Lehigh University
Lehigh Preserve

Theses and Dissertations

1-1-1979

An implementation of a coroutine mechanism in a block structured language.

Allan R. Frank

Follow this and additional works at: http://preserve.lehigh.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation
Frank, Allan R., "An implementation of a coroutine mechanism in a block structured language." (1979). *Theses and Dissertations.* Paper 1864.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

AN IMPLEMENTATION OF A COROUTINE MECHANISM IN A BLOCK STRUCTURED LANGUAGE

by

Allan R. Frank

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1979

ProQuest Number: EP76136

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest EP76136

Published by ProQuest LLC (2015). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

Sept. 9, 1979
(date)

Professor in Charge

Chairman of the Division

TABLE OF CONTENTS

Title: An Implementation Of A Coroutine Mechanism In A Block Structured Language

Author: Allan R. Frank

ABSTRACT

The purpose of this paper is to document and discuss an implementation of a coroutine mechanism in a block-structured language. PASCAL-S was utilized as the language base. Though it is only a subset of the more powerful block-structured language, PASCAL, the PASCAL-S coroutine implementation effort was facilitated greatly by the relative simplicity of the compiler structure while still providing a sufficiently powerful repertoire of features.

Specifically, the implementation effort consisted of identifying, designing, implementing, and testing the syntactic and semantic constructs necessary to support a coroutine capability. The language additions/modifications encompassed five language extentions; COLINK declaration, COPROC definition, RESUME statement, DETACH statement, and NEWTOP statement.

The COLINK declaration is placed in a program following any appropriate CONST, TYPE, or VAR declarations. For example, just as a VAR declaration delineates those identifiers which represent program variables, so a COLINK declaration delineates coroutines.

The coroutines look much like standard PASCAL procedures except, instead of being identified by the keyword PROCEDURE, they

are preceeded by the word COPROC. For purposes of the PASCAL-S implementation, any reference to COPROC is synonomous with coroutine.

There are, of course, very critical syntactic and semantic differences between a PROCEDURE and a COPROC. These differences, as well as attendant similarities are discussed within the body of the paper.

The remaining three control structures, RESUME, DETACH, and NEWTOP, are all statements added to the PASCAL-S repertoire. They apply specifically to manipulation and control of COPROC (coroutine) actions. The RESUME verb is very similar to a procedure call except that it initiates the invocation of a COPROC from the body of another COPROC. This is the primary means of achieving coroutine logic flow. Here, instead of always entering at the top of its code, a COPROC, when invoked by a RESUME, will begin execution at the point of its last exit.

The DETACH statement is an artificial means of transferring control from a coroutine cycle (COPROC) to the original calling procedure. This is much like a subroutine return in FORTRAN, or a natural exit from a called procedure in PASCAL except in this case, the DETACH allows an intermediate exit at any point in a COPROC. Transfer of control is not to the calling COPROC which may have issued the RESUME previously, but rather to the coroutine procedure body which invoked the COPROC originally.

Finally, the NEWTOP statement allows the calling program to reset any particular COPROC entry point back to the physical beginning of its code. In many cases, it may be desirable to ensure that a subsequent COPROC activation begins execution at the top of its program code (much like a call to a subroutine) rather than some indeterminate point. Upon issuing a DETACH from a COPROC, its new entry point is set to the point of the intermediate exit. Invoking a NEWTOP statement performs a selective reset of a COPROC entry point.

I. INTRODUCTION

The purpose of this paper is to document and discuss an implementation of a coroutine mechanism in a block-structured language. In order to facilitate the achievement of the implementation objectives, the choice was made to utilize a minimal PASCAL subset embodied in the PASCAL-S interpreter [5]. This language subset includes many of the features of the larger compiler, however it does not include such features as SETS, variant RECORDS, POINTERS, GOTOs, PACKED ARRAYs, and the like. In addition, PASCAL-S is limited to such data types as INTEGER, REAL, BOOLEAN, and CHARACTER. The major benefit of utilizing this PASCAL implementation is the simplicity of its compiling actions, P-machine pseudo-code generation and interpretation.

The implementation effort consisted of identifying, designing, implementing, and testing the syntactic and semantic constructs comprising the necessary elements of the coroutine mechanisms. This report serves to record the results of these efforts. For purposes of presentation, the following topical headings appear in subsequent sections of this paper:

- II. COROUTINES -- A GENERAL OVERVIEW -

  Here, a brief discussion of the nature of coroutine structures is given, with a specific focus on describing both its practical and theoretical qualities as they may relate to implementation on sequential automata. The description of coroutines will provide the basis of understanding from which the remainder of the paper is based.

- III. OBJECTIVES OF IMPLEMENTATION -

  This section will delineate the specific objectives relative to the project to implement coroutine structures in PASCAL-S. These objectives or goals were formulated in order to provide a set of broad guidelines useful in the planning and design stages of the project.

- IV. SYNTAX AND SCOPE -

  This section provides a detailed description of each of the syntactic structures added to the PASCAL-S interpreter which were necessary to implement the coroutine mechanism. In addition to a comprehensive explanation of language extensions, syntax, and use, the scope rules are also defined

in order to provide potential users with an indepth understanding of the nature and limitations of the new features.

- V. IMPLEMENTATION DESCRIPTION — PASCAL-S MODIFICATIONS -

Following the comprehensive explanation of the necessary language extensions as presented in the previous Section IV, this section provides a detailed insight into the nature and extent of programmed changes/additions to the PASCAL-S interpreter. Here, an extensive description of the physical methods of achieving the coroutine features, as presented, is given, and thus provides the necessary documentation from which future enhancements to the language structures can be made without disturbing the concepts and methods employed in this project.

- VI. POTENTIAL FOR EXPANSION -

In this section a brief exposition is presented which discusses the practicality and applicability of future enhancements to this present PASCAL-S coroutine implementation project. Special attention is given to the limitations of the coroutine extensions developed as a result of this project, and insight is given into the nature of the effort required in order to further enhance these structures.

II. COROUTINES — A GENERAL OVERVIEW

The notion of a coroutine has been generally attributed to the works of Conway [1]. Coroutines are modified subroutines with the capability of maintaining intermediate entry and exit points. Normally, a subroutine is a sequence of code which is called by the main program body. Entry to the subroutine always occurs at the top of its code (first line of code). Regardless of what point the subroutine exits and transfers control back to the calling program, subsequent calls to it will always cause re-entry to be made at its top of code.

A coroutine, on the other hand "remembers" the point at which it was last exited. Subsequent calls to a coroutine will cause re-entry to the code after the point it had last been exited. Thus, the first time a coroutine is called, it performs exactly like a subroutine; it begins execution at the top of its code. However, when it is exited, it records the place at which it should next resume execution. Should it again be invoked, transfer of control will proceed to the point at which it has recorded the previous intermediate exit. If, previously, the exit had been at its natural end point, a subsequent call to the coroutine would again cause execution to begin at the top of its code.

FIGURE 2.1

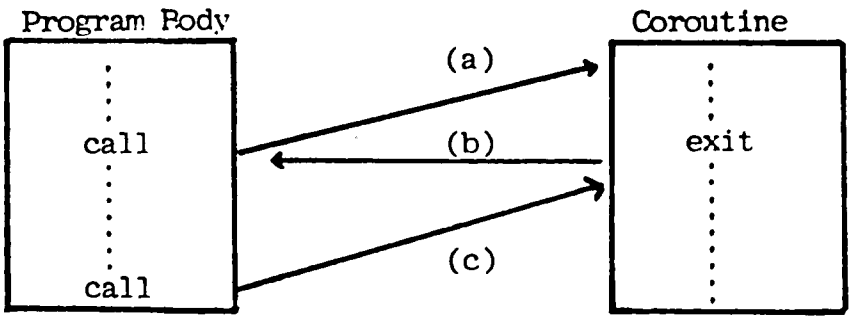


Figure 2.1 illustrates the coroutine actions described previously. The initial call to a coroutine (a) causes entry at the physical top of its code. Upon an intermediate exit (b) control is transferred back to the point after the initial call. Finally, a subsequent call to the coroutine (c) causes re-entry to occur at the point after the last intermediate exit.

As noted by Conway, the coroutine concept may be utilized for any program which can be shown to be separable. A discussion of separability is outside the scope of this project; however it is important to note that under this condition, separable program modules can be translated into coroutines. These coroutines then interact as if they are each the main program calling other subroutines (coroutines).

Developing further the coroutine concept in a block-structured environment, Wang and Dahl [2] pointed out that coroutines could be used to simulate parallel processes even though the mechanism

itself is implemented in a sequential program. Their discussion of the "quasi-parallel" Simula 67 language was the key to the development of the PASCAL-S implementation to be discussed subsequently.

Coroutines can be considered to be sequential processes performed in parallel. This attribute brings with it several potential implications. First, if a program can be broken into a set of distinct coroutines, it could be implemented on a set of parallel processing computers, each executing a specific coroutine. A sufficiently comprehensive compiler could be developed to compile individual coroutine modules for the individual processors. Communication would be accomplished via a controlled data structure which ensures the proper synchronization of the coroutine modules.

The second implication of the quasi-parallel nature of coroutines is their ability to interact in such a way as to facilitate the implementation of multi-pass programs using a single-pass structure. This capability was amply demonstrated by Conway in his design of a one-pass COBOL compiler which was built around a number of coroutine segments.

The coroutine implementation developed in the block-structured language PASCAL-S contains the necessary constructs to perform coroutine actions. The design was based upon the Conway,

and Wang and Dahl works which describe in detail, the theory and use of coroutine structures. Hopefully, subsequent experimentation with this new coroutine implementation will provide further insights into the practicality of coroutines for a variety of algorithmic designs.

III. OBJECTIVES OF IMPLEMENTATION

In order to study the applicability and practicality of utilizing coroutine mechanisms in a block-structured language as universally supported as PASCAL, this project was designed to provide a relatively straight-forward and flexible means for application experimentation. Specifically, the coroutine augmentation of the PASCAL-S interepreter was intended to achieve the following general objectives:

- Definition of appropriate syntactic structures necessary for effective implementation utilizing formats and language syntax compatable with that currently supported by the PASCAL-S interpreter;
- Definition of appropriate procedural and data scope necessary for effective implementation while maintaining the scope rules currently supported by the PASCAL-S interpreter;
- Support of all language facilities currently implemented within the PASCAL-S interpreter:
- Use of implementation techniques (i.e. programming) which can be readily modified to facilitate the future expansion of any coroutine features implemented in this current version; and

- Design of all program code necessary to support the coroutine features in a manner which ensures that the aesthetic balance of current PASCAL-S program code is not disturbed.

IV. SYNTAX AND SCOPE

A. Syntax Overview

The purpose of this section is to discuss in detail the syntactic structure of the coroutine implementation performed upon the PASCAL-S interpreter. This discussion will center on a description of the various syntax enhancements/modifications made to the translator. In addition, where appropriate, the scope rules and limitations are identified. Appendix A details in an illustrative format the syntax diagrams for this enhanced PASCAL-S language.

Specifically, this section documents the nature of the language additions identified as follows:

- COLINK
- COPROC
- RESUME
- DETACH
- NEWTOP

The COLINK declaration is placed in a program following any appropriate CONST,TYPE, or VAR declarations. For example, just as a VAR declaration delineates those identifiers which represent program variables, so a COLINK declaration delineates coroutines.

The coroutines look much like standard PASCAL procedures except, instead of being identified by the keyword PROCEDURE, they are preceded by the work COPROC. For purposes of the PASCAL-S implementation, any reference to COPROC is synonomous with coroutine.

There are, of course, very critical syntactic and semantic defferences between a PROCEDURE and a COPROC. These differences, as well as attendant similarities will be discussed in a latter subsection.

The remaining three control structures, RESUME, DETACH, and NEWTOP, are all statements added to the PASCAL-S repertoire. They apply specifically to manipulation and control of COPROC (coroutine) actions. The RESUME verb is very similar to a procedure call except that it initiates the invocation of a COPROC from the body of another COPROC. This is the primary means of achieving coroutine logic flow. Here, instead of always entering at the top of its code, a COPROC, when invoked by a RESUME, will begin execution at the point of its last exit.

The DETACH statement is an artificial means of transferring control from a coroutine cycle (COPROC) to the original calling procedure. This is much like a subroutine return in FORTRAN, or a natural exit from a called procedure in PASCAL except in this case, the DETACH allows an intermediate exit at any point in a

COPROC. Transfer of control is not to the calling COPROC which may have issued the RESUME previously, but rather to the coroutine procedure body which invoked the COPROC originally.

Finally, the NEWTOP statement allows the calling program to reset any particular COPROC entry point back to the physical beginning of its code. This would be useful since upon issuing a DETACH from a COPROC, its new entry point is set to the point of the intermediate exit. In many cases, it may be desirable to ensure that a subsequent COPROC activation begins execution at the top of its program code (much like a call to a subroutine) rather than some indeterminate point. Invoking a NEWTOP statement performs a selective reset of a COPROC entry point.

The remainder of this section will describe in detail the nature and limitations of the five previously cited syntactic additions to the PASCAL-S interpreter. It is these five new declaration and control verbs which provide the necessary mechanisms to support a coroutine structure within PASCAL-S.

B. COLINK

As described briefly previously, the COLINK declaration is intended to identify a COPROC prior to its formal declaration.

This is similar to declaring variables (VAR) prior to their use. In the case of a COPROC, this requirement eliminates the need for a complicated forward referencing mechanism which would have been necessary to resolve addressing relative to a COPROC referencing a neighboring COPROC further down in the program.

Syntactically, a COLINK declaration identifies all COPROC bodies within the subsequent block activation level. This is accomplished by inserting the following declaration for each COPROC:

```
COLINK   coprocname1;
         coprocname2;
                 .
                 .
                 .
         coprocnameN;
```

Each coprocname is the symbolic identifier to be used for the COPROC declarations to follow. Appendix A contains the syntax diagram for the COLINK declaration.

The COLINK declaration is contained in the declaration part of the PASCAL-S program and/or within its constituent procedures and coroutines. It would only be used if a COPROC is to be utilized. If none are to be used, the COLINK declaration section must not be included.

The standard PASCAL-S implementation has no forward referencing capability. This in itself impacts program development in

only a few selective cases. Due to the one-pass nature of the PASCAL-S interpreter's compiling actions, a procedure must have been identified and catalogued in the appropriate symbol table(s) prior to reference by a call to that procedure.

FIGURE 4.1

```
      PROGRAM Test(output);
      VAR
        I:  Integer;
(1)   PROCEDURE  Adder;
        BEGIN
          I: = I + 2;
        END(*Adder*):
      BEGIN
        I: = 10;
        Writeln(≠ here is a number ≠, I);
(2)     Adder;
        Writeln(≠ here is 2 + that number ≠, I);
      END(*main*).
```

Figure 4.1 illustrates a normal PASCAL-S program where a procedure "Adder" is subsequently referenced in the main body of the program. PASCAL-S will have entered the procedure name (1) in its symbol table prior to its compiling the reference to it (2). This is possible due to the one-pass downward left-right direction of compiling actions.

In the case of the COPROC implementation, one COPROC must reference another at the same block level. This is illustrated in Figure 4.2 below.

FIGURE 4.2

```
        COPROC A;
          BEGIN
            .
            .
(1)         RESUME(B);
            .
            .
          END(*A*);
(2)     COPROC B;
          BEGIN
            .
            .
            RESUME(A);
            .
            .
          END(*B*);
```

As indicated, within the body of COPROC A, a reference (1) is made to COPROC B, before COPROC B has been declared to the compiler (2). This type of forward referencing will always occur when COPROC to COPROC communication is used.

To alleviate this, the COLINK declaration must be used to explicitly identify any subsequent COPROC to appear in the following block activation level. Figure 4.3 illustrates the use of this COLINK feature.

FIGURE 4.3

```
PROGRAM  Test;
VAR
  I:  Integer;
COLINK
  A;
  B;
COPROC  A;
  BEGIN
    .
    .
    .
    RESUME(B);
    .
    .
    .
  END(*A*);
COPROC  B;
  BEGIN
    .
    .
    .
    RESUME(A);
    .
    .
    .
  END(*B*);
BEGIN(*MAIN*)
  .
  .
  .
  A;
  .
  .
  .
END(*MAIN*).
```

The COLINK declaration provides a simple mechanism to facilitate the forward referencing of COPROC B (within COPROC A), prior to the COPROC declaration.

The explicit definition of a COPROC not only provides a means to implement forward referencing, but it also facilitates

the validation of proper scope relations between a COPROC and PROCEDURE body. As will be discussed in greater detail in the subsequent subsection on COPROC declarations, COPROC to COPROC communication is limited to direct interaction at the same block level. In other words, one COPROC cannot communicate as a coroutine with another COPROC which is nested within the first. They must both be at the same activation level; whereas nesting would imply that the first coroutine would have a level lower than the second.

FIGURE 4.4

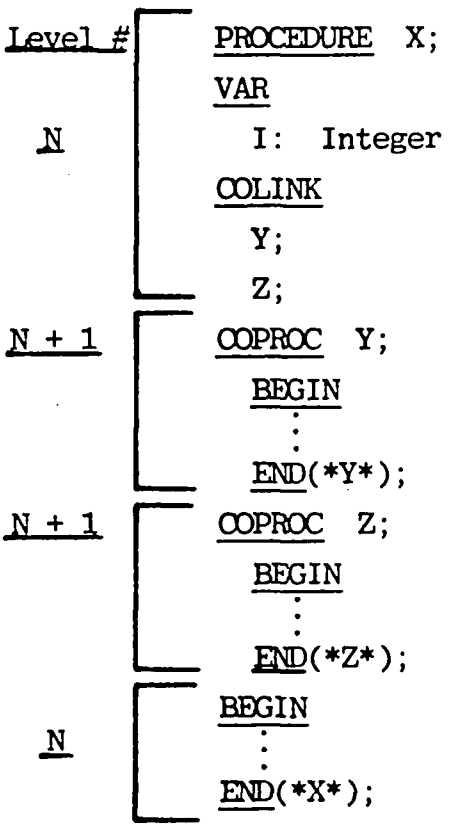
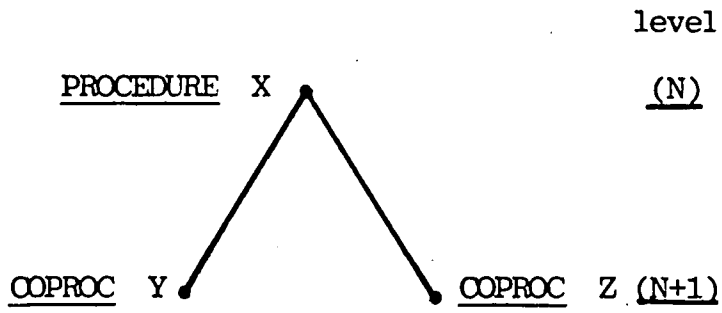


Figure 4.4 illustrates a procedure which contains two COPROC declarations. Notice that the COLINK definition appears in the declaration body of the parent procedure (Level N), identifying those COPROC bodies existing at level N+1.

FIGURE 4.5



A tree diagram can be utilized to generalize the scope legality of COLINK declarations. Figure 4.5 graphically depicts the scope relation of the program in Figure 4.4. The PROCEDURE X (parent node), contains two children nodes, each of which is a COPROC.

Rule: The COLINK declaration must appear within the declaration body of the parent node.

FIGURE 4.6

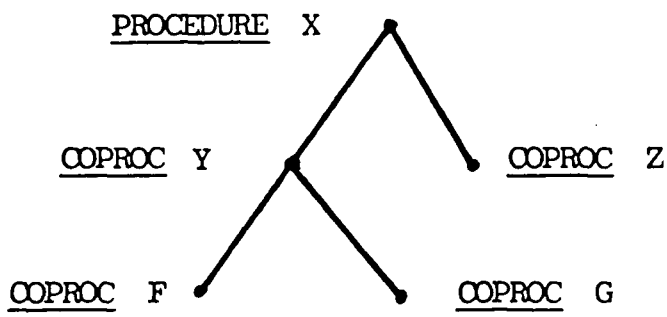


Figure 4.6 illustrates a tree structure representing a hypothetical PROCEDURE X which contains four COPROC bodies (Y, Z, F, G). However, COPROC F and G are both contained in COPROC Y. In this case, applying the rule underlined previously, a COLINK declaration would be made in PROCEDURE X identifying COPROC Y and Z. Also, a COLINK declaration would be made in COPROC Y identifying COPROC F and G, (Y is the parent of F and G).

Figure 4.7 is a program translation of the tree structure represented in Figure 4.6, and illustrates the proper application of COLINK declarations (see following page).

FIGURE 4.7

```
PROCEDURE  X;
VAR
  I:  Integer;
COLINK  Y; Z;
  COPROC  Y;
  COLINK  F: G:
    COPROC  F;
      BEGIN
        .
        .
      END(*F*);
    COPROC  G;
      BEGIN
      .
      .
      END(*F*);
    BEGIN
    .
    .
    END(*Y*);
  COPROC  Z;
    BEGIN
    .
    .
    END(*Z*);
BEGIN
.
.
END(*X*);
```

C. COPROC

The control structure which in itself represents the implementation of coroutines in PASCAL-S is embodied in the variant procedure type COPROC. A COPROC is very similar to a standard PASCAL procedure, and in fact, takes very much the same syntactic format of a procedure with only a few very distinct exceptions. Appendix A contains the syntax diagram for a COPROC declaration. Semantically, a COPROC differs greatly from a normal procedure with its ability to exit and re-enter at intermediate points within its procedure body. Normally, every call to a PASCAL procedure will always invoke execution of its code at the top of the procedure body and exit at its natural bottom. On the other hand, a coroutine-like procedure (COPROC) will begin execution at the top of its code only on its initial call. Subsequent calls to it (via a RESUME) will cause execution to begin at the line after its last point of exit.

FIGURE 4.8

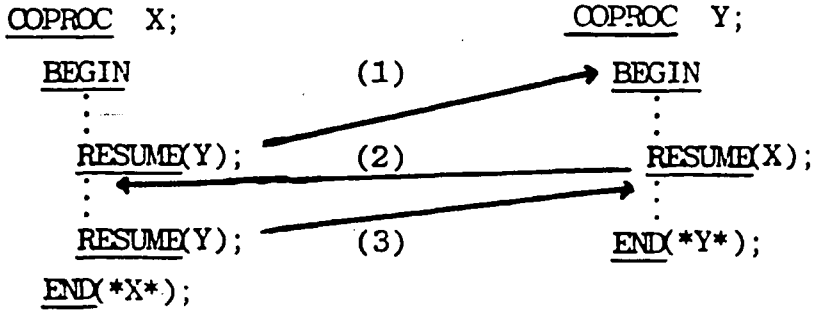


Figure 4.8 above provides an illustrative example of two COPROC bodies communicating with each other. The initial call by COPROC X to COPROC Y via the RESUME statement (1) invokes entry at the top of COPROC Y. The subsequent call to X from within Y (2) re-enters X at the last point of its intermediate exit (after the first RESUME). Finally, the subsequent RESUME of Y (3) will cause entry at the point of its last exit.

An individual COPROC has many of the same properties as a PROCEDURE including:

- Nesting
  - A COPROC may be nested within a PROCEDURE
  - A COPROC may be nested within another COPROC
  - A COPROC may contain a PROCEDURE nested within it
- Calling
  - A COPROC may call another COPROC (at same level via RESUME)
  - A COPROC may call a PROCEDURE (via normal procedure call)
  - A COPROC may be called by the PROCEDURE and/or COPROC which contains it (via normal procedure call)

- Statements
  - A COPROC may contain any statement or combination of statements that could be found in PROCEDURE bodies.

However, a COPROC differs from the standard PROCEDURE in that it does not allow the following constructs to be defined within it:

- Parameter List
- TYPE declaration
- VAR declaration
- CONST declaration
- Recursive call

In general, as should be evident from the exclusions noted above, the current COPROC implementation does not allow the definition of any type of local variables (however, this does not preclude a procedure nested within it from declaring variables local to it) This restriction limits the interaction between COPROC procedure bodies to the manipulation of data elements defined globally (or within the procedure body containing them). This method of data addressing significantly reduces the overhead which would have been required if local variables had to be continually swapped on the run-time stack. The subsequent section discussing the specifics of the PASCAL-S implementation

provides further insight into the mechanism of run-time overhead.

For purposes of explanation, it may be useful to further discuss the communication between a COPROC and a calling PROCEDURE. Coroutine-like actions are invoked only between two or more neighboring COPROC bodies. The communication between the group of inter-related COPROC bodies can be conveniently called a "cycle".

FIGURE 4.9

COPROC CYCLE

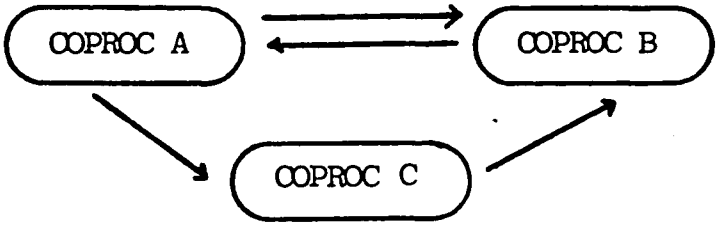


Figure 4.9 graphically illustrates a COPROC cycle. This cycle actually represents the inter-COPROC calls initiated via RESUME statements.

In order to initiate a cycle, a COPROC must initially be called by the main program or by the PROCEDURE/COPROC/FUNCTION containing it. This initial call is performed like a standard procedure call; simply invoke the COPROC name. When any individual COPROC in the cycle reaches its natural exit point, END, transfer of control will be returned to the point after this initial call.

Thus, if only one COPROC were defined in a cycle, it would function exactly like a PROCEDURE when it is called using the standard PASCAL procedure call syntax.

D. RESUME

The RESUME statement is the principle means of initiating coroutine actions. When invoked within a COPROC it transfers control to the respective entry point of the selected COPROC within the cycle. The statement immediately following the RESUME becomes the new entry point for that COPROC. A subsequent call (RESUME) to it by another COPROC will transfer control to this new entry point. Thus, COPROC to COPROC communication via the RESUME operation is the sole means of creating coroutine actions.

Appendix A illustrates the syntax flow of a RESUME statement. In addition, the following rules apply to its use:

1. The COPROC object of a RESUME must have been declared by the same COLINK declaration as the COPROC in which the RESUME resides.
2. Only a COPROC can be called via a RESUME statement; A PROCEDURE or FUNCTION is invoked using the standard PASCAL call.

Rule (1) further enforces the restriction that for COPROC to COPROC communication each COPROC must be defined at the same block level.

FIGURE 4.10

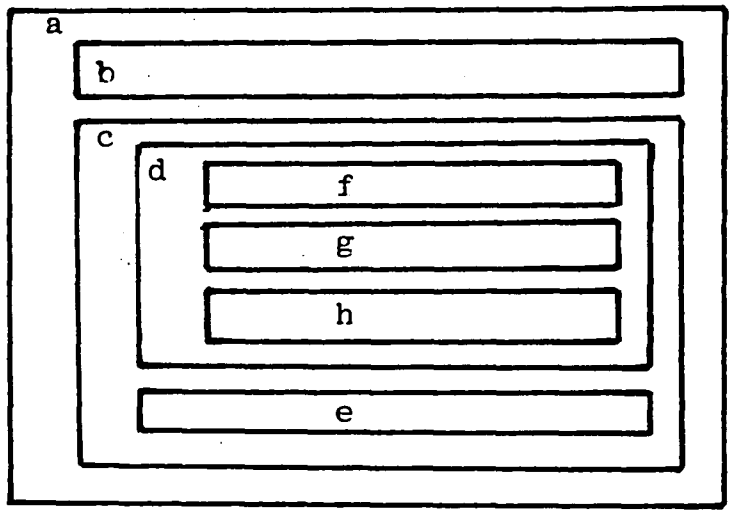


Figure 4.10 illustrates a scope diagram which is useful in summarizing the legal scope relations of COPROC and PROCEDURE/FUNCTION interaction. Under one scenario, "a", "b", "c", and "f" are PROCEDURE definitions with "d", "e", "g", and "h" defined as COPROC bodies. Since the above example includes nesting, the following chart summarizes the legal scope of inter-COPROC communication:

COPROC	can RESUME	COPROC
d		e
e		d
g		h
h		g

In addition, the COPROC bodies can call the following procedures via the standard PASCAL call:

COPROC	can call	PROCEDURE
d		a,b,c
e		a,b,c
f		a,b,c,f
g		a,b,c,f

It should be pointed out that even though each of the COPROC definitions indicated above can call PROCEDURE bodies a,b, and c, such an occurrence could lead to recursion. This would be possible since the COPROC bodies are nested within those procedures. Though the current implementation will allow a recursive call, anomolous results may result if it causes re-entry to a COPROC. This is because COPROC entry/exit points are not stored on a stack, but rather within a static symbol table entry. The subsequent section on the technical implementation

approach will shed further light on this limitation.

The following summary depicts the legal PROCEDURE to COPROC calls which are legal within the context of the previous example:

PROCEDURE	can call	COPROC
a		-
b		-
c		d,e
f		-

In this case,"c" is the only PROCEDURE allowed to access a COPROC. This applies to a previously delineated rule which stated that only the parent procedure containing COPROC(s) may invoke the cycle. In this case, COPROC "d" and COPROC "e" are the children of PROCEDURE "c".

There are several COLINK declarations which would appear if the previous Figure 4.10 were to be programmed. The following summary delineates the required declarations:

in PROCEDURE/COPROC	declare	COLINK
a		-
b		-
c		d;e;
d		g;h;
e		-
f		-
g		-

The above example may be beneficial in relating the previous sub-section describing the COLINK declaration with the other PASCAL-S scope definitions.

E. DETACH

The DETACH statement is a mechanism included in this PASCAL-S implementation which facilitates transfer of control from a COPROC cycle to the procedure body which invoked the initial call to that cycle. Its properties are quite similar to that of a natural COPROC exit. When a COPROC in a cycle reaches its natural END, control transfers to the point in the procedure body which initiated the cycle. A DETACH can be placed anywhere within the body of a COPROC. Upon encountering the DETACH, control is transferred to the point after the initial call in the applicable procedure body.

As should be evident, the DETACH imitates the semantics of the final END in a COPROC. However, there is one important difference. When a COPROC reaches its natural END point, its re-entry point for a subsequent RESUME is reset back to the physical top of its code. On the other hand, a DETACH does not reset its re-entry point to the top, but rather to the statement after the DETACH. This is, in effect, similar to an intermediate exit. This feature may be useful in allowing a return to the controlling procedure body and re-entry into

the COPROC at the point of DETACH. Appendix A illustrates the rather straightforward syntax of a DETACH statement. As alluded to earlier, it may be placed anywhere within a COPROC procedure body.

Care must be taken when using a DETACH especially if re-entry into the COPROC cycle is attempted. To further clarify this assertion requires a brief explanation of the technical implementation approach (a more detailed discussion will follow in Section V). In order to effectuate a coroutine mechanism it was necessary to expand the use of the run-time stack. Each time a COPROC is invoked via a RESUME, the COPROC issuing the RESUME has its current stack values frozen on the stack. Thus, if an intermediate exit is performed from within a FOR, REPEAT, or WHILE loop, their control variables, which are wholly stack dependent, are saved. When re-entry is invoked into the COPROC, those frozen stack values are restored onto the top of the stack again. When a DETACH or natural END is encountered, the stack values are deactivated for each COPROC in the cycle. Thus, if re-entry is attempted into a COPROC which previously issued a DETACH within a stack-dependent control loop (i.e. FOR, REPEAT, WHILE statements), catastrophic program failure will surely result.

FIGURE 4.11

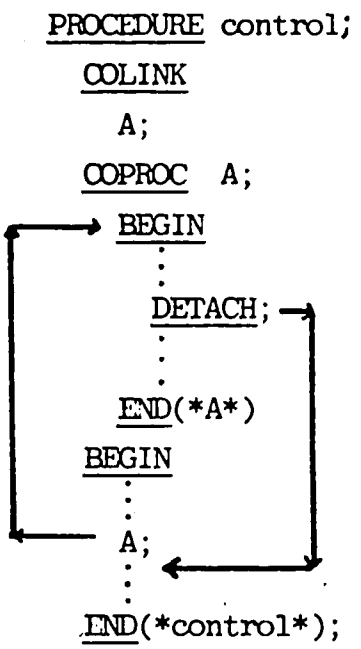


Figure 4.11 outlines a program skeleton utilizing the DETACH. Though the flow lines do not indicate it, if "A" was again invoked in the main program as a consequence of the DETACH, control would transfer to the statement immediately after that DETACH. This allows a calling procedure to communicate with a single COPROC cycle maintaining the coroutine qualities of that COPROC.

FIGURE 4.12

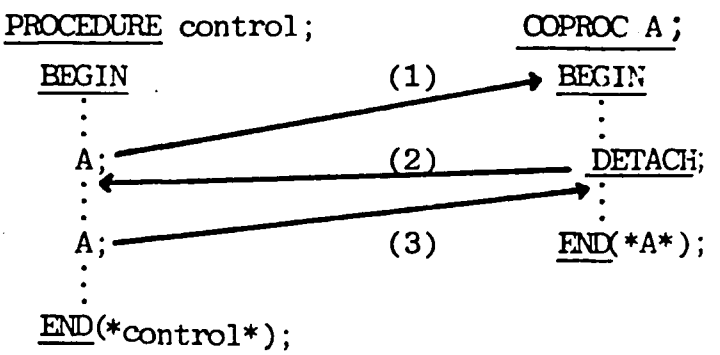


Figure 4.12 further illustrates this capability by rearranging the previous example. The initial call to "A" (1) causes entry to the COPROC at its physical top. The subsequent DETACH (2) transfers control back to the point after the initial call in PROCEDURE "control". Finally, a subsequent call to "A" (3) re-enters the COPROC at the point after the DETACH. This parallels the intermediate entry which could have ensued if it had been invoked by a RESUME from another COPROC within the cycle.

F. NEWTOP

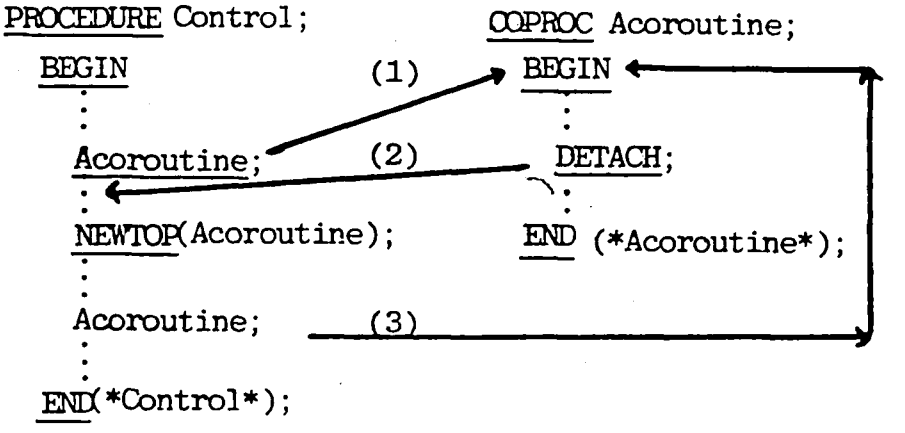
The NEWTOP statement provides the means to reset any COPROC entry point back to the physical top of its code. It may be usefully performed by the procedure block which initiates the COPROC cycle call in order to ensure that each COPROC is properly

initialized. Since a COPROC cycle can be exited if the natural END of any COPROC in that cycle is reached or if a DETACH is performed, the programmer may find it desirable in some cases to issue a NEWTOP for each COPROC in the cycle prior to re-calling that cycle. Otherwise, each COPROC in the cycle will have undefined entry points which were set by previous RESUME, DETACH, and/or END statements. Appendix A graphically depicts the syntax diagram of the NEWTOP statement. The COPROC identifiers which may be included within the NEWTOP verb must have been declared by a COLINK in the declaration section of the parent PROCEDURE. In general, the scope of a NEWTOP statement is identical to an initial call to a COPROC by the procedure body in which it is contained. If that particular procedure block can legally invoke a COPROC via the standard procedure call (see previous discussion on the COPROC definition), then it can also issue a NEWTOP for it. This will always be the parent PROCEDURE, FUNCTION, or COPROC in which the child COPROC referenced by the NEWTOP resides.

Referring back to the previous discussion of the DETACH statement, it must again be stressed that care must be taken when performing an exit from inside a FOR, REPEAT, or WHILE loop within a COPROC. This is because all loop control variables stored on the run-time stack are lost upon exit. Thus, a mechanism must be provided to ensure that a subsequent call to that COPROC will

not cause re-entry into the context of the previously defined control loop. To do so, would inevitably lead to anomolous program behavior. Issuing a NEWTOP to reset the entry points of each applicable COPROC will ensure that re-entry will begin back at the top of the routines. Since a NEWTOP verb affects the point of COPROC entry, it must be used with care and its ramifications understood. This is especially critical in cases where the intermediate entry point to a given COPROC has been modified from that which would have existed had the NEWTOP not been invoked.

FIGURE 4.13



In summary, Figure 4.13 illustrates the control flow of a parent PROCEDURE and its COPROC. Initially Acoroutine is invoked (1) at the top of its code. Upon encountering a DETACH

(2), control is transferred to the point in the PROCEDURE after the initiating call. The NEWTOP statement implicity resets the Acoroutine entry point to its top of code, and finally, a subsequent call to Acoroutine (3) transfers control to the top of code rather than to the prior intermediate exit point after the DETACH.

V. IMPLEMENTATION DESCRIPTION — PASCAL-S MODIFICATIONS

Whereas Section IV presented a detailed description of the syntactic and semantic characteristics of the various PASCAL-S additions which were necessary to implement the coroutine capability, this section provides an explanation of the underlying program changes made to the interpreter to accomplish the capability. An assumption is made that the reader has an understanding of the mechanics of implementing block-structured languages. References will be made to techniques and program structures which are based upon those utilized by Niklaus Wirth [3], [4] within his implementation of PASCAL and its related subset language PL-Ø. This includes the use of displays, dynamic and static links, and complicated stack-related data structures. As a preface to further discussions it may be useful to very briefly describe the current PASCAL-S compiler.

A. PASCAL-S OVERVIEW

PASCAL-S is written in the full CDC-6ØØØ version of PASCAL and is based upon recursive descent parsing. Code generation is performed in one pass during the syntactic analysis of program source code statements. No code optimization is performed. Further simplifying the compiling actions is the P-machine

concept built into its design. The code emitted during compilation is not absolute machine code. Instead, it produces code for a hypothetical virtual machine; the P-machine. At the end of source code translation to P-code, the virtual machine proceeds to interpret the intermediate code into physical machine actions. This interpretation step is, in fact, the process of "executing" the user program.

Any program coded in the PASCAL-S language can be compiled on the full PASCAL compiler. The reverse is not true. The following PASCAL constructs are not implemented in the PASCAL-S compiler:

- String constants or variables
- File, set, pointer, scalar, or packed variables
- Variant record structures
- Function or procedure names in a parameter list
- Label declarations

Appendix A details the syntax diagrams of the PASCAL-S language with the coroutine additions explicitly indicated.

From the standpoint of compiling and execution, there are two data structures contained in the compiler which are key to maintaining the block-structured nature of the language.

These two elements, symbol table and stack, are also key to the coroutine implementation. Therefore, a brief discussion of their design may be in order.

Unlike most conventional static compilers, PASCAL-S utilizes the symbol table heavily at run-time. The stack is the most important data structure to PASCAL-S (as it is to PASCAL), and is the mechanism which coordinates the run-time interaction of procedure bodies and data elements.

1. Symbol Table

The symbol table is used to store various attributes about each identifier found in the source program. At compile time these attributes are utilized to verify legal scope, syntax, and type relations with other program elements. At run-time, the symbol table is accessed to identify such variable or constant attributes as type, value, and level.

Identifiers are entered into the symbol table in order of appearence, yet the given scope relation must also be maintained. To accomplish this, the entries for each block level are linked together. Thus, the table can be scanned two ways; sequentially from newest to oldest entry regardless of level, or searched for entries in a given level (with entry indicated by a display).

Specifically, each symbol table entry contains the following mnemonic attributes:

- NAME
- LINK
- OBJ
- TYP
- REF
- NORMAL
- LEV
- ADR

NAME refers to the identifier name as identified by the lexical scanner. LINK is a pointer to the last (prior) symbol table entry made for an identifier in a given block level. OBJ indicates the nature of the entry as a constant, variable, type, procedure, or function identifier. TYP indicates the type of the constant or variable identified as an integer, real, boolean, character, array, or record. REF is used for procedure identifier entries to point to the block activation record in the block table. For array variables it used to point to the proper array table entry. NORMAL is a flag used for parameterlist items to indicate whether they are value or variable parameters. LEV specifies the block level where the identifier was scanned. Finally, ADR contains the address of the physical top of the code generated for a particular procedure or function. For variables, it specifies the relative displacement from the beginning of the activation record. For

a type identifier, it contains the size of the data structure.

FIGURE 5.1

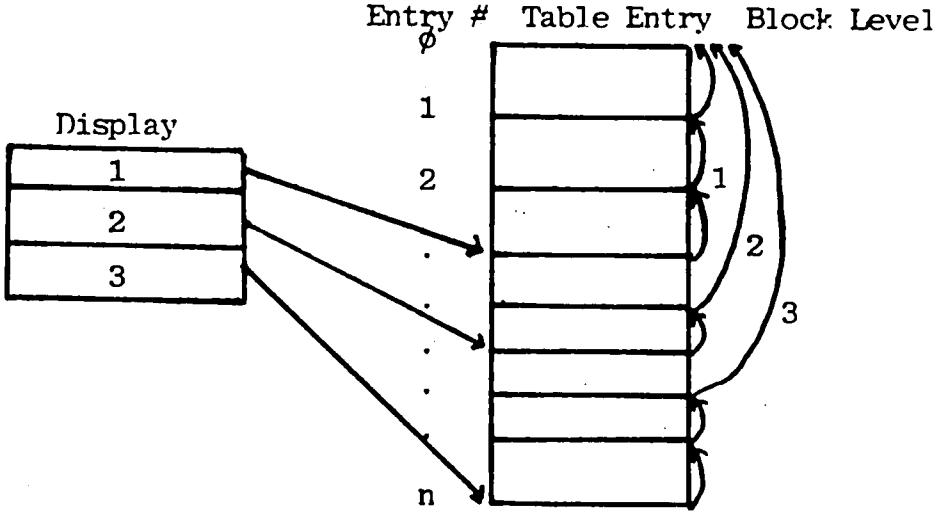


Figure 5.1 illustrates the symbol table structure. The entries are allocated sequentially. However, all identifiers in a given block are linked together. The display is continually updated to indicate the last entry for each block in the table. As indicated, identifier attribute LINK is used as the block linking mechanism, ultimately pointing to a null entry #ϕ.

The coroutine implementation to be discussed subsequently required additional attributes to be assigned and maintained in the symbol table. Currently, symbol table attributes are not modified at run-time. This is not the case with the coroutine implementation.

2. Run-time Stack

The run-time stack is clearly the most significant data structure in the PASCAL-S system. It is the stack where activation records and their attendant local data elements are manipulated by the P-code interpreter. As with the symbol table, a discussion of the current PASCAL-S stack structure is useful since the coroutine implementation significantly affects the operation of the run-time stack.

Each time a procedure/function call is invoked, an activation record is created at the end of the stack for the procedure/function being called. An activation record consists of reserved space for the data values declared local to that procedure body. In addition, the record contains five control fields which are used to implement calls, returns, and function value storage. When a procedure/function is called, the applicable static block table entry (created at compile time) is interrogated to determine the proper total displacement to be reserved for local data values, and four of the five fields are initialized. With space reserved for the five fields and local data values, the remaining stack space can be used for run-time use.

FIGURE 5.2 -- ACTIVATION RECORD

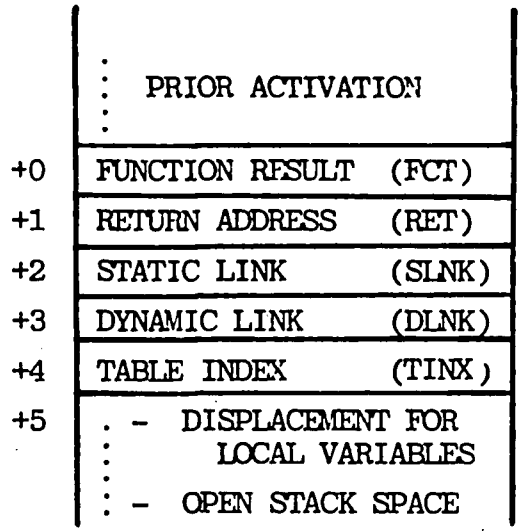


Figure 5.2 illustrates an activation record as it may appear on a stack. To the right of each entry a short mnemonic is parenthically enclosed. These abbreviations are used throughout subsequent discussions when addressing these fields.

Note: These names are not actually used in the compiler, but are used here to facilitate explanation.

At the very base of the record (FCT) a FUNCTION would store its value upon existing back to the calling point. RET, the Return Address (base + 1), is the relative location containing the address of the code just after the point of the originating call. The Static Link (SLNK) is a pointer to the block activation record which maintains the scope relation of procedure blocks defined at compile time. This is necessary to access appropriate data values outside the block. The Dynamic

Link (DLNK) points to the base of the activation record placed previously on the stack and still active. This is necessary for valid transfer of control upon exit from a given procedure/function body. Finally, the Table Index (TINX) is a pointer to the static symbol table entry of the called procedure/function identifier. This is needed to access its static level for subsequent use in non-local variable manipulation. In effect, the stack contains block activation records for procedures/functions still in process. They are linked together in two ways. One way (dynamic link) provides a history of prior procedure activations. The second way (static link) maintains the variable scope relationship for data access of non-local data elements in other activation instances.

As stated earlier, the coroutine implementation requires that the stack activation record concept be modified as well as the static symbol table. The remainder of this section describes these and other modifications.

B. COROUTINE IMPLEMENTATION

1. Code Generation

In order to develop a run-time coroutine capability it was necessary to augment the P-machine with four additional primitive P-codes. Various combinations of these codes are generated to implement the DETACH, RESUME, NEWTOP, and COPROC natural exit.

In addition, the P-code (19) which had already existed to perform a standard procedure/function call was modified to also handle the initial call to a COPROC cycle from an external procedure body.

TABLE 5.1

P-MACHINE ADDITIONS

P-CODE	FUNCTION	# OF OPERANDS
64	RESUME	2
65	COPROC EXIT	0
66	SET ENTRY TO TOP OF COPROC	1
67	SET ENTRY TO INTERMEDIATE PT.	1

The P-code additions summarized in Table 5.1 are combined, as cited previously, to implement the various language additions. However, the COLINK declaration simply initiates entry of a COPROC identifier into the symbol table and requires no explicit code generation.

Upon encountering a RESUME, Procedure "resumestatement" is invoked. This procedure performs the following actions:

1. Verifies that the RESUME is contained inside the body of a COPROC;

2. Searches in the symbol table for the COPROC identifier which is the object of the RESUME and makes sure it is the same block level as the containing COPROC;

3. Emits P-code (64, x, y) where x points to the symbol table entry of the COPROC in which the RESUME statement resides, and y points to the symbol table entry of the COPROC object of the RESUME;

4. If the static block level of the COPROC object of the RESUME is less than the level of the block containing the RESUME statement then it emits P-code (3, x, y) where x = object level and y = containing COPROC level.

Upon encountering a COPROC declaration, Procedure "koproc-declaration" is invoked. This procedure performs the following actions:

1. Verifies that the COPROC identifier already exists in the symbol table (via proceeding COLINK declaration);

2. Calls Procedure block recursively to compile the procedure body;

3. On return from "block", code is emitted for a COPROC natural exit. It emits P-code (66, x) where x is a pointer to the symbol table entry of the COPROC identifier. This instruction resets the COPROC entry point to the physical top of its code.

4. Emits P-code (65) which performs a general COPROC deactivation/exit.

Upon encountering a DETACH statement Procedure "detachstatement" is invoked. This procedure performs the following actions:

1. Verifies that the DETACH statement is contained within the body of a COPROC;

2. Emits P-code (67, x), where x is a pointer to the symbol table entry of the COPROC in which the DETACH statement resides;

3. Emits P-code (65) which performs a general COPROC deactivation/exit.

Upon encountering a NEWTOP statement Procedure "newtop-statement" is invoked. This procedure performs the following functions for each COPROC identifier specified:

1. Verifies that the COPROC object is in the symbol table with a level indicating that it is a COPROC which is nested (contained) within the procedure body where the NEWTOP statement resides;

2. Emits P-code (66, x) where x is a pointer to the symbol table entry of the COPROC object named in the NEWTOP statement. This instruction sets that object's entry point to the physical top of its code.

Upon encountering an initial call to a COPROC cycle from inside a PROCEDURE, Procedure "call" is invoked as it would be for and PROCEDURE or FUNCTION call. This procedure performs the following functions as they relate to a COPROC call:

1. Emits P-code (18, x), where x is a pointer to the symbol table entry of the COPROC being called. This instruction marks the stack for the subsequent call instruction;
2. Emits P-code (19,x), where x is the size of a parameterlist (not used for COPROC). This instruction performs the call;
3. If the level of COPROC being called is less than the block level where the call resides, then it emits P-code (3, x,y), where x is the level of the current block, and y is the level of the containing COPROC.

2. Stack/Symbol Table Enhancements

In order to implement the capability of COPROC to COPROC communication, it was necessary to consider the effect repeated calls would have on the run-time stack. Normally, each time a PROCEDURE or FUNCTION is called from within a procedure body, its activation record is placed on the stack on top of the current record. If this PROCEDURE/FUNCTION was again called (recursively), another activation record would be placed on the stack on top of the current record. Until a procedure body finally reaches its natural exit point, its activation record remains on the stack. By definition, when a COPROC issues a RESUME to another COPROC in a cycle, its actions must be temporarily suspended (along with any temporary values on the stack). That COPROC just called may, in addition, issue its own RESUME to the COPROC that just called it. Though this is considered recursion in the normal sense, under a coroutine philosophy, this is simply two-way communication. If COPROC to COPROC communication were implemented exactly as a PROCEDURE or FUNCTION, each time a RESUME occured, a new activation record would be placed on the stack for the COPROC being re-entered. Under most circumstances, this would lead to the same limitation of implementing deeply nested recursive algorithms. That is, running out of memory to store the activation records on the stack.

Realizing that implementing a quasi-recursive coroutine structure in a block-structured language must be performed without utilizing the conventional technique of stacking activation records, a data structure was designed which allowed the present PASCAL-S stack to be utilized, while re-defining the concept of a block activation for COPROC cycles. Thus, the stack enhancements implemented are transparent to normal PROCEDURE of FUNCTION activations. Only the RESUME of a given COPROC within a cycle causes the new stack definitions to be used.

Specifically, the maintenance of COPROC activations on the stack redefines the conventional notion of activation records. Now, instead of creating an activation record for each call (RESUME), the block activation occurs only at the time of the initial call to the COPROC cycle.

A subsequent RESUME of a COPROC within the cycle does not cause the creation of a completely new activation record. Instead, the same activation record is maintained. The stack area of the COPROC to be reactivated is preserved within the record, and a new stack area is initialized for the called COPROC.

Since the COPROC object of a RESUME may have been previously active, and thus preserved on the stack, the initialization process may consist of swapping pieces of the activation record's stack area (stack segments). In this case, the stack segment attributed

to the COPROC being reactivated (the object of a RESUME) would be moved to the end of the stack to enable its activity to continue utilizing the stack for storing temporary values and evaluating expressions.

The displacement of stack segments requires a considerable amount of overhead to properly implement a RESUME. This is due to the amount of internal "bookkeeping" required to maintain the boundaries of the stack segment for each COPROC which has been activated within a cycle. In addition, preserved stack segments must be continually swapped in order to place the segment of the COPROC to be reactivated into position at the end of the stack. Any stack area freed by the displacement of this segment is re-used by shifting segments back into its former location.

FIGURE 5.3

CYCLE ACTIVATION RECORD

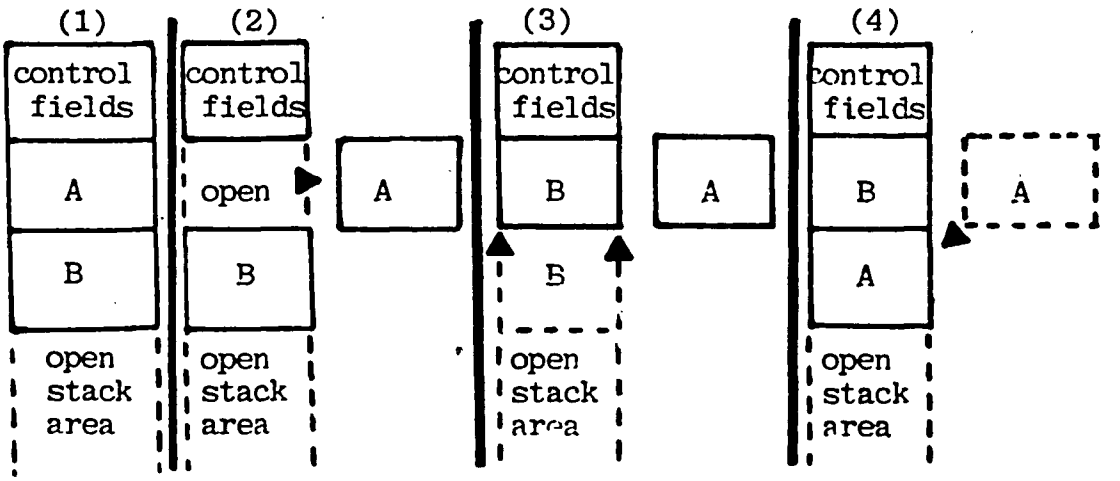


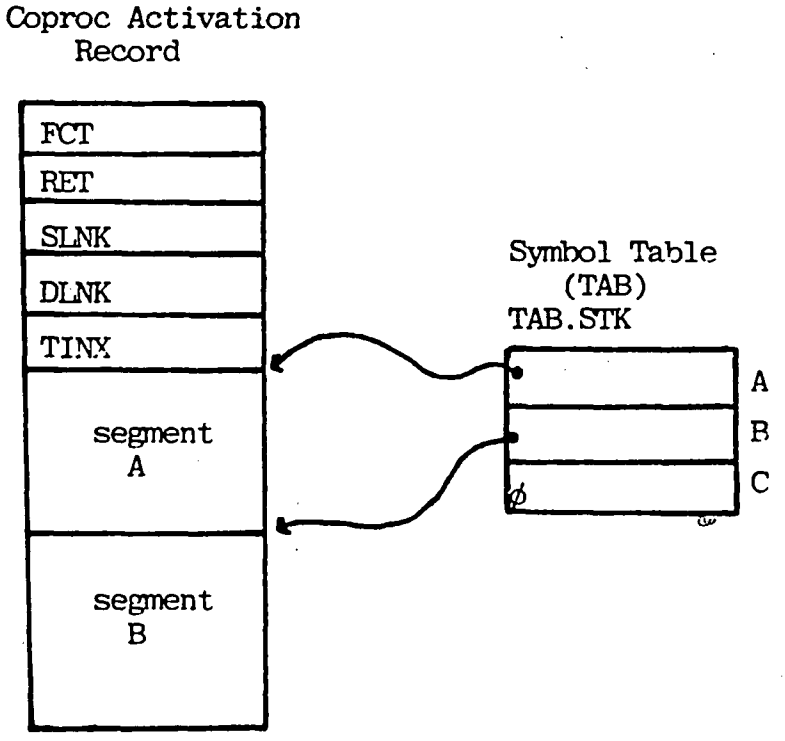
Figure 5.3 illustrates the generalized four step sequence which occurs to re-activate (RESUME) a COPROC previously preserved on the stack. The block activation record consists of the same five control fields (function result, return address, static link, dynamic link, and table index) as a normal PROCEDURE or FUNCTION. In addition, a stack segment is maintained for each COPROC previously activated (and requiring stack preservation). In illustration (1), COPROC A and COPROC B have been previously activated and require stack preservation (temporary storage of local control variables). If the statement RESUME(A) were executed within the body of B, it would be necessary to reactivate A. Here, the stack segment preserved for A would be moved to temporary storage (2). The segment currently in use by B at the end of the stack area would be preserved and shifted into the unused space within the activation record left by swapping out A's stack segment (3). Finally, the stack segment previously saved for A would be returned to the activation record (4). Now, however, it is placed at the open end of the stack area. Reactivation of COPROC A can now proceed since its stack segment is in place for potential expansion.

In order to implement the cycle activation record stack segmentation structure, the PASCAL-S static symbol table ("TAB") was augmented to include a field, STK, for each identified entry (in addition to NAME, LINK, OBJ, TYP, REF,

NORMAL, LEV, ADR). This field, really applicable only to a COPROC identifier, is used at run-time as a display into the active stack segments. In addition, it serves to indicate the status (active/inactive) of a particular COPROC. If a COPROC has yet to be activated (via RESUME) within a cycle, TAB.STK would contain a value of zero indicating no stack segment is active for that COPROC.

Note: STK is reset to zero when the COPROC is deactivated via a DETACH or natural exit.

FIGURE 5.4



As illustrated in Figure 5.4, the additional field STK in the symbol table entries for COPROC A and COPROC B acts as a display into the stack segments of a currently activated COPROC block, pointing to the location of each segment on the stack.

It should be pointed out that, though not indicated, several of the control fields take on a different meaning for a COPROC than for a PROCEDURE or FUNCTION. For a COPROC activation, the RET field will contain the address of code immediately after the point of the initial call into the cycle. This field will remain constant regardless of subsequent RESUME statements. Its contents will be utilized only upon the issuance of a DETACH or encounter of a natural end within a COPROC. In order to store the intermediate (return address) entry/exit point for a given COPROC, another field within its symbol table (TAB) entry has been implemented (in addition to STK). This new field, TOP, always contains the address of the entry point applicable to the COPROC at any given time. Initially, a COPROC would have its TOP set to the physical top of its code. However, a subsequent intermediate exit to another COPROC (via a RESUME) would cause its TOP to be set to the point in the code immediately after this exit point.

The control field TINX (which previously contained a pointer to the symbol table entry of the particular block's PROCEDURE/FUNCTION identifier) now contains a pointer to the

symbol table entry of the COPROC whose stack segment resides at the open end of the stack. Referring back to Figure 5.4, TINX would contain a pointer to the symbol table entry for COPROC identifier B. When stack segments are shifted and swapped, TINX is updated to reflect the change.

FIGURE 5.5

REVISED SYMBOL TABLE ENTRY (TAB)

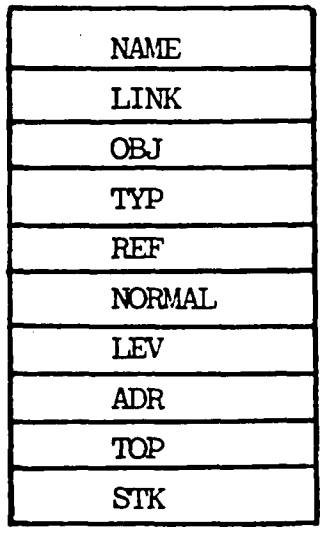


Figure 5.5 summarizes the augmented symbol table format which has been implemented to accommodate the coroutine capability. As indicated, TOP and STK are applicable to a COPROC identifier. TOP always points to the entry point into its code. Initially it is the physical top. Subsequently, it is updated for intermediate entries and exits. STK is a display pointer into the active stack segment for that COPROC. It is zero if no stack segment is active.

Note: ADR always points the physical top of code for a COPROC. This is identical to a PROCEDURE or FUNCTION, where ADR points to the entry point of the procedure body (which in the case of Procedures or Functions is always the physical top of the code). In order to maintain logical programming flow, ADR is no longer used to perform entry into a PROCEDURE/FUNCTION. Instead, TOP is used (since it never changes for them).

In order to facilitate manipulation of individual stack segments, a data structure was implemented within each segment which serves to effectively link them together, and in turn, to the static control fields at the beginning of the COPROC cycle activation record. At the beginning of each stack segment, two fields are defined upon initial activation of a given COPROC. The first field, LEN, contains the length of succeeding stack elements. Thus, a given segment always has a total length of LEN+1. This field is updated when a stack segment must be preserved. At this time it is updated to include any stack element (temporary control variables) which must be saved. The second field, TPTR,

is a pointer to the symbol table entry of the COPROC identifier representing this stack segment. Upon initial activation, a stack segment has its LEN set to 1. Its total length is always LEN+1 (= 2) which is in this initial case, only the two control fields LEN and TPTR. If a COPROC issues a RESUME within the body of a FOR, REPEAT, or WHILE statement, the stack segment would also contain the associated control loop variables. These values would be preserved if necessary, and LEN would reflect the total segment size.

Note: The mnemonic LEN and TPTR are used to facilitate discussion and are not the variable names found within the compiler. There, these fields are always addressed only as offsets from the base of the activation record.

FIGURE 5.6

COPROC CYCLE ACTIVATION RECORD

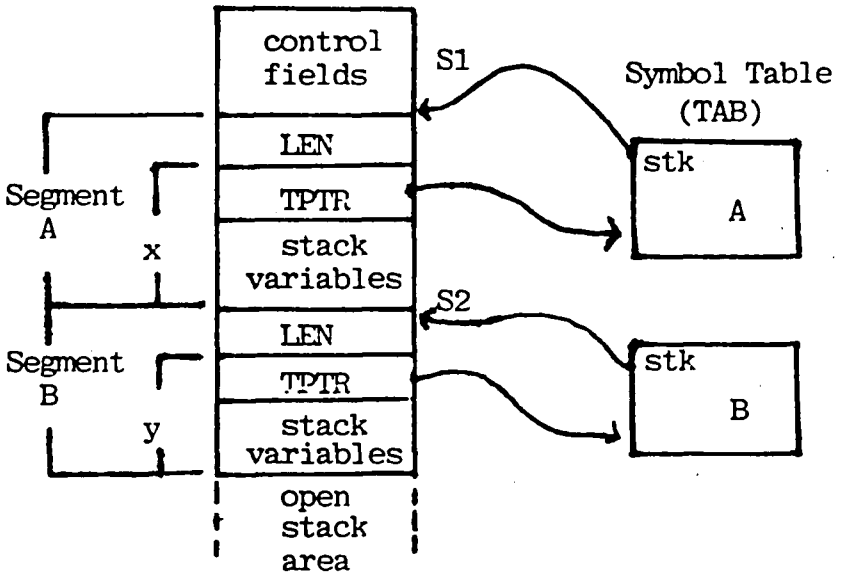


Figure 5.6 illustrates a familiar COPROC cycle activation record containing two stack segments preserved for COPROC A and COPROC B. LEN for segment A contains a value x which is equal to the number of stack variables in that stack + 1 (this is TPTR). TPTR points to the appropriate symbol table entry. Likewise for segment B. The P-machine code (64) implementing a RESUME utilitizes these fields for segment maintenance. This will be described in detail subsequently. However, to put it simply for now, it is possible to travel down the stack segment chain merely by knowing the length of each segment. The first

segment (S1) always starts after the five activation control fields (base + 5). From there, the next segment(S2) (if any) would start at the point S1+LEN(S1)+1, and so on down the chain of segments. Here, S1 = base + 5, LEN(S1) = contents of location S1. Since LEN is the first field in any stack segment it can be addressed in the first case by interrogating S1.

3. P-Code Actions

This sub-section serves to document the specific actions performed by the newly implemented P-codes (64, 65, 66, 67) as summarized earlier in Table 5.1. In addition, it will be necessary to describe the change made to the previously existing P-code (19) which has existed to implement normal PROCEDURE or FUNCTION calls. It was modified to also perform an initial call to a COPROC cycle.

Upon encountering the P-code (19) generated to perform the initial call to a COPROC cycle, the following actions are invoked by the new PASCAL-S P-Machine, assuming that the previous code executed was a P-code 18, "mark stack", (normally invoked prior to any type of call in order to allocate space for a new activation record):

1. Initialize the Activation Record control fields as follows:

- TINX:=Symbol table pointer to the COPROC object
- RET:=Address of code after initial call
- SLINK:=Base address of the last activation record residing at the same static level as this COPROC identifier
- DLINK:=Base address of the last activation record accessed
- FCT let undefined;

2. Initialize base address pointer (B) to beginning of activation record;
3. Initialize stack pointer to beginning of open stack area (after control fields);
4. Set the Program Counter to the entry point of the COPROC as specified in TOP contained in its static symbol table entry;
5. Increment the stack pointed by 2 in order to reserve space for the initial stack segment fields LEN and TPTR;
6. Initialize LEN to zero and TPTR to the location of the COPROC symbol table entry;
7. Update the COPROC stack segment display (STK) contained in its static symbol table entry to point to the beginning of this segment (address of LEN).

It is interesting to note that the first four of the seven activities delineated above are also used in the call of a normal PROCEDURE or FUNCTION. The remaining three steps are performed only if the call is to a COPROC (thus initiating a cycle).

Upon encountering the P-Code (64) generated to perform a RESUME, the following actions are invoked by the PASCAL-S P-Machine:

1. The TOP of the COPROC being exited is updated to the point where subsequent intermediate entry must occur (immediately after the RESUME).
2. The cycle activation record control field TINX is updated to point to the symbol table entry of the COPROC object of the RESUME.
3. The Program counter is set to the intermediate entry point (TOP) of the COPROC object of the RESUME.
4. If any temporary variables have been stored on the stack by the COPROC being deactivated, the LEN of its stack segment is updated to preserve this space. Otherwise, the stack need not be preserved. In this case,

the stack pointer is decremented by 2 to relinquish the stack segment space and the COPROC is formally deactivated by setting its STK (in its symbol table entry) to zero.

5. The symbol table entry of the COPROC object of the RESUME (callee) is interrogated to see if it has an active stack (STK $\neq \phi$).

   - If the segment is now at the end of the stack then the stack pointer will already be placed to resume operation (to TPTR).

   - If the segment is <u>not</u> at the end of the stack area (in the middle of the preserved stack segment space for that cycle) then, 1) the entire segment is moved into a temporary area (COHOLD), 2) any other stack segments residing at a point after the moved segment are pushed up into the fragmented space created by the move operation (garbage collection), and 3) the stack segment copied into COHOLD is moved back into the activation area. This time, to the end of the stack. The segment is now activated.

6. If the COPROC object is inactive and thus has <u>no</u> preserved stack segment (STK = ϕ), then the stack pointer is incremented by 2 to allocate space for a new stack segment (LEN and TPTR). Len is initialized to zero, and TPTR to the table entry (same as TINX).

Note: The temporary area COHOLD is the name of the actual data structure implemented in the new compiler.

Upon encountering the P-Code (65) generated to perform a generalized COPROC exit, the following actions are invoked by the PASCAL-S P-Machine:

1. Each COPROC in the cycle is deactivated by setting its STK (in its symbol table entry) field to zero. This is accomplished by following the chain of stack segments and utilizing each TPTR entry to gain access to the proper symbol table location.

2. The stack pointer is decremented to the point just before the current COPROC cycle activation record. This will effectively erase the record from the stack.

3. The Program Counter is set to the address contained in the activation record control field RET. This will transfer control to the point just after the initial call to the COPROC cycle in the calling procedure body.

4. The base address pointer (B) is set to the address contained in the COPROC activation record control field DLINK. This is the dynamic link back to the activation record location on the stack of the calling procedure body.

Upon encountering the P-Code (66) generated to perform a reset of a COPROC entry point to the physical top of its code, the following action is invoked by the PASCAL-S P-Machine:

1. The particular COPROC static symbol table entry field TOP is set equal to its field ADR. ADR always contains the address of the physical TOP of its code. TOP is the field used upon a RESUME do determine where to re-enter the COPROC.

Upon encountering the P-Code (67) generated to perform an update of a COPROC intermediate entry point, the following action is invoked by the PASCAL-S P-Machine:

1. The particular COPROC static symbol table entry field TOP is set equal to the current value of the Program Counter + 1. The + 1 value assumes that the next instruction to be performed will be P-Code (65), COPROC Exit. Thus, the intermediate re-entry point must be set to the point after this P-Code. TOP will be used upon issuance of a RESUME to determine where to re-enter the COPROC.

VI. POTENTIAL FOR EXPANSION

The coroutine implementation as presently devised was intended to serve as a solid foundation from which future enhancements can be based. In order to facilitate construction of the initial PASCAL-S coroutine implementation, the trade-offs between simplicity and power were considered. The main thrust of this current effort was to develop the instruction formats and control structures which can produce coroutine actions. As a result, the PASCAL-S language has been augumented with a whole new repertoire of extensions (COLINK, COPROC, RESUME, DETACH, NEWTOP). Each of these elements contribute to the implementation of the specific coroutine actions. However, in this language version little effort has been expended to implement explicit definition and control of the data structures through which coroutines communicate among themselves.

Referring back to Section IV, Syntax and Scope, it was stressed that coroutines, as implemented in COPROC procedure bodies, can contain no local variables (through procedures contained within them may). Thus, COPROC actions can affect any variable data global to it. This limitation creates the potential hazard of unintentionally affecting data values. The whole purpose of normally implementing the definition of local variables is to

explicitly delineate their scope. This in turn provides the programmer with a convenient tool for controlling data manipulation. Since the current coroutine implementation does not allow for the definition of local variables within a COPROC body, unintentional and uncontrolled data manipulation may result.

In terms of expansion possibilities, any future enhancements to this PASCAL-S coroutine implementation should primarily center on providing local variable definition within COPROC bodies. It should be pointed out that the current implementation was designed with this thought in mind. Therefore, the ability to define local variables will not require extraordinary modifications to the new PASCAL-S compiler.

Specifically, the following steps provide a generalized outline of the tasks which would be required to implement the definition of local variables within a COPROC:

- Within the main body of procedure "Block", eliminate the code segment which disallows the calls to procedures "Constantdeclaration", "Typedeclaration", and "Variabledeclaration".
- Modify P-Code (64) which performs the RESUME actions to allocate space on the run-time stack for the displacement of local variables contained in a given COPROC.

Of the modifications outlined above, only the last (P-Code (64)) enhancement may require non-trivial consideration. This is due to the current implementation of stack segments for COPROC to COPROC communication (see Section V-B,2, Stack/Symbol Table Enhancements). Variables cannot be allocated via a simple displacement constant relative to the beginning of the activation record. Instead, displacement must occur from the beginning of a stack segment. Since each stack segment starts with two control fields (LEN and TPTR), local variable allocation must begin immediately after these fields. It should be recognized that the introduction of local variables within a stack segment will generate increased overhead on a RESUME since more elements must be swapped and moved on the stack.

In addition to the implementation of local variables within a COPROC, significant thought must be given to the support of parameterlists for initial cycle activation and/or COPROC to COPROC communication. Specifically, the use of a parameterlist may be implemented in the initial call of a COPROC cycle, or as an extension of the RESUME process from within a COPROC. Unfortunately, the implementation of both these concepts requires significant planning. The parameters needed may differ for inter-CORPOC communication than for COPROC to maintain procedure communication.

Since a COPROC body can, by definition, only communicate with another COPROC within its cycle, the extent of data communication can be well defined. Therefore, it may be appropriate to eliminate the use of a parameterlist within a COPROC declaration to define those variables to be used for inter-COPROC communication. Instead, the attendant COLINK declaration could be augumented to include the declaration of variables local to the whole cycle (not just an individual COPROC). These variables may be used for inter-COPROC communication. This approach may be preferable to maintaining local parameter values in the dynamic stack segment since it eliminates much of the overhead needed to maintain (swap) additional segment elements. Also, a pre-declaration (via COLINK) of variables local to the cycle, would allow them to be easily maintained in the static region of the cycle activation record; just as local variables are maintained in a normal PROCEDURE or FUNCTION environment.

Even if no parameterlists are maintained within individual COPROC declarations, a means must still be devised to support the passing of parameters from the initial calling procedure body to the COPROC cycle. Since, syntactically, a normal call to a COPROC is used to initiate the cycle, it is only natural to append a parameterlist to the call. However, the individual COPROC would not include a similar parameterlist. Instead, it

again may be useful to declare such parameters within the COLINK declaration of the cycle. These values are to be used for COPROC to main program controlled communication. Implementation of such an approach would allow use of the static region of the cycle activation record for parameter displacement.

Hypothetically, the COLINK enhancements could syntactically look like this:

COLINK

COROUTINES = $COPROC_1$, $COPROC_2$,..., $COPROC_n$:

BUFFER = ($parameterlist_1$);

CHANNEL = ($parameterlist_2$);

Here, each COPROC to be defined within the cycle would be contained in the COROUTINES sub-declaration. The variables local to the cycle for inter-COPROC communication would be delineated in the BUFFER sub-declaration. Finally, the parameterlist to be used for communication to the initial calling procedure body would be declared in the CHANNEL sub-declaration. These variables, like those of the BUFFER sub-declaration are considered local to the entire cycle. This means they are unaffected by any individual COPROC activation.

These enhancements may or may not be optimally constructed. However, they do appear to be easily implementable, and indeed, do provide a convenient means to support local variables within a COPROC, controlled variable declarations for communication within a COPROC cycle, and controlled variable declaration for communication between a COPROC cycle and its initiating procedure body. Further efforts in this area may provide insights into a more practical means of implementing these constructs.

LIST OF REFERENCES

1. Conway, Melvin E., "Design of a Separable Transition-Diagram Compiler", Comm. ACM 6(July 1963), pp. 396-408.

2. Wang, Arne, and Dahl, Ole-Johan, "Coroutine Sequencing in a Block Structured Environment", BIT 11(1971), pp. 425-449.

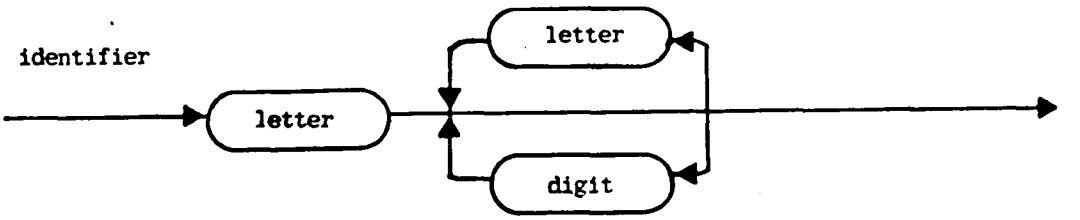
3. Wirth, Niklaus, Algorithms + Data Structures = Programs, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1976.

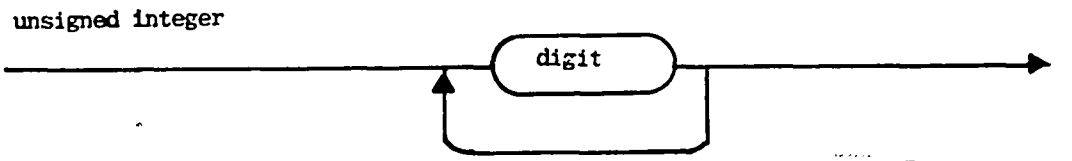
4. Jensen, Kathleen, and Wirth, Niklaus, Pascal User Manual and Report, Berlin Heidelberg New York: Springer-Verlag, 1974.

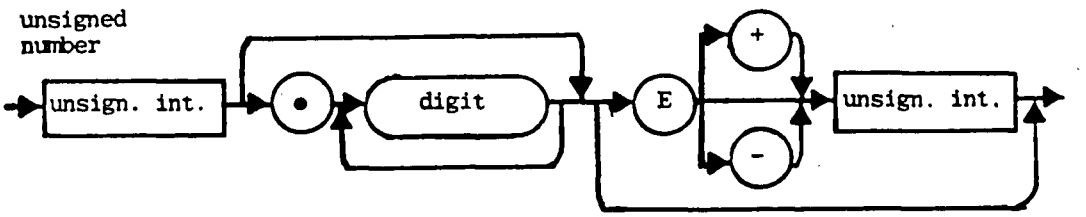
5. Wirth, Niklaus,Pascal-S Compiler, E.T.H. Zurich, Switzerland, January 3, 1976.

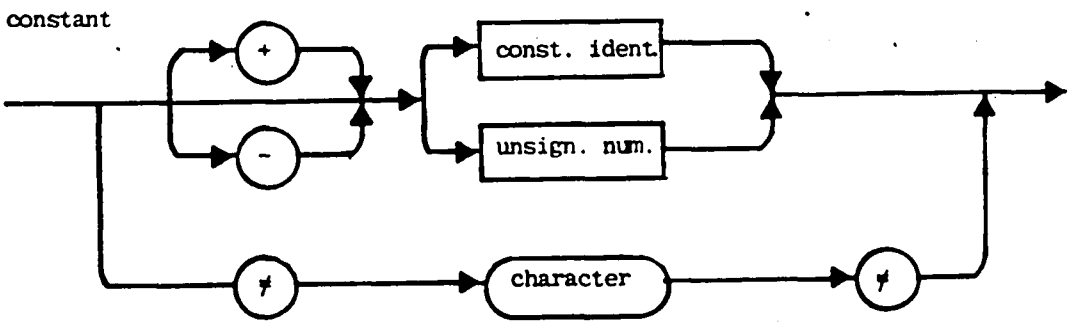
Note: A copy of the Pascal-S coroutine implementation source program is on file in the Division of Computing Science, Lehigh University, 1979.

PASCAL-S IMPLEMENTATION
SYNTAX DIAGRAMS



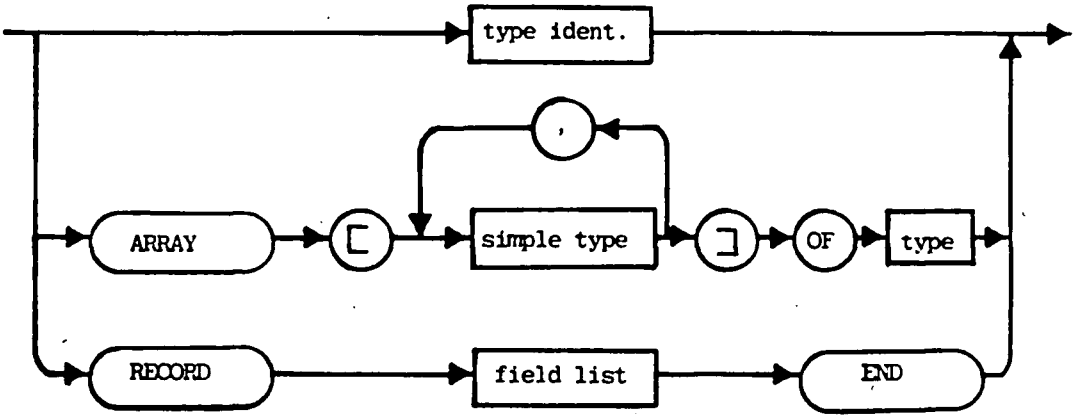




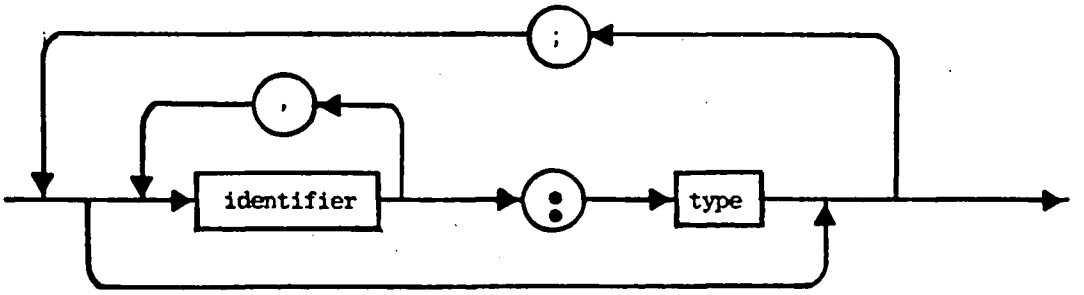


array type
(simple)

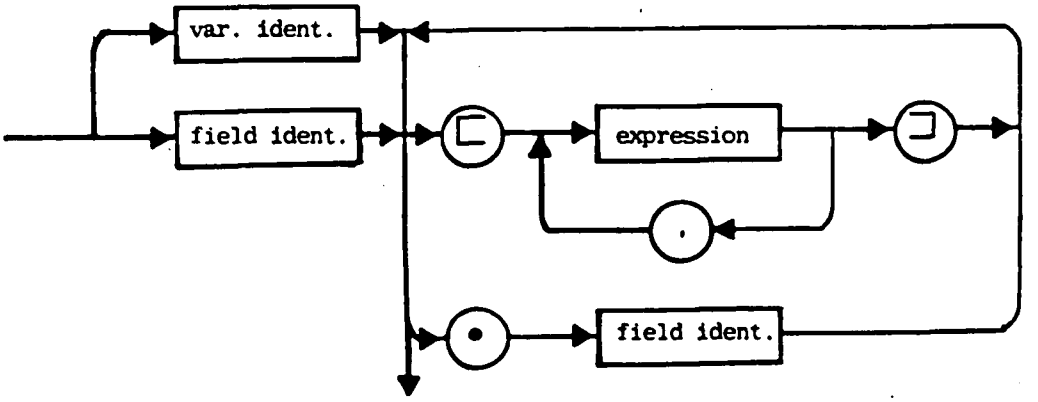
type



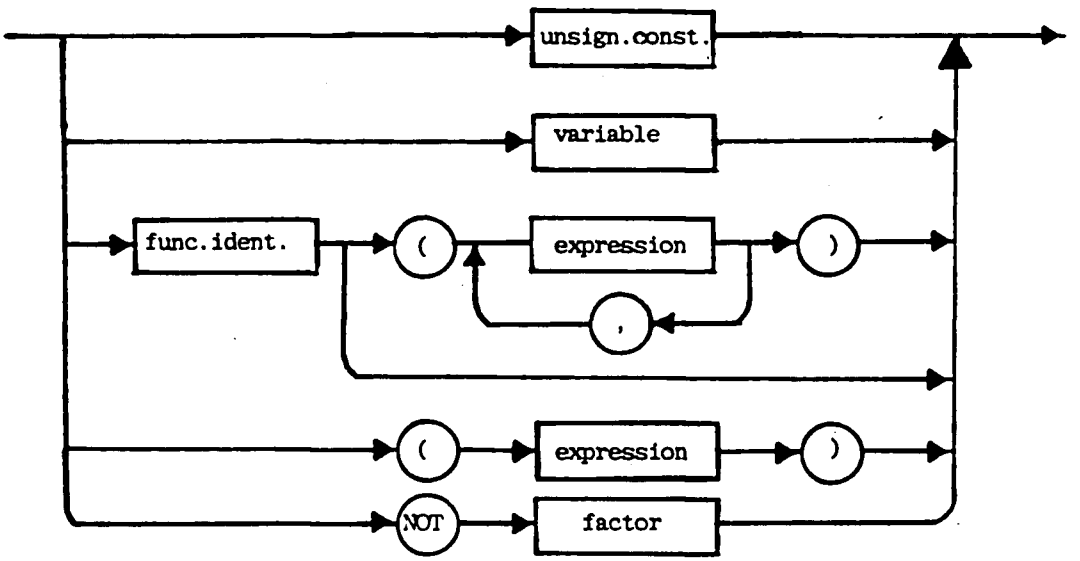
field list



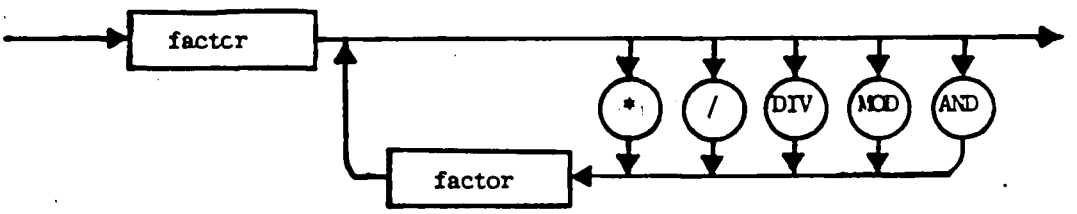
variable



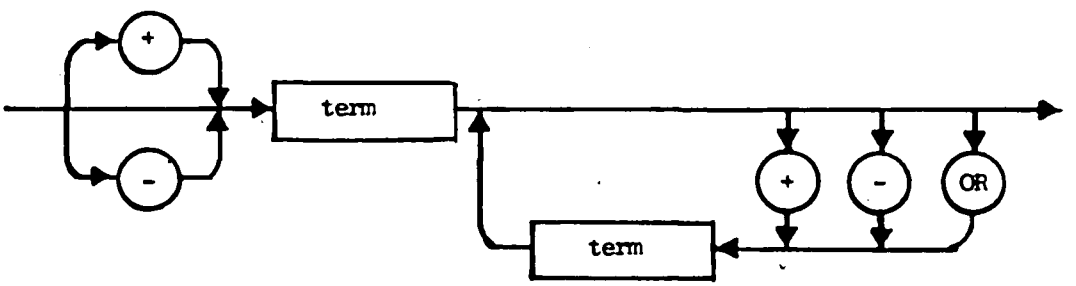
factor



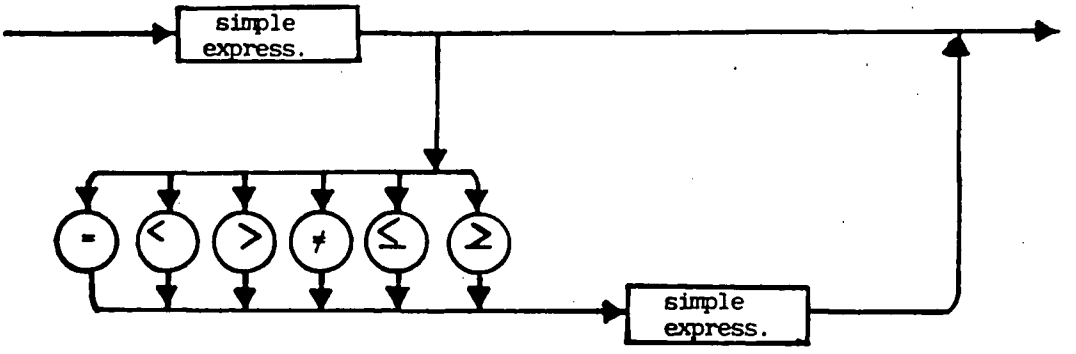
term



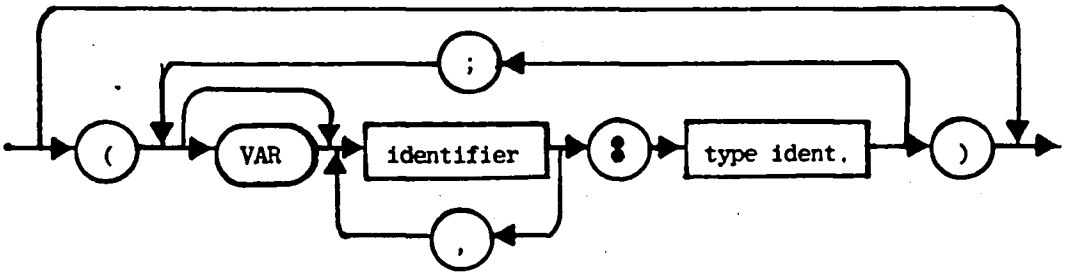
simple expression

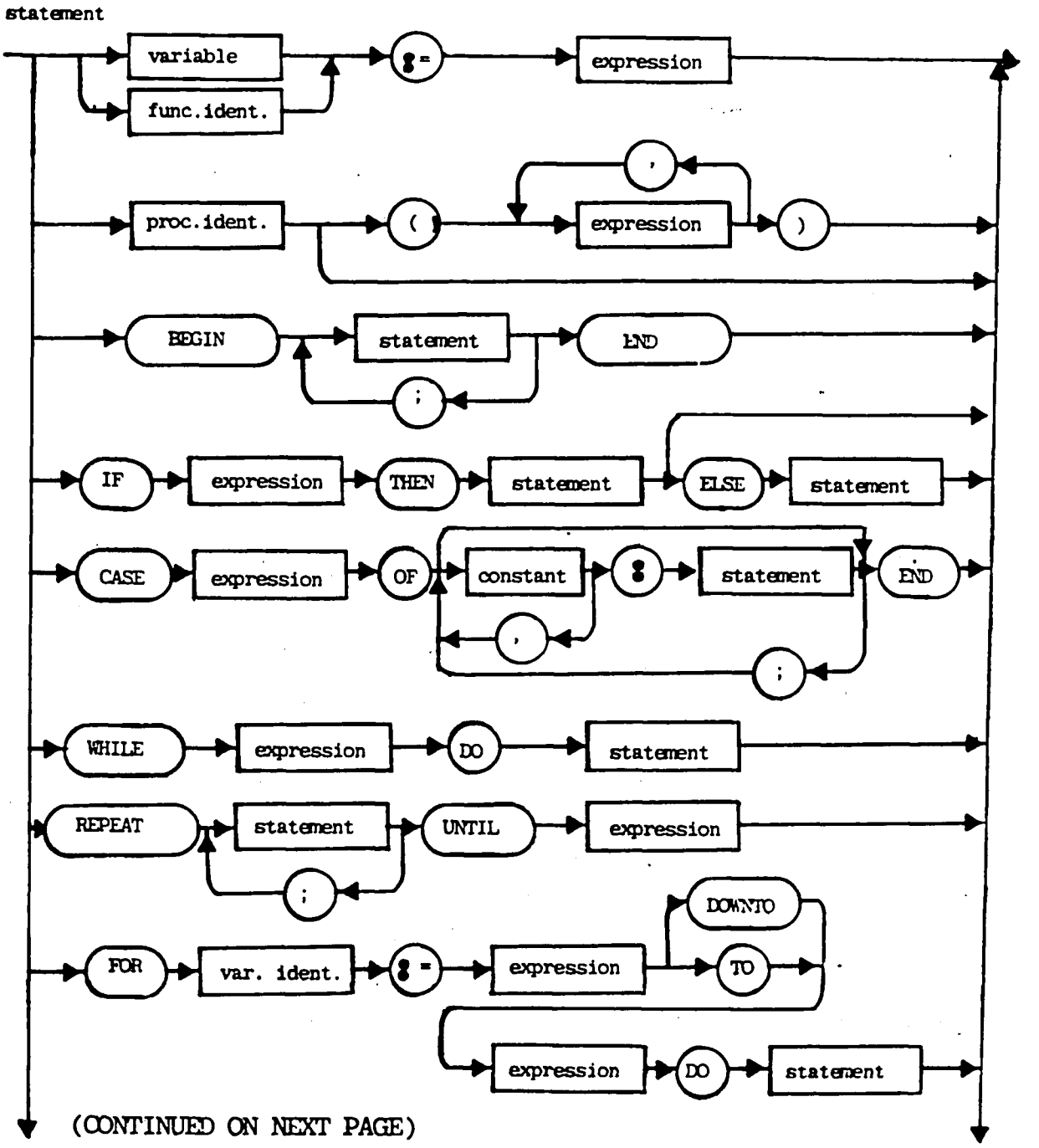


expression



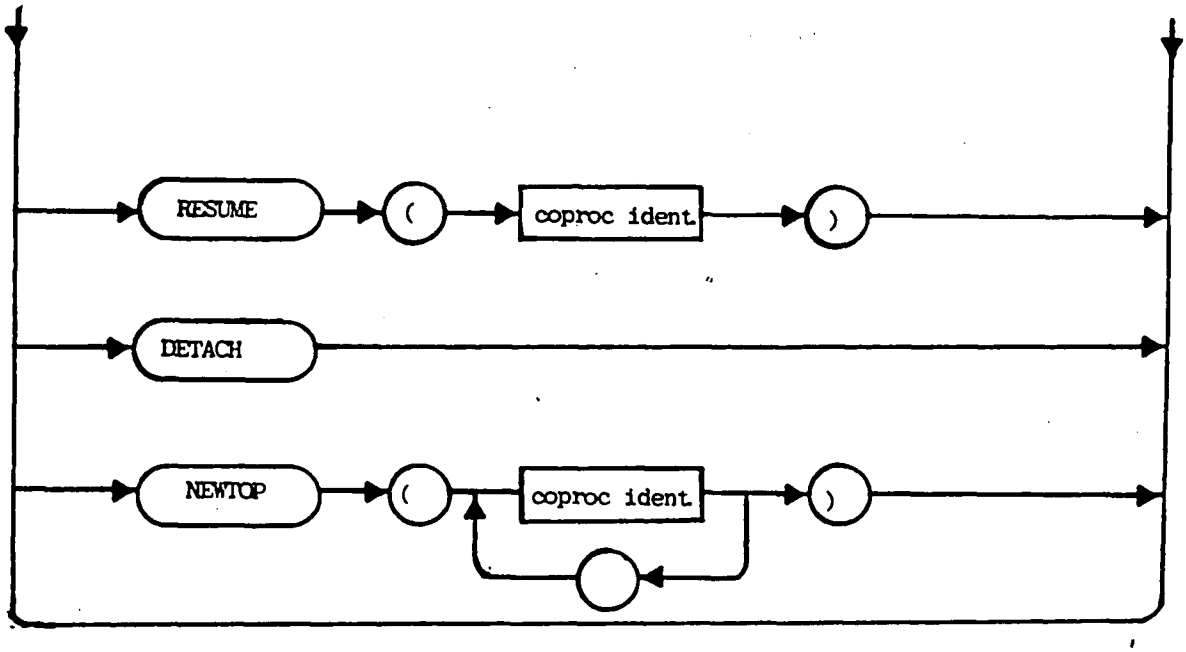
parameter list

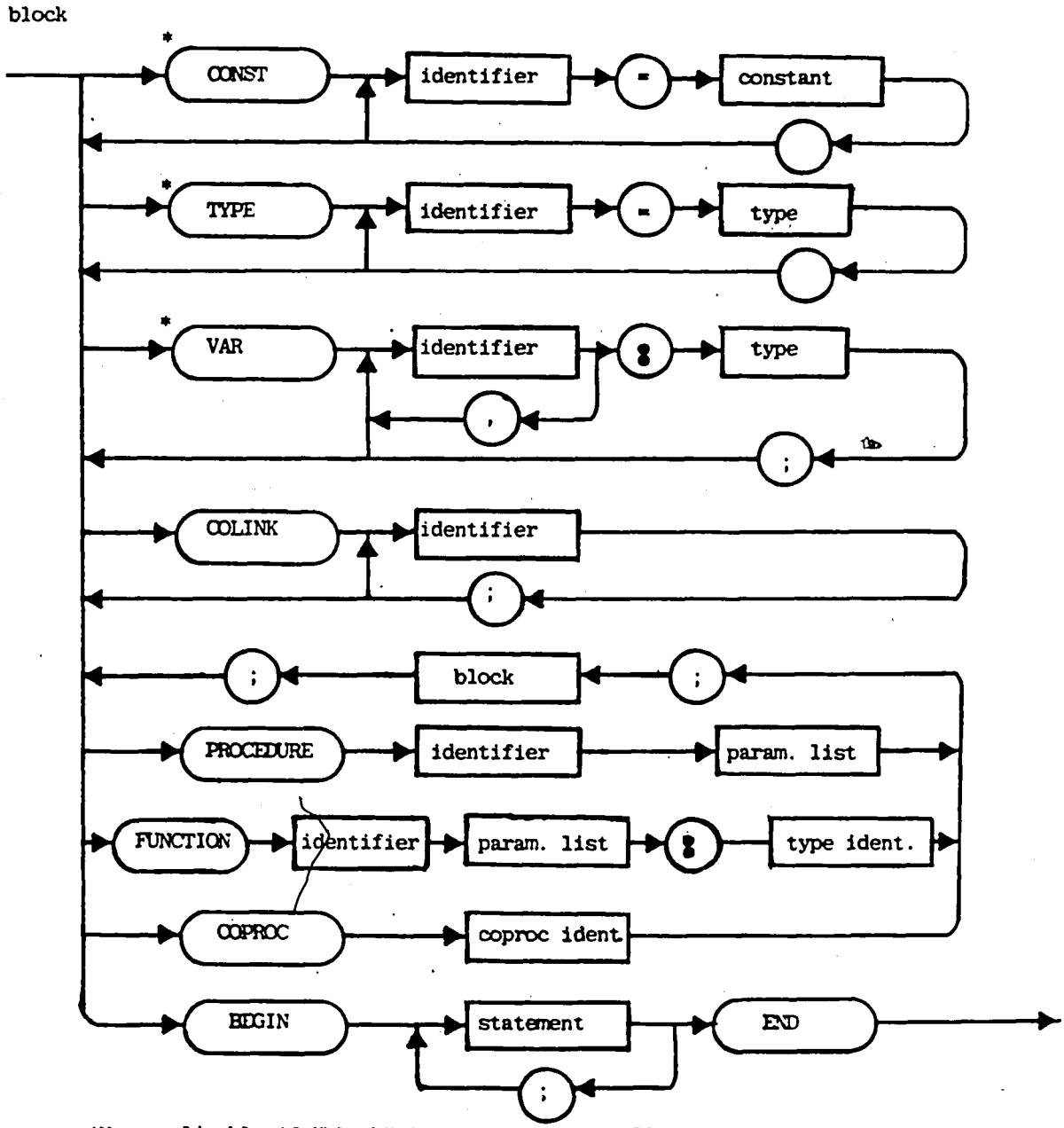




(CONTINUED ON NEXT PAGE)

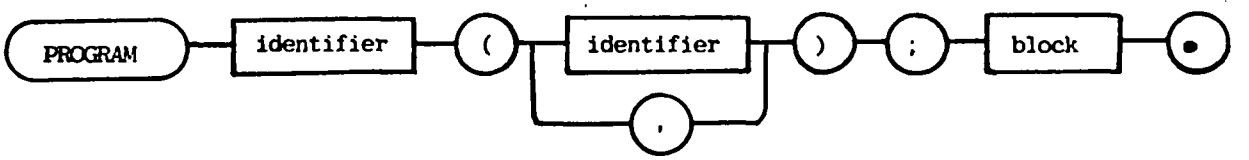
("STATEMENT" CONTINUED)





*Not applicable if "block" is entered as a COPROC.

program



BIOGRAPHY OF AUTHOR

Allan R. Frank was born on January 16, 1955 in Allentown, Pennsylvania. His parents, Bernard and Muriel Frank still reside there. Mr. Frank received a B.S. in Accounting, cum laude (1976), and M.B.A. in Finance (1978) from Lehigh University. He is currently completing the requirements for the M.S. in Information Science from the same institution. Mr. Frank is employed by Arthur Young & Company, Philadelphia, as a management consultant in the Information Systems/EDP practice area. Mr. Frank is in the process of completing the requirements to become a Certified Public Accountant in Pennsylvania, and has had prior professional experience as an auditor with the same firm. In addition, he has had professional experience as a programmer/analyst for a major east coast retailing chain.