ESD-TR-67-430

MTR-516

# USING THE READILY AVAILABLE ALGEBRAIC LANGUAGE AS A COMPILER ENVIRONMENT

APRIL 1968

G. P. Steil, Jr.

Prepared for

DEPUTY FOR COMMAND SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 512B
Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract AF19(628)-5165

AD0669092

ESD-TR-67-430 MTR-516

# USING THE READILY AVAILABLE ALGEBRAIC LANGUAGE AS A COMPILER ENVIRONMENT

APRIL 1968

G. P. Steil, Jr.

Prepared for

DEPUTY FOR COMMAND SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 512B
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract AF19(628)-5165

## FOREWORD

This report describes the use of an algebraic language in constructing a simple compiler. It was written by The MITRE Corporation, Bedford, Massachusetts, in partial fulfillment of Project 512B under contract number AF 19(628)-5165.

## ABSTRACT

The use of algebraic command languages for things other than preparing numerical algorithms has become somewhat popular, in particular for writing compilers. The author feels that the technique of using an algebraic command language for implementing a compiler is a good solid practical idea deserving some additional attention. He feels that this technique will be found particularly useful by organizations not in the business of building commercial compilers, but interested in the implementation of a small special-purpose language, such as a query language for a model of a command system. The purpose of this paper is to describe this technique to such an audience and to comment on the extent of its applicability.

## TABLE OF CONTENTS

## LIST OF FIGURES

## SECTION I

## INTRODUCTION

Six years ago, Robert W. Floyd remarked in his article "A Descriptive Language for Symbol Manipulation:" "The algebraic command languages (ALGOL, IT, FORTRAN, UNICODE), although useful in preparing numerical algorithms, have not in the author's opinion proven themselves useful for symbol manipulation algorithms, particularly compilers."[1] Robert W. Floyd would probably not say that today, for the use of algebraic command languages for things other than preparing numerical algorithms has become somewhat popular, in particular for writing compilers. For example, the Burroughs Corporation has programmed large portions of the software for the B5000 and B8500 systems using a dialect of ALGOL, MITRE has programmed FORSIM IV in FORTRAN IV, and according to the recruitment ads in Datamation (at the date of this writing), United Airlines intends to program most of its passenger reservation system in Univac's FORTRAN V. The author feels that the technique of using an algebraic command language for implementing a compiler is a good solid practical idea deserving some additional attention. He feels that this technique will be found particularly useful by organizations not in the business of building commercial compilers, but interested in the implementation of a small special-purpose language, such as a query language for a model of a command system. The purpose of this paper is to describe this technique to such an audience and to comment on the extent of its applicability.

Using a tool for a purpose other than the one for which it was intended requires an explanation. Usually the substitution of one tool for another takes place when the proper tool is not available, and its fabrication is a non-trivial process. In our case, the tool is a compiler builder.

To say that compiler builders do not exist or are not available would not be quite accurate, for work on such tools and the theory behind them has been going on at a large number of places for quite some time. This work has not been without significant results: certain classes of ALGOL-like (phase structure) languages have been identified as being unambiguous; algorithms have been devised which will accept a definition of the syntax of a language and determine if the language is in such a class; and algorithms have been devised for these classes of language that mechanically generate other algorithms capable of scanning source text and producing representations of the source text syntax in a variety of formats. Around these basic principles have been built a variety of systems for programming compilers that are indeed compiler builders or "compiler compilers." But at the present time these systems are by no means as readily available as conventional software, in particular the algebraic languages. Furthermore, despite the progress that has been made, the advent of the instant compiler is still quite a way off. A large amount of the work of implementing a compiler is still conventional programming and debugging.

This paper suggests that in the absence of a readily available compiler building tool that the available algebraic language be adapted to such a purpose, as has been done already at several places. It sketches an approach to such an adaptation, and it shows that while this approach has the disadvantage of being relatively informal (in the sense of taking strict advantage of the available theory), it has the advantage that a compiler constructed as we are about to describe can be maintained and modified by programmers with no special training or experience. It also shows how this scheme lends itself readily to what we have come to call "creative adhocery" and has the audacity to suggest that such an adaptation of an algebraic language has a usefulness as great as many of the fancier compiler compilers of recent derivation.

## GENERAL APPROACH

Our general approach in making a compiler building tool out of an algebraic language is to augment the algebraic language and its operating environment with a small set of tightly coded machine language subroutines. These machine language subroutines provide the data manipulating capabilities that are necessary for the construction of a simple compiler and are not found in the algebraic language. Communication between the algebraic language and the machine language subroutines is facilitated by defining the interface between them entirely in terms of integers. That is, every symbolic datum that need be manipulated by a program in the algebraic language is represented as an integer. The extra machine language programs are not, however, a general-purpose symbol manipulation capability. Such a task could become quite complex, and that would subvert our basic goal of providing an economical base for constructing a simple compiler.

A compiler to be implemented in this environment is structured as follows: Each syntactic element in the source language is represented by a procedure in the algebraic language, which is entirely responsible for the translation of an instance of that syntactic element into some object language. In performing its job of translation, this procedure may appeal to other procedures representing syntactic elements;

procedures which build and manipulate symbol tables, procedures which generate code, and so forth. Thus at the highest level there is a procedure named PROGRAM which, when appealed to, compiles an entire program. At a lower level a procedure named BOOLEAN is responsible for translating a Boolean expression and returning the translation as its value. This way of structuring translation, "top-to-bottom" analysis, is frequently employed in syntax-directed compilers, where the rules of syntax are stored explicitly in a specially formatted table. The difference here is that each syntactic element, rather than being represented in a table, is represented by a procedure. The advantage of doing it this way can be seen by studying the design of syntax directed translators employing the table method. Although the syntax definition is usually neatly packaged, the algorithm which generates the object code (or the output to the next stage of translation) is usually specified by a series of "actions" attached to the syntax definition which are as ad hoc as the syntax definition is general. <u>But the algorithm which generates the object code is the non-trivial part of translation</u>. Representing syntax by procedure structure allows the translation algorithms to be expressed in a general-purpose programming language rather than in a set of ad hoc "actions." The formal structuring of a compiler in a way that is rigidly tied to syntax is the contribution of the syntax directed compiler. But it is the author's experience that the dream of feeding a language specification as data to a general-purpose program can lead the design of a compiler astray.

Looking through the literature the author notes that both Lietzke[2] and Irwin[3] have suggested basically what is suggested here. The difference is that we are pushing the idea further. Lietzke was interested in employing a set of procedures that paralleled language syntax only for the purpose of diagnostics (in the Share ALGOL compiler), and Irwin only in generating random sentences of a grammar; we are suggesting that nearly the whole compilation job can be done in such an environment.

In constructing the examples in the sections that follow we have assumed that the algebraic language which is to be used as a simple compiler environment is ALGOL. Hereafter the term "ALGOL procedure" will be used interchangeably with "algebraic language procedure."

## SECTION II

## UTILITY PROCEDURES

The utility procedures used to augment the normal ALGOL operating environment are described in this section. They include an editor for segmenting a source string, two elementary list operators, two procedures for composing an object string, and one procedure for generating identifiers. Unless otherwise specified, it is intended that they be programmed in machine language.

### *editor*

As any compiler writer will tell you, a large percentage of compilation time in a typical compiler is spent in the simple process of examining the source string one character at a time, breaking it into "atoms," and reducing each atom to its internal representation. This aspect of compilation is usually separated off as a tightly coded subroutine, and no exception is made here. The machine language subroutine which we postulate to do this job is known as the *editor*. Its input arguments could include such things as the source or location of a string in the source language, a list of characters which are to serve as separators, and other information regarding the editing of the source text into atoms. But for our purposes here we will assume the *editor* itself bears the full responsibility for knowing or finding out these details, and consequently has no input arguments. Its output value is an integer which may be thought of as a pointer to the first atom of the edited source string. We shall see how to use this pointer shortly.

For the purpose of our examples below we will assume that the *editor* segments source strings according to the following rules:

(1) All characters except *A*,*B*,*C*,...,*Z*,0,1,2,...9 are *separators*.

(2) All separators except blank (*b*) are *atoms*.

(3) Any sequence of non-separators between two separators is an *atom*.

Example: The *editor* segments the source string

*A*+*T*(*ANQ*;7)-*F*;*IFbbbANY*

into the atoms

*A* + *T* ( *ANQ* ; 7 ) - *F* ; *IF* *ANY* nextatom and test

Two machine language subroutines are used to scan along an edited source string.

nextatom fetches the next atom in a source string. nextatom has one argument, an integer serving as a pointer to an atom of an edited string. nextatom returns as an output value, an integer serving as a pointer to the next atom in the edited string. nextatom returns a special terminal integer, say $\Delta$, if there is no next atom.

Example: Assume the source string

ANS + 47*B

is ready at an input source when the following segment of code is executed:

```
integer X;

X : = nextatom (nextatom(editor))
```

The contents of the variable X will be replaced by an integer serving as a pointer to the atom 47.

The phrase "integer serving as a pointer" occurs so frequently that we will use in its place simply the word "pointer." Thus we will say "the contents of the variable X will be replaced by a pointer to the atom 47."

test is a logical valued procedure used to compare an arbitrary string to an atom in an edited source string, and on the basis of the comparison advance a pointer pointing to the edited source string. Thus test requires two arguments: a string and a pointer; and test returns two values: true or false, and a pointer to the next atom in the source string when the comparison yields true. To make the use of the function test as natural as possible, the following conventions have been adopted regarding the input arguments to test and the values it returns. test is defined to have one explicit argument, the string to be compared; and one implicit argument, the name of a variable containing a pointer to an atom in an edited source string. The implicit argument is always the name of the global variable input (where it is not possible for external machine language subroutines to reference variables in the algebraic language programs, the procedure test can be redefined to have two explicit arguments). If the value of the comparison is true, test returns the value true and replaces the pointer in the variable named by the second argument (in our case, the implicit argument "input") by the value of the procedure nextatom operated on that pointer. Otherwise, test returns false. Using these conventions the function test appears to have one input argument, a string, and one resultant value, either true or false.

Example: Assume the variable input has been declared appropriately and that it contains a pointer to the beginning of the following edited source string:

```
    + 1 * 2
```

Then if the statement

```
test('+')
```

is executed, the variable input will subsequently contain a pointer to the atom 1. If, with input pointing to the atom 1,

```
test('*')
```

is executed, the variable input will not be changed.

Assuming the same edited string is pointed to from the variable input, the value of the following expression will be false (and input will subsequently point to the atom *):

```
if test('+') and not test('1') then true else test('1')
```

## gather and scatter

gather and scatter are two very simple list processing procedures. gather accepts two integer arguments and returns a single integer as a value. The integer which gather returns as a value can be thought of as a list whose elements are the two integers supplied as arguments. scatter does just the opposite. scatter accepts three arguments, the first of which is an integer previously returned by a gather operation, the other two arguments are the names of variables which are to receive the two integers represented by the first argument.

Example:

```
    integer a,b,c;
```

```
a := gather (2,4);

b := gather (7,a);

scatter (b,a,c);

scatter (c,c,a)
```

After this sequence of code has been executed, the variable a contains the integer 4, and the variable c the integer 2.

gather and scatter are readily programmed in ALGOL, and are shown below. gatherarray and other undefined identifiers are assumed declared in a higher level procedure and will be discussed later.

Note that for debugging purposes it is convenient to be able to distinguish the integers manipulated by gather and scatter from the ones being manipulated by editor, nextatom and test.

```
integer procedure gather (arg1,arg2) ; value arg1, arg2;
   integer arg1, arg2;
   begin integer work1;
      if nextavailable = 9999999999 then
         error (Ø)
      else
         begin
            work1 := nextavailable;
            nextavailable := gatherarray [work1];
            gatherarray [work1] := arg1;
            gatherarray [work1+1] := arg2;
            gather := work1
         end
   end gather
```

```
procedure scatter (from, to1, to2) ; value from;
   integer from, to1, to2;
   begin
      to1 := gatherarray [from];
      to2 := gatherarray [from +1];
      gatherarray [from] := nextavailable;
      nextavailable := from
   end scatter
```

join and unwind

join and unwind comprise a mechanism for generating object code. join is used during translation to tie together pieces of code as they are generated, and unwind is used subsequent to translation to tidy up the final string of object code.

join, like test, is defined with special conventions that make its use as natural as possible. In fact, as we are about to define it, join violates two of the rules of ALGOL. First, it has a variable number of arguments; and second, it allows any of its arguments to be either strings or integers. Both of these conveniences can be dropped, if necessary. join can be defined as a procedure with two arguments (and then applied repetitively); and since the set of string arguments that are normally given to it is small, they can easily be represented as integers (in particular, we could define a simple function which mapped the strings of interest into integer representations). But it has been our experience that ALGOL compilers are quite permissive about the ways in which external (machine language) procedures are called, and that in fact the join statements as they are defined are permitted.

As stated above, join accepts a variable number of arguments. Each argument may be either a string, which represents itself, or an integer. No integer argument to join represents itself. Zero represents the null argument and is ignored. A non-zero integer may represent either:

(1) an atom in an edited source string;

(2) a field in a symbol table (to be discussed below); or

(3) the result of a previous join operation.

join returns as its output value an integer which represents the concatenation of its arguments. In this way join is similar to gather. join and gather are used in different ways, however. gather is used whenever a temporary compression of data is called for, whereas join permanently associates data until the end of translation and is intended primarily for the construction of object code.

unwind is used at the end of translation for the final construction of object code. Whether or not unwind is actually needed depends on how join is implemented. The suggested implementation is as follows: join examines its arguments in the sequence in which they are supplied. If an argument is a string, or an integer representing the result of a previous join operation, the argument is written on "tape" without any further evaluation. If the argument is an integer representing an atom in an edited source string, or a field in a symbol table, then the actual atom or value of the field is fetched and written on the tape. When the arguments have been exhausted a terminal mark is written on the tape, join returns as its value the integer address of the first value it wrote on the tape.

The function of unwind is: given an integer address to the join tape, construct a single string consisting of the orderly concatenation of all the values written at that place on the tape. Where a pointer to another place on the tape occurs, appeal to unwind recursively to fetch the string consisting of the orderly concatenation of all the values written at that place on the tape, and so forth. Thus unwind has one argument, an integer address to the join tape, and returns as its value a pointer to an orderly concatenated string which is the object code of a translation. By orderly concatenation we mean concatenation according to some simple rules that make sense for whatever object code (or class of object codes) is being generated. If, for example, ALGOL is the object code being generated, then it would make sense to form the concatenation by inserting a space between values.

join is used throughout the translation process to form pieces of the object code, which may then be manipulated as simple integers regardless of their length. unwind is used only at the end of translation to unravel the join tape and produce the final version of the object code. Clearly the integers which represent the result of join operations must be distinguishable from other integer representations.

Example:

```
integer a,b,c;

a := join ('R','S','T');

b := join ('P','Q',a,'U');

c := join (b,b);

a := unwind (c);
```

After this string of code has been executed, the variable a will contain a pointer to the object code "P Q R S T U P Q R S T U".

generatedname

When generating object code it becomes necessary on occasion to invent an identifier for use as the name of a label or a variable. generatedname is a procedure with no arguments that returns an integer representing an identifier. generatedname returns a unique value each time it is called.

Example:

```
integer a,b;

a := generatedname;

b := join (a,':GO',a,';')
```

After the execution of the above code, the variable b contains a representation of the object code:

```
GN1 : GO GN1;
```

where GN1 is an identifier that might be generated by generatedname.

## error

error is a procedure that generates diagnostic messages In a full blown system error would normally have a full set of input parameters. These might specify an index to a standard error message stored in a catalog of error messages, parameters to be substituted in the standard message, specification of where the message is to appear, and so forth. For our purposes here error will accept just one input parameter, an integer designating a standard error message.

Example:

error (67)

## Other Utility Procedures

The utility procedures that have been outlined above form the basis of any complete set that would be required for the implementation of a simple language, and are sufficient for all the examples that have been included in this article. Depending on exactly what language is to be implemented, some additional utility procedures may be required. In the informal framework we've described, they can be added ad infinitum according to the whim of the designer. The basic idea is to separate into a machine language utility procedure those algorithms which either consume a large amount of space or time, or cannot otherwise be conveniently programmed in ALGOL. The general appearance of the complete compiler is that while the majority of the logic of translation is in ALGOL, the majority of time-consuming work is done in the machine language utility procedures.

One set of utility procedures that are conspicuously missing in the discussion above are symbol table manipulation procedures. They have been left out in part because they are not essential for the examples to be employed below (and certainly not essential to the concepts discussed), but also because we feel the design of symbol tables should be tailored to each individual language, even to the extent of their interface with the procedures that use them.

## SECTION III

## GLOBAL VARIABLES AND THE COMPILER PROCEDURE

The general approach to the structuring of a simple compiler that is being delineated here is to represent each syntactic element in the source language with a procedure in ALGOL (or other algebraic language) which is entirely responsible for the translation of an instance of that syntactic element into object language. At the top of a hierarchy of such procedures there is, of course, one "main" procedure. This main procedure has the responsibility of declaring all global quantities (simple variables, arrays, and procedures), and performing any initialization and finalization that may need to be done.

An example of a main procedure for a simple compiler, which is named <u>compiler</u>, is shown in Figure 1. <u>compiler</u> is quite transparent, except (alas) when it comes to its parameters. Because the most important of its parameters are not acceptable ALGOL data types, and because the integer representation scheme applies only inside the compiler itself, it is necessary to go underground and specify the communication of parameters to and from <u>compiler</u> with conventions that lie outside the scope of the ALGOL language.

The input values supplied to a compiler normally include:

(1) a source text

(2) a list of resources available

(3) rules to be followed during translation (ranging from the specification of an end of text symbol to a complete language syntax specification)

(4) indicators of operating mode (on-line or off-line, debugging or production, etc.)

(5) designation of the disposition of output values.

The output values produced by a compiler normally include:

(1) an object text

(2) diagnostics

(3) listings of the source and object texts

The conventions that have been assumed by _compiler_ and its substructure of procedures in the acquisition and disposition of input and output values are as follows:

(1) the procedure _editor_ has the responsibility for obtaining the source text and delivering a listing of the source text.

(2) resources available are listed as explicit parameters (only one is listed in Figure 1)

(3) the procedure _unwind_ has the responsibility for delivering the object program

(4) the procedure _error_ has the responsibility for disposing of diagnostics.

Presumably _editor_, _unwind_, and _error_ are machine language procedures that have been programmed to agree on the acquisition and disposition of input and output values.

For the purpose of the examples which follow, the reader should keep in mind the following things about global variables and the _compiler_ procedure:

(1) the variable _input_ contains a pointer to the current atom of the source string.

(2) the _variable_ translation is used to communicate the translation of a syntactic element from the procedure that translates it to the procedure calling for the translation.

(3) a procedure for each syntactic element is declared in the procedure heading of _compiler_; each procedure that corresponds to a syntactic element returns a value of **true** when it successfully scans and translates an instance of the syntactic element it represents, and a value of **false** otherwise.

## SECTION IV

## A SIMPLE EXAMPLE

The syntactic element <program> from the Revised ALGOL 60 report[4] has been chosen as a simple example. A program is defined in Panini Backus Form[5] as follows:

```
<program> ::= <block>|<compound statement>
```

An ALGOL procedure for translating this syntactic element is:

```
Boolean procedure program;
    begin
        program := if block then true else compoundstatement
    end
```

```
Boolean procedure compiler (gatherarraylimit); value gatherarraylimit;
    integer gatherarraylimit;
    comment other appropriate arguments are inserted in place above;
    begin
        integer array gatherarray [1 : gatherarraylimit];
        integer input, nextavailable, translation;
        procedure initializegatherarray
            begin integer i;
                for i := step 2 until gatherarraylimit - 3 do
                   gatherarray [i] := i+2;
                gatherarray [if gatherarraylimit/2 = gatherarraylimit+1)/2
                   then gatherarraylimit-2 else gatherarraylimit-1] :=
                   9999999999;
                nextavailable := 1
            end
            comment all other global array, simple variable and
               procedure declarations are inserted here;
            initializegatherarray;
            input := editor;
            translation := Ø;
            comment all other initialization steps are inserted here;
            compiler := program;
            unwind (translation);
            comment all other finalization steps are inserted here;
    end compiler
```

Figure 1. The Main Procedure compiler

## SECTION V
## FLOYD'S EXAMPLE

The example chosen by Floyd in [1] is a simple assignment statement defined by these Panini Backus expressions:

<assignment statement> ::= <left part list> <arithmetic expression>

<left part list> ::= <variable> :=|<left part list> <variable> :=

<arithmetic expression> ::= <term>|<adding operator> <term>|
    <arithmetic expression> <adding operator> <term>

<term> ::= <factor>|<term> <multiplying operator> <factor>

<factor> ::= <primary>|<factor>↑<primary>

<primary> ::= <procedure identifier>(<arithmetic expression>)|
    (<arithmetic expression>) |<variable>

The object language into which simple assignment statements are to be translated is a simplified ALGOL in which only one operation is permitted to the right of ":=", and only one variable to the left.

One way in which this example is more difficult than the preceding one is that some actual translation will have to be done, and the utility procedures will need to be employed. Another way in which this example is more difficult is that left recursion is used in four of the six definitions. In top-to-bottom analysis, left recursion is a problem that has to be handled as a special case To see the problem, consider the definition:

<A> ::= <A> B | C

A procedure _A_ is constructed with the responsibility for translating syntactic elements of type A. But the first step _A_ takes is to appeal to _A_ in order to check out the first alternative of the definition of A. An endless sequence of operations results. This difficulty is overcome in this way:

The alternative definitions for a syntactic element are grouped into two categories: those which begin with an instance of the syntactic element being defined, and those which do not. Each of the definitions that do not so begin are attempted first. If none of those definitions can be satisfied, then there is no hope for scanning and translating an instance of the syntactic element being attempted, and the attempt is given up. If one of those definitions is successful, then instead of being satisfied with success, the left recursive definitions are examined. The first element of each left recursive definition is deleted (the recursive mention of the syntactic element being translated) and the remaining definition segments are matched against the source string repetitively until no more scanning can be done.

Thus, if the string

B C D B E C F D

is tested against the definition

<A> ::= <A> B | C

it would be rejected because it does not satisfy the definition C.

The string

C B B B B B C D B C F B

would, on the other hand, yield an instance of A, namely,

C B B B B B.

The non-recursive definition (C) being satisfied, the remaining definition segment of the left recursive definition (B) is successfully matched against the source string five times.

ALGOL procedures for translating assignment statements as defined above are shown in Figures 2 through 8. Note the definition of <u>gather</u> 1 to extend the usefulness of the utility procedures.

The reader will probably find it useful to consider an example. For the source string

```
P := Q := A * sin(B)/C ↑ (-D)
```

the object string

```
T1 := sin(B);

T2 := A*T1;

T3 := C ↑ -D;
```

```
Boolean procedure assignmentstatement;
    begin integer work1, work2, work3, work4;
        if leftpartlist then
            begin
                scatter (translation, work1, work2);
                arithmeticexpression;
                scatter (translation, work3, work4);
                translation := join (work3, ';', work1, work4, ';', work2);
                assignmentstatement := true
            end
        else
            assignmentstatement := false
    end assignmentstatement
```

Figure 2. Translating <assignment statement>

```
Boolean procedure leftpartlist;
     begin integer saveinput, work1, work2;
          saveinput := input;
          if variable then
               begin
                    if test (':=') then
                         begin
                              work1 := translation;
                              work2 := Ø;
                              go to leftrecursivealternate
                         end
               end
          input := saveinput;
          leftpartlist := false;
          go to end;
          leftrecursivealternate:
               saveinput := input;
               if variable then
                    begin
                         if test (':=') then
```

Figure 3. Translating <left part list>

```
                    begin
                        work2 := join (work1, ':=', translation,
                           ';', work2);
                        work1 := translation;
                        go to leftrecursivealternate
                    end
                end
            input := saveinput;
            translation := gather (join (work1, ':='), work2);
            leftpartlist := true;
end:   end leftpartlist
```

Figure 3. Translating <left part list> (concluded)

```
Boolean procedure arithmeticexpression;
     begin integer work1, work2;
         if term then
             begin
                 work1 := translation;
                 go to leftrecursivealternate
             end;
         if test ('+') then
             begin
                 term;
                 work1 := translation;
                 go to leftrecursivealternate
             end;
         if test ('-') then
             begin
                 term;
                 work1 := gather1 (gather (Ø,Ø), join ('-'), translation);
                 go to leftrecursivealternate
             end;
         arithmeticexpression := false;
         go to end;
         leftrecursivealternate
             if addingoperator then
```

Figure 4. Translating <arithmetic expression>

```
                    begin
                        work2 := translation;
                        term;
                        work1 := gather1 (work1, work2, translation);
                        go to leftrecursivealternate
                    end;
                arithmeticexpression := true;
                translation := work1;
end:   end arithmeticexpression
```

Figure 4. Translating <arithmetic expression> (concluded)

```
Boolean procedure term;

     begin integer work1, work2;

          if factor then work1 := translation

          else begin term := false; go to end end;

          leftrecursivealternate:

               if multiplyingoperator then

                    begin

                         work2 := translation;

                         factor;

                         work1 := gather1 (work1, work2, translation);

                         go to leftrecursivealternate

                    end;

               term := true;

               translation := work1;

end:  end term
```

Figure 5. Translating <term>

```
Boolean procedure factor;
     begin integer work1;
         if primary then work1 := translation
         else begin factor := false; go to end end;
         leftrecursivealternate:
             if test ('↑') then
                 begin
                     primary;
                     work1 := gather1 (work1, join ('↑'), translation);
                     go to leftrecursivealternate
                 end;
             factor := true;
             translation := work1;
end:  end factor
```

Figure 6. Translating <factor>

```
Boolean procedure primary;
    begin integer work1, work2, work3, temp;
        primary := true;
        if procedureidentifier then
            begin
                work1 := translation;
                test ('(');
                arithmeticexpression;
                test (')');
                scatter (translation, work2, work3);
                temp := generatedtemp;
                translation := gather (join (work2, ';', temp, ':=',
                   work1, '(', work3, ')'), temp)
            end
        else
            if test ('(') then
                begin
                    arithmeticexpression;
                    test (')')
                end
        else
            if variable then translation := gather (∅, translation)
        else
            primary := false
end primary
```

Figure 7. Translating <primary>

```
integer procedure gather1 (leftoperand, operator, rightoperand);
    value leftoperand, operator, rightoperand;
    integer leftoperand, operator, rightoperand;
        begin integer leftcode, leftvalue, rightcode, rightvalue, temp;
            scatter (leftoperand, leftcode, leftvalue);
            scatter (rightoperand, rightcode, rightvalue);
            temp := generatedtemp;
            gather1 := gather (join (leftcode, ';', rightcode, ';',
               temp, ':=', leftvalue, operator, rightvalue), temp)
        end gather1
```

Figure 8. gather 1 : an Extension of the Utility Procedures

```
        T4 := T2/T3;

        Q := T4;

        P := Q;
```

will be generated.

A comparison of Figures 2 - 8 with Floyd's original example will show the former to be far lengthier, but no less lucid for it: and if the utility procedures have been implemented cleverly, no less efficient.

## SECTION VI

## TRANSLATING PANINI BACKUS FORM

The regularity of Figures 2 through 8 suggests that the procedures there may themselves be generated mechanically. Considering Panini Backus Form as a source language, its definition in Panini Backus Form looks like this (circles are used to distinguish constants in the language being defined from constants in the metalanguage):

```
<statement> ::= <variable> := <alternate list>

<alternate list> ::= <alternate>|<alternate list>|<alternate>

<alternate> ::= <factor>|<alternate> <factor>

<factor> ::= <variable>|<basic symbol string>

<variable> ::=<<identifier>>
```

A set of ALGOL procedures for translating this language into an ALGOL equivalent are shown in Figures 9 through 13. Two simplifying assumptions have been made:

(1) that it is valid to accept the first alternative in a definition that matches without checking to see if there is a longer one that matches; and

(2) that no definition has more than one left recursive alternate.

The object code that is produced from a source string of Panini Backus Form is sufficient for scanning instances of the language defined, but additions must be made by hand before translations can be made.

An example of a source string in Panini Backus Form that has been translated by the procedures in Figures 9 through 13 is the five lines above. The object code, suitably doctored so that a translation is made to ALGOL, is shown in Figures 9 through 13.

## SECTION VII

## ERROR HANDLING AND CREATIVE ADHOCERY

Unfortunately, this compiler scheme has no facility for the automatic detection and reporting of errors in the source text. The algorithms generated from Panini Backus Form by the procedures of the preceding section simply back up when the source text is not syntactically well-formed. This means that when the slightest error is embedded in the source text, the highest level procedure simply returns false, with no indication as to how many errors were found, what kind of errors were found, or whether or not the errors found would have prevented meaningful execution. But although there is no automatic facility for error handling, there are no obstacles barring the implementation of a well formulated error policy. With simple modifications to the algorithms mechanically generated by the procedures of preceding section, the procedure representing a syntactic element can be made to:

(1) generate an error message, back up (the source text), and return false,

(2) generate an error message, scan forward to an appropriate point, and return false,

(3) generate an error message, scan forward to an appropriate point, set up a default piece of object code in translation, and return true,

(4) generate an error message, back up, and wait for the source text to be modified on line, or

(5) almost anything else along these lines.

Distinctions can be made between "hard" errors and "soft" errors, and fancy listings that include a snapshot of the segment of the source text containing the error can be produced. The important thing is that the designer specify an error policy that is consistent, and relates well to the nature of the source language, and the environment(s) in which the compiler is to operate.

```
Boolean procedure statement;
     begin integer saveinput, work1, work2;
          saveinput := input;
          if variable then
               begin
                    work1 := translation;
                    if test ('::=') then
                        begin
                            if alternatelist (work1) then
                                begin
                                    scatter (translation, translation, work2);
                                    translation := join ('Boolean procedure',
                                       work1,'; begin integer saveinput;
                                       saveinput := input;', translation,
                                       ';input := saveinput;', work1, ':=
                                       false; go to end; leftrecursivealternate:',
                                       work2,';',work1,'  := true; end: end ',
                                       work1);
                                    go to leftrecursivealternate
                                end
                            end
                        end;
                    input := saveinput;
                    statement := false;
```

Figure 9. Translating <statement>

```
                    go to end;
                    leftrecursivealternate:
                    statement := true;
end:  end statement
```

Figure 9. Translating < statement > (concluded)

```
Boolean procedure alternatelist (categoryname); value categoryname;
   integer categoryname;
   begin integer saveinput, work1, work2, work3;
      saveinput := input;
      if alternate (categoryname) then
         begin
            scatter (translation, work1, work2);
            go to leftrecursivealternate
         end;
      input := saveinput;
      alternatelist := false;
      go to end;
      leftrecursivealternate:
         saveinput := input;
         if alternate (categoryname) then
            begin
               scatter (translation, translation, work3);
               if not work3 = Ø and not work2 = Ø then error
                  (integer ('more than one left recursive
                  alternate'));
               work1 := join (work1, ';', translation);
               work2 := join (work2, work3);
               go to leftrecursivealternate
            end
```

Figure 10. Translating < alternatelist >

```
                input := saveinput;
                translation := gather (work1, work2);
                alternatelist := true;
end:  end alternatelist
```

Figure 10. Translating < alternatelist> (concluded)

```
Boolean procedure alternate (categoryname); value categoryname;
    integer categoryname;
    begin integer saveinput, work1, work2, work3;
        saveinput := input;
        if factor then
            begin
                work1 := translation;
                work2 := work3 := ∅;
                go to leftrecursivealternate
            end;
        input := saveinput;
        alternate := false;
        go to end;
        leftrecursivealternate:
            saveinput := input;
            if factor then
                begin
                    work2 := join (work2, 'if', translation,
                      'then begin');
                    work3 := join ('end', work3);
                    go to leftrecursivealternate
                end;
```

Figure 11. Translating <alternate>

```
                    input := saveinput;
               work2 := join (work2, 'go to leftrecursivealternate', work3);
               if compare (work1, categoryname) then
                    translation := gather (∅, join ('saveinput := input;',
                       work2, '; input := saveinput'))
               else
                    translation := gather (join ('if', work1, 'then begin',
                       work2, 'end'),∅);
               alternate := true;
     end: end alternate
```

Figure 11. Translating < alternate > (concluded)

```
Boolean procedure factor;
    begin integer saveinput;
        saveinput := input;
        if variable then
            begin
                go to leftrecursivealternate
            end;
        if basicsymbolstring then
            begin
                translation := join ('test(',translation,')');
                go to leftrecursivealternate
            end;
        input := saveinput;
        factor := false;
        go to end;
        leftrecursivealternate:
        factor := true;
end:  end factor
```

Figure 12. Translating <factor>

```
Boolean procedure variable;
     begin integer saveinput;
          saveinput := input;
          if test ('<') then
               begin
                    if identifier then
                         begin
                              if test ('>') then
                                   begin
                                        go to leftrecursivealternate
                                   end
                         end
               end
          input := saveinput;
          variable := false;
          go to end;
          leftrecursivealternate:
          variable := true;
end:  end variable
```

Figure 13. Translating < variable >.

To illustrate one solution to an error handling problem, and also to show what we mean by creative adhocery, the remainder of this section is devoted to the implementation of the syntactic element in ALGOL 60, compound tail.

The definition of compound tail occurs in Section 4.1 of the Revised ALGOL 60 report, part of which is printed below:

```
<program> ::= <block>|<compound statement>

<block> ::= <unlabeled block>|<label> : <block>

<unlabeled block> ::= <block head> ; <compound tail>

<block head> ::= begin <declaration>|<block head> ; <declaration>

<compound tail> ::= <statement>end|<statement> ; <compound tail>

<compound statement> ::= <unlabeled compound>|<label> : <compound statement>
```

One gross error strategy for ALGOL 60 is to allow each syntactic element at the level of statement or below to either generate error messages, back up and return **false**; or simply back up and return **false**. The choice of these two actions depends on whether the failure to scan off an instance of a syntactic element is deemed to be an error or not. Responsibility for scanning forward to an appropriate point for the resumption of normal processing is centralized in compound tail.

Figure 14 shows the results of processing the Panini Backus definition of compound tail with the programs of the previous section.

Figure 15 shows the same procedure after some preliminary modifications have taken place. The appeal to the procedure *statement* has been "factored out," and some unnecessary **begin** **end** pairs have been removed.

In Figure 16 the necessary code to perform a simple translation to object code has been added. This additional code defines the translation of a compound tail to be the concatenation of the translations of its components. But to make things not quite so trivial the procedure *superjoin* has been defined to perform the concatenation, and the following convention assumed: a translation in the context of any of the syntactic elements of Section III of the ALGOL 60 report consists of two halves, a left half and a right half, which have been *gathered*. The procedure *super* join concatenates the left halves to the left halves and the right halves to the right halves

```
Boolean procedure compoundtail;
      begin integer saveinput;
           saveinput := input;
           if statement then
                begin
                     if test ('end') then
                          begin
                               go to leftrecursivealternate
                          end
                end;
           input := saveinput;
           if statement then
                begin
                     if test (';') then
                          begin
                               if compoundtail then
                                    begin
                                         go to leftrecursivealternate
                                    end
                          end
                end
```

Figure 14. Translating < compound tail > : a Mechanically Generated Scanning Algorithm

```
        input := saveinput;
        compoundtail := false;
        go to end;
        leftrecursivealternate:
        compoundtail := true;
end:  end compoundtail
```

Figure 14. Translating < compound tail > : a Mechanically Generated Scanning Algorithm (concluded)

```
Boolean procedure compoundtail;
     begin integer saveinput;
          saveinput := input;
          if statement then
               begin
                    if test ('end') then go to leftrecursivealternate;
                    if test (';') then
                         begin
                              if compoundtail then
                                   begin
                                        go to leftrecursivealternate
                                   end
                         end
               end
          input := saveinput;
          compoundtail := false;
          go to end;
          leftrecursivealternate:
          compoundtail := true;
end:  end compoundtail
```

Figure 15. Translating < compound tail > : a Cleaned Up Scanning Algorithm

```
Boolean procedure compoundtail;
    begin integer saveinput, work1;
         saveinput := input;
         if statement then
              begin
                   if test ('end') then go to leftrecursivealternate;
                   if test (';') then
                        begin
                             work1 := translation;
                             if compoundtail then
                                  begin
                                       translation := superjoin (work1,
                                          translation);
                                       go to leftrecursivealternate
                                  end
                        end
              end
         input := saveinput;
         compoundtail := false;
         go to end;
         leftrecursivealternate:
         compoundtail := true;
end:   end compoundtail
```

Figure 16. Translating <compound tail> : the Addition of Code to Perform the Translation

and forms a new gathered translation. In this way initialization steps required by each component (the left halves) can be grouped together. This technique has been mentioned here only as an example of the creative adhocery that can be applied in building translation procedures; it is not intended as a complete solution to the problems of compiling ALGOL 60 statements. The procedure superjoin is shown in Figure 17.

Figure 18 reflects some further thinking about the implications of centralizing error recovery in compoundtail. It has been decided that compoundtail will always return **true**, and the procedure has been simplified accordingly. It has also been realized that because of the definition of the empty statement, the procedure statement will always return **true**; the attendant simplifications have been made.

Finally in Figure 19 the code for generating two error messages and spacing forward to the next translatable segment has been added.

```
integer procedure superjoin (leftoperand, rightoperand);
    value leftoperand, rightoperand;
    integer leftoperand, rightoperand;
        begin integer leftvalue1, leftvalue2, rightvalue1, rightvalue2;
            scatter (leftoperand, leftvalue1, leftvalue2);
            scatter (rightoperand, rightvalue1, rightvalue2);
            superjoin := gather (join (leftvalue1, rightvalue1),
               join (leftvalue2, rightvalue2))
        end superjoin
```

Figure 17. superjoin : an Extension of the Utility Procedure

```
Boolean procedure compoundtail;
    begin integer work1;
         statement;
         work1 := translation;
         if test ('end') then go to leftrecursivealternate;
         if test (';') then
              begin
                   compoundtail;
                   translation := superjoin (work1, translation);
                   go to leftrecursivealternate
              end
         leftrecursivealternate:
         compoundtail := true
    end compoundtail
```

Figure 18. Translating <compound tail > : Realizing the Implications of Centralizing Error Recovery Here

```
Boolean procedure compoundtail;
    begin integer saveinput, work1;
        statement;
        work1 := translation;
        if test ('end') then go to leftrecursivealternate;
        if test (';') then
            l1:
            begin
                compoundtail;
                translation := superjoin (work1, translation);
                go to leftrecursivealternate
            end;
        if saveinput ≠ input then
            begin
                error (1); comment 'a ; or end appears to be missing';
                go to l1
            end;
        error (2); comment 'an untranslatable statement appears to have
           been encountered';
        l2:
        input := saveinput := nextatom (input);
        if test ('end') then go to leftrecursivealternate;
        if test (';') then go to l1;
```

Figure 19. Translating < compound tail > : the Addition of Code to Space Forward and Generate Error Messages

```
         if not (test ('go') V test ('if') V test ('for') V test ('comment')
            V test ('begin') V test ('own') V test ('Boolean') V test
            ('integer') V test ('real') V test ('array') V test ('switch')
            V test ('procedure')) then go to 12;
         input := saveinput;
         go to 11;
         leftrecursivealternate:
         compoundtail := true
     end compoundtail
```

Figure 19. Translating < compound tail > : the Addition of Code to Space Forward and Generate Error Messages (concluded)

## SECTION VIII

## SYNTACTIC MACROS

Syntactic macros were introduced by Cheatham in [6]. Syntactic macros differ from ordinary macros in two ways:

(1) instances of arbitrary syntactic elements are allowed as parameters, and

(2) the scope of definition of a macro can be limited to certain syntactic contexts.

Thus syntactic macros clean up two of the problems that have always plagued conventional macros: the problem of delimiting parameters, and the problem of the macro definition being triggered in contexts where it is not desired.

A capability similar to syntactic macros (although less formal) can be provided within the compiler environment described here. The technique is to describe the syntax of the macro form in Panini Backus Form, decide what its equivalence is in the normal source language, and build a procedure which scans the macro form, produces the normal source language, and then appeals to the appropriate procedure in the normal source language compiler to perform the second level of translation. One additional utility capability is needed. The procedures <u>append</u> and <u>append</u>l build a stream of atoms similar to the stream produced by <u>editor</u>. The procedure <u>append</u> has two arguments: a pointer to a stream to which it is to append a new atom (or zero, which represents a null stream), and the new atom expressed as a string. The procedure <u>append</u>l has three arguments: a pointer to a stream to which it is to append additional atoms, a pointer to an existing stream of atoms which marks the first atom to be appended from that stream, and a second pointer to that same existing stream which marks the point at which no more atoms are to be extracted.

*Taken from [6].

For example*, suppose that an application for ALGOL involves the heavy use of square arrays, and we would like to be able to say:

```
    matrix A [n]
```

instead of

```
    array A [1 : n, 1 : n]
```

The syntax for such a matrix declaration can be defined by the Panini Backus metaexpression:

```
<matrix declaration> ::= matrix<identifier> [<arithmetic expression>]
```

When this metaexpression is processed by the programs of Section 7, the procedure in Figure 20 is the result. Figure 21 shows the same procedure after it has been suitably modified to serve as a macro.

The production of macro procedures can be automated somewhat by the introduction of statements in the normal source language for the purpose of defining macros. For the example considered above such a statement might look like this:

```
macro matrix (i is identifier, n is arithmeticexpression);

    context declaration;

{matrix i [n]} means {array i [1:n, 1:n]}
```

This statement declares the macro matrix. It has two parameters: i, which should be scanned as an identifier, and n, which should be scanned as an arithmetic expression. The scope of definition of the macro is as if it were another alternative in the definition of the syntactic element, declaration, of the normal source language. The final line of the example defines the mapping from the macro form (between the first pair of braces) to the normal source language (between the second pair of braces).

A specification for the complete syntax of a macro declaration statement, as it might appear if it were part of ALGOL 60, is shown in Figure 22. Its implementation in the compiler environment described here is a fairly straightforward process, although a little creative adhocery is required. The reader may wish to try his hand at it.

```
Boolean procedure matrixdeclaration:
    begin integer saveinput;
        saveinput := input;
        if test ('matrix') then
            begin
                if identifier then
                    begin
                        if test ('[') then
                            begin
                                if arithmeticexpression then
                                    begin
                                        if test (']') then
                                            begin
                                                go to leftrecursivealternate
                                            end
                                    end
                            end
                    end
            end
        input := saveinput;
        matrixdeclaration := false;
        go to end;
        leftrecursivealternate:
        matrixdeclaration := true;
end:end matrixdeclaration
```

Figure 20. Expanding < matrix declaration > : a Mechanically Generated Scanning Algorithm

```
Boolean procedure matrixdeclaration;
    begin integer saveinput, stream, saveinput1;
         saveinput := input;
         if test ('matrix') then
              begin
                   string := Ø
                   saveinput1 := input;
                   if identifier then
                        begin
                             append (stream, 'array');
                             append1 (stream, saveinput1, input);
                             if test ('[') then
                                  begin
                                       saveinput1 := input;
                                       if arithmeticexpression then
                                            begin
                                                 append (stream, '[');
                                                 append (stream, ']');
                                                 append (stream, ':');
                                                 append1(stream, saveinput1,
                                                   input);
                                                 append (stream, ',');
```

Figure 21. Expanding < matrix declaration >: the Addition of Code to Perform the Expansion

```
                                        append (stream, '1');
                                        append (stream, ':');
                                        append1(stream, saveinput1,
                                          input);
                                        append (stream, ']');
                                        if test (']') then
                                            begin
                                                go to
                                                  leftrecursivealternate
                                            end
                                   end
                             end
                       end
                 end
          input := saveinput;
          matrixdeclaration := false;
          go to end;
          leftrecursivealternate:
          saveinput := input;
          input := stream;
          matrixdeclaration := arraydeclaration;
          input := saveinput;
     end:  end matrixdeclaration
```

Figure 21. Expanding< matrix declaration >: the Addition of Code to Perform the Expansion (concluded)

```
<macro declaration> ::= macro<macro heading><macro body>
<macro heading> ::=<macro identifier><parameter specification part>;
   <allowable context specification part>
<macro identifier> ::=<identifier>
<parameter specification part> ::=<empty>|(<parameter specification list>)
<parameter specification list> ::=<parameter specifier>|
   <parameter specification list> , <parameter specifier>
<parameter specifier> ::=<formal parameter>is <syntactic element identifier>
<allowable context specification part> ::=<empty>|
   context <allowable context specification list>;
<allowable context specification list> ::=<syntactic element identifier>|
   <allowable context specification list>,<syntactic element identifier>
<syntactic element identifier> ::=<identifier>
<macro body> ::=<macro form>|<macro form> means <defining form>
<macro form> ::={<open sequence>}
<defining form> ::={<open sequence>}
<open sequence> ::= <proper sequence>|{<open sequence>}|
   <open sequence><open sequence>
<proper sequence>::=<sequence element>|<proper sequence><sequence element>
<sequence element>::=<formal parameter>|<identifier>|<string>|
   <any basic symbol except{or}>
```

Figure 22. Syntax of a Macro Declaration Statement

## SECTION IX

## EXPERIENCE

The compiler design presented here was used at MITRE for the implementation of a file processing language, PROFILE, which is part of the C-10 file management system operating on the IBM 1410 (see[7]). PROFILE combines features of command and control query languages with features of algorithmic languages. The implementation on the 1410 was not done in ALGOL 60 but in the C-10 language, STEP, and later in a special macro version of AUTOCODER. ALGOL was used in this paper so that a wider audience could be obtained.

PROFILE is a language that evolved over a period of two years, and during that span of time it underwent four separate implementations. It was our experience that using the technique described here, an implementation required about six man months of effort. But the really desirable characteristic we realized with this technique was the easy maintenance and modification of the compiler by programmers who by no means considered themselves compiler experts. A compiler written in ALGOL as we have described here can be maintained and modified by any ALGOL programmer.

## SECTION X

## REMARK

The utility of algebraic languages having been firmly established for things other than numerical algorithms (especially systems programming), one would think it was time to reconsider the intent and design of higher level algorithmic languages _from first principles_. Surely the approach taken by PL/I, that every conceivable data structure and programming facility be permanently cemented into the language, is not satisfactory.

## SECTION XI

## SUMMARY

A design for the construction of a simple compiler using an algebraic programming language as the primary tool has been described. The chief advantage of this approach is the ready availability of the algebraic language. The basic technique used is to augment the capabilities of the algebraic language with additional data structures implemented by means of external machine language utility procedures. A set of basic utility procedures have been suggested, and examples of language implementations have been given. Panini Backus Form has been considered as a source language for such a compiler, error recovery techniques have been suggested, and syntactic macros have been discussed.

## REFERENCES

1. Robert W. Floyd, "A Descriptive Language for Symbol Manipulation," Journal of the ACM, Vol 8, No 4 (October 1961).

2. Majorie P. Lietzke, "A Method of Syntax-checking ALGOL 60," Communications of the ACM, Vol 7, No 8 (August 1964).

3. Larry Irwin, "Implementing Phrase-Structure Productions in PL/I," Communications of the ACM, Vol 10, No 7 (July 1967).

4. J. W. Backus, et al, "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, Vol 6, No 1 (January 1963).

5. P. Z. Ingerman, letter to the Editor, Communications of the ACM, Vol 10, No 3 (March 1967).

6. T. E. Cheatham, Jr., The Introduction of Definitional Facilities into Higher-Level Programming Languages, CA-6605-0611, Computer Associates, Wakefield, Massachusetts, May 1966. See also Proceedings, AFIPS 1966 Fall Joint Computer Conference, Spartan Book Co., Washington, D.C.

7. G. Steil, Jr., File Management on a Small Computer, MTP-47, The MITRE Corporation, Bedford, Massachusetts, February 1967. See also Proceedings, AFIPS, 1967 Spring Joint Computer Conference, Thompson Book Co., Washington, D.C.

8. T. E. Cheatham, Jr., The Theory and Construction of Compilers, CA-6606-0111, Computer Associates, Inc., Wakefield, Massachusetts, June 1966 (draft report).

9. Julien Green, R. M. Shapiro, F. R. Helt, Jr., R. G. Franciotti, and E. H. Thiel, "Remarks on ALGOL and Symbol Manipulation," Communications of the ACM, Vol 2, No. 9 (September 1959).

10. Joseph W. Smith, "Syntactic and Semantic Augments to ALGOL," Communications of the ACM, Vol 3, No 4 (April 1960).

11. J. W. Carr and J. W. Manson, "Two Subroutines for Symbol Manipulation with an Algebraic Compiler," Communications of the ACM, Vol 4, No 2 (February 1961).

12. M. Brassem and J. Coher, "A Description in ALGOL of a Simplified ALGOL Compiler," IMAG-Groupe ALGOL, No. 27 (April 1964).

13. D. V. Schorre, "Meta II - A Syntax-oriented Compiler Writing Language," Proceedings, ACM 19th National Conference, Philadelphia, Pa., Association for Computing Machinery, New York, N.Y.

Security Classification

**DOCUMENT CONTROL DATA - R & D**

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

| | | |
|---|---|---|
| 1. ORIGINATING ACTIVITY (Corporate author) The MITRE Corporation Bedford, Massachusetts | | 2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED |
| | | 2b. GROUP N/A |
| 3. REPORT TITLE USING THE READILY AVAILABLE ALGEBRAIC LANGUAGE AS A COMPILER ENVIRONMENT | | |
| 4. DESCRIPTIVE NOTES (Type of report and inclusive dates) N/A | | |
| 5. AUTHOR(S) (First name, middle initial, last name) Gilbert P. Steil, Jr. | | |
| 6. REPORT DATE April 1968 | 7a. TOTAL NO. OF PAGES 62 | 7b. NO. OF REFS 13 |
| 8a. CONTRACT OR GRANT NO AF 19(628)-5165 | 9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-67-430 | |
| b. PROJECT NO | | |
| c. | 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) MTR-516 | |
| d. | | |
| 10. DISTRIBUTION STATEMENT This document has been approved for public release and sale; its distribution is unlimited. | | |
| 11. SUPPLEMENTARY NOTES N/A | 12. SPONSORING MILITARY ACTIVITY Deputy for Command Systems, Electronic Systems Division, L.G. Hanscom Field, Bedford, Mass. | |

13. ABSTRACT

The use of algebraic command languages for things other than preparing numerical algorithms has become somewhat popular, in particular for writing compilers. The author feels that the technique of using an algebraic command language for implementing a compiler is a good solid practical idea deserving some additional attention. He feels that this technique will be found particularly useful by organizations not in the business of building commercial compilers, but interested in the implementation of a small special-purpose language, such as a query language for a model of a command system. The purpose of this paper is to describe this technique to such an audience and to comment on the extent of its applicability.

Security Classification

| 14 KEY WORDS | LINK A | | LINK B | | LINK C | |
|---|---|---|---|---|---|---|
| | ROLE | WT | ROLE | WT | ROLE | WT |
| COMPILER | | | | | | |
| ALGOL | | | | | | |
| PARSING TECHNIQUES | | | | | | |

Security Classification