左隅構文解析における局所後続列集合を導入した構文誤りの診断

武田 正之†

本論文ではボトムアップに構文解析（左隅構文解析）を行う際に発生する構文誤りについて考察し，さらにその構文誤りの診断方法（誤り原因の判定と回復方法）について論じている．誤りの診断には対象言語に依存した構文誤りの知識（誤りやすさのヒューリスティックス）を導入するだけで対処するのではなく，対象言語の生成規則と構文解析木を用いたより一般的な誤り診断の方法論を提案している．本手法は以下の特徴を持つ．(1)局所的な *FOLLOW* 集合（局所後続列集合）を定義し，構文木上の文脈を考慮した誤り診断を単純な機構で実現，(2)キーボードモデルを反映した記号列間距離を用いたつづり間違いの判定，(3)誤り発生位置の1つ前の入力記号が誤っている可能性の的確な指摘．Pascal におけるよくある構文誤りに対しては，本手法を用いた簡単な誤り原因判断機構でも十分有効であることが確認できた．

A Syntax Error Diagnosis with Local Follow Set for Left-Corner Parser

Masayuki Takeda†

This paper concerns syntax errors during left-corner bottom-up parsing, and presents a method of their diagnosis. We first define two practical concepts for syntax error diagnosis; local follow sets and similarity of terminal symbol sequences. Next, we propose simple but general diagnosis method for syntax errors based on the context of a parse tree. Finally some experimental results for language (Pascal)-oriented editor are shown and analyzed.

1. はじめに

本論文ではボトムアップに構文解析（左隅構文解析）を行う際に発生する構文誤りについて考察し，さらにその構文誤りの診断方法（誤り原因の判定と回復方法）について論じる．

構文解析方法は下降型と上昇型に大別できる．LL構文解析や再帰降下構文解析に代表される下降型では，左再帰規則によって構文解析系が無限ループに陥る危険性と，繰り返し計算による解析効率の低下という欠点がある．一方，LR 構文解析に代表される上昇型手法は扱える言語のクラスが十分大きく，実用的なプログラム言語をほとんど含んでいる．

上昇型と下降型を組み合せたような構文解析が決定的に行える文法クラスとして，文献1)では「左隅文法(Left-Corner Grammars)」LC(k) を提案している．

† 東京理科大学理工学部情報科学科
Department of Information Sciences, Faculty of Science & Technology, Science University of Tokyo

左隅構文解析法は上昇型構文解析法の一種であるが，同時に下降型構文解析法の特色も兼ね備えている．つまり，左隅構文解析とは，生成規則の右辺の先頭（左隅）は上昇型で解析し，右辺の残りの部分は下降型で解析するという方法である．文献2)では左隅構文解析表を利用した決定性構文解析系の例が示されている．

プログラム言語 Pascal は LALR(1)文法であるが[3]，LC(1)文法ではない．ここで LC(1)文法とは，1記号先読みするだけで決定的に左隅構文解析できる文法クラスである．たとえば代入文と手続き呼び出し文とを考えるとその先頭の記号はどちらも識別子であり，構文情報だけでは解析を決定的に行えない．ここで一般に，LL(k)⊂LC(k)⊂LR(k) であることが知られている[4]．

左隅構文解析の代表的なものとしては，構文解析の枠組みとして位置づけられる上昇型チャート法[5]や，非決定性解析系としてはBUP[6]やSAX[7]そしてPAX[8]など数多く提案されている．

文献9)～11)には誤りの検出と回復について解説がある．しかしこれらの方法は，コンパイラなどにおい

てできるだけ多くの誤りを発見するという目的を持っている．そのため，誤り検出後，その誤りから回復して解析を続行できるように，まだ読んでいない入力記号をいかに読み飛ばしたり，変更したりすればよいかという戦略を採用している．また，誤り発生位置が必ずしも誤り原因とは限らない．たとえば，Pascal の if 文において **else** の前にセミコロンが挿入された場合，LR 構文解析では **else** の位置で誤りが発生し，**else** 以降の読み飛ばしが試みられてしまう．

本論文では構文誤り位置の文脈情報を利用した的確な誤り診断を目標としている．たとえば，LL 構文解析ならば，開始記号から誤り位置の記号までの完全な導出過程が得られるが，扱える文法クラスが狭すぎる．一方，LR 構文解析では，扱えるクラスは広がるが，開始記号からの文脈情報があまり利用できなくなる．つまり，LR 構文解析では，複数の可能な構文木を同時に扱いそれを状態として表現しているためである．そこで本論文では左隅構文解析をとりあげ，できるだけ文脈情報を利用した誤り診断手法の実現を目標とした．

本論文では，誤り検出後，その誤りの真の原因を判定して，誤り回復を行うという点に特徴があり，生成規則と構文木を用いた一般的な誤り診断の方法論を提案している．誤り原因を診断するには，その誤り原因に対する修正を施した結果その先の解析が続行できるかどうかを調べる必要があるが，実現が複雑で効率も低下する．そこで本論文では実際に入力記号をいくつか先読みして仮に構文解析する代わりに，局所的な *FOLLOW* 集合（局所後続列集合）を利用した誤り診断方法を示す．この局所後続列集合という概念はすでに文献12)で用いられており，最左導出の過程における局所的な *FOLLOW* 集合を意味している．しかし，本論文における局所後続列集合はこの定義とは異なり，さらに解析木における節の局所後続列集合の概念も導入している．これにより，解析木上の文脈を考慮した誤り診断を単純な機構で実現することが可能になった．

また，2つの記号列がどの程度似ているかを定量化するためのキーボードモデルを導入して記号列間距離を定義し，それを用いたつづり間違いの判定も示す．構文木上の文脈を考慮することで，上記の if 文における **else** の前のセミコロンが誤り原因であることも的確に指摘できるわけである．

以下，本論文の第2章では左隅構文解析の概要と構文誤りについて示す．第3章では局所後続列集合と記号列間距離を定義し，それを用いた誤り診断の方法論を示す．第4章では本手法の特徴と課題について述べる．

2. 左隅構文解析

本章では，左隅構文解析の概要と構文誤りについて説明する．

2.1 左隅構文解析の概要

左隅構文解析法は，上昇型構文解析法の一種であるが，同時に下降型構文解析法の特色も兼ね備えている．

文脈自由文法 G を $G=(N, \Sigma, P, Z)$ とする．ここで，N は非終端記号の集合，Σ は終端記号の集合，P は生成規則の集合，$Z \in N$ は開始記号である．

生成規則 $p \in P$ は，

$p: X_0 \rightarrow X_1 X_2 \cdots X_{np}$

の形とする．ただし，$X_0 \in N$, $X_i \in (N \cup \Sigma)$, $i=1, 2, \cdots, np$ とする．

解析木をボトムアップに構成していくときに，生成規則の右辺の先頭にある非終端記号（X_1）が決定すれば，その生成規則（p）を選択し，右辺の残りの（非）終端記号列（$X_2 \cdots X_{np}$）がこれから見つけるべき目標となる．そして，部分的な構文木が完成するたびに，それが現在の目標と一致するかどうか確認され，もし一致するならば右辺の残りの要素が埋め合わされて，解析木が成長していく．この過程を開始記号（Z）を目標として，与えられた終端記号全体を葉とする木が完成するまで繰り返す（**図1**）．

構文解析に曖昧性がある場合には，後戻りしたり（BUP[6]），可能性を同時に扱う（SAX[7]，PAX[8]）ことで対応している．すなわち BUP では，解析の途中で

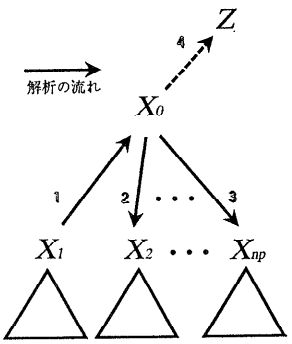


図 1 左隅構文解析
Fig. 1 Left-corner parsing.

可能な処理が複数あればそのうちの1つを選び，処理が行き詰まれば別の可能性が残されていた時点まで後戻りして，ほかの可能性を選択する．

文の解析木においてはその木の左端の子となり得るものだけが文の先頭要素として許される．このことはいかなる部分木の解析時にもいえることである．そこで文の解析途中において，どのような非終端記号が左隅にくるべきかという上からの予測（トップダウン予測）を利用することで，解析空間を縮小することが可能である．これは，チャート法[5]における到達可能性，BUP や SAX では link 述語により実現されている．つまり，どの非終端記号がどの非終端記号の左端の子孫となり得るかという関係を，文法規則全体から事前に計算しておくわけである．

2.2 構文誤り

左隅構文解析では，「必ず出現すべき終端記号が現れなかった場合」と「還元するための生成規則が存在しない場合」に構文誤りが発生する．それぞれの場合に応じ，以下のようにして誤り発生箇所に構文上くる可能性がある終端記号（以後，誤り候補と呼ぶ）を特定できる．ここでは，Pascal の for 文を用いて説明する．

statement→'**for**' *identifier* ':=' *expression*
　　to_or_downto expression '**do**'
　　statement
to_or_downto→'**to**'
to_or_downto→'**downto**'

（1） 必ず出現すべき終端記号が現れなかった場合

生成規則上で右辺左端以外の箇所にある終端記号が，入力記号と一致していない場合に構文誤りが発生する．for 文中では，":=" や "**do**" がこれに該当する．この場合の誤り候補は生成規則中の終端記号そのものとなる．例えば for 文中で "**do**" が存在しない時には，誤り候補を "**do**" と特定できる．

（2） 還元するための生成規則が存在しない場合

部分木の根の(非)終端記号を右辺の先頭に持つ生成規則が存在しない場合に構文誤りが発生する．ただし，生成規則の左辺が現在の目標から予測されているかどうか（トップダウン予測）も考慮にいれる．したがって，入力記号が現在の目標から予測される非終端記号を根とする部分木の最左端の葉になる可能性がない場合に構文誤りが発生する．for 文中で *to_or_downto* を根とする部分木を作成する際に，"**to**" または "**downto**" 以外の入力記号が存在する場合がこれに該当する．この場合，*to_or_downto* を根とする部分木の最左端の葉になり得る終端記号（"**to**" または "**downto**"）が誤り候補となる．

2.3 ε規則の扱い

構文の誤り候補を特定する際には，ε規則にも注意しなければならない．ある入力記号の位置で構文誤りが発生した場合，その左どなりの部分木でε規則が適用されているならば，ε規則の左辺の非終端記号を根とするε規則以外の可能な部分木の最左端の葉（終端記号）も誤り候補となり得る．例えば Pascal に準じた次の生成規則を考える（一部は省略）．

block→*var_decl_part proc_decls*
　　'**begin**' *statements* '**end**'
var_decl_part→'**var**' *var_decl var_decls*
var_decl_part→ε
proc_decls→*proc_decl proc_decls*
proc_decls→ε
proc_decl→'**procedure**' '**id**'…
var_decl→*id_list* ':' *type* ';'
var_decls→*var_decl var_decls*
var_decls→ε
id_list→'**id**' *more_ids*
more_ids→',' *id_list*
more_ids→ε
type→'**id**'

block を根とする部分木を作成するときに，"**var**" を先頭に持つ入力記号列を与えた場合，*var_decl_part* には1番目の規則が適用され，ε 規則は適用されない．しかし，入力記号 "**var**" を "**ver**" と誤った場合，*var_decl_part*, *proc_decls* ともにε規則が適用され，2.2 節の(1)によって誤り候補は "**begin**" と特定される．さらに，誤り発生位置の左どなりの部分木でε規則が適用されているので，*var_decl_part* および *proc_decls* を根とする部分木の最左端の葉になり得る終端記号（"**var**", "**procedure**"）も誤り候補

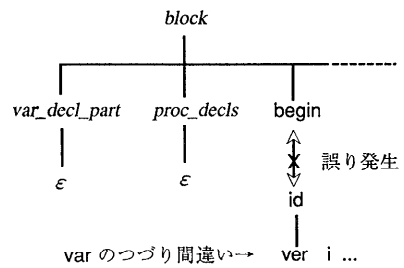


図 2 ε規則を伴う構文誤り
Fig. 2 Syntax error with ε-rules.

となる（**図2**）.

誤り発生位置の左どなりの部分木にε規則が適用されているかどうかを判断するには，これまでに生成された解析木を保存しておく必要がある．したがって，以下では解析木を作り，その上で構文誤りの診断を行う方法を提案する.

3. 構文誤り診断

構文誤りの診断，すなわちその誤りの原因と回復方法を考えるとき，上述の誤り候補を羅列するだけでは必ずしも適切な診断は行えない．以下では，(1)局所後続列集合の概念を導入して，構文解析木上の文脈を考慮した誤り診断，(2)キーボードモデルを反映した記号列間距離を用いたつづり間違いの判定，(3)誤り発生位置の1つ前の入力記号が誤っている可能性の指摘方法について述べる.

(1)は単純な機構を用いて効率良い誤り診断の実現を目的としている．(2)はつづり間違いの検出にキーボードの配置をも考慮して，より適切な誤り指摘をめざしている．(3)は Pascal などではよくある構文誤りで，たとえば if 文において **else** の前にセミコロン“;”が挿入された場合などに相当する.

3.1 局所後続列集合

誤り原因を判定するには，その誤り原因に対する修正を施した結果，その先の解析を続行できるかどうかを調べてみる必要がある．このためには仮に構文解析を進めるという処理を伴うため，実現が複雑で，効率も低下する．実際，誤り候補の数が数十にも及ぶことも少なくない．それらの誤り候補について，このような処理を行うことは実現上問題がある．そこで，本論文では局所後続列集合という概念を導入する．実際に入力記号をいくつか先読みして構文解析する代りに，この局所後続列集合を利用して誤り原因を判定するのである.

以下では特に断らない限り，$A, B \in N, X, Y \in (N \cup \Sigma)$, $a, b, u, v, w \in \Sigma^*$, $\alpha, \beta \in (N \cup \Sigma)^*$ とする．また，構文木上の(非)終端記号Xの生起（木の節）を添え字 t を付加し X^t と表して区別する.

通常，非終端記号 A の後続列集合 $FOLLOW_k(A)$ は次のように定義される.

定義 1. 後続列集合 $FOLLOW_k(A)$

$$FOLLOW_k(A) = \{w \mid Z \Rightarrow^* \alpha A \beta \text{ なるすべての導出に対して } w \in FIRST_k(\beta)\}.$$

ここで，

$FIRST_k(\beta) = \{first_k(w) \mid \beta \Rightarrow^* w\}$,

$first_k(w) = u$, ただし，$w = uv$,

$|u| = k$, $v \neq \varepsilon$, または $|u| \leqq k$, $v = \varepsilon$ である.

$|u|$ は u の長さを意味する.

しかし，ここでは文脈に着目し，その文脈を考慮した後続列集合，局所後続列集合 $LFOLLOW$ (Local Follow Set) を考える．文脈によって，同じ記号でもその後に続く可能性がある終端記号は違ったものになる．たとえば，Pascal において，代入文や if 文などの中で使われる“)”の後には {';', ',', '+', '−', '×', …} が続き，手続き宣言の頭部で使われる“)”の後には {';'} が続く．つまり手続き宣言の頭部で“)”が使われた場合，その後に {',', '+', '−', '×', …} が続く可能性はない.

定義 2. 局所後続列集合 $LFOLLOW_k(A, X)$

$$LFOLLOW_k(A, X) = \{w \mid A \Rightarrow^* \alpha X \beta \text{ なるすべての導出に対して } w \in FIRST_k(\beta)\}.$$

定義1と定義2より，$A \in N$ に対しては明らかに $LFOLLOW_k(Z, A) = FOLLOW_k(A)$ が成り立つ.

上述の Pascal の例では，

$LFOLLOW_1(proc\_decl, ')') = \{';'\}$,

$LFOLLOW_1(statement, ')') = \{';', ',', '+', '-', '\times', \cdots\}$

となる.

トップダウンに解析を行えば，解析途中でも開始記号Zから葉までの完全な導出過程を得ることができる．しかし，左隅構文解析では解析をボトムアップに行うので，解析途中において開始記号からの完全な導出過程を得ることはできない(**図3**参照)．ただし，開

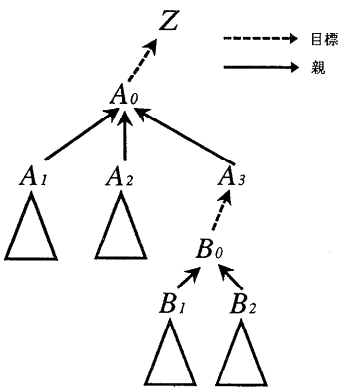


図 3 解析途中の左隅構文木
Fig. 3 Partial left-corner parse tree.

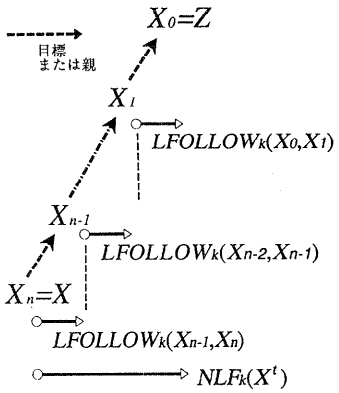


図 4 節局所後続列集合 *NLF* の概念
Fig. 4 The concept of *NLF*.

始記号 Z から始まって，ある節 X^t までにいたる目標となっている非終端記号は分かっている．つまり節 X^t に対しては，次のいずれかの場合が成り立つ．

(a) 節 X^t の親 A^t が存在する．

(b) 節 X^t の親がまだ存在しないとき，X^t の先祖となる目標 B^t が存在する．

これまでに生成された構文解析木から上記のような導出の情報を得て，その文脈上の後続列集合，節局所後続列集合 *NLF* (Node Local Follow Set) を定義する（**図4**参照）．

定義 3. 節局所後続列集合 $NLF_k(X^t)$

$$NLF_k(X^t)=LFOLLOW_k(X_{n-1},X_n)\oplus_k\cdots LFOLLOW_k(X_1,X_2)\oplus_k LFOLLOW_k(X_0,X_1).$$

ここで，$X_0^t=Z^t$，$X_n^t=X^t$，X_{i-1}^t は X_i^t の親または目標の節である $(i=1,\cdots,n)$．また，

$$U\oplus_k V=\{w\mid w=first_k(uv),u\in U,v\in V\}.$$

$k=1$ の場合，構文木上の節 X^t の節局所後続列集合 $NLF_1(X^t)$ は次のアルゴリズムによって構成できる．

以下，特にことわりのない限り $k=1$ の場合を扱い，添字 k を省略する．

アルゴリズム 1. 節局所後続列集合 *NLF* の構成

入力：構文木と節 X^t

出力：節局所後続列集合 $NLF_1(X^t)$

(1) $NLF:=\{\varepsilon\}$；

(2) $Y^t:=X^t$ の親または目標の非終端記号；
$NLF:=(NLF-\{\varepsilon\})\cup LFOLLOW(Y,X)$；

(3) Y が開始記号 (Z) ならば(5)へ；

(4) $\varepsilon\notin NLF$ ならば(5)へ；
$\varepsilon\in NLF$ ならば Y^t をあらたに X^t とし(2)へ；

(5) *NLF* を出力する．

3.2 記号列間距離

2つの記号列がどの程度似ているかを記号列間に距離を定義することによって定量化する．まず，2つの記号列を DP マッチングアルゴリズム[13]を用いてマッチングをとる．たとえば，次の2つの記号列 α と β を考える．

α=MPQRSTVWPYT,
β=MNPQYSTWYT

この2つの記号列は一見あまり似ていないように見えるが，記号列 α をもとに N が出現し，R が Y に変化し，V と P が欠落したと考えると，この2つの記号列は4箇所が異なるだけである．ダミー記号＊を用いて表現すると次のようになる．

α'=M＊PQRSTVWPYT,
β'=MNPQYST＊W＊YT

このように記号列の違いをダミー記号＊を用いると，出現・変化・欠落という3種類の形で表現することができる．

次にマッチングがとられた2つの記号列の同じ位置にある記号間の距離（キーボードモデルによって定義される）を求め，その総和をとる．記号間の距離の総和をとっただけでは，長さ 10 の記号列中の1字だけを間違った場合と長さ1の記号の1字を間違った場合の記号列間距離にほとんど差がなくなってしまうので，総和を記号列の長さで割ったものを記号列間距離とする．

3.3 誤り原因の判定

誤り発生箇所に構文上くる可能性がある終端記号(誤り候補)は，2.2節および2.3節で述べた方法で特定される．そこで，いくつかの誤り原因を考え，各誤り候補がどの誤り原因に関連しているかを調べ，その誤り原因に応じて誤り診断情報を提示する方法を以下で示す．

まず，誤り原因を6通りに分類する．それぞれ例を用いて以下に示す．

誤り原因の分類

(1) つづり間違い
procedure を **proceudre** と間違う．

(2) 連接（空白の脱落）による誤り
x :=x **div** 2 を
x :=x div2 と間違う．

（3） 終端記号が1つ脱落する誤り
x :=0 ; y :=0 を
x :=0 y :=0 と間違う.

（4） 余計な終端記号が1つ挿入されている誤り
if x＝0 **then** y :=0 **else** y :=1 ; を
if x＝0 **then** y :=0 ; **else** y :=1 ; と間違う.

（5） 対象言語特有の誤り（ある終端記号を別の終端記号と勘違いして使用する誤り）
procedure foo(a : integer ; b : integer); を
procedure foo(a : integer, b : integer); と間違う.

（6） その他の誤り（上記以外の誤り）

上記(1)から(5)の各誤り原因を次に示す条件により判断する.

誤り原因の判定条件

（1） つづり間違い
(1-1) 誤り位置の入力記号と誤り候補との記号列間距離が近く，かつ
(1-2) 誤り候補の節局所後続列集合中に誤り位置の次の入力記号が存在する.

（2） 連接による誤り
(2-1) 誤り位置の入力記号の頭部と誤り候補が一致する.

（3） 終端記号の脱落による誤り
(3-1) 誤り候補の節局所後続列集合中に誤り位置の入力記号が存在し，かつ
(3-2) 誤り位置の入力記号の節局所後続列集合中に誤り位置の次の入力記号が存在する.

（4） 終端記号の挿入による誤り
(4-1) 誤り候補が誤り位置の次の入力記号と一致する.

（5） 対象言語特有の誤り
誤り候補とその文脈（誤り候補までの導出過程）を用いて，「対象言語特有の誤り」の知識（後述）を参照し，誤り候補と勘違いしやすい終端記号を求める.
(5-1) 勘違いしやすい終端記号が誤り位置の入力記号と一致し，かつ
(5-2) 誤り候補の節局所後続列集合中に誤り位置の次の入力記号が存在する.

ここで，(1-2)，(3-2)，(5-2) における条件は，誤り位置の次の入力記号を先読みすることによって，誤り原因判断の精度を向上させることに寄与している.

対象とする言語特有の勘違いしやすい誤り情報を「対象言語特有の誤り」の知識（データベース）として用意しておく．たとえば Pascal においては，手続き宣言の引数並びの区切り記号 “ ;” を “,” と，代入文の “ :=” を “=” とそれぞれ勘違いしやすいといった情報である．このような情報をその文脈とともに次のような形式で準備しておく.

非終端記号 X から導出される終端記号 a が終端記号 b と勘違いされやすいならば，

mistake(X, a, b).

を登録しておく.

上記 Pascal の例を次に示す.

mistake(*formal_para_sections*, ' ; ', ' , ').
mistake(*assignment*, ' :=', '=').

このような知識は，必要にして十分なだけ準備しておくわけではなく，頻繁に起こり得る対象言語に依存した構文誤りの知識を導入することで，より的確な診断情報を効率よく提示するために利用されている.

誤り診断例） 図2で示した “**var**” のつづり間違いによる誤りを診断してみる．誤り候補は2.3節で述べたように “**var**”, “**procedure**”, “**begin**” である．ここでは，“**var**” が「つづり間違いによる誤り」と「終端記号の脱落による誤り」に関連しているかどうかを判断してみる.

（a） “**var**” のつづり間違いによる誤りの診断

“**var**” と “**ver**” の記号列間距離は近いので，条件(1-1)は成立する．2.3節より “**var**” は *var_decl_part* から導出されるので，“**var**” の節局所後続列集合は，

NLF ('**var**'')
＝*LFOLLOW*(*var_decl_part*, '**var**')
＝{'**id**'}

となる．ここで **id** は識別子である．誤り位置 “**ver**” の次の入力記号は **id**（識別子 “i”）であり，条件(1-2)も成立する．したがって，“**var**” のつづり間違いによる誤りと判断される.

（b） “**var**” の脱落による誤りの診断

上述のように，“**var**” の節局所後続列集合は {'**id**'} であり，誤り位置の入力記号も **id**（識別子 “**ver**”）であるので，条件(3-1)は成立する．“**var**” の節局所後続列集合 {'**id**'} は *var_decl_part* から導出されたので，誤り位置の入力記号 “**ver**” の節局所後続列集合は，

NLF('**id**'')
＝*LFOLLOW*(*var_decl_part*, '**id**')
＝{'**;**', '**:**', '**,**'}

となる．この集合中には誤り位置“**ver**”の次の入力記号 **id**（識別子“i”）は存在しないので条件 (3-2) は成立せず，“**var**”の脱落による誤りではないと判断される．

誤り原因の判定条件は排他的ではない．例えば，代入文 x:=y div2; において誤り原因を判定してみると，次のように複数の誤り原因が得られる．

- 連接による誤り：
  div 2 を div2 と間違う．
- 終端記号の脱落による誤り：
  識別子“y”の後ろに“;”や
  演算子（+，−，×など）がない．
- 終端記号の挿入による誤り：
  識別子“div2”が余計．

これらの誤り原因を提示する際にはその順序を考慮する必要がある．そこで，本手法を組み込んだ言語指向エディタでは，複数の誤り候補や原因がある場合には，各終端記号ごとに誤り原因に対する優先度情報を用意し，それにしたがって診断情報の提示順序を決定している（第4章参照）．つまり，記号“**div**”は連接による誤りが起こりやすいという情報によって，**div** 2 を div2 と間違う誤りを最初に提示できるようになっている．

以上のようにして判定された誤り原因に応じて誤り診断情報を提示する．ここでは 3.3 節の最初で用いた例について示すことにする．

誤り診断情報の提示

（1） つづり間違い
　予約語 **procedure** を **proceudre** とつづり間違いしている．
（2） 連接による誤り
　div と 2 の間に空白がない．
（3） 終端記号の脱落による誤り
　識別子 y の前に“;”がない．
（4） 終端記号の挿入による誤り
　else の前の“;”が余計．
（5） 対象言語特有の誤り
　“;”を“,”と勘違いしている．
（6） その他の誤り
　foo がきていない．（**foo** を誤り候補とする）

3.4 構文誤りの的確な指摘

誤りが発生した位置より数個前の入力記号の場所に本当の誤り原因があることが多々ある．一般に，数個前にさかのぼってこのような誤り原因を指摘することは困難であるので，以下では誤りが発生した位置の1つ前の場所における誤りの可能性を指摘する方法を述べる．実際，Pascal におけるよくある構文誤りを検討してみると，誤り発生位置の1つ前の場所に誤り原因があることが意外に多い．代表的な例を以下に2つ示す．

（1） if 文において **else** の前にセミコロン“;”が挿入された場合，if 文はそのセミコロンで終了したと解釈され構文解析が進む．その結果，次の入力記号 **else** の位置で誤りが発生し，前節で述べた方法だけでは誤り候補は，{**if, for, id**（識別子）, …} となってしまう．このため **else** の前のセミコロンが余計であることが指摘できない．

（2） 変数宣言の後で，**procedure** や **begin** をつづり間違いした場合，それらは識別子と認識されるので，変数宣言の続きとして構文解析が進む．その結果，次の入力記号の位置で誤りが発生し，誤り候補は {':', ',', '} となる．このため **procedure** や **begin** のつづり間違いを指摘できない．

このような場合にも的確に誤り指摘を行うには，まず作成された構文木を探索することにより，誤り発生位置の1つ前の場所に構文上くる可能性がある終端記号を探す（下記のアルゴリズムによる）．次に，誤り発生位置の1つ前の場所を誤りであると仮定し，その入力記号について，前節で示した方法で誤りの原因（つづり間違い，連節による誤り，終端記号の脱落による誤り，終端記号の挿入による誤り，対象言語特有の誤り）を調べる．そして誤りの可能性があれば，誤り診断情報を提示する．

アルゴリズム 2. 誤り発生位置の1つ前の場所に構文上くる可能性がある終端記号を探す

入力：構文木，誤り発生位置の1つ前の入力記号
出力：誤り発生位置の1つ前の場所に構文上くる可能性がある終端記号の集合

（1） 節 X' :=誤り発生位置の1つ前の入力記号；
　V:={ }；
（2） 節 X' が構文木の根の場合（すなわち X が開始記号 Z のとき），（6）へ．
（3） 節 X' が構文木の上で長男の場合，節 X' の親または目標となる節をあらたに X' とし（2）へ．
（4） 節 X' の左どなりの構文木で ε 規則が適用されているならば（**図5** の B_1, B_2 を根とする部分木），その ε 規則の左辺の非終端記号を根とする ε 規則以外の可能な部分木の最左端の葉になり得る終端記号の

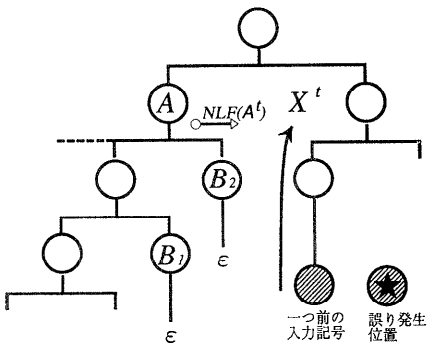


図 5 ε規則を含む構文木上の探索
Fig. 5 Traverse on parse tree with ε-rules.

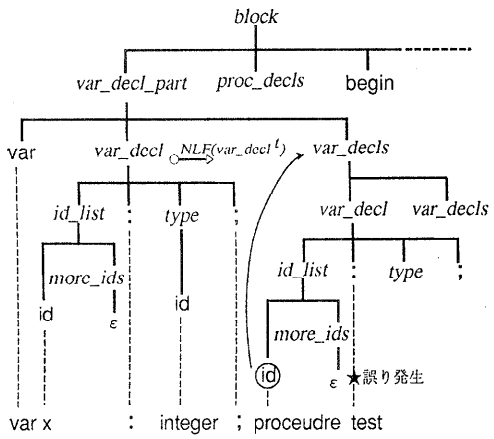


図 6 つづり間違いの診断
Fig. 6 Spelling error diagnosis.

集合 U を求め，$V:=V\cup U$；
(5) 節 X^t の左どなりの節（図5の A^t）の節局所後続列集合 $NLF(A^t)$ を求め，$V:=V\cup NLF(A^t)$；
(6) 集合 V を出力する．

変数宣言の後で，**procedure** を **proceudre** とつづり間違いした場合の誤り診断例を**図6**に示す．入力記号列を，

var x: integer; proceudre test

とするとき，**test** の位置で構文誤りが発生する．その1つ前の **proceudre** の場所に構文上くる可能性がある終端記号をアルゴリズム1および2によって求めると，

$$NLF(var\_decl^t)$$
$$=(LFOLLOW(var\_decl\_part, var\_decl)-\{\varepsilon\})\cup LFOLLOW(block, var\_decl\_part)$$
$$=\{\textbf{id}, \textbf{procedure}, \textbf{begin}\}$$

となる．ここで，

$$LFOLLOW(var\_decl\_part, var\_decl)=\{\textbf{id}, \varepsilon\},$$
$$LFOLLOW(block, var\_decl\_part)=\{\textbf{procedure}, \textbf{begin}\}$$

である．したがって，**procedure** を **proceudre** とつづり間違いをした誤りを指摘できる．

4. 評　価

LR 構文解析と比較した場合，左隅構文解析では大きな LR 構文解析表を作成する必要がなく，小型な処理系の実現が可能である．また，属性の意味情報を利用して解析を決定的に行える LC(1)属性文法[14]では，手続き型言語Cなどによる効率よい処理系が実現できる．そこで現在，本手法を組み込んだ言語指向エディタ[15]が開発されている．このエディタはプログラム編集，静的意味解析，誤り診断・回復支援などの機能を持ち，知識ベースを導入することで，異なる言語への適用性やシステムの柔軟性を実現することを目的としている．このエディタには，本論文では割愛したが，複数の誤り候補や原因に対する優先順位の決定と，自動修正機能も付加してある．

誤り診断情報が複数存在する場合，各終端記号ごとに誤り原因に対する優先度情報が用意されているので，それにしたがって提示順序を決定する．さらに利用者による誤り修正の頻度に応じて優先度情報を更新することで，対象言語の誤り傾向を学習する機能も実現されている．

3.3 節で述べた5種類の誤り原因に関しては，簡単にその修正処理を自動化することができる．まず各種誤り診断情報を提示し，利用者に選択させる．そして，選択された誤りに対する修正方法を提示し，利用者に自動修正の是非を求める．たとえば，**else** の前の“;”を削除するか否かを問い合わせるなどである．利用者が自動修正を選択したならば，言語指向エディタが誤りの修正と回復をはかり，解析を続行する．

本手法を組み込む以前の言語指向エディタと比較すると，構文誤り処理に関しては次の点が改善された．(1)従来の言語指向エディタでは構文誤りに関する誤り提示情報や誤り回復知識が対象言語の知識記述中に埋め込まれていて繁雑であった．今回のシステムでは知識記述中には構文誤りに関する記述をいっさい行わずに済んだ．(2)構文誤りに関する誤り診断情報が具体的で理解しやすくなった．(3)従来は指摘できなか

ったいくつかの構文誤りに対しても的確に指摘できるようになった．(4)単純な構文誤りの場合にはマウスの操作だけで修正作業が行え，利用者の負担が軽減した．

しかし本手法では誤り候補 **id**（識別子）に関して，誤り原因の判定（つづり間違い，連接による誤り）と自動修正（識別子の挿入と置換）に制限がある．すなわちこれらの場合には，記号表を参照する必要があるからである．この記号表は識別子の二重宣言や未宣言使用をチェックするために意味解析時に用いられるが，どのようなデータ構造を持ち，どのように参照するかという情報は意味記述の方法に依存する．本論文では記号表に関連する処理はすべて意味解析時に行うという方針をとり，構文誤り診断は意味解析とは独立にして汎用性を持たせることを目標としたのでこのような制限がでてきた．

構文木上の文脈を考慮した誤り診断のために局所後続列集合を導入したが，実際にその先の入力記号を解析しているわけではないので，おかしな誤りを指摘してしまうこともある．しかし，Pascalにおけるよくある構文誤りに対しては，本手法を用いた簡単な誤り原因判断機構でも十分有効であることが確認できた．ただし，次の例のように for 文中で **for** を書き忘れた場合，それに続く文は代入文 (x :=1) だと認識され構文解析が進み，その結果 **to** の場所で誤りが発生する．

x :=1 **to** 10 **do**…

この時の誤り候補は，{';', '+', '−', '×',…} となり，誤り発生位置より数記号も前にある **for** の脱落による誤りまでは指摘できない．このような誤り原因を診断するには，誤り発生位置から逆にさかのぼって部分木を仮に構成してみる手法を考案しているが，この方法については稿を改めて別の機会に論じたい．

5. おわりに

左隅構文解析時に発生する構文誤りについて考察し，さらにその構文誤りの診断方法（誤り原因の判定と回復方法）について論じた．誤りの診断には対象言語に依存した構文誤りの知識を導入するだけではなく，対象言語の生成規則と構文解析木を用いたより一般的な誤り診断の方法論を提案した．

本手法は以下の特徴を持つ．(1)局所後続列集合を定義し，構文木上の文脈を考慮した誤り診断を単純な機構で実現，(2)キーボードモデルを反映した記号列間距離を用いたつづり間違いの判定，(3)誤り発生位置の1つ前の入力記号が誤っている可能性の的確な指摘．

本手法を組み込んだ言語指向エディタを使用してみると，Pascal におけるよくある構文誤りに対しては，本論文で提案した簡単な誤り原因判断機構でも十分有効であることが確認できた．

LR 構文解析系に対して本誤り診断手法の応用を考えてみると，構文木上の親や目標となる節が陽には存在しないので，そのままでは節局所後続列集合 *NLF* は構成できない．しかし，LR 状態に対して *NLF* に相当する集合を準備することで適用可能となる．

左隅構文解析系の中でも非決定性構文解析系では LR 文法よりも広いクラスを扱うことができる．特に PAX は並列構文解析系であり，今後の課題として，並列構文解析に伴う誤り診断への議論の展開を考えている．

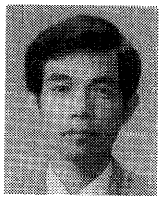
謝辞　本研究を行う機会を与えていただいた東京理科大学井上謙蔵教授，および熱心に議論いただいた栗林茂雄氏ほか情報基礎研究室 LoE グループの諸氏に感謝いたします．

参考文献

1) Rosenkrantz, D. J. and Lewis II, P. M.: Deterministic Left Corner Parsing, *IEEE Conf. Record 11th Annual Symposium on Switching and Automata Theory*, pp. 139-152 (1970).
2) Aho, A. V. and Ullman, J. D.: *The Theory of Parsing, Translation and Compiling*, Vol. 1 and Vol. 2, Prentice-Hall, Englewood Cliffs, N. J. (1972).
3) Kastens, U., Hutt, B. and Zimmermann, E.: *GAG: A Practical Compiler Generator, Lecture Notes in Computer Science, 141*, p. 156, Springer (1982).
4) op den Akker, R., Melichar, B. and Tarhio, J.: *The Hierarchy of LR-Attributed Grammars, Lecture Notes in Computer Science, 461*, pp. 13-28, Springer (1990).
5) Kay, M.: Algorithm Schemata and Data Structures in Syntactic Processing, Tech. Rep. CSL-80-12, XEROX PARC (Oct. 1980).
6) Matsumoto, Y. et al.: BUP: A Bottom-up Parser Embedded in Prolog, *New Generation Computing*, Vol. 1, No. 2, pp. 145-158 (1983).
7) 松本裕治，杉村領一：論理型言語に基づく構文解析システム SAX，コンピュータソフトウェア，Vol. 3, No. 4, pp. 4-11 (1986).
8) Matumoto, Y.: A Parallel Parsing System for Natural Language Analysis, *Proc. 3rd In-*

ternational Conf. on Logic Programming, Lecture Notes in Computer Science, 225, pp. 396-409, Springer (1986).
9) Aho, A. V. and Ullman, J. D.: *Principles of Compiler Design*, Addison-Wesley (1977), (邦訳) 土居範久ほか：コンパイラ，培風館 (1986).
10) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers—Principles, Techniques, and Tools*, Addison-Wesley (1986), (邦訳) 原田賢一：コンパイラ，サイエンス社 (1990).
11) 佐々政孝：プログラミング言語処理系，岩波書店 (1989).
12) Kenzo, I. and Fujiwara, F.: On LLC(k) Parsing Method of LR(k) Grammars, *J. Inf. Process.*, Vol. 6, No. 4, pp. 206-217 (1983).
13) Needleman, S. B. and Wunsch, C. D.: A General Method Applicable to Search for Similarities in the Amino Acid Sequence of Two Proteins, *J. Mol. Biol.*, Vol. 48, pp. 443-453 (1970).
14) 武田正之：左隅構文解析法と Prolog によるその構文・意味解析，情報処理学会論文誌，Vol. 33, No. 1, pp. 11-17 (1992).
15) 武田正之：知識ベースに基づく Language-oriented Editor, 情報処理学会論文誌，Vol. 28, No. 11, pp. 1154-1161 (1987).

(平成4年3月5日受付)
(平成4年10月8日採録)



武田 正之（正会員）

昭和 28 年生．昭和 52 年東京理科大学理工学部電気工学科卒業．昭和 57 年東京工業大学大学院博士課程（電子物理工学専攻）修了．同年東京理科大学理工学部情報科学科助手となり，現在同大学講師．工学博士．著書(共著)に「Prolog とその応用2」，総研出版（昭和 60 年）がある．プログラミング言語の意味，証明論，知識情報処理に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，人工知能学会，ACM 各会員．