# AN LALR(1) PARSER GRAMMAR FOR FORTRAN

by

Russ C. Rauhauser
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

#CU-CS-129-78 June, 1978

INTERIM TECHNICAL REPORT

U.S. ARMY RESEARCH OFFICE

CONTRACT NO. DAAG29-78-G-0046

Approved for public release;
Distribution Unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

# REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| Field | Entry |
|---|---|
| 1. REPORT NUMBER | CU-CS-129-78 |
| 2. GOVT ACCESSION NO. | |
| 3. RECIPIENT'S CATALOG NUMBER | |
| 4. TITLE (and Subtitle) | "An LALR(1) Parser Grammar for FORTRAN |
| 5. TYPE OF REPORT & PERIOD COVERED | |
| 6. PERFORMING ORG. REPORT NUMBER | |
| 7. AUTHOR(s) | Russ C. Rauhauser |
| 8. CONTRACT OR GRANT NUMBER(s) | DAAG29-78-G-0046<br>MCS77-02194 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS | Dept. of Computer Science<br>Univ. of Colorado at Boulder<br>Boulder, Colorado 80309 |
| 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS | |
| 11. CONTROLLING OFFICE NAME AND ADDRESS | U. S. Army Research Office<br>Post Office Box 12211<br>Research Triangle Park, NC 27709 |
| 12. REPORT DATE | June 1978 |
| 13. NUMBER OF PAGES | 55 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | |
| 15. SECURITY CLASS. (of this report) | Unclassified |
| 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE | NA |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

FORTRAN grammar, parser grammar, LALR(1), structure tree

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)


An automatic parser generator is a tool for quickly implementing programming language parsers. Parser generators based upon LR parsing have been built for grammars satisfying the LR(0), SLR(1), and LALR(1) properties. Speed of the resulting parser is comparable to that of a hand coded recursive descent parser.

DAVE, an automatic program testing aid, requires a flexible, easy-to-implement parser. This report presents an LALR(1) grammar for ANSI standard FORTRAN,


DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## CONTENTS

| | | Page |
|---|---|---|
| I. | INTRODUCTION | |
| II. | THE GRAMMAR | |
| | 1. Meta-language review | 3 |
| | 2. Grammar listing | 15 |
| III. | DISCUSSION | |
| | 1. Information Sources | 25 |
| | 2. Completeness | 25 |
| | 3. Scanner Interface | 26 |
| | 4. Right-recursion | 28 |
| APPENDICES | | |
| | A - Structure Tree Diagrams | 31 |
| | B - Expression Grammar Transformations | 45 |
| BIBLIOGRAPHY | | 55 |

FIGURES


| FIGURE | | Page |
|---|---|---|
| 1. | Meta-language keywords and special symbols . . . . . . . . | 4 |
| 2. | Tokens representing FORTRAN keywords and special characters . . . . . . . . . . . . . . . . . . . . | 5 |
| 3. | Tokens requiring subrosa information . . . . . . . . . . . | 7 |
| 4. | Structure tree representing A = B + C . . . . . . . . . . . | 10 |
| 5. | Tree node stack during parse of A = B + C . . . . . . . . . | 12 |
| 6. | Parse stack during parse of the FORTRAN statement EXTERNAL A, B, C . . . . . . . . . . . . . . . . . . . . | 29 |
| B1. | Simple grammar for FORTRAN expressions . . . . . . . . . | 46 |
| B2. | Grammar with logical and arithmetic expressions combined . . . . . . . . . . . . . . . . . . . . . . . . | 48 |
| B3. | Expression grammar after four steps of back substitution . . . . . . . . . . . . . . . . . . . . . . | 50 |
| B4. | Grammar after many steps of back substitution and simplification . . . . . . . . . . . . . . . . . . . | 52 |
| B5. | Expression grammar after back substitution in the arithmetic expression sub-grammar . . . . . . . . . | 53 |
| B6. | Final LALR(1) grammar for FORTRAN expressions . . . . . . | 54 |

## Abstract

An automatic parser generator is a tool for quickly implementing programming language parsers. Parser generators based upon LR parsing have been built for grammars satisfying the LR(0), SLR(1), and LALR(1) properties. Speed of the resulting parser is comparable to that of a hand coded recursive descent parser.

DAVE, an automatic program testing aid, requires a flexible, easy-to-implement parser. This report presents an LALR(1) grammar for ANSI standard FORTRAN, suitable as input to an automatic parser generator. Its use in building DAVE provides a measure of the desired flexibility, since new parsers for FORTRAN dialects may be produced by simply modifying the existing grammar.

A powerful meta-language is used to describe the grammar. Its features are summarized, including the method for specifying automatic construction of (intermediate-text) structure trees during parsing. The report concludes with a discussion of some of the more important decisions made during development of the grammar.

# I. Introduction

Context-free grammars are widely recognized as appropriate tools for describing the syntax of programming languages. Their formality has allowed the language designer to communicate precisely and unambiguously his intended structure, and more recently has allowed the language implementor to automatically generate the parsing phase of his compiler.

FORTRAN was developed before the usefulness of grammars was fully appreciated. Its standard document [1] uses English prose to communicate syntactic structure. Since FORTRAN has already been widely implemented, a FORTRAN grammar might appear to be of little practical use today.

Recent interest in the development of software validation tools, however, has kept the market for efficient, easy-to-generate FORTRAN parsers very much alive. The DAVE project [2], currently under improvement at the University of Colorado, is an example.

DAVE is an automatic program testing aid which performs a static analysis of programs written in ANSI standard FORTRAN. Experience with DAVE has uncovered a sizeable demand for diagnostic aids capable of analyzing FORTRAN "dialects" as well. One solution is to provide a flexible tool which may be easily converted to any of the FORTRAN variants. The use of an automatic parser generator provides a step toward the desired flexibility, since new parsers may be produced by simply modifying a basic grammar.

The purpose of this report is to present a FORTRAN grammar which:

1) captures the structure of ANSI standard FORTRAN at the parsing level, and
2) satisfies the LALR(1) property, a condition required by the BOBSW parser generator system.

Details of the exact grammar requirements of the BOBSW system, and its use in producing a parser for the DAVE project, may be found in [3].

It is assumed that the reader is familiar with grammars, their relation to programming languages, and the parsing process. A good elementary treatment may be found in Gries [4]. Hopcroft and Ullman [5] provide a more theoretical approach to grammars and their properties. The operation of a parser generator based upon the LALR(1) property is described in LaLonde [6].

Section II contains a listing of the FORTRAN parser grammar and a review of the meta-language used to describe it. Section III concludes with a discussion of some important decisions made during development of the grammar.

## II. The Grammar

Keywords and special symbols belonging to the meta-language used in this report are shown in Figure 1. Although keywords will appear underlined in the grammar listing to follow, they are actually reserved and may not be used as nonterminal symbols.

The meta-language has been designed to accept nested "sub-grammar" definitions. To facilitate machine checks on proper nesting, each grammar is delimited by a pair of keywords:

```
parser Fortran_compilation_unit:
   :
end    Fortran_compilation_unit
```

The name following parser identifies the goal symbol of the grammar, and must exactly match the name following end. A terminating colon on the parser line allows the grammar writer to optionally omit the preceeding name (it may never be omitted from the end line). In this case, the first production of the grammar serves to identify its goal symbol.

Productions are unordered and written in free form syntax. A sharp (#), which may appear anywhere in the grammar, indicates that the remaining portion of the current line is to be treated as a comment.

Non-terminal symbols are written as a sequence of one or more alphabetic, numeric, or underbar characters, beginning with an alphabetic. Terminal symbols are delimited by single quotes, and may consist of any sequence of printable characters except the single quote and blank.

The terminal symbols of a parser grammar correspond to tokens received from a scanner module at parse time. Two kinds of tokens can be identified. The first kind may be described as representing entities having a unique form in the source language. A complete list of such tokens for FORTRAN, representing keywords and special characters, is given in Figure 2. A careful inspection of this list will reveal the absence of several FORTRAN keywords and the addition of several new ones. A discussion of issues relating to these anomalies is deferred to Section III.

| | |
|---|---|
| parser | : |
| end | # |
| list | ' (single quote) |
| rlist | < |
| | > |
| | -> |
| | ; |
| | | |
| | ( |
| | ) |
| | + |
| | * |
| | => |

Figure 1.

Keywords and special symbols belonging to the meta-language used to describe the grammar of this report.

| | |
|---|---|
| SUBROUTINE | DATA |
| FUNCTION | COMMON |
| EXTERNAL | DO |
| DIMENSION | ASSIGN |
| EQUIVALENCE | TO |
| LOGICAL | GOTO |
| INTEGER | CALL |
| DOUBLEPRECISION | STOP |
| COMPLEX | PAUSE |
| REAL | READ |
| OR | WRITE |
| AND | REWIND |
| NOT | BACKSPACE |
| BLOCKDATA | ENDFILE |
| RETURN | FORMAT |
| CONTINUE | EOS |
| END | LOGIF |
| | ARITHIF |

(
)
=
,
/
-
+
*
**

Figure 2

Complete list of terminal symbols (tokens) which represent FORTRAN keywords and special characters.

The second kind of token represents entities which do not have a unique representation in the source language. For example, a given program may contain many different integer constants. When the scanner module returns a token of type "integer constant", it must also include some subrosa information indicating the exact integer chosen by the programmer. Figure 3 gives a complete list of all such tokens as they will appear in the FORTRAN grammar of this report. Sample subrosa information is shown for each token. Notice that angle brackets are used to distinguish tokens requiring subrosa information from the simple tokens of Figure 2.

The following example illustrates how a production may be written in its most basic form.

```
Slash -> '/';
```

The production separator symbol (->) is preceeded by a single nonterminal and followed by a sequence of zero or more terminals, nonterminals, and meta-symbols. A semicolon terminates the production.

Very often a nonterminal symbol will appear as the left hand side of more than one production. There are two (equivalent) ways of conveniently grouping such productions together:

```
Field
-> Basic_field
-> Group1;
```

and

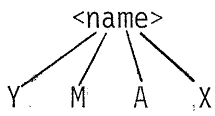
```
Field
-> Basic_field|Group1;
```

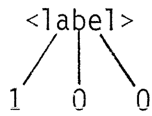
The vertical bar indicates alternation, and may also appear with parentheses to effect a kind of "distributive" property. For example,

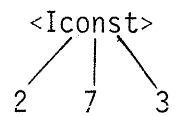
```
Program_unit
-> (Subprogram|Program_body) 'END';
```

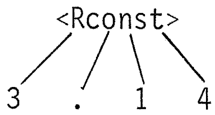
is equivalent to

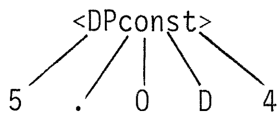
```
Program_unit
-> Subprogram 'END'
-> Program_body 'END';
```

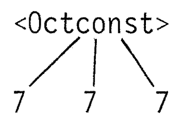


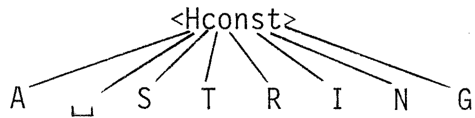



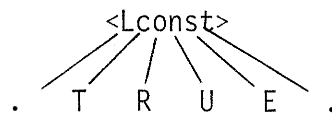



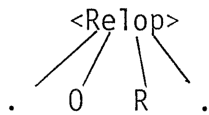



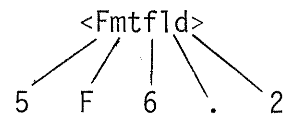




Figure 3

Complete list of terminal
symbols (tokens) which require
associated subrosa information.

Two meta-symbols have been included to more conveniently describe the concept of repetition. A trailing plus character (+) indicates "one or more" of the entity which preceeds it. For example,

```
Sep -> Slash +;
```

is written to express the fact that a separator may consist of one or more slashes. This same concept may be described without the plus, but requires one additional production:

```
Sep
-> Slash
-> Sep Slash;
```

Similarly, a trailing asterisk (*) indicates "zero or more" of the entity which preceeds it.

Keywords list and rlist have been included to more conveniently capture the syntax of ordinary lists of objects. For example, the production

```
Ext
-> 'EXTERNAL' '<name>' list ',' ;
```

is written to indicate that a FORTRAN external statement must contain the word EXTERNAL followed by a list of one or more names separated by commas. Expressing the list concept without a special keyword requires an additional nonterminal and two more productions:

```
Ext
-> 'EXTERNAL' Name_list;
Name_list
-> '<name>'
-> Name_list ',' '<name>';
```

The keyword rlist is distinguished from list only by the fact that its elimination results in a right recursive expansion instead of a left recursive one:

```
Ext
-> 'EXTERNAL' '<name>' rlist ',' ;
```

expands to

```
Ext
-> 'EXTERNAL' Name_list;
Name_list
-> '<name>'
-> '<name>' ',' Name_list;
```

Right recursion is sometimes necessary to achieve the LALR(1) property, as will be demonstrated in section III.

The basic activity of a programming language parser is to discover the structure of an input program and to verify that the syntactic rules of the language have not been violated. In many cases, the parser also transforms source code into a suitable intermediate form so that later processing is made easier. One possibility is conversion to a structure tree, where relationships among the syntactic units of a program are represented in tree form. For example, the FORTRAN assignment statement

A=B+C

could be represented in intermediate form by the structure tree shown in Figure 4.

The meta-language used here contains mechanisms which allow the grammar writer to specify tree building activities. A brief summary of LR parsing is given to help explain how tree construction may be combined with the parsing process.

LR parsing may be viewed as a sequence of read and reduce actions. During a read action, the parser requests the next input token from the scanner module and, depending upon its current state and the token received, moves to a next state. The new token is pushed onto a parse stack, where a summary of the "already seen" portion of source text is maintained.

Reduce actions become possible whenever the top symbol(s) of the parse stack exactly match the symbol(s) on the right hand side of a grammar production. (For LR(1) parsing, the appropriateness of a reduction may be determined by looking no more than one token ahead in the input stream.) During a reduce action, the matching symbols are removed from the stack and replaced by the single nonterminal which appears on the left hand side of

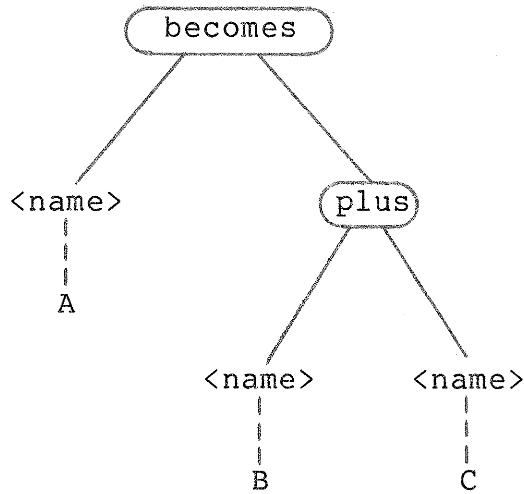



Figure 4

Structure tree corresponding to the FORTRAN assignment statement A=B+C. Note that the variable names are actually sub-rosa information attached to <name> leaves, and are not considered nodes of the tree.

that production, and a new state is entered. The objective is to continue with read and reduce actions until there are no more input tokens to read and only the goal symbol of the grammar remains on the parse stack.

Tree construction is carried out during the reduce actions of parsing. An additional stack, called the tree node stack, is added to facilitate the linking of nodes into a tree. The exact process is best described by an example.

Suppose the grammar writer would like to specify the construction of structure trees for assignment statements. For example, he would like a parse of A=B+C to result in the tree of Figure 4. Assume for now that just two grammar productions are needed to describe assignment statements:

```
Basic_stmt -> '<name>' '=' Expression;
Expression -> '<name>' '+' '<name>';
```

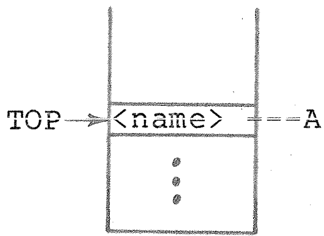
His major task will be to imagine how such statements will be parsed, and to identify the order of the various reduce actions that will take place. To illustrate, a parse of A=B+C is described.

First, the parser receives a <name> token from the scanner, with "A" included as subrosa information. Since this activity is a read action, the <name> token is pushed onto the parse stack. Whenever the parser's tree-building option is turned ON, receipt of a terminal symbol delimited by angle brackets will also result in the creation of a corresponding tree node. This new node is then pushed onto the tree node stack as shown in Figure 5(a).

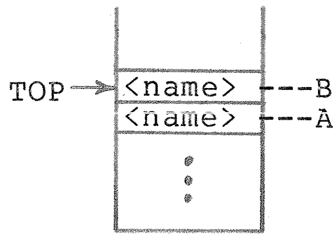
Next, the scanner supplies a token representing the FORTRAN equals sign. Although this token participates in parse stack activities, it does not result in creation of a new tree node since surrounding angle brackets are not present in the production for Basic_stmt.

Receipt of the next <name> token, corresponding to variable B, results in actions identical to those for variable A. The modified tree node stack is shown in Figure 5(b). Note that the parse is now "following" the production for Expression.

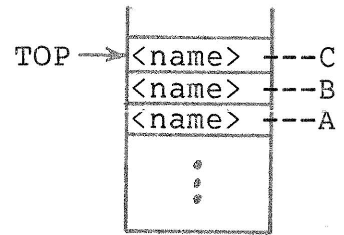
The next token, representing a FORTRAN plus symbol, results only in parse stack activities (why?). Finally, a <name> token corresponding to variable C is read, resulting in the tree node stack of Figure 5(c).



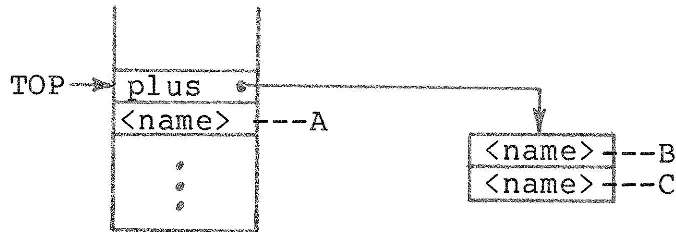

(a)
After receipt of
token representing
variable A



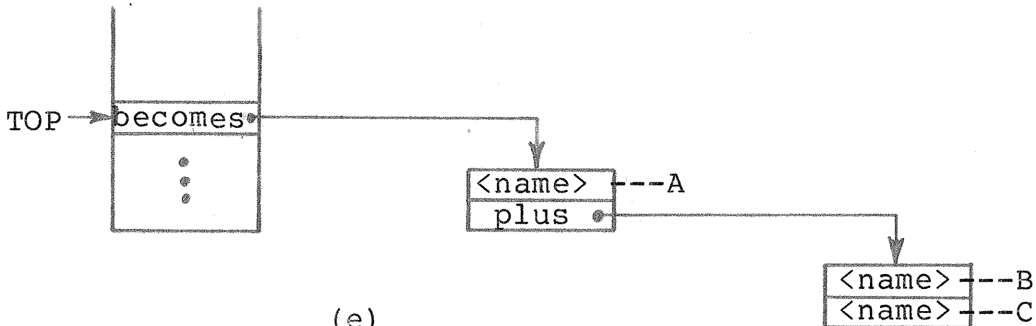

(b)
After receipt of
token representing
variable B




(c)
After receipt of
token representing
variable C




(d)
After reduction of
Expression ⟶ '<name>' '+' '<name>' ⟹ 'plus';




(e)
After reduction of
Basic_stmt ⟶ '<name>' '=' Expression ⟹ 'becomes';
(Compare with the tree of Figure 4)

<u>Figure 5</u>
Modifications of tree node stack
during parse of the FORTRAN statement
A = B + C

It now happens that the top three symbols of the parse stack exactly match the symbols on the right hand side of the production for Expression. A reduce action involving this production is therefore indicated, and carried out. The grammar writer may specify that tree building activities should also be performed at this time by augmenting his grammar production with a double right arrow meta-symbol (=>).

For example, if he writes

```
Expression
-> '<name>' '+' '<name>' => 'plus';
```

then during any reduce action involving that production, a new tree node is created and labeled "plus". This node is then automatically linked into the existing tree structure by means of the following actions:

1) The two <name> nodes at the top of the tree node stack, corresponding to the two <name> nodes on the right hand side of the production for Expression, are linked as sons of the new "plus" node.
2) The sons are then popped from the tree node stack and replaced by their parent.

The result of these actions for the current example is shown in Figure 5(d).

The parsing process continues with a reduce action involving the production for Basic_stmt. The fact that more tree building is desired may be indicated by writing

```
Basic_stmt
-> '<name>' '=' Expression => 'becomes';
```

In this case, a new "becomes" node is created and linked into the existing tree structure as shown in Figure 5(e). Notice that the resulting tree is identical to the one shown in Figure 4, and has been created "bottom up".

During linking of the "becomes" node, the Expression nonterminal in the production for Basic_stmt corresponds to a single node on the tree node stack (specifically, the "plus" node). It should be noted that nonterminals which preceed a + or * repetition meta-symbol may correspond

to more than one stack entry. An automatic counting mechanism is provided to handle these cases.

This completes the review of meta-language features. A listing of the FORTRAN parser grammar follows. Although construction of an intermediate structure tree has been completely specified by means of the double right arrow, the reader may wish to consult Appendix A for a more graphic description of tree shape.

```
parser  Fortran_compilation_unit:

#Overall program structure:

Fortran_compilation_unit
->  Program_unit +                                          =>  'compile';

Program_unit
->  '<label>' Subprogram 'END'                              =>  'labeled'
->  (Subprogram|Program_body) 'END'
->  '<label>' 'BLOCKDATA' 'EOS' Blockdata_stmts             =>  'labeled'
->  'BLOCKDATA' 'EOS' Blockdata_stmts;

Blockdata_stmts
->  Specification* Data_stmt* 'END'                         =>  'blockdata';

Subprogram
->  'SUBROUTINE' '<name>' Subrtn_parameters 'EOS'
    Program_body                                            =>  'subroutine'
->  Rtrn_type 'FUNCTION' '<name>' Parameter_list 'EOS'
    Program_body                                            =>  'function';

Subrtn_parameters
->  Parameter_list
->                                                          =>  'parameters';

Parameter_list
->  '('  ('<name>' list ',')  ')'                           =>  'parameters';

Rtrn_type
->  Type
->                                                          =>  'default';

Type
->  'INTEGER'                                               =>  'integer'
->  'REAL'                                                  =>  'real'
->  'DOUBLEPRECISION'                                       =>  'doubleprecision'
->  'COMPLEX'                                               =>  'complex'
->  'LOGICAL'                                               =>  'logical';

Program_body
->  Body_group1* Body_group2   Body_group3*                 =>  'body';
```

```
Body_group1
->   Specification
->   External_stmt
->   Format_stmt;

Body_group2
->   Executable_stmt
->   Function_or_array;

Body_group3
->   Executable_stmt
->   Function_or_array
->   Format_stmt
->   Data_stmt;
```

```
# FORTRAN declarations:

Specification
->  Spec 'EOS'                  -> '<label>' Spec 'EOS'  =>  'labeled';

Spec
->  'DIMENSION' (Array_dcln list ',')                    =>  'dimension'
->  'COMMON' Com_block1   Com_block_rest*                =>  'common'
->  'EQUIVALENCE' (Equiv_list list ',')                  =>  'equivalence'
->  Type (Dcln_element list ',')                         =>  'declaration';

Array_dcln
->  '<name>' '(' Subscr_list ')' Type_placeholder        =>  'array';

Subscr_list
->  Integer                                              =>  ','
->  Integer ',' Integer                                  =>  ','
->  Integer ',' Integer ',' Integer                      =>  ',';

Integer
->  '<Iconst>'                  -> '<name>'; # Integer variable

Type_placeholder
->                                                       =>  'default';

Com_block1
->  Com_name1  Dcln_list                                 =>  'block';

Com_name1
->  '/' '<name>' '/'
->  ('/' '/' | )                                         =>  'blank';

Com_block_rest
->  Com_name_rest   Dcln_list                            =>  'block';

Com_name_rest
->  '/' '<name>' '/'
->  '/' '/'                                              =>  'blank';

Dcln_list
->  Common_dcln_element  list ','                        =>  ',';

Common_dcln_element
->  ('<name>'|Common-array);
```

```
Common_array
->  '<name>' '(' Iconst_list ')' Type_placeholder          =>  'array';

Equiv_list
->  '(' Declarator ',' (Declarator list ',') ')'           =>  'share';

Declarator
->  '<name>'
->  '<name>' '(' Iconst_list ')'                           =>  'element';

Iconst_list
->  '<Iconst>'                                             =>  ','
->  '<Iconst>' ',' '<Iconst>'                              =>  ','
->  '<Iconst>' ',' '<Iconst>' ',' '<Iconst>'               =>  ',';

Dcln_element
->  ('<name>'|Array_dcln);

External_stmt
->  Ext 'EOS'                 -> '<label>' Ext 'EOS'       =>  'labeled';

Ext
->  'EXTERNAL'     ('<name>' list ',')                     =>  'external';

Data_stmt
->  Data 'EOS'                -> '<label>' Data 'EOS'      =>  'labeled';

Data
->  'DATA'     (Data_pair list ',')                        =>  'data';

Data_pair
->  Declarator_list '/' Data_list '/'                      =>  'pair';

Declarator_list
->  Declarator list ','                                    =>  'declarators';

Data_list
->  Data_item list ','                                     =>  'dataitems';

Data_item
->  ('<Hconst>'|'<Lconst>'|Data_number)
->  '<Iconst>' '*' ('<Hconst>'|'<Lconst>'|Data_number)     =>  '*';
```

```
Data_number
->   Complex_const
->   Number
->   '+' Number                  -> '-' Number             =>   'neg';

Complex_const
->   '(' Cconst_element ',' Cconst_element ')'             =>   'cconst';

Cconst_element
->   '<Rconst>'
->   '+' '<Rconst>'              ->'-' '<Rconst>'          =>   'neg';

Number
->   '<Iconst>'   -> '<Rconst>'   -> '<DPconst>';
```

```
# FORTRAN format statements:

Format_stmt
->  '<label>' Fmt 'EOS'                                        =>  'labeled';

Fmt
->  'FORMAT' '(' Slash* ')'                                    =>  'format'
->  'FORMAT' '(' Slash* (Field list Sep) Slash* ')'            =>  'format';

Slash
->  '/';

Sep
->  ','                        -> Slash +;

Field
->  Basic_field                -> Group1

Basic_field
->  '<Hconst>'                 -> '<Fmtfld>';

Group1
->  Repeat_count  Fmt1                                         =>  'group';

Fmt1
->  '(' Slash* ')'                                             =>  'format';
->  '(' Slash* (Field1  list Sep) Slash* ')'                   =>  'format';

Field1
->  Basic_field                -> Group2

Group2
->  Repeat_count Fmt2                                          =>  'group';

Fmt2
->  '(' Slash* ')'                                             =>  'format'
->  '(' Slash* (Basic_field list Sep) Slash* ')'               =>  'format';

Repeat_count
->  '<Iconst>'                 ->                              =>  'one';
```

```
# FORTRAN function_or_array statements:

Function_or_array
->  Foa 'EOS'                  -> '<label>' Foa 'EOS'    =>  'labeled';

Foa
->  '<name>' '(' Exprn_list ')' '=' Expression           =>  'foa';

# FORTRAN executable statements:

Executable_stmt
->  Exec 'EOS'                 -> '<label>' Exec 'EOS'   =>  'labeled';

Exec
->  'DO' '<label>' '<name>' '=' Do_parameters            =>  'do'
->  'LOGIF' '(' Logical_exprn ')' Basic_stmt             =>  'logif'
->  'LOGIF' '(' Paren_name ')' Basic_stmt                =>  'logif'
->  Basic-stmt;

Do_parameters
->  Integer ',' Integer                                  =>  ','
->  Integer ',' Integer ',' Integer                      =>  ',';

Basic_stmt
->  '<name>' '=' Expression                              =>  'becomes'
->  'ASSIGN' '<label>' 'TO' '<name>'                     =>  'assign'
->  'GOTO' '<label>'                                     =>  'goto'
->  'GOTO' '(' Label_list ')' ',' '<name>'               =>  'compgo'
->  'GOTO' '<name>' ',' '(' Label_list ')'               =>  'assigngo'
->  'ARITHIF' '(' Arith_exprn ')' '<label>' ','
    '<label>' ',' '<label>'                              =>  'arithif'
->  'CALL' '<name>' Call_args                            =>  'call'
->  'RETURN'                                             =>  'return'
->  'CONTINUE'                                           =>  'continue'
->  'STOP'                                               =>  'stop'
->  'STOP' '<Octconst>'                                  =>  'stop'
->  'PAUSE'                                              =>  'pause'
->  'PAUSE' '<Octconst>'                                 =>  'pause'
->  'REWIND' Integer                                     =>  'rewind'
```

```
->  'BACKSPACE' Integer                                  =>  'backspace'
->  'ENDFILE' Integer                                    =>  'endfile'
->  'READ' '(' Integer Frmt ')' Possible_IO_list         =>  'read'
->  'WRITE' '(' Integer ',' Form ')' Possible_IO_list    =>  'write'
->  'WRITE' '(' Integer Form_placeholder ')' IO_list     =>  'write';

Label_list
->  '<label>' list ','                                   =>  ',';

Call_args
->                                                       =>  ','
->  '(' ('<Hconst>'|Expression) list ',' ')'             =>  ',';

Frmt
->  Form_placeholder
->  ',' Form;

Form
->  ('<label>'|'<name>')                                 =>  'fmt';

Form_placeholder
->                                                       =>  'fmt';

Possible_IO_list
->                                                       =>  'iolist'
->  IO_list;

IO_list
->  (Named_value|'(' Named_value rlist ',' ')'
    |'(' Iteration_list ')' ) rlist ','                  =>  'iolist';

Iteration_list
->  (Named_value|'(' Named_value rlist ',' ')'
    |'(' Iteration_list ')' ) ',' Do-specification       =>  'iterate'
->  (Named_value|'(' Named_value rlist ',' ')'
    |'(' Iteration_list ')' ) ',' Iteration_list         =>  'iterate';

Do_specification
->  '<name>' '=' Do_parameters                           =>  'do_spec';
```

```
# FORTRAN expressions:

Expression
->  (Logical_exprn|Arith_exprn);

Logical_exprn
->  L_term
->  Logical_exprn 'OR' (L_term|Paren_name)                    =>  'or'
->  Paren_name 'OR' (L_term|Paren_name)                       =>  'or';

L_term
->  L_factor
->  L_term 'AND' (L_factor|Paren_name)                        =>  'and'
->  Paren_name 'AND' (L_factor|Paren_name)                    =>  'and';

L_factor
->  L_primary
->  'NOT' (L_primary|Paren_name)                              =>  'not';

L_primary
->  '<Lconst>'
->  Relational_exprn
->  '(' Logical_exprn  ')';

Relational_exprn
->  Arith_exprn '<Relop>' Arith_exprn                         =>  'relop';

Arith_exprn
->  Paren_name
->  Simple_AE;

Simple_AE
->  A_term
->  Simple_AE '+' (A_term|Paren_name)                         =>  'plus'
->  Simple_AE '-' (A_term|Paren_name)                         =>  'minus'
->  Paren_name '+' (A_term|Paren_name)                        =>  'plus'
->  Paren_name '-' (A_term|Paren_name)                        =>  'minus'
->  '+' (A_term|Paren_name)
->  '-' (A_term|Paren_name)                                   =>  'neg';
```

```
A_term
->  A_factor
->  A_term '*' (A_factor|Paren_name)                        =>  'mult'
->  A_term '/' (A_factor|Paren_name)                        =>  'div'
->  Paren_name '*' (A_factor|Paren_name)                    =>  'mult'
->  Paren_name '/' (A_factor|Paren_name)                    =>  'div';

A_factor
->  A_primary
->  A_primary '**' (A_primary|Paren_name)                   =>  'pwr'
->  Paren_name '**' (A_primary|Paren_name)                  =>  'pwr';

A_primary
->  Number
->  Complex_const
->  '(' Simple_AE ')';

Paren_name
->  Named_value
->  '(' Paren_name ')'                                      =>  'parens';

Named_value
->  '<name>'
->  '<name>' '(' Exprn_list ')'                             =>  'apply';

Exprn_list
->  Expression list ','                                     =>  ',';

end Fortran_compilation_unit
```

## III. Discussion

The FORTRAN parser grammar was derived in two steps. First, a straightforward grammar was written to capture ANSI standard FORTRAN, without regard to the LALR(1) property. The resulting grammar was then modified to attain LALR(1). A discussion of issues relating to these steps is given in the four sub-sections below.

### Information sources

The document entitled "USA Standard FORTRAN, X3.9 - 1966" [1] served as the basic reference for syntactic structure. Syntax charts developed by McIlroy [8] were later used to verify that the initial grammar was a "correct" interpretation of the standard.

### Completeness

Some aspects of FORTRAN syntax are not easily specified in a parser grammar. The following syntax rules must be processed after the parsing phase (references to the standard are shown in parentheses).

1) The integer constant zero may not appear
   a) as a declarator subscript in an array declaration (7.2.1.1), or
   b) as a data item replication factor in a DATA initialization statement (7.2.2), or
   c) as a parameter in a DO statement (7.1.2.8).

2) A statement label must be greater than zero (3.4).

3) Statement function definitions must precede the first executable statement of the given program unit (9.1.1) (Some statement function definitions cannot be syntactically distinguished from assignment statements in which an array element appears to the left of the equals sign).

4) The dummy arguments of a statement function definition must be distinct variable names (8.1.1).

5) The expression appearing to the right of the equals sign in a statement function definition may only contain
   a) Non-Hollerith constants
   b) Variable references
   c) Intrinsic function references

d) References to previously defined statement functions

e) External function references

Note that array element references are excluded. (8.1.1)

6) A RETURN statement may not appear in the main program (7.1.2.5).

7) Since arrays must be defined with 1, 2, or 3 dimensions (7.2.1.1), array elements must be specified with no more than 3 subscripts (5.1.3.2).

8) Array element subscripts must be written as one of the following constructs

```
c * v + k
c * v - k
c * v
v + k
v - k
  v
  k
```

where c and k are integer constants and v is an integer variable reference (5.1.3.3).

9) The number of subscripts of an array element in an EQUIVALENCE statement must correspond to the dimensionality of the array declarator or must be one (7.2.1.4).

Scanner Interface

A scanner interface may be specified by listing all of the token types to be passed from scanner to parser, together with conventions regarding the transmission of subrosa information. Figures 2 and 3 list the FORTRAN token types used in the parser grammar of this report. Although the choice of token types for FORTRAN is generally straightforward, several decisions were guided by more subtle considerations and are worthy of special mention.

The end-of-statement (EOS) token is made necessary by the fact that READ and WRITE statements need not contain input/output lists. For example, both WRITE(6,1000) and WRITE(6,1000)YMAX are legal statements according to the rules of ANSI standard FORTRAN. Suppose that EOS tokens were not supplied by the scanner module, and a FORTRAN program

contained the following statement sequence:

```
        ⋮
WRITE(6,1000)
YMAX = 1
        ⋮
```

A problem occurs when parsing reaches the end of the WRITE statement: it is impossible for a parser with only single character look-ahead to tell whether the next token, YMAX, is part of the WRITE statement or part of the following assignment statement. The inclusion of an intervening EOS token (generated by the scanner) resolves this ambiguity.

Early versions of the FORTRAN grammar expressed IF statement syntax by means of the following productions:

```
Exec → 'IF' '(' Logical_exprn ')' Basic_stmt;
Basic_stmt → 'IF' '(' Arith_exprn ')' '<label>' ','
             '<label>' ',' '<label>';
```

Unfortunately, these productions are not LALR(1). The problem occurs when an IF statement of the form IF(A)··· is encountered. The parser cannot decide (with just single character look-ahead) whether to reduce the named value, A, to a logical expression or an arithmetic expression.

Discovery of this problem led to the realization that ambiguities involving named value appear in other contexts as well. Appendix B gives a complete account of the problem, and details the extensive set of expression grammar transformations necessary to solve it.

The results of Appendix B add one more production to the description of IF statement, but do not solve the original problem:

```
Exec → 'IF' '(' Logical_exprn ')' Basic_stmt
     → 'IF' '(' Paren_name ')' Basic_stmt;
Basic_stmt → 'IF' '(' Arith_exprn ')' '<label>' ','
             '<label>' ',' '<label>';
```

Now when the parser encounters a Paren_name (i.e., a named value surrounded by zero or more sets of parentheses), it cannot decide whether to continue reading or to reduce that Paren_name to Arith_exprn. Intuitively, the reason is that the parser does not know which kind of IF statement

is being parsed until after the reduce decision has been made.

The problem is solved by providing two token types for the IF keyword, one for logical if statements (LOGIF) and one for arithmetic if statements (ARITHIF):

```
Exec → 'LOGIF' '(' Logical_exprn ')' Basic_stmt
     → 'LOGIF' '(' Paren_name ')' Basic_stmt;
Basic_stmt → 'ARITHIF' '(' Arith_exprn ')' '<label>' ','
             '<label>' ',' '<label>';
```

Right Recursion

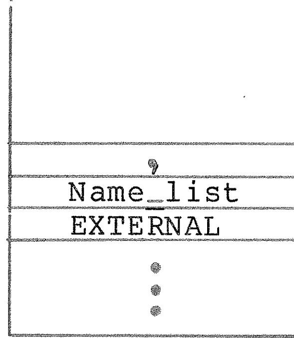
Recall from section II that both list and rlist may be used to express the syntax of ordinary lists of objects. Their only distinguishing feature is that list results in a left recursive expansion, while rlist results in a right recursive one. Although Figure 6 clearly demonstrates that left recursion is preferred in LR parsing because it results in a smaller parse stack, right recursion is sometimes necessary to achieve the LALR(1) property.

The FORTRAN standard describes the syntax of input/output lists as follows:

> "A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by a comma. Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma. A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses."

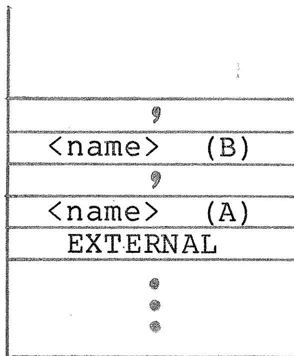
This complex (and confusing!) structure may be expressed by the following grammar productions, where the nonterminal Named_value stands for variable name, array element name, and array name, and Iteration_list may be considered a synonym for DO-implied list:

```
IO_list
→ (Named_value | '(' Named_value rlist ',' ')'
  | '(' Iteration_list ')' ) rlist ',' ;
Iteration_list
→ (Named_value | '(' Named_value rlist ',' ')'
```




(a)

With left recursive
grammar production
Ext ——> 'EXTERNAL' '<name>' list ',';




(b)

With right recursive
grammar production
Ext ——> 'EXTERNAL' '<name>' rlist ',';

Figure 6

Parse stack just before the name
C is read during parse of the FORTRAN statement
EXTERNAL A,B,C

```
    | '(' Iteration_list ')' ) ',' Do_specification
  → (Named_value | '(' Named_value rlist ',' ')'
    | '(' Iteration_list ')' ) ',' Iteration_list;
```

To see that right recursion is necessary, consider a parse of the statement

```
WRITE(6,1000) (A(I),I,I=1,5)
```

Receipt of the opening parenthesis of the I/O list indicates to the parser that either a named value list or an iteration list follows. If left recursion has been used to specify named value lists, then a read/reduce conflict occurs after receipt of the next token, representing array element A(I). The parser cannot decide whether to immediately reduce this token to Named_value_list, or to continue reading because an iteration list is involved. The basic problem, then, is that the parser cannot distinguish between named value lists and iteration lists until either a closing parenthesis is read (indicating the former), or the receipt of a FORTRAN equals sign indicates that a Do-specification is being parsed. The use of right recursion (rlist) in specifying named value lists solves the problem by delaying all reductions until the entire I/O list has been seen.

# APPENDIX A

## STRUCTURE TREE DIAGRAMS

# DIAGRAM CONVENTIONS

## NODES

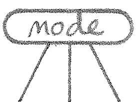
'mode' -- node having no sons.

(mode) -- node which may have sons. If no sons are shown, look elsewhere on the same page for a definition.

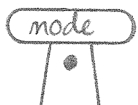
<mode> -- leaf node which has sub-rosa information.

NODE -- class of nodes. Look on page with that heading for definitions.

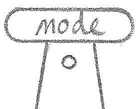
## NUMBER OF SONS



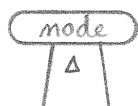
-- exact number of sons as shown.



-- one or more sons of the specified type(s).



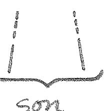
-- zero or more sons of the specified type(s).



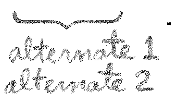
-- 1, 2, or 3 sons of the specified type(s).

## SPECIFICATION OF SONS

-- only possible choice for the son shown.

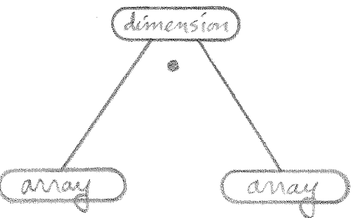


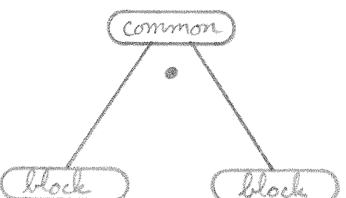
-- only possible choice for the group of sons shown.

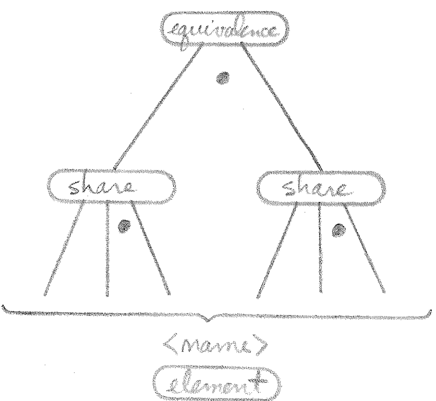


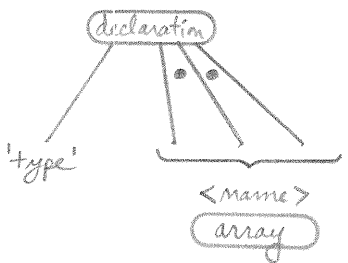
-- alternate choices for either a single son or the group of sons shown.

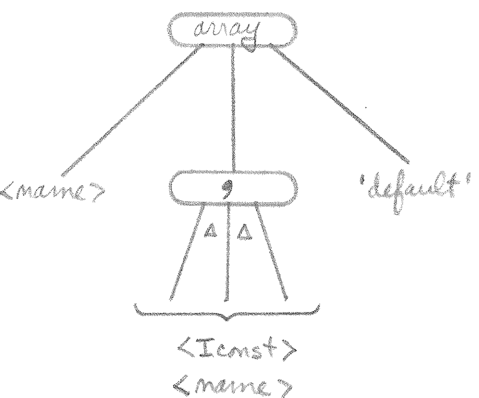
# SPECIFICATION

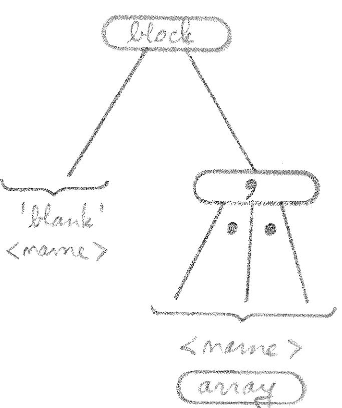


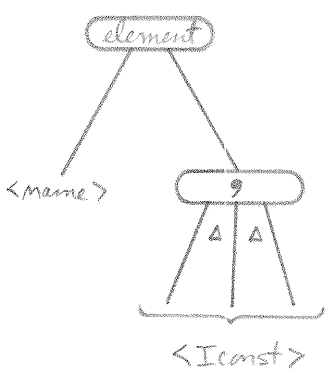



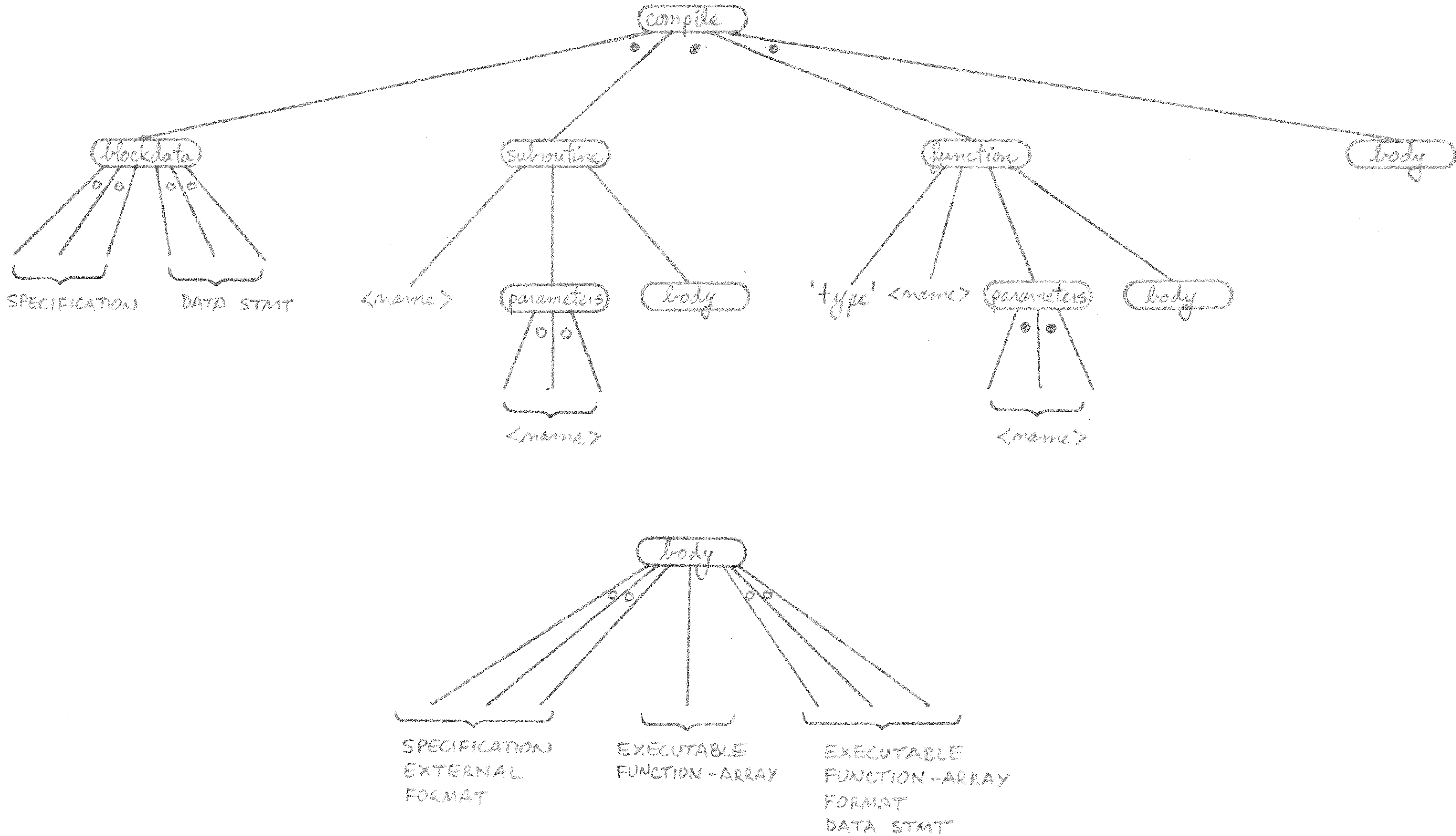



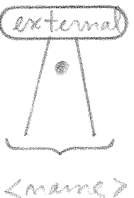
# OVERALL PROGRAM STRUCTURE



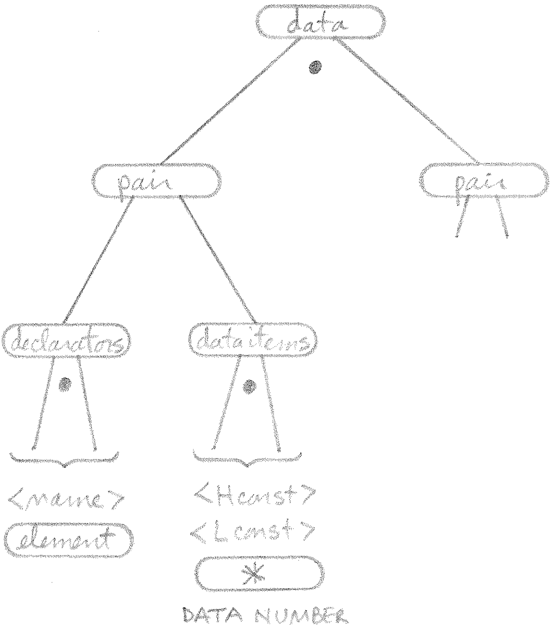

NOTES:

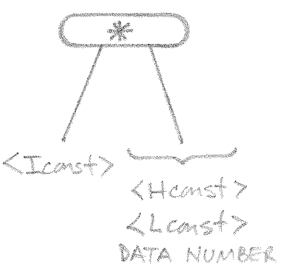
- ANY STATEMENT MAY BE LABELED. FORMAT STMT MUST BE LABELED.
- 'type' MAY BE ONE OF: [INTEGER, REAL, DOUBLEPRECISION, COMPLEX, LOGICAL, DEFAULT]

## EXTERNAL

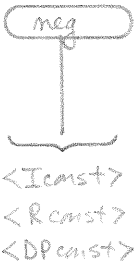


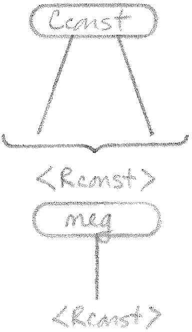

## DATA STMT



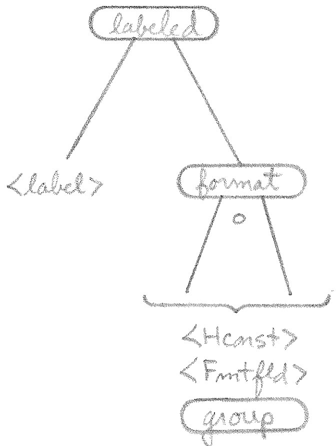



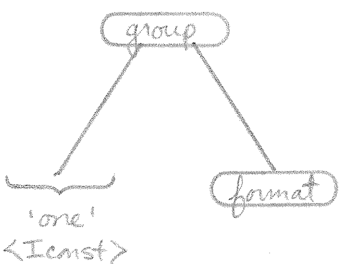

## DATA NUMBER



## FORMAT

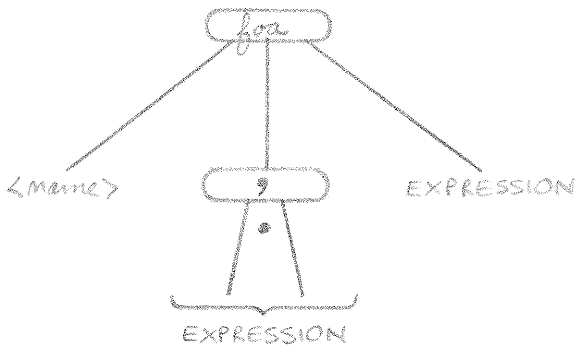


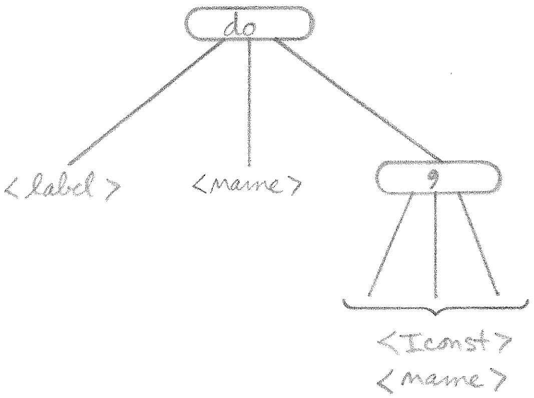

NOTE:

—NO PATH FROM *labeled* NODE TO LEAF MAY CONTAIN MORE THAN THREE *format* NODES

## FUNCTION - ARRAY



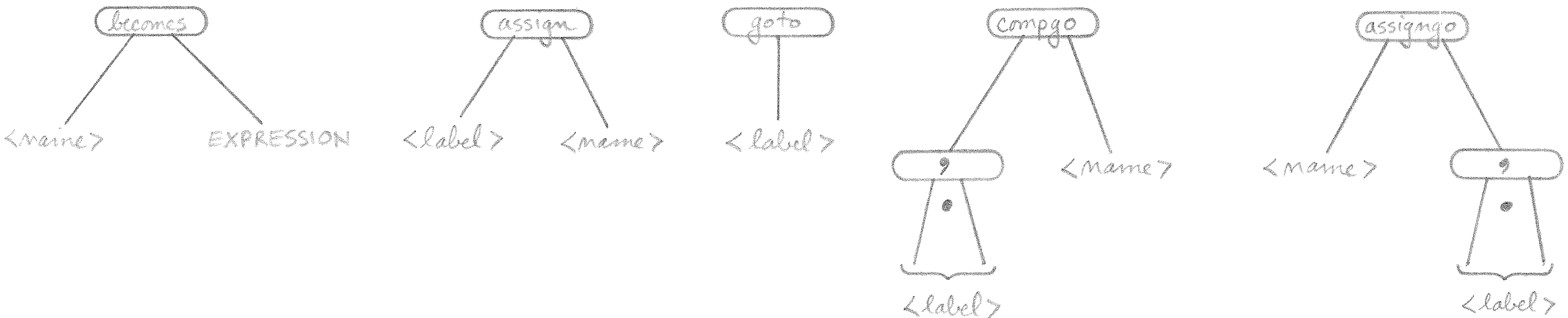
# EXECUTABLE



NOTE:

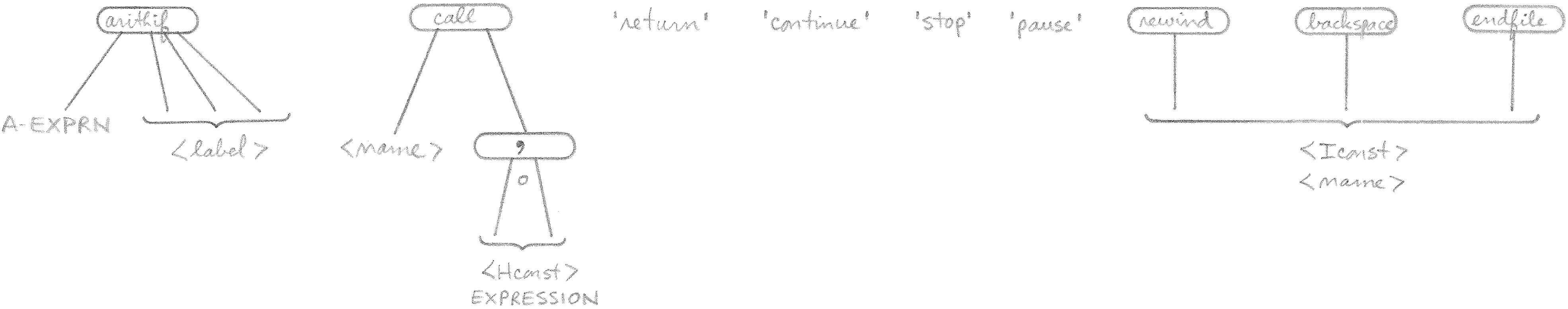
—THERE MAY BE TWO OR THREE DO PARAMETERS (THREE SHOWN)

# BASIC STMT



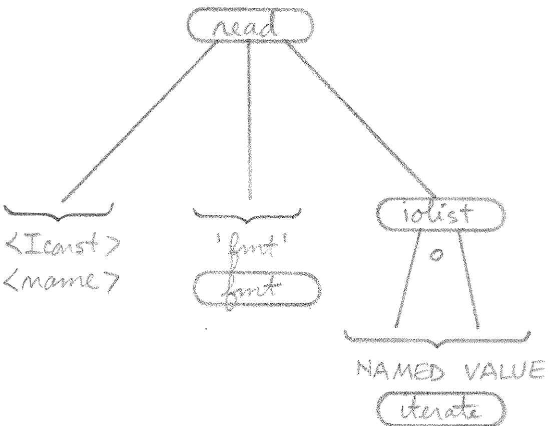

(CONTINUED ON NEXT PAGE)

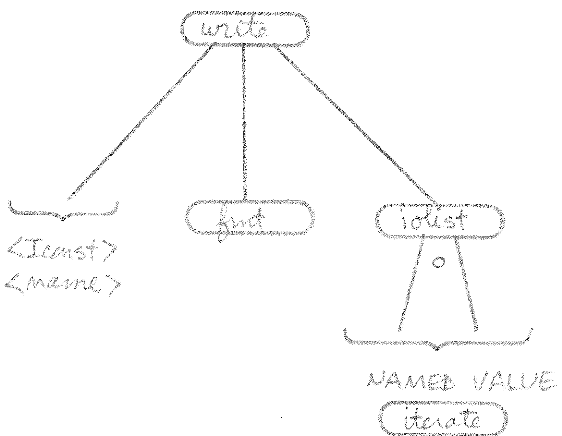
## BASIC STMT (CONTINUED)

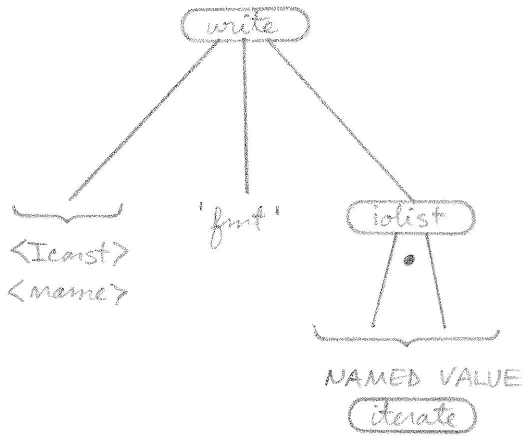


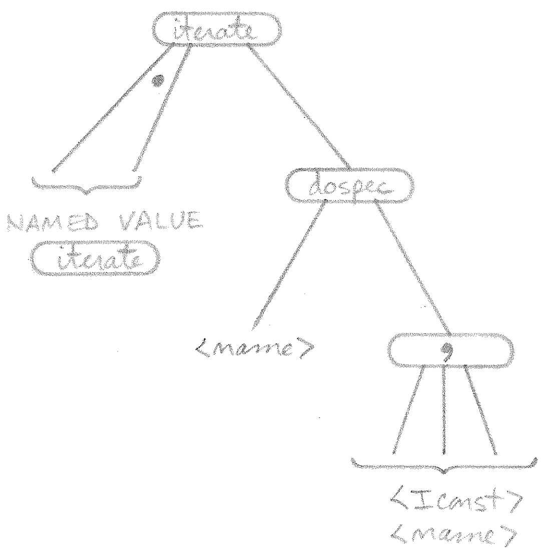

(CONTINUED ON NEXT PAGE)

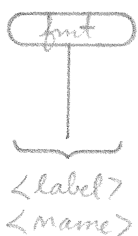
# BASIC STMT (CONTINUED)



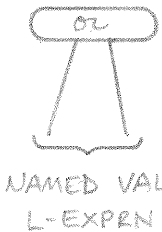

NOTE:

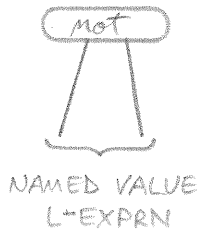
— dospec MAY HAVE TWO OR THREE PARAMETERS (THREE SHOWN)

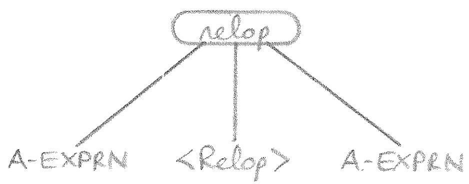
## EXPRESSION

L-EXPRN A-EXPRN

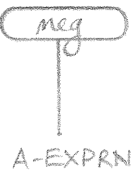
## L-EXPRN

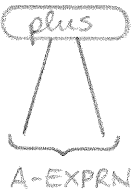


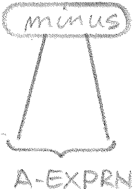


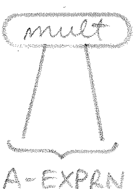



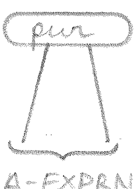
## A-EXPRN



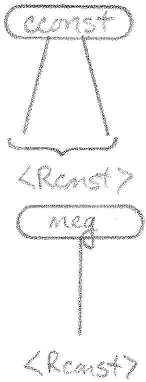





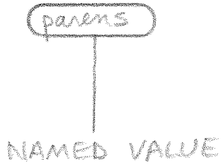

(CONTINUED ON NEXT PAGE)

# A-EXPRN (CONTINUED)

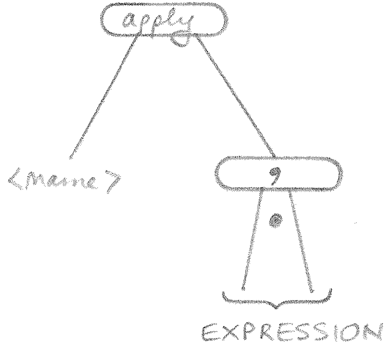
NAMED VALUE <Iconst> <Rconst> <DPconst>



# NAMED VALUE




<name>



## Appendix B

### Expression Grammar Transformations

A straightforward grammar for FORTRAN expressions is shown in Figure B1. The following abbreviations are used:

E - expression
LE - logical expression
LT - logical term
LF - logical factor
LP - logical primary
<Lconst> - logical constant (.TRUE., .FALSE.)
RE - relational expression
AE - arithmetic expression
AT - arithmetic term
AF - arithmetic factor
AP - arithmetic primary
N - number (integer, real, or double precision constant)
CC - complex constant
NV - named value (simple variable, array element, or function call)

Notice that this grammar allows a named value (NV) to appear in any context where either a logical expression (LE) or an arithmetic expression (AE) is required. For example, in the FORTRAN logical if statement

```
IF (X) GO TO 10
```

the named value X plays the role of a logical expression, while in the arithmetic if statement

```
IF (Y) 20, 30, 40
```

Y takes the part of an arithmetic expression.

A serious problem occurs, however, when a named value is asked to fill the role of a general expression (E). For example, the right-hand-side of an assignment statement simply requires an expression; either logical or arithmetic will do. When a named value is encountered in this context, the parser does not know how to reduce that named value to expression: should it first reduce to logical primary (LP) and then continue with reductions involving logical entities, or should it first

```
E → AE
E → LE
```

```
LE → LT
LE → LE 'or' LT
LT → LF
LT → LT 'and' LF
LF → LP
LF → 'not' LP
LP → '<Lconst>'
LP → RE
LP → '(' LE ')'
LP → NV
```

```
AE → AT
AE → AE '+' AT
AE → AE '-' AT
AE → '+' AT
AE → '-' AT
AT → AF
AT → AT '*' AF
AT → AT '/' AF
AF → AP
AF → AP '**' AP
AP → N
AP → CC
AP → '(' AE ')'
AP → NV
```

Figure B1

A simple grammar for FORTRAN expressions

reduce to arithmetic primary (AP) and take the "arithmetic route"? Both paths eventually lead to expression.

As a result, the grammar shown in Figure B1 is not LALR(1). Intuiitively, the reason is that type attributes of named values are not known during parsing.

One possible solution to the type distinction problem is to combine the separate sub-grammars for logical and arithmetic expressions into a single grammar, similar to the approach taken in Pascal [7]. The resulting grammar is shown in Figure B2. Extra processing will now be required during later phases of analysis to verify that:

1) expressions do not inappropriately contain both logical and arithmetic operators, and
2) logical and arithmetic expressions correctly appear in contexts where they are required.

Unfortunately, the grammar of Figure B2 is not LALR(1) either. The production required for relational expression (RE) has caused the nonterminal E to become both left- and right-recursive. Pascal avoids this problem by placing the syntactic description for relational expression "higher" in the grammar. This arrangement has a side effect of requiring parentheses in logical expressions of the form:

(X<5) AND (Y>3).

Since the 1966 ANSI Standard clearly indicates that such parentheses are not required in FORTRAN, it is not possible to similarly modify the grammar of Figure B2. Thus, the combined sub-grammar approach to attaining the LALR(1) property must be abandoned.

Consider again the simple grammar of Figure B1. Another possible solution is to remove one of the productions LP → NV or AP → NV. This may be accomplished by use of the well-known "back substitution" technique, a process which guarantees that the language being generated does not change (see Lemma 4.2 in [5]). LP → NV is arbitrarily chosen for removal.

During back subsititution, two new productions are added to compensate for the loss of LP → NV:

LF → 'not' NV
LF → NV.

E → T
E → '+' T
E → '-' T
E → E '+' T
E → E '-' T
E → E 'or' T
T → F
T → T '*' F
T → T '/' F
T → T 'and' F
F → P
F → P '**' P
F → 'not' P
P → N
P → CC
P → '(' E ')'
P → '<Lconst>'
P → RE
P → NV

RE → E '<Relop>' E

Figure B2

Grammar with logical and arithmetic expressions combined.
The production for RE causes E to become both left- and right-recursive.

Notice that the second of these is another of the form $\alpha \rightarrow$ NV (where $\alpha$ is some non-terminal) and must therefore be removed. Figure B3 shows the grammar that results after four such applications of back substitution. The table below indicates the production removed at each step:

| STEP | PRODUCTION REMOVED |
|---|---|
| 1 | LP → NV |
| 2 | LF → NV |
| 3 | LT → NV |
| 4 | LE → NV |

The important consequence of this action has been to remove LP → NV in favor of E → NV. On the surface it appears that a similar removal of AP → NV will solve the problem, since E → NV would then be the only remaining production of the form $\alpha \rightarrow$ NV. However, the back substitution designed to eliminate LP → NV has uncovered a deeper problem. Consider, for example, the FORTRAN assignment statement X = (Y), in which a parenthesized named value appears in a context where a general expression is required. There are still two possibilities for reduction to E:

1) By using LP → '(' NV ')' as the first step in the reduction, or
2) By first reducing NV to AE (using AP → NV as a first step) and then by reducing '(' AE ')' to E (via the production AP → '(' AE ')').

It is now clear that the problem with the simple grammar of Figure B1 is not just one of reducing NV to E, but involves the reduction of parenthesized NV's as well, where nesting levels may be arbitrarily deep!

With this in mind, imagine the effects of continuing with more rounds of back substitution. Each round begins with elimination of a production

LP → '('...'(' NV ')'...')',

where the depth of nesting has increased by one from the previous round. The grammar grows larger and larger, but a convenient pattern has emerged: newly added productions are similar to those seen in Figure B3, but with ever-increasing sets of parentheses surrounding those positions where an NV appears.

If this process were to continue indefinitely, a new non-terminal could be introduced to take advantage of this pattern:

E → AE

E → LE

E → NV

LE → LT

LE → LE 'or' LT

LE → LE 'or  NV

LE → NV 'or' LT

LE → NV 'or' NV

LT → LF

LT → LT 'and' LF

LT → LT 'and' NV

LT → NV 'and' LF

LT → NV 'and' NV

LF → LP

LF → 'not' LP

LF → 'not' NV

LP → '<Lconst>'

LP → RE

LP → '(' LE ')'

LP → '(' NV ')'

AE → AT

AE → AE '+' AT

AE → AE '-' AT

AE → '+' AT

AE → '-' AT

AT → AF

AT → AT '*' AF

AT → AT '/' AF

AF → AP

AF → AP '**' AP

AP → N

AP → CC

AP → '(' AE ')'

AP → NV

EX → 'LOGIF' '(' LE ')' BS

EX → 'LOGIF' '(' NV ')' BS

Figure B3

Expression grammar after four steps of back substitution. The additional production required for logical-if statement is also shown, with abbreviations:

EX - executable statement

'LOGIF' - IF token for logical-if statement

BS - basic statement (any executable except DO or logical-if)

PNV → NV

PNV → '(' PNV ')'.

This non-terminal, pronounced "parenthesized named value", captures the notion of a named value enclosed by zero or more sets of parentheses. It may be used to "collapse" similar productions, thereby shortening the grammar without changing the language generated.

Figure B4 shows the simplified grammar which results. Notice that a potentially troublesome production, of the form LP → '('•••'(' NV ')'•••')', has been safely dropped from the grammar since after a sufficiently large number of back substitutions it represents a logical primary containing so many parentheses that there is not room to fit them all into a standard FORTRAN statement (limited to 19 continuation lines).

When a similar process is applied to arithmetic expressions, as shown in Figure B5, the original problem finally disappears. Standalone named values (possibly enclosed in parentheses) may be unambiguously reduced, first to PNV and then to E. Named values appearing in more complicated expressions are recognized by the productions which were added during back substitution.

Although the grammar of Figure B5 satisfies the LALR(1) property, it is convenient to simplify it by means of the following steps:

1) Replace all occurrences of AE in the grammar to SAE (simple arithmetic expression). The productions for E become:

   E → LE

   E → SAE

   E → PNV

2) Introduce a new "intermediate" non-terminal AE, such that:

   E → LE

   E → AE

   AE → SAE

   AE → PNV

3) Use the new non-terminal to collapse productions involving basic statement (BS) and relational expression (RE).

The final LALR(1) grammar for FORTRAN expressions is shown in Figure B6.

```
E → AE
E → LE
E → PNV

PNV → NV
PNV → '(' PNV ')'

LE → LT
LE → LE 'or' LT
LE → LE 'or' PNV
LE → PNV 'or' LT
LE → PNV 'or' PNV
LT → LF
LT → LT 'and' LF
LT → LT 'and' PNV
LT → PNV 'and' LF
LT → PNV 'and' PNV
LF → LP
LF → 'not' LP
LF → 'not' PNV
LP → '<Lconst>'
LP → RE
LP → '(' LE ')'

AE → AT
AE → AE '+' AT
AE → AE '-' AT
AE → '+' AT
AE → '-' AT
AT → AF
AT → AT '*' AF
AT → AT '/' AF
AF → AP
AF → AP '**' AP
AP → N
AP → CC
AP → '(' AE ')'
AP → NV

EX → 'LOGIF' '(' LE ')' BS
EX → 'LOGIF' '(' PNV ')' BS
```

Figure B4

Grammar after many steps of back substitution and subsequent simplification via the introduced nonterminal PNV.

```
E → AE
E → LE
E → PNV

PNV → NV
PNV → '(' PNV ')'

LE → LT
LE → LE 'or' LT
LE → LE 'or' PNV
LE → PNV 'or' LT
LE → PNV 'or' PNV
LT → LF
LT → LT 'and' LF
LT → LT 'and' PNV
LT → PNV 'and' LF
LT → PNV 'and' PNV
LF → LP
LF → 'not' LP
LF → 'not' PNV
LP → '<Lconst>'
LP → RE
LP → '(' LE ')'

EX → 'LOGIF' '(' LE ')' BS
EX → 'LOGIF' '(' PNV ')' BS

AE → AT
AE → AE ('+'|'-') AT
AE → AE ('+'|'-') PNV
AE → PNV ('+'|'-') AT
AE → PNV ('+'|'-') PNV
AE → ('+'|'-') AT
AE → ('+'|'-') PNV
AT → AF
AT → AT ('*'|'/') AF
AT → AT ('*'|'/') PNV
AT → PNV ('*'|'/') AF
AT → PNV ('*'|'/') PNV
AF → AP
AF → AP '**' AP
AF → AP '**' PNV
AF → PNV '**' AP
AF → PNV '**' PNV
AP → N
AP → CC
AP → '(' AE ')'

BS → 'ARITHIF' '(' AE ')' '<label>' ',' '<label>' ',' '<label>'
BS → 'ARITHIF' '(' PNV ')' '<label>' ',' '<label>' ',' '<label>'
RE → AE '<Relop>' AE
RE → AE '<Relop>' PNV
RE → PNV '<Relop>' AE
RE → PNV '<Relop>' PNV
```

Figure B5

Resulting grammar after back substitution and simplification in the arithmetic expression sub-grammar. Additional productions required for basic statement (BS) and relational expression (RE) are also shown.

```
E → AE
E → LE

PNV → NV
PNV → '(' PNV ')'
```

```
LE → LT
LE → LE 'or' LT
LE → LE 'or' PNV
LE → PNV 'or' LT
LE → PNV 'or' PNV
LT → LF
LT → LT 'and' LF
LT → LT 'and' PNV
LT → PNV 'and' LF
LT → PNV 'and' PNV
LF → LP
LF → 'not' LP
LF → 'not' PNV
LP → '<Lconst>'
LP → RE
LP → '(' LE ')'


EX → 'LOGIF' '(' LE ')' BS
EX → 'LOGIF' '(' PNV ')' BS
```

```
AE → PNV
AE → SAE

SAE → AT
SAE → SAE ('+'|'-') AT
SAE → SAE ('+'|'-') PNV
SAE → PNV ('+'|'-') AT
SAE → PNV ('+'|'-') PNV
SAE → ('+'|'-') AT
SAE → ('+'|'-') PNV
AT → AF
AT → AT ('*'|'/') AF
AT → AT ('*'|'/') PNV
AT → PNV ('*'|'/') AF
AT → PNV ('*'|'/') PNV
AF → AP
AF → AP '**' AP
AF → AP '**' PNV
AF → PNV '**' AP
AF → PNV '**' PNV
AP → N
AP → CC
AP → '(' SAE ')'
```

```
BS → 'ARITHIF' '(' AE ')' '<label>' ',' '<label>' ',' '<label>'
RE → AE '<Relop>' AE
```

Figure B6

Final LALR(1) grammar for FORTRAN expressions

## Bibliography

[1] American National Standards Institute, FORTRAN, ANSI X3.9, 1966.

[2] Osterweil, L. J. and Fosdick, L. D., DAVE - a validation, error detection, and documentation system for FORTRAN programs. Software practice and experience, 6, 4, October - December, 1976.

[3] Rauhauser, R. C., Use of the BOBSW system to generate a parser for the DAVE II project. Tech. report CU-CS- 78, University of Colorado, forthcoming.

[4] Gries, D., Compiler construction for digital computers. John Wiley and Sons, New York, 1971.

[5] Hopcroft, J. E. and Ullman, J. D., Formal languages and their relation to automata. Addison-Wesley, Reading, Mass., 1969.

[6] LaLonde, W. R., An efficient LALR-parser-generator. Tech. report CSRG-2, University of Toronto, 1971.

[7] Jensen, K. and Wirth, N., Pascal user manual and report. Springer-Verlag, New York, Second Ed., 1975.

[8] McIlroy, M. D., ANS FORTRAN charts. Computer science technical report #13, Bell Laboratories, Murray Hill, N. J.