < 10^{49}$$

Examples of floating point values are -3.1415927 and 2.8400000×10^{-6}. Of course, floating point values include integers like 1.0000000 and 365.00000 also.

The name floating point comes from the fact that decimal points seem to hop around during floating point calculations. The reason for having two types of arithmetic is that integer arithmetic is faster for computing subscripts and for counting.

Any variable used in a program may be either integer or floating point, but not both. Those variables which are to be integer-valued are declared as such on an "INTEGER" declaration (see Chapter VI). Integer variables take on only integer values, and floating point variables take on only floating point values. It is possible, however, to use both integer variables and floating point variables in the same formula.

Precise rules about the integer and floating point arithmetic and allowable conversion between the two are listed in Appendix B.

Constants

Integer constants are represented in the language by numbers (e.g., 0 1728 100000).

A floating point constant can be represented in several ways:

a. As two numbers with an imbedded decimal point (e.g., 3.1415927   1.0   0.000004   0.0).

b. As a number followed by a power of ten by which it is to be multiplied. The power of ten is indicated after a double asterisk. For example, 1×10^6 can be written either 1**6 or 1**+6. 12×10^{-20} can be written as 12**-20.

c. As two numbers with imbedded decimal point together with a power-of-ten specification (e.g., 1.0**6 2.84**-6).

Notice that 300000.0 3**5 3**+5 30**4 3.0**5 0.3**6 and so on are all equivalent forms for representing three-hundred-thousand, in floating point form.

Variables

Variables come in three flavors: simple variables, vector variables, and matrix variables. The different kinds are distinguished by the number of subscripts: simple variables are unsubscripted, vector variables have one subscript, and matrix variables have two. Subscripted variables are also called "arrays."

An indentifier is associated with each variable, and it is called the "name" of the variable. The same identifier cannot be used for two different variables, not even if one is an array and the other is not.

Examples of simple variables are:

```
OMEGA    I    W13    W14.
```

There is no connection between the variables W13 and W14.

Vector variables have their subscripts set off by parentheses; for example:

```
X(3)    PSI(I)    W(13)    W(14).
```

In this case W(13) and W(14) are part of the same vector, W.

Matrix variables have their subscripts set off by parentheses and a comma; for example:

```
A(1,1)    M(I,J).
```

Subscripts can be of any degree of complexity; they may even contain subscripted variables themselves. We might have

```
    X(I+J)     or     M(X(X(1)),M(A(5,I),W(12))).
```

All subscripts start at 1 and go up to some maximum which is specified on an ARRAY declaration. We will discuss this declaration in Chapter VI.

It is almost always best to use integer arithmetic in subscripts, and floating point belongs there only in peculiar cases. If the subscript is floating point, however, the digits to the right of the decimal point are dropped, there is no rounding. You must be careful in this case, for 3.9999999 is a floating point number very close to 4, but if it is the value of a subscript it is chopped to 3. (This number isn't very hard to achieve, either: 4.0 divided by 3.0 is 1.3333333; the latter times 3.0 is 3.9999999.)

IV. EXPRESSIONS AND PROPOSITIONS

Arithmetic Expressions

Variables and constants are combined with parentheses and operators to form arithmetic expressions. Let us attempt to give a fairly precise definition of the term "expression."

a. A variable or a constant standing alone is an expression.

b. If E is an expression whose first character is not "+" or "-", then +E and -E are expressions.

c. If E is an expression, then (E) is an expression meaning the quantity E taken as a unit. Thus, the negative of X+Y can be written -(X+Y).

d. If E is an expression whose first character is not "+" or "-", and if F is any expression, then F + E and F - E are expressions, meaning the sum and difference of F and E.

e. If E and F are any expressions which are not both constants, then

E·F   E/F   and   E*F

are expressions which mean "E times F", "E divided by F", and "E to the F power", respectively.

f. If E is an expression, so is ABS(E), meaning the absolute value or magnitude of E.

g. An expression in which all variables and constants (exclusive of those appearing in subscripts) are integer is called an integer expression; otherwise, it is a floating point expression.

h. If E and F are integer expressions, MOD(E,F) is an integer expression whose value is the remainder of E divided by F.[1]

i. If E is a floating point expression, then

SIN(E), COS(E), ARCTAN(E), LOG(E), EXP(E), SQRT(E).

---

1. Here F must have a positive value at running time, or the sign of the answer will not agree on the 205 and the 220 computers.

are expressions. The SIN, COS functions take E in radians; if E is in degrees, write SIN(0.017453281(E)) or COS(0.017453281(E)). The ARCTAN function produces answers in radians between -1.5707963 and +1.5707963; to get answers in degrees, write 57.295779 ARCTAN(E). LOG(E) is the natural logarithm; to get base ten logarithms, write 0.43429448LOG(E); to get base ten antilogarithms, write EXP(2.3025851(E)). You may have noticed that the multiply symbol is missing in these examples; that is all right, as we will see below. Other functions besides these may be part of the library at your installation.

j. Additional functions may be defined at compilation time through the medium of procedures (see Appendix C).

Interpretation of Expressions

If you were to insist upon a strict literal interpretation of rules d) and e) above, you would find there is some ambiguity. In the expression X + Y · Z there is some question as to whether X should first be added to Y and the result multiplied by Z, or Y multiplied by Z first and then X added to the result. In other words, does it mean (X + Y) · Z or X + (Y · Z)? Parentheses should be used to express the exact meaning; but if they are not given, the compiler interprets expressions according to the usual mathematical conventions.

To be explicit, whenever there is a choice between two operations as to which is to be done first, taking to a power and negation are done first, then multiplication, then division, and lastly addition or subtraction. In case of ties, the operation proceeds from right to left.

These rules need not be learned if parentheses are filled in explicitly. The insertion of redundant parentheses in no way hurts the compiled output. However, the following examples serve to explain the rules given above:

X + Y · Z	is interpreted	X + (Y · Z)
W · X / Y · Z	is interpreted	(W·X) / (Y·Z)
B * 2 - 4·A·C	is interpreted	(B*2)-(4·(A·C))
X - Y - Z	is interpreted	X + (-Y -Z)
X / Y / Z	is interpreted	X / (Y / Z)
- X * 3	is interpreted	- (X * 3)

The "·" symbol for multiplication may be omitted when no ambiguity is introduced. These cases are the following:

Let I stand for the name of any function or any variable, N for any constant, and W for any simple variable; the cases are

```
)I    )N    )(    W(    N(    NI
```

which may be used instead of

)·I   )·N   )·(W·(N·(N·I

For example, we can write

3N   (A+B)(A-2B)   4(X*2-Y*2)ALPHA.

Notice that the expression A(J) means A·J if A is a simple variable, and A_J if A is a vector.

Mixed Arithmetic

Suppose I and J are integer variables and X is a floating point variable. Then let us consider the evaluation of X·(I/J) and of (X·I)/J: These two formulas are not equivalent expressions! In the first expression we are first to divide I by J. So far there is no mixed arithmetic, for I and J are both integral; thus, the division is done integer-wise, disregarding the remainder. Then, we are to multiply the result by X, so we convert the result to floating point and multiply by X. In the second example, however, we are to do the multiplication first, so first the value of I is converted into floating point and multiplied by X. Then the value of J is converted into floating point and the division is finally carried out.

Now in general, conversion inside an expression from integer to floating occurs only when we are dealing with one of the operations + - · / or * where one of the quantities is integer and the other is floating. In this case, if the integer part is simply a constant (like 123), the compiler automatically re-reads the constant as if it had been written in floating point form (like 123.0).

As a final example of an arithmetic expression, here is a representation of the famous "Wolontis Function":

$$\frac{\sin x}{\sqrt{1 - e^{-x^3}}}$$

In Algol form, this is written:

SIN(X) / SQRT(1-EXP(-X*3))

Propositions

A proposition is a declarative statement which is either true or false. If E and F are any arithmetic expressions, the following are propositions:

E	GTR	F	meaning	$E > F$
E	GEQ	F	meaning	$E \geq F$

E	LSS	F	meaning	$E < F$
E	LEQ	F	meaning	$E \leq F$
E	EQL	F	meaning	$E = F$
E	NEQ	F	meaning	$E \neq F$

We also have the proposition "PCS(1)." This proposition means "the Skip Switch is on" on the 205 and "Program Control Switch Zero is on" on the 220.

If P is any proposition, NOT (P) is also a proposition. And if P and Q are both propositions, we may write "(P) AND (Q)," which is another proposition meaning both P and Q are true, and "(P) OR (Q)" which is a proposition meaning either P or Q or both are true.

Examples of propositions:

```
                 X EQL 0
  (X GTR 0) AND (NOT((PCS(1)) OR (Y*2 LSS 2)))
```

The Use of Blank Spaces

Blank spaces can be inserted freely in most places, to make the input language look more readable. SPACES MUST NOT BE INSERTED IN THE FOLLOWING PLACES, HOWEVER:

a. Between the identifier of a vector or matrix and the left parenthesis following.

b. In the midst of a constant or identifier.

c. Between a prefix character and the three dots following (see the INTEGER declaration).

d. Between the two dots of a "colon" (see Labeled Statements).

e. Between the label and the left parenthesis following in a FORMAT, INPUT, OUTPUT or Procedure Declaration (see Chapter VII).

Of course, spaces <u>must</u> occur between adjacent words; it would be incorrect to write "XEQL 0" or "X EQLO."

V. STATEMENTS

So far, we have learned only the building blocks of the language. Now we are ready to write whole statements, which are imperative sentences telling the machine how to go about solving the problem.

Assignment (Replacement) Statements

The statement V = E where V is any variable and E any expression, means "substitute the value of E into the value of V."

```
     Examples:            Y = SIN(X)
                     A(I,J) = A(J,I)
                          I = I + 1
```

The last example says to set I equal to its former value plus one.

If one side of the equals sign is integer and the other is floating, the expression is converted to agree with the variable.

STOP statements

The statement "STOP" commands the computer to halt.

The statement "STOP E" (where E is an expression) commands the computer to halt displaying the value of E in Register A. Depressing "START" or "CONTINUOUS" on the 205 console will re-start the program where it left off.

Compound Statements

Any number of statements may be combined and treated as a single statement by separating them with semicolons and enclosing them in "statement parentheses" BEGIN and END. For example,

```
     BEGIN  X = X + 1; I = I + 1; W = 0 END
```

is a compound statement formed from three assignment statements.

Labeled Statements

A name or "label" can be attached to a statement, and that statement can be referenced by that label anywhere else in the program. If Σ is a statement, and L is an identifier, the label L is attached to Σ by writing "L..Σ". The two dots represent a colon, tipped on its side. An identifier used as a label cannot be used as a variable name.

Control Statements

The order in which statements are normally executed is the sequential order in which they are written. But there are two statements which can be used to change this normal order.

The statement "GO L" or "GO TO L" (where L is a statement label) is an instruction to start executing statements beginning with the statement labelled L.

The statement

```
SWITCH E, (L1,L2,...,Ln)
```

(where E is an expression, preferably fixed point, and the L's are statement labels) is interpreted as follows: If E is floating point, it is first changed to an integer by throwing away the fractional part. (As with subscripts, it is best to use integer arithmetic here.)

```
If  E = 0,   nothing happens.
If  E = 1,   action is like GO TO L1.
If  E = 2,   action is like GO TO L2.
      .                  .
      .                  .
If  E = n,   action is like GO TO Ln.
```

If E is negative or bigger than n, anything might happen, and usually does. The number of labels, n, is virtually unlimited.

Conditional Statements

If Σ is any statement, and P is any proposition, the following is a conditional statement:

IF P; Σ

It means "if P is a true proposition, execute statement Σ; otherwise, do not execute Σ." For example,

```
IF  X  EQL  0 ;   GO TO START
```

is a conditional statement which transfers control to START if X equals zero.

The reader may think at first that it is quite restrictive that Σ be only a single statement; but, of course, Σ could be a compound statement, such as

```
IF X EQL 0; BEGIN I = I + 1; GO TO START END.
```

Another form of conditional statement is the "UNTIL" statement. If P is any proposition and Σ is any statement, an UNTIL statement can be formed by writing

```
UNTIL P; Σ
```

which means "execute statement Σ until P becomes true."

For example,

```
UNTIL PCS(1); BEGIN I = I + 1; STOP I END
```

causes the computer to stop if the Skip Switch is not on. The value of a number I is displayed in the A Register. If the operator pushes START but fails to turn the Skip Switch on, the 205 will stop again, displaying a number one higher than last time. This process keeps on until the operator quits monkeying around and finally turns the Skip Switch on. If the Switch was on at the very beginning, however, no stop would occur. Thus, if I was initially zero, the value of I after this UNTIL statement is passed is precisely the number of times the 205 stopped.

It should now be clear that the statement

```
UNTIL P; Σ
```

is precisely equivalent to the statement

```
L..IF NOT (P); BEGIN Σ; GO TO L END
```

Example Program

Statements (and Declarations, which we will discuss later) are separated from each other by semicolons. To illustrate the statements we have discussed so far, let us write a program to solve a simple problem.

The problem? We are given 100 numbers X(1), X(2), ..., X(100) where X is the name of a vector variable. We would like to find the sum of

$$X(1) + \frac{1}{2} X(2) + \frac{1}{3} X(3) + \ldots + \frac{1}{100} X(100);$$

after that sum is calculated, the computer should halt displaying the answer. (The INTEGER and ARRAY declarations used in the following program should have an obvious meaning even though we haven't defined them as yet.)

Burroughs Algebraic Compiler Language	Remarks[1]
INTEGER I;	This designates I as an integer variable.

Burroughs Algebraic Compiler Language	Remarks[1]
ARRAY X(100);	X is a vector with 100 elements
I = 1; SUM = 0;	Initialize I and SUM
ADD..SUM=SUM+X(I)/I;	Add I-th term to SUM
IF I=100; GO TO HALT;	Are we finished?
I = I + 1;	Increment I
GO TO ADD;	Loop back to ADD statement
HALT..STOP SUM;	Display answer
FINISH;	Signifies end of program.

Another way is as follows:

```
INTEGER I; ARRAY X(100); I=1; SUM=0;
UNTIL I GTR 100; BEGIN SUM=SUM+X(I)/I; I=I+1 END;
STOP SUM; FINISH;
```

FOR Statements

Repetitive operations like the above occur so commonly, a special statement has been included to accommodate them. With a FOR statement, the above program can be shortened to the following:

```
INTEGER I; ARRAY X(100); SUM=0;
FOR I=(1,1,100); SUM=SUM+X(I)/I;
STOP SUM; FINISH;
```

In order words, if Σ is a statement, "FOR I=(1,1,100);Σ" is a statement which causes Σ to be executed repeatedly for the successive values of I = 1,2,3,...,100. There are several forms of FOR statements:

a. FOR V = E1,E2,...,En; Σ (where V is a <u>simple</u> variable, Σ is a statement, and the E1, E2, to En are expressions) means to execute Σ repeatedly, giving V the successive values E1,E2,...,En. At the end of the process, V = En.

b. FOR V = (E1,E2,E3); Σ (where V is a <u>simple</u> variable, Σ is a statement, E1 and E3 are any expressions, and E2 is an expression which does not begin with the character "-", and where E2 has a positive value) means

1. These remarks are not part of the compiler language, they are just explanations to the reader.

to execute Σ repeatedly, giving V the successive values E1, E1+E2, E1+2E2, etc. until E1+nE2 is greater than E3. At the end of the process, V will be greater than E3. If E1 is greater than E3, Σ will never be executed at all.

c. FOR V = (E1,-E2,E3); Σ (where V is a simple variable, Σ is a statement, E1 and E3 are any expressions, and E2 is an expression which has positive value) means to execute Σ repeatedly, giving V the successive values E1, E1-E2, E1-2E2, etc. until E1-nE2 is less than E3. At the end of the process, V will be less than E3. If E1 is less than E3, Σ will never be executed at all.

d. The three forms above can be combined. For example, consider the statement

```
FOR I = 2, (3,2,7),(23,-6,11),13,19; Σ.
```

It causes Σ to be executed for I equal to 2,3,5,7,23, 17,11,13,19 in that order.

It is illegal for the statement Σ to contain any labels which are referred to from outside of Σ. If Σ has the form IF P; Σ or UNTIL P; Σ it must be enclosed in a BEGIN - END pair. For example:

```
FOR I = 2, (3,4,12); BEGIN IF V(I) EQL 0; GO TO A END.
```

Other Statements

The ENTER, RETURN, and Procedure statements are discussed elsewhere in this manual.

VI. DECLARATIONS

Statements are instructions to the computer when the problem is being run; declarations, by way of contrast, are instructions to the compiler program when the program is being translated into machine language.

The first declarations will be described by examples.

INTEGER Declarations

INTEGER I, J..., W, ZETA

This declaration tells the compiler that the variables I, W, and ZETA are integer variables, and that all simple variables beginning with the letter J are to be integer variables.

Integer declarations must occur before the variables are used. I, W, and ZETA might not be simple variables; they may be arrays (in which case the entire array is an integer), or any of them may even be the name of a procedure used as a function (see Appendix C).

J in the above declaration is called a prefix. There are several restrictions on the use of prefixes:

a. A prefix must be only one letter in length.

b. When a prefix is used, no array variables beginning with that letter may be used, unless they are declared to be integer also.

c. Although normally the same identifier cannot be used for two different things, it is possible to use the identifier (the single letter) of a prefix as the identifier of a simple variable. But, it cannot be used also as a label. Therefore, in the above example, "J" could not be used as a label elsewhere in the program.

Array Declarations

ARRAY ZETA(2,7), X(100), Y(40,40)

This declaration tells the compiler that ZETA is a matrix with two rows and seven columns, that X is a 100-element vector, and that Y is a 40x40 matrix.

If an array is to be an integer array, the INTEGER declaration must precede the ARRAY declaration.

Subroutines

A block of statements may be set apart as a subroutine and given a label. This is a handy way to execute a series of instructions and to return afterwards to various places.

A Subroutine Declaration is written as follows:

```
      SUBROUTINE L; BEGIN Σ1; Σ2; ...; Σn END
```

where L is the label of the subroutine and $\Sigma 1$, $\Sigma 2$, etc. are statements (not declarations!).

At least one of the statements must be the statement "RETURN." Here is what happens: To use the subroutine, you give the statement "ENTER L" and this signals the program to start executing the subroutine beginning at $\Sigma 1$. As soon as a "RETURN" statement is encountered, control of the program returns to the statement following the "ENTER L" statement.

Note that the Subroutine Declaration itself does not cause $\Sigma 1$, $\Sigma 2$, ..., Σn to be executed; they are all by-passed at the place where the Subroutine Declaration itself appears. The declaration merely serves to associate the label L with a certain block of statements, and the subroutine is executed only when it is entered through an "ENTER" statement.

It is illegal to GO TO a labeled statement within the Subroutine Declaration from outside the Subroutine Declaration.

Comment Declarations

```
      COMMENT THIS IS A COMMENT
```

tells the compiler to ignore the string of symbols THIS IS A COMMENT because they are just explanatory remarks for the reader. The string of comment symbols can be chosen in any way from among the basic symbols of the Algol alphabet, except they may not contain a semicolon or begin with two periods.

Finish Declarations

At the end of every program in Algol language, the declaration "FINISH;" must be placed, telling the compiler that there is nothing more to the program.

Other Declarations

The INPUT, OUTPUT, FORMAT, and PROCEDURE declarations are discussed elsewhere in this manual.

VII. INPUT-OUTPUT

The preceding language description is sufficient to do most computer problems, except there is still no way to read data in and to punch answers out. This section describes the flexible input-output operations available.

Since there are two versions of the compiler, one for paper tape input-output and one for punched card input-output, data handling for the two versions is somewhat different. But, actually, both versions have many features in common, and the main differences are in the actual preparation of input data. The latter problem is discussed in Appendix D.

Input Declarations

An input declaration serves to give a name to a group of variables. For example,

```
INPUT  DATA1(X,Y,I,A(1),B(I+1))
```

associates the name DATA1 with the list of five variables X, Y, I, A(1), and B(I+1).

That is all an input declaration does. Like any other declaration, it is not an instruction to the computer to read in the list of variables; it merely means that the identifier DATA1 is to refer to these five variables.

Read Statements

Actual reading of data is accomplished via READ or PTREAD statements. These are special cases of Procedure Statements, the subject of Appendix C, but these input-output procedures can be used without knowledge of the full generality available in other Procedures.

The statement "READ(;;DATA1)" means "read in data into the variables specified on the INPUT Declaration for DATA1." As an example, suppose DATA1 was the list of five variables in the illustration above. Then, the READ statement means to put the first item of data into X, the second item into Y, the third into I, the fourth into A(1), and the fifth into B(I+1). Notice that the subscript in B(I+1) uses the new value of I which has just been read in.

READ is used for card input only. To get paper tape input, the statement "PTREAD(;;L)" where L is the label of an INPUT declaration, is completely analogous to "READ(;;L)." Notice the semicolons in the midst of these statements. Formats for preparing data acceptable to READ and PTREAD are described in Appendix D.

Output Declarations

An output declaration serves to give a name to a group of expressions. It is analogous to an input declaration, except arbitrary expressions and not only variables are allowed. For example,

```
        OUTPUT RESULTS(W,A(1), A(2)+A(I), ABS(4X))
```

associates the name RESULTS with the list of four expressions W, A(1), A(2)+A(I), and ABS(4X).

Paper-Tape Write Statements

The statement "PTWRITE(;;L)" where L is the label of an OUTPUT declaration, causes the values of that list of expressions to be punched onto paper tape, in the same format as used for paper tape input (except for the control word and blank tape between records).[1] PTWRITE is available only on the non-Cardatron version of the compiler.

Format Declarations

Output which is written without PTWRITE is displayed according to formats specified by the programmer. These formats are given as a string of characters in a FORMAT declaration. For example, the declaration

```
                FORMAT LINE1(5I15,P)
```

states that LINE1 is the name of the format string "5I15,P." The meaning of the latter string is "five integer numbers in fields 15 characters wide, punched on a card."

There are four types of field specifications allowed in a Format declaration:

Type	Meaning
Bn	A field of n blank columns.
In	A field of n columns, with an integer value right-justified in that field with leading zeroes suppressed.

1. To punch a control word on paper tape, use the statement "PTWRITE(;;CONTROL)." This causes a control word and 18 inches of blank tape to be punched. This statement must be deleted if the problem statement is to be processed by the Burroughs Algebraic Compiler for the 220 since paper tape control words are not used on that machine.

Type	Meaning
Fn.m	A field of n columns, with a floating point value truncated to m significant figures. Here m must be between 1 and 8. The format is as follows (shown for m equal to 8): $\pm$.kkkkkkkk,$\pm$ee Here 14 columns are used, and the meaning is the number $\pm$.kkkkkkkk times $10^{\pm ee}$. The number n must be at least m+6, in order to leave space for the signs, decimal point, comma, and two-digit exponent. If n is larger than m+6, extra spaces will be added at the left.
Xn.m	A field of n columns, with a floating point value truncated to m decimal places.

The programmer is responsible for seeing that I is used for integer expressions only, and that F and X are used for floating point expressions only.

The letters I, F, X may be preceded by an integer, which means the same thing repeated that many times. Otherwise, field specifications are separated by commas. For example,

```
2I11     is the same as  I11,I11
4F20.8   is the same as F20.8,F20.8,F20.8,F20.8
```

Alphanumeric information for titles and headings is indicated by "* alphanumeric string *", where the alphanumeric string may include any character except an asterisk, including spaces. If there are n characters between the asterisks, it represents a field n columns wide.

The kind of output desired is designated by using one of the letters P, T, or W.

P means punch the preceding fields on a card.

Tn means execute n carriage returns and type the preceding fields on the Flexowriter.

Wn means print the preceding fields on the 407, and activate the "t-relays" according to digit n.[1] (See page 22 for footnote.)

Each format string must end with either a P, T, or W specification, and only one occurrence of P, T, or W is allowed per format. In other words, one of these three specifications is used to terminate each format string. P, T OR W CAN NOT BE USED TO PROCESS MORE THAN 100, 80, OR 120 CHARACTERS, RESPECTIVELY, PER LINE OF OUTPUT.

As examples of formats, let us consider three particular format strings:

a. (*THE ANSWERS ARE*, 2I6,F15.6,T2)
Suppose this string is used with the three answers 1230 (integer), 323 (integer), and 3.1415927. What happens? The Flexowriter will do two carriage returns and will type out

```
        THE ANSWERS ARE  1230    323     .314159, 01
```

with the exact spacing as shown.

b. (B10,2X10.6,P)
Suppose this string is used in conjunction with the two floating point values 323.0 and 3.1415927. What happens here? A card is punched saying

```
 323.000000  3.141592
10-        20-        30-        40-
```

c. (W1)
This simple string would be used without any numerical answers, and it would simply print a line of nothing on the 407, then would skip the paper to the beginning of the next page.

WRITE Statements

The statement

```
WRITE(;;L1,L2)
```

where L1 is the label of an output list and L2 is the label of a format string, causes the values of the expressions L1 to be written according to the format L2. The number of expressions in L1 must be equal to the number called for in the format string.

---

1. The actual action of t-relays varies with board wiring. The conventional action is given in Technical Bulletin #17.
   - n=0, single space before printing
   - n=1, single space before, skip to next page after printing
   - n=2, single space before and extra space after printing
   - n=3, skip to next page before printing
   - n=4, double space before printing
   - n=5, skip to channel 2 punch in carriage tape before printing
   - n=6, double space before printing, extra space after
   - n=7, skip to channel 3 punch in carriage tape before printing.

The statement

```
WRITE(;;L2)
```

where L2 is the label of a format string, causes the format string to be printed only. This is for printing alphanumeric title lines.

WRITE may be used in the paper tape version of the compiler, but, of course, the operation "T" is the only operation of the set "P, T and W" which can be used in format strings.

VIII. OPERATING THE PROGRAM (A Recipe)

The statements are punched onto either cards or paper tape as described in Appendix D. Then the program is compiled as follows:

<u>Cardatron Version</u>

1. Press General Clear, Input Setup buttons on Cardatron Control Unit, push clear on Input Unit control panel. Put Algol deck, followed by your statements, followed by the library deck, in hopper of input unit and press Start on that unit. Ready the punch hopper with blank cards. If an on-line 407 is available besides the punch unit, set it up with a standard 120-120 board (Technical Bulletin #17). If you want to get a listing of the instructions of your program as they are compiled, turn the Skip Switch off; otherwise, put it on to skip this extra printout.

<u>Paper-Tape Version</u>

1. Put Algol tape into photoreader. Set input switch to Optical Reader and hit Clear, Continuous. After the tape has run in, the program will stop. Put your input tape into the photoreader. (At this point, if you want merely a Flexowriter type-out of your problem statements, manually transfer control with a CUB 3765; see Appendix D.) If you want to get a Flexowriter listing of your compiled program rather than a punched paper tape, set the Output switch to Page; otherwise, set it to Tape and ready the high-speed punch unit.

2. Push Continuous on the 205 console and stand back. There are several ways for the computer to stop as it goes through your program, and only one of them means good news. The halts are characterized by the display in the Address lights on the console, which can be read like the spots on dice or dominoes.

Stop #1:	This is the normal stop in Step #1 above after the Algol tape has just been read in. Don't panic at this point.
Stop #2:	If in Card mode, you had a bad card read or a keypunching error. If in Paper-Tape mode, a tape preparation error has given an invalid two-digit alphanumeric code.
Stop #3:	A paper-tape record is more than sixty words long. You have overlaid part of the compiler program and there's no hope for you.

Stop #4: Oh-oh. Your input program doesn't make sense, and the compiler is now mixed up. Turn the Output Switch to Page and hit Continuous; the Flexowriter will type out something which is an attempt to describe your error to you. The type-out has the following form:

$\pm$nnnnnnnnnn nnnnnnnnnn.kkkk (alphabetic information)

Here the n's are of no concern to you, but if any communication is directed to Burroughs questioning some error stop, they should be reported.

The alphabetic information is the last 20 characters of your input language which the compiler has analyzed. This indicates approximately where the compiler got confused about your statements. These 20 characters are not exactly in the form you wrote them. If several spaces in a row were encountered, only a single space will show here. And since the Flexowriter is incapable of typing all the special characters, a "spaceLspace" is substituted for each left parenthesis, and "spaceRspace" for each right parenthesis, and "spaceZspace" for each equal sign; and a percent sign % is substituted for an asterisk.

The digits kkkk may indicate what kind of error occurred, but this is not guaranteed. Sometimes the compiler gets so mixed up it doesn't even know why it is confused. A table of the meaning of each kkkk error appears in Appendix E.

Stop #5: Congratulations! Your program is compiled, and no errors have been detected in your grammar. Go to Step #3.

If the 205 does not stop for one of the above reasons, your program is most likely so wild that there is no standard thing to do next. Note the contents of A, R, B and C and then try transferring control manually to location 2973 and proceed as in Stop #4.

Cardatron Version

3. Depress Continuous once more and the library deck will run in and will be added to your program.

Paper-Tape Version

3. Put the library routines tape in the photoreader and hit Continuous; the subroutines will now be

Cardatron Version	Paper-Tape Version
When the deck is all in, the punch hopper contains a self-loading deck which will work your program (when followed by the data cards, if any).	added to your program. After this is done, the punched output tape is a self-loading program.

4. At this point, you can once more hit Continuous (after making sure the output selector switch is set to page) and you will get a type-out telling which locations of the 205 drum are not used by your output program.

5. When running your compiled program, the machine may stop again. These stops are either caused by a STOP statement in your program (identified by a wedge-shaped pattern in the address lights)[1] or

Stop #6: An error detected by a library subroutine. Set the Output Selector switch to Page and hit Continuous; an error type-out will occur in the form

```
+000000tttt±xxxxxxxxxx
```

and the machine will stop again, Stop #7. Here tttt is an error type, and ±xxxxxxxxxx is an input argument to the library function where the error was detected.

tttt	Meaning
0004:	Taking the square root of a negative number.
0005:	Taking the logarithm of zero or a negative number.
0013:	Attempting to change a floating-point value which is too large into an integer value.
2332:	EXP routine, with xxxxxxxxxx too large or too small.

Stop #7: This stop occurs immediately after Stop #6. At this point the operator should change the contents of register A to the value he wishes

1. 08 0137 xxxx is produced for STOP statements. 08 1359 xxxx may occur if the programmer improperly omits the RETURN statement required in the SUBROUTINE or PROCEDURE declarations.

as the result of the function which was in error. Upon hitting Continuous, this new value will be typed and the program will take off again.

Breakpoint Conventions

During the running program, there is a breakpoint digit of 1 on all CUB commands; of 2 at the end of each library routine; and of 4 at the end of each replacement, GO, or ENTER statement. Operators familiar with 205 machine language to a certain extent will be able to use these breakpoint digits when debugging.

"IF NOTHING ELSE WORKS, READ THE INSTRUCTIONS".

APPENDIX A

Reserved Words

The following words have appeared elsewhere in this manual, and they have special meaning to the compiler. They are not to be used for any other purpose than that described.

ABS	ENTER	IF	OR	SQRT
AND	EQL	INPUT	OUTPUT	STOP
ARCTAN	EXP	INTEGER	PCS	SUBROUTINE
ARRAY	FINISH	LEQ	PROCEDURE	SWITCH
BEGIN	FOR	LOG	PTREAD	TO
COMMENT	FORMAT	LSS	PTWRITE	UNTIL
CONTROL	GEQ	MOD	READ	WRITE
COS	GO	NEQ	RETURN	
END	GTR	NOT	SIN	

In addition, the following words are also to be reserved if the program is ever to be compiled using the Burroughs Algebraic Compiler for the 220.

ARCCOS	ENTIRE	FLOAT	MIN	ROMXX
ARCSIN	EQIV	FLOATING	MONITOR	SEGMENT
BOOLEAN	ERROR	FUNCTION	OTHERWISE	STATEMENT
COSH	EXTERNAL	IMPL	OVERLAY	SIGN
EITHER	FIX	MAX	REAL	SINH
				TAN
				TANH

APPENDIX B

Arithmetic in the 205

The arithmetic operations you get inside the 205 when using the compiler are in most cases what you would expect, but there are some pecularities.

Integer arithmetic: Overflow occurs after addition or subtraction develops an 11-digit number, or after a division by zero. After division, the remainder is lost, only the quotient is saved; after MOD, the quotient is lost, only the remainder (which is given the sign of the quotient) is saved. If a product of more than 10 digits is developed after integer multiplication, no warning is given, and only the low-order ten digits survive.[1]

Floating-point arithmetic: Overflow occurs if the answer becomes greater than 10^{49}, or occasionally for numbers nearly this size. Numbers in the range 10^{-50} or less are set to zero. The result of a floating point operation is always 'truncated' or 'chopped' not rounded, to eight significant figures. When taking floating point numbers to powers, using the * symbol, only positive numbers can be raised to a floating point power since the logarithm is taken.

Overflow: If overflow occurs, the 205 will stop and might be quite hard to re-start properly.

Conversion between floating point and integer arithmetic: When going from floating to integer, the floating number must be less than 90000000 in absolute value. When going the other way, any integer value is allowed, but nine- and ten-digit numbers are truncated to eight significant figures.

Library functions: Library functions are designed for speed of operation and for at least seven significant figures of accuracy; they are usually off by at most three in the eighth place, except when taking a floating point power where the combined LOG, EXP can build up worse errors.

---

1. This fact can be used to advantage when generating random numbers; at the beginning of your program, say INTEGER RANDOM; RANDOM=6250739481; then when a random number is desired, say RANDOM=9677214091 RANDOM; R=RANDOM/1**10 and R will be a floating point random number between 0 and 1.

APPENDIX C

Procedures

A procedure declaration is a method of defining arbitrary functions of any number of variables with any number of outputs. For example, we might have a lot of use for Bessel functions in a program; a Bessel function procedure could be written, or looked up in a book of Algol procedures, and this procedure could be incorporated in the program to easily compute Bessel functions.

There is a lot to explain about procedures and, perhaps, the best way to begin is to give some simple examples. Suppose the program we are writing includes a lot of references to the function F(X,Y) which is defined to be zero if X+Y is negative, the square root of X+Y otherwise. We would define this function as a procedure as follows:

```
BEGIN PROCEDURE F(X,Y); BEGIN
   IF X+Y LSS 0; BEGIN F()=0; RETURN END;
   F()=SQRT(X+Y); RETURN    END END
```

This procedure declaration is put at the beginning of the problem statement. The notation F()=SQRT(X+Y) means the value of the function is to be SQRT(X+Y). Then in the program proper which follows, we can write   X=F(Y,Z)+F(2K,Z).

As another example, we can define a procedure to multiply an MxN matrix A by an NxP matrix B, and get the answer as an MxP matric C. Such a procedure is defined thus:

```
BEGIN PROCEDURE MULT(M,N,P,A(,),B(,);C(,))   ; BEGIN
   INTEGER M,N,P,I,J,K;
   FOR I=(1,1,M); FOR J=(1,1,P); BEGIN
      C(I,J)=0;
      FOR K=(1,1,N); C(I,J)=C(I,J)+A(I,K)B(K,J) END;
   RETURN END END
```

(This procedure is certainly not the fastest nor best way to accomplish matrix multiplication, and the reader is reminded that this is merely an example of a procedure declaration, and it can be greatly improved as far as running speed is concerned; such improvements are just confusing at this stage of the exposition, so they haven't been mentioned here.)

Later in the program proper, the statement

```
MULT(N,N,N,A(,),A(,);B(,))
```

for example, set the NxN matrix B equal to the square of the NxN matrix A.

Now that some examples have been given, let us lay down the ground rules.

Rule #1: A Procedure Declaration has the form

BEGIN PROCEDURE NAME(Parameter list);BEGIN $\alpha1$;$\alpha2$; ...;αn END END

The parameter list will be explained later. $\alpha1$ through αn are statements or declarations; any declaration may be used here except a Procedure or Finish declaration.

Rule #2: Scope of names: All reserved words (see Appendix A) and all names of procedures defined in previous Procedure Declarations are "global," that is, they have the same meaning inside of the procedure declaration and out. The word "AND," for example, has its ordinary meaning inside of the procedure declaration, of course. But all other identifiers (for example, X) represent different entities in each procedure declaration in which they occur and still other entities in the program proper. In fact, the same identifier used as a simple variable in one procedure declaration may be used as an array in another and as a label outside of both. Identifiers, thus, are "bound" within procedures, and if someone else has written a procedure declaration you may copy it without worrying if they've used the same identifiers you have.

Rule #3: A RETURN statement must appear at least once in a Procedure Declaration; this has a similar meaning to its use inside of Subroutine Declarations. Note that Subroutine Declarations can actually be included inside of Procedure Declarations, and that a RETURN statement inside of one of those is a Subroutine-Return, not a Procedure-Return.

Rule #4: There are two distinct classes of procedures:

a. Procedures to be used as a function, such as our F(X,Y) defined earlier. Such procedures have one or more inputs, and only one output. In this case, if the function value is to be integer, it is specified on an INTEGER declaration; e.g., INTEGER NAME. The value of the function must then be given by NAME()=expression, just before a RETURN statement. The name of a procedure cannot be used inside of its own declaration in any other way.

b. Procedures to be used as whole statements. These procedures cannot be used later on in the program as parts of arithmetic expressions, as those of type a) are; they are always called as a separate "procedure statement." Examples of procedure statements are the READ and WRITE statements, and the MULT statement given above. Notice that MULT cannot be used as a function by its very nature, since it has MxP values, the elements of C(,), while a function has only one value.

Rule #5: The parameter list of a procedure to be used as a function contains no semicolons; the parameter list of a procedure to be called as a whole statement contains one or two semicolons. Thus, the number of semicolons tells the compiler what sort of procedure this is.

Rule #6: Three types of strings can be part of the parameter list:

a. Input parameters. Input parameters represent the values which are inputs to the procedure. These can be simple variables, like X and Y in F(X,Y) or like M,N, P in MULT, or they can be arrays written with blank subscripts, as V() a vector, or as the matrices A(,) and B(,) in MULT.

The very appearance of an array as an input parameter (or as an output parameter) defines that identifier to signify an array variable within the procedure declaration; an ARRAY declaration must not be used for input or output parameter arrays. Moreover, an array which is an input (or output) parameter has arbitrary size, depending upon which array is substituted for it when the procedure is called. The matrices A, B, and C in MULT can be 5x5 when called out once, 4x4 when used later, non-square at another use, and so on, depending upon the size of the matrices which are to be multiplied by MULT.

b. Output parameters. Output parameters represent the variables into which the results of the procedure are to be stored. Like input parameters, these can be simple variables or arrays. For example, C(,) is an output parameter in MULT.[1] (See page 33 for footnote.)

c. Label parameters. These are also inputs to the procedure; they represent labels of statements, of input, output, or format declarations, of subroutines, or of other procedures. The labels representing other procedures are written with parentheses, e.g., F(). Examples of the use of label parameters are the READ and WRITE procedures, and the SIMPS procedure to be given later.

Rule #7: Parameter lists of procedures can have only the following forms:

a. (input parameters) <u>Function-procedures</u> must use this form.

b. (input parameters;output parameters)

c. (input parameters; ; label parameters)

d. (input parameters;output parameters; label parameters)

e. (;output parameters)

f. (;;label parameters)

g. (;output parameters;label parameters)

The placing of semicolons is important in these forms.

Rule #8: Each procedure declaration must occur before any use of that procedure. (This rule is, of course, necessitated by our conventions about implied multiplication.)

Rule #9: When using ("calling") the procedure, the parameter list is of the same form as the parameter list of the declaration (see Rule #7). If the parameter list was of type a), the procedure must be called as a function; otherwise, it must comprise a whole statement. The parameters written when calling a procedure are to be written in the same order as those in the declaration which they are to replace, and they must be

---

1. Simple variables used as output parameters should <u>not</u> be used inside the procedure declaration anywhere except on the left-hand side of an equal sign, or as output parameters in another procedure callout. If use on the right-hand side of the equal sign is required, the name of a simple output variable must be enclosed in parentheses for correct operation; e.g., (S).

integer or floating point in correspondence with their use inside the procedure.

When calling a procedure, the parameters written are substituted for the corresponding things in the parameter list of the procedure declaration as the calculation is done, and the procedure is terminated when the RETURN is encountered.

a. Simple variables. In an input parameter list, any expression of the same type (integer or floating) may be substituted for a simple variable in the procedure declaration. F(2K,Z) in the above example, where 2K is substituted for the simple input variable X.

   In an output parameter list, however, only a simple variable standing alone may correspond to a simple variable in this declaration, because quantities are going to be stored in this location.

b. Array variables. In those positions corresponding to vector variables in the procedure declaration, the name of a vector array followed by () should be used when calling the procedure. In those positions corresponding to matrix variables in the declaration, use the name of a matrix variable followed by (,) when calling the procedure. This applies to both input and output parameters. See the example in MULT.

c. Labels. Non-procedure labels are designated by their name alone, as in READ. Names of procedures in a label parameter list are designated with the name followed by ().

As a final example of Procedures, here is a complete program which includes the definition of two procedures, the first one being a useful routine to accomplish numerical integration using Simpson's Rule.

```
BEGIN PROCEDURE SIMPS(A,B,EPSILON,BOUND;VALUE;F());  BEGIN
   COMMENT  A,B  ARE LIMITS OF INTEGRATION. EPSILON IS
   PERMISSIBLE DIFFERENCE BETWEEN TWO SUCCESSIVE SUMS.
   BOUND IS UPPER BOUND FOR ABS(F(X)) IN THE INTERVAL
   (A,B). VALUE IS THE ANSWER. F() IS THE FUNCTION TO BE
   INTEGRATED BY SIMPSONS RULE;
      TH=B-A;  IBAR=BOUND·TH;  N=1; J=0.5(F(A)+F(B))TH;
      LOOP.  H=0.5TH;  S=0;  FOR K=(A+H,TH,B); S=S+F(K);
      I=J+4H·S; IF ABS(I-IBAR)LEQ EPSILON; BEGIN VALUE=I/3;
```

```
          RETURN END;  IBAR=I;   J=0.25(I+J);   N=N+N;   TH=H;
          GO TO LOOP END END;
    BEGIN PROCEDURE DARCTAN(X); BEGIN DARCTAN()=1/(X*2+1);
    RETURN END END;
    COMMENT NOW THE PROGRAM PROPER FOLLOWS;
       SIMPS(0.0,1.0,1**-5,2.0;S;DARCTAN());
       WRITE(;;A,B);  STOP;
       OUTPUT A(S); FORMAT B(X11.8,T1); FINISH;
```

A study of this program reveals that the first procedure is a general-purpose routine to accomplish numerical integration, and the DARCTAN procedure is a function which evaluates $1/(x^2+1)$. The program proper then evaluates

$$\int_0^1 \frac{dx}{x^2+1}$$

The correct answer is $\pi/4$, and the value obtained was actually .7853981 after three iterations of "LOOP," correct to six places.

APPENDIX D

INPUT FORMATS

Cardatron Input: Compiler Statements

Algol language is punched onto cards as follows·

a. Column 1 contains a "2" punch.

b. Columns 2-72 contain the statements of the program. The rule here is that column 2 on one card always immediately follows column 72 of the previous card; however, since spaces may be freely inserted after a semicolon, common practice is to punch a single statement on each card, and then to leave the remainder of the card blank.

c. Columns 73-80 are ignored by the compiler, and they may be used for numbering the deck or something.

Cardatron Input: Data for READ Routine

This free format consists of stringing numbers on the cards sequentially, separated by one or more spaces, with no particular fields defined in advance for data. Column 1 must contain the digit 5, columns 77-80 must be blank, and the rest of the card contains from 1 to 38 data values.

Data which is to be stored into an integer-valued variable must be written as a number, without decimal point. Data which is to be stored into a floating-point-valued variable must be written either:

a. with a decimal point, not necessarily imbedded, e.g.,

```
3.1415927      .03      500.      -200.8
```

b. with a decimal point, followed by a comma and scale factor, e.g.,

```
3.12145,11    -105,-3
```

Notice that the output of the WRITE routine is in format acceptable to the READ routine (except for the 5-punch in column 1 which is easy to add by prefacing the format string with *5*).

The READ routine will read in as many cards as it needs to get values for the input list. For example, if the input list contains seven variables, and the first data card contains only four values, another card will be read. If this second card has more than the remaining three values, however, all the extra values will be lost.

Minus signs on these cards must be the "11-punch" minus signs, not the extra minus signs found on Fortran keypunches.

Paper-Tape Input: General

All paper-tape records must be sixty or less words in length, followed by the standard control word

7 0000 30 0000.

Between individual groups of words, at least 15 inches of blank tape should appear to allow for acceleration and deceleration of the optical reader.

Paper-Tape Input: Compiler Statements

Compiler language is translated into a numeric code by converting each character of its alphabet into a two-digit number. A table for this translation is listed at the end of this Appendix. Notice that even the digits 0 through 9 must be converted into a pair of digits according to this code.

Groups of ten digits (corresponding to five characters of the original language) form a word, and this word must be given a sign of zero. As an example of this tape preparation, here is the beginning of the first example program of part V, in the form suitable for paper tape input:

Numeric Code	Algol Correspondent
0 4955634547	INTEG
0 4559004913	ER I;
0 4159594168	ARRAY
0 0067248180	X(10
0 8004134933	0);I=
0 8113626454	I;SUM
etc.	etc.
0 4649554962	FINIS
0 4813000000	H;
7 0000300000	control word

Paper-Tape Input: Data for PTREAD Routine

The sign digit should be 0 for positive values, 1 for negative values. Data which is to be stored in an integer variable must be written as a ten-digit number (with leading zeroes added if necessary). Data for floating point variables must be in the 205 floating point format, which is:

a. Zero is written 0000000000

b. Non-zero values are written in the form

yyxxxxxxxx

meaning the number .xxxxxxxx times 10^{yy-50}. For example,

1.0	is written	5110000000
14.7	is written	5214700000
2.84×10^{-6}	is written	4528400000

Minus zero (namely 1 0000000000) may <u>never</u> be used as a data value for PTREAD!

Up to 60 items of data can be used on any paper-tape record, and they need not all be used in one PTREAD statement; any values not used in one PTREAD will be used on the next PTREAD. On the other hand, a single PTREAD may call for several tape records, if the input list has a lot of variables in it. (This operation differs from the Cardatron READ, which always begins by reading a fresh card, ignoring any possible unused values from the previous cycle.)

<u>Table of Two-Digit Codes for Paper Tape</u>

A	41	J	51		
B	42	K	52	S	62
C	43	L	53	T	63
D	44	M	54	U	64
E	45	N	55	V	65
F	46	O	56	W	66
G	47	P	57	X	67
H	48	Q	58	Y	68
I	49	R	59	Z	69

0	80
1	81
2	82
3	83
4	84
5	85
6	86
7	87
8	88
9	99

=	33	,	23	;	13	.	03	-	20
		(	24	*	14	)	04	/	21
space	00	+	10						

The grouping shown here is so presented to facilitate memorizing the code.

APPENDIX E

Compiler Error Code Numbers

.7061	Symbol table full; too many identifiers used.
.7084	Name table full; too many long identifiers used.
.0987	(If FINISH$ was last sensed) Program exceeds 4000 locations. (If FINISH$ was not last sensed) SIN, SQRT, etc. of an integer quantity.
.0989	Unmatched right parenthesis or END.
.0998	Name of array, or name of library function or procedure, not followed by a left parenthesis.
.0999	Wrong number of subscripts on an array variable.
.1000	(If FINISH$ was last sensed) Unmatched left parenthesis, or missing right operand, possibly goes back far into the program where the error actually occurred. (If FINISH$ was not last sensed) Improper Proposition.
.1001	Improper parameter list in Procedure Declaration.
.1002	Improper statement label.
.1005	Improper identifier used as a label.
.2178	Improper identifier used as a label.
.7078	Compiler mixed up trying to define a label.

APPENDIX E

Compiler Error Code Numbers

.7061	Symbol table full; too many identifiers used.
.7084	Name table full; too many long identifiers used.
.0987	(If FINISH$ was last sensed) Program exceeds 4000 locations. (If FINISH$ was not last sensed) SIN, SQRT, etc. of an integer quantity.
.0989	Unmatched right parenthesis or END.
.0998	Name of array, or name of library function or procedure, not followed by a left parenthesis.
.0999	Wrong number of subscripts on an array variable.
.1000	(If FINISH$ was last sensed) Unmatched left parenthesis, or missing right operand, possibly goes back far into the program where the error actually occurred. (If FINISH$ was not last sensed) Improper Proposition.
.1001	Improper parameter list in Procedure Declaration.
.1002	Improper statement label.
.1005	Improper identifier used as a label.
.2178	Improper identifier used as a label.
.7078	Compiler mixed up trying to define a label.

APPENDIX F

Computer Equipment Configuration Requirements

This compiler was written for a 205 with automatic floating point unit and with a flexowriter modified for Cardatron code compatability. In addition, the machine must have either a paper-tape input-output system or a Cardatron system. The Cardatron system should preferably have two output stations, but a single output unit is sufficient.

The standard version of the program has the following unit designations on its Cardatron instructions:

Input unit, #1
Output punch, #2
Output printer, #3

These can be changed if desired, by referring to the following table of all locations which refer to the Cardatron units:

3692	unit 1	object program load routine
3693	unit 1	object program load routine
3698	unit 1	object program load routine
3745	unit 2	load format band 2
3749	unit 2	punch first card
3750	unit 1	load format band 1
3751	unit 3	load format band 4
3752	unit 3	skip to top of page
3753	unit 3	load format band 3
3758	unit 2	punch second card
3759	unit 2	load format band 2
3975	unit 3	write compiler instruction
3985	unit 1	read compiler statements
3986	unit 3	write card just read
0044	unit 2	punch object program

When modifications are made to unit designations, the library function decks will have to be modified as well, and a trial-and-error procedure is suggested for accomplishing this.

APPENDIX G

Meaning of Compiler Output

Here are some brief hints for those who would like to examine the 205 coding which the compiler produces in detail. What is not spelled out here can be observed by making trial runs feeding various weird statements to the compiler program.

Each output program begins in location 0071. Since compilation proceeds in one pass, the instructions turned out may not have the final form which they will have at running time. For example, when compiling the sample program of part V, when the phrase "GO TO HALT" is encountered Algol has no idea how far ahead the statement "HALT" might be, so it can't turn out the actual address of HALT at that time. Therefore, it puts out a coded instruction and the loading routine later fixes everything up.

Table entries for this fixing-up procedure start in location 3999 and work downwards. These table entries are in the form

xx yyyy zzzz.

xx is ignored. The actual meaning is something like "change the address part of the instruction in location zzzz to yyyy. If the address which was just changed was zero, that's all; otherwise, the former value of that address refers back to another instruction which is also supposed to be changed to yyyy. This looking-back process is continued until all references to yyyy have been fixed up, signified by a zero in the last address to be changed."

For example, from the coding

```
0100      0 0001 30 0000
0101      0 0000 02 7101
0102      0 0004 12 6015
0103      0 0000 20 0100
0104      0 0000 72 7106
0105      0 0001 30 0103
0106      0 0001 30 0105
3999      0 0001 07 0106
```

the loading routine would produce

```
0100      0 0001 30 0107
0101      0 0000 02 7101
0102      0 0004 12 6015
0103      0 0000 20 0107
```

```
0104      0 0000 72 7106
0105      0 0000 30 0107
0106      0 0001 30 0107.
```

Locations 0000-0070 are reserved for special use. Locations 4980-5019 are used for temporary storage; locations 6000-6019 for simple variable; and locations starting down from 3999 are used for array variables, simple variables which didn't fit into loop six, and library routines such as WRITE, SQRT, etc.