LR属性文法に基づいたインクリメンタルな属性評価

中井 央[†] 佐々政孝[††]
山下義行[†††] 中田育男[†††]

一度解析したプログラムの変更に対して，変更箇所から影響を受ける範囲のみの再解析を行うインクリメンタルな解析について研究が行われている．構文解析に関しては，従来からある構文解析法をもとにインクリメンタルな構文解析法が提案されてきており，また，意味解析に関しては，属性文法に基づいたインクリメンタルな属性評価法が提案されている．近年のプログラミング言語は1パス型属性文法で記述できるものも多く，この属性評価は効率が良いため，それを対象にした構文・意味解析器生成系も作成されている．しかしながら，インクリメンタルな解析においては1パス型属性文法に基づいたものは提案されていない．また，従来提案されたインクリメンタルな解析法ではエラーの存在に対してはほとんど考慮されていなかった．本論文では，LR構文解析と同時に属性評価が可能なLR属性文法に基づいたインクリメンタルな属性評価法を提案する．まず，最も簡単なアルゴリズムを述べ，次に，部分木の再利用によってエラー発生時の処理が行え，再解析の効率を向上できるように拡張したアルゴリズムを述べる．これらに基づいて，処理系を実現し，解析時間の測定を行った．前者の簡単なアルゴリズムの場合は，最初の解析に比べ，再解析時間がそれと同程度かかる場合が存在するが，後者の拡張したアルゴリズムでは部分木の再利用により，かなり効率良く解析でき，1回以上再解析をする場合は，インクリメンタルな方法の方が時間が短かくなることが分かった．

Incremental Attribute Evaluation Based on LR-attributed Grammar

HISASHI NAKAI,[†] MASATAKA SASSA,[†††] YOSHIYUKI YAMASHITA[††] and IKUO NAKATA[††]

Incremental program analysis is a method that re-analyses only the regions affected by the changes of the source program. As for incremental parsing, several methods have been proposed based on ordinary parsing methods. A couple of methods have also been proposed for incremental semantic analysis based on attribute grammars (AGs). Most modern programming languages can be described by one-pass AGs. However, no incremental methods based on one-pass AGs have been proposed. Moreover, the occurrence of errors has not been considered so far in almost all incremental analysis methods. In this paper, we propose an incremental analysis method based on LR-AG, that can evaluate attributes during LR parsing. We first show a simple algorithm, then show an extended algorithm that can reuse subtrees even in the presence of errors in the source program, which is therefore more efficient. We implemented these algorithms and measured their analysis time. In the simple incremental algorithm, re-analysis sometimes takes almost the same time as the full analysis time. We show that in the extended algorithm re-analysis is far more efficient than the simple one; if re-analysis is made more than once, the total analysis time is less than that using the non-incremental analysis.

1. はじめに

プログラム開発時の修正に対し，修正箇所から影響を受ける範囲のみを再コンパイルすることができれば開発にかかる時間的なコストを軽減できる．このようなコンパイルをインクリメンタルなコンパイルという．

インクリメンタルなコンパイラとしては構造エディタと組み合わせたものがある．構造エディタでは，ユーザがプログラムの構造に従って入力していくことによってプログラムを作成していく．これは，入力が構文に従った正しいものでなければならないため，構文的なエラーをただちに発見できるという利点はあったが，逆にユーザは，ある言語の構文的な側面を熟知して

† 筑波大学工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† 東京工業大学理学部情報科学科
Tokyo Institute of Technology
††† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

いる必要があり，通常のテキストエディタでのプログラム作成に比べて，煩わしさがともなうという欠点もあった．

構造に従った入力からではなく，プログラムの自由な変更に基づいて構文解析をインクリメンタルに行うための手法は様々提案されてきた．その場合，以前の解析結果を利用するためには解析結果を木構造などの形で保存しておく必要がある．インクリメンタルな構文解析では，解析木のうち，変更箇所から影響を受ける範囲だけを再構築することになる．LR構文解析法をもとにした方法には，文献5)，1)，13)，7)などがある．

コンパイラの（静的な）意味解析については，属性文法を用いて形式的に扱うことも一般的になってきた．属性評価を構文解析木の上で行い，その属性評価をインクリメンタルに行うものには，文献9)，14)，6)などがある．これらは，構造エディタやインクリメンタルな構文解析によって構文解析木が更新されたときに，それによって影響を受ける属性を特定し再計算するものである．

しかしながら，最近のプログラミング言語は，1パスで構文解析も意味解析も行えるよう設計されているものが多い．このため，属性文法にもある制限を設けることによって，構文解析と同時に解析木を作らずに1パスで効率良く属性評価を行える1パス型の属性文法が提案されている．従来，1パス型属性文法に基づいたインクリメンタルな解析は提案されていなかった．そこで，本論文では，1パス型の属性文法に基づいたインクリメンタルな属性評価法を提案する．

本論文で述べる方法の特徴は次のとおりである．(1) 1パス型属性文法に基づいているため，インクリメンタルな解析においても構文解析と意味解析を同時に扱うことができる．再解析のために解析木に相当するものは作成するが，それでも再解析時間は大きく短縮される．(2) 従来のインクリメンタルな構文解析法の多くは，エラーが起こった場合の処理に関しては述べていなかった．LR構文解析の途中でエラーが見つかった場合，そこまでの解析結果は部分木の列として残っている．本論文で述べる方法では，そのようなソースプログラムにエラーが存在する場合にも，部分木の再利用を行ってインクリメンタルに解析を行うことができる．(3) エラーが存在しない場合でも，部分木の再利用により，インクリメンタルな構文解析や属性評価を効率良く行うことができる．(4) アルゴリズムの面では，複数箇所の変更に対応した一般的なものを提案した．

本論文は次の構成になっている．2章では最も単純な形でのインクリメンタルなLR構文解析法を述べ，この構文解析法に従って1パス属性評価を行う方法を述べる．3章では，エラーの存在を考慮し，エラー発生時に，そこまでに得られている解析情報を再利用するために前章の方法を拡張する．4章で，処理系の実現に関して述べ，実験結果を示す．この結果は3章で行う拡張が2章で述べるアルゴリズムの最適化であることを示している．5章でまとめを述べる．

2. インクリメンタルな属性評価

まず，インクリメンタルなLR構文解析法を述べ，次に構文解析と同時に属性を評価するための条件を付け加えたインクリメンタルな属性評価法を述べる．以降ではLR(1)構文解析の場合について述べる．LR(k)の場合については，文献11)を参照されたい．

2.1 インクリメンタルなLR構文解析

本論文では，ソースプログラムとして，字句解析後のトークン列を扱うことにする．したがって以降の$x_0, x_i, y_i, z_i (1 \leq i \leq n)$は，それぞれトークン列である．入力列$w = x_0y_1x_1 \cdots y_nx_n$を構文解析した後に，$y_i$を$z_i$に置き換えた入力列$w' = x_0z_1x_1 \cdots z_nx_n$を$w$の解析結果をできる限り用いて解析することを考える．ここで，$y_i \neq z_i$であり，$n > 1$のときは，$x_j (1 \leq j \leq n-1) \neq \varepsilon$であるとする．しばらくの間，説明を簡単にするため$n = 1$として話を進める．

通常，LR解析器の状態は，解析スタックと残り入力を組にした配置（configuration）を用いて表される．インクリメンタルな構文解析および次節で述べる属性評価を行うためには，解析木を保存しておく必要がある☆．このため，ここでは解析スタックの要素を，LR状態とその状態への遷移によって生成された解析木のノードを組にしたものとする．状態I_nにおける配置は，

$$((I_0, \$)(I_1, N_{X_1}) \cdots (I_n, N_{X_n}),\ r)$$

と表すことができる．ここでI_i，N_{X_i}，rはそれぞれLR状態，解析木のノード，入力の残りの部分を表す．X_iは該当する文法記号を表している．\$は初期状態に対応する特殊なノードを表している．

$x_0y_1x_1$を解析した解析木と$x_0z_1x_1$を解析した解析木とを考えたとき，**図1**で影☆☆をつけた部分のように両方の解析木で共通の部分が存在するかもしれ

☆ インクリメンタルな構文解析のみを考えた場合には，必ずしも解析木は必要ではない[1),13)]．

☆☆ 実際には影をつけた部分とそうでない部分の境界線は複雑な図形になる．

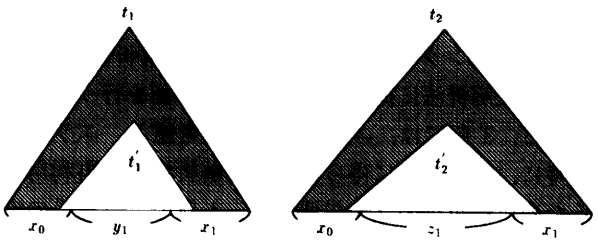

図1 $w = x_0y_1x_1$ と $w' = x_0z_1x_1$ の解析木
Fig. 1 Parse Tree of $w = x_0y_1x_1$ and $w' = x_0z_1x_1$.

ない．特に x_0 や x_1 が大きい場合，この可能性は高い．インクリメンタルな構文解析は，このような共通部分の解析を省略することが目的である．このことは，図1における t_1' の部分を t_2' で置き換えて t_2 を得ることであるともいえる．

LR構文解析は左から右，下から上へと解析木を作っていくような構文解析法である．左から右へと見ていくため，x_0 の部分の解析は w と w' で同様に行われることになる．first(u) はトークン列 u の最初のトークンを，last(u) は u の最後のトークンを表すことにすると，少なくとも last(x_0) のシフトの時点までの解析が同じことは保証できる．

z_1 の部分は変更箇所であるから通常の解析動作によって解析をする必要がある．z_1 の解析を終え，先読み記号 t が x_1 のトークンになった後のある時点で，現在の解析における配置が変更前の解析における t を先読み記号とした配置と同じであれば，それ以降の解析を省略できる．すなわち，インクリメンタルな解析を終了できる．しかしながら，すべての状態で配置を比較することは効率が悪い．実際には以下で述べる条件が成り立つ場合にインクリメンタルな構文解析を終了できる☆．これを構文照合条件と呼ぶ．

まず，条件を述べるための補助関数を定義する．

補助関数 prefix(N)

解析木のノード N に対して，以下のノードを返す．

(1) N の左の兄となるノードが存在するときはそのノード
(2) そうでなければ，左の兄を持つような N から最も近い先祖の左の兄のノード
(3) 上のいずれでもない場合は初期状態に対応する特殊なノード\$ □

例を**図2**(a)に示す．図中で，黒丸，白丸，網かけの丸は，それぞれ prefix (N) の定義の (1)，(2)，(3) の場合の N に相当する．それらからの矢印が prefix (N) を指す．

☆ このことの証明は文献11)にある．

補助関数 ancest(N)

解析木の葉ノード N に対して，N を最右の子孫とするノードのうち，最も木のルートに近い先祖のノードを返す☆☆．もし，N がどんなノードの最右の子孫ともならない場合は N 自身を返す． □

例を図2(b)に示す．

構文照合条件

入力列 $w = x_0y_1x_1 \cdots y_nx_n$ の解析結果をもとにした入力列 $w' = x_0z_1x_1 \cdots z_nx_n$ に対するインクリメンタルな構文解析において，z_i に対するインクリメンタルな構文解析を終了できるための条件：

いま，$A \rightarrow \alpha$ による還元が起こり，配置 $((I_0, \$)(I_1, N_{X_1}) \cdots (I_{m-1}, N_{X_{m-1}})(I_m, N_A), r)$ となったとする．

(1) first(r) が $x_j (j \geq i)$ 中の記号であり，
(2) 入力列 w における first(r) の1つ前の記号を c としたときの ancest(c) の文法記号が A であり，
(3) $N_{X_{m-1}}$ と prefix(ancest(c)) が同じである． □

インクリメンタルな構文解析のアルゴリズムは以下のようになる．以下で，push(ST, a) は，スタック ST へ要素 a をプッシュして a の値を返す関数であり，同様に pop(ST) はスタック ST のトップの要素をポップしてその値を返す関数である．ここでは，ソースプログラムにエラーがある場合の動作に関しては省略する．

アルゴリズム1 {インクリメンタルな構文解析}

入力：入力列 $w = x_0y_1x_1 \cdots y_nx_n$ に対する構文解析木，および w から $w' = x_0z_1x_1 \cdots z_nx_n$ への変更情報

出力：w' に対する構文解析木

```
i = 1; /* 修正箇所の数を n とする */
1. if (i=n+1) インクリメンタルな構文解析を終了する;
2. /* last(x_{i-1}) までの解析スタックを復元する.
      ここでは last(x) はトークン列 x の最後の
      トークンに該当するノードを返す関数とする.
      各ノードには該当する LR 状態も
      格納されているとする.
      ST をノードを格納するスタック,
      PS を解析スタックとする. */
   ST, PS を (空に) 初期化する;
   N = push (ST, last (x_{i-1}));
```

☆☆ 文献11)では，ancest の決め方はいろいろあるが，LR文法では左再帰を用いた記述が多いことから，このような決め方が妥当であると述べられている．本論文もそれに従うことにする．

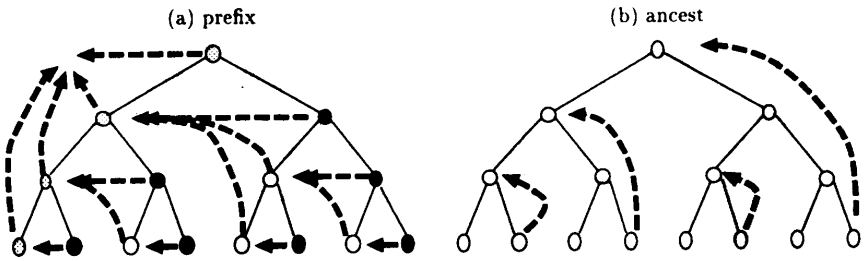

図2 補助関数 prefix と ancest
Fig. 2 Auxiliary functions, prefix and ancest.

```
count=1;
while (prefix(N) ≠ $){
    /* $ は初期状態に対応するノード */
    N = push(ST, prefix(N));
    count = count + 1;
}
/* 以下, ST を逆順にして,
   解析スタック PS を復元する. */
push(PS, $);
while (count> 0){
   push(PS, pop(ST)); ☆
   count = count − 1;
}
/* これによって配置は
   (PS, z_i x_i ···) となる. */
3. /* z_i 以降を通常の方法で解析する. */
   switch (アクション表を引いた結果){
      シフト :通常のシフト動作をする;/* ※注 */
      還元 : 通常の還元動作をする;
             /* 構文照合条件をチェックする. */
             if (構文照合条件){
                ancest(c) 以下の旧部分木を
                N_A 以下の部分木に置き換える;
                i = i + 1;
                goto 1;
             }
      受理 : 通常の終了をする;
   }
   goto 3;
```

{アルゴリズム 1 終わり}

※注 構文照合条件が成り立つまでに先読みが x_i から z_{i+1} へ移る場合があるが，その場合は字句解析側で i の値を 1 つ増やすとする．

2.2 インクリメンタルな属性評価

この節では LR 属性文法[10]に基づいて LR 構文解析と同時に属性評価を行う場合のインクリメンタルな属性評価法として，前節のアルゴリズム 1 をもとにしたものについて述べる．

LR 属性文法に基づく属性評価では解析木を作る必要はないが，ここでは，インクリメンタルな属性評価のために解析木を作りながら解析を進めていくこととし，各文法記号に付随する属性は該当する解析木のノードに付けることにする☆☆．このように各ノードに関連する属性を付加した解析木を属性つき解析木という．

インクリメンタルな属性評価を行うためには，まず，アルゴリズム 1 のうち，2. の部分を拡張し，属性評価のための属性スタックも復元しなければならない．これは，上で述べた属性つき解析木から該当する属性を取り出し，スタックに積んでいくことで実現できる．次に構文照合条件を属性のことを考慮に入れたものに拡張しなければならない．

いま，$A \to \alpha$ による還元が起こり，構文照合条件を満たしているとする．構文照合条件を満たしているため，この時点では A の合成属性の値さえ以前の属性評価時と同じであれば，古い A 以下の部分木を新しいもので置き換えられる．なぜなら，LR 属性は L 属性なので左から右への属性依存しかなく，また，N_A をルートとする部分木から外側へ属性を渡す唯一の箇所は A の合成属性を通じてであるため，A の合成属性が以前の評価時の値と同じであれば，以降の属性評価も同じになることを保証できるからである．

ここで，もし，A の合成属性が以前と異なる値に

☆ 文献 11) では，ノードには LR 状態を格納せず，文法記号（および属性）のみを格納し，スタックの復元時に LR 状態を計算によって求めている．ここでは，ノードに LR 状態も保存するとし，ここでの push(PS, pop(ST)) の部分は，上で述べた配置における解析スタックの LR 状態とノードへのポインタを組にした要素 (I_i, N_{X_i}) を PS に移している．

☆☆ 実際の実現では LR 属性の属性評価法に基づいて，合成属性は該当するノードに，継承属性は該当するノードが N のとき prefix(N) に付けられることになる．

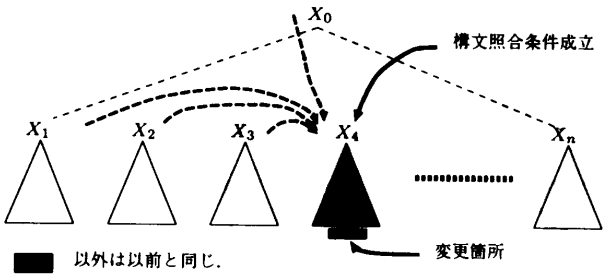

図3 属性照合条件
Fig. 3 Attribute matching condition.

なったとすると，N_A の右側の兄弟となるノードの文法記号の継承属性もしくは N_A の親となるノードの文法記号の合成属性の値に影響を及ぼす可能性がある．このため，次に構文照合条件が成り立つところ（普通は，N_A の親への還元時）まで，再解析（属性評価）を続ける必要がある．

例を用いて説明する．図 3 において，X_4 への還元によって構文照合条件が成立しているとする．この場合，X_4 をルートとする部分木のみが再解析によって新たに作成された部分である．このため，X_4 の合成属性の値は以前の値と違う可能性がある．X_0 の継承属性と X_1，X_2，X_3 の合成属性は以前と同じである．このことと属性文法が L 属性であることから，X_4 の合成属性さえ以前の値と同じであれば，$X_5,\cdots,X_n$ の属性の値および X_0 の合成属性の値は以前と同じであることを保証できる．

逆にもし，X_4 の合成属性が以前と違う値になった場合，$X_5,\cdots,X_n$ の継承属性と X_0 の合成属性のいずれかもしくはすべてが X_4 の値を用いて評価されている可能性があるので再計算の必要があり，そのため，少なくとも X_0 への還元まで解析（属性評価）を続行しなければならない．

以上から，構文照合条件を属性の評価を含めたものに拡張するには以下の条件を追加すればよい．これを属性照合条件という．

属性照合条件

構文照合条件が成り立っていて，かつ，解析スタックのトップの文法記号に関する合成属性と構文照合条件中の ancest(c) に付随している合成属性が同一である． □

アルゴリズム 2 {LR 属性文法に基づいたインクリメンタルな構文解析・属性評価}

入力：入力列 $w = x_0y_1x_1\cdots y_nx_n$ に対する属性つき解析木，および w から $w' = x_0z_1x_1\cdots z_nx_n$ への変更情報

出力：w' に対する属性つき解析木

```
i=1; /* 修正箇所の数を n とする */
1. if (i=n+1) インクリメンタルな構文解析・属性評価を終了する;
2. /* アルゴリズム 1 と同様に
      last(x_{i-1}) をシフトしたところまでの
      解析スタックを復元する.
      このとき，継承属性スタックと合成属性
      スタックも復元する.
      アルゴリズム 1 とほぼ同様であるので
      省略 */
3. /* z_i 以降を通常の方法で解析する. */
   switch (アクション表を引いた結果){
      シフト：通常のシフト動作をする;
              /* ※注 */
      還元：  通常の還元動作をする;
              /* 属性照合条件をチェックする.*/
              if (属性照合条件){
                 ancest(c) 以下の旧部分木を
                 N_A 以下の部分木に置き換える;
                 i = i + 1;
                 goto 1;
              }
      受理：  通常の終了をする;
   }
   goto 3;
```

{アルゴリズム 2 終わり}

※注 アルゴリズム 1 と同様に属性照合条件が成り立つまでに先読みが x_i から z_{i+1} へ移る場合があるが，その場合は字句解析側で i の値を 1 つ増やすとする．

3. 拡　　張

従来提案されたインクリメンタルな解析法では，入力にエラーがないことを前提としているものがほとんどであった．つまり，エラーなしに入力全体が解析されることを仮定して再解析を論じていた．しかし，インクリメンタルな解析法は，もともとインタラクティブな環境におけるユーザの修正に迅速に応えるためのものであることを考えれば，エラーが発生した場合の処理も考慮すべきであると考える．

この章では，前章で述べたアルゴリズムに対して，エラーの存在を考慮に入れた拡張を行う．この拡張は，エラーに対する処理のみでなく，前章で述べたアルゴリズムの最適化にもなっている．

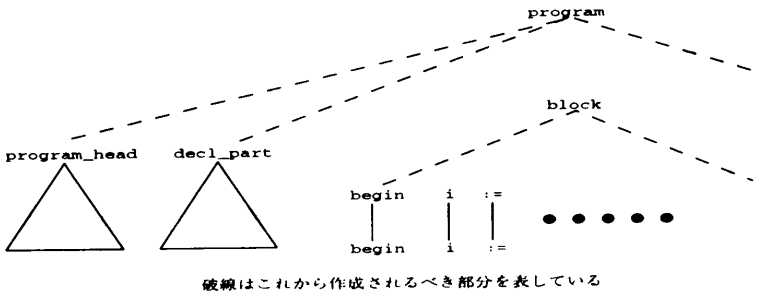

図4 LR構文解析の途中でできる部分木の列
Fig. 4 The sequence of subtrees made during LR parsing.

3.1 インクリメンタルな構文解析法の拡張

この章では，文献8)のアルゴリズムをLR属性文法に基づいて属性評価が行えるように拡張する．そこでまず，この節では，前章で述べたインクリメンタルな構文解析法に対して，部分木の再利用を行うことで，エラーを考慮した拡張を行った文献8)のアルゴリズムについて述べる．詳細は文献8)を参照されたい．

コンパイラがエラーを発見したときにユーザがそのトークン以降を修正した場合は，エラー発見時の最後にシフトされたトークンまでの解析を復元し，そこから解析を続けていけばよい．しかしながら，実際にはコンパイラがエラーを発見した箇所よりも前にあるトークンをユーザが修正する場合がある．この場合，コンパイラがエラーを発見した時点では，複数の部分的な解析木ができていることになり（図4），これらの各部分木を利用してインクリメンタルな構文解析を行うのが本方法である．また，エラーがない場合でも部分木の再利用により，より効率良くインクリメンタルな構文解析が行える．

修正箇所の直前のトークンをシフトするところまでの解析の復元は，前章で述べた補助関数prefix(N)を以下のように拡張することにより，アルゴリズム1の2.と同様にして得られる（図5）．

補助関数 prefix(N)

解析木のノード N に対して，以下のノードを返す．

(1) N の左の兄となるノードが存在するときはそのノード

(2) そうでなければ，左の兄を持つような N から最も近い先祖の左の兄のノード

(3) 上のいずれでもない場合は N を含む最も大きな部分木の1つ左に存在する部分木のルートノード（$を含む） □

上の(3)により，図5にあるように部分木のルートノードを返すようにしたところが拡張になっている．

部分木の再利用は省略可能な解析動作の発見によって行う．詳細は文献8)を参照されたい．構文規則 $X_0 \to X_1 \cdots X_m$ について，ある $X_j\,(1 \le j \le m)$ で親のノード(X_0)への還元までの解析動作を省略できるとは，$X_{j+1} \cdots X_m$（およびそれらの子孫）に関する解析動作のすべてが省略可能であることをいう．この場合，$X_{j+1}, \cdots, X_m$ をそれぞれルートとする部分木を再利用し，X_0 への還元までを行う．

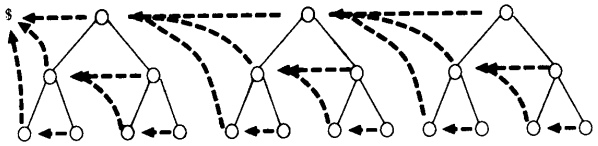

図5 prefixの拡張
Fig. 5 The extended version of prefix.

以下にアルゴリズムを示すが，修正箇所が複数の場合，部分木を再利用する際には注意が必要である．すなわち，X_0 をルートとする部分木の葉に次の修正箇所が含まれている場合には，X_0 をルートとする部分木は再利用できない．

アルゴリズム3 {部分木を再利用するインクリメンタルな構文解析}

入力：入力列 $w = x_0y_1x_1 \cdots y_nx_n$ に対する構文解析木，および w から $w' = x_0z_1x_1 \cdots z_nx_n$ への変更情報

出力：w' に対する構文解析木

```
i = 1; /* 修正箇所の数を n とする．*/
 1. if (i=n+1) インクリメンタルな構文解析を終了する;
 2. /* アルゴリズム1と同様に
       last(x_{i-1}) をシフトしたところまでの
       解析スタックを復元する．*/
 3. /* z_i 以降を通常の方法で解析する．*/
    switch (アクション表を引いた結果){
       シフト :通常のシフト動作をする;
                /* ※注1 */
```

```
t = シフトしたトークン;
if (t が x_i 中のトークンでない)
  /* t は z_i 中にある */
  goto 3;
I_1 = t をシフトしたときの LR 状態;
I_2 = 元の解析においてtをシフト
      したときの LR 状態;
N = 元の解析においてtをシフト
      したときに作ったノード;
while (I_1 = I_2) { /* 再利用可能 */
   if (N の親のノードが存在する
      場合){
      if (N の親をルートとする
         部分木に z_{i+1} 中の
         トークンが含まれている)
         /* 再利用不可 */
         break;
      親のノードへの還元までの
      解析動作を省略して
      部分木を再利用;
      I_1 = 現在の解析スタック
            においてNの親ノードの
            文法記号を読んだこと
            によって遷移した状態;
      I_2 = 元の解析において
            Nの親ノードの文法記号
            を読んだことによって
            遷移した状態;
      if (構文照合条件) {
        /* ※注2 */
        対応する旧い部分木を
        Nの親ノード以下の部分木に
        置き換える;
        i=i+1;
        goto 1;
      }
      N = Nの親ノード;
   }
   else{ /* Nの親ノードが
            存在しない場合 */
      if (前回，解析が停止したところ
         までに z_{i+1} 中のトークン
         が含まれている)
         break;
      前回，解析が停止したところま
            での部分木を再利用;
            /* ※注3 */
            break;
         }
      }
      還元： 通常の還元動作をする;
            /* 構文照合条件をチェックする．*/
            if (構文照合条件){
               ancest(c) 以下の旧部分木を
               N_A 以下の部分木に置き換える;
               i = i + 1;
               goto 1;
            }
      受理： 通常の終了をする;
   }
   goto 3;
```

{アルゴリズム 3 終わり}

※注1　アルゴリズム1と同様に構文照合条件が成り立つまでに先読みが x_i から z_{i+1} へ移る場合があるが，その場合は字句解析側でiの値を1つ増やすとする．

※注2　このwhile文の中ではすでに構文照合条件の1と2は成り立っているため，実際には条件の3をチェックすればよい．

※注3　エラーによって解析が中止された場合，エラーを発見した箇所までには，一般に複数の部分木が存在する．上のアルゴリズム中でNの親が存在しない場合は，前回の解析においてNの親となるノードの最右の子孫まで解析が進まなかったことを表している．したがって，Nより右に存在する複数の部分木を，アルゴリズム中の2で解析スタックを復元したのと同じやり方で再利用すればよい．

3.2 インクリメンタルな属性評価法の拡張

アルゴリズム3をもとに，LR属性文法に基づいて属性評価を同時に行うことを考える．

ある構文規則 $X_0 \to X_1 \cdots X_m$ に対して，LR属性では，X_{i+1} の継承属性は X_0 の継承属性および $X_1, \cdots, X_i$ の合成属性に依存し，X_{i+1} の合成属性は X_{i+1} の継承属性と X_{i+1} をルートとする部分木の葉の属性に依存する．いま，X_i $(1 \leq i \leq m)$ への還元（X_i がトークンの場合はシフト）が起こり，構文に関して X_0 への還元までの動作が省略できるとする．すなわち，$X_{i+1}, \cdots, X_m$ のそれぞれへの解析動作すべてを省略し，X_0 への還元を行うことができるとする．

このことは，$X_{i+1},\cdots,X_m$ をそれぞれルートとする部分木の葉は以前と同じであることを意味する．

この条件の下で，属性を含めて $X_{i+1},\cdots,X_m$ をルートとする部分木を再利用できるためには X_0 の継承属性と $X_1,\cdots,X_i$ の合成属性が以前と同じであればよい．すなわち，X_0 の継承属性と $X_1,\cdots,X_i$ の合成属性が以前と同じであれば，X_{i+1} の合成属性は以前と同じ値になることが保証でき，そうであれば，X_{i+2} の合成属性も以前と同じ値であることが保証でき，となり，X_m の合成属性まですべての値が以前と同じであることを保証できる．

そこで，構文についての還元が省略できるとき，属性を含めて還元が省略できる条件は次のようになる．

属性照合条件 2

構文規則 $X_0 \rightarrow X_1\cdots X_m$ に対して，X_i $(1 \leq i \leq m)$ への還元が起こったときに，構文に関して X_0 までの還元が省略可能であり，かつ X_0 の継承属性と各 X_j $(1 \leq j \leq i)$ の合成属性の値がそれぞれ以前の解析時の値と同じである． □

以下のアルゴリズム 4 では，還元の省略を x_i 中のトークンから始める．x_i 中のトークンの合成属性は以前と同じ値を持つから，属性照合条件 2 の属性値のチェックをするときは，X_0 の継承属性の値と X_j $(1 \leq j < i)$ の合成属性が以前の解析時と同じかどうかだけ調べればよい．すなわち，これらが以前と同じ値を持つ場合には X_i の合成属性は以前と同じ値を持つため，その比較をする必要はない．

アルゴリズム 4 {部分木を再利用するインクリメンタルな属性評価}

入力：入力列 $w = x_0y_1x_1\cdots y_nx_n$ に対する属性つき解析木，および w から $w' = x_0z_1x_1\cdots z_nx_n$ への変更情報

出力：w' に対する属性つき解析木

```
i = 1; /* 修正箇所の数を n とする．*/
1. if (i=n+1) インクリメンタルな構文解析・属性評
   価を終了する;
2. /* アルゴリズム 2 と同様に
      last(x_{i-1}) をシフトしたところまでの
      解析スタックと属性スタックを復元する．*/
3. /* z_i 以降を通常の方法で解析する．*/
   switch (アクション表を引いた結果){
     シフト :通常のシフト動作をする;
             t = シフトしたトークン;
             if (t が x_i 中のトークンでない)
             goto 3;
     I1 = t をシフトしたときの LR 状態;
     I2 =元の解析において t をシフト
         したときの LR 状態;
     N = 元の解析においてｔをシフト
         したときに作ったノード;
     while (I1 = I2) { /* 再利用可能 */
       if (N の親のノードが存在する
       場合){
         if (属性照合条件 2){
           親のノードまでの還元を
           省略して部分木を再利用;
           I1 = 現在の解析スタック
                においてＮの親ノー
                ドの文法記号を読んだ
                ことによって遷移した
                状態;
           I2 = 元の解析において
                Ｎの親ノードの文法記
                号を読んだことによっ
                て遷移した状態;
           if (構文照合条件の (3)){
             ancest(c) 以下の旧部分
             木を N_A 以下の部分木で
             置き換える;
             i = i + 1;
             goto 1;
           }
         }
         else break;
         N = Ｎの親ノード;
       }
       else { /* Ｎの親のノードが存在し
                 ない */
         if (前回，解析が停止したところ
             までの部分木に z_{i+1} 中の
             トークンが含まれている)
             break;
         if (Ｎの兄たちの合成属性とＮ
             の親の継承属性が以前と同じ
             値) /* ※注 1 */
             前回，解析が停止したとこ
             ろまでの部分木を再利用;
         break;
       }
     }
```

```
    還元： 通常の還元動作をする;
           /* 属性照合条件をチェックする. */
           if (属性照合条件){
              ancest(c) 以下の旧部分木を
              N_A 以下の部分木で置き換える;
              i = i + 1;
              goto 1;
           }
    受理： 通常の終了をする;
  }
  goto 3;
{アルゴリズム 4 終わり}
```

※注 1 LR 属性では，N の親の継承属性は N の親の最左の子孫がシフトされる直前に求まる．

4. 実現および実験と考察

この章では，まずインクリメンタルな構文解析・属性評価器（以下，解析器と呼ぶこともある）の実現および実現された解析器に対する実験に関して述べ，ついでその結果について考察を述べる．

4.1 実現および実験

解析器の実現は LR 属性文法に基づいた属性評価器生成系 Rie[12] をもとに行った．構文解析のドライバ部分である Rie のスケルトンファイルをインクリメンタルな解析を行うように修正し，属性値の比較などのため，Rie が解析器プログラムを出力する部分に若干手を加えることにより，インクリメンタルな属性評価器生成系を作成した．

字句解析部分については，本来，エディタにユーザのインタラクティブな変更に対処する機構を導入し，それを構文解析・属性評価器と接続すべきであるが，今回は実験を行うための簡易な字句解析器を用いた（文献 8) 参照）．

実験は，プログラミング言語の静的な意味解析を扱うものとして，Pascal-S プログラムの意味チェックを行うものと，構文照合条件は成り立つけれども属性照合条件が成り立たない例として，+，×，(，) と数値からなる簡単な算術式による計算を行うものについて，構文解析と属性評価にかかる時間の測定を行った．解析時間の測定は Silicon Graphics 社の Indy（OS: IRIX5.2, CPU: MIPS R4600, 100 MHz）上で，pixie コマンドを用いてプロファイルをとった☆．

Pascal-S 言語の意味チェッカに対しては，π の値を求めるプログラム中の while 文を置き換える実験およびクイックソートプログラム（QS）中の repeat 文を置き換える実験を行った．測定結果を**表 1** に示す．参考としてアルゴリズム 1，3 および構文解析器生成系 Bison[2] によって作成した構文解析器による構文解析時間を付す．

表 1 インクリメンタルな属性評価器による解析時間（その 1）
Table 1 Analysis time of incremental evaluator (part 1).

		(1) 最初の解析	(2) 再解析	(1)+(2)
π 計算	アルゴリズム 2	80.19	32.73	112.92
	アルゴリズム 4	89.70	2.85	92.55
	通常の Rie	52.20	52.21	104.41
	アルゴリズム 1	58.09	18.77	76.86
	アルゴリズム 3	65.19	2.16	67.35
	通常の Bison	36.27	36.28	72.55
QS	アルゴリズム 2	37.72	16.49	54.21
	アルゴリズム 4	42.18	5.27	47.45
	通常の Rie	24.65	24.73	49.38
	アルゴリズム 1	27.46	12.05	39.51
	アルゴリズム 3	30.81	3.70	34.51
	通常の Bison	17.13	17.19	34.32

ここでは解析手続きの実行時間を測定している． 単位は ms

アルゴリズム 2，4 の構文解析部分は，アルゴリズム 1，3 に基づいている．このため，属性計算を含めた場合，構文解析時間に各属性評価の時間を加算したものがインクリメンタルな属性評価にかかる時間であるといえる．文献 8) ではアルゴリズム 1，3 に対して，文法の観点から，修正位置の違いによるインクリメンタルな構文解析の効果を検討した．解析途中に起こる属性計算は各規則によってそれぞれ実行時間が違うため，一文の挿入や削除では違いが現れるかもしれないが，複合文に関しての操作では様々な属性計算が平均的に含まれていると考え，本論文では複合文への変更に対する実行時間の測定のみを行った．表中の「通常の Rie」や「通常の Bison」は，Rie や Bison によって作成された，解析木や属性つき解析木を作成しない処理系による時間を表している．

算術式の計算については，算術式を Perl を使って自動生成し，数値の一部を別の値に置き換えたもの (c1,c2) と演算子を別のものに置き換えたもの (c3) に関して測定した．測定結果を**表 2** に示す．

4.2 考 察

まず，Pascal-S プログラムの意味チェッカに対する実験について考察する（表 1）．インクリメンタルな解析をするために解析木を作らなければならないので，表 1 の「最初の解析」の時間については，通常の Rie による解析木を作成しない解析器による解析（以降では，通常の Rie によって作成された解析器による解析

☆ pixie コマンドを用いると，プロファイルするときに実行にかかったマシンインストラクション数とサイクル数が報告される．

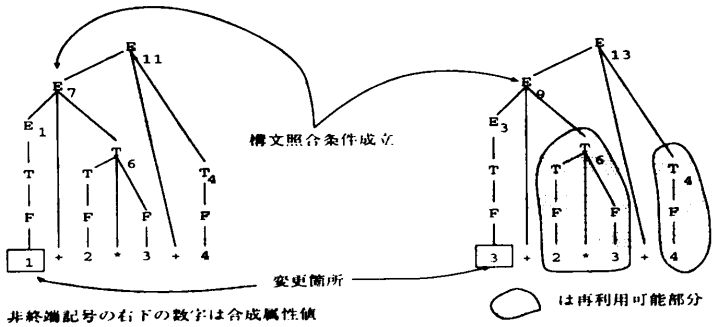

図6 c1の解析のようす
Fig. 6 Snapshot of parsing c1.

表2 インクリメンタルな属性評価器による解析時間（その2）
Table 2 Analysis time of incremental evaluator (part 2).

		(1) 最初の解析	(2) 再解析	(1)+(2)
c1	アルゴリズム2	138.12	155.13	293.25
	アルゴリズム4	155.24	6.62	161.86
	通常のRie	91.00	91.00	182.00
c2	アルゴリズム2	138.12	0.36	138.48
	アルゴリズム4	155.24	0.38	155.62
	通常のRie	91.00	91.00	182.00
c3	アルゴリズム2	138.12	155.17	293.29
	アルゴリズム4	155.24	6.66	161.90
	通常のRie	91.00	91.03	182.03

ここでは解析手続きの実行時間を測定している。 単位はms

を単に通常の解析ということにする）に比べ，本論文で述べたアルゴリズムにはオーバヘッドがかかっている．逆に表1の「再解析」の時間にのみ着目すると，本論文で述べたアルゴリズムでの解析時間は通常の解析時間よりも速く解析が行われている．実験の結果では，素朴なアルゴリズム2は，通常の解析に対して，約60%の解析時間で再解析を行い，改良したアルゴリズム4は，通常の解析に対して，約5～20%の解析時間で再解析を行うことができている．特にアルゴリズム4では，部分木の再利用が功を奏している．

表1の「(1)+(2)」の欄には最初の解析時間と再解析時間を合計したものを載せている．通常の解析によるものと比べて，アルゴリズム4の最初の解析時間と再解析時間の合計はやや短かくなっている．アルゴリズム4の再解析時間は通常の解析の1～2割であるので，再解析を1回以上行うものと考えれば，インクリメンタルな方法の方がずっと効率が良いといえる．

次に，悪条件での振舞いを調べるため，構文照合条件が成立しても属性照合条件が成り立たない例について述べる（表2）．ここで用いたのは，算術式の文法において，合成属性を利用して式の値を計算するものである．この場合，素朴なアルゴリズム2では，最悪の場合，再解析時間が最初の解析時間よりも大きくなる場合もある．

表2のc1は，ある数値を別の数値に置き換えた例である．この解析の様子を図6に示す☆．属性照合条件を調べるたびに合成属性の値が以前の値と違うため，合成属性の計算は解析木のルートまで持ち越された．属性照合条件のチェックにかかるオーバヘッドとインクリメンタルな解析のための解析木のノードの作成にかかるオーバヘッドにより，素朴なアルゴリズム2では最初の解析時間よりも再解析時間の方が大きくなってしまった．Rieによる通常の解析と比べても約1.7倍遅い解析時間となった．改良したアルゴリズム4では，他の部分式の値は再利用できる（図6中の網かけ部分）ため，再解析時間を大幅に短縮することができた．このため，再解析時間は最初の解析時間の4%程度で行うことができ，通常の解析と比べても約7%の解析時間で解析を行うことができた．

c2は，変更箇所はc1と同じであるがc1とは別の数値に置き換えた例である．しかし，変更箇所を含んだごく小さな部分式の計算結果である合成属性が以前と同じ値を持ったために再解析を始めてすぐに属性照合条件が成立し，アルゴリズム2，4のどちらも最初の解析時間に対して再解析時間は0.2%程度で終了することになった．通常の解析に比べると，約0.4%の解析時間となっている．

c3は，ある演算子を別の演算子に置き換えた例である．c1と同様に属性照合条件が解析木のルートまで成り立たなかったため，アルゴリズム2では，再解析時間が最初の解析時間よりも大きくなっている．アルゴリズム4では，部分木の再利用によって効率的に再解析を行うことができている．

c1～c3についても，表1と同様に最初の解析時間と再解析時間の合計を表2に掲載している．素朴なアルゴリズム2に関しては，再解析時間が最初の解析時間よりもかかる場合が存在するために必ずしも良い

☆ 実際には100個の数値を含む式を用いて測定した．

結果になるとはいえないが，改良したアルゴリズム4では，できる限りの再計算を省略しているため，再解析時間は通常のRieの1割以下であり，Pascal-Sの意味チェッカの場合と同様に1回以上再解析をする場合にはインクリメンタルな方法は効率が良いといえる．

一般にプログラムの作成においては複数回のコンパイルが行われるため，以上の実験から，インクリメンタルな解析方法は有効であるといえる．

算術式の例は，やや極端なものではあるが，インクリメンタルな属性評価においてアルゴリズム4がアルゴリズム2の最適化になっていることを示している．

5. おわりに

5.1 まとめ

1パス型属性文法であるLR属性文法に基づいたインクリメンタルな属性評価法として2つのアルゴリズムを提案した．1パス型属性文法を用いることで最初の解析も再解析も一般にかなり効率良く行えることが期待できる．我々の提案したアルゴリズムのうち，1番目のアルゴリズムは，ソースプログラムにエラーがない場合に構文解析と同時に属性評価を行う単純なアルゴリズムである．2番目のアルゴリズムは，部分木の再利用を行うことでエラーがあったときの修正に対しても，以前の解析情報を再利用することができるアルゴリズムである．また，エラーのない場合でも，部分木の再利用によって1番目のアルゴリズムの最適化になっている．

これらのアルゴリズムを生成系として実現し，生成された解析器の解析時間の測定を行った．どちらのアルゴリズムでも，最初の解析に要する時間に対して再解析が効率良く行えていることが分かった．また，2番目のアルゴリズム（アルゴリズム4）は，ソースプログラムにエラーのある場合にも適用できるだけでなく，部分木を再利用することによって，1番目のアルゴリズム（アルゴリズム2）より再解析に要する時間がかなり短縮されていることを確認した．2番目のアルゴリズムを用いれば，再解析を1回以上行う場合に，インクリメンタルな解析の方が解析時間が短かくなることが分かった．

本論文で述べたアルゴリズムは，統合的なプログラミング環境に適用できるほか，エラーの自動修正を行うコンパイラもしくはエラーの修正をユーザと対話的に行うコンパイラにも適用可能であると考える．すなわち，解析段階でシステムがエラーを発見したとき，適切な修正位置と修正方法（たとえば，意味エラーの原因となるトークンの特定とその置換）が，自動的または対話的に求められるならば，その修正に対して本論文で述べたアルゴリズムによる再解析を用いることにより，解析の続行が可能となる．

5.2 今後の課題

今後の課題として，プログラミング言語における宣言部分を変更した場合のインクリメンタルな属性評価法の研究があげられる．宣言文の（合成）属性をまとめたものとして得られる記号表はそれ以降の文やブロックに対する継承属性となる．このため，宣言を変更すると，それ以降の継承属性はすべて以前と違う値をとることになる．したがって，上述のアルゴリズムに従ってインクリメンタルな属性評価を行っている場合，変更箇所以降で属性照合条件は成り立たないため，すべて解析し直すことになってしまう．また，記号表の変更は，その記号表中のエントリへの参照に対しても影響を与える．たとえば，以下のようなプログラムを考える．

```
 1  program y;
 2  ...
 3  var  x : integer;
 4
 5  procedure z;
 6  var x : integer;
 7  begin
 8    ...
 9    x := 0;
10    ...
11  end;
12
13  begin
14    ...
15    x := 1;
16    ...
17  end.
```

この場合に，6行目の宣言を削除すると9行目の`x`は3行目で宣言された`x`を参照しなければならない．

このような参照関係を処理するための機構は，本論文で述べた再解析アルゴリズムとは別の問題であると考える．このような問題に対しては文献3)や4)といった研究が行われている．

今後は，インクリメンタルな属性評価機構にこのようなインクリメンタルな記号表処理を加え，統合的なシステムを作り，評価を行うことが課題である．

参考文献

1) Celentano, A.: Incremental LR Parsers, *Acta Inf.*, Vol.10, pp.307–321 (1978).
2) Donnelly, C. and Stallman, R.: *Bison Reference Manual*, FSF (1991).
3) Fritzson, P.: Incremental Symbol Processing,

Compiler Compilers and High Speed Compilation, Hammer, D. (Ed.), LNCS, Vol.371, pp.11–38, Springer-Verlag (1988).
4) Hedin, G.: Incremental Attribute Evaluation with Side-effects, *Compiler Compilers and High Speed Compilation*, Hammer, D. (Ed.), LNCS, Vol.371, pp.175–189, Springer-Verlag (1988).
5) Jalili, F. and Gallier, J.H.: Building Friendly Parsers, *ACM 9th Symposium on the Principles of Programming Languages*, pp.196–206 (1982).
6) Jalili, F.: A General Incremental Evaluator for Attribute Grammars, *Science of Computer Programming*, Vol.5, pp.83–96 (1985).
7) Larchevêque, J.-M.: Optimal Incremental Parsing, *ACM Trans. Prog. Lang. Syst.*, Vol.17, No.1, pp.1–15 (1995).
8) 中井 央，山下義行，中田育男：インクリメンタルなLR構文解析の一方式の提案とその評価，情報処理学会論文誌，Vol.37, No.3, pp.371–383 (1996).
9) Reps, T., Teitelbaum, T. and Demers, A.: Incremental Context-Dependent Analysis for Language-Based Editors, *ACM Trans. Prog. Lang. Syst.*, Vol.5, No.3, pp.449–477 (1983).
10) Sassa, M., Ishizuka, H. and Nakata, I.: A Contribution to LR-attributed Grammars, *J. Inf. Process.*, Vol.8, No.3, pp.196–206 (1985).
11) Sassa, M.: Incremental Attribute Evaluation and Parsing Based on ECLR-attributed G rammar, Technical Report ISE-TR-88-66, Inst. of Inf. Science, Univ. of Tsukuba (1988).
12) 佐々政孝，石塚治志，中田育男：1パス型属性文法に基づくコンパイラ生成系Rie，コンピュータソフトウェア，Vol.10, No.3, pp.20–36 (1993).
13) Yeh, D.: On Incremental Shift-reduce Parsing, *BIT*, Vol.23, pp.36–48 (1983).
14) Yeh, D.: On Incremental Evaluation of Ordered Attributed Grammars, *BIT*, Vol.23, pp.308–320 (1983).

(平成8年6月5日受付)
(平成8年9月12日採録)

中井 央（学生会員）
1968年生．1992年筑波大学第3学群情報学類卒業．現在同工学研究科在学中．日本ソフトウエア科学会会員．

佐々 政孝（正会員）
1948年生．1970年東京大学理学部物理学科卒業．1974年同大学院博士課程中退，東京工業大学理学部情報科学科助手．1981年筑波大学電子・情報工学系，1992年東京工業大学理学部．現在同大学情報理工学研究科数理・計算科学専攻教授．理学博士．1987～1988年ヘルシンキ大学客員研究員．プログラミング言語，属性文法，コンパイラ生成系，プログラミング環境に興味を持つ．著書に「プログラミング言語処理系」（岩波書店）がある．日本ソフトウェア科学会，ACM，IEEE各会員．

山下 義行（正会員）
1959年生．1982年大阪大学理学部物理学科卒業，日立マイクロコンピュータ・エンジニアリング（株）入社．1986年退社．1987年筑波大学大学院博士課程電子・情報工学専攻入学．1989年退学，東京大学大型計算機センター助手．1992年，筑波大学電子・情報工学系講師．1995年，同助教授．プログラミング言語，コンピュータ・グラフィックスの研究に従事．情報処理学会，日本ソフトウェア科学会会員．

中田 育男（正会員）
1935年生．1958年東京大学理学部数学科卒業．1960年同大学院修士課程修了．1960～1979年（株）日立製作所中央研究所，同システム開発研究所勤務．1979年4月より筑波大学電子・情報工学系教授．理学博士．プログラム言語，言語処理系，ソフトウェア工学などに興味を持っている．著書「コンパイラ」(産業図書)，「基礎FORTRAN」(岩波書店)，「コンパイラ」(オーム社)．ソフトウェア科学会，電子情報通信学会，ACM，IEEE各会員．