LR(k) Parser Generator and its Application to FORTRAN Compilers

Tomihiko Kojima*, Masamichi Kato**, and Ikuo Nakata**

1. Introduction

We developed a parser generating system, LR(k) ANALYZER, which constructs an optimized SLR(1) or LALR(k) parser in tabular form from a given syntax description in BNF (Backus-Naur Form) with some additional information concerning the semantics of a language.

Using this system in 1971 we implemented FORTRAN compilers in order to investigate the kind of semantic routines which should be added to the generated parser in constructing the compiler as well as the practicality of the compiler itself. LR(k) ANALYZER is used now as a tool for compiler design and development.

We reformed F. L. DeRemer's algorithm[2)] to construct a parser and make it suitable to computer processing. We also added some optimization algorithms to it.

LR(k) ANALYZER can generate an efficient parser by considering whether each syntax rule has semantic actions or not.

We will describe the automatic generation of a parser by this system and give practical examples applied to the construction of FORTRAN compilers.

LR(k) ANALYZER, which is coded in FORTRAN, is a 4500 statement-size program.

2. LR(k) ANALYZER

2.1 Descriptive form of Input Grammars

The following is the input information for LR(k) ANALYZER.

(1) Syntax rules of a language written in BNF.

(2) Output symbols at reduction states which can be written after each syntax rule.

(3) Action routine names. Action routines are semantic routines that are executed at the time of syntax analysis. They can be inserted in the right hand side of each syntax rule.

(4) The symbol designating the syntax rules which has no semantic actions in the reduction states (such as ⟨FACTOR⟩ → ⟨PRIMARY⟩). This symbol is used to optimize a parser.

Fig. 1 shows a simple example of an input grammar G. The symbols enclosed in (( )) are output symbols and the symbols enclosed in ≪ ≫ are action routine names. The symbol !

```
<PROGRAM>      ::= <DECLARATION><PROG BODY>END(( END ))
<DECLARATION>::= <TYPE ST>! | <DECLARATION><TYPE ST>!
<TYPE ST>      ::= REAL <<R_TYPE SET>> <DCL LIST>;! |
                   INTEGER <<I_TYPE SET>> <DCL LIST>;!
<DCL LIST>     ::= IDENTIFIET | <DCL LIST>,IDENTIFIER
<PROG BODY>    ::= <PROG BODY><ASSIGNMENT>! |
                   <ASSIGNMENT>!
<ASSIGNMENT> ::= IDENTIFIER=<E>;(( = ))
<E>            ::= <E>+<T>(( + )) | <E>-<T>(( - )) | <T>!
<T>            ::= <T>*<F>(( * )) | <T>/<F>(( / )) | <F>!
<F>            ::= <P>**<F>(( ** )) | <P>!
<P>            ::= (<E>)! | IDENTIFIER! | CONSTANT!
```

Fig. 1 Input Grammar G

This paper first appeared in Japanese in Joho－Shori (Journal of the Information Processing Society of Japan), Vol. 15, No. 2 (1974), pp. 93～100.

* Central Research Laboratory, Hitachi, Ltd., Kokubunji, Tokyo, Japan
** Systems Development Laboratory, Hitachi, Ltd., Totsuka, Yokohama, Japan

after syntax rules indicates that they have no semantic actions in reduction states. This example is used in the construction of a translator which converts source programs to reverse Polish lists. In this example, it is assumed that identifiers and constants in assignment statements are immediately output after they have been scanned, and they are not designated as output symbols.

An empty symbol can be also used in input grammars.

2.2 Generation of Parsers

In the LALR(k) parser generator developed by W. R. Lalonde[3], the LR(0) machines are constructed by representing a configuration set with a bit matrix which is based on the Knuth-Earley algorithm. In our system, however, we construct LR(0) machines based on the table where state names are inserted in the syntax rules as shown in Fig. 2 in order to simplify the processing of the computer.

LR(k) ANALYZER constructs a parser from this table through the following step 1 to step 6.

Step 1 constructs a push-down automaton from the characteristic finite state automaton as shown in Fig. 2.

In this step the terminal transition table and the non-terminal transition table are generated. For example, the terminal transition, (STATE 18 * STATE 19), and the two non-terminal transitions, (<T> <T> STATE 18) and (STATE 19 <F> #14) are obtained from the 15th line in Fig. 2. The terminal transition (STATE 18 * STATE 19) means that STATE 19 becomes the new current state when the input symbol '*' is read in STATE 18. On the other hand, the non-terminal transition, (STATE 19 <F> #14) means that #14 becomes the new current state if the state name, STATE 19, is located on the top of the stack in the look-back state corresponding to the non-terminal symbol <F>. The non-terminal symbol <T>, on the left side of (<T> <T> STATE 18), is considered to be the sum of all states where <T> can be read. In this example, as the states where <T> can be read are STATES 12, 15, 17 and 24, the <T> on the left side of (<T> <T> STATE 18) is replaced by these four states.

Step 2 makes the push-down automaton deterministic (except in the case of inadequate states) by merging, splitting and deleting the states. For example, STATE 14 in Fig. 2 is split into STATE 13 and STATE 25. As a result, STATE 14 becomes unnecessary and is deleted. STATES 3, 8 and 10 are also deleted in the same manner.

```
<PROGRAM>         <DECLARATION>     STATE  1          <PROG BODY>       STATE  2          END               #  1              OUT  1
<DECLARATION>     <TYPE ST>         #  2
<DECLARATION>     <DECLARATION>     STATE  3          <TYPE ST>         #  3
<TYPE ST>         REAL              STATE  4          ACT  1            <DCL LIST>        STATE  5          ;                 #  4
<TYPE ST>         INTEGER           STATE  6          ACT  2            <DCL LIST>        STATE  7          ;                 #  5
<DCL LIST>        IDENTIFIER        #  6
<DCL LIST>        <DCL LIST>        STATE  8          ,                 STATE  9          IDENTIFIER        #  7
<PROG BODY>       <PROG BODY>       STATE 10          <ASSIGNMENT>      #  8
<PROG BODY>       <ASSIGNMENT>      #  9
<ASSIGNMENT>      IDENTIFIER        STATE 11          =                 STATE 12          <E>               STATE 13          ;
                  # 10              OUT  2
<E>               <E>               STATE 14          +                 STATE 15          <T>               # 11              OUT  3
<E>               <E>               STATE 14          -                 STATE 17          <T>               # 12              OUT  4
<E>               <T>               # 13
<T>               <T>               STATE 18          *                 STATE 19          <F>               # 14              OUT  5
<T>               <T>               STATE 18          /                 STATE 21          <F>               # 15              OUT  6
<T>               <F>               # 16
<F>               <P>               STATE 22          **                STATE 23          <F>               # 17              OUT  7
<F>               <P>               # 18
<P>               (                 STATE 24          <E>               STATE 25          )                 # 19
<P>               IDENTIFIER        # 20
<P>               CONSTANT          # 21
```

Fig. 2 Characteristic Automaton of the Grammar G

Step 3 computes read-states which should be stacked (called stacked-states). It is necessary to decrease the number of stacked-states not only to save stack's house-keeping time but also to simplify the parser. Stacked-states can be determined from the non-terminal transition table. We need not "look back" the stack if the destination state can be definitely determined no matter what states are located on the top of the stack. This decreases the number of stacked-states. In the case of grammar G in Fig. 1, the stacked-states are STATES 12, 15, 17, 19, 21, 23 and 24 (they are preceded by @ in Table 1).

Step 4 computes look-ahead sets if inadequate states exist. LR(k) ANALYZER first computes SLR(1) look-ahead sets and then LALR(k) (where $k \leqq 3$) look-ahead sets if there exist inadequate states which are not SLR(1), in this case LALR(3) grammar can be accepted.

Step 5 computes the number of stacked-states which must be poped up in each reduction state.

Step 6 optimizes the parser.

2.3 Optimization of Parsers

LR(k) ANALYZER optimizes parsers as follows.

(1) Avoidance of look-ahead

Four look-ahead states occur from the finite automaton of Fig. 2. Fig. 3 shows one of them, that is, the transition, (STATE 15 ⟨T⟩ LA 2) is generated to eliminate inadequacy occuring from the two transitions, (STATE 15 ⟨T⟩ STATE 18) and (STATE 15 ⟨T⟩ #11), where LA 2 is a look-ahead state.

The look-ahead set for STATE 18 is { *, / } and that for #11 is { ;, +, -,) }. The part enclosed in asterisks in Fig. 3 shows an optimized form of the look-ahead transition. As STATE 18 is not a stacked-state and has no action routines, in the look-ahead state LA 2, the input symbol is read (not "looked ahead") if the input symbol is '*' or '/'. As a result either STATE 19 or STATE 21 becomes the new current state. In the case of other input symbols, it is not read and the reduction state #11 becomes the new current state.

STATE 18 is deleted because no control is passed. This results in absolutely no change in the behavior of the parser.

LR(k) ANALYZER represents a parser by using four kinds of transition tables--terminal transition table, look-ahead table, look-back table and reduction table. In the case of the grammar G in Fig. 1, Table 1 ~ Table 4 are obtained as these four tables. The following descriptions follow these tables.

(2) Sharing of terminal transitions

If the set of terminal transitions starting from one state is a subset of the set of those starting from another state, the former transitions are shared. For example, in Table 1, the transitions starting from STATE 12 are also used as the transitions starting from STATES 15, 17, 19, 21, 23 and 24.

```
    THIS INADEQUATE STATE IS SLR(1)
STATE 15   <T>   STATE 18   *   STATE 19
                            /   STATE 21
STATE 15   <T>       # 11   ;       # 10
                            +   STATE 15
                            -   STATE 17
                            )       # 19
STATE 18 IS NOT A STACK-STATE, THEREFORE
***************************************
*(STATE 15)   <T>   LA 2   *   STATE 19*
*                          /   STATE 21*
*                      OTHERS      # 11*
***************************************
```

Fig. 3 Optimization of a Look Ahead Transition

(3) Optimization of the look-back table

The look-back table is obtained from the non-terminal transition table. The states located on the top of the stack in each look-back state are grouped per destination state. The group having the most members is considered as one state called "OTHERS". This also results in absolutely no change in the behaviour of the parser.

. The transition tables are made remarkably compact by the above optimizations (2) and (3).

(4) Deletion of unnecessary reduction states

As the syntax rules with no semantic actions are designated in input grammar, LR(k) ANALYZER can find and delete unnecessary reduction states. Thirteen reduction states are deleted in Table 4.

3. Further Optimization

Besides automatic optimization of parsers by LR(k) ANALYZER, optimizations by hand or by using simulator are also carried out. This includes the following optimization.

(1) Partial sharing of terminal transitions

(2) Pruning of arithmetic expressions

(3) Making the table smaller by using the characteristics of a language

(4) Rearranging the terminal transitions by tracing the dynamic action of parsers using LR(k) SIMULATOR.

Pruning of arithmetic expression is described using the above transition tables.

Table 1 Terminal Transition Table

current state	input symbol	next state
STATE 0	REAL	STATE 4 /
	INTEGER	STATE 6 /
STATE 1	REAL	STATE 4
	INTEGER	STATE 6
	IDENTIFIER	STATE 11
STATE 2	END	# 1
	IDENTIFIER	STATE 11
STATE 4	<R_TYPE SET>	
	IDENTIFIER	STATE 5
STATE 5	;	STATE 1
	,	STATE 9
STATE 6	<I_TYPE SET>	
	'SAME AS'	STATE 4
STATE 9	'SAME AS'	STATE 4
STATE 11	=	STATE 12
@ STATE 12	IDENTIFIER	LA 4
	(	STATE 24
	CONSTATNT	LA 4
STATE 13	;	# 10
	+	STATE 15
	-	STATE 17
@ STATE 15	'SAME AS'	STATE 12
@ STATE 17	'SAME AS'	STATE 12
@ STATE 19	'SAME AS'	STATE 12
@ STATE 21	'SAME AS'	STATE 12
@ STATE 23	'SAME AS'	STATE 12
@ STATE 24	'SAME AS'	STATE 12
STATE 25	+	STATE 15
	-	STATE 17
	)	# 19

Table 2 Look Ahead Table

current state	input symbol	next state
LA1	*	STATE 19
	/	STATE 21
	OTHERS	LB 1
LA 2	*	STATE 19
	/	STATE 21
	OTHERS	# 11
LA 3	*	STATE 19
	/	STATE 21
	OTHERS	# 12
LA 4	**	STATE 23
	OTHERS	LB 3

Table 3 Look Back Table

current state	top of stack	next state
LB 1 (<E>)	STATE 12	STATE 13
	OTHERS	STATE 25
LB 2 (<T>)	STATE 15	LA 2
	STATE 17	LA 3
	OTHERS	LA 1
LB 3 (<F>)	STATE 19	# 14
	STATE 21	# 15
	STATE 23	# 17
	OTHERS	LB 2

Table 4 Reduction Table

current state	the number of pop-up	output symbol	next state
# 1	POP 2	END	EXIT
# 10	POP 1	=	STATE 2
# 11	POP 1	+	LB 1
# 12	POP 1	-	LB 1
# 14	POP 1	*	LB 2
# 15	POP 1	/	LB 2
# 17	POP 1	**	LB 3
# 19	POP 1		LA 4

Table 5 Optimized Look Ahead Table

current state	input symbol	next state
LA 1	*	STATE 19
	/	STATE 21
	OTHERS	LB 2
LA 2	**	STATE 23
	OTHERS	LB 3

In the states where the non-terminal symbol < E > is read, i.e., STATE 12 and STATE 24, control can be directly transfered to the < E >'s look-back state, i.e., LB 1, when the identifier or the constant has been read if the next input symbol belongs to the set { +, -,), ; }. As simple assignment statements such as A = B appears more frequently, the short cut mentioned above should be prepared in the transition system so that they are parsed rapidly.

We often reverse the operation order of look-back and look-ahead. In Table 3, for example, look-ahead is done in states LA 1, LA 2 or LA 3 after look-back has been done in state LB 2. In this case it is more efficient, however, to look-ahead the input symbol first. Then Table 2 is reduced to 5 lines as shown in Table 5 ("next states" in another tables should be suitably modified).

Now we are going to describe the LR(k) SIMULATOR which consists of a simple syllable reader and an interpreter of parsing tables. It is a program to trace dynamic actions of parsers and translate source programs to intermediate languages.

This program verifies whether parsers optimized by hand operate correctly. Statistical data relating to state transitions and execution frequency of operations are also output. Using this information we can optimize parsers by rearranging the transitions. The LR(k) SIMULATOR is also used to estimate the efficiency of compilers, and thus one can find the key points to further improve them.

4. FORTRAN

We applied LR(k) ANALYZER to FORTRAN, ALGOL and SNOBOL 4 grammars. As a result we were able to find out that all these grammars are close to SLR(1) grammar. To set up one of these examples, FORTRAN will be described.

4.1 Analysis of FORTRAN Grammars

In case identifiers are written in different names according to their attributes in FORTRAN grammar (for example, < ARRAY NAME >), this grammar is close to SLR(1) grammar. In this case, the syllable reader need not necessarily generate different token-kinds per attribute of identifiers. The attribute can be checked at the time of syntax analysis by providing a parser with the operator "CHECK". In FORTRAN grammar, only one inadequate state is not an SLR(1) state (except for a few states where we must use semantic information). This is the state where a control variable in < DO-implied list > and a variable as < Input/Output list element > should be distinguished. However, this state is also an LALR(1) state.

Next we input the grammar where almost all identifiers were written under the same name independently of their attributes. This time, the grammar became ambiguous. It can often be solved by look-aheads of only one input symbol if detection of semantic errors is postponed to the next phase. But it is sometimes more practical to use semantic information. As a result, we found that FORTRAN grammar is close to SLR(1) grammar and LR(k) parsing techniques can easily be applied to this language.

4.2 Data Structure of Parsing Tables

We constructed a parsing table based on the output of LR(k) ANALYZER. The data structure of the table which consists of a state table and a transition table is nearly the same as that of DeRemer's parsing table. Fig. 4 shows the contents of one entry of each table.

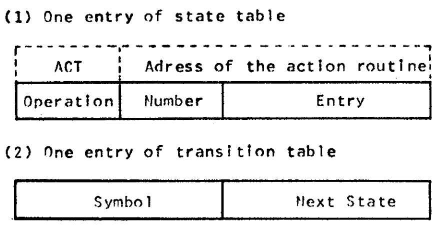
(1) One entry of state table
ACT
Adress of the action routine
Operation
Number
Entry
(2) One entry of transition table
Symbol
Next State

Fig. 4 Data Structure of the Parsing Table

The operation part of a state table contains an operator such as READ, STACK and READ, LOOK-AHEAD, LOOK-BACK, POP-UP, OUTPUT, CHECK, CALL, etc. which operates in the state. The number part contains the number of transitions starting from the state or the number of the stacked-state which must be poped-up in the state. The entry part usually contains the pointer which points to the entry of the transition table, but if the operator is "OUTPUT" or "POP-UP" then the output symbol or the destination state is located there.

In the states where action routines are used, the operator ACT and the address of the action routine are added. The relative address from the top of the state table is used as the state number.

The symbol part in the transition table contains the symbol to be pattern-matched (token-kind or state number).

The size of the parsing tables of FORTRAN grammars were respectively 858, 1364, 1416 bytes with respect to the specification level 3000, 5000, 7000 of JIS (Japanese Industrial Standard).

5. Application to FORTRAN Compilers

We constructed FORTRAN parsers using LR(k) ANALYZER and implemented FORTRAN compilers based on the parsers. We will describe in this section the application method of LR(k) ANALYZER to the 1-pass resident FORTRAN compiler for mini-computer HITAC 10 with 4 KW main memory (1 KW = 1024 Words, 1 Word = 16 bits).

The language specification is close to JIS level 3000, but we set the rules that key words are reserved words and a blank is a delimiter in order to simplify the syllable reader. The object program is in the form of reverse Polish notation.

The following is the process used to construct a parser using LR(k) ANALYZER.

First we describe FORTRAN grammar in BNF. Token-kinds, which are output by the syllable reader, are treated as terminal symbols of the grammar. For example, the relational operator '.EQ.', '.GT.', '.LT.', ... may be defined as one terminal symbol. The syllable reader needs only to return the token-kind of relational operator and the index indicating the kind of operator such as '.EQ.', '.GT.'. We may also define such FORTRAN grammar as ⟨ expression ⟩ is a terminal symbol, and after constructing the parsing table for this grammar we can add the parsing table for the ⟨ expression ⟩ grammar.

We extend the FORTRAN grammar so that an arithmetic expression can be described as a subscript expression in order to simplify the compiler. This FORTRAN grammar (including ⟨expression⟩ grammar) can be represented by 90 syntax rules.

Then we design the object code of each statement, and decide what symbols should be output in each state and insert the necessary action routine names into the syntax rules taking the semantics into consideration.

The grammar obtained in this way is input into LR(k) ANALYZER. Compiler designers draw the transition diagram based on the output of LR(k) ANALYZER and optimize it. First we analyzed the grammar without the optimization phase of LR(k) ANALYZER. The parsing table size was 631 words. However, it was reduced to 353 words by using the optimization phase and finally reduced to 280 words by hand.

The action routines written in assembler language became small routines (2 steps～50 steps). There were approximate 20 action routines. For example, these routines perform the following processing.

(1) To check the DO terminal statements
(2) To write identifiers' attributes into the symbol table
(3) To count the number of parameters.

We implemented a 1-pass resident FORTRAN compiler for 4 KW mini-computer use, by adding an input/output routine, library function routines and execution-time routines. The size of the compiler is 1.4 KW. The execution-time routines are 800 W and the user area is 600 W. The size of the action routines in the compiler is 279 W and that of the interpreter of the parsing table is 116 W. The average compiling speed per statement was 19 ms.

6. Conclusion

Aiming at a CAD system for compilers, we were able to develop LR(k) ANALYZER which constructs a parser in tabular form and implemented practical FORTRAN compilers in a short period using this system. Our goal is to generate the whole parser by giving the necessary information in input grammar.

This time, however, we were able to reach the step where we could designate the form of the object language and action routine names.

We want to introduce a formal description concerning error correction and error recovery. We intend to improve this system so that it can be used for the estimation as well as the design and the construction of compilers.

References

1) D. E. Knuth: On the Translation of Languages from Left to Right, Information and Control, Vol. 8, No. 5, pp. 607～639, Dec. 1965

2) F. L. DeRemer: Practical Translation for LR(k) Languages, Ph. D. Thesis. MIT, Cambridge, Mass. Oct. 1969

3) W. R. Lalonde: An Efficient LALR Parser Generator, Technical Report CSRG-2, Toronto, Feb. 1971