

インクリメンタルな LR 構文解析器におけるエラー処理方式の提案

7G-5

中井 央

山下義行

中田育男

筑波大学

1 はじめに

インクリメンタルなコンパイラとは一度解析したソースプログラムに対し、再解析時には不要な再計算を避けるコンパイラのことである。

近年、プログラミングの効率をあげるためにこのようなインクリメンタルなコンパイラの研究がされてきた。これらの研究では、入力となるソースプログラムは、構文のあるいは意味的に正しいものであることが仮定されていた。しかし、実際のコーディングの過程では、エラーが発生し、エラー箇所の修正とコンパイルを繰り返している。

本研究ではこの点に着目し、エラーを含んだプログラムの修正とコンパイルの繰り返しにおいて、以前のコンパイルで得られた情報を用いた再解析方法について述べる。なお、構文解析は LR 構文解析に従うものとする。

2 準備

本研究では、LR(1) 解析によって解析木を作っていく、最初のエラー発見時に停止するとする。よって、エラー発見時には部分木の列ができている。

エラーが発生した時、ユーザはエラー発生箇所を修正するかも知れないし、それ以前を修正するかも知れない。前者の場合、これまでの解析情報を用いて解析を続けることは容易である。以降では後者の場合について述べていく。

修正前のソーステキストを

$$w = x_0 y_1 x_1 \cdots y_m x_m x_n$$

とし、修正後のソーステキストを

$$w' = x_0 y'_1 x_1 \cdots y'_m x_m x_n$$

とする。ここで x_i, y_j, y'_j はトークン列である。 x_0, x_m は ϵ かもしれないが、 $x_1 \cdots x_{m-1}$ は ϵ ではない。 x_n も ϵ ではないとする。また、 y_j, y'_j は ϵ となりうるが、同時にはない。以降では簡単のため $m = 1$ として話を進めていく。

インクリメンタルな解析では、1) y'_i を解析する準備をする、2) y'_i を解析する、3) x_i の解析をする、その際修正前の解析結果の利用を図る、を各 i について行う。

3 インクリメンタルな構文解析

1) は、解析スタックを x_{i-1} の最後のトークンをシフトしたところまで復元する。2) は 1) に基づいて、 x_i の最初のトークンをシフトするまで通常の解析を行う。以降では 3) について述べる。

x_i の解析を行った時、以前と同じ形になる部分木が含まれている場合がある。このような部分木をあらかじめ発見できれば、その部分木に該当する解析を省略することができる。以下では、このような部分木を発見する方法について述べる。

これは LR 構文解析の性質をうまく利用してやることによって行うことができる。LR 構文解析の動きを説明するためにその状況を表す配置 (configuration) を

$$(I_0 X_1 I_1 X_2 I_2 \cdots X_m I_m, a_i a_{i+1} \cdots a_n \$)$$

の形で表す。ここで第 1 要素はスタックの内容で I_k は状態、 X_k は文法記号である。第 2 要素は残り入力である。入力の終りには特別な記号 $\$$ をつけておく。 I_m が現在の状態である。

LR 構文解析の性質から以下のことがいえる。(# 1)

LR 構文解析における配置 $(\alpha X I_k, xy \$)$ と $(\beta X I_l, xz \$)$ で $I_k = I_l$ の場合を考える。ここで $\alpha = I_0 X_1 I_1 \cdots X_m I_m, \beta = I_0 X'_1 I'_1 \cdots X'_n I'_n$ の形をしているとする。

$xy \$, xz \$$ は残り入力を表しており、 $y = z = \epsilon$ の場合もある。この時、

$$(\alpha X I_k, xy \$) \vdash^* (\alpha X I_k \gamma, x' y \$) \vdash (\alpha' Y I'_k, x' y \$)$$

となるならば、後者についても

$$(\beta X I_l, xz \$) \vdash^* (\beta X I_l \gamma', x' z \$) \vdash (\beta' Y I'_l, x' z \$)$$

となる。ここで、 $\gamma = X''_1 I''_1 \cdots X''_p I''_p$ ($p \geq 0$) であり、両者の最後の遷移は $Y \rightarrow \delta X X'_1 \cdots X'_p$ の還元によるものである。

このことはすなわち、二つの文 w, w' がある時、ある時点でスタックの先頭の文法記号と状態が等しく、残り入力が等しい時、スタックの先頭にある文法記号 X を含んだ還元 $Y \rightarrow \delta X X'_1 \cdots X'_p$ が行われるまでの動作が等しいことを表している。

この性質を利用して解析の省略を行う方法をアルゴリズム A として述べる。

$$w = x_0 y_1 x_1 x_n$$

$$w' = x_0 y'_1 x_1 x_n$$

の修正部分以降の解析に入った状態では

$$(I_0, w \$) \vdash^+ (\alpha, x_1 x_n \$) \quad (\alpha \text{ は } \alpha' B I \text{ の形})$$

$$(I_0, w' \$) \vdash^+ (\beta, x_1 x_n \$)$$

となる。

ここで w における x_1 の解析結果を w' の解析に利用することを考える。それには x_1 から得られている部分木に対して (# 1) を適用すればよい。

今、 $x_1 = a_1 a_2 \cdots a_m$ ($m \geq 1$) とし、 x_n の最初のトークン b_1 を先読み記号とした時までの解析木が得られるとする。 x_1 の最初のトークン a_1 をシフトした直後の配置は

$$(\alpha a_1 I, a_2 \cdots a_m x_n \$)$$

$$(\beta a_1 I', a_2 \cdots a_m x_n \$)$$

となる。以下のアルゴリズム A はこの状態を初期状態として適用される。アルゴリズム A の中では配置は一般に

$$(\alpha X I, a_j \cdots a_m x_n \$)$$

$$(\beta X I', a_j \cdots a_m x_n \$)$$

の形であるとする。また、pop はスタックを $\alpha X I \rightarrow \alpha$ とする操作、push(YI) はスタックを $\alpha \rightarrow \alpha Y I$ とする操作である。

アルゴリズム A

An Error handling method on Incremental LR parsers

Hisashi Nakai, Yoshiyuki YAMASHITA, Ikuo NAKATA

(Univ. of Tsukuba)

```

while  $j \leq m$  do
  begin
    while  $I = I'$  do { 両スタックトップの状態が等しい }
      begin
        length := brosx(X);
        {  $X$  の親  $Y$  が元の解析木に存在するかどうか
          を返す関数  $Y$  が存在する時, すなわち
           $(\alpha XI, a_j \dots a_m x_n \$) \vdash^*$ 
           $(\alpha XI\gamma, a_k \dots a_m x_n \$) =$ 
           $(\alpha_1 \alpha_2 XI\gamma, a_k \dots a_m x_n \$) \vdash$ 
           $(\alpha_1 YI_1, a_k \dots a_m x_n \$)$ 
          の時,  $\alpha_2$  に含まれる文法記号の数 ( $\geq 0$ )
          を返し, そうでない時負の数を返す. }
        if (length  $\geq 0$ ) then
          {  $X$  を含んだ次の還元までの省略 }
          begin
             $(\beta XI, a_j \dots a_m x_n \$) =$ 
             $(\beta_1 \beta_2 XI, a_j \dots a_m x_n \$) \vdash^+$ 
             $(\beta_1 YI'_1, a_k \dots a_m x_n \$)$ 
            {  $\alpha_2$  と  $\beta_2$  に含まれる文法記号列は等しい }
            すなわち,
            for  $i := 0$  to length do
              pop;
              push( $YI'_1$ );
               $j := k$ ;
            end
          end
          else
            begin
              { 以降の構文解析は  $x_n$  を見るまですべて省略できる }
              {  $(\alpha XI, a_j \dots a_m x_n \$) \vdash^+ (\alpha XI\alpha' a_m I', x_n)$  より }
               $(\beta XI, a_j \dots a_m x_n \$) \vdash^+ (\beta XI\alpha' a_m I', x_n \$)$  (注1)
              return; { アルゴリズム A を抜ける }
            end;
          end;
        end; { while  $I = I'$  do }
      { 次のシフトまで通常の解析を行う }
       $(\alpha XI, a_j \dots a_m x_n \$) \vdash^{n_w} (\alpha' a_j I_1, a_{j+1} \dots a_m x_n \$)$ ; (注2)
       $(\beta XI, a_j \dots a_m x_n \$) \vdash^{n_w'} (\beta' a_j I'_1, a_{j+1} \dots a_m x_n \$)$ ;
    end;
  end;

```

(注1) 親となるはずのノードが無いと言うことはそこまでの解析は行われていないことになる。よって、LR 構文解析の性質により、エラーによって解析を中断したところまでは同じである。このことから、エラーによって解析を中断した時に、最後にシフトしたトークンまでの配置の復元を行って、アルゴリズム A を終了する。

(注2) $n_w, n_w' > 1$ のとき各還元でも、状態が一致するかもしれない。しかし、それを調べるのは最悪で $(n_w - 1) * (n_w' - 1)$ の比較が必要となる。また、そのような比較を行うためにはスタックを元に戻す操作も必要になる。そのような操作の効率は良くない。よって、ここでは、還元動作中は通常に解析を行い、シフトで同期をとることにする。

4 インクリメンタルな意味解析

[2] では、属性値の再計算の省略も含めた 1 パス型のインクリメンタルな構文意味解析のアルゴリズムが述べられている。今回述べたアルゴリズムも属性の再評価に関して拡張することができる。

属性評価は LR 属性文法 [3] に従っているとする。スタックトップにある文法記号が X である時、ある生成規則 " $Y \rightarrow \delta X \delta'$ " による還元までの動作が省略可能である

とする。この時、構文に関しての省略だけでなく、属性計算に関しての省略が可能であるか考える。すなわち、 δ' に関する属性評価と Y の合成属性評価が省略可能か考える。それは Y の継承属性と δ, X の各文法記号の合成属性が以前と同じであれば省略可能である。

Y の合成属性は Y の継承属性と生成規則の右辺の文法記号の合成属性に依存している。また、 δ' の各文法記号の合成属性は δ, X の各文法記号の合成属性が以前解析した時と同じであれば以前と同じである。このことから、構文の省略が可能であることがわかった時点で、 Y の継承属性と δ, X の各文法記号の合成属性が以前と同じことがわかれば良い。これらの値はこの時点ですべて得ることができる。

よって、アルゴリズム A を拡張し、1 パス型のインクリメンタルな構文意味解析に用いることができる。すなわち、アルゴリズム A の * のブロックに入る前に上に述べた属性に関する条件を満たしているかどうか調べれば良い。

5 おわりに

エラーを考慮したインクリメンタルな LR 構文解析法について、そのアルゴリズムを述べた。[2] では、修正部分を含めた部分木が一つにまとまった時点でそれ以降の解析が省略できるかどうか判定しており、その部分木を得る過程での解析の省略は考慮していない。本報告のアルゴリズムではその部分木を構成する下位の部分木についても解析の省略を図っている。

最後に実現法と効率について述べる。[2] のアルゴリズムは [1] で実現された。[1] では、簡単な実験を行いインクリメンタルな解析の有効性を確認している。今回述べたアルゴリズムはこの処理系を拡張することで比較的容易に実現できる。[1] では、属性つき解析木の各ノードは文法記号、継承属性、合成属性を要素として持っていた。継承属性は LR 属性文法の性質から、現在の LR 状態の LR マーカーの次にある非終端記号に関するものがしまわれる。合成属性は同じノードにしまわれている文法記号の合成属性である。配置を復元するに当たっては [1] ではノードに状態に関する情報をもたせず、解析木をたどること（計算）によって得ていた。これは配置の復元が頻繁に行われるものではないので計算で簡単に得られる情報は保存しない方針であったからである。今回述べたアルゴリズムでは、状態の比較を頻繁に行うので状態の計算をあらためてしなくてすむように配置での文法記号 X_i と組になっている状態 I_i を一緒にしまうことにする。

今後は、実現とその評価を行う予定である。実用的な処理系を作るためにはさらに再解析時のエラーとそこまでの情報の再利用についても考慮すべきであり、また、ソーステキストの変更箇所をインクリメンタルに見つける方法と実現した処理系との統合に関しては今後の課題である。

参考文献

- [1] 荻原一隆: ECLR 属性文法に基づくインクリメンタルな構文・意味解析の研究, 筑波大学理工学研究科修士論文 (1993).
- [2] Sassa, M.: Incremental Attribute Evaluation and Parsing Based on ECLR-attributed Grammar, Tech. Report ISE-TR-88-86, Inst. of Inf. Science, Univ. of Tsukuba (1988).
- [3] Sassa, M., Ishizuka, H. and Nakata, I.: A Contribution to LR-attributed Grammars, J. Inf. Process., Vol. 8, No.3 (1985), pp.196-206.