A PRACTICAL PARSER WITH COMBINED PARSING TECHNIQUES

Donghui Wang

RTI International, 3040 E. Cornwallis Road, P.O. Box 12194, Research Triangle Park, NC 27709

ABSTRACT

This paper introduces a practical solution for dramatically enlarging the capabilities of an established parser, a task that presents substantial challenges. During the development of new procedures for SUDAAN®, a commercial statistical software package, we found the existing parser to be inadequate for new situations. Like many other parsers, the one in use could be characterized as a no-repair, no-guesswork, and no-backtracking look-ahead left-to-right LALR(1) parser [1, p. 300]. This paper describes how the parser was enhanced to handle extra syntax for sophisticated mathematical and logical expressions. The new parser adds a noncanonical parsing technique, along with a Shunting-Yard-style algorithm and other techniques as a second step after the original canonical LALR [2], resulting in a powerful and efficient two-level parsing approach. Adding a second step to the successful one-step parser offered a way to preserve existing, well-tested capabilities while adding capabilities for parsing more complex syntax.

KEYWORDS

Noncanonical, LALR(1), SUDAAN, parser, parser expansion, operator precedence

1. INTRODUCTION

The canonical look-ahead left-to-right (LALR) parsing technique has been widely used in computer language processing for its simplicity and efficiency [1, pp. 300–313]. SUDAAN, a statistical software package developed and distributed by RTI International [3], first applied the LR parsing technique in the early 1980s, and then gradually transformed its parser to a canonical LALR(1) parser. For the past several development cycles, the SUDAAN parser has remained deterministic, with no repair, no guesswork, and no backtracking. The LALR(1) parser has been able to handle all SUDAAN syntax without dramatic expansion until recently.

When demands for greater functionality mounted from users, the SUDAAN team decided to extend the syntax to allow user-defined functions in some statements, including complicated and nested mathematical and logical expressions. Because the numerous current users were highly familiar with SUDAAN syntax, the team decided to avoid any major syntax change. Thus, the range of new symbols that could be added to the extended syntax was limited, and some of the existing symbols had to be recycled for additional use.

The SUDAAN LALR(1) parser, with its narrow look-ahead window, would encounter many conflicts within the proposed, more flexible syntax. After considering alternatives, the team chose to retain the LR parsing technique as the basic approach, while extending the look-ahead window when necessary to resolve syntactic conflicts. This leads to a noncanonical LALR approach.

DOI : 10.5121/ijsea.2015.6401

Unlike other SUDAAN procedures, the newly developed procedures allow the user to define unlimited statements with complicated mathematical and logical expressions that involve unlimited variables from the data set as well as the user-defined temporary variables. The only restriction is the amount of available memory space in the user's system.

Allowing the user free rein raises a challenge for any parser. The length and complexity of each statement may grow unpredictably. To prevent unacceptable delays, the parser must manage tokens efficiently, check for ambiguous syntax, check for relationships among user-defined variables, and ensure that the time required for processing stays linear.

To overcome these challenges, we put more sophisticated parsing techniques into practice and made them work together efficiently. The techniques include noncanonical parsing [1, pp. 343–379], Shunting-Yard-style algorithm [2], and operator precedence parsing [1, pp. 266–272]. The SUDAAN team combined new techniques with the existing LALR parser to form a combined parser. We assigned a second meaning to some symbols to keep current syntax unchanged. We expanded the types of statements that could be parsed. In addition, we introduced a new concept, in that user-defined variables can be referenced within the same procedure call by other statements.

This manuscript describes the functionality of the new SUDAAN parser; to do so, it is necessary to provide precise vocabulary as well as mathematical context. The paper is organized as follows. Section 2 presents terminology, basic definitions, and SUDAAN-specific terminology and definitions. Section 3 presents the strategy for choosing suitable techniques. Section 4 presents the grammars, parsing steps, and algorithm to construct the new SUDAAN combined parser. Section 5 discusses the algorithm's efficiency, followed by the conclusion in Section 6.

2. TERMINOLOGY OF PARSING

Because the language of parsing is not familiar to everyone but is extremely important in this discussion, we first offer a brief glossary of terms. These terms and a more comprehensive discussion of parsing can be found in parsing textbooks, such as [1].

- **Bottom-up parsing:** A parsing approach that examines the lowest level of detail first, followed by more complex structures.
- **Canonical LR parser:** An LR(k) parser with a single look-ahead terminal, where k represents the number of symbols considered in look-aheads.
- **LALR(k) parser**: A look-ahead LR(0) parser where k indicates the number of symbols considered in look-aheads. The term k is often omitted when k = 1.
- **LR parser**: A bottom-up parser that parses from the left with reduction of the rightmost derivation (handle) as soon as a derivation is recognized.
- **Noncanonical LR parser:** An LR parser with a larger set of grammars, as compared with its canonical counterpart, which sometimes postpones decisions that would otherwise be required to create parse trees in post-order.
- **Nonterminal:** A symbol that can be replaced by the parsing system.
- **Productions:** Rules that describe valid strings for the parsing system.
- **Sentential form:** A string that begins with a start symbol, contains at least one nonterminal symbol, and ends with a terminal symbol.
- **Start:** A symbol that indicates the beginning of the string to be parsed.
- **Terminal:** A symbol that may appear in the inputs to or outputs from the production rules of a formal grammar and that cannot be changed under the rules of the grammar.

- **Context-free grammar**: A context-free grammar can be represented by (V_n, V_t, P, S), where V_n is a finite nonempty set of symbols called *nonterminals*, V_t is a finite set of symbols distinct from those in V_n called *terminals*, P is a finite set of pairs called productions, and S is a distinguished symbol in V_n called the *start symbol*. Each production is written $A \rightarrow x$ and has a left part A in V_n and a right part x in V*, where $V = V_n \cup V_t$. V* denotes the set of all strings composed of symbols in V [4].
- **LALR(1) parser**: A look-ahead LR(0) parser with a look-ahead of 1 token. The LALR(1) parser is very powerful—almost as powerful as LR(1), yet it does not require a large amount of memory, and it is time efficient. The LALR(1) parsing technique may very well be the most used parsing method in the world today. Probably the most famous LALR(1) parser generators are *yacc* and its GNU version *bison* [1, p. 301].
- **Chomsky Normal Form**: A grammar where every production is either of the form $A \rightarrow BC$ or $A \rightarrow c$ (where A, B, C are arbitrary variables, and c is an arbitrary symbol).

3. CHOOSING A SUITABLE TECHNIQUE

The most important criteria for choosing suitable techniques for the extended parser is performance. It would be meaningless if the users have to endure unbearable execution times for new functionalities. To ensure good performance, we decided to choose techniques that guarantee the linearity of the combined parser. Some grammars offer optimal linear parsing, such as the synchronous context-free grammars (SCFGs) [5]. SCFGs can be seen as a natural extension of the well-known rewriting formalism of context-free grammars (CFGs) [6]. An SCFG is a string rewriting system based on synchronous rules. Informally, a synchronous rule is composed of two CFG rules along with a bijective pairing between all the occurrences of the nonterminal symbols in the right-hand side of the first rule and all the occurrences of the nonterminal symbols in the right-hand side of the second rule [6]. For our purpose of extending the existing parser, SCFG seems to be more than what we need. Thus, we stay with CFG grammars because CFG is what the existing parser has been using.

For the algorithm, we decided to implement Shunting-Yard over other algorithm choices for parsing mathematical expressions. Other algorithms that can be used for content-free grammars and for parsing mathematical expressions include, for example, the Cocke-Younger-Kasami (CYK) parsing algorithm [7]. The CYK algorithm has simple and elegant structure, and the original algorithm is based on multiplication of matrices over sets of nonterminals. It has also been proven that the CYK algorithm can use matrices over sets of parse trees [8]. The CYK algorithm, however, requires that the grammars have to be in Chomsky Normal Form. This would add extra burden to the development work. We decided to use the operator precedence parsing approach because it fits the nature of SUDAAN syntax. Therefore, the Shunting-Yard algorithm becomes the best candidate.

Additionally, we chose not to replace the existing parser. Instead, we extended the existing parser and used the basic bottom-up parsing approach with Shunting-Yard algorithm to form a balanced parser for the software package.

4. SUDAAN PARSER EXPANSION

Until recently, the SUDAAN statistical package allowed only simple syntax, for which it needed only a simple and straightforward LALR(1) parser. As user demand grew for greater flexibility and functionality, the SUDAAN team decided to expand its syntax dramatically, including an extension to noncanonical parsing methods. As a result, the SUDAAN parser now contains two major parts: a level 1 parser, and a level 2 parser.

4.1. SUDAAN Syntax

Table 1 gives a brief example of SUDAAN programming. This part of the program takes an input file that contains personal demographic data, sums up person's weight and height, and calculates body mass index (BMI).

Table 1. SUDAAN program example

```
1 PROC VARGEN DATA = "mydata" DESIGN = WR;
2   XSUM      weight: myweight / NAME = "Weight Total";
3   XSUM      height: myheight / NAME = "Height Total";
4   PARAMETER bmi: weight/height / NAME = "BMI";
```

The glossary below defines the terminology used to explain how parsing works for this example. Some of these terms and phrases have a more general meaning outside of the SUDAAN system. See [3] for more complete coverage of SUDAAN syntax, statements, and procedures.

- **Token**: SUDAAN tokens are reserved words for procedures and statements, literals, strings, operators, and user input texts.
- **Statements**: The lines that contain identifier definition and executable code. Each statement ends with a semicolon. Multiple statements can be on the same input line as long as they are separated by a semicolon.
- **Procedural statement**: Any statement that contains a procedure identifier and parameters.
- **Computational statement**: Any statement that contains a function identifier, variable, expression, or option.
- **Terminal symbol matrix**: This matrix is designed to define the relationship between terminal symbols. Moreover, it defines ways in which symbols are allowed to follow each other. For example, the symbol <and> indicates the "AND" operator, and the symbol <or> indicates the "OR" operator. These two cannot follow each other. However, the symbol <and> is allowed to follow the symbol <var> which indicates a data set variable. This design enables the parser to quickly pick up syntax errors and speed the process.

4.2. SUDAAN's Parsing Process

Consider the example statements of Table 1. As is commonly done in parsing, SUDAAN's scanner tokenizes all symbols from the input program file from left to right, moving from top to bottom until all input is processed. Soon after the scanner finishes its work, SUDAAN performs initial reductions according to its basic reduction rules. All symbols are reduced to terminal symbols. Symbols that are ambiguous are examined again by the parser to make a final decision.
After scanning, the software proceeds with two-step parsing, as mentioned in the introduction of this paper. The level 1 parser is the simple canonical parser that was carried into this version of the software from prior versions. It handles all procedural-level syntax, including procedure name, data source, design specifics, output statements, and all noncomputational statements. It also handles all reserved statement names. It does not deal with contents of computational statements or user-defined variables or functions. The level 1 parser applies full reduction rules to the tokenized input symbols sent by the scanner and tries to resolve nondeterministic symbols. However, without peeking and backtracking, some symbols remain nondeterministic. The level 1 parser processes the procedural statements along with the reserved computational statement names like XSUM and PARAMETER in statements 2, 3, and 4; the user-defined names on the left side of the colon ("weight," "height," "sbmi," and "sbmi1"); and label assignment (NAME = "BMI").

Following completion of level 1 parsing, the process moves to level 2. The level 2 parser is the noncanonical parser that was developed to meet the challenge of growing user demand. It only deals with computational statements, particularly the contents on the right side of the colon character, as shown in statements 2, 3, and 4 in Table 1. The level 2 parser resolves all undetermined symbols. If a symbol is still nondeterministic after second-level parsing, the program halts, and an error is issued. In the newly expanded parser, SUDAAN has adapted the precedence parsing technique [1, pp. 266–272] and constructed an operator-precedence table. The parser will either reduce or shift based on the rank of the precedence of the terminal symbol. This parsing step is described in greater technical detail below.

4.3. Construction Principles

Look again at the program excerpt shown in Table 1. It calculates the human BMI for different strata—that is, for different groupings of people. For example, the data for a gender variable can be stratified into male and female strata. Statements 2 and 3 summarize the people's weights and heights, and statement 4 computes the BMI.

Consider the syntax of statement 4, where PARAMETER is the statement name and "bmi" is the temporary variable that will contain the result of the computation:

PARAMETER bmi: weight/height / NAME = "BMI";

The colon indicates the start of the computation part of the statement, and the semicolon indicates the end of the statement. The SUDAAN scanner tokenizes the statement, and SUDAAN makes the basic reductions.

After the scanner tokenizes the statement, the input text becomes:

<reservedWord> <var> <colon> <var> <slash> <var> <slash> <reservedWord> <equals> <string> <semicolon>
Letting $ be the start symbol, the basic SUDAAN production rules are:

- $ → PARAMETER_STMT;
- PARAMETER_STMT → <reservedWord> <var> <colon> EXPRESSION <slash> OPTION_LIST
- OPTION_LIST → OPTION
- OPTION → <reservedWord> <equals> <string>
- EXPRESSION → DIVISION
- DIVISION → EXPRESSION <slash> EXPRESSION
- EXPRESSION → <var>

Based on the production rules, we can construct a complete parsing tree of statement 4 from Table 1; this tree is shown in Figure 1. SUDAAN scans and parses the input text at the bottom of the parsing tree in one forward pass. The parser builds the parsing tree from bottom up and left to right. Subparsing trees exist within the main parser tree, such as EXPRESSION, OPTION_LIST, and branches like <colon>. The combined parser links every branch and subtree together during its pass through the statement.

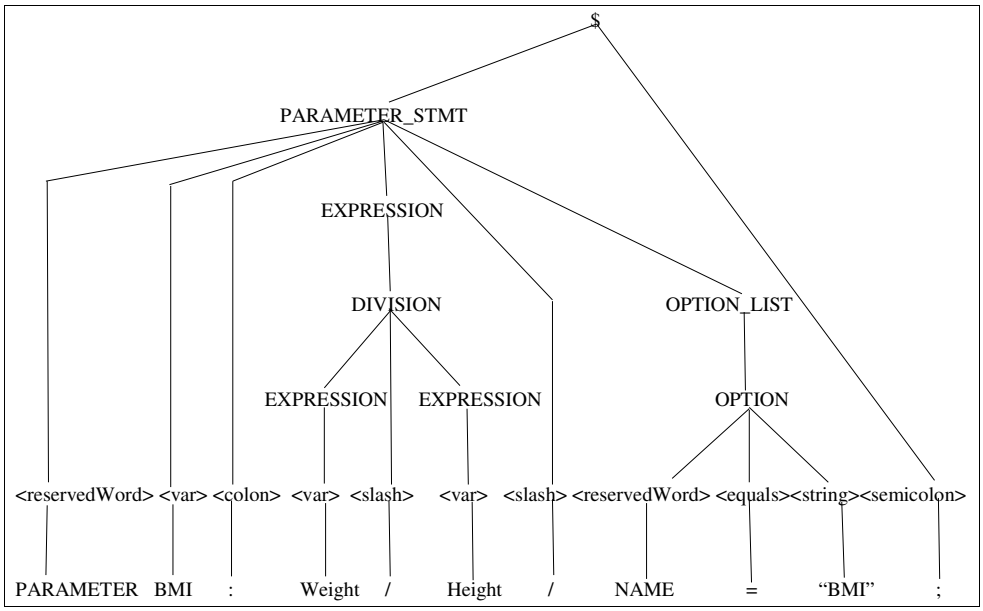

Figure 1. Complete parsing tree for statement 4

In the parser's one forward pass, there are some ambiguous symbols that need to be handled. For example, we can extract the reduction rules from the parsing tree as:

1) / → <slash> → EXPRESSION → DIVISION → EXPRESSION → PARAMETER_STMT → $
2) / → <slash> → PARAMETER_STMT → $

Notice that the symbol "/" in (1) and (2) is initially reduced to <slash>. The symbol be further reduced to either division or separator, depending on the symbol that comes next. To choose between the two reductions, the parser must know which terminal symbol will follow—a symbol for a variable or a symbol for a reserved word. The parser solves the conflict by associating the distinct look-aheads' height and NAME with the reduction to <var> and <reservedWord>, respectively.

Some basic rules must be followed to form the noncanonical level 2 parser look-ahead set. Most important, no null symbols are allowed in the set because the need is to produce totally reduced look-aheads. The totally reduced look-aheads form a subset of the noncanonical look-ahead set such that none of its elements can be further reduced [9].

4.4. Parsing Steps

To show how the combined parser works, consider statement 4 of Table 1:

PARAMETER bmi: weight/height / NAME = "BMI";

The level 1 parser starts processing the input symbols. The first symbol is PARAMETER, which is reduced to <reservedWord>, and the parser identifies it as the statement identifier. Because the reserved word is PARAMETER, the parser recognizes that this is a PARAMETER statement; it cannot be any other statement. The parser is expecting a variable to follow; if not, the parsing halts, and an error is given. In our case, the next symbol is "bmi"; it is reduced to <var>, and

parsing proceeds forward. The parser is expecting that the next symbol is a colon, which it is, so parsing proceeds forward. The parser is now expecting to parse EXPRESSION next, based on the production rule for a PARAMETER statement. The parser hands over the process to a level 2 parser to handle EXPRESSION.

The first symbol passed into the level 2 parser is Weight, and the scanner recognizes it as a variable, <var>. The next symbol is /, which is scanned as <slash>. The level 2 parser peeks ahead to determine whether the <slash> is an operator that is part of EXPRESSION, and it finds that the next symbol is Height, another variable. Using a Shunting-Yard algorithm, the parser looks to see if the expression continues. The next token is the second <slash>. As before, the parser peeks ahead to determine whether this <slash> is an operator that is part of EXPRESSION. The next token is a <reservedWord>, which determines that this <slash> is not an operator; therefore, the EXPRESSION has ended. The parser reduces <var> <slash> <var> to DIVISION and then reduces DIVISION to EXPRESSION and passes control back to the level 1 parser to complete parsing the PARAMETER statement.

Based on the production rules, the level 1 parser needs to parse the OPTION_LIST. The next symbol passed in is NAME, which is reduced to <reservedWord>, and the parser recognizes it as the option identifier. The parser is expecting an equal symbol to follow next; if not, the parsing halts, and an error is given. The next symbol is "="; after it is reduced to <equals>, the parsing proceeds forward. The parser is expecting a user input string to follow next; if not, the parsing halts, and an error is given. The next symbol is "BMI"; this is reduced to <string>, and the parsing proceeds forward. Based on the production rules, parsing of OPTION is already complete. The parser reduces <reservedWord>, <equals>, and <string> to OPTION. Because there is only one OPTION in the OPTION_LIST, the parser reduces OPTION to OPTION_LIST and then reduces the whole sequence of <reservedWord> <var> <colon> EXPRESSION <slash> OPTION_LIST to PARAMETER_STMT. The next symbol is ";" and is reduced to <semicolon>. This is the end of the parsing process for statement 4 of Table 1 based on the production rules, and so the parser reduces PARAMETER_STMT and <semicolon> to $, and the parsing ends.

4.5. Algorithm

SUDAAN's expanded parser uses two main pushdown stacks, one stack named "Work" (for hosting all symbols for a specific statement), and one named "Polish" (for using a Shunting-Yard algorithm to produce Reverse Polish Notation) for symbols that resemble a complete mathematical expression within a statement.

For the level 2 parser, we have adapted the precedence parsing technique and constructed an operator-precedence table. The parser will either reduce or shift based on the rank of the precedence of the terminal symbol.

Work starts empty, and Polish starts with the initial state $. After processing each statement, the symbols will be stored in corresponding objects for later computation. The Work stack is initialized once for every statement. Polish may be initialized multiple times during the parsing of a complete statement. Table 2 describes the algorithm used by the level 2 parser to parse the expression.

Table 2. The level 2 parser algorithm

(1) Initialize Work and Polish.

(2) Determine the symbol X sent from the scanner with reduction rule X → x. If X is an ambiguous symbol, peek next symbol and check if it is a reserve word. If yes, reduce X to statement-close symbol and go to (7), else continue to (3).

(3) If Work is not empty, loop through Work, else go to (4).

  (3.1) Peek Work, get symbol Z with reduction rule Z → z (z is terminal).

  (3.2) If Precedence (z) is greater than or equal to Precedence (x), then Pop Work to symbol Z, push Z to Polish, apply reduction rules if Z → z is not the final deterministic result. If Z is nondeterministic, program halts, error is issued, else continue.

  (3.3) Validate mathematical expression based on production rules. If rules violated, program halts, error is issued, else continue.

  (3.4) Save parsed symbol to object list. Continue to (5).

(4) If X is not colon symbol, Push X to Work.

(5) Preserve current symbol X, push it to look-ahead set. Peek next symbol Y with reduction rule Y → y, compare Y to the look-ahead, if violate production rules, program halts, gives error, else continue.

(6) If X is the statement-close symbol and Work is empty, and Polish has only the starting symbol, then go to (1), else go to (2).

(7) End of statement process. If more statements exist, then go to (1), else go to (8).

(8) End of input process.

5. ALGORITHM EFFICIENCY

Section 4 described how the SUDAAN parser combines the original LALR(1) parser and the additional noncanonical parsing and other techniques. The original parser is used as a first level to process all fixed syntax with commonly used grammars and reduction rules. The first level parses the statements that do not contain user-defined expressions. In terms of algorithmic efficiency, the process is linear because it uses only linear grammars, and the process proceeds from left to right with no peeking and backtracking. The first-level parser has been thoroughly tested and still handles most of the SUDAAN statements, working correctly for those. It passes the remaining statements along to the level 2 parser.

The newly expanded level 2 parser handles all user-defined variables and expressions with a dynamically expandable look-ahead window. When the look-ahead window expands, the parser will not allow an infinite process and gives an error if no decision is made within a certain number of steps. Furthermore, the level 2 parser applies the operator-precedence grammars that allow linear-time parsing [1, pp. 548].

To demonstrate how the extended parser performances, let us execute the three programs that are listed in the Appendix. The data set that is used in the demonstration contains seven records and six variables.

- Program 1 contains very simple user-defined terms and no computational terms.

- Program 2 contains user-defined terms including computational terms, and its parsing is described in detail in Section 4 above.
- Program 3 shows an even more complex set of terms, including user-defined computational terms.

Table 3 shows the performance results in detail.

Table 3. SUDAAN program performance

Program	Real Time Used	CPU Time Used
1	0.03 seconds	0.01 seconds
2	0.04 seconds	0.03 seconds
3	0.05 seconds	0.03 seconds

Even though the data set used is very small, Table 3 shows that the parser takes minimum time to parse statements. Some statements are simple, like the one in program 1. Some are relatively complicated, like the last statement in program 3.

6. CONCLUSION

This paper has presented a solution that leverages several parsing techniques to form a combined two-level parser. The team used theoretically sound parsing methods to solve real-world problems in expanding the SUDAAN statistical software's capabilities. For any widely used software with a broad customer base, it is important to avoid significant change to the syntax to guarantee continued usability of the system, and the approach described here permitted compatibility with prior syntax.

The work described here allowed extension of the software's syntax to handle a vast range of user-defined expressions for statistical analysis. The resulting two-level parser is efficient and fast, and provides the necessary new capabilities. The decision to use operator-precedence approach was made with some hesitation because the use of operator-precedence parsing technique and Shunting-Yard algorithm is not very popular in practice, given the fact that the operator precedence parsing method has several disadvantages—including the problem that for some grammars the relations do not necessarily invoke the correct shift-reduce actions [10, p. 274]. Yet we have demonstrated that the techniques we selected are a good fit for our situation.

ACKNOWLEDGMENTS

I sincerely thank Ms. M. Rita Thissen and Dr. Albert D. Bethke, RTI International, for their assistance with this paper. Without their tremendous help, I would not have been able to finish the paper. I also recognize the important efforts of all the members of the SUDAAN team at RTI International.

REFERENCES

[1] Grune, D., Ceriel, J., & Jacobs, H. (2008) Parsing Techniques: A Practical Guide. 2nd ed. New York: Springer.
[2] Dijkstra, E.W. (1961) "Mathematics centrum," MTW, Vol. 2, pp54-56.
[3] RTI International (2012) SUDAAN 11 Language Manual, Volume 1. Research Triangle Park, NC: RTI International.

[4] Tai, K.C. (1979, October) "Noncanonical SLR(1) grammars," ACM Transactions on Programming Languages and Systems, Vol. 1, No. 2, pp295-320.
[5] Chiang, D. (2005) "A hierarchical phrase-based model for statistical machine translation," Proc. 43rd Annual Meeting of the ACL, pp263-270.
[6] Crescenzi, P, Gildea, D, Marino, A, & Rossi, G. (2015, November) "Synchronous context-free grammars and optimal linear parsing strategies," Journal of Computer and System Sciences, Vol. 81, No. 7, pp1333-1356.
[7] Younger, D. (1967) "Recognition and parsing of context-free languages in time O(n3)," Information and Computation, Vol. 10, No. 2, pp189-208.
[8] Firsov, D, & Uustalu, T. (2014, September-November) "Certified CYK parsing of context-free languages," Journal of Logical and Algebraic Methods in Programming, Vol. 83, Nos. 5-6, pp459-468.
[9] Schmitz, S. (2006) Noncanonical LALR(1) parsing. Developments in Language Theory (Lecture Notes in Computer Science, Vol. 4036, Ibarra, O.H., & Dang Z., eds. New York: Springer.
[10] Nijholt, A. (1988) Computers and Languages Theory and Practice. Amsterdam, Netherlands: Elsevier Science Publishers B.V..

APPENDIX

The three SUDAAN programs that are used in Section 5 to demonstrate the performance of the parser.

Program 1:

```
PROC VARGEN DATA = "mydata" DESIGN = WR;
XSUM      weight: myweight / NAME = "Weight Total";
Class Gender;
```

Program 2:

```
PROC VARGEN DATA = "mydata" DESIGN = WR;
XSUM      weight: myweight / NAME = "Weight Total";
XSUM      height: myheight / NAME = "Height Total";
PARAMETER bmi: weight/height / NAME = "BMI";
Class Gender;
```

Program 3:

```
PROC VARGEN DATA = "mydata" DESIGN = WR;
XSUM      weight: myweight / NAME = "Weight Total";
XSUM      height: myheight / NAME = "Height Total";
PARAMETER bmi: weight/height / NAME = "BMI";
PARAMETER bmi1: 0.3*(bmi**3) + (weight**4)/(height**6)/ NAME= "Compute BMI";
Class Gender;
```

AUTHOR

Donghui Wang was born in China in 1963. He received the B.S., M.S. degrees from Sichuan University, Chengdu, Sichuan, China in 1985 and Michigan State University, East Lansing, Michigan, USA in 1996, respectively. He joined RTI International, RTP, NC, USA, in 1998. Since 2004, he has been the lead developer of SUDAAN statistical software package.