ショートノート

構造エディタのためのインクリメンタル LL パーサの一構成法†

佐藤 豊††* 板野肯三†††

構造エディタでプログラムを編集したときに，変更による影響が及ぶ範囲の構文だけを再解析するインクリメンタルパーサの構成法について述べる．このパーサは再帰下降パーサをもとにして，構文解析状態の回復と，解析済みの入力記号列の非終端記号としての読み飛ばし，および再解析の終了を判定するアルゴリズムを加えたものである．

1. まえがき

構造エディタは，言語の構文に基づくプログラムの構造を維持し，プログラムの文字列に挿入や削除などの変更が行われたとき，変更の影響が及ぶ範囲の構文を解析し直してプログラムの構造を更新する．このとき，過去の解析結果を生かして再解析の量を減少させる技法として，インクリメンタル・パーサが用いられる．このインクリメンタル・パーサを LR 文法に基づいて構成する方法は文献 1)～3) などに示されているが，LL 文法向きの構成法はいまだに確立されていない．

LL 文法に基づいた初期の実現例として，再帰下降パーサを使用したものが文献 4) に示されているが，これは再解析に備えてあらかじめ複数の入力記号位置での構文解析状態を保存しておき，後にこれを回復してプログラムの残り全体を解析し直すので，性能面で問題がある．より効率的なインクリメンタル LL(1) パーサの1つが文献 5) で示されているが，これは言語に依存して分割した "fragment" と呼ぶ構文の単位を再解析の範囲としているので，言語独立性の点で問題がある．

本論文に示すインクリメンタル・パーサは，任意の入力記号から解析を再開でき，過去の解析結果（構文木）を再利用して解析を必要最小限で終了する．バックトラックにより，LL(k) 文法を認識するが，文献 5) のように言語に依存する構文の単位を仮定していないので，LL(k) 文法で表せる言語一般に適応できる．再解析は最初の解析と同様に入力記号の列を解析しながら行うが，その間，過去の解析で生成された構文木から利用できる部分木を切り出して再利用し，対応する入力記号の列を読み飛ばす．そして，過去に解析されたことのある入力記号を前回と同一の解析状態で解析しようとしたとき，再解析を終了する．

2. プログラムと構文情報の表現形式

はじめに，本方式の基礎となる，プログラムの構造と構文解析状態の表現形式，および言語の構文情報を表現する関数を定義する．

2.1 プログラムと解析状態の表現形式

プログラムは最初，双方向にリンクされたトークンの列として与えられる．パーサは，これを解析して構文木を生成し，解析したトークンと対応する構文木のノードを相互にリンクする．終端記号に対するノードは作らず，トークンで表す．トークンとノードのデータ構造を以下に示す．

```
record Token : {トークンのデータ構造}
  pre, suc : ↑Token ; {前後のトークンへのポインタ}
  np : ↑Node ; {対応するノードへのポインタ}
  i : integer {対応する生成規則の右辺での記号位置}
end
record Node : {ノードのデータ構造}
  parent : ↑Node ; {親ノードへのポインタ}
  pi : integer ; {親ノード中での記号位置}
  children : array of ↑Node ; {子ノードへのポインタの配列}
  tp : ↑Token ; {対応する先頭のトークンへのポイン
```

† Design of an Incremental LL Parser for a Structure Editor by YUTAKA SATO (Doctoral Program in Engineering, University of Tsukuba) and KOZO ITANO (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学工学研究科
††† 筑波大学電子・情報工学系
* 現在 電子技術総合研究所

タ}

n, a, i : integer {n : 非終端記号, a : 生成規則の選択肢, i : 生成規則の右辺での記号位置}

end

パーサ自身が持つ解析状態は，現在どのノード上でどのトークンを解析しているかを示す2つのポインタ(現在のトークン：CT，現在のノード：CN)だけであり，それ以外の状態はトークン列と構文木の中に埋め込んで保持する．

2.2 対象言語の構文の表現関数

対象とする言語の構文を表す関数として，以下のものが与えられているとする．この中で，引数Nはノードへのポインタ，Tはトークンへのポインタを表す．

function Symbol(N): {非終端記号 N.n を左辺とする N.a 番目の生成規則の，右辺の N.i 番目の記号}

function SymType(N): {Symbol(N) の型 (TERMINAL または NONTERMINAL)}

function Match(N, T): {Symbol(N) と T の記号が一致するなら TRUE, そうでないなら FALSE}

function Optional(N): {非終端記号 N.n が空の生成規則を持つなら TRUE, そうでないなら FALSE}

function NumAlternatives(N): {非終端記号 N.n を左辺とする生成規則の数}

function NumSymbols(N): {生成規則 (N.n, N.a) の右辺の記号の数}

function ChildIndex(N): {j | 生成規則 (N.n, N.a) の右辺中で，N.i 番目の記号は j 番目の非終端記号}

これらの関数は，言語の文法の BNF 表現から機械的に生成でき，対象とする言語に対してこれらの関数を与えれば，パーサはその言語に対応する．

3. 構文解析の基本アルゴリズム

構文解析のアルゴリズムは，最初の解析と再解析で基本的に同一である．本節では，再解析時の処理を除いた，基本アルゴリズムについて述べる．以下のアルゴリズムのなかで，下線を施した3か所の部分が，再解析時の処理に対する入り口である．アルゴリズムの記述にあたり，簡単のために，構文エラーの処理や，ノードの消去などは省略する．空のポインタは Λ で表し，配列の添字や選択肢の番号は0から始まるとする．

3.1 解析アルゴリズムの本体

最初の解析では，手続き Initialize で構文木の根を生成して，ポインタ CT および CN の初期化を行い，トークンの列を解析する手続き ParseTokens を呼ぶ．

```
procedure Parse: {最初の解析}
  Initialize; ParseTokens □
procedure Initialize: {初期化}
  CT←プログラムの先頭のトークン;
  CN←NewNode(文法の開始記号) □
```

手続き ParseTokens は，トークンごとに関数 Parse1Token を呼ぶ．Parse1Token は，1トークンを構文解析し，結果としてマッチングに成功したか，失敗したか，あるいは空の生成規則であったかを返す．ParseTokens では，この結果が成功なら NextTokens で次のトークンへ進み NextGoal で次のマッチングの目標を設定する．結果が失敗なら，バックトラックのための手続き DeepBack (3.2 節参照) を呼ぶ．

```
procedure ParseTokens: {トークン列の解析}
  repeat  if ReParseDone then return;
    r←Parse1Token;
    if r=FAIL then DeepBack else
    if r=EMPTY then NextGoal else
    if r=SUCCESS then NextToken; NextGoal
  until CT=Λ □
function Parse1Token: {1トークンの解析}
  if SymType(CN)=NONTERMINAL then
    if Skip then return SUCCESS else
    begin DownNode; return Parse1Token  end;
  if Match(CN, CT) then return SUCCESS  else
  if ShallowBack then return Parse1Token  else
  if Optional(CN) then return EMPTY  else
  return FAIL □
procedure NextGoal: {次の目標記号への上昇}
  CN.i←CN.i+1;
  while CN.i<NumSymbols(CN) do
  begin UpNode; CN.i←CN.i+1 end □
```

このなかで，Skip と ReParseDone は再解析時に特有の処理である (4.2 節，4.3 節参照)．構文木のノードは，Parse1Token で非終端記号を展開する時に DownNode 中で生成され，現在のトークンに結合される．また，トークンはその解析の終了時に，NextToken 中で現在のノードに結合される．

3.2 バックトラックのアルゴリズム

関数 ShallowBack は，同一のトークンの再解釈を行うための浅いバックトラックを行う．文法が LL(1) なら，パース表を用いて Parse1Token 中の ShallowBack は避けられる．$1<k$ である LL(k) 文法を認識するには，前のトークンに戻って再解釈する必要があり，この深いバックトラックを手続き DeepBack で行う．DeepBack の中で参照している ReBack は，再解析時に特有なバックトラックを行う関数である (4.4 節参照)．

```
function ShallowBack: {1トークン内でのバックトラック：選択肢の残っているノードまで戻る}
  repeat  CN a←CN.a+1;
          if CN.a<NumAlternatives(CN)
          then return TRUE;
          UpNode;
  until CN.tp≠CT;
  return FALSE □
procedure DeepBack; {前のトークンへのバックトラック：選択肢の残っているトークンまで戻る}
  repeat  if ReBack then return;
          PreToken; CN←CT.np; CN.i←CT.i;
  until ShallowBack □
procedure PreToken: (入力トークンを後退させる：選択を行ったトークンまで戻る}
  repeat CT.np←Λ; CT.i←0; CT←CT.pre
  until CT.np.tp=CT □
```

3.3 データ構造の操作プリミティブ

トークン列や構文木を操作するプリミティブ手続きと関数を以下に示す．

```
procedure NextToken: {入力トークンを前進させる}
  CT.np←CN; CT.i←CN.i; CT←CT.suc. □
procedure DownNode: {ノードの生成と下降}
  n←CN; CN←NewNode(Symbol(n));
  JoinNode(n, CN); CN.tp←CT □
function NewNode(S): {ノードの生成と初期化}
  n←新しいノード; n.n←S; return n □
procedure JoinNode(P, C): {親子ノードの結合}
  P.children[ChildIndex(P)]←C;
  C.parent←P; C.pi←P.i □
procedure UpNode: {1ノード分の上昇}
  CN.parent.i←CN.pi; CN←CN.parent □
```

再解析時に解析状態を維持するために，親子ノードを結合する JoinNode で親の記号位置 i を子ノードの pi に保存し，ノードを上昇する手続き UpNode で上昇した先の親ノードの記号位置を再設定している．

4. 再解析のアルゴリズム

変更前のプログラムを xyz とし (x, y, z は任意長のトークン列)，変更後のプログラムを $x\bar{y}z$ とする．(y と $\bar{y}$ の置換または y の削除あるいは $\bar{y}$ の挿入が行われた)．挿入または置換されたトークンは構文木に結合されていない．再解析は x の末尾のトークンから開始し，$\bar{y}$ を解析した後，z 中のトークンで終了する．x および z 中の解析済みのトークンに結合されている構文木は再利用される．

4.1 解析状態の回復と再解析

再解析ではまず，変更を受けたトークン列の直前の (x の末尾の) トークンで，前回の解析が成功した時点の状態を回復して，解析を再開する．この際，再解析を始めたトークンをポインタ BT で記憶しておく．

```
procedure ReParse: {再解析}
  CT←ȳz の先頭のトークン;
  BT←CT; Recover; ParseTokens □
procedure Recover: {解析状態の回復}
  if CT.pre=Λ then Initialize; else
  begin CN←CT.pre.np; CN.i←CT.pre.i;
    NextGoal;
    end □
```

4.2 読み飛ばし

再解析で z の部分が前回と異なる解釈をされた場合，この部分に結合していた構文木は根を持つ構文木から切り離されるが，トークンに結合したまま残される．したがって z 中のトークンは終端記号だけでなく，それを最初の終端記号として展開された部分木の根の非終端記号を表す．そこで，非終端記号 X の展開時に，解析済みのトークン T を入力したとき，T を先頭トークンとする構文木の非終端記号の組の中に X と一致するものがあれば，その構文木を結合して，その構文木に対応する最後のトークンまで読み飛ばす．

```
function Skip: {記号列の読み飛ばし}
  if CT.np≠Λ then begin
    n←CT.np;
    repeat if n.n=Symbol(CN) then begin
           JoinNode(CN, n); CT←LastToken(n);
           CN←CT.np; CN.i←CT.i;
```

```
            return TRUE
            end;
    n←n.parent
    until n.tp≠CT;
  end; return FALSE □
function LastToken(N): {ノードNが覆う最後のトークン}{N以下のノードの最後の記号を再帰的にたどる}
```

4.3 再解析の終了判定

再解析は基本的に，プログラムの残り全体（zの最後まで）が読み飛ばされて終了するが，解析されたことのあるトークン列を，前回と同一の解析状態で解析しようとしたときに終了すればよい．これは，前回と同一のノードの同一の終端記号として，同一のトークンが入力されたとき，あるいは，現在のノードで解析しようとした非終端記号に対応する枝が既に存在し，かつそれが現在のトークンを先頭として生成されたものであるとき，を判定すればよい．

```
procedure ReParseDone: {再解析の終了判定}
  if CT.np≠Λ then begin
    if SymType(CN)=TERMINAL then begin
      if CT.np=CN and CT.i=CN.i
          then return TRUE
    end else
      if CN.children[ChildIndex(CN)]=
        TopNode(CT)
        then return TRUE
  end; return FALSE □
function TopNode(T): {トークンTに対応する最上位ノード}
```

4.4 再解析時のバックトラック

文法が LL(k) (k>1) である場合，変更の影響が，変更を受けていないxの後半の（BT より左側の）部分に及ぶことがあり，この部分列に対して別の解釈を行う必要がある．そこで，再解析時に BT より左側のトークンに最初にバックトラックしたときに，そのトークンの解析状態を無効にしてから再解析を行う．

```
function ReBack:{再解析時の最初のバックトラック}
  if CT≠BT then return FALSE
  PreToken; BT←CT; Recover; return TRUE□
```

5. むすび

本方式による再解析の時間は，プログラムの長さではなく，変更が加えられた位置の構文の深さで決まる．したがって，プログラム全体が数千行でも，数行であっても，同一の変更に対する再解析の時間はほとんど変らない．このパーサを SUN3/52M の上で言語Cで実現し，性能の測定のためにCや PASCAL のプログラムに対して，関数の分割と結合，文の挿入や削除，if-else 文と if 文あるいは if 文と while 文の相互変換などを行ったが，数十ミリ秒を越える再解析時間を要する例は見つけられなかった．このパーサは，COSMOS プログラミング・システム[6] の構造エディタ SSE[7),8)] に組み込まれ，使用されている．実際の実現は，純粋な再帰下降型ではなく，部分的にパース制御表や，バックトラックのカット機構を用いた．これらを利用して，構文エラーからの回復や，複数の箇所が変更されたプログラムの再解析も，同様に高速に行えるようになった．

参考文献

1) Celentano, A.: Incremental LR Parsers, *Acta Inf.*, Vol. 10, pp. 307-321 (1978).
2) Ghezzi, C. and Mandrioli, D.: Incremental Parsing, *ACM Trans. Program. Lang. Syst.*, Vol. 1, pp. 58-70 (1979).
3) Agrawal, R. and Detro, K. D.: An Efficient Incremental LR Parser for Grammers with Epsilon Productions, *Acta Inf.*, Vol. 19, pp. 369-376 (1983).
4) Wilcox, T. R. et al.: The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System, *Comm. ACM*, Vol. 19, No. 11, pp. 609-616 (1976).
5) Schwartz, M. D.: Incremental Compilation in Magpie, Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, *SIGPLAN Notices*, Vol. 19, No. 6, pp. 122-131 (1984).
6) 佐藤，板野：COSMOS：対話型統合的プログラミングシステム，情報処理学会コンピュータシステム・シンポジウム，pp. 115-124 (1985).
7) 佐藤，板野：構造エディタとインタプリタの統括的記述とその生成系，ソフトウェア科学会コンピュータ・ソフトウェア，Vol. 4, No. 2, pp. 39-50 (1987).
8) 佐藤，板野：COSMOS プログラミング・システムの構文指向画面エディタ SSE，ソフトウェア科学会 構造エディタに関するワークショップ，資料1-4 (1986).

(昭和 61 年8月 28 日受付)
(昭和 62 年4月 15 日採録)

佐藤　豊（正会員）

昭和35年生．昭和57年筑波大学第三学群情報学類卒業．昭和59年同大学院修士課程理工学研究科修了．昭和62年同大学院博士課程工学研究科修了．工学博士．同年電子技術総合研究所入所．在学中は，プログラミング・システムのユーザ・インタフェースおよび構成法の研究に従事．ソフトウェア科学会会員．

板野　肯三（正会員）

昭和23年生．昭和46年東京大学理学部物理学科卒業．昭和48年同大学大学院修士課程修了．昭和51年同博士課程単位取得後退学．理学博士．筑波大学計算機センタ準研究員，同大学電子・情報工学系助手，講師を経て，現在，同助教授．コンピュータアーキテクチャ，オペレーティングシステム，プログラミング言語処理系に興味を持つ．ソフトウェア科学会，IEEE，ACM 各会員．