論　　文

LR(*k*)-Parser 生成システムとその FORTRAN コンパイラへの応用*

小島富彦**・加藤正道***・中田育男***

Abstract

In this paper the authors describe a parser generating system, LR(*k*) ANALYZER II, which constructs an optimized SLR(1)-or LALR(*k*)-parser in tabular form from given syntax descriptions in BNF and some semantic information of a language.

Results of experiment on the implementation of FORTRAN compilers using this system are also described.

1. まえがき

ソフトウェアの生産性向上は，最近ますます重要な課題となってきており，特にコンパイラについては，多くの努力がなされてきた．D. E. Knuth[1] は，LR(*k*)-parser（構文解析ルーチン）の構成法を創案したが，そのままでは構文解析表が大きくなり過ぎるので，F. L. DeRemer[2] は，これを小さくさせるために，simple LR(*k*)，あるいは LALR(*k*)-parser の構成法を考案した．

われわれは，この手法を用いて作成した parser に，どのような意味処理を追加すれば，コンパイラとして完成するか，でき上がったコンパイラの実用性はどうかなどを調べる目的で，1971 年，プログラミング言語の文法から parser あるいは translator を「表」の形で作り出すシステム，LR(*k*) アナライザを開発し，これを用いて，いくつかの FORTRAN コンパイラを作成した．翌年には，このシステムを改良した LR(*k*) アナライザ II を開発し，現在これを，コンパイラの設計・作成のための治工具として利用している．われわれは DeRemer のアルゴリズムを，そのまま使うのではなく，そのアルゴリズムを計算機で扱い易い形に直すとともに，parser の最適化を自動的に行なわせるためのいくつかのアルゴリズムを付け加えた．LR(*k*) アナライザ II では，入力文法中に，parser の出力記号列や意味処理ルーチン名も挿入することができるようにさせ，意味処理を伴なう構文規則と，伴わない構文規則とを明確に区別することにより，効率のよい parser を生成させることができるようにした．

以下，LR(*k*) アナライザ II による parser の自動作成について述べた後，このシステムを FORTRAN コンパイラの作成に適用した実例について述べる．

なお，LR(*k*) アナライザ II は，FORTRAN で書かれていて，約 4500 ステートメントのプログラムである．

2. LR(*k*) アナライザ II

ここでは，まず LR(*k*) アナライザ II の入力文法の記述形式について述べ，次に入力情報から LR(*k*)-parser を作成し，それを最適化する手法について述べる．

2.1 入力文法の記述形式

LR(*k*) アナライザ II の入力情報の基本となるものは，言語の構文規則をバッカス記法（BNF）で記述したものである．構文解析を行ないながら同時に中間語を出力するような parser を生成させるために，各構文規則の後に，還元（reduction）の際の出力記号列を書き込めるようにした．

LR(*k*) 手法で使う標準的なオペレーションだけで構文解析時の処理を，すべて行なうことはできないので，これを補うために，われわれはアクションルーチンを導入した．アクションルーチンとは，構文解析時

* LR (*k*) Parser Generator and its Application to FORTRAN Compilers, by Tomihiko KOJIMA (Central Research Laboratory, HITACHI, Ltd.), Masamichi KATO and Ikuo NAKATA (Systems Deveropment Laboratory, HITACHI, Ltd.)

** (株)日立製作所中央研究所
*** (株)日立製作所システム開発研究所

に行なうセマンティクス処理ルーチンのことである．BNF の右辺の任意の場所に，アクションルーチン名を書き入れることができるようにした．

構文規則の中には，例えば〈FACTOR〉→〈PRIMARY〉のように，中間語の出力とか意味処理などを伴わないものも多くある．このような構文規則を，入力文法の中で明示してやることにより，効率のよい parser を小さく構成させるようにした．

Fig. 1 は，簡単な入力文法Gの記述例である．《 》で囲まれている記号は出力記号であり，《 》で囲まれている記号は，アクションルーチン名である．！が付いている構文規則は，意味処理を伴わないことを示している．この例は，ソースプログラムを逆ポーランド記法に変換するトランスレータを作成する場合であり，ここでは識別子および定数は，それらが読み込まれたとき直ちに出力されると仮定しているので，出力記号としては指定していない．なお，入力文法中で空記号を使うことも許されている．

2.2 Parser の作成

W. R. Lalonde[3] が作った LALR(k)-parser 生成システムでは，Knuth-Earley のアルゴリズムに基づいて，ビット行列を使って configuration sets を表現することにより，LR(0)-machine を構成したが，われわれは，計算機での処理を簡単にするために，図2のような，構文規則の中に状態名を挿入した表を基にして，LR(0)-machine を構成するようにした．

```
<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>     ::= IDENTIFIER | <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

Fig. 2 のような特性有限状態オートマトンから，LR(k) アナライザ II は，以下の Step 1 から Step 6 までの処理手順を経て，parser を作成する．

Step 1. 文法の導出関係を計算して，プッシュダウンオートマトンを作成する．

Step 2. 状態の統合・分割・消去を実施して，このプッシュダウンオートマトンが，不適合状態 (inadequate states) を除いては決定性になるようにする．ここで，不適合状態とは，1つの入力記号またはスタック上の状態名に対し，2つ以上の遷移先があり，その中に1つ以上の還元状態 (reduce states) が含まれているような状態をいう．(このとき，遷移先の中の読込み状態 (read states) は，高々1つしかない．)

Step 3. 読込み状態（入力記号を読み込む状態）のうち，スタックに積む必要のあるものと，ないものとを区別する．

Step 4. 不適合状態があれば，先読み集合を計算する．

Step 5. 還元状態における，スタックのポップ・アップ数を計算する．

Step 6. parser を最適化する．

Step 1 では，Fig. 2 のような特性有限オートマトンから，終端遷移表と非終端遷移表とを作る．例えば，Fig. 2 の 15 行目から，(STATE 18 ＊ STATE 19) という終端遷移と，(〈T〉 〈T〉 STATE 18)，(STATE 19 〈F〉 #14) という2つの非終端遷移とが得られる．(STATE 18 ＊ STATE 19) は，STATE 18 という状態で入力記号 '＊' を読んだとき，次の状態が STATE 19 になることを意味する遷移である．一方，(STATE 19 〈F〉 #14) は，〈F〉に還元する状態で，一定数だけスタックをポップアップした

```
<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           ACT  1      STATE  4       <DCL LIST>   STATE  5     ;            # 4
<TYPE ST>      INTEGER        ACT  2      STATE  6       <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

のち，スタックの一番上に，STATE 19 という状態名が乗っていれば，次の状態は，#14 という還元状態になることを意味する遷移である．

ここで (〈T〉 〈T〉 STATE 18) という遷移の左端の〈T〉は，〈T〉を読んでもよい状態の集合であると考える．この集合がどのような読込み状態からなっているのかを調べるために，文法の導出関係を計算する．その結果，〈T〉を読んでよい状態が，STATE 12, 15, 17, 24 の4つであることがわかると，(〈T〉〈T〉 STATE 18) は，左端の〈T〉を，これらの4つの読込み状態で置換した遷移群と，入れ換えられる．

Step 2 では，このようにして作ったプッシュダウンオートマトンが，不適合状態を除いて考えても，なお決定性になっていないことがあるので，状態の統合・分割・消去を施すことにより，オートマトンを決定性にする．例えば，Fig. 2 で STATE 14 は，STATE 13 でもありかつ STATE 25 でもあるような状態であるので，STATE 14 を，STATE 13 と STATE 25 とに分割する．その結果，STATE 14 は不要となり消去される．同様にして，STATE 3, 8, 10 も消去される．これとは逆に，2つ以上の読込み状態を統合して，1つの新しい読込み状態を生成する場合もある．この場合にも新しい読込み状態が生成されたことにより不要になる読込み状態があればそれらは消去される．なお，オートマトン中に，空記号による遷移があれば，これも取除いておく．

以上の処理により，オートマトンにおいて決定性でない箇所は，高々，不適合状態だけになる．

Step 3 では，読込み状態のうちスタックに積む必要のあるもの(これをスタック状態とよぶ)を計算する．単に，スタックの house-keeping の時間を節約するためだけでなく，parser を簡略化する上で，スタック状態を少なくすることは必須である．還元状態でスタックをポップ・アップした後，スタックの一番上に乗っている状態名を見て，次の遷移先を決定する動作をルック・バック (look-back) と呼んでいるが，このルック・バックを行なったとき，スタックの一番上に乗っている可能性のある読込み状態名だけが，スタック状態の候補となる．これらの状態は，非終端遷移表により見つけられる．スタックの一番上にどのような状態名が乗っていようとも次の遷移先が一意的に決定できる場合は，ルック・バックを行なわないようにする．これにより，スタック状態は，さらに少なくなる．Step 2 で，状態の統合・分割を行なったので，スタック状態が増えることを心配したが，現実の計算機言語の文法の例で試したところ，これによるスタック状態の増加は，ほとんどなかった．Fig. 1 の文法Gの場合スタック状態は，STATE 12, 15, 17, 19, 21, 23, 24 の以上7つである．(これらには，Table 1 で @ が付けられている．) FORTRAN の場合でも，すべての読込み状態のうち，スタック状態は，半分以下である．

Step 4 では，入力文法が，LR(0) 文法でないとき，不適合状態に対し，先読み集合 (look ahead sets) を計算して，オートマトンを決定性にする．LR(k) アナライザ II は，まず simple LR(1)-parser を作成し，もし，SLR(1) にならない不適合状態があれば，そこだけ，LALR(k) 手法を使って，必要であれば最高3記号先まで調べる．したがって，LALR(3) 文法まで受理することができる．

2.3 最適化

DeRemer, Lalonde は，いくつかの LR(k)-parser の最適化手法を提案したが，われわれは，これらの最適化を，自動的に行なわせるために，アルゴリズム化し，LR(k) アナライザ II に組み入れた．以下，parser の自動最適化について述べる．

(1) 先読みの回避

Fig. 2 の有限オートマトンからは，4つの先読み状態が生じる．Fig. 3 は，そのうちの1つを示したものである．つまり，遷移 (STATE 15 〈T〉 STATE 18) と，遷移 (STATE 15 〈T〉 #11) とから生じた不適合性 (inadequacy) を消すために，(STATE 15 〈T〉 LA 2) という遷移ができる．ここで，LA 2 は先読み状態である．

STATE 18 に対する先読み集合は {*, /} であり，#11 に対するそれは，{;, +, −} である．Fig. 3 で，星印で囲ってある部分は，この先読み状態を最適化したものである．つまり，STATE 18 がスタック状態ではないので，先読み状態 LA 2 で，次の入力記号を見たとき，'*' であれば，それを読み込んで，STATE 19 に遷移し，'/' であれば，同じくこれを読み込

```
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

んで，STATE 21 に遷移してしまう．その他の入力記号の場合には，直ちに #11 に遷移する．このとき，エラーチェック機能が低下することはない．また，STATE 18 は，コントロールが渡ってこなくなることが確められた後，消去される．このような最適化により，ほとんどの先読み状態では，実際に入力記号の先読みを行なわなくて済む．Fig. 1 の文法Gは，SLR (1)文法であり，先読み状態は4つあるが，いずれも，このような最適化により，「読込みと分岐」で済んでしまう．FORTRAN の場合では，この最適化の結果，先読みすべき所は3箇所になったが (Table 5 参照)，それらも人手により，「読込みと分岐」で，済ますことができるようになった．

LR(k) アナライザ II は，終端遷移表，先読み遷移表，ルック・バック表，還元表の4種類の遷移表によって parser を表現する．Fig. 1 の文法Gの場合は，その4つの表として，Table 1～Table 4 が得られる．以下，これらの表に即して述べる．

(2) 終端遷移群の共有

1つの読込み状態から出ている終端遷移群が，他の読込み状態から出ている遷移群に含まれるときは，遷移群の共有を行なう．例えば Table 1 で，STATE 15, 17, 19, 21, 23, 24 から出ている遷移群は，STATE 12から出ている遷移群に一致するので，遷移群の共有ができる．また，STATE 0 (初期状態) から出ている遷移群は，STATE 1 から出ている遷移群に含まれるので，右端に '/' 印が付けられている．

(3) ルック・バック表の最適化

ルック・バックにより遷移先状態を決定するのに，ルック・バック表 (Table 3) が使われる．この表は，非終端遷移表から得られる．遷移先状態ごとに，スタックの一番上に乗っている状態名をグループ分けし，メンバーの最も多いグループを 'OTHERS' としてまとめ，表を小さくするとともに，チェックの回数を減らす．ここでも，エラーチェック機能は，低下しない．

以上の，(2)，(3)の最適化により，遷移表は大幅に縮小される．

(4) 不要な還元状態の消去

セマンティクス処理を伴わない構文規則を，入力文法中で指定したので，LR(k) アナライザ II は，この情報を使って，還元状態でのポップ・アップ数が0になることを確めてから，削除可能な還元状態をオートマトンから消去する．Fig. 1 の入力文法では，全部で13個の還元状態が消去された (Table 4 参照)．この最適化により，スタック状態がさらに減少し，ルック・

Table 1 Terminal transition table

状 態	入力記号	遷 移 先
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	ACT 1	
	IDENTIFIER	STATE 5
STATE 5	;	STATE 1
		STATE 9
STATE 6	,	
	ACT 2	
	'SAME AS'	STATE 4
STATE 9	'SAME AS'	STATE 4
STATE 11	=	STATE 12
@ STATE 12	IDENTIFIER	LA 4
	(	STATE 24
	CONSTANT	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

状 態	入力記号	遷 移 先
LA 1	*	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

状 態	スタックのトップ	遷移先
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

状 態	ポップアップ数	出力記号	遷移先
# 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 Effect of optimization (FORTRAN JIS 5000)

	最適化前	最適化後
終端遷移の個数	629	185
非終端遷移 〃	508	93
先読み状態 〃	31	3
還元状態 〃	147	96

バック表が縮小されるという波及効果が生じる．parser の効率を上げる上で，この処理は欠くことのできないものである．

以上に述べた最適化処理により，FORTRAN の場合，Table 5 に示すような効果が得られた．

3. アナライザ以後

LR(k) アナライザ II による parser の自動最適化の他に，人手やシミュレータによる最適化も行なう．この中には，次のような処理が含まれる．

（1） 終端遷移群の部分的共有を行なう．

（2） 算術式の刈込みを行なう．

（3） 言語の特性を用いて，表を小さくする．

（4） LR(k) シミュレータを使用して，parser の動作解析を行なう．

算術式の刈込みを，先ほどの遷移表を使って述べる．非終端記号〈E〉を読んでよい状態，つまりSTATE 12 と STATE 24 では，識別子や定数を読み込んだとき，次の入力記号が，{+, −,), ;} に属していれば，〈E〉に対応するルック・バック状態，つまりLB 1 に直接遷移することができる．実際，〈代入文〉としては，A=B のような簡単な形をしたものの出現頻度が圧倒的に大きいので，これらを特に速く構文解析できるように，オートマトンの遷移系に，上のような近道を作ってやるのである．算術式以外にも，この手法を適用することができるが，どのような状況の下で，このような刈込みができるかということを，LR(k) アナライザ II の出力情報により知り，それを見て，メモリとの兼合いを考慮して行なう．FORTRAN 文法の例では，算術式の刈込みだけでは，遷移表のメモリ増加は，約 1.5% であった．

一般に，言語に依存する性質を利用すれば，遷移表をさらに小さくすることができるが，われわれは，しばしば，ルック・バックと，入力記号の先読みとの動作順を，交換している．例えば，Table 3 では，状態 LB 2 でルック・バックを行なって，LA 1, LA 2, LA 3 に分岐してから，入力記号の先読みを行なっているが，ここの所は，先に，入力記号を先読みした方が，有利である．そうすれば，Table 2 が，Table 6 に示したように 5 行に縮められる．(このとき，他の表にも遷移先の修正を相応に施す．) このような箇所は，先読み遷移表から，容易に見つけることができる．

Table 6 Optimized look ahead table

状態	入力記号	遷移先
LA 1	*	STATE 19
	/	STATE 21
	OTHERS	LB 2
LA 2	**	STATE 23
	OTHERS	LB 5

次に，LR(k) シミュレータについて述べる．

LR(k) シミュレータは，簡単なシラブルリーダと，構文解析表のインタプリンタとを合わせた機能をもっていて，ソースプログラムを中間語に変換しながら，parser の動作解析を行なうプログラムである．これを使って，人手で最適化したあとの parser が正しく動作することを確めたり，アクションルーチンを実行するタイミングをつかんだりする．状態遷移に関する統計データも出力されるので，それを使って，遷移表の並べ換え等の最適化をすることもできる．また，これをもとに，実際にコンパイラを作成する場合の事前評価を行なったり，性能を上げるための“キー・ポイント”を見つけたりするのに利用する．

4. FORTRAN の場合

FORTRAN, ALGOL, SNOBOL 4 などの文法を LR(k) アナライザ II に通してみた．これらの文法は，少しアクションルーチンを使えば，どれも，ほぼ SLR (1) 文法になることがわかった．この 1 つの例として，FORTRAN の場合について述べる．

4.1 文法の解析

FORTRAN 文法で，識別子を属性により区別して書いた場合（例えば，〈ARRAY NAME〉等）には，この文法は，ほぼ SLR(1) 文法となり，特に問題はなかった．この場合，シラブルリーダが，識別子の属性別に token を返す必要は必ずしもない．parser に ‘check’ というオペレータを持たせておいて，構文解析時に，属性を調べるようにしてもよい．FORTRAN 文法で，SLR(1) 文法の範囲から，はみ出ている所は，意味情報を使わなければならない 2〜3 箇所を除いては，たった 1 箇所だけであった．それは，入出力並びの〈DO 形仕様〉での制御変数と，〈入出力並び要素〉

としての変数とを区別する所であるが，この箇所もLALR(1) 文法の範囲には，入っていた．

次に，識別子を属性により区別しない文法を入力してみた．今度は，〈算術一次子〉か〈論理一次子〉か区別できない箇所が生じた．このことは，初めから予想されていたが，どのような所が，LALR(1) 文法の範囲に入らないかを調べるために行なってみた．

構文解析オートマトンでの状態によって，変数を，どちらの〈一次子〉に還元するべきかが判定できる場合が，かなりあった．状態により区別できないときは，入力記号の先読みを行なうが，先読み記号が演算子であった場合には，演算子の種類により区別できる．先読み記号が，')' や ',' や，「文の終りを示す記号」であったときには，どちらの〈一次子〉に還元しても，次の入力記号を読込む前に，その変数は〈式〉にまで還元されてしまう．その間に，何らのセマンティクス処理やスタック操作も行なわれないので，直接，〈式〉に対応するルック・バック状態に遷移してしまえばよい．

一般に，宣言情報を使わないときには，LR(3) 文法の範囲に入らない所が生じるが，このとき上で説明したような状況になっていることが多い．したがって，1文字の先読みで解決できることが多いが，意味情報を用いた方が効率のよいときもある．これらの利害得失を計るために，LR(k) アナライザ II に，文法を少しずつ変えて入力してみると，見通しがよくなる．

結果として，FORTRAN の文法は，ほぼ SLR(1) 文法となり，LR(k) 構文解析手法が適用しやすい言語であることがわかった．

4.2 構文解析表

LR(k) アナライザ II の出力した遷移表より，構文解析表を作成した．この表のデータ構造は，ほぼ DeRemer の方式に従っていて，状態表と遷移表からなっている．各表の1エントリーの構成を，Fig. 4 に示す．

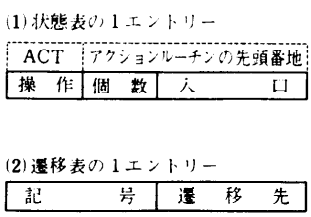

Fig. 4 Data structure of the parsing table

状態表の操作部には，その状態にて行なうオペレーションが入り，read, stack and read, look-ahead, look-back, pop-up and output, check 等がある．個数部には，その状態から出ている遷移の個数またはスタックのポップ・アップ数が入る．入口部には遷移表のエントリーを指すポインターが入る．アクションルーチンを実行する状態では，Fig. 4 で点線で囲ったように，オペレータ ACT と，そのアクションルーチンの先頭番地が付加される．なお，状態表の先頭からの相対番地が，状態番号として使われる．

一方，遷移表の記号部には，パターンマッチされる記号 (token の種類または状態番号) か，あるいは出力記号が入る．

このようなデータ構造を採用して構文解析表を作成したときの表の大きさを，Table 7 に示す．

Table 7 Size of FORTRAN parsing tables

水 準	表の大きさ(バイト)
3000	858
5000	1364
7000	1416

5. FORTRAN コンパイラへの適用

われわれは，LR(k) アナライザを用いて，FORTRAN の parser を作り，それをもとにして，超小型計算機 HITAC 10 で FORTRAN コンパイラを作成した．このコンパイラは，1パスであり，われわれは4K語用と8K語用（1語 16 ビット，1K語＝1024 語）の2つを作った．シラブルリーダを単純にさせるため，キー語は予約語とし，空白は区切り記号とした．

4K 語用の方は，JIS の水準 3000 から，COMMON 文，文関数定義文，EQUIVALENCE 文，補助入出力文を除いた言語仕様であり，目的プログラムは，ポリッシュリストの形をしている．一方，8K 語用の方は，ほぼ JIS の水準 3000 の言語仕様で，目的プログラムは機械語である．この他に，LR(k) parser を用いて，JIS 水準 5000 の FORTRAN コンパイラを，小型計算機用に設計した．以下，LR(k) アナライザの適用について，HITAC 10 (4K 語用) で行なった方式を述べる．

LR(k) アナライザを用いて parser を完成させるまでには，次の手順を踏む．まず FORTRAN の文法をバッカス記法で記述する．このとき，何を終端記号と

するかが問題になる．文法のすべてをバッカス記法で書けば，終端記号は，英字，数字，区切り記号などになるが，そのままで LR(k)-parser を作ったのでは効率がよくないので，ソースプログラムを単語分けする文字列解析ルーチン (lexical analyzer) があるものと想定して，このルーチンからの出力を終端記号として扱うようにする．

HITAC 10 の FORTRAN の場合は，コンパイラを簡単にするため添字式に一般式が書けるという文法の拡張を行なった．この FORTRAN は，90 個の構文規則で記述できた．LR(1) 文法は，広い範囲の言語を覆うので，構文規則を定義するとき，文法上の制限を気にすることなく記述できるので，構文規則数が少なくて済む．

次に，各ステートメントのオブジェクトコードを設計して，どの状態で何を出力するかを決め，バッカス記法で記述した構文規則の対応する部分に書き加える．さらに，意味処理を考慮して，必要になるアクションルーチンの名前も，構文規則中に挿入する．Fig. 1 の文法Gでは，〈型宣言文〉の中に，実数型，整数型のタイプをセットするアクションルーチン名が挿入されている．

このようにして得られた文法を LR(k) アナライザに入力する．(入力媒体は，カードまたはディスクファイルである．) 能率のよい parser を作るために，コンパイラ設計者は，出力結果をもとにして遷移図を描いてみる．parser を遷移図の形で表わすことは，全体を把握しやすくし，アクションルーチンの実行時点や目的コードを出力する時点が正しいかなどの判断を行ないやすくする．また意味処理を考慮した状態の統合などの最適化を行ないやすくする．例えば，'.EQ.', '.GT.', '.LT.'… などをまとめて関係演算子という 1 つの終端記号にすれば，パターンマッチの回数を減らすことができる．シラブルリーダでは，関係演算子というクラスと，'.EQ.', '.GT.' というオペレータの種類を示すインデックスとを返すようにする．

こうして，入力データを修正し，再度 LR(k) アナライザにかけて調べることを繰返し，さらに人間の手でも parser の最適化を行なう．HITAC 10 の FORTRAN の場合には，この繰り返しを 4 回行なった．はじめは LR(k) アナライザの最適化フェイズを通さずに行なった．そのとき parser の大きさは 631 語であったが最適化フェイズを通した結果 353 語になり，人手により，最終的に 280 語の表にまとまった．

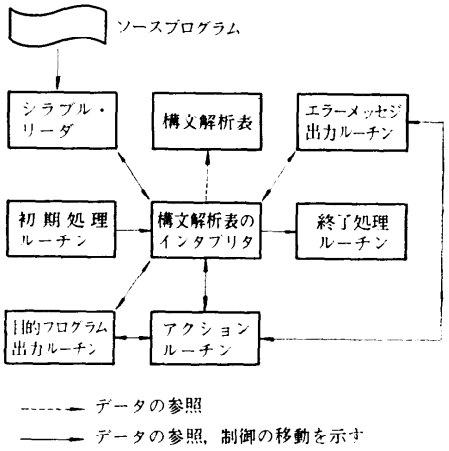

Fig. 5 Structure of the HITAC 10 FORTRAN compiler

LR(k)-parser の表に，Fig. 5 で示した各ルーチンを追加して，コンパイラを構成した．

HITAC 10 での FORTRAN の構文解析表は，Fig. 4 で示したデータ構造と僅かに異なっている．操作部の先頭 1 ビットを，アクションルーチンを実行する状態であるか否かを示すフラグとして使い，フラグが 1 のときは，すぐ次の 1 語に，アクションルーチンの先頭番地が入っている．

アクションルーチンは，アセンブラ語で記述されていて，2 ステップから 50 ステップまでの小さなルーチンであり，約 20 個ある．これらのルーチンでは，例えば，次のような処理を行なう．

(1) DO ループの端末文かどうか判定する．

(2) 識別子のタイプをシンボルテーブルに登録する．

(3) 引数の個数を数える．

識別子の属性を調べるアクションルーチンの使用頻度が大きかったので，われわれは，'check' というオペレータを設けて，構文解析表の操作部で扱うようにした．

エラーチェックは，構文解析表を解釈実行するときに行なうものと，アクションルーチンで行なうものとがある．前者では，入力記号と，遷移表の記号部とのパターンマッチングで一致がとれなかったときに構文エラーを検出し，後者では，主としてセマンティクス・エラーのチェックを行なう．

エラーの起ったあとの処理は，HITAC 10 の場合，

スタック（状態スタックしか使わない）を，文の始めまで，ポップ・アップし，ソースステートメントの残りの部分を読み飛ばしているが，メモリに余裕があれば，つぎの区切り記号の‘,’や‘)’まで読み飛ばして，その区切り記号がくることによって還元されるものを求め，スタックをポップ・アップするという細かい処理を行なうこともできる．

さらに，入出力ルーチン，ライブラリ関数，実行時ルーチンなどを加えて，コア 4K 語用の1パス常駐 FORTRAN コンパイラを実現した．コンパイラの大きさは，1.4K 語，実行時ルーチンは 800 語，ユーザエリアは 600 語である．また，コンパイラの中の，アクションルーチンは 279 語，構文解析表のインタプリタは，116 語で作成できた．

このコンパイラの1ステートメント当りのコンパイル速度の平均は，19 ms であった．

6. むすび

コンパイラの CAD システムをめざして，parser を表の形で自動作成するシステム LR(k) アナライザ II を開発し，これを用いて，実用的な FORTRAN コンパイラを短期間で作成することができた．入力文法中で，必要な情報をすべて与えて，parser 全体を生成させることが目標であるが，今回は，オブジェクトの形やアクションルーチン名を指定できるようにするところまで行なった．今後，構文エラーに関する何らかの形式的な記述も導入して，エラー処理やエラー回復ルーチンまで考慮して行きたい．また，コンパイラの設計・作成に加え，評価も行えるようなシステムにまとめて行くつもりである．

参考文献

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.
4) 加藤，中田，小島：LR(k) アナライザとその FORTRAN ミニ・コンパイラへの適用，昭 47 情報処理学会大会，講演番号 24.
5) 林 達也：CFG-PL 変換について，情報処理，Vol. 12, No. 3 (1971).
6) 関本彰次：LR(1) 文法の諸性質とそれに基づく parser の構成法，情報処理，Vol. 13, No. 3 (1972).

(昭和 48 年 9 月 8 日受付)
(昭和 48 年 11 月 1 日再受付)