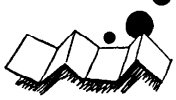
解　説

● 構文解析法からみた最近の計算機言語理論の動向†



稲垣康善††　大山口通夫†††

1. まえがき

近年，コンパイラ自動生成系の実現に関する多くの論文が発表され，コンパイラ自動生成系によるコンパイラ作りが広く行われるようになった[8),9),15),16)]．コンパイラ自動生成系は，コンパイラ作成に必要な労力を軽減する道具（tool）の意味で用いられることが多いが，どのような入力仕様を採用するかあるいはどの程度の品質または規模のコンパイラを生成するかによって，さまざまなものが考えられる．いずれにせよ大まかに言えば，コンパイラは，単語(語彙)解析部，構文解析部，意味解析部及びコード生成部から構成されている．したがって，コンパイラ自動生成系は入力仕様からこれらの部分を自動生成できることが望ましい．現在，実現されているコンパイラ自動生成系では，単語解析部及び構文解析部の自動生成に関して，ほぼ満足のいくものができ上がっているが，意味解析部及びコード生成部の自動生成に関しては，今後の研究に残されているものが少なくない．現在，コード生成部の自動生成に関する研究が盛んに行われている[10),11)]．構文解析部は，コンパイラの最も基本的な要素の1つであり，またその自動生成に関して理論的成果が果たした役割の大きい部分でもある．さらに構文解析部の誤り回復機能はコンパイラ作成において非常に重要であり，その機能の自動生成あるいはその系統的な構成法に関して，現在，盛んに研究されている[6)~8),15),21)~25)]．

以上のことから，本小文では構文解析法を中心としたコンパイラ作成に関する最近の話題について解説する．すなわち，2章では，従来の構文解析法について，3章では拡張文脈自由文法のパーザ（構文解析機械）である ELL パーザ及び ELR パーザについて，4章ではパーザの誤り回復について解説する．

† Current trends in the theory of formal languages and parsing algorithms by Yasuyoshi INAGAKI (Department of Electrical Engineering, Faculty of Engineering, Nagoya University) and Michio OYAMAGUCHI (Department of Electronics, Faculty of Engineering, Mie University).

†† 名古屋大学工学部電気工学科
††† 三重大学工学部電子工学科

2. 構文解析

プログラミング言語の文法は，通常，文脈自由文法(BNF 記法）で記述されるが，実用的な観点からその文法に要請される条件として以下のものが上げられる．

（i）　文法があいまいでないこと††††
（ii）　速い構文解析が可能であること
（iii）　構文解析に必要なメモリ量が小さいこと
（iv）　パーザの実現が容易であること

このような条件を満たすもので，現在良く知られている代表的な文法のクラスに，LL 文法並びに LR 文法がある．これらの文法のパーザ（構文解析機械）は長さ n の入力に対して，n に比例した時間とメモリ量で構文木を出力することができる[1)-4)]．LL 文法及び LR 文法のパーザは表駆動型のものが良く知られており，これは文法から自動的に構成できることから，コンパイラ自動生成系で生成されるコンパイラの構文解析部としてよく用いられている．以下では LL 文法，LR 文法及びこれらの表駆動型のパーザについて解説する．

文脈自由文法 G は $G=(V_N, V_T, P, S)$ の4つ組で定義される．ここで，V_N，V_T はそれぞれ非終端記号，終端記号の有限集合であり，P は生成規則の集合で $V_N\times(V_N\cup V_T)^*$ の部分集合であり，また，$S\in V_N$ は開始記号である．$(A,\alpha)\in P$ を通常 $A\to\alpha$ と表現する．ただし，$A\in V_N$，$\alpha\in(V_N\cup V_T)^*$．$A\to\alpha\in P$ ならば $\beta A\gamma\Rightarrow\beta\alpha\gamma$ であるとして関係$\Rightarrow$を定義する．ただし，$\beta,\gamma\in(V_N\cup V_T)^*$．ここで，$\beta A\gamma\Rightarrow\beta\alpha\gamma$ を導出とよび，さらに $\beta\in V_T^*$ の場合，最左導出とよび，$\gamma\in V_T^*$ の場合，最右導出とよぶ．$\overset{*}{\Rightarrow}$を$\Rightarrow$の反射的推移閉包とすると，文法Gによって生成される文の集合は $\{w\mid S\overset{*}{\Rightarrow}w, w\in V_T^*\}$ である．以後，記号 A，B，…を非終端記号，a，b，c，…を終端記号，x，y，z 等を終端記号列，α，β，γ 等を $(V_N\cup V_T)^*$ の元，及び X，X_i を $(V_N\cup V_T)$ の元として用いる．

†††† あいまいな文法であっても，別の条件の付加（たとえば，生成規則に優先順位の付加）によって，1つの構文木を選択する方法もある[8)]．

次の　**例1**の文脈自由文法を考える．この例ではPの各生成規則に自然数を割り当て，各規則を対応する自然数で参照する．入力列$a\ b\ d\ c$の構文木は**図-1**のようになる．

例 1　文脈自由文法の例

$G=(V_N=\{S, B, C\}, V_T=\{a, b, c, d\}, P, S)$

$P=\{1: S\rightarrow aBC,$

$2: B\rightarrow bBd,\ 3: B\rightarrow\varepsilon^{\dagger},\ 4: C\rightarrow c\}$

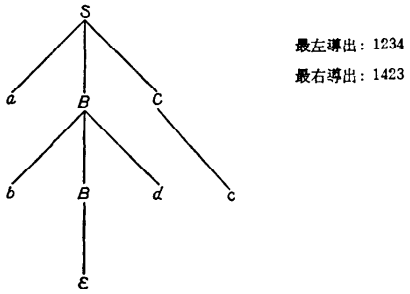


図-1　入力列$a\ b\ d\ c$の構文木

図-1の構文木は，その最左導出である生成規則列1234，またはその最右導出である生成規則列1423によって一意に決まることから，パーザは構文木のかわりにそのような生成規則列を出力する．表駆動型のLLパーザでは最左導出を出力し，LRパーザでは，最右導出の鏡像††を出力する．

通常，木の構成方法との対応から，LLパーザは下降型の構文解析法，LRパーザは上昇型の構文解析法とよばれている．これらのパーザは入力列を左から右に後戻りせず走査することによって，構文解析結果を出力するプッシュダウン変換機械（**図-2**）とみなせるが，その各時点の動作はプッシュダウンスタックのトップ記号，長さkの入力記号列（先読み記号列）及び機械の内部状態によって決定される．ここでkの値は実用的な観点から，通常，$k=1$または0の値が用いられる．したがって文法からそのパーザを自動生成することはプッシュダウン変換機械，すなわちその動作表とそのドライバを文法から自動生成することと言えよう．以下では，パーザの自動生成に関してもう少し詳しく説明する．

2.1 LL パーザ

ここでは，先読み記号列の長さ$k=1$でLL構文解析ができるLL(1)文法を考え，そのパーザの構成方法

† 空語（長さ0の語）をεで表わす．

†† 生成規則列1423の鏡像は3241である．

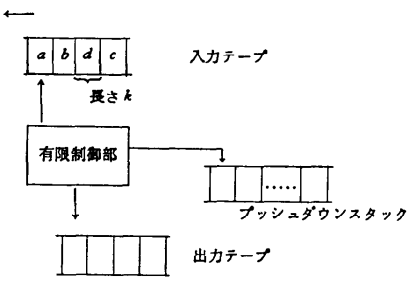


図-2　プッシュダウン変換機械

について説明する．LL(1)文法 $G=(V_N, V_T, P, S)$ とは次の条件（1）を満たす文脈自由文法である．

条件（1）：開始記号Sからの2つの最左導出

$$S\overset{*}{\Rightarrow}xA\alpha\Rightarrow x\beta\alpha\overset{*}{\Rightarrow}xy\quad\text{及び}$$

$$S\overset{*}{\Rightarrow}xA\alpha\Rightarrow x\gamma\alpha\overset{*}{\Rightarrow}xz$$

が存在し，$\mathrm{FIRST}_1(y)=\mathrm{FIRST}_1(z)$ ならば $\beta=\gamma$ が成立する．ここで $\mathrm{FIRST}_1(y)$ は記号列yの最初の（一番左の）記号を示す†．

条件（1）を満たす LL (1)文法 $G=(V_N, V_T, P, S)$ は次の条件（2）をも満たすことが知られている．

条件（2）：$A\rightarrow\alpha$ と $A\rightarrow\beta$ とが別の生成規則であるとき，

$$\mathrm{FIRST}_1(\alpha\mathrm{FOLLOW}_1(A))\cap$$
$$\mathrm{FIRST}_1(\beta\mathrm{FOLLOW}_1(A))=\phi$$

が成立する．ここで，

$$\mathrm{FOLLOW}_1(A)=\{a\mid S\overset{*}{\Rightarrow}\gamma Aa'\ \text{及び}$$
$$a\in\mathrm{FIRST}_1(a')\}\ ^{\dagger\dagger}$$

条件（2）の添字1をkにかえた条件を満たす文法は強 LL(k)文法と言われ，$k=1$の場合のみ，“LL(1)文法ならば強 LL(1)文法である”，が成立する．LL(1)文法のパーザは，1状態のプッシュダウン変換機械であり，その変換機械の時点表示を（入力列，プッシュダウン記号列，出力列）の3つ組で表わすとき，最初の時点表示は (xy, S, ε) である．ここで xy はパーザへの入力列であり，Sは文法の開始記号である．パーザがxを読み終ったときの時点表示は (y, α, π) である．ただし，πは $S\overset{*}{\Rightarrow}x\alpha$ の最左導出である．

パーザの次の動作はスタックのトップ記号（すなわちαの最も左の記号）Xが V_N の元の場合，条件（2）より，$y=ay'$とすると，$a\in\mathrm{FIRST}_1(\beta\mathrm{FOLLOW}_1(X))$をみたす生成規則 $i: X\rightarrow\beta$ を適用する．すなわち，

† $y=\varepsilon$ の場合，$\mathrm{FIRST}_1(y)=\varepsilon$である．定義域を非終端記号を含んだ系列に拡張すると，$\mathrm{FIRST}_1(\alpha)=\{\mathrm{FIRST}_1(x)\mid\alpha\overset{*}{\Rightarrow}x\in V_T^*\}$ となる．さらに定義域は記号列の集合に自然に拡張される．

†† 定義域は記号列または記号列の集合に自然に拡張される．

時点表示は $(y, \beta\alpha', \pi i)$ となる．ここで，$\alpha = X\alpha'$．$\mathrm{FIRST}_1(\beta\mathrm{FOLLOW}_1(X))$ を規則 $X \to \beta$ の先読み記号集合という．スタックのトップ記号 X が V_T の元の場合，入力列 y が X で始まる（すなわち $y = Xy'$）ならば単にスタックのトップ記号を取り去る．すなわち時点表示は (y', α', π) となる．このように LL(1) 文法のパーザを構成するためには，すべての非終端記号 X と入力記号 a に対する動作を示す表，すなわち，どの生成規則を適用するかの動作表を構成すればよいことになる．また，その動作表は各規則 $X \to \beta$ に対してその先読み記号集合を計算することにより求まる．

以上のことから，LL(1)パーザが自動生成できることは容易にわかるであろう．ところで，FIRST_1 及び FOLLOW_1 の計算方法についてであるが，それを効率良く求める方法の1つに，それをブール行列の閉包 (transitive closure) を求める問題に還元して計算する方法がある[4]．したがって，実現の容易な Warshall[27] のアルゴリズムを用いると，n^3 に比例した時間で計算できる．ここで n は与えられた文法のサイズである．さらに Hunt[III] らのスパース (sparse) 行列†の閉包を求めるアルゴリズムを用いると，n^2 に比例した時間で計算できる[14]．

LL(1)パーザの例として，**例1**の文法Gのパーザを考える．文法Gの生成規則の先読み記号集合を**図-3**に，そのパーザの動作表を**図-4**に示す．

2.2 LR パーザ

ここではLR(1)文法のパーザの構成方法についてまず説明し，次に LALR(1)文法及び SLR(1)文法について述べる．LR(1)文法 $G=(V_N, V_T, P, S)$ とは次の条件(3)を満たす文脈自由文法である．

条件(3)：Gの拡大文法 $G'=(V_N \cup \{S'\}, V_T, P \cup \{S' \to S\}, S')$ の開始記号 S' からの2つの最右導出

$$S' \overset{*}{\Rightarrow} \alpha A x \Rightarrow \alpha\beta x \quad \text{及び}$$
$$S' \overset{*}{\Rightarrow} \gamma B y \Rightarrow \alpha\beta z$$

が存在し，$\mathrm{FIRST}_1(x) = \mathrm{FIRST}_1(z)$ ならば $\alpha=\gamma$，$A=B$ 及び $y=z$ が成立する．

LR(1)パーザの構成に基本となるものはLR(1)項の集合という概念である．生成規則 $A \to \alpha\beta$ と入力記号 a に対して，対 $[A \to \alpha \cdot \beta, a]$ は LR(1)項と呼ばれる．パーザの各状態は LR(1)項のある集合に対応し，その初期状態は項 $[S' \to \cdot S, \varepsilon]$ の閉包に対応する．ここで閉包とは，たとえば $S \to \gamma$ が生成規則であれば，

† スパース行列とは値が0でない配列要素の個数が行列の次元数 n に比例している場合をいう．

$S \to aBC$: $\mathrm{FIRST}_1(aBC) = \{a\}$
$B \to bBd$: $\mathrm{FIRST}_1(bBd) = \{b\}$
$B \to \varepsilon$: $\mathrm{FIRST}_1(\{\varepsilon\}\mathrm{FOLLOW}_1(B)) = \{c, d\}$
$C \to c$: $\mathrm{FIRST}_1(c) = \{c\}$

図-3　例1の文法Gの生成規則の先読み記号集合

V_N \ V_T	a	b	c	d
S	$S \to aBC$			
B		$B \to bBd$	$B \to \varepsilon$	$B \to \varepsilon$
C			$C \to c$	

図-4　文法Gの LL(1) パーザの動作表

項 $[S \to \cdot\gamma, \varepsilon]$ を $[S' \to \cdot S, \varepsilon]$ からなる集合に加えることである．形式的に定義すると項の集合 $\mathcal{S}$ の閉包とは次の条件(4)をみたす最小の項の集合 $\mathcal{S}'$ のことである．

条件(4)：(i) $\mathcal{S} \subseteq \mathcal{S}'$ 及び (ii) $[A \to \alpha \cdot B\beta, a] \in \mathcal{S}'$ であり，$B \to \gamma$ が生成規則ならば，$[B \to \cdot\gamma, b] \in \mathcal{S}'$ である．ここで $b \in \mathrm{FIRST}_1(\beta a)$.

項 $[S' \to \cdot S, \varepsilon]$ の閉包を $\mathcal{S}_0$ とすると，パーザの他の状態に対応する項の集合は次の関数 GOTO によって計算されるものである．記号 X に対して $\mathrm{GOTO}(\mathcal{S}_0, X)$ は次のように定義される．$[B \to \alpha \cdot X\beta, a] \in \mathcal{S}_0$ ならば $[B \to \alpha X \cdot \beta, a] \in \mathcal{S}$ と定める．項の集合 $\mathcal{S}$ の閉包が $\mathrm{GOTO}(\mathcal{S}_0, X)$ である．定義域を拡張して，記号列 αX に対して，$\mathrm{GOTO}(\mathcal{S}_0, \alpha X) = \mathrm{GOTO}(\mathrm{GOTO}(\mathcal{S}_0, \alpha), X)$ と定義する．ここで $\mathrm{GOTO}(\mathcal{S}_0, \varepsilon) = \mathcal{S}_0$ とする．すべての記号列 α に対して，$\mathrm{GOTO}(\mathcal{S}_0, \alpha)$ の種類は有限であり，したがってパーザの状態数も有限となる．

パーザのプッシュダウンスタック記号列は，通常，$\mathcal{S}_0 X_1 \mathcal{S}_1 \cdots X_n \mathcal{S}_{n-1} X_n \mathcal{S}_n$ と表わされる．ここで，$\mathcal{S}_i$ はパーザの状態（すなわち，項の集合）であり，$\mathcal{S}_i = \mathrm{GOTO}(\mathcal{S}_0, X_1 \cdots X_i)$ が成立する．この記法は冗長であることから，単にスタック列を $X_1 \cdots X_n$ と書き，パーザの内部状態を $\mathcal{S}_n$ と書く記法もある．ここでは後者の記法を採用して，パーザの時点表示を（スタック記号列，状態，入力列，出力列）と表わすと，最初の時点表示は $(\varepsilon, \mathcal{S}_0, x_1 x_2 \cdots x_n, \varepsilon)$ である．入力 $x_1 \cdots x_i$ を読み終った時点で，その時点表示を $(X_1 \cdots X_j, \mathcal{S}, x_{i+1} \cdots x_n, \pi)$ とすると，その次の動作は状態 $\mathcal{S}$ と入力 x_{i+1} に依存して以下のいくつかの場合が考えられる．

(1) $[A \to \alpha \cdot, x_{i+1}]$ が $\mathcal{S}$ の元で，$A \to \alpha$ が $S' \to S$ でないなら，$X_{j-\ell} X_{j-\ell+1} \cdots X_j = \alpha$ とすると，時点表示は $(X_1 \cdots X_{j-\ell-1} A, \mathcal{S}', x_{i+1} \cdots x_n, \pi m)$ となる．ここで $\mathcal{S}' = \mathrm{GOTO}(\mathcal{S}_0, X_1 \cdots X_{j-\ell-1} A)$ であり $A \to \alpha$ は m 番目

の生成規則とする．この動作を還元という．

（2） $[A\to\alpha\cdot x_{i+1}\beta, b]$ が $\mathcal{S}$ の元ならば，時点表示は $(X_1\cdots X_j x_{i+1}, \mathcal{S}', X_{i+2}\cdots x_n, \pi)$ となる．ここで $\mathcal{S}'=\mathrm{GOTO}(\mathcal{S}_0, X_1\cdots X_j x_{i+1})$．この動作をシフトという†．

（3） $[S'\to S\cdot, \varepsilon]$ が $\mathcal{S}$ の元で，$x_{i+1}\cdots x_n=\varepsilon$ ならば，パーザは受理状態に入る．

（4） （1）～（3）のいずれでもない場合，パーザは誤りを検出した状態に入る．

LR(1)文法の場合，上記（1）～（3）の動作において，2つ以上の異なった動作は同時に起こらないことが保証されている††．

以上より，LR パーザの構成はパーザの状態と入力記号に対する動作表を作ればよく，またその動作表も上記の方法で自動生成できる．

LR(1)パーザの欠点は，実用的な観点からみて，その動作表が非常に多くのメモリ量を必要とすることである．この理由から，LR(1)文法の部分クラスである LALR(1)文法及び SLR(1)文法のパーザが用いられる場合が多い．たとえば YACC システム[8]は LALR(1)文法のパーザ生成機械である†††．以下では LALR(1)文法及び SLR 文法について述べる．

LALR(1)文法は以下のように定義される．LR(1)文法から構成されたパーザの動作表において，各状態を項の第一要素（たとえば，項 $[A\to\alpha\cdot\beta, a]$ の第一要素は $A\to\alpha\cdot\beta$ である）のみに着目して，同じ集合とみなせる状態を同一視する．このようにしてできた動作表において衝突が生じないなら，その文法を LALR(1)文法という．このように LALR(1)パーザの動作表のサイズは LR(1)パーザの動作表に較べて，かなり小さくなることが予想される．LALR(1)パーザの動作表は，もちろん，定義のように LR(1)パーザの動作表をまず構成してそれから求めることができるが，それは効率の良い方法とはいえない．効率を考慮した他の方法がいくつか提案されているが[2),12)]，まだ決定打はないようである．

SLR(1)文法は以下のように定義される．項の第二要素を無視する以外は LR(1)文法からパーザの状態集合を構成したのと同じ方法で，状態集合を構成する．次に，各状態の各項 $A\to\alpha\cdot\beta$ すべてを $\{[A\to\alpha\cdot\beta, a]\mid a\in\mathrm{FOLLOW}_1(A)\}$ で置き換えた状態を考え，さきと同じように動作表を構成する．このようにしてできた動作表において衝突が生じないなら，その文法を SLR(1)文法という．SLR(1)文法のパーザは実現が容易であり，パーザの動作表が小さいという特色をもつ†．しかしながら，SLR(1)文法のクラスは LALR(1)の文法のクラスに真に含まれる．

すなわち，LALR(1)パーザは構成できるが，SLR(1)パーザを構成できない文法が知られている．ついでながら，従来，LL(1)文法は LALR(1)及び SLR(1)文法でもあるといわれてきたが［文献1)の問題7，4，6］，最近この反例が示された．すなわち，LL(1)ではあるが LALR(1)でない文法の存在が示された[13)]．ただし，“LL(1)文法は LR(1)文法でもある”は真である[1)]．

SLR(1)パーザの例として**例1**の文法 G のパーザを考える．

パーザの状態及び GOTO 関数を**図-5**に，そのパーザの動作表を**図-6**に示す．

$\mathcal{S}_0$: $\{S'\to\cdot S,\ S\to\cdot aBC\}$　$\mathcal{S}_1$: $\{S'\to S\cdot\}$

$\mathcal{S}_2$: $\{S\to a\cdot BC,\ B\to\cdot bBd,\ B\to\cdot\}$　$\mathcal{S}_3$: $\{S\to aB\cdot C,\ C\to\cdot c\}$　$\mathcal{S}_4$: $\{B\to b\cdot Bd,\ B\to\cdot bBd,\ B\to\cdot\}$

$\mathcal{S}_5$: $\{S\to aBC\cdot\}$　$\mathcal{S}_6$: $\{C\to c\cdot\}$

$\mathcal{S}_7$: $\{B\to bB\cdot d\}$　$\mathcal{S}_8$: $\{B\to bBd\cdot\}$

（ただし，$\mathrm{FOLLOW}_1(S')=\mathrm{FOLLOW}_1(S)=\mathrm{FOLLOW}_1(C)=\{\varepsilon\}$
$\mathrm{FOLLOW}_1(B)=\{c, d\}$）

$\mathrm{GOTO}(\mathcal{S}_0, S)=\mathcal{S}_1$　$\mathrm{GOTO}(\mathcal{S}_0, a)=\mathcal{S}_2$
$\mathrm{GOTO}(\mathcal{S}_2, B)=\mathcal{S}_3$　$\mathrm{GOTO}(\mathcal{S}_2, b)=\mathcal{S}_4$
$\mathrm{GOTO}(\mathcal{S}_3, C)=\mathcal{S}_5$　$\mathrm{GOTO}(\mathcal{S}_3, c)=\mathcal{S}_6$
$\mathrm{GOTO}(\mathcal{S}_4, b)=\mathcal{S}_4$　$\mathrm{GOTO}(\mathcal{S}_4, B)=\mathcal{S}_7$
$\mathrm{GOTO}(\mathcal{S}_7, d)=\mathcal{S}_8$

図-5 文法Gの SLR(1)パーザの状態及び GOTO 関数

	a	b	c	d	ε
$\mathcal{S}_0$	シフト				
$\mathcal{S}_1$					受 理
$\mathcal{S}_2$		シフト	還元3	還元3	
$\mathcal{S}_3$			シフト		
$\mathcal{S}_4$		シフト	還元3	還元3	
$\mathcal{S}_5$					還元1
$\mathcal{S}_6$					還元4
$\mathcal{S}_7$				シフト	
$\mathcal{S}_8$			還元2	還元2	

（空欄はエラー）

図-6 文法Gの SLR(1)パーザの動作表

† シフト動作の条件は文献2)から採用した．

†† （1）の2つの異なった動作が可能なら，還元-還元衝突（conflict），（1）と（2）の動作が可能なら，シフト-還元衝突といわれる．

††† 正確にいうと，YACC では文法規則及び演算子の優先順位を入力仕様に与えることにより，パーザの衝突を解消しているので，LALR（1）文法より広い文法のクラスのパーザを生成する．また，文法規則の還元動作時に行うコード生成ルーチンの記述を入力仕様に許していることから，単なるパーザ生成機械とはいえない[8)]．

† 著者らが調べたところによると，パーザの状態数は PASCAL で約400である．

最後に，誤り検出に関して，LR(1)パーザ，LALR(1)パーザ及び SLR(1)パーザを比較してみよう．LR(1)パーザでは，誤った入力記号に対して，瞬時にその誤りを検出できるのに対し，LALR(1)パーザ及び SLR(1)パーザではその誤りを検出するまでに，数回の還元動作を行ってしまう場合がある．同様に，LALR(1)パーザで瞬時に誤りを検出できるような場合でも，SLR(1)パーザではその誤り検出の遅れが予想される．この誤り検出の遅れは誤り回復の問題と密接に関連しており，したがって，このことはどのパーザを採用するかを決める際に考慮しなければならない評価基準の一つであると思われる．これと関連した問題が，LR パーザの動作表に要するメモリ量を節約するために，動作表を簡略化して表現する場合に起きる．すなわち，動作表を簡略化する幾つかの方法が提案されているが[21),28)]，多くの場合簡略化法の採用による誤り回復機能の劣化が避けられない．したがって，これに対して十分考慮する必要があろう．

以上で LR パーザについての概説を終えるが，最近号の ACM SIGPLAN Notices に LR パーザに関する文献集[29)]が載っていたので付言する．

3. 拡張文脈自由文法とそのパーザ

PASCAL 以来，構文図式による文法表現が普及し始めている[6),7)]．その理由は，構文図式の簡潔で理解しやすい点を考えれば当然であろう．構文図式は文脈自由文法の生成規則の右辺の記述に正規表現（あるいは非決定性有限オートマトン）を許したもの，すなわち拡張文脈自由文法 (ecfg) の図式表現である．構文図式から，パーザを自動生成する方法について最近盛んに研究されており[15)~20)]，この章ではこの自動生成について解説する．

まず始めに，ecfg の形式的な定義を与える．記号の有限集合Vの正規表現は帰納的に次のように定義される．

（1） ϕ，ε及びVの各元 a は正規表現である．これらはそれぞれ集合 ϕ(空集合)，$\{\varepsilon\}$ 及び $\{a\}$ を表わす．

（2） p 及び q が正規表現なら，$(p|q)$，(pq) 及び $(p)^*$ は正規表現である．これらはそれぞれ集合 P∪Q，PQ 及び P^* を表わす．ただし p 及び q はそれぞれ集合P及びQを表わすとする．

以後，正規表現 e の表わす集合を $L(e)$ で示すことにする．ecfg $G=(V_N, V_T, P, S)$ は，生成規則の集合Pの各元が $A\to\alpha$ (ただし，$A\in V_N$ 及び α は $(V_N\cup V_T)$ 上の正規表現）であることを除いて，通常の文脈自由文法と同じである．ここで，生成規則の右辺の記述として正規表現のかわりに，非決定性有限オートマトン (nfa) を用いることもできる．生成規則 $A\to e$ に対して，α が $L(e)$ の元であれば，導出 $\beta A\gamma\Rightarrow\beta\alpha\gamma$ を定義する．ここで $\beta,\gamma\in(V_N\cup V_T)^*$．$\overset{*}{\Rightarrow}$を⇒の反射的推移閉包とすると，文法Gによって生成される文の集合は $\{w|S\overset{*}{\Rightarrow}w, w\in V_T^*\}$ である†．

例 2 に正規表現を用いた ecfg，**例 3** に nfa を用いた ecfg を示す．**例 3** の nfa の記述において，オートマトンの状態を丸○，最終状態を二重丸◎及び初期状態を四角□で示す．

例 2　正規表現を用いて表わされた ecfg の例

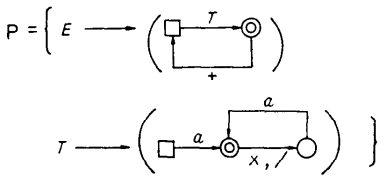
ecfg　$G_1=(V_N=\{E,T\}, V_T=\{a,+,\times,/\}, P, E)$

$P=\{E\to T(+T)^*,$

$T\to a((\times|/)a)^*\}$

例 3　非決定性有限オートマトンを用いて表わされた ecfg の例

ecfg　$G_1=(V_N, V_T, P, E)$



ecfgGの規則の右辺の正規表現（またはその部分表現） e に対して，$FOLLOW_1(e)$ 及び $FIRST_1(e)$ は **2** 章で定義したのと同様に定義される．ecfgGが ELL(1)文法であるとは，Gの各規則 $A\to e$ に対して，次の条件(5)を満たすときである．

条件(5)：(i) e の部分表現 $e_1|e_2$ すべてに対して，

$FIRST_1(e_1FOLLOW_1(e_1))\cap$

$FIRST_1(e_2FOLLOW_1(e_2))=\phi$

及び，(ii) e の部分表現 e_1^* すべてに対して，

$FIRST_1(e_1)\cap FOLLOW_1(e_1^*)=\phi$

が成立する．

ELL(1)文法は **2** 章の LL(1)文法の自然な拡張であり，条件(5)より，**2** 章の LL(1)パーザと同様な下降型構文解析が可能となる．しかしながら，ELL(1)文法に対しては，再帰降下 (recursive descent) パーザを用いる場合が多いようである††．

ecfgG$=(V_N, V_T, P, S)$ が ELR(1)文法であるとは次

† ecfgGから，それと同じ集合を受理する通常の文脈自由文法を構成することができる．

†† 再帰降下パーザについては **4.1** 節で解説する．

の条件(6)を満たすときである.

条件(6): Gの拡大文法 $G'=(V_N \cup \{S'\}, V_T, P \cup \{S' \to S\}, S')$ の開始記号からの2つの最右導出

$S' \overset{*}{\Rightarrow} \alpha A x \Rightarrow \alpha\beta x$ 及び

$S' \overset{*}{\Rightarrow} \gamma B y \Rightarrow \alpha\beta z$

が存在し, $\mathrm{FIRST}_1(x)=\mathrm{FIRST}_1(z)$ ならば $\alpha=\gamma$, $A=B$ 及び $y=z$ が成立する†.

このように, ELR(1)文法は LR(1)文法の自然な拡張となっている. この定義は ELL(1)の定義と異なって, 生成規則の右辺の正規表現の構造に依存していないことに注意すべきである. たとえば, $\beta=abb$ として, 生成規則を $A \to (ab|a)(b)^*$ とするとき, 導出 $A \Rightarrow \beta$ の β は $(ab)(b)$ と $(a)(bb)$ の二通りに選ぶことができる. しかしながら, この定義では, 二通りに選べるかどうかを全く問題にしていない††.

ELR(1)文法のパーザを構成する方法は二通り知られており, その一つは ELR(1)文法から等価な LR(1)文法に変換し, LR(1)パーザを構成する方法であり[9], もう一つは ELR 文法から直接 ELR(1)パーザを構成する方法である[7),18]. 前者の方法では, 等価な LR(1)文法が新しい非終端記号を必要とし, 元の ELR(1)文法の構造と較べて, 不必要に複雑な文法構造となってしまう欠点があり, 後者の方が望ましい. 後者の方法に似た方法として, ELR(1)文法から, その文法と構造があまり異ならない等価な ELR(1)文法に変換して, 変換された ELR(1)文法のパーザを直接構成する方法がある[20]. ELR(1)文法から直接構成するパーザは2章の LR(1)文法のパーザと似た方法で構成される[18]. すなわち, LR(1)パーザの項に対応するものは ELR(1)文法の生成規則の右辺で用いられた nfa の状態及び入力記号の対であり, 項の集合をパーザの状態とみてその動作表を構成する手続きは非常に似かよっている. しかしながら, その構成では, 生成規則を認識したときの還元動作の部分は, 本質的に異なる. なぜなら, LR(1)パーザの還元動作は生成規則の右辺の長さがあらかじめ定まっているため, 単にその長さのスタック記号をポップするだけでよいが, ELR(1)パーザでは生成規則の右辺の長さがいくらでも大きくなるため, プッシュダウンスタックを後戻りして, 規則の右辺の左端を決定する動作が必要となるからである. したがってこの規則の右辺の左端を効率よく見つけることができる方法が問題となるが, ここではシフト操作の逆操作を行うパーザの状態を導入して解決している.

以上のように ELR(1)パーザの自動生成に関する研究が盛んに行われているが, まだ研究すべき問題が多く残されている. まず第一に, 任意の ELR(1)文法から効率の良いパーザを自動生成する簡単な方法を見つけることがあげられる. さらに, ELR(1)パーザ及び LR(1)パーザの比較検討, 並びに ELR(1)パーザを構文解析部にもつコンパイラの試作及びその評価, 検討等が今後の課題であろう.

4. 誤り回復

コンパイラは, 一度に, 構文誤りをできるだけ多く発見し, 適切な誤り診断を行わねばならない. したがって, 構文解析部において, プログラム中の構文誤りを発見したとき, 残りのプログラムの構文解析が続行できるように, 誤り回復の機能が必要となる. 構文の誤り回復に関する論文は非常に多く見受けられ, 現在も盛んに研究されているが, 従来, 良く知られている誤り回復の方法として最小のハミング距離を用いる方法, あるいはパニックモードを用いる方法等がある. 前者の場合, 元のプログラム全体に文字の挿入, 削除を適用して, その適用回数が最小である構文的に正しいプログラムに誤り回復する方法であるが, これは一般的に時間がかかりすぎて, あまり実用的でない. したがって, この方法を実際に採用する場合, 構文誤りを発見したとき, 一定の長さのプログラム部分に対象を限定して, その部分を, 他の部分を考慮せずにハミング距離最小の正しい部分と置き換える操作をくり返すといった局所的な誤り訂正法が用いられる[21]. しかしながら, 現在ではこのハミング距離を使った方法はあまり用いられず, ハミング距離ではなく, ある適当な評価関数(たとえば, 削除は1点, 挿入は2点というように割り当て, その合計がある一定値以下のもので置き換える. 実際は削除, 挿入の対象となるトークンの種類によって, 異なるコストを与える)を採用し, さらに文脈の情報も加味した誤り回復法が用いられている.

パニックモードを用いる方法では, 構文誤りを発見したとき, 残りのプログラム中に適当な特殊記号(たとえばセミコロン';'または'**end**')を見つけるまで, その間のプログラム部分を読み捨てることによって, 誤り回復を行う. これは実現が簡単であるため現在よ

† 条件(6)の $\mathrm{FIRST}_1(x)=\mathrm{FIRST}_1(z)$ を $\mathrm{FIRST}_k(x)=\mathrm{FIRST}_k(z)$ に置き換えれば, ELR(k) 文法の定義となる.

†† このような二通り以上に選べる正規表現の使用を許さない, ELR(1)文法の定義もある[18].

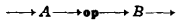
く用いられている方法であり，コンパイラ自動生成系で生成されるコンパイラにもこの類のものがしばしば採用されている．パニックモードを用いる方法では，読み捨てる部分が多くなる可能性があり（特に，自由書式のプログラミング言語においてそうである），したがってその部分の構文の誤りを検出できなくなるという欠点をもつ．それゆえ，読み捨てる部分をできるだけ少なくする方法が望まれるが，そのための改良案に関する論文がいくつか報告されている[6)~8),15),21)~25)]．

一般的に構文の誤り回復の問題は，プログラムを書いた人の意図についての何らかの仮定（たとえば，プログラマがミスをしやすい語句は何かといった仮定）を含み，さらにその正しい回復機能と処理速度のトレードオフを含む，非常に難かしい問題といえよう．現在，誤り回復に関していろいろな方法論が提案されかなりの成果が上げられているが，その決定打はまだないようである．しかしながら，構文規則のみをうまく利用して，ある程度成功している例がいくつか見受けられる．以下では，そのような誤り回復法を数例取り上げて解説する．

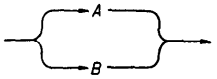
4.1 下降型構文解析法による誤り回復

パニックモードを用いる方法で，読み捨てる部分をできるだけ少なくする方法として，Wirth[6)]らによる下向き構文解析法で用いられたエラー回復の方法が上げられる．この方法では，エラー回復を行う場所として，文の終りなどを示す記号（';' または **'end'**）のほかにさらに他の記号をもエラー回復場所の発見に，積極的に利用する方法である．このような記号をエラー回復記号と呼ぶことにすると，たとえば，構文の解析途中で左括弧'('を見つけると，右括弧')' もエラー回復記号に選ぶことにより，読み捨ての部分を減らすことが可能となるわけである．以下では，この Wirth らの方法をうまく定式化して，その方法を実際に concurrent PASCAL コンパイラに採用した，Hartmann[7)]の方法について解説する．LL(1)文法または ELR(1)文法の再帰降下パーザは，各非終端記号に一つの手続きが対応し，これらの手続きが相互に呼び合う形で構成される[2),4)~7)]．この文法を構文図式で表現した場合，各規則は次の3つの型，すなわち連接，分岐及びループの組み合せで表現される．

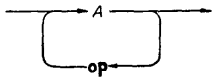
①連接 (sequence)



②分岐 (branch)



③ループ (loop)



ここで，AとBは非終端記号，**op** は終端記号である．各非終端記号に対応する手続きに，誤り回復記号の集合をパラメータで渡すことにすると，上記の3つの型の図式に対応するパーザの構成要素が自動的に求められる．すなわち，①及び②の型を例に取上げ PASCAL を用いて記述すると**図-7** 及び **8** の手続きとなる．図-7, 8の記述において，symbol は終端記号の数え上げ，**type** symbols=**set of** symbol，さらに次の3つの手続き get, error および check が仮定されている．get はプログラムから次の symbol を読む手続きであり，error および check はそれぞれ**図-9** および **10** の手続きである．

以上に述べた方法を具体的に説明するために，

① **procedure** sequence (keys: symbols)
begin
A (keys **or** [**op**] **or** B handles);
{B handles is $FIRST_1$(B)}
if nextsymbol=**op**
then get (nextsymbol)
else error (sequenceerror, keys **or** B handles);
B (keys)
end;

図-7 連接 (sequence) のプログラム

② **procedure** branch (keys: symbols);
begin
check (brancherror, keys **or** A handles **or** B handles);
if nextsymbol **in** A handles
then A (keys)
else
if nextsymbol **in** B handles
then B (keys)
else error (brancherror, keys)
end;

図-8 分岐 (branch) のプログラム

procedure error (reason: integer; recoverypoints: symbols);
begin give error message (reason);
while not (nextsymbol **in** recoverypoints)
do get (nextsymbol)
end;

図-9 error のプログラム

procedure check (errorno: integer; permissible: symbols);
begin if not (nextsymbol **in** permissible)
then error (errorno, permissible)
end;

図-10 check のプログラム

PASCAL の構文規則を例にとり上げる．たとえば①の例として，代入文

→ variable → := → expression →

があるが，① の A, **op**, B をそれぞれ variable, :=, expression で置き換えれば，代入文の手続きが得られる．

構文規則が，さらに複雑な形をしている場合は，その構文規則の手続きは，これらの基本要素から自然に導かれる．

Hartmann の誤り回復法は簡単に実現できるばかりでなく，機械的にプログラムに組み込むことができる点に特徴がある．したがって，パーザ生成機械がこのようなエラー回復機能を有するパーザを自動生成することも可能となるであろう．これと似た方法で，誤り回復を行う ELL(1)文法の再帰降下パーザの自動生成に関する論文としては，Lewi らの文献5)を参照されたい．

4.2 上昇型構文解析法による誤り回復

パニックモードによる誤り回復は上昇型構文解析法においてもよく用いられる方法である．その一例として，LALR パーザ生成機械である YACC システム[8]の方法がある．すなわち，YACC では誤り回復点を文法で指定できるように，予約語 error を採用している．たとえば，文法に規則 S→error ';' を追加すると，そのパーザは入力列の中に誤りを発見した場合，スタック中の S→·error の項を含む状態までスタックの内容をポップし，';'をみつけるまで入力列を読み進むことによって，あたかも S が認識されたかのようにして誤り回復を行う．

上昇型構文解析の誤り回復法には，パニックモードと異なるより強力な方法が Graham[23], Mickunas[24] らによって提案されている．それは入力列の中に誤りを発見した場合，読み飛ばしをせず，文字の挿入，削除による誤り回復を行う方法であり，その特色は誤りを発見したときハミング距離法と異なり，残りの入力列の情報とあらかじめ定めたコスト関数を利用して適切な誤り回復を行う点にある．以下では Graham らの単純順位パーザの誤り回復法を LR パーザの誤り回復法に拡張した Mickunas らの方法についてその概略を説明する．説明を簡単にするため，以下では評価関数に関する記述を削除する．

LR パーザが構文解析を行い，ある時点で誤りを検出したとする．すなわち，パーザがある状態 q におり，スタックの内容が α で，入力 a_1a_2……を読んだとき誤りを検出したとする．その時点表示を，

$\overset{\alpha}{\rightarrow}q\,?\,a_1a_2\cdots\cdots(1)$

と書くことにしよう．ここで，$\overset{\alpha}{\rightarrow}q$ はパーザの初期状態からシフト動作で q に至ることを示す．Mickunas らの方法では，まず文字の挿入による修復を考え，どのような文字を挿入したら良いかを決定するため，次のような方法を用いている．まず入力 a_1 でシフト動作をする状態集合 $\mathcal{S}(a_1)$ を考え，状態 q から $\mathcal{S}(a_1)$ のある元 p に到達可能な入力記号 y を挿入するわけであるが，そのような状態 p を選ぶために，入力 a_1a_2…… に対する状態 p からの動作を調べそこから得られる情報を利用する．すなわち，状態 p から入力 a_1a_2……の構文解析を続行したとき，新たな誤りを発見することなく，元の誤り発見点を含む還元動作に至るような状態 p を優先して選ぶという方法をとる．さらにそのような状態 p に対して，q から p に至る入力記号 y_p が存在するものを選ぶ．もしそのような入力 y_p が存在しない場合スタック列 α の最右記号 X (すなわち $\alpha=\alpha'X$) を誤り発見記号とみなす．すなわち(1)式のかわりに

$\overset{\alpha'}{\rightarrow}q'?Xa_1a_2\cdots\cdots(2)$

を考え，同じ操作をくり返す．ただし，入力 X でシフト動作する状態が，シフトした結果，さきに定義した状態集合 $\mathcal{S}(a_1)$ の元とならない（すなわち，$r\overset{X}{\rightarrow}p\in\mathcal{S}(a_1)$ となる状態 r が存在しない）場合は次のことをする．もし X が終端記号ならば X を削除して，$\overset{\alpha'}{\rightarrow}q'?a_1a_2$……を(1)のかわりに考える．$X$ が非終端記号ならば，X に還元された終端記号列を z とすると，z の最右記号の還元動作が行われる時点表示を(1)のかわりに考える．

この方法の例として，入力列……; X:=Y THEN GOTO L ELSE z:=1; ……を考える．もし THEN のところで誤りを発見したと仮定すると，時点表示は

…; ⟨id⟩:=⟨id⟩? THEN GOTO L…

となる．誤り回復を行う前に THEN 以降を構文解析すると，…; ⟨id⟩:=⟨id⟩? THEN ⟨statement⟩ ELSE ⟨statement⟩; ……となるであろう．上記の方法で誤り回復を行うと次のようになる．

; ⟨id⟩:=?⟨id⟩THEN…

; ⟨id⟩: ?=⟨id⟩THEN…

; ⟨id⟩?=⟨id⟩THEN…(: の削除)

; ?⟨id⟩=⟨id⟩THEN…

; IF⟨id⟩=⟨id⟩THEN…(IF の挿入)

以上のように Mickunas らの誤り回復は強力である．しかし，強力である反面，誤り回復のために費や

す時間が無視できず，あまり実用的でないという欠点をもつ．したがって，実用的な誤り回復手続きを開発する場合には，このような手法をどのようなもので代用して効率を上げるかということが問題となる．これに関して，一つの実用的な改良案が Graham[25] らによって提案されているので参照されたい．

5. あとがき

コンパイラ自動生成系によるコンパイラ作りが盛んになるに従い，改めて構文解析法に対する関心が高まり，またその研究も再び活発に行われるようになってきた．このことから，本稿ではコンパイラ自動生成系の実現の基礎となる構文解析法並びにその最近の話題について解説した．紙面の都合上，この分野の一部しか記述できなかったが，理論と実際の橋渡しになるような解説を心がけたつもりである．ご批判並びにご教示を頂ければ幸いである．

参考文献

(著書)

1) Aho, A. V., Ullman, J. D.: The Theory of parsing, Translation and Compiling, Vols. 1, 2, Prentice-Hall (1972, 1973).
2) Aho, A. V., Ullman, J. D.: Principles of Compiler Design, Addison-Wesley (1977).
3) Ullman, J. D.: Applications of Language Theory to Compiler Design, In Currents in The Theory of Computation, A. V. Aho, Ed., Prentice-Hall (守屋悦朗訳：最近の計算理論，近代科学社 (1976)).
4) Lewis II, P. M., Rosenkrantz, D. J., Stearns, R. E.: Compiler Design Theory, Addison-Wesley (1976).
5) Lewi, J., De Vlaminck, K., Huens, J., Huybrechts, M.: A Programming Methodology in Compiler Construction, part 1, North-Holland (1979).
6) Wirth, N.: Algorithms+Data Stractures=Programs, Prentice-Hall (1976). (片山卓也訳：アルゴリズム＋データ構造＝プログラム，日本コンピュータ協会 (1979)).
7) Hartmann A. C.: A Concurrent Pascal Compiler for Minicomputers, Springer, Lecture Notes in Computer Science, 50 (1977).

(論文)

8) Johnson, S. C.: YACC: Yet Another Compiler-Compiler, Computing Science Technical Report #32, Bell Laboratories (1978).
9) Aho, A. V.: Translator Writing Systems: Where Do They Now Stand?, IEEE Computer, August, 9-14 (1980).
10) Glansville, R. S., Graham, S. L.: A new method for compiler code generation, Fifth ACM Symp. Principles of Programming Languages, 231-240 (1978).
11) Graham, S. L.: Table-driven code generation, IEEE Computer, August, 25-34 (1980).
12) DeRemer, F., Pennello, T. J.: Efficient computation of LALR(1) look-ahead sets, Proc. ACM SIGPLAN Symp. Compiler Construction, 176-187 (1979).
13) Heibrunner, S.: A parsing automata approach to LR theory, Theoret. Comput. Science 15(2), 117-157 (1981).
14) Hunt III, H. B., Szymanski, T. G. and Ullman, J. D.: Operations on sparse relations, Comm. ACM 20, 171-176 (1977).
15) Lewi, J., De Vlaminck, K., Huens, J., Huybrechts, M.: The ELL(1) parser generator and the error recovery mechanism, Acta Informatica 10, 209-228 (1978).
16) Madsen, O. L., Kristensen, B. B.: LR-parsing of extended context free grammars, Acta Informatica 7, 61-73 (1976).
17) LaLonde, W. R.: Regular right part grammars and their parsers, Comm. ACM 20, 731-741 (1977).
18) LaLonde, W. R.: Constructing LR parsers for regular right part grammars, Acta Informatica 11, 177-193 (1979).
19) Heilbrunner, S.: On the definition of ELR(k) and ELL(k) grammars, Acta Informatica 11, 169-176 (1979).
20) Purdom, P. W., Brown, C. A.: Parsing extended LR(k) grammars, Acta Informatica 15, 115-127 (1981).
21) Lévy, J. P.: Automatic correction of syntax-errors in programming languages, Acta Informatica 4, 271-292 (1975).
22) Pemberton, S.: Comments on an error-recovery scheme by Hartmann, Software-practice and experience 10, 231-240 (1980).
23) Graham, S. L. and Rhodes, S. P.: Practical syntactic error recovery, Comm. ACM 18, 639-650 (1975).
24) Mickunas, M. D., Modry, J. A.: Automatic error recovery for LR parsers, Comm. ACM 21, 459-465 (1978).
25) Graham, S. L., Haley, C. B., Joy, W. N.: Practical LR error recovery, Proc. ACM SIGPLAN Symp. Compiler Construction, 168-175 (1975).
26) Sassa, M., Tokuda, J., Shinogi, T., Inoue, K.: Design and implementation of a multipass-compiler generator, Journal of Information Processing, Vol. 3, No. 2, 77-86 (1980).
27) Warshall, S.: A theorem on boolean matrices, J. ACM Vol. 9, No. 1, 11-12 (1962).
28) Aho, A. V., Johnson, S. C.: LR Parsing, Comput. Surv., Vol. 6, No. 2, 99-124 (1974).
29) Burgess, C., James, L.: An indexed bibliography for LR grammars and parsers, ACM SIGPLAN Notices, Vol. 16, 14-26 (1981).

(昭和56年11月2日受付)