PROCEDURE WRITING: HOW TO IMPLEMENT CUSTOM PROCEDURE STATEMENTS

Richard L. Gimarc, Boole & Babbage

0. Abstract

This paper presents a technique for implementing custom procedure statements in user-written SAS procedures. The steps required to implement a custom procedure statement are presented with sufficient detail such that potential procedure writers will be able to design and implement user-friendly procedures with non-trivial supplementary procedure statements. The design and implementation of a custom procedure statement for PROC M204 is used as an example of the technique presented.

1. Introduction

SAS is one of the few software packages that encourages its users to extend its capabilities by adding their own code to the SAS system. User code can be integrated into SAS to provide new procedures, formats, functions, and CALLED procedures. Of these, user-written procedures appear to be the most popular (or publicized?). One only needs to browse past SAS User's Group International (SUGI) conference proceedings or the SAS Supplemental Library User's Guide to be convinced of this.

The most visible component of a user-written procedure is its user interface, that is, the SAS code required to invoke the procedure. The user interface should be simple to use and flexible. Simplicity and flexibility are essential. Without them, the procedure will not reach the audience for which it was intended.

During the design of a SAS procedure the procedure writer should strive to make it appear as 'SAS-like' as possible. That is, the PROC statement and any supplemental procedure statements should maintain the style and flavor of the SAS language. Any radical departure from that style will make the procedure difficult to use and earmark it as one of the 'oddball' user-written SAS procedures.

This paper addresses one aspect of procedure writing: how to design and implement custom procedure statements. A custom procedure statement is a supplemental procedure statement which is used to supply additional information to a SAS procedure in a non-list format. (A list format statement consists of a keyword followed by an unordered list of items such as a VAR statement. The PLOT statement of PROC PLOT is an example of a non-list statement; its syntax allows for more than just a list of items.)

2. Procedure Writing Overview

A user-written SAS procedure consists of two components: the parser module and the procedure module. The parser module is responsible for parsing the control statements supplied by the user in his SAS program. The procedure module is the workhorse of the PROC. It uses the information gathered by the parser to read the necessary input, perform the requested computations, and output the results.

The conventions to be followed and the interfaces available for use by the parser and procedure modules are described in the SAS Programmer's Guide. This manual, however, is rather brief in its discussion of a custom statement parser. In the sections that follow a more complete discussion of the steps involved in writing a custom statement parser will be given.

3. Compiling a Custom SAS Statement

A user-written procedure must read, verify, and interpret any custom procedure statements supplied by the user. In other words, the procedure must compile the custom procedure statements.

Compiler theory advocates viewing the compilation process as a sequence of phases. Each phase takes as input one representation of the source program and produces as output another representation. The commonly accepted compiler phases are

1. lexical analysis
2. syntax analysis
3. intermediate code generation
4. code optimization
5. code generation

The lexical analysis phase reads the source program and partitions it into logical groups of characters. These groups of characters are called 'words' in the SAS environment (or tokens, for compiler theory enthusiasts). Syntax analysis reads the words produced by the
lexical analyzer and builds syntactic structures. The final three phases take the syntactic structures and transform them into executable code.

The primary purpose of a custom statement parser is to examine statements unique to a given SAS PROC and pass to the procedure module information regarding the content of the statements. This requires only phases 1 and 2. Phases 4 and 5 will not be discussed since they are not relevant to this discussion of a custom statement parser. Phase 3 will be discussed in Section 5 where a sample custom statement parser is presented.

Most of the processing required for lexical analysis is performed by the SAS parsing mechanism. As described in the next section, the SAS parser partitions each source statement into words. What remains is for the PROC's parser or procedure module to perform the syntax analysis.

Syntax analysis will take as input the words output by the lexical analyzer and build syntactic structures. Syntax analysis can be performed in one of two places, the PROC's parser or procedure module.

Performing syntax analysis in the parser module enables errors to be detected as early as possible. There are two disadvantages to this approach though. First, the parser must have access to all information necessary to detect valid syntactic structures. This may not always be possible. And second, the syntax analyzer must be written in assembler and must logically fit into the structure of the parser. This is a non-trivial task.

The alternative is to perform syntax analysis in the procedure module. This approach allows the procedure writer the luxury of writing the syntax analyzer in a high-level language and guaranteeing that the syntax analyzer has access to all information necessary to perform syntax analysis. This paper will discuss performing syntax analysis in the PROC's procedure module.

4. SAS Statement Components

The first step to be performed in compiling a custom procedure statement is to partition the statement into words (lexical analysis). This section discusses the partitioning process.

The SAS parsing mechanism scans the procedure information statements appearing in the user's SAS program and partitions each statement in logical groups of characters called words. A word data structure (WDS) is built for each word (see Figure 1) and contains all the pertinent information about a word viewed in isolation. Field WIT indicates the type of word. A word can be a label, literal, special symbol, integer, or floating point number. Numbers are characterized as either floating point or integer. In either case, both the internal and external representations of the number are stored in WII/WIF and WII, respectively. The actual character string representation of the word as seen in the source is stored left-justified and padded with blanks in WII with its length in WIL.

The SAS parsing mechanism provides the parser with a 3-word window of each source statement. That is, only three words are presented to the parser via their respective WDS's at any one time. Word 1 is the current word, word 2 is the next word, and word 3 is the last word in the window. The SAS WORD macro is used to advance the 3-word window across a statement. By repeatedly using the WORD macro the window can be advanced across the statement until end-of-statement is detected by word 1 being a semicolon.

Words are the basic components of a custom procedure statement. In the sample parser discussed in the next section, a technique will be presented which enables the parser and the procedure modules to examine the sequence of words output by the SAS parsing mechanism. Also, a technique for verifying and interpreting the syntactic structure of the string of words (a custom procedure statement) will be presented.

5. An Example: PROC M204

PROC M204 was presented a SUGI '83 [2]. Briefly, this procedure provides an interface to the Model 204 database management system. With this procedure a SAS user can build a SAS data set containing records extracted from a Model 204 database. A custom statement parser was written for this procedure to handle an IF statement. The implementation of the IF statement parser will now be discussed.

5.1 IF Statement

The IF statement is used by PROC M204 to specify a selection criteria for extracting records from a Model 204 database. The selection criteria specified by the user will be translated into a Model 204 retrieval specification and passed to Model 204 for record retrieval.

One of the design goals of PROC M204 was
to make it as 'SAS-like' as possible. Hence, the format and function of the PROC M204 IF statement was chosen to be that of SAS's subsetting IF statement. This required the following:

1. A grammar to describe the syntax of the IF statement.
2. A syntax analysis phase to verify the IF statement syntax.
3. An intermediate code generation phase to translate the IF statement into a Model 204 retrieval specification.

The implementation of the compiler phases required to compile the PROC M204 IF statement are discussed next.

5.2 Lexical Analysis

Figure 2 shows a partial listing of the parser module for PROC M204. Processing of an IF statement proceeds as follows:

1. The SAS parsing mechanism scans the word IF. The SAS supervisor does not recognize this word and passes control to the PROC M204 parser through the statement recognition exit STMTREC, label CKSTMT, specified on the SASEND macro.

2. The STMTRRC exit establishes addressability and examines the current word (word 1). If word 1 is IF, then this is an IF statement and control passes to label IFSTMT. Otherwise, the SAS parser will flag the statement as unrecognized and invalid.

3. IF statement processing begins at label IFSTMT. Before scanning the words of the statement the IF statement data structure IF STMT (see Figure 3) is initialized using the STMTST routine.

4. Next, the IF statement is scanned word by word. Each word of the IF statement is saved in the IF statement data structure field IF_WORD_AREA along with extra fields from the WDS describing the word type and length. These words will be read by the syntax analyzer to verify syntactic structure.

5. When the semi-colon is read, signalling end-of-statement, the IF statement structure is linked into the statement structure list using the STMTEND routine and control returns to the SAS parser.

5.3 Syntax Analysis

Syntax analysis of the IF statement is performed in the PROC's procedure module. This phase will read the words saved by the lexical analyzer (M204 parser) and verify the syntactic structure.

The first step is to read the words stored by the parser. Figure 4 shows the PL/I code necessary for this. The SAS STLIST routine is called to obtain the address of the first statement structure in the statement structure list. Each structure in the list is examined. If a structure is for an IF statement, then the partial WDS's stored by the parser into IF_WORD_AREA can be retrieved one by one by noting the word length (IF_LENGTH) and adjusting the structure pointer IF_WORD_PTR accordingly.

To verify syntactic structure, a grammar describing the format of the IF statement was required. The grammar developed is a subset of SAS's subsetting IF statement and is shown in Figure 5.

Given a grammar and the sequence of words output by the lexical analyzer, the next step is to determine whether the sequence of words form syntactically valid IF statements. That is, the statement derivation illustrated above must be automated. A picture parser, or sometimes called a tabular recursive descent parser, was developed for this task.

5.4 Picture Parser

The first step in developing a picture parser for the IF statement grammar was to transform the grammar into a set of transition diagrams (see Figure 6). The transition diagrams are a collection of finite state machines (FSM's). For the grammar developed, each production was transformed into an FSM. Each state is labeled with a state number and the arcs between states are labeled with a word or the name of another FSM. Syntax analysis is performed by reading the sequence of words output by the lexical analyzer and traversing the FSM's describing the IF statement grammar. The algorithm given below describes the operation of the picture parser.

5.5 Algorithm: Picture Parser

1. Initialization: Set current state S to state 1. Set current word W to the value of the first word saved by the lexical analyzer.

2. Given the current word W, examine the
arcs leaving the current state S. If an arc is labeled with W, then traverse the arc, transition to the new state, and fetch the next word. If an arc is labeled with the name of another FSM, then perform a FSM 'call'. If no arcs are labeled with W or the name of another FSM, but there is an ε-arc, then traverse the ε-arc and transition to the new state. Otherwise, a syntax error has been detected.

3. If the current state S is zero, then perform a FSM 'return'.

4. Go to step 2.

A FSM 'call' is performed as follows. The location of the arc making the FSM call is PUSHed onto a stack. The current word W remains unchanged and the current state S is set to the initial state of the called FSM. A FSM 'return' is performed when the current state is zero (step 3). In this case the top arc location is POPed from the stack and traversed. Traversing the arc results in a new current state S being set but the current word W remains unchanged.

The steps of the picture parser algorithm are repeated until all valid state transitions have been made. A syntax error is detected when a FSM 'return' is performed with an empty stack or the algorithm becomes stuck in a state, unable to find an arc to traverse. If the algorithm terminates with current state S equal to 15 (the terminal state) and an empty stack, then an IF statement has successfully been parsed. Otherwise, a syntax error has been detected.

As an example, Figure 7 shows a trace of the steps performed in parsing the following IF statement:

IF SALARY = 10000;

This discussion has deliberately been kept brief. The interested reader is referred to [1] for a more thorough discussion of syntax analysis.

5.6 Intermediate Code Generation

The intermediate code generation phase transforms the syntactic structures built by the syntax analysis phase into a Model 204 retrieval specification. A retrieval specification is a character string which contains the selection criteria to be used to retrieve records from a Model 204 database. For example, the Model 204 retrieval specification for the sample PROC M204 IF statement shown in the last section is

'SALARY IS 10000;'

This transformation is easy with a picture parser. Each time an arc is traversed by the picture parser, the syntax analyzer knows a little bit more about the structure of the statement. This allows intermediate code generation to be performed concurrently with syntax analysis by attaching semantic actions to the arcs of the picture parser's transition diagrams so that whenever an arc with an attached semantic action is traversed the semantic action is performed.

The semantic actions required to transform the sample IF statement shown above are as follows:

1. Attach to 'name' arc
   STRING := STRING || name

2. Attach to '=' arc
   STRING := STRING || ' IS '

3. Attach to 'number' arc
   STRING := STRING || number

4. Attach to ';' arc
   STRING := STRING || ';'

Thus, as the picture parser traverses the transition diagrams, the Model 204 retrieval specification will be built up in the character string named STRING. After the last IF statement is parsed, STRING can be passed to Model 204 for record selection.

6. Conclusion

The importance of designing a user-friendly user interface for a SAS procedure cannot be over-emphasized. The use of custom procedure statements is one way to do this. The steps required for writing a custom statement parser have been presented and illustrated with a discussion of the PROC M204 IF statement parser. This example should serve as a working model for future SAS procedure writers.

7. Acknowledgements

The author would like to thank Steve Ryder of the Texas Education Agency for taking the time to review and critique this paper.
Figure 1. Word data structure for word 1.

Figure 2. Partial PROC M204 statement parser.
CALL STLIST( IF_STMT_PTR );
DO WHILE ( UNSPEC( IF_STMT_PTR ) ^= 'O'B );
  IF IF_NAME ^= 'IF ' THEN DO;
    DONE = #FALSE#;
    IF_STMT_PTR = ADDR( IF_STMT_PTR );
    DO WHILE ( DONE );
      /* Process word in IF_STMT. DONE will be set */
      /* to #TRUE# when the WDS containing the */
      /* semi-colon marking the end-of-statement */
      /* is seen. */
      IF_STMT_PTR = ADDR( IF_STMT_PTR );
    /* chain forward to next WDS */
    IF_STMT_PTR = ADDR( SUBSTR( IF_STMT_CHARS, IF_STMT_PTR+1, 1 ) );
  END;
  IF_STMT_PTR = ADDR( IF_STMT_PTR ); /* chain to next statement structure */
END;
Figure 4. PL/1 code to access IF statement words.

**Figure 5. PROC M204 IF statement grammar.**

**Figure 3. IF statement data structures.**

```plaintext
DCL 1 IF_STMT BASED(IF_STMT_PTR),
  IF_NEXT PTR, /* ptr to nxt stmt */
  IF_NAME CHAR(4), /* CL4 'IF ' */
  IF_FILLER CHAR(4), /* unused */
  IF_OPTS PTR, /* ptr to stmt end */
  IF_ENDIAN PTR, /* stmt options */
  IF_PARM FLOAT B(53), /* stmt parameters */
  IF_WORD_PTR CHAR(1); /* stmt WDS's */

DCL 1 IF_WDS BASED(IF_WDS_PTR),
  IF_LENGTH FIXED BIN(15), /* word length */
  IF_FILLER BIT(8), /* unused */
  IF_TYPE BIT(8), /* word type */
  IF_WORD_CHARS CHAR(200); /* chars in word */
```
Figure 6. PROC M204 IF statement transition diagrams.

<table>
<thead>
<tr>
<th>Current Word</th>
<th>Current State</th>
<th>Stack</th>
<th>Next Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>1</td>
<td>(empty)</td>
<td>traverse 'IF' arc</td>
</tr>
<tr>
<td>SALARY</td>
<td>2</td>
<td>(empty)</td>
<td>call 'ex' FSM</td>
</tr>
<tr>
<td>SALARY</td>
<td>4</td>
<td>ex</td>
<td>call 'se' FSM</td>
</tr>
<tr>
<td>SALARY</td>
<td>7</td>
<td>se/ex</td>
<td>call 'tm' FSM</td>
</tr>
<tr>
<td>=</td>
<td>10</td>
<td>tm/se/ex</td>
<td>traverse 'name' arc</td>
</tr>
<tr>
<td>10000</td>
<td>12</td>
<td>tm/se/ex</td>
<td>traverse 'number' arc</td>
</tr>
<tr>
<td>;</td>
<td>0</td>
<td>tm/se/ex</td>
<td>return from 'tm' FSM; traverse 'tm' arc</td>
</tr>
<tr>
<td>;</td>
<td>8</td>
<td>se/ex</td>
<td>traverse 'se' arc</td>
</tr>
<tr>
<td>;</td>
<td>0</td>
<td>se/ex</td>
<td>return from 'se' FSM; traverse 'ex' arc</td>
</tr>
<tr>
<td>;</td>
<td>5</td>
<td>ex</td>
<td>traverse 'ex' arc</td>
</tr>
<tr>
<td>;</td>
<td>0</td>
<td>ex</td>
<td>return from 'ex' FSM; traverse 'ex' arc</td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td>(empty)</td>
<td>traverse ';' arc</td>
</tr>
<tr>
<td>(none)</td>
<td>15</td>
<td>(empty)</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Where:  
ex = expression  
se = simple_expression  
tm = term

Figure 7. IF statement parse trace.