technical contributions

A PROGRAMMING LANGUAGE FOR MINI-COMPUTER SYSTEMS

Frank L Friedman
Victor B Schneider

Department of Computer Science
Purdue University
West Lafayette, Indiana 47906

I. INTRODUCTION AND OVERVIEW

This paper concerns the development of a family of higher level languages which are to serve as the target languages for the decompilation of assembler coded modules of small computer operating systems. The main objective is the design of a machine independent language to serve as the nucleus of these higher level languages, each of which will have to possess characteristics unique to the operating system for which it was designed. The work involved in the development of the nucleus language is part of a research effort intended to show that assembler coded systems programs for a specific class M of machines[1] can be concisely represented in a higher level language and that this higher level representation can be easily moved from one machine in M to another. It is our primary purpose here to describe the main features of this nucleus language and present some reasons why these features have been included in the language.

The major objectives influencing the design of the nucleus language are:
A.) The language is to contain only those data and control structures which will be easily recognized in the assembler coded program, and several additional structures which will be useful in providing a higher level representation of the assembler code without requiring complete program re-organization. (Thus, in a sense, the data structures and control mechanisms to be translated are an important factor in the definition of the nucleus language.)
B.) Programs resulting from the decompilation process should be readable, and should admit a simple and nicely-nested, control structure.
C.) The nucleus language should be portable, extendable, and compilable by a fairly simple compiler. That is, the compiler should be several orders of magnitude less complex than a small operating system.

[1] M is a class of machines characterized by the architectural characteristics listed in Appendix A.

The data and control structures that have been incorporated into the language have been selected empirically - on the basis of our experiences in manually re-writing several small systems routines for the IBM1130 and the CDC6500, and our discussions with systems programmers at Purdue. An index loop and a simple cycle construct are the only two structures available for the specification of code repetition (see section II.C). The general n-way decision box can be handled via a labelled case statement, or, in the case of n=2, by the if ... then ... else statement (II.C).

In an effort to restrict the use of the goto as much as possible, exit, repeat, and case constructs were introduced into the language (II.B.1). In addition, the paucity of run time overhead required to handle procedures permits the use of parameterless procedures to facilitate linked programming in the neat and orderly fashion provided by most assembler languages with no overhead whatsoever! Nevertheless, we are dealing with already existing programs, and our primary mission is the decompilation of these programs, and not their re-organization. Therefore, we felt it wise to retain the goto, although we intend for its use to be limited to situations not easily handled by the other control structures (II.B.4).

The nucleus language is a type-ignorant language having but two very simple data structures: (1) the singly-dimensioned array; and (2) an $n \geq 1$ storage cell[1] structure of variable-length bit strings. The lengths of these bit strings, and hence the format of the n-cell sequence, are specified via a special group declaration (III.B).

It is our intention that the process of decompiling assembler coded modules in a higher level language be only partially automated (see section IV). We believe that attempts to automate this process 'too far' beyond control flow and variable usage analysis, the coalescing of register/core computations, and loop and procedure recognition, to be impractical. However, we hope that our research, although not directed toward a complete automation, may give us a better indication of exactly how far is 'too far'.

[1] A storage cell, as referenced herein, may be thought of as a word or a byte, with the bit size of either dependent upon implementation. The language requires the specification of either word or byte in front of all declarations that effect storage allocation.

II. THE CONTROL STRUCTURES

A. The Program

Programs consist of declaration blocks, scope blocks (sequences of executable statements), and a definition block (see section II.D. for further details). The scope block merely provides a partitioning of a program for readibility, but it does not provide for name independence or any other structural independence normally associated with a 'block'. Thus, the nucleus language is essentially not a block oriented language in the sense of ALGOL 60 or PL/I.

Programs will be able to communicate with each other and with the system via a procedure calling mechanism, and a common pool of storage similar to the COMPOOL of JOVIAL [Sh 63].

B. Hiding the 'goto'

The desire to construct a language that admits only programs with a simple and nicely-nested control structure resulted in the decision to purposely limit the ways in which the goto could be employed at the source level. We were not completely successful in dealing with this problem (see 4. below), but we have considerably reduced the need for the goto by 'hiding' it behind constructs introduced specifically to aid in such camouflage. These special constructs are the exit, repeat, parameterless procedure, and the case. Descriptions of the first two of these constructs is presented here. the case construct is discussed in section C, and the procedure is covered in section D.

1. Exit - The idea for the exit construct originally came from BLISS [Wu 71a], but the version currently used in the nucleus language has been simplified . The exit construct may be used to effect an exit from various levels of three nestable constructs in the nucleus language, the for, cycle, and the case. (Exit from a procedure is coded with a return statement.)

The two forms of exit provided are

```
<exit> ::= exit
         | exit <identifier>
```

The use of the identifier in the exit construct is as a range identifier' [RH 72].

In addition to improving program readability, the identifier may also serve to indicate the control structure from which the exit is to be taken. For example:

```
OUTER: for I ← 1 step ...
            ...
            INNER: cycle:
                   ...; repeat; ...
                   exit INNER;
                   ...
                   end INNER;
            exit OUTER;
            ...
            end OUTER;
```

The exit statement causes a transfer of control out of the designated structure to the first statement that follows the end of that structure.

When the identifier is omitted, exit is taken from the inner-most structure containing the exit. When the identifier is included, it is expected to match the range identifier preceding the first word (for, cycle, or case) of one of the control structures containing the exit. Control is then transfered to the first statement following that structure.

2. Repeat - The repeat statement may be used only within a cycle structure. The repeat causes program control to continue with the first statement of the designated cycle. It may be unconditional or conditional (repeat until <relation>). In the latter case, the repeat causes an exit from the designated cycle when the <relation> takes on the value TRUE. When used with a 'range identifier' (see 1,Exit), the repeat causes a transfer of control to the first statement of the named cycle. When the range identifier is not present, control is transfered to the first statement of the inner-most cycle.

The repeat construct is quite useful in coding nested cycles when the statements in the inner loop are also the beginning statements of the outer loop. In the absence of the repeat, additional cycles would be needed to handle such nested loops, unless the goto was employed.

3. As previously indicated, we were not completely successful in our attempt to banish the most primitive and unrestricted form of the goto. The main reason for this failure is that we are not primarily concerned with writing new programs, but rather with decompiling already existing programs in the nucleus language. Based on our empirical evidence, such programs are likely to contain numerous control sequences which would be quite difficult to code without the use of the goto. Examples of such sequences

are:

1. Abnormal exits from a block or procedure.
2. Certain code sequences requiring node splitting or while construction with additional flags for control flow designations .
3. Code which uses a navigation vector" to determine the sequence of execution of program modules. Such code is often characterized by the need for a single path, non-returning transfer instruction. The goto fills this need better than any other control structure that has been offered in its place.

The goto also provides the only means of transfer out of a scopeblock . The scopeblock exit was abandoned in favor of the goto for the purpose of forcing explicit reference to the target of the goto.

We are not prepared to argue whether or not any of the above-described control structures are necessary or even desirable in a program. The point is that they exist, have been debugged, and, for our purposes, they must be dealt with as efficiently as possible. We feel that this may be best accomplished by retaining the goto in our language, althought we hope that its use will be limited to situations that are not easily handled with the other constructs of the language.

C. Fundamental Control Structures: The Loop and the Decision Box

We indicated earlier that the control mechanisms used in the programs to be decompiled would play a substantial role in the selection of the control structures to be included in the nucleus language. In the programs examined so far, three types of fundamental control mechanisms stand out: (1) single entry, k-exit loops ($k \geq 1$); (2) n-way 'decision boxes' ($n \geq 2$): and (3) procedures. In this section, we will confine our discussion to the first two of these mechanisms. The procedure is discussed in detail in section D.

Loop structures

The loop structure type may be classified further into two sub-types according to the repeat control used. In one type of loop, the range and increment of a counter (or index) is specified and this information is used to control loop repetition. In the other loop type, the logical value of a relation is used explicitly to control loop activity.

It is important to note that no effort is being made to distinguish loops with i exits from those with j ($i \neq j$) exits. In addition, the position of the iteration test(s) inside the loop is irrelevant. As long as the appropriate structured loop escape constructs (see previous section) are admitted into the language, the distinction between the loop structures described above is sufficient for our purposes.

Loop structures controlled by a counter are represented by a for statement of the form

```
 for <identifier> ← <expression> step <expression> until
                    <expression> do:
                                   ...
          end
```

All other loop structures may be represented simply as

cycle:...end

The question of the ease of recognition of these two basic loop constructs is still open, although their presence can be demonstrated in the examples of system code that we have examined.

The n-way decision box

There is little question that some convenient representation of the n-way decision box is desirable in a higher level language. The key question concerns the form which this representation is to take. Two different forms, the jump table and the case statement, are the most frequently used in other languages (see, for example, [Sa 69, on NELIAC], [Co 69], [Wu 71a],[CH 71], and [Ra 72]. We have chosen the labelled case construct [CH 71]:

```
   case <expression> in {<labels> <statement list>}+¹
              else: <statement list> end
```

where

```
              <labels> ::= {<label>:}+
```

and

```
              <label> ::= <constant>²
```

Here, the selection expression is evaluated, and control is passed to the statement list labelled by the resulting value. If no statement list is so labelled, the statement list labelled by else is executed.

¹{ ... }+ indicates that the content of the brackets occurs one or more times.
{ ... }* indicates that the content of the brackets occurs zero or more times.

² A <constant> may be a signed number or a (compile time) equivalenced identifier.

There are a number of reasons for choosing the labelled case statement over the jump table and the other variations considered. We shall list only a few.

1. The use of the jump table places a heavier burden on the programmer then does the use of the labelled case statement shown above. Regardless of the way in which the jump table is constructed, it is the programmer's responsibilty to convert the data to be used in the selection process to the correct integer value (modulo the size of the jump table).
2. The labelled _case_ statement more accurately reflects the activity that is most often associated with the n-way decision box, namely, the selection of one item from a finite, unordered set of elements.
3. The use of the labelled _case_ statement should yield a more readable program, as well as a program that is more likely to be correct after its initial coding, and after each modification to a case statement. For example, we need not worry about switches getting out of order, or cases being added or deleted. As has been shown [AM 71] that the _cycle_, _for_, and _case_ structures described above are indeed sufficient to enable the coding of every flowchartable program. We anticipate that, when combined with the exit and the repeat constructs, they will prove to be both adequate and convenient constructs in any systems implementation language.

D. Procedures

Both the _procedure_ and the _program_, P, are comprised of three different types of blocks.

i) one or more _declaration blocks_ each containing information to be shared in the procedures local to P (if any)
ii) one or more _scope blocks_ each containing executable code
iii) one _definition block_ containing the definitions of all procedures that are referenced in P.

Execution of a procedure is terminated via the _return_ statement. Neither recursive nor re-entrant procedures are supported in the nucleus laguage.

In the nucleus language, all procedures are function procedures with side effects (they may alter the value of any parameter passed, and they also may be assigned a unique value upon return). The value of the procedure may be defined by the variable (identifier, group variable, or array designator) specified in the return statement(s) of the procedure. If the variable is ommitted, the procedure value is assumed to be irrelevant to the calling program.

As is the case with scope blocks, local procedures provide no name or structural independence relative to the outer procedures and programs in which they are declared. For example, it is not possible for an outer procedure to jump into the middle of an inner procedure, but the latter may transfer into the middle of an outer procedure, regardless of the original point of call. Hopefully, this will be a feature that will be used only sparingly , if at all, in the programs that we decompile.

The lack of name and structural independence has enabled us to implement procedures with no run time overhead except for the passing of parameters. The parameterless procedure is, therefore, a convenient and efficient means for representing groups of statements whose execution is required at several different places in a program. Facilities for such representaion are available in assembly languages, most of which have an operation code for 'branch and store instruction register' which enables access to a statement group while at the same time saving the address of the next sequential instruction for the subsequent return.

E. The Assignment Statement

One other construct of the nucleus language, the assignment statement, deserves comment here because of its unique role in the expression syntax of the nucleus language (see Appendix A). The two most important aspects of this role are:

1. Parenthesized expressions may appear on the lefthand side of the assignment operator '←'.
2. Parenthesized assignment statements may be used as operands in expressions.

The value of the assignment is the content of the storage location into which the value of the right hand side is stored. The destination of the assignment is either the location of the designator or the address given by the value of the calculated expression in its leftpart. .[1] These features, combined with the syntactic structure of the language, and the monadic operators loc (location of), and "." (indirect reference) yield a succinct higher level representation of sequences of lengthy arithmetic and masking calculations at the machine code level.

[1] The leftpart of an assignment statement may be a designator or a parenthesized expression. A designator may be an identifier, an array reference, a group component reference, a partial cell reference, or a loc reference.

Example: The IBM1130 Disk Operating System code sequence

```
LD   FWAREA
OR   DZ910
A    DZ965
STO  DZ330+1
A    DZ960
A    DZ280+1
STO  DZ280+1
```

has a target language representation

```
(DZ280+1) ← ((DZ330+1) ← FWAREA ∨ DZ910 + DZ965)
            + DZ960 + .(locDZ280 + 1);
```

III. DATA STRUCTURES AND OPERATIONS IN THE NUCLEUS LANGUAGE

The problem of the automatic recognition of the data types and structures used in the assembler language code is a research topic in its own right. Even the manual recognition of types and structures is often a formidable task. At this point, we are willing to speculate that the singly-dimensioned and typeless array is the most complicated structure that need be recognized by an automatic program structure analyzer. Therefore, the current version of the target language syntax is essentially type-ignorant, although it does include facilities for handling internal, integer, and string constants, as well as special pointer variables. The current syntax is also virtually structureless, including only facilities for the declaration and manipulation of singly-dimensioned and typeless arrays; and for grouping together individually named strings of bits into a simple structure called a group (see III. B, for a brief discussion of a group).

A. Declarations

The purpose of the declaration is to introduce a name, specify its use (as a procedure, group etc.), and (possibly) to specify its initial value . There are four types of declarations in the nucleus language: declarations for simple variables, array variables, group variables, and procedures.

All simple variables, groups, arrays, and procedures, must be declared, and these declarations must appear within some declaration block of the program in which they are to be used.[1] The scope of the declared item is the entire program in which the declaration occurs, but an item must be declared before it is referenced.

We recall that neither the scope block nor the procedure provide any name independence in the nucleus language. Therefore, an identifier already declared at some point in a program may not be re-declared locally anywhere in that program or in a procedure that is local to the program. This restriction may be judged as somewhat inconvenient from a programming point of view, but it does simplify the compilation process. Furthermore, the re-definition feature is not at all useful in the automatic decompilation of assembler language programs. The purpose of allowing declaration blocks between any pair of program statements is solely to enable the programmer to localize item declarations to ease the strain on his memory and to make the program more readable.

In the remainder of this section, we will confine our attention to the simple variable and array declarations, since the procedure declaration is straight-forward and the group declaration is covered in the next section.

Simple variable declarations may occur in any one of three forms.

1) as an element of the declaration list (placed between commas) causing an uninitialized storage cell to be allocated;
2) as a declaration list element of the form
   <identifier> = <expression>[1]
   which causes a storage cell initialized to the value of the expression to be allocated;
3) as a declaration list element of the form
   <identifier> ≡ <expression>[1]
   which causes the value of the expression to be used wherever the identifier appears in the program;

Array declarations may be in one of two forms:

1) as a declaration list element of the form
   <identifier>[<unsigned constant>]
   which causes n = (value of the unsigned constant) uninitialized cells to be allocated to the identifier;
2) as a declaration list element of the form
   <identifier>[<unsigned number>] =

[1]In the case of the simple variable, the required declaration is viewed as an important debugging aid, especially for large programs.
[1] The value of the such an expression must be completely determinable at compile time.

(<constant tuple>)[2]
which causes a one-one assignment of the m≤n constant tuple elements to the first m cells allocated to the n-cell array.

Simple variable declaration 3) and array declaration 2) have been added to the language for decompilation and programming convenience, since such equivalence and initialization features are provided in most assembly languages.

Two implementation dependent "length" characteristics (word and byte) must be specified for each element (simple variable, array or group) that is declared. These enable the user to instruct the compiler to conserve storage whenever possible by assigning units of shorter length (byte) to those program elements such as small integers and characters which do not require the longer (word) storage structure. In the Microdata 1621 implementation of the nucleus language, the byte is 8 bits and the word is 16 bits in length.

More refined partitioning of program storage cells may be accomplished via the use of the special group declaration described below.

B. Group Variables

The group declaration may be used to facilitate the naming and organization of collections of partial words which may have frequent use in a program. The use of a group can enable the storage of several memory objects of different length in a contiguous block.

A group variable is a pointer to a named and ordered sequence of partial storage cells. Group variables are specified in the group variable list part of a group declaration:

<group declaration> ::=

where

```
<group variable > ::= <identifier>
                     |<array designator>
```

For example, the declaration of a group reflecting the format of the IBM1130 Disk Operating System Input/Output

[2] A constant tuple is a parenthesized list of string or numeric constants which are assigned left to right to the cells of the specified array as in a FORTRAN DATA statement.

Command on a 16-bit machine might appear as

```
group IOCC [ADDRESS(16), AREA(5), FUNCTION(3), MODIFIER(8)]
                    FILE1CC, TEMPCC[6]
```

Partial storage cell assignments are made from left to right in the component list. The number of storage cells in the object machine needed to store the components of the group is decided by the compiler as it moves left-to-right Whenever possible, more than one component may be assigned to a single computer cell, but overlapping a partial cell assignment between two cells is not permitted. The above example would effect the allocation of seven groups of format IOCC, and the assignment of the pointers to these groups to the group variables FILE1CC, and TEMPCC[1] thru TEMPCC[6]. In the above example, the compiler would assign two 16-bit words to each IOCC group. ADDRESS would occupy the first of these two words, and the other three components would occupy the second word.

References to the components of a group are of the form

```
AREA of FILE1CC, ADDRESS of TEMPCC[3]
```

The partial cells of a group may be assigned initial values in the declaration section of a program, or be referenced in the scope block of the program, via references (subsequent to the group declaration) to the group components as shown above.

No other type specifications or declarations are available in SIMPL. Type specification can, at some expense, allow the compiler to report on possible misuse of the language, but this advantage is small compared to the limitations and restrictions that types impose, especially when decompiling is involved.

IV. INPUT/OUTPUT

Input/output is handled in string form only. The input/output statement is

```
xio(<device> , <address> , <function> , <modifier> )
```

In general, the parameters of xio are references to the components of a group variable containing the bit strings which define the I/O device, the first word address of the area of the data transfer, and the function to be performed. in order to provide maximum flexibility In handling I/O for the machines in the class M, the code generated for the xio instruction provides an interface between the host machine

I/O facilities and the requested funtion for the system being moved to the host machine. Essentially, this interface code uses the information in the hardware status registers of the host machine to simulate the setting of the corresponding registers for the source machine. The source machine registers are represented in memory cells in the host machine.

Only strings of characters may be transmitted, and the transmission may be on a character basis (under program control) or on a 'record' basis (under the control of the I/O device.) In the latter case, the number of characters to be transmitted is usually contained in the first word of the buffer pointed to by the <address> parameter of the I/O call. To help with conversion from integer to string and back, two built-in procedures INCH (for integer to character conversion) and CHIN (for character to integer) conversion will be available.

V. ADDITIONAL FEATURES

The following are currently under consideration for implementation in the language:

1. A facility to permit the inclusion of assembler language code in line in the higher level language code.
2. A simple macro substitution facility similar to that of SAL [La 69].

We do not anticipate that these features will be useful in the decompiling process. However, they will be helpful to those who use the nucleus language to write new code (e. g., for writing built-in functions and for manually completing the decompilation work).

VI. THE ROLE OF THE NUCLEUS LANGUAGE

The nucleus language is to serve as a practical vehicle for the quick implementation of already existing small computer operating systems on machines with no operating system software. Our ultimate goal is to produce a higher level language which can be used with (possibly) some extensions to re-work an already existing operating system so that the higher level version, when compiled, can serve as the operating system on another machine. Our success in working toward this goal will depend, to a very large extent, in how well we are able to solve the I/O and system communication problems. Work in these two areas is in progress.

In addition, two other problems closely related to the above, but of considerable importance in their own right, must be dealt with carefully.

A. We anticipate that the process of accomodating the language to specific hardware may necessitate the addition of a number of machine dependent operators corresponding to those order codes of the source assembler language which are not easily expressed in terms of the nucleus language. Such a language extention could be accomplished in at least two ways:

1. By adding these operators directly to the syntax of the language at the desired operator precedence level.
2. By adding global functions written in terms of the nucleus language which simulate the action specified by the order code in question. These global functions could be defined in a common area that would be accessible to all programs in the operating system.

B. We will also need to be able to simulate the activity of certain hardware registers and indicators in the nucleus language extension. This could be done by adding new declarations to the syntax, or, equivalently, by designating global variables in the common area, corresponding to these registers and indicators.

We hope to automate the decompilation of the operating system as much as is practically possible. A working decompiler written by Barry Housel, a Purdue Ph. D. candidate, will be used to produce complete and automatic control flow and variable usage analysis for the programs to be decompiled, and to automatically coalesce register and core/register manipulations into single higher level statements whenever possible. This decompiler already performs some loop and loop index recognition functions, and work is under way to include procedure recognition and other features as well. The code generators of the decompiler are currently being changed to produce code for the nucleus language, and a compiler for the nucleus language implementation on the Microdata 1621 is being debugged.

VII. A FINAL WORD

It is important to realize that the nucleus language is intended as the target language for decompilation and not simply as a systems programming language. It is therefore quite simple in structure, especially with respect to the data structures that have been incorporated into its syntax. Neverthless, it is hoped that research directed toward the development of such a higher level language might also contribute to a better understanding of the code and data structures most often employed by experienced systems programmers. In addition, this research may also shed some light upon The answers to some of the questions dealing with such subjects as:

(1) the practical problems of translating a program with goto's to one without them;

(2) the extendability and transferability of the syntax of any language similar to the nucleus language;

(3) the relative merits of goto versus goto-less systems oriented programming languages;

(4) the general portability of small computer operating systems.

This, in itself, should be of considerable value in all efforts toward the development of higher level languages for systems implementation by helping us to gain new insights into the relations between programming languages, program structures, and operating systems.

Acknowledgements

I am indebted to Professors Victor B. Schneider and John E. Pomeranz for their many helpful comments and suggestions. I would also like to thank my colleague M. Dennis Mickunas for his assistance in the use of the Translator Writing System which he has developed at Purdue.

APPENDIX A

ARCHITECTURAL CHARACTERISTICS OF THE CLASS OF MACHINES M

1) byte oriented, sequential CPU with a limited instruction set

2) one or more operand registers, a program counter, and various index registers and condition indicators

3) one's complement logical operations

4) two's complement arithmetic operations

5) conditional transfers and/or skip instructions

6) a program interrupt facility of at least 1 level

7) instructions for accessing hardware status words

8) basic I/O instructions for byte-by-byte or cycle-steal I/O

APPENDIX B
LANGUAGE SYNTAX[1]

In the following, the syntax for <name>, <number>, <equid>, <procid>, <funcid>, grpid>, and <arraid> are assumed to be known.

```
      *****************
***** MAIN COMPONENTS *****
      *****************

<integer> ::= <number> /SAVNUM

<identifier> ::= <name> /SAVNAM

<program> ::= /PGMSTR <begin> <program block>
                   <end> /PGMEND

<begin> ::= begin | <name> : /LABELP <begin>

<program block> ::= <scope block> <definition block>
                  | <scope block>

<scope block> ::= <statement list>
                | declare /ENTDCL <declaration block> end
                    /EXTDCL ; <scope block>

<definition block> ::= <procedure definition>
                     | <definition block> <procedure definition>

<end> ::= end | end <name>


      **********************
***** SECONDARY COMPONENTS *****
      **********************

<statement list> ::= <statement> ; /CLRSTF
                   | <statement list> <statement> ; /CLRSTF

<declaration block> ::= <declaration>
                      | <declaration block> ; /CLRLTH <declaration>
```

[1]The capitalized identifiers preceded by slashes are the names of the routines defining the "rule of translation" for the production to which they are attached. This definition is given by the assembler language code that is generated by the named routine in the language compiler. For more details, refer to [Sc 69]. The specification of the rules of translation in the grammar is not yet complete.

```
<declaration> ::= <procedure declaration>
                | <length> <group declaration>
                | <length> <tuple>

<procedure definition> ::= <procedure heading>
                    <program block> <endprc>

<endprc> ::= end /PRCEND
           | end <procid> /PRNEND

<procedure heading> ::= procedure <procid> ; /PRCDEF
                | procedure <procid> /PRCDEF
                    ( <formal parameters> ) ; /SAVFCT

<formal parameters> ::= <identifier> /ADTFCT
                | <formal parameters> , <identifier> /ADTFCT


     *********************
***** DECLARATIVE ELEMENTS *****
     *********************

<tuple> ::= <tuple element>
          | <tuple> , <tuple element>

<tuple element> ::= <identifier> /VARDEF
                | <array designator> /ARRADS
                | <identifier> = /IVLNAM <simple expression>
                    /GENVAL
                | <identifier> ≡ /EQUNAM <simple expression>
                    /SETEQU
                | <array designator> = ( /GETBAS
                    <constant tuple> ) /GENRST

<constant tuple> ::= <constant tuple element> /ARRAIN
                | <constant tuple> , <constant tuple element>
                    /ARRAIN

<constant tuple element> ::= <constant>
                | <string>
                | <integer> /MULSAV * <constant>


     **********************************
***** PROCEDURE AND ARRAY DECLARATIONS *****
     **********************************

<procedure declaration> ::= procedure <identifier>
                    /PRCNAM
               |external /SETEXT procedure <identifier>
                    /PRCNAM
               | <procedure declaration> , <identifier>
```

```
                    /PRCNAM

<length> ::= byte /SETBYT
             | word /SETWRD

<array designator> ::= <identifier> /ARRNAM
                    [ <unsigned constant> /SETLTH ]


     *********************
***** GROUP SPECIFICATION *****
     *********************

<group declaration> ::= group <identifier> /TYPNAM
                    [ <component list> /TYPLTH ]
                    <group variable list>

<component list> ::= <component> /BNDTST
             | <component list> , <component> /BNDTST

<component> ::= <identifier> ( <unsigned constant>
                    /SETBLT )
             |<identifier> ( <unsigned constant>
                    /SETBLT ) ≡ <constant> /EQUSET

<group variable list> ::= <group variable>
             | <group variable list> , <group variable>

<group variable> ::= <identifier> /GRPNAM
             | <array designator> /GRPNAM


     ******************
***** BASIC STATEMENTS *****
     ******************

<statement> ::= <identifier> /LABELS : <statement>
             | <exit>
             | <repeat>
             | <assignment>
             | goto <simple expression> /GENJMP
             | move ( <variable> , <variable> ,
                  <unsigned constant> ) /GENMVC
             | <procedure call>
             | return /EXTPRC | return <name> /LDEXPC
             | <case clause> <alternatives> <else clause>
                  <endcase>
             | <if clause - then clause - 1 >
                  <statement list> <else clause> <endif>
             | cycle /ENTCYL : <statement list> <endcycle>
             | <for head> <statement list> <endfor>
             | <io statement>
```

```
                       | asm ( <unsigned constant> ,
                            <unsigned constant> ,
                            <unsigned constant> , <secondary> )
                       | wait
                       | <save condition>
                       | <on condition>
                       | <set condition>


     **********************
***** STATEMENT COMPONENTS *****
     **********************

<on condition> ::= on <condition> do : <statement list>
                       end

<set condition> ::= set <condition> to <cvalue>

<save condition> ::= save <condition> in <identifier>

<endfor> ::= end /GNEFOR
               | end <identifier> /GNENFR

<endcycle> ::= end /GNECYL
               | end <identifier> /GNENCY

<endif> ::= end /GNEIF
               | end <identifier> /GNENIF

<endcase> ::= end /GNECAS
               | end <identifier> /GNENCS

<io statement> ::= xio (<device>,<address>,<function>,
                       <modifier>)

<for head> ::= for /ENTFOR <identifier> = <expression>
                       /SAVBAS step <expression> /SAVSTP until
                       <expression> /SAVBND do /HDLGEN :

<alternatives> ::= <alternatives> <labels> /CASSEL
                       <statement list> /EXTCAS
               | <labels> /CASSEL <statement list> /EXTCAS

<labels> ::= <constant> /LABELB :
               | <labels> <constant> /LABELB :

<repeat> ::= repeat /RPTCYL
               | repeat <name> /RPTNCY
               | repeat /ENTRPT until <relation>
                       /GENRPU
               | repeat <name> /ENTNRP until
                       <relation> /GENRPU
```

```
<procedure call> ::= <procid> /PRCCAL
               | <procid> /PRCCAL ( <expression list> )
                    /NUMATS

<expression list> ::= <expression> /ADDACT
               | <expression list> , <expression> /ADDACT

<exit> ::= exit /EXITWO
               | exit <identifier> /EXITNA

<assignment> ::= <leftpart1> <expression> /GNSIND
               | <leftpart2> <expression> /GNSIND
               | <leftpart1> <assignment> /GNSIND
               | <leftpart2> <assignment> /GNSIND

<leftpart1> ::= <designator> = /GENSTA
               | <l-designator> /GENSTA

<leftpart2> ::= ( <expression> ) = /GENSTA

<if clause - then clause - 1> ::= <if clause> <then clause>

<if clause - then clause - 2> ::= <if clause> <then clause>

<case clause> ::= case /ENTCAS <simple expression> in

<if clause> ::= if /ENTIF <relation> /GENIF

<then clause> ::= then /LABGN1 :

<else clause> ::= else /LABGN2 : <statement list>


     *************
***** EXPRESSIONS *****
     *************

<expression> ::= <if clause - then clause - 2> <expression>
                    else : <expression> end
               | <simple expression>

<relation> ::= <simple expression> <relop>
                    <simple expression> /SIMREL
               | compare ( <variable> , <relop> ,
                    <variable> , <unsigned constant> )
                    /COMVAR
               | compare ( <variable> , <relop> , <string>
                    , <unsigned constant> ) /COMSTR
               | <condition> <cvalue>

<simple expression> ::= <term>
               | + <term>
```

```
                        | <m1> <term>
                        | <simple expression> + <term> /ADD
                        | <simple expression> - <term> /SUB
                        | <simple expression> ∨ <term> /OR
                        | <simple expression> or <term> /OR
                        | <simple expression> xor <term> /XOR

<relop> ::= < /LT | lt /LT
                        | > /GT | gt /GT
                        | = /EQ | eq /EQ
                        | ≠ /NE | ne /NE
                        | ≥ /GE | ge /GE
                        | ≤ /LE | le /LE


<m2> ::= -

<term> ::= <factor>
                        | <term> ∧ <factor> /AND
                        | <term> and <factor> /AND
                        | <term> * <factor> /MUL
                        | <term> / <factor> /DIV
                        | <term> mod <factor> /MOD

<factor> ::= <shiftor>
                        | ¬ <shiftor> /NOTGEN

<shiftor> ::= <secondary>
                        | <secondary> <shiftop> <constant> /SHFTPRC

<shiftop> ::= slo | sro | sre | slc

<secondary> ::= <primary>
                        | .  <primary> /GENIND
                        | <function call>

<function call> ::= <funcid> /FNCALL
                        | <funcid> /FNCALL
                              (<expression list>) /NUMATS

<primary> ::= <designator>
                        | <r-designator>
                        | ( <expression> )
                        | <unsigned constant> /LOADCN
                        | (<assignment>)

<designator> ::= loc <variable> /GENADR
                        | <grpid> /LOADGP

<l-designator> ::= <variable> = /GENADR
                        | <variable> ( <unsigned constant> /LFTBIT
                              → <unsigned constant>/RHTBIT ) /VBTOMS
```

```
                  | <identifier> of <group variable> /GPTOMS

<r-designator> ::= <variable> /LOADVL
                  | <variable> ( <unsigned constant> /LFTBIT
                       → <unsigned constant>/RHTBIT ) /MSTOVB
                  | <identifier> of <group variable> /MSTOGP

<variable> ::= <identifier> /LOADV
                  | arraid /ARRREF [ <expression> ]

<constant> ::= <unsigned constant>
                  | <m2> <unsigned number> /NEGGEN

<unsigned constant> ::= <unsigned number>
                  | <equid> /LOADEQ

<unsigned number> ::= <integer>
                  | ↑ /SAVHEX

<condition> ::= oflow | intrp0 | intrp1 | intrp2
                  | intrp3 | intrp4 | cstop

<cvalue> ::= on | off

<device> ::= <designator>

<address> ::= <designator>

<function> ::= <designator>

<modifier> ::= <designator>

<string> ::= ↓ /SAVSTR
```

BIBLIOGRAPHY

[AM 71] Ashcroft, Edward and Zohar Manna, "The Translation of GOTO Programs to WHILE Programs," IFIP, 1971, Booklet TA-2, pp. 147-52.

[Br 72] Brown, P. J., "Levels of Language for Portable Software, " CACM (15,12), December, 1972, pp. 1059-62.

[CH 71] Clark, B. L., and J. J. Horning, "System Language For Project SUE," SIGPLAN (6,9), October, 1971, pp. 79-85.

[Co 69] Coulouris, G. F. "A Machine Independent Assembly Language for Systems Programs," Annual Review in Automatic Programming (6,2), 1969, pp. 89-104.

[Co 73] Cox, George, private communication, 1973.

[Ho 73] Housel, Barry C., A Study of Decompiling Machine Languages Into High-Level Machine Independent Languages, Ph. D. Thesis, Purdue University, Department of Computer Sciences, West Lafayette, Indiana, August, 1973.

[IBM 68] IBM Corporation, IBM 1130 Disk Monitor System, Version 2, Programmer and Operators Guide, File Number 1130-36, Form C26-3717-4,1968.

[Ka 72] Katzan, Harry, A PL/I Approach to Programming Languages, Auerbach, Inc, Princeton, N. J., 1972.

[KF 71] Knuth, Donald E., and Robert Floyd, "Notes on Avoiding GOTO Statements," Information Processing Letters (1,1), February, 1971, pp. 23-31.

[La 69] Lang, Charles A., "SAL - Systems Assembly Languages, " SJCC, 1969, pp. 543-556.

[Le 72] Leavenworth, B. M. (editor), "The GOTO Controversy,"

[MHW 70] McKeeman, W. M., J. J. Horning and D. B. Wortman, SIGPLAN Notices (7,11), November, 1972, pp. 54-91. A Compiler Generator, Prentice-Hall Inc., Englewodd Cliffs, N. J., 1970.

[RH 72] Rain, Mark and P. Holager,, "The Present ... Word about Labels In MARY," Machine Oriented Languages Bulletin Number 1, Norwegian Institute of Technology, University of Trondheim, Norway, 1972, pp. 19-26.

[Ri 69] Richards, Martin, "BCPL:A Tool for Compiler Writing and System Programming," SJCC, 1969, pp. 557-566.

[Sa 69] Sammett, Jean, Programming Languages: History and Fundamentals, (especially the sections on NELIAC and JOVIAL), 1969.

[Sc 69] Schneider, V. B., " A System For Designing Fast Programming Language Translators," Proc. AFIPS Conf., Volume 34, SJCC 1969, pp. 777-792.

[Sh 63] Shaw, J. C., "JOVIAL - A Programming Language for Real Time Command Systems," Annual Review in Automatic Programming, 1963, pp. 53-120.

[We 72] Wegner, Eberhard, , "A Hierarchy of Control Structures, " Machine Oriented Languages Bulletin, Number 1, Norwegian Institute of Technology, University of Trondheim, Norway, 1972, pp. 5-12.

[WH 66] Wirth, N. and C. A. R. Hoare, "A Contribution to the Development of ALGOL," CACM (9,6), June, 1966, pp. 413-431.

[Wi 68] Wirth, N., "PL360, A Programming Language for the 360 Computers," JACM (15,1), January, 1968, pp. 37-74.

[Wi 71] Wirth, N., "The Programming Language PASCAL, " Acta Informatica 1, 1971, pp. 35-63.

[Wu 71] Wulf, W. A., "Programming Without the GOTO," IFIP, 1971, Booklet,TA-3, pp. 84-88.

[Wu 71a] Wulf, W. A. et. al., "Reflections on a Systems Programming Language," SIGPLAN (6,9) October, 1971, pp. 42-49.