A Coroutine Approach to Parsing

HANAN SAMET
University of Maryland

A method is presented for parsing syntactic constructs that are permitted to appear independently anywhere in a program. Some examples include comments, macros, and constructs for conditional compilation. Each such construct is defined by its own grammar and parsed by a separate coroutine. The coroutine model of parsing allows the program text to be parsed in one pass despite the syntactic inconsistencies among the program text and the additional constructs. The usefulness of the model is demonstrated by showing how a production language parsing method is extended to handle multiple syntax specifications. The implementation of conditional compilation by carrying along two parses in a coroutine manner is also given. The utility of the model is further demonstrated by showing its adaptation to a recursive descent parser.

Key Words and Phrases: parsing, coroutines, compilers, production language, conditional compilation, extensible languages, macros
CR Categories: 4.12, 4.20, 4.32, 5.23

1. INTRODUCTION

Programming language syntax is usually specified by some form of a context-free grammar which can be parsed by any one of a number of techniques [10]. However, in addition to the primary syntax, a program text can often include other constructs which can be placed between any two tokens. In other words, we say that "they are permitted to occur anywhere in the program." Although the grammars for these constructs are relatively simple, the spontaneous nature of the constructs makes it difficult to embed their productions in parsers of most programming languages. Examples of such constructs are comments, macros, and constructs for conditional compilation. In the latter two cases, evaluation of the constructs yields components of the primary syntax.

In [5] Conway describes the organization of a compiler that makes use of a number of stages which interact in a coroutine fashion. These stages include a lexical analyzer, a parser ("diagrammer" according to [5]), a code generator, etc. In this paper we use a similar coroutine organization to facilitate the one-pass parsing of program text containing a mixture of primary syntax and extra language constructs.

This paper was motivated by a desire to implement conditional compilation in

Author's address: Department of Computer Science, University of Maryland, College Park, MD 20742.

```
const TEST = false;
      ⋮
if TEST then ⟨somecode⟩
```

Fig. 1. Program segment illustrating conditional compilation.

```
begin
      ⋮
  IFC INTEGER_ARITHMETIC THENC integer
  ELSEC real
  ENDC
  SALARY, COMMISSION;
      ⋮
end
```

Fig. 2. Another program segment illustrating conditional compilation.

a high-level language. Section 2 contains a discussion of this and similar features and the merits of a number of alternative implementations. Section 3 elaborates on the coroutine model of parsing. Section 4 demonstrates the utility of the model by showing how it can be incorporated in a production language system [10, 11]. In fact, this is how conditional compilation was implemented in SAIL [16]. Although other parsing methods have been treated similarly by the author (e.g., recursive descent [11] as shown in the appendix), production language is used because of its compact representation.[1] We conclude our presentation with a sample implementation of conditional compilation.

In the discussion that follows we refer to the syntax of the programming language as the *primary syntax.*

2. MOTIVATION

Conditional compilation, the high-level analog of conditional assembly, is a facility whereby the compile-time evaluation of a Boolean expression determines whether an associated portion of program text is translated by the compiler.

The effect of conditional compilation can be achieved using an optimizing compiler which evaluates all expressions comprised only of compile-time constants and eliminates unreachable code. Such a compiler would, for example, produce no code for the program fragment in Figure 1.

However, if conditional compilation is achieved in this way, then it can only be obtained for the conditional constructs of the programming language. For instance, conditional declarations would normally not be possible (e.g., Algol 60 [14]). In addition, the optimization capability may cause considerable compiler overhead.

Another way to achieve conditional compilation is to introduce a separate syntax for this purpose. For example, Figure 2 uses compile-time directives IFC, THENC, ELSEC, and ENDC and the compile-time variable

[1] Production language is less commonly used today for compiler construction. However, it is the subject of much research in artificial intelligence as a basis for rule-driven inference systems (e.g., [6]).

Fig. 3. Coroutine state diagrams.

INTEGER_ARITHMETIC. Here we are declaring variables SALARY and COMMISSION to be of type **integer** or **real** depending on whether the compile-time variable INTEGER_ARITHMETIC is true or false.

One can introduce compile-time variables as macros. Alternatively, one can have a compiler which always evaluates expressions comprised of constants. Even greater generality is possible when the compiler permits a combination of the previous two methods. Typically, conditional compilation units can be nested, provided that they are well bracketed.

Another common approach, as exhibited in PL/I [15], is to implement conditional compilation as a preprocessing phase. This approach, although general, is inferior to a one-pass approach because all decisions are made purely on the basis of syntactic information. In contrast, the one-pass approach enables decisions to be made on the basis of semantic information previously accumulated by the compiler. For example, the conditional compilation condition may involve querying the symbol table as to the type of a variable, whether or not a variable has been declared, etc.

Our solution to the apparently conflicting goals of (1) allowing certain constructs to appear "anywhere in the program" and yet (2) making use of semantic information in parsing these constructs is to employ a coroutine-like mechanism to parse different languages each having a different and independent grammar. A multipass algorithm fulfills the first goal; however, only a one-pass algorithm fulfills the second goal. Knuth mentions that it is often possible to transform a multipass algorithm to a one-pass algorithm using coroutines and cites space and time advantages of the one-pass solution [12]. Thus the coroutine solution allows us to meet both of our goals. This is a stronger solution than a subroutine-like interaction which is all that is necessary to handle comments or macro definitions (see [9] for a discussion of related ideas). Note that parse tokens of one syntax may be embedded within those of another syntax. For example, the conditional compilation syntax calls for a conditional expression which is a sequence of parse tokens generated by the programming language grammar. In such a case, the conditional expression is a recursive instance of the primary syntax. Also observe that if the conditional expression contains conditional compilation statements, then there is a further recursive instance of the conditional compilation syntax as well.

3. MODEL

We view the parse of each syntax as a coroutine. At any point in time, each coroutine can be in one of three states: active, suspended, or terminated. The transitions between the states are primarily determined by the definition of the programming language. Figure 3 contains a state diagram illustrating the possible transitions.

```
begin
      ⋮
  IFC A > IFC B > C THENC D
      ELSEC E
      ENDC ...
    THENC ...
  ELSEC ...
  ENDC
      ⋮
end
```

Fig. 4. Example program.

Assume that there exist n different syntax specifications corresponding to the primary syntax and $n - 1$ extralanguage constructs. At any instant, only one of the syntaxes can have its associated coroutine in an active state. The coroutines associated with the remaining $n - 1$ syntaxes are in the suspended state. Actually associated with each syntax is a stack of suspended coroutines. The potential need for a stack arises whenever a coroutine is initiated whose corresponding syntax already has a coroutine in an active or suspended state. The stack is required should the newly initiated coroutine make a transition to a suspended state. A coroutine is said to be in a terminated state when it can no longer make further transitions to other states (e.g., its corresponding code segment is exited).

Whenever a new syntax is added to a programming language, a set of reserved tokens termed initiators and resumers is designated. Initiators signal that a parse of the corresponding syntax is to start. Resumers signal that a suspended coroutine of the associated syntax is to be made active (i.e., resumed) while the currently active coroutine makes a transition to a suspended state. As an example, IFC is an initiator for conditional compilation and THENC is a resumer.

A coroutine can be initiated in two ways. One way is implicit and occurs when an initiator symbol of a syntax is encountered. Alternatively, a coroutine may be initiated explicitly by an active coroutine. For example, the grammar of the language may dictate that the primary syntax coroutine is invoked by the coroutine for conditional compilation to obtain a constant Boolean expression following an IFC symbol. The previously active coroutine enters a suspended state.

A coroutine can be resumed in two ways. The first is implicit and arises when a resumer symbol of a syntax whose coroutine is in a suspended state is encountered while a coroutine associated with another syntax is active. Second, a coroutine may be resumed explicitly by an active coroutine. For example, the grammar of the language may specify that the coroutine for the primary syntax be resumed following the processing of an ELSEC. In both cases, the previously active coroutine enters a suspended state. Note that since coroutine-like interaction is only defined between different syntaxes, we do not permit an active coroutine to resume a suspended instance of itself.

As an example of the coroutine control mechanism, consider the program fragment in Figure 4 which contains a pair of embedded conditional compilation statements. Table I contains snapshots of the status of the various coroutines as the program is being parsed. Suspended STACK 1 and coroutines J, K, and L are

Table I. Snapshots of the Parse of the Program of Figure 4

Step	Token	Active	Suspended stack 1	Suspended stack 2	Terminated
1	...	J			
2	IFC	X	J		
3	A >	K	J	X	
4	IFC	Y	K,J	X	
5	B > C	L	K,J	Y,X	
6	THENC	Y	K,J	X	L
7	assume B > C *true*	K	J	Y,X	
8	ELSEC	Y	K,J	X	
9	ENDC	K	J	X	Y
10	THENC	X	J		K
11	assume A > D *true*	J	X		
12	ELSEC ...	X	J		
13	ENDC ...	J			X

associated with the primary syntax. Suspended STACK 2 and coroutines X and Y are associated with the conditional compilation syntax. Initially, coroutine J is in the active state, and no coroutines are in the suspended states. For example, entry 6 denotes the status of the parse immediately after parsing the first THENC symbol. In this case we have the following situation:

(1) coroutine Y of the conditional compilation syntax is the active coroutine,
(2) coroutines K and J of the primary syntax are in the suspended state with coroutine K on top of the suspended stack;
(3) coroutine X of the conditional compilation syntax is in the suspended state;
(4) coroutine L of the primary syntax has just entered the terminated state.

4. EXAMPLE

To illustrate the usefulness of our model, we demonstrate how a production language parsing system can be extended to handle multiple syntaxes correctly with a minimal amount of effort. Production language has been chosen because it lends itself easily to a compact description of a programming language and the actions to be taken upon successful reductions. We first present a brief overview of production language. Next we describe the necessary extension to enable multiple syntaxes to be parsed in a coroutine manner. Finally we show how conditional compilation is implemented using these extensions.

4.1 Production Language and Its Implementation

Production language is primarily a means of describing the syntax of a programming language and the actions to be taken upon successful reductions. The principles of one such system (used in the implementation of SAIL) are illustrated by the following production. We use a variant of production language that also enables the specification of semantic action [8].

```
aa: bb cc ⇒ dd
    EXEC ROUTINE1 ROUTINE2
    SCAN 2
    CALL ee RETURN GO ff ERROR gg
```

aa is the label of the production. bb and cc are the parse tokens that must be on top of the parse stack for the reduction to be made. If a match does occur, then bb and cc are replaced by dd and the remaining part of the production is executed in sequence. If the ⇒ symbol is absent, then the occurrence of a match means that the parse stack is unchanged (equivalent to lhs ⇒ lhs). bb and cc may also be specific constants such as 0, 1, *true*, *false*, or even reserved words.

ROUTINE1 and ROUTINE2 are subroutines (usually referred to as EXEC routines) which are executed upon a successful match. They are typically used for semantic action such as code generation as well as parser state changing.

SCAN ⟨number⟩ indicates how many parse tokens must be returned by the scanner prior to executing the next production.

CALL, RETURN, and GO denote actions to be taken upon a successful match. CALL ee indicates that production ee is to be invoked recursively. It is useful when the next syntactic construct is known, and it is desired to resume the current parse once the desired construct has been obtained. RETURN is the dual of CALL and signals that the current production exits recursively upon success. GO ff indicates the next production to be executed. ERROR gg denotes the next production to be attempted upon failure (i.e., bb and cc do not match the top two entries on the parse stack). If ERROR is not used, then the immediately following production is attempted next.

To completely specify a language, all the symbols used in the reductions must be declared. There are four types of symbols: terminal symbols, reserved words, nonterminal symbols, and classes. Of these, the first three have their customary meaning and classes are sets of symbols. Each class identifier is prefixed with the @ character. When a match is attempted and a class name is seen, then any member of the class will match it.

4.2 Extension to Production Language to Handle Multiple Syntaxes

Recall from Section 3 that the coroutines associated with the syntaxes can be initiated and resumed either implicitly or explicitly. Implicit initiation and resumption are achieved by adding two more types of symbols, initiators and resumers. Initiators are specified along with the name of their corresponding syntax and the label of the production at which the initiated coroutine is to start. Both initiators and resumers are treated as reserved words and the symbol table is initialized to contain them as well as type bits indicating whether the symbols are initiators or resumers. In addition, the symbol table entry of an initiator includes the name of the syntax and the label of the associated production.

Explicit coroutine initiation is achieved by the construct INIT followed by the name of the corresponding syntax and the label of the production at which the coroutine is to start. This mechanism is analogous to a CALL with the additional task of starting a coroutine. Explicit coroutine resumption is achieved by the introduction of the RESUME construct followed by the name of a syntax. RESUME appears in a production in the same place where SCAN appears and has the meaning that the currently active coroutine is placed in a suspended state and a coroutine associated with the designated syntax makes a transition from the suspended state to an active state. In addition, the scanner is invoked to

```
IF0:                IFC ⇒ IFC (1)                                  INIT 1 CB1 GO IF1     ERROR OUT
IF1:               TRUE ⇒ TRUE1                    RESUME 1                GO IF3
IF2:              FALSE ⇒ FALSE1   EXEC SWPOFF     SCAN                    GO IF4     ERROR OUT
IF3:  @CTRUE ENDC ⇒                                                        ADIEU 1
       TRUE1 ELSEC ⇒ FALSE2        EXEC SWPOFF     SCAN                    GO IF4     ERROR OUT
IF4:                IFC ⇒ NOCOND                   SCAN                    GO IF4
      NOCOND ENDC ⇒                                SCAN                    GO IF4
      FALSE1 ELSEC ⇒ TRUE2         EXEC SWPON      RESUME 1                GO IF3
      @CFALSE ENDC ⇒               EXEC SWPON                              ADIEU 1
                SG ⇒                               SCAN                    GO IF4
CB1:            IFC                                SCAN CALL CONBEX        GO CB2     ERROR OUT
CB2:  IFC E THENC ⇒ E (2)                                                  ADIEU 2    ERROR OUT
OUT:
```

Fig. 5. Conditional compilation productions.

furnish as many parse tokens as the resumed coroutine had yet to furnish upon its most recent transition to the suspended state.

Coroutine termination is achieved by the construct ADIEU[2] followed by the name of a syntax. The effect of this construct is similar to a combination of RESUME and RETURN. The result is that the currently active coroutine is terminated, and a coroutine associated with the designated syntax makes a transition to the active state. Once again, the scanner must be prompted to furnish as many parse tokens as the resumed coroutine had yet to furnish upon its most recent transition to the suspended state.

Clearly, each instance of a coroutine that is associated with a syntax will have its own parse stack. At times it may be useful for one coroutine to examine, as well as manipulate, the parse stack of a coroutine associated with another syntax (e.g., the result of parsing the constant Boolean expression in conditional compilation; see Figure 5). This is achieved by optionally appending to each parse token in a production a number corresponding to the syntax (and hence the parse stack) to which it belongs. We assume that, when scanning a left-hand side of a production to the left, starting with the ⇒ symbol, the elements are encountered in the same order in which they appear on the top of the corresponding parse stacks. For example, execution of the production

$$\text{A (1) B} \Rightarrow \text{C (1)}$$

results in an attempt to match B with the top of the parse stack corresponding to the currently active coroutine. Next, an attempt is made to match A with the top of the parse stack associated with syntax 1. A successful match results in A (1) and B being removed from their corresponding parse stacks, and C is placed on the parse stack associated with the coroutine corresponding to syntax 1.

Finally, a capability is needed for selectively disabling and enabling implicit coroutine initiation and resumption for certain syntaxes (e.g., between the ELSEC and matching ENDC symbols of a conditional compilation statement whose constant Boolean expression has the value *true*). This is accomplished by use of EXEC routines named SWPOFF and SWPON whose arguments indicate the names of the syntaxes for which implicit coroutine initiation and resumption are to be disabled and enabled, respectively. If no arguments are specified, then

[2] This term was first used in CONNIVER [13].

implicit coroutine initiation and resumption are disabled and enabled, respectively, for all syntaxes.

4.3 Conditional Compilation

Conditional compilation has been described in Section 2. The necessary productions and symbol declarations for conditional compilation are shown in Figures 5 and 6. We label the conditional compilation syntax as syntax 2 and the primary syntax as syntax 1. The following is a scenario of the workings of the system when an IFC is encountered while in the midst of parsing the primary syntax.

Conditional compilation starts with production IF0 in control having been initiated in an implicit manner. Its primary role is to initiate another coroutine corresponding to syntax 1 to obtain a constant Boolean expression. Once such an expression has been obtained, a return is made to production IF1. Note that the IFC has been placed on the parse stack of the coroutine of syntax 1.

The actual constant Boolean expression is obtained by productions CB1 and CB2. CB1 serves to invoke the set of productions associated with parsing a constant Boolean expression. The IFC, which has been placed on the parse stack by the suspended coroutine, serves as a left context for the expression while the SCAN prior to the call to CONBEX provides a one-symbol lookahead to delimit the constant Boolean expression from the right so that the end of the expression can be detected. Note that once the constant Boolean expression has been obtained, the parse token corresponding to the expression is moved to the parse stack of the conditional compilation syntax. Also observe that the coroutine associated with obtaining the constant Boolean expression is terminated in production CB2 and the currently suspended conditional compilation coroutine is made the active coroutine.

Production IF1 corresponds to the constant Boolean expression being *true*. Therefore, the suspended coroutine (i.e., corresponding to syntax 1) is resumed,

```
⟨INITIATORS⟩
IFC    2    IFO
⟨RESUMERS⟩
ENDC 2
ELSEC 2
⟨RESERVED WORDS⟩
THENC
⟨CONSTANTS⟩
TRUE FALSE
⟨NON-TERMINAL SYMBOLS⟩
TRUE1 TRUE2 FALSE1 FALSE2 NOCOND
⟨CLASSES⟩
@CTRUE TRUE1 TRUE2
@CFALSE FALSE1 FALSE2
```

Fig. 6. Conditional compilation symbol declarations.

whereas the conditional compilation coroutine is placed in a suspended state to be resumed at production IF3.

Production IF2 corresponds to the constant Boolean expression being *false*. In this case, the suspended coroutine (corresponding to syntax 1) remains in a suspended state and the productions starting at IF4 are invoked to scan successive parse tokens until a matching ENDC or ELSEC is obtained at the same nesting level as the IFC. This step is aided by the nonterminal symbol NOCOND. EXEC routine SWPOFF results in disabling implicit coroutine initiation and resumption, i.e., when the constant Boolean expression is *false*, coroutines cannot be initiated or resumed via the scanner in the program segment between THENC and ELSEC or THENC and ENDC.

When the top two elements of the parse stack are FALSE1 and ELSEC, the suspended coroutine is resumed, and the conditional compilation coroutine is placed in a suspended state to be resumed at production IF3. When the top element of the parse stack is ENDC and the next to the top element is a member of class CFALSE, then the current conditional compilation coroutine is terminated (ADIEU) and the suspended coroutine associated with the primary syntax is resumed. Both of the previous cases result in the execution of EXEC routine SWPON, which has the opposite effect of SWPOFF. It enables implicit coroutine initiation or resumption and thus coroutines can once again be initiated or resumed via the scanner. SG is a symbol that matches all parse tokens.

The two productions starting at IF3 ensure that proper action is taken once the program segment corresponding to the value of the constant Boolean expression has been parsed. When the top element of the parse stack is ENDC, and the next to the top element is a member of class CTRUE, then the current active conditional compilation coroutine is terminated (ADIEU) and the suspended coroutine associated with the primary syntax is resumed. When the top two elements of the parse stack are TRUE1 and ELSEC, SWPOFF is executed. Then the set of productions at IF4 causes tokens to be skipped until a matching ENDC is encountered at the same nesting level.

The suffixes 1 and 2 on TRUE and FALSE aid in identifying the component of the conditional compilation statement that is currently being parsed, namely, 1 corresponds to the program segment between THENC and ELSEC or THENC and ENDC, whereas 2 corresponds to the program segment between ELSEC and ENDC. In particular, the suffixes are useful in determining proper action to be taken when an ELSEC is encountered.

5. CONCLUDING REMARKS

Similar techniques to those discussed herein have been successfully used to implement a conditional compilation facility in the SAIL compiler. In that application, use of such methods enabled the construction of a more powerful compile-time facility, namely, macros whose definitions could appear anywhere in the program text. Other features which were implemented include the compile-time equivalents of the following constructs: FOR loops, WHILE statements, CASE statements, and FOR loops based on a list of variable bindings rather than on an iterative numeric variable. In addition, the concept of two syntaxes enables SAIL macros to have recursive definitions.

Conditional compilation and macros as described here have been used to extend SAIL [3] to serve as the data manipulation language in conjunction with DBMS-10 [7], a CODASYL-based [4] database management system. The DBMS-10 database operations (termed verbs) are implemented by use of macros. The close interaction between the compiler and the compile-time facility (i.e., the one-pass characteristic rather than a preprocessor) enables the expression of the database operations in SAIL to be quite general. For example, the SAIL implementation of the verbs provides for their invocation with parameters of different type (e.g., string, integer, etc.). The macros corresponding to the verbs perform compile-time type checking and generate procedure calls to the actual DBMS-10 routines with the appropriate parameters. Thus the programmer is provided with a greater degree of flexibility than is attainable with conventional data manipulation languages such as Cobol [1] or Fortran [2].

The extension to production language described in Section 4.2 forces the specification, for each state transition, of the syntax that is being resumed. A more general scheme is one that would also permit a syntax to interact with more than one other syntax. Such a scheme can be achieved by letting the special symbol *R* denote the syntax that was most recently suspended. Given syntaxes A and B, this is equivalent to saying that once syntax B has been initiated by syntax A, all subsequent references to *R* by productions of syntax B refer to syntax A (e.g., INIT *R*, RESUME *R*, ADIEU *R*, and ⟨parse token⟩ (*R*)). Using such a method, for example, permits the conditional compilation syntax to be embedded in more than one syntax.

The concept of multiple syntaxes could also be applied to render scanner implementations more comprehensible. In particular, concepts such as lines, buffers, and even source file switching[3] could be implemented as separate syntaxes with the appropriate initiators and resumers. For example, in SAIL, source file switching is achieved by the command:

```
REQUIRE "FILENAME" SOURCE ! FILE;
```

Upon encountering the above command, the scanner obtains all subsequent parse tokens from file FILENAME. When the special "end of file" character is seen, the scanner once again obtains parse tokens from the original file. In SAIL, this feature is currently restricted to statement level. However, viewing the REQUIRE symbol as an initiator for the source file switching syntax permits source file switching to occur anywhere in the program text. The special "end of file" character serves as a resumer symbol for implicitly resuming the coroutine associated with the source file switching syntax.

Our method for parsing multiple syntaxes can be applied to other bottom-up parsing methods by employing techniques similar to those used in our adaptation of production language. To implement our method for a top-down parsing technique, such as recursive descent, requires that the parsing technique have a coroutine control structure. For an example of an implementation using a recursive descent parser, see the appendix where the productions given in Figure 5 are

[3] Source file switching is a feature which allows the programmer to specify that parse tokens are to be obtained from another file.

encoded using SAIL's multiprocessing capabilities (thereby enabling the implementation of coroutines).

APPENDIX. RECURSIVE DESCENT PARSER

In this section we present an implementation of conditional compilation for an environment employing a recursive descent parser. Once again, we only use two syntax specifications, the primary syntax and the conditional compilation syntax. The procedures are encoded in a liberal variant of SAIL. The effect of a coroutine is achieved by making use of SAIL's process constructs (e.g., **sprout** and **resume**) since the coroutine construct is not explicitly present in SAIL.

It is assumed that there exists a procedure named Parser that encodes the remaining productions of the language in which the following conditional compilation productions are embedded. CONBEX is a procedure that corresponds to a set of productions that parse a constant Boolean expression. Expression is a predicate that indicates whether or not its parse token argument is an expression. GetParseToken is a standard routine that reads characters from the input buffer. In general, the names of the procedures are identical to the labels of the productions and the names of the EXEC routines in Figure 5.

The program makes use of CoroutineTable, a two-dimensional array of type Coroutine, to store identifying names[4] for the various coroutine instances that may be in the active or suspended states. There is one row per syntax where each row is viewed as a stack. CurrentCoroutine is an array that indicates for each syntax the index of the CoroutineTable entry corresponding to the instance of the coroutine currently in the active or most recently suspended state (i.e., the column number). ParseStacks is an array of parse token stacks consisting of one stack for each syntax. It is accessed by function ParseStack relative to its top; e.g., ParseStack(0) and ParseStack(−1) correspond to the topmost and next to the topmost elements, respectively. ParseStackTop is an array of pointers to the tops of the parse stacks of the various syntaxes.

Note that rather than having one parse stack per instance of a coroutine (recall that there may be active and suspended coroutines associated with each syntax), we have one parse stack per syntax. This can be justified by observing that once a coroutine, say P, corresponding to syntax S is initiated, the parse stack of any suspended coroutine associated with syntax S will not be accessed until coroutine P has terminated.

Coroutines are initiated, resumed, and terminated by use of procedures SuspendMeAndInitiate, SuspendMeAndResume, and KillMeAndResume which correspond to INIT, RESUME, and ADIEU, respectively, of Figure 5. SuspendMeAndInitiate is invoked with parameters NewSyntax and Pname, sets CurrentSyntax to NewSyntax, and makes use of SAIL's **sprout** command to start procedure Pname in a coroutine manner. SuspendMeAndResume is invoked with parameter ResumedSyntax, which corresponds to the syntax that is to be resumed. It sets CurrentSyntax to ResumedSyntax and makes use of SAIL's

[4] The names are actually of type **item**. **item** in a SAIL data type that is analogous to a pointer. In this case, each coroutine (implemented as a SAIL process) has a unique identifying *item* associated with it. **itemvars** are SAIL variables whose value is of type **item**.

resume command to resume the corresponding coroutine. KillMeAndResume is invoked with parameter ResumedSyntax and terminates the active coroutine, sets CurrentSyntax to ResumedSyntax, and resumes the appropriate coroutine.

Note that procedures PushParseStack, PopParseStack, and ParseStack are activated with an optional parameter, as signified by the parentheses following the parameter declaration. The value enclosed by the parentheses denotes a default value when the parameter is absent. If the optional parameter is present, then it denotes the syntax whose parse stack is to be manipulated or referenced, while if absent, then the parse stack associated with syntax CurrentSyntax is manipulated or referenced.

```
begin
/* type definitions*/
   define ParseToken = "string";
   define Syntax = "integer";
   define Coroutine = "itemvar";
   define CoroutineName = "procedure";

/* constant definitions*/
   define MaxNumOfCoroutines = 100;
   define MaxParseStackSize = 50;
   define NumOfSyntaxes = 2;

/* storage declarations*/
   Coroutine array CoroutineTable [1:NumOfSyntaxes, 0:MaxNumOfCoroutines];
   ParseToken array ParseStacks [1:NumOfSyntaxes, 0:MaxParseStackSize];
   preload ! with [NumOfSyntaxes] 0; /*initialize ParseStackTop to 0*/
   integer array ParseStackTop [1: NumOfSyntaxes];
   integer array CurrentCoroutine [1: NumOfSyntaxes];
   SyntaxCurrent Syntax;
   integer I, J; /*loop variables*/
   Boolean SwappingOk;

/* procedure definitions*/
recursive procedure IF0;
begin
  if ParseStack (0) = IFC then
    begin
      PopParseStack;
      PushParseStack (IFC, 1);
      SuspendMeAndInitiate (1, CB1);
      IF1;
    end
  else OUT;
end;

recursive procedure IF1;
begin
  if ParseStack (0) = TRUE then
    begin
      PopParseStack;
      PushParseStack (TRUE1);
      SuspendMeAndResume (1);
      IF3;
    end
  else if ParseStack (0) = FALSE then
    begin
```

```
        PopParseStack;
        PushParseStack (FALSE1);
        SWPOFF;
        Scan (1);
        IF4;
      end
  else OUT;
end;

recursive procedure IF3;
begin
  if MemberCtrue (ParseStack (−1)) and ParseStack (0) = ENDC then
    begin
      PopParseStack;
      PopParseStack;
      KillMeAndResume (1);
    end
  else if ParseStack (−1) = TRUE1 and ParseStack (0) = ELSEC then
    begin
      PopParseStack;
      PopParseStack;
      PushParseStack (FALSE2);
      SWPOFF;
      Scan (1);
      IF4;
    end
  else OUT;
end;

recursive procedure IF4;
begin
  if ParseStack (0) = IFC then
    begin
      PopParseStack;
      PushParseStack (NOCOND);
      Scan (1);
      IF4;
    end
  else if ParseStack (−1) = NOCOND and ParseStack (0) = ENDC then
    begin
      PopParseStack;
      PopParseStack;
      Scan (1);
      IF4;
    end
  else if ParseStack (−1) = FALSE1 and ParseStack (0) = ELSEC then
    begin
      PopParseStack;
      PopParseStack;
      PushParseStack (TRUE2);
      SWPON;
      SuspendMeAndResume (1);
      IF3;
    end
  else if MemberCfalse (ParseStack (−1)) and ParseStack (0) = ENDC then
    begin
      PopParseStack;
```

```
        PopParseStack;
        SWPON;
        KillMeAndResume (1);
      end
    else
      begin
        PopParseStack;
        Scan (1);
        IF4;
      end;
end;

recursive procedure CB1;
begin
  if ParseStack (0) = IFC then
    begin
      Scan (1);
      CONBEX; /*get a constant Boolean expression*/
      CB2;
    end
  else OUT;
end;

recursive procedure CB2;
begin
  ParseToken Pt;
  if ParseStack (0) = THENC and Expression (ParseStack (−1)) and
    ParseStack (−2) = IFC then
    begin
      Pt := ParseStack (−1);
      PopParseStack;
      PopParseStack;
      PopParseStack;
      PushParseStack (Pt, 2);
      KillMeAndResume (2);
    end
  else OUT;
end;

procedure OUT;
PRINT ("invalid parse token");

procedure SWPOFF;
SwappingOk := FALSE;

procedure SWPON;
SwappingOk := TRUE;

recursive procedure Scan (integer ScanCount);
begin
  ParseToken Result;
  while ScanCount > 0 do
    begin
      Result := GetParseToken;
      if SwappingOk then
        begin
          if Result = IFC then
            begin
```

```
                PushParseStack (IFC, 2);
                SuspendMeAndInitiate (2, IF0);
              end
            else if Result = ELSEC or Result = ENDC then
              begin
                PushParseStack (Result, 2);
                SuspendMeAndResume (2);
              end
            else
              begin
                PushParseStack (Result);
                ScanCount := ScanCount −1;
              end;
          end
        else
          begin
            PushParseStack (Result);
            ScanCount := ScanCount −1;
          end;
      end;
end;

Boolean procedure MemberCtrue (ParseToken Token);
return (Token = TRUE1 or Token = TRUE2);

Boolean procedure MemberCfalse (ParseToken Token);
return (Token = FALSE1 or Token = FALSE2);

procedure PushParseStack (ParseToken Token; Syntax DestSyntax (Current Syntax));
begin
  ParseStackTop [DestSyntax] := ParseStackTop [DestSyntax] + 1;
  ParseStacks [DestSyntax, ParseStackTop [DestSyntax]] := Token;
end;

procedure PopParseStack (Syntax SourceSyntax (CurrentSyntax));
ParseStackTop [SourceSyntax] := ParseStackTop [SourceSyntax] − 1;

ParseToken procedure ParseStack (integer Offset;
                                 Syntax TokenSyntax (Current Syntax));
return (ParseStacks [TokenSyntax, ParseStackTop [TokenSyntax] + Offset]);

recursive procedure SuspendMeAndInitiate (Syntax NewSyntax;
                                          CoroutineName Pname);
begin
  CurrentSyntax := NewSyntax;
  CurrentCoroutine [NewSyntax] := CurrentCoroutine [New Syntax] + 1;
  /*run the new coroutine and suspend the currently active coroutine*/
  sprout (CoroutineTable [NewSyntax, CurrentCoroutine [NewSyntax]], Pname,
          suspme);
end;

recursive procedure SuspendMeAndResume (Syntax ResumedSyntax);
begin
  CurrentSyntax := ResumedSyntax;
  /*resume the suspended coroutine and suspend the active coroutine*/
  resume (CoroutineTable [ResumedSyntax, CurrentCoroutine [ResumedSyntax]],
          any);
end;
```

```
recursive procedure KillMeAndResume (Syntax ResumedSyntax);
begin
  Current Coroutine [CurrentSyntax] := CurrentCoroutine [CurrentSyntax] − 1;
  CurrentSyntax := ResumedSyntax;
  /*terminate the active coroutine and run the suspended coroutine*/
  resume (CoroutineTable [ResumedSyntax, CurrentCoroutine [ResumedSyntax]], any,
          killme);
end;
/*main body starts here*/
  for I := 1 step 1 until NumOfSyntaxes do
  /*allocate the CoroutineTable array*/
    begin
      for J := 0 step 1 until MaxNumOfCoroutines do CoroutineTable [I, J] := new;
      CurrentCoroutine [I] := 0;
    end;
  SwappingOk := TRUE;
  CurrentSyntax := 1;
  SuspendMeAndInitiate (1, Parser);
end;
```

ACKNOWLEDGMENTS

I am grateful to the Stanford Artificial Intelligence Laboratory for furnishing the necessary computer time. I have benefited greatly from discussions with Jerry Feldman, Jim Low, Robert Noonan, John Reiser, Dan Swinehart, Russ Taylor, and Randy Trigg. I also thank Sue Graham for her editing help.

REFERENCES

1. American National Standard Programming Language COBOL X3.23-1974. American National Standards Institute, Inc., New York, 1974.
2. American National Standard FORTRAN. American National Standards Institute, New York, 1966.
3. BUCHANAN, J., FENNELL, R.D., AND SAMET, H. A data base management system for the federal courts. Harvard Graduate School of Business Administration, Harvard University, Cambridge, Mass. Submitted for publication.
4. CODASYL Database Task Group. CODASYL Database Task Group Report, April 1971 (available from ACM, New York).
5. CONWAY, M.E. Design of a separable transition-diagram compiler. *Commun. ACM 6*, 7 (July 1963), 396–408.
6. DAVIS, R., AND KING, J. An overview of production systems. Stanford Computer Science Dep., Stanford Univ., Stanford, Calif., Artificial Intelligence Project Memo AIM-271, 1975.
7. DEC system 10, data base management system programmer's procedures manual. Digital Equipment Corp., Maynard, Mass., Doc. DEC-10-APPMA-B-D.
8. FELDMAN, J.A. A formal semantics for computer languages and its application in a compiler-compiler. *Commun. ACM 9*, 1 (Jan. 1966), 3–9.
9. FISHER, D.A. Control structures for programming languages. Ph.D. dissertation, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., 1970, p. 163.
10. FLOYD, R.W. A descriptive language for symbol manipulation. *J. ACM 8*, 4 (Oct. 1961), 579–584.
11. GRIES, D. *Compiler Construction for Digital Computers.* Wiley, New York, 1971.
12. KNUTH, D.E. *The Art of Computer Programming: Fundamental Algorithms, Vol.* 1. Addison-Wesley, Reading, Mass., 1973, p. 195.
13. MCDERMOTT, D.V., AND SUSSMAN, G.J. The Conniver reference manual. AI Memo 259, M.I.T. Project MAC, M.I.T., Cambridge, Mass. May 1972.

14. NAUR, P. (Ed.) Revised report on the algorithmic language ALGOL 60. *Commun. ACM 3*, 5 (May 1960), 299–314.
15. IBM. PL/I language specifications. Order No. GY33-6033, IBM Corp., New York, 1971.
16. REISER, J.F. (Ed.) SAIL. Stanford Artificial Intelligence Project Memo AIM-289, Computer Science Dep., Stanford Univ., Stanford, Calif., 1976.

Received April 1979; revised November 1979 and February 1980; accepted February 1980