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

中井 央[†] 山下義行[††] 中田育男[††]

プログラムの開発時にほんのわずかな変更箇所に対するソースプログラム全体の再コンパイルは大きなオーバーヘッドとなる．このような場合に変更箇所から影響を受ける範囲のみを再コンパイルできれば，コンパイルにかかる時間を大幅に削減することができる．このようなコンパイルのことをインクリメンタルなコンパイルという．このインクリメンタルなコンパイルの，特に構文解析の部分については古くから研究が行われている．構文解析木を用いたインクリメンタルなLR構文解析法として佐々のものがある．佐々の方法は1パスでのインクリメンタルな属性評価も考慮しているために構文解析法はシンプルなものである．そのため，その再解析操作には，以前と同じ解析木の構築になる場合が考慮されていない．本論文ではこのような部分を再利用することによって，さらにその効率をあげるための方法を提案する．このような部分木の再利用は，従来に提案されたアルゴリズムではほとんど論じられていなかったエラーが存在する場合にも適用することができる．解析木の部分木を再利用する方法としては，Jalili の提案したものがあるが，この方法では木の再利用のために解析木をルートからトップダウンに分割していく．本方法ではLR構文解析の動作に沿ってボトムアップに再利用可能な木を見つけていくため，より効率よく部分木を再利用できる．本論文ではこの3つの方法を共通の道具を使って実現し，本方法が他の方法よりも効率がよいことを示す．

An Incremental LR Parsing Method and its Evaluation

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

If a source program is modified, then the modified program has to be recompiled as a whole even if the modification only involves a few symbols. Much time could be saved if the new code can be obtained by recompiling a limited portion of the source which contains the modification. This is the way in which so-called incremental compilers work. Several parsing methods for incremental compilation have been investigated. One of the incremental LR parsing methods is the Sassa's method. It is simple, but there may be subtrees re-constructed in the reparsing which are the same as in the previous parsing. In this paper, we propose a more efficient incremental parsing method that reuses these subtrees. The reuse of subtrees can be applied to erroneous source programs. Jalili's method is similar to ours. It also uses the subtrees to reconstruct a parse tree. His method to find the reusable subtrees is top-down, while our method to find reusable nodes is bottom-up. In our method, these are found during the LR parsing action, so it is more efficient. We implemented and evaluated these three methods, and we showed that our method is more efficient than others.

1. はじめに

プログラムの開発時にプログラム全体から見てごくわずかの変更を行った場合にも，通常はプログラム全体を与えてコンパイルし直さなければならない．コンパイラが変更箇所を認識し，変更箇所から影響を受ける範囲内だけで再コンパイルを行うことができれば，再コンパイルにかかる時間を大幅に削減することができる．このようなコンパイルのことをインクリメンタルなコンパイルという．

コンパイラの処理の中でも特に構文解析の部分を対象としたインクリメンタルな構文解析についてはかなり以前から論じられてきている[1),3),8),11),13),16),17)]．

LR構文解析をもとにしたインクリメンタルな構文解析法は，基本的には，1）変更箇所までの構文解析スタックの復元，2）通常のLR構文解析法による変更箇所以降の解析，3）変更箇所を含んだ部分木の置き換え，からなる．構文解析に続く（もしくは同時に行われる）意味解析の部分のインクリメンタルな評価もすることを考慮すれば，構文解析木（以下，解析木）

† 筑波大学工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

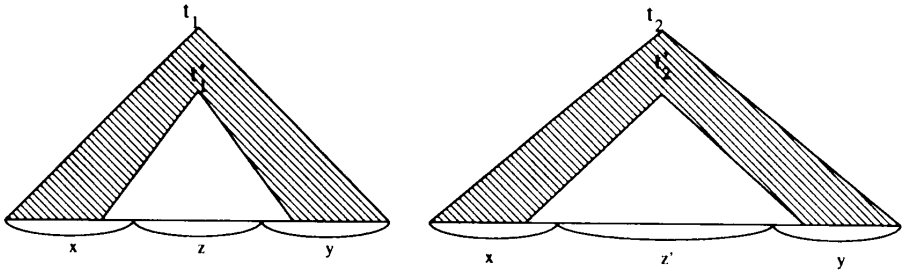

図1 入力 $w = xzy$ と $w' = xz'y$ の解析木
Fig. 1 Parse trees of $w = xzy$ and $w' = xz'y$.

を作り，それを用いてインクリメンタルに構文解析を行うことが望ましい．佐々のアルゴリズム[11),13)]はこれに該当する．

しかしながら，これらのほとんどの論文では，正しい入力の場合のみが考えられていた．すなわち，エラーの存在に関しては特に論じられていなかった．LR構文解析の場合，エラーを発見したときには，そこまでの入力によって，いくつかの部分木ができている．インクリメンタルな構文解析としては，エラーの存在をも考慮し，エラー修正後にはエラーの存在箇所までに行われた解析情報を再利用できることが望ましい．

このためには部分木を再利用する方法が必要となる．この1つにJalili の提案したものがある[8)]．これは，再利用可能な部分木を見つけ出すために木を分割し，変更箇所から新たに構成された部分と分割された部分木を合成することによって解析木を再構築する方法である．

Jalili のアルゴリズムはLR構文解析であるにもかかわらず，木の再利用は木全体から部分木へと見ていく，トップダウンなアプローチである．このことから，Jalili のアルゴリズムは部分木の再利用という点で必ずしも最適ではなく，再利用を行うための動作が逆に負荷となる場合も存在する．

本論文では，佐々のアルゴリズムをもとにエラーの存在する場合も考慮するために木の再利用を考えたアルゴリズムを示す．それはLR構文解析の自然な拡張であり，佐々の方法の改善になっている．また，Jalili のアプローチとは対照的に，木の再利用はボトムアップパーザで木を生成する動きにそったアプローチとなる．また，文法からの観点による考察と実験によって，両者に劣らぬ性能であることを示す．

2. アルゴリズム

インクリメンタルな構文解析について述べられた従来の論文では，エラーの存在についてはあまり取りあげられることはなかった．しかしながら，インクリメンタルな構文解析を含めた統合的なシステムを考える場合，そのアルゴリズムには，エラーが存在する場合も考慮されなければならない．

この章では，まずLR構文解析法を用いてインクリメンタルに構文解析を行うための一般的な考えを述べる．インクリメンタルな構文解析におけるエラーの存在について述べた後，本論文のもとになった佐々のアルゴリズム，本論文と同様に木の再利用を行うJalili のアルゴリズムを述べ，本論文でのアルゴリズムを述べる．

3つのアルゴリズムはいずれも複数カ所の修正を許すが，以下では説明を簡単にするため，修正箇所は1つであると仮定して述べる．

2.1 一般的なインクリメンタルなLR構文解析法

まず，図1を使ってインクリメンタルなLR構文解析の概略を説明する．ある入力記号列 $w = xzy$ があり，これに対して解析木を作りながら構文解析が行われているとする．この解析木を t_1 とする．部分的な解析木の葉を左から右へ並べたのものは，その解析木のへり（frontier）と呼ばれる[14)]が，ここで t_1 で z をへりの中に含むある部分木を t_1' とする．変更後の入力記号列を $w' = xz'y$ とし，解析木を作りながら構文解析するとし，その解析木を t_2 とする．t_2 の z' をへりとして含む部分木を t_2' として，図1の影をつけた2つの部分が等しくなるものを見つければ，t_1' を t_2' で置き換えることで w の解析結果から w' の解析結果を得ることができる．

LR(k) 構文解析の場合，入力記号列 w の z を w' の z' に置き換えても，入力の最初から z の最初の記号より k 個前までの解析は以前と同じであることが保証される．このことから，w' に関する解析は，z' によって影響を受け始めるところから解析を始めることができる．このことは，x のうち z から k 個前の部分までをへりとする部分木列を再利用することを意味する．

それ以降は，入力の終わりまで通常のLR構文解析

をすればよい．しかし，実際にはそれ以降の部分木列中には以前の解析結果と同じ形になる部分木が多く含まれる可能性がある．すなわち，上で述べたようにして t_1' を t_2' に置き換えてもよいことが分かれば，部分木の置換えを行うことによって，それ以降は解析をしなくてよい．すなわち，それ以降の部分木を再利用できる．

実際の構文解析では $k=1$ とするものがほとんどであるので，以降では $k=1$ として話を進める．

2.2 エラーの存在を考慮したインクリメンタルな構文解析

通常の構文解析においてもエラーの存在に対してはいろいろな対処法がある．最も簡単な方法はエラーを発見した箇所でただちに構文解析を終了するものである．より複雑な方法として，できる限りエラー処理から復帰して最後までソースファイルを読むようにする方法もある．

インクリメンタルな構文解析は，前回の解析情報を用いて，前回のソースとこれから解析しようとするソースの差分に基づいて行われる．エラーが存在した場合にはそこまでの解析を無効にすることも考えられるが，エラーを発見したところまでの解析情報をエラー修正後の解析に利用することも考えられる．

LR構文解析でエラーを発見した場合，その時点までに複数の部分木ができていることになる．そのエラーに対する修正は，必ずしもエラーを発見した位置に行われるとは限らない．その修正に対して，部分木をできる限り再利用することが，エラーが存在する場合に対応したインクリメンタルな構文解析といえる．Jalili法および本論文のアルゴリズムではそれを実現することができる．

2.3 LR構文解析の性質

以降で使われるLR(1)構文解析（以降では1を省略してLR構文解析またはLR解析という）の性質について述べる．

構文解析中の解析器の状態を表すのに次の配置を用いることにする．配置は

(スタックの内容，　残り入力)

で表す．スタックの1つの要素は文法記号 X とLR状態 I の組 XI で表す（初期状態 I_0 には対応する文法記号はない）．1回の状態の遷移を $\vdash$ で表し，その反射推移閉包を $\vdash^*$ で表すとする．

通常のLR構文解析の性質から以下の性質1を導くことができる．ここで，解析スタック $I_0X_1I_1\cdots X_mI_m$ から文法記号だけを抜き出す関数 EXG を

$$EXG(I_0X_1I_1\cdots X_mI_m)=X_1\cdots X_m$$

と定義する．

性質1

配置 $(\alpha XI_k,x)$ と $(\beta XI_l,x)$ について $I_k=I_l$ のとき，

$$(\alpha XI_k,x)\vdash^*(\alpha XI_k\gamma,x')\vdash(\alpha'YI_k',x')$$

ならば，

$$(\beta XI_l,x)\vdash^*(\beta XI_l\gamma,x')\vdash(\beta'YI_l',x')$$

が成り立つ．ただし，構文規則

$$Y\rightarrow\delta XX_1''\cdots X_p''$$

が存在し，かつ

$$\gamma=X_1''I_1''\cdots X_p''I_p''$$
$$EXG(\alpha)=EXG(\alpha')\delta$$
$$EXG(\beta)=EXG(\beta')\delta$$

が成り立つとする． □

2.4 佐々のアルゴリズム

この節では佐々のアルゴリズムについて簡単に述べる．この方法は大きく分けると以下の処理からなる．

処理1: 変更箇所以前の再利用
処理2: 変更箇所の解析
処理3: インクリメンタルな構文解析の終了

このアルゴリズムでは，解析スタックの1つの要素はLR状態とシフトまたは還元によって生成された解析木のノードへのポインタを組にしたものである．また，補助関数として以下のものを定義する．このprefix (x) の値は，構文解析時に x をスタックに積む直前にスタックのトップにあったノードへのポインタである．

定義1: prefix (x)

構文解析によって解析木が得られているとする．解析木中のノード x に対するprefix (x) の値を以下のように定義する．

1) x に左の兄となるノードが存在するときはそのノードへのポインタ
2) そうでなければ，左の兄となるノードを持つような x から最も近い先祖の左の兄となるノードへのポインタ
3) 上のいずれでもない場合はnullポインタ □

2.1節で述べたように変更箇所の直前のトークンをシフトした時点までの解析は前回のものと同様である．処理1はこれに基づき，解析木をたどりながら配置を復元する．

処理2は，変更箇所が挿入もしくは置換（異なるトークン列への置き換え）の場合，新たに加わったトーク

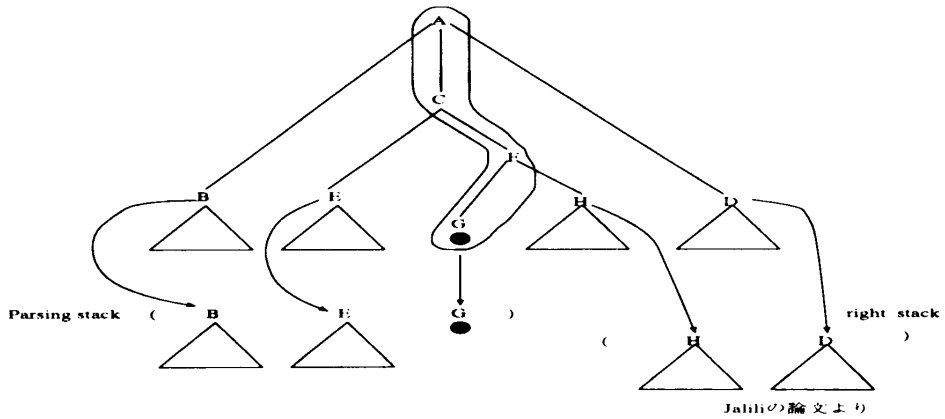

図2 Jalili のアルゴリズムにおける配置の復元
Fig. 2 Re-configuration using Jalili method.

ンに対して通常の LR 構文解析を行う．このとき，変更箇所の解析によってできたノードは以前の解析時に生成されたノードとは別に保存しておく．削除の場合はただちに処理 3 へいく．

処理 3 では，変更箇所の解析後，還元が行われるたびにその還元によって得られた部分木が図 1 の t_2' にあたるものであるかを調べる．より正確には以下の条件を調べる．

ここで，還元時の先読み記号を b とし，前回の解析における b の 1 つ手前のトークンを a としたとき，a を最右の葉として持つノードのうち最もルートノードに近いものを $f(b)$ の値として返す関数を $f(b)$ とすると，置換えのための条件は次のようになる．

構文照合条件

1) 先読み記号が前回と同じ b である．
2) 現在の解析スタックの先頭にあるポインタが指すノードの持つ文法記号と $f(b)$ の返すノードの持つ文法記号が同じものである．
3) prefix $(f(b))$ とスタックの先頭から 2 番目にある解析木のノードへのポインタが同じものである．

これらの条件をすべて満たしたとき，インクリメンタルな構文解析を終了することができる．

2.5 Jalili のアルゴリズム

この節では Jalili のアルゴリズムの概略を説明する．

Jalili の方法は木を分割し，それによってできた部分木を再利用する．図 **2** に木の分割の例をあげる．分割はあるトークン G を表す葉からその葉を含む（部分）木のルートまでのパスによって行う．このとき，このパスを境として左右に部分木列ができる．G を含めてパスの左側にできた部分木列の各ルートの文法記号を並べたものは，G をシフトした時点の解析スタックに等しい．ここでは G を含めてパスの左側にできた部分木列の各ルートの文法記号を G がその先頭となるようにスタックに積み，これを解析スタックとする．また，パスの右側の部分木列は以降で再利用される可能性があるので，G に近い側の部分木が先頭となるように別のスタックを設けて積んでおく．後者のスタックを以降では右スタックと呼ぶことにする．

Jalili のアルゴリズムは以下の処理からなる．

処理 1: 変更箇所の 1 つ前のトークンによる木の分割（配置の復元）
処理 2: 変更箇所に対する処理
処理 3: 右スタック上の部分木の再利用と木の統合

処理 1 は佐々のアルゴリズムの処理 1 と同じである．

Jalili の方法では，変更箇所は挿入もしくは削除のどちらかによるものであり，置換はこの 2 つの組合せで処理される．挿入の場合は，そのトークン列に対して解析木を作りながら通常の解析を行う．削除の場合は，まず右スタックの先頭の部分木に着目する．この部分木の葉がすべて削除されたトークンからなる場合は，この木を右スタックから削除する．しかし，削除されたトークンがこの部分木の葉の一部である場合は，削除されたトークン列の最後のトークンからその部分木のルートまでのパスで分割する．このとき，そのパスの左の部分木列は捨てられ，右の部分木列は右スタックの先頭に追加される．

変更箇所の処理が終わったら，その時点で右スタックに存在する部分木の再利用を考える．再利用できるための条件は 2.3 節の LR 構文解析の性質を用いている．すなわち，右スタックの先頭の部分木のルートを t とすると，元の解析木における prefix (t) での LR 状態と現在の解析スタックの先頭の LR 状態が同じであれば t は再利用可能となる．再利用ができないと

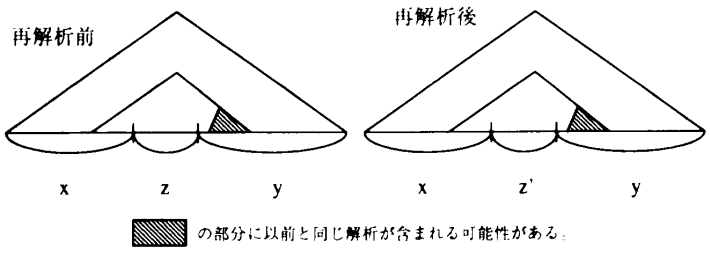

図3 冗長な再解析
Fig. 3 Redundant reparsing.

は，右スタックの先頭にある部分木のルートの文法記号を読む直前の状態が違うことを意味する．したがって，その最左のトークンを読むところから解析し直さなければならない．このため，右スタックの先頭の部分木の最左のトークンからそのルートまでのパスでその部分木を分割し，パスの右側にできた部分木列を右スタックの先頭に追加する．

最左のトークンは新たに読んだものとして以降右スタックの部分木が再利用できるかどうか調べながら解析を入力がなくなるまで続行する．

この方法ではエラーを発見した場合にも，それまでにできている部分木を対象としてインクリメンタルな構文解析を行うことができる．

2.6 本論文でのアルゴリズム

この節では，本論文で提案するアルゴリズムを述べる．

従来の論文では，入力が正しいことを前提としたアルゴリズムの提案がほとんどであった．2.2節で述べたようにエラーが存在した場合にエラー発見時までの解析情報を用いてインクリメンタルな構文解析を行うには，アルゴリズムとして部分木列に対する操作を行えなければならない．

本論文では，このようなエラーの存在を考慮したアルゴリズムを提案する．エラーの存在を考慮し，部分木を扱えるようにすることは，佐々のアルゴリズムの効率向上にもなる．すなわち，佐々のアルゴリズムでは変更箇所の解析（処理2）後，構文照合条件が成り立つまでに以前の解析と同様の解析結果になる部分が含まれることを考慮していない（**図3**）．本アルゴリズムでは，このような部分を再利用することになる．

本アルゴリズムではJalili のアルゴリズムと同様に2.3節で述べたLR構文解析における性質を用いている．ただし，先に述べたJalili のアルゴリズムとの相違点は，Jalili のアルゴリズムがトップダウンに解析木を見て部分木の再利用を試みるのに対して，ボトムアップなアプローチをとっていることである．このことによる違いは後の章で実例に従って述べる．

以下にそのアルゴリズムを記す．

（インクリメンタルなパーザのアルゴリズム）
step1: 配置の復元を行う;
step2: 状態と先読み記号から次の動作を決め，以下の還元，シフト，受理，エラーのいずれかに分岐する．
還元: 還元をする;
if 佐々の構文照合条件が成立 then
begin
木の置換をする;
goto **受理** (*1);
end;
else
goto **step2**;
endif
シフト: シフトし，次の状態 s を求める;
以前の解析で同じ記号を読んだときの状態を s' とする;
while ($s = s'$)
ノードの再利用をする (*2 別述)
goto **step2**;
受理: 解析終了
エラー: エラー処理
（インクリメンタルなパーザのアルゴリズムおわり）

シフト時のノードの再利用 (*2) について

2.3節で述べたLR構文解析の性質に照らして考えると，ここでのシフトは，X をシフトして配置が $(\alpha X I_k, x)$ になることに相当し，親ノードとそのトークンの右の弟（たち）はそれぞれ $(\alpha X I_k, x) \vdash^* (\alpha X I_k \gamma, x') \vdash (\alpha' Y I_l', x')$, $(\beta X I_l, x) \vdash^* (\beta X I_l \gamma, x') \vdash (\beta' Y I_l', x')$ の Y と γ に相当する．したがって Y への還元までに生成されるノードを再利用できる．さらに，この Y について I_k' と I_l' が等しければ，Y を含んだ還元までの動作が省略可能になる（**図4**）．

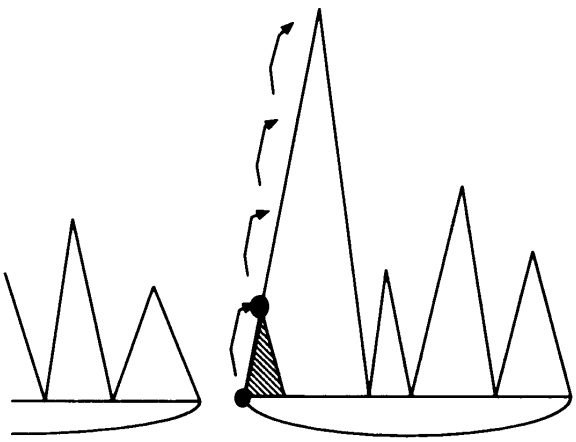

図4 還元までの省略とその繰返し適用
Fig. 4 Repetitive application of reduction elimination.

2.7 複数の修正箇所への対応

佐々およびJalilの方法は複数箇所の修正を許している．佐々の方法では,「次の修正箇所」までに木の置換が行えた場合は，次の修正箇所を解析するために前節のアルゴリズムの (*1) のところが goto step1; となる．

本方法においても複数の修正箇所を許した場合，ノードを再利用して還元が省略できるためには，再利用しようとするノードの葉の中に「次の修正箇所」に対応する部分が含まれていてはならない．したがって，木の再利用の部分は以下のようになる．

(複数の修正箇所への対応)

```
シフト: シフトし，次の状態 s を求める;
        以前の解析で同じ記号を読んだときの
        状態を s' とする;
        while (s = s')
            if (再利用するノードの葉に
                「次の修正箇所」が含まれている)
                break; /* while ループを抜ける */
            ノードの再利用をする;
        goto 動作;
```

また，エラーの存在を考慮した場合のインクリメンタルな構文解析では，未解析の部分は「次の修正箇所」と同じ扱いにすればよい．

3. 実現および実験

以上で述べた各方法によるインクリメンタルな解析器を用いて実験を行った．この章ではまず，実験に用いた解析器の実現に関して述べ，実験結果を提示する．実験に対する考察は次章で与える．

3.1 字句解析について

インクリメンタルな構文解析を行うためには構文解析器に適切な情報を与えてくれる字句解析器が必要である．インタラクティブな環境のためにインクリメンタルなコンパイルが考えられるので，このような字句解析はユーザからの入力に密接に関係し，通常はエディタと関連付けて考えられるべきものである[5),6)]．しかし，ここではパーザの動作の確認とその実行時間の測定を重視するので，字句解析器は簡略化したものを用いた．

通常，字句解析器はディスクファイルからソースファイルを読み込み，字句に切り分ける作業をする．しかし，ディスクへのアクセスは解析器内部での計算に比べ格段に時間がかかることから，パーザの実行時間の計測が正確に行えなくなると考え，ソースファイルはあらかじめプログラム中に置き，メモリからの読み出しを行うものとした．

3.2 構文解析について

前章で述べた3つのアルゴリズムに対する構文解析器をGNU bison[4)]を改造することによって作成した．

実験用の解析器は，変更を行う前後のソースファイルからその差分を抽出する部分，最初の解析を行う部分，抽出された変更箇所についてインクリメンタルな再解析を行う部分の3つからなる．

変更箇所の抽出に関してはLCS法[9)]を用いた．上で述べたように実際には，変更箇所を抽出するためにはユーザのインタラクティブな入力に対応したり，すでに存在する解析木との対応を考えたりしなければならない．LCS法を用いて変更前後のソースファイルの中身を見比べるだけでは，実際に人間が行った変更とは違うように解釈する場合があるが，今回はパーザの動作に重点をおいたため，このような不都合が生じる場合はあらかじめ与えるソースファイルを変更し，不都合が生じないようにした．

最初の解析を行う部分についてはbisonの構文解析ドライバを変更して，解析時に木のノードを生成するようにした．

再解析を行う部分は，それぞれの方法に合わせて，配置の復元を行う部分，変更箇所の解析を行う部分，構文照合条件を調べる部分もしくは木の再利用をする部分を追加したものである．

3.3 実験結果

この節では実験結果を示す．

実験は pascal-S[2)]の構文解析器を作成して，SPARCStation1 上で行った．実行時間の計測にはUNIXの gprof コマンドを用いた．

実験のための修正箇所としてはできるだけ多くのパターンを考える必要があるが，構文解析の効率は文法（構文規則）にも依存する．それらを考慮して以下の項目の修正ファイルを与えることにした．ここで複合文とはpascalにおけるbeginとendなどで括った一連の文や繰返し文などのことである．文法からの観点については次章で詳しく考察する．

1 ステートメント列への1ステートメントの挿入と削除
  1.1 ステートメント列の前への挿入と削除
  1.2 ステートメント列の間への挿入と削除
  1.3 ステートメント列の後への挿入と削除
2 複合文の直前への1ステートメントの挿入
3 複合文に対する操作

実行結果をまとめたものを**表1**に示す．「最初の解析」は最初のソース全体の解析時間であり，「再解析」は修正を行った後のインクリメンタルな再解析にかかった時間である．項目1では，**図5**のプログラムを用いて，挿入位置の違いによる効率の変化を調べている．図5中の(1)～(7)が上記の挿入と削除の操作のそれぞれの位置に対応する．挿入ではそれぞれの位置に“**writeln;**”を挿入し，削除ではあらかじめその位置に“**writeln;**”が存在するファイルから“**writeln;**”を取り除いた．項目2では，quicksortのプログラム☆中のrepeat文の直前に“**writeln;**”を挿入した．これは項目1の挿入の場合とこの場合とで，挿入した直後にくるステートメントの大きさの違いがどのような影響を与えるかを調べるためである．項目3の複合文に対する操作ではquicksortのプログラムのrepeat文をwhile文に置き換えた．

表1によれば再解析の時間はほとんどすべての場合，本論文のアルゴリズムによるものが一番小さい．Jaliliの方法では再利用はできているが効率が挿入位置に依存していると見ることができるし，佐々の方法では直後の文の大きさに依存していると見ることができる．それらの実行結果に対する詳しい考察は，実際の挿入などの操作と対応する文法との関係を考慮して次章で述べる．なお，たとえば，表1の1の1～7は同じ時間になるはずであるが，そうなっていないのは測定誤差のためである．

4. 文法から見た考察

インクリメンタルな構文解析の動作を文法の観点から検討してみる．たとえば，手続き型のプログラミング言語の文法ではプログラムは手続きや関数の並びとして記述される．

言語を文法的に見ると，ある要素の繰返しがしばしば現れる．以下に例として，ステートメントの繰返しを表す文法をG1として記す．

```
G1:
  stmt_seq : stmt_seq SEM stmt     (1)
           | stmt                  (2)
           ;
```

これはPascal-Sのyacc記述のうちのステートメントの繰返しを表す部分の記述である．1つのステートメントを表すstmtは，それよりも小さな構文要素（たとえば式など）からなる．そこで，この繰返しの一要素であるステートメント単位での修正に着目し，前章

☆ 文献7)のp.50にあるpascalによるプログラムを参考にして作成した約60行のプログラム．

表1 実験結果
Table 1 Result of execution.

		最初の解析			再解析		
		本方法	Jalili	佐々	本方法	Jalili	佐々
1		挿入					
	1	7.13	7.16	6.40	0.65	2.61	0.74
	2	7.10	6.98	6.42	0.63	2.13	0.80
	3	7.07	7.05	6.44	0.64	1.81	0.77
	4	7.11	7.11	6.44	0.64	1.53	0.77
	5	7.05	7.10	6.43	0.64	1.24	0.73
	6	7.15	7.03	6.40	0.63	0.93	0.76
	7	7.10	7.09	6.38	0.46	0.67	0.42
		削除					
	1	7.14	7.33	6.68	0.43	2.58	0.57
	2	7.12	7.26	6.63	0.47	1.99	0.63
	3	7.02	7.42	6.67	0.47	1.64	0.62
	4	7.15	7.43	6.61	0.47	1.38	0.61
	5	7.08	7.40	6.70	0.46	1.12	0.59
	6	7.13	7.34	6.60	0.44	0.80	0.61
	7	7.15	7.37	6.62	0.25	0.53	0.26
2		29.64	29.54	26.16	1.33	2.90	11.06
3		30.42	31.02	28.84	3.32	4.44	12.64

単位はms

```
program test(output);
const m = 10;
var x:integer;
begin
                -----  (1)
  x:=m;
  write(x);
                -----  (2)
  x:=m;
  write(x);
                -----  (3)
  x:=m;
  write(x);
                -----  (4)
  x:=m;
  write(x);
                -----  (5)
  x:=m;
  write(x);
                -----  (6)
  x:=m;
  write(x);
                -----  (7)
end.
```

図5 実験用ソースファイル
Fig. 5 Source file for experiment.

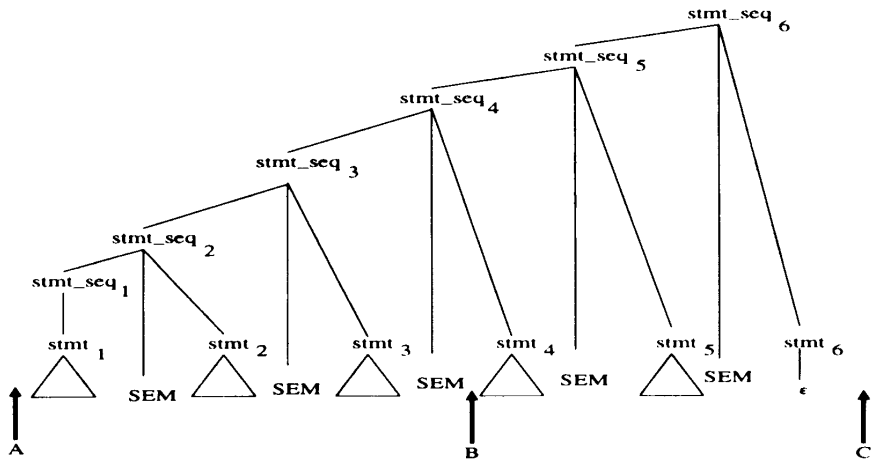

図 6 文法 G1 の解析木の例
Fig. 6 A parse tree of grammar G1.

で述べた 3 つのインクリメンタルな構文解析法での効果を考察してみる．

G1 によって生成される解析木は，たとえば**図 6** のようになる．SEM はセミコロン ';' を表している．

再解析にかかるコストを，再解析の操作の対象となるノード数 n の関数と考えて，$Cost(n)$ と表すことにする．

4.1 文の挿入

G1 によって導出される文が図 6 のように解析されていて，そこに新たに文 "x_0;" を加える場合を考える．x_0 は stmt から導出される文であるとする．まず，stmt : x_0 による還元が起こる．この stmt を stmt_0 と表すことにする．この stmt_0 について G1 の (1) もしくは (2) による還元が起こり，stmt_seq のノードが作成される．これを stmt_seq_0 と表す．その後ろにセミコロンが続いているので，セミコロンのシフトが起こる．ここまでの動作は各解析器で同じである．

ステートメント列に対してあるステートメントを挿入するのは文法的に見て次の 3 つが考えられる（図 6）．

1. ステートメント列の前にステートメントを挿入する（A の位置）
2. ステートメントとステートメントの間にステートメントを挿入する（B の位置）
3. ステートメント列の後にステートメントを挿入する（C の位置）

佐々の方法では，その後ろに続く stmt の再解析が起こる．元の木にあるこの部分を stmt_1 とすると，stmt_1 とまったく同じものであるが元の木とは別に解析木が作成される．この stmt を stmt_1' とする．この後，G1 の (1) に従って，stmt_seq: stmt_seq_0 SEM stmt_1' の還元が起こり，この時点で置き換えてもよいことになる（**図 7**）．

したがって，コストは挿入位置とは関係なく，stmt_1' に含まれるノード数 n に依存する．また，stmt_seq: stmt_seq_0 SEM stmt_1' の還元によるコストも含むため，コストは $Cost(1+n)$ と表すことができる．

次に Jalili の方法に関して検討する．Jalili の方法においては図 6 の A の位置もしくは B（もしくは C）の位置への挿入の 2 通りに分けて考えなければならない．

A の位置の場合，図 6 の解析木全体の再利用をまず試みるがその条件を満たさないため，この部分木の分割を行う．

分割は**図 8** の点線で示す部分に沿って行われる．stmt_seq のノードを除く作業と後にそれらを（もう一度）統合する作業にかかるコストはこのパス上にある stmt_seq のノード数 p に比例する☆．厳密には p に stmt_seq の子ノードの数 k をかけたものに比例する．したがって，コストは $Cost(2kp)$ となる．また，stmt_1 の最左のパスによる分割とそれらを統合する作業にかかるコストは stmt_1 の最左のパス上のノード数 q に比例する．こちらも厳密には q に各ノードの子ノード数 m をかけたものに比例する☆☆．したがって，コストは $Cost(2mq)$ となる．

☆ 実際にはシフトや還元のときにノードを作成するコストと右スタック上のノードを再利用するコストは厳密には異なるが，ここでは同様に扱うことにする．

☆☆ k は G1 の範囲内では (2) の規則によって 1 回のみ起こる還元以外は同じ数 (3) である．しかし，m はそれぞれの適用規則によって変わるので，ここではそれらの平均と考える．

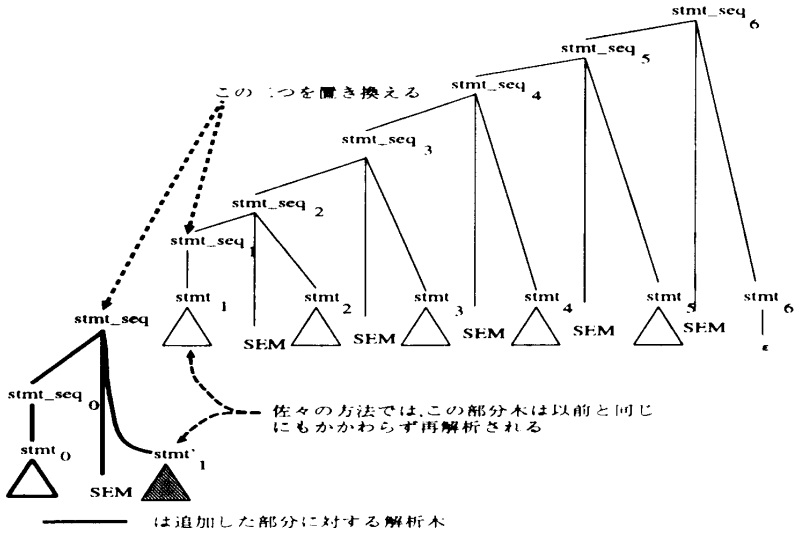

図7 ステートメント列の先頭への挿入（佐々の場合）
Fig. 7 Insertion to the head of a statement sequence (Sassa method).

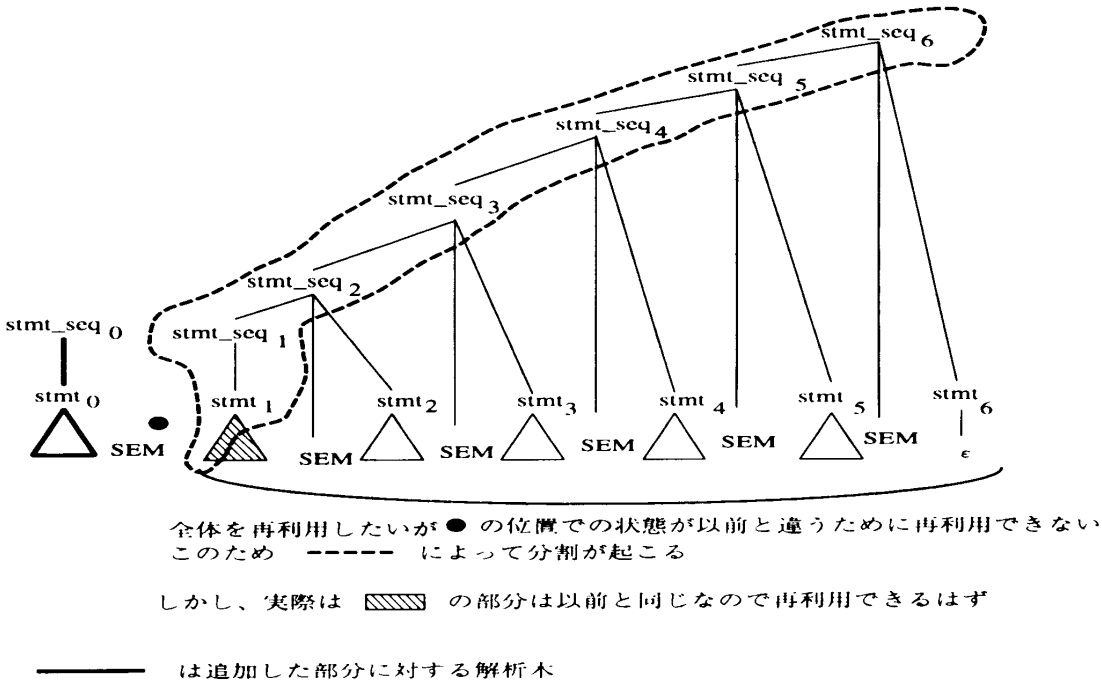

図8 ステートメント列の先頭への挿入（Jaliliの場合）
Fig. 8 Insertion to the head of a statement sequence (Jalili method).

挿入する位置がステートメント列の前ではない場合，すなわち図6のBまたはCの場合には，配置の復元によって分割されたstmt_seqの数を $p'(<p)$，同様に各ノードの持つ子ノードの数を k とすると，これらの統合のみを考えるので，コストは $Cost(kp')$ となる．

同様に本アルゴリズムに対して考察する．佐々の方法で述べた stmt_1 を再利用するのが本方法である．再利用は葉から根の方へと順にノードを見ていく．このため，stmt_1 の再利用にはその最左のトークンから stmt_1 のノードまでにあるノード数かかることになる．したがって佐々の方法での n をJaliliの方法での q に置き換えたものが本方法によるコスト $Cost(1+q)$ である．

ここで問題を簡単にするために stmt_1 が二分木からなるとすると $q=\log_2 n$ である．また，佐々法と本方法はG1の範囲内で構文解析を完結するが，Jalili法ではこの後全体のルートまで統合が行われる．すなわ

ち，ルートからG1のレベルまでの分割に対する統合が行われる．このときの統合にかかわるノード数を r とする．

したがってコストは

佐々法：$Cost(1+n)$

本方法：$Cost(1+q) = Cost(1+\log_2 n)$

Jalili法：$Cost(2kp) + Cost(2mq) + Cost(r)$

$= Cost(2kp + 2m\ \log_2 n + r)$

もしくは $Cost(kp' + r)$

となる．$q < n$ なので，本方法は佐々法よりも効率良く構文解析が行われることが分かる．

実験結果である表1の項目2や3に見られるように再利用部分（n）が特に大きい場合は，本方法およびJalili法は，佐々法よりもぬきんでて効率が良いことが分かる．

Jaliliの方法は挿入位置に効率が依存していることが分かる．それは，項目1の1，1の2，1の3と見ていくにつれ，再解析時間が短縮していることからも分かる．すなわち，効率が p あるいは p' の大きさに依存している．1の7では本方法とJalili法では先に見たG1のレベルでの解析コストはほぼ同じであるにもかかわらず，全体の解析時間に差異があることからも $Cost(r)$ はかなり大きいものであることが分かる．プログラムの規模が大きくなって，ブロックのネストの深さが増すにつれ $Cost(r)$ は大きくなる．

実際のプログラミング言語の文法では $q \ll \log_2 n$ である．たとえば，pascalなどのfor文は

```
forstmt : FOR for-expr DO stmt;
```

となっている．ここで大文字はトークンを表している．この場合，for-exprやstmtに含まれるノードの数に関係なく，forstmtに対する q は3である．通常，$kp' \geq q$ と見なせる．したがって，Jaliliの方法でのコストは本方法でのコストに対して少なくとも $Cost(r)$ の分だけ余計にかかると見ることができる．

解析の履歴をもとにして，再解析にかかわるノード数を求めた．これを**表2**に示す．

まず，項目1でのJalili法のコストに着目すると，(1)を除いて，(2)～(7)は等間隔でコストが減少している．表1の結果も同様になっており，上で述べたことが実証されている．

同様に本方法と佐々の方法についても，求めたコストは，(1)～(7)でほぼ同一であり，表1の実験結果もそれを実証している．

表2の項目2で本方法のコストは項目1のものと同じであるが，表1では項目1よりも項目2に対して，多くの時間がかかっている．これは再利用できる木の大きさの違いによるものである．木の再利用は両者で同じコストで行えるが，実際には字句解析からの入力を適切な位置まで移動する必要があり，項目2は項目1でのトークン数の約10倍の移動が必要となるため，実行時間が増大している．

表2 再解析にかかわるノード数
Table 2 Numbers of nodes for reparsing.

		中井	Jalili				佐々
		$q+1$	kp	mq	r	x	n
1	(1)	4	19	3	4	48	10
	(2)	5	33	0	4	37	10
	(3)	5	27	0	4	31	10
	(4)	5	21	0	4	25	10
	(5)	5	15	0	4	19	10
	(6)	5	9	0	4	13	10
	(7)	2	3	0	4	7	2
2		4	9	0	20	29	253
3		62	7	18	20	66	318

x は項目1の(1)では $2kp+2mq+r$（最初のstmt_seqへの還元のみ $k=1$），(2)～(7)および項目2では $kp+r$．項目3では $2kp+2mq+r$．項目3での本方法における q は解析終了までにかかわったノード数．

4.2 文を削除する場合

削除に関しては挿入時に新たに挿入した部分に対してかかるコストがないので，当然，挿入に比べて解析時間は短くなる．修正箇所以降の解析に関しては各挿入位置について前節で見てきたものが同じ位置での削除にそのままあてはまる．このことは表1の1からも見てとることができる．

4.3 右再帰を用いた構文記述の場合

今までは，左再帰を用いた文法記述G1をもとに考察してきた．同様にある文を繰り返し導出するために右再帰を用いて以下のG2のように文法を記述することができる．

```
G2:
  stmt_seq : stmt SEM stmt_seq
           | stmt
           ;
```

G2に従ってステートメント列が**図9**のように解析木を作成して解析されている場合を考える．“$stmt_2$;”の後ろに“$stmt_0$;”を挿入したとする．

佐々の方法では，構文照合条件が成り立つためには作成されていた解析木の最右の子孫まで，すなわち，$stmt_5$ の最右のトークンまでをすべて解析し直さなければならない．このことから，挿入位置より右にあるstmtの数を p とし，1つのステートメントあたり n のノードからなっていると仮定すると，“$stmt_0$;”挿入後の再解析のコストは $Cost(np)$ となる．

Jaliliの場合および本研究の場合では挿入位置の直後

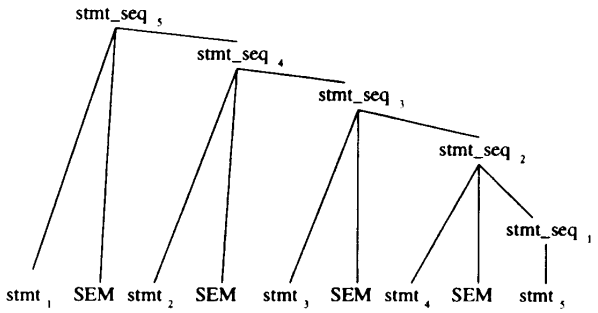

図9 右再帰を用いた構文記述による解析木の例
Fig. 9 A parse tree using syntax representation with right recursion.

の部分木を再利用できるため，ステートメント列の最右までの再解析は起こらない．Jalili の方法では $Cost(1)$ である．本方法では $stmt_3$ のルートから最左のトークンまでの木の高さ q に依存するので $Cost(1+q)$ である．

Jalili の方法では，左再帰的に文法が書かれている場合に解析の効率が挿入位置に依存するが，佐々の方法では，文法が右再帰的に書かれている場合に効率が挿入位置に依存する．本方法はどちらの書き方でも挿入位置には依存しない．

4.4 複合文に関する操作の場合

複合文に対する変更について述べる．

佐々の方法では，変更箇所以降の解析部分が大きくなればそれに従って効率も悪くなる．Pascal の begin, end を挿入する場合を考えてみると，begin の挿入の解析後 end を読むまでの間を以前と同様に再解析することになる．このため begin, end で括った範囲の大きさに比例して効率も悪くなる．表 2 の項目 3 に対する部分もこのことを表している．

本方法および Jalili の方法については，実験の項目 3 に即して考えてみる．repeat 文を while 文で括り直した場合，begin の挿入により stmt_seq を読むときの LR 状態が以前のものと変わるため，括られていたステートメント列全体の解析結果を再利用できない．

ここでは stmt の再利用までが可能であり，stmt_seq は再利用できない．すなわち，stmt_seq を読んだ直後の LR 状態が以前の解析の場合とは異なる．これらは一般的な場合にもいえる．

このことから，括られる部分に存在する stmt_seq の数を p，各 stmt_seq の子ノードの数を k，各 stmt の最左のパスに沿った高さを平均して q，各 stmt の子ノードの数を m，各 stmt に含まれるノード数を n とし，4.1 節と同様に括られる部分より上の範囲にかかわるノード数を r とすると，コストは以下のようになる．

佐々法 : $Cost(pn+p)$
本方法 : $Cost(pq+p)$
Jalili 法 : $Cost(kp+2mq+r)$

すなわち，佐々法では，begin, end で括られた部分の stmt_seq への還元を含むすべてのノード数に比例する．本方法では，各 stmt の再利用にかかるコスト $Cost(q)$ が stmt の個数あり，それらはそれぞれ stmt_seq へ還元されるため，$Cost(pq+p)$ となる．Jalili 法では begin の挿入によって最初の stmt は分割され統合され直す．これに $Cost(2mq)$ かかり，後続する各 stmt はそれぞれ stmt_seq へ還元されるため，また，括られた範囲外にかかるコストは $Cost(r)$ であるため，コストは $Cost(kp+2q+r)$ となる．

佐々法は全ノードを再構築するため，本方法や Jalili 法に比べて効率が悪いことが分かる．佐々の方法は本方法に比べて，表 1 の 2 は約 10 倍，3 では，約 4 倍遅くなっている．

括られた範囲内では，通常，k は q よりやや小さい．たとえば，G1 では k は 3 であり，表 2 の項目 1 の (1)～(7) から，本方法で 1 つの stmt を再利用する場合の q は 1～4 であることが分かる．したがって，本方法と Jalili 法を比べた場合，同程度の効率または Jalili 法が若干効率が良いといえる．

しかしながら，4.1 節でも述べたように Jalili 法は括られた範囲を越えてプログラム全体の統合まで行う．項目 3 の実験結果では，本方法の方が実行時間が短かい．また，表 2 から r がかなり大きいことが分かる．このことから，複合文に関しては本方法と Jalili 法ではほぼ同じ効率であると見なしてよい．

4.5 ま と め

文法 G1 のような形は，多くのプログラミング言語で文の列，式の列，変数の並びなどの形で頻発する．同様に多くのプログラミング言語では begin や end あるいは '(' と ')' で括るような構造を持ち，これらもプログラム中に頻発する．この章では，このような構文要素に焦点をあててインクリメンタルな構文解析の効率を検討し，提案したアルゴリズムは構文の記述の仕方にかからわらず，従来のアルゴリズムよりも，効率良く再解析を行えることを示した．

インクリメンタルな構文解析を含んだ実用的な処理系を考える場合，エラーの存在を考慮しなければならない．2 章で述べたようにエラーの存在を考慮すると木の再利用は不可欠であり，この点でも提案したアル

ゴリズムは特に有効である．

5. 関連研究

文献3)は，LR (1) 構文解析のインクリメンタルな解析法を提案している最も古いもののひとつである．解析の方法は，解析中の配置を保存しておき，再解析時には保存してある配置と現在の解析スタックによるマッチングを行うことで変更箇所に対する再解析を終了させている．再解析のためのデータ構造として，木表現を用いて解析スタックを表している．また，入力記号列と解析スタックの対応づけを行うためのデータ構造も用意している．このデータ構造の各要素は解析スタックを表した木表現のノードである．しかし，このような2つのデータ構造によって情報を保存するのは空間効率も悪く，またマッチングの効率も良くないため，その改良についても述べている．本方法やJalili の方法などで着目している木の再利用に関しては，概念的なことであるとしてそのようなことがあると述べているに過ぎず，取り扱ってはいない．

Yeh はインクリメンタルな解析に関していくつかの論文を出している．文献16)は，インクリメンタルな構文解析に関するもので，これは文献3)で用いられたデータ構造にヒントを得て，独自のデータ構造を考案したものである．その後，属性評価をインクリメンタルに行う方法を提案している[15]．ここで属性評価にあたっては文献16)などのインクリメンタルな構文解析法によって解析木が作成されていることを前提にインクリメンタルな属性評価法を述べている．属性文法を用いて意味解析をもインクリメンタルに行う場合には解析木が不可欠である．さらに文献17)では文献16)をもとにインクリメンタルな構文解析器の自動生成について述べたものである．再利用できる部分木が再解析範囲に存在することを述べているが，再利用する方法については述べられていない．

GALAXY[1]ではテキストベースのエディタでの編集結果からまず，字句としての差分を求めるkmn法を提案している．また，この方法を用いてインクリメンタルな字句解析を行っている．GALAXYでは構文解析法は再帰降下型構文解析法を用いているため，インクリメンタルな字句解析時に変更した字句から影響を受ける部分木を求めて構文解析の範囲を決定するようにしている．

本論文では構文解析器の実験のため簡易な字句解析器を用いたが，実用的なプログラミング開発環境を構築するにあたってはkmn法は有効である．

6. おわりに

本論文では，従来提案されていた解析木を用いたインクリメンタルな構文解析法の問題点をあげ，改良案を提案し，実現をして比較を行った．また，従来の論文では解析方法と実行時間については述べられていたが，どのような構文に対して有効であるかについての議論はなかった．本論文ではプログラミング言語の構文という観点から，特に繰返しを表す構文規則と変更位置について論じた．この結果，本方法は従来の方法の問題点を補い，より効率の良い構文解析が行えることが分かった．

本方法はエラーのあるプログラムにも適用可能である．エラーがあった場合，その修正を行った後の再解析にもエラー発生箇所までの解析結果の再利用が可能となる[10]．

佐々の方法はさらにLR属性文法[12]に基づいた1パス型の属性評価器をもとにインクリメンタルな属性評価方法を提案している[11]．属性評価を含めてインクリメンタルなフロントエンドを作成し評価することは今後の課題である．

参考文献

1) Beetem, J.F. and Beetem, A.F.: Incremental Scanning and Parsing With Galaxy, *IEEE Trans. Softw. Eng.*, Vol.17, No.7, pp.641–651 (1991).
2) Berry, R.: *Programming Language Translation*, Ellis Horwood (1981).
3) Celentano, A.: Incremental LR Parsers, *Acta Info.*, Vol.10, pp. 307–321 (1978).
4) Donnelly, C. and Stallman, R.: *Bison Reference Manual*, FSF (1991).
5) 原田賢一：構造エディタ，情報処理，Vol.25, No.8, pp.767–776 (1984).
6) 原田賢一編：構造エディタ，共立出版 (1987).
7) インタフェース編集部：実践ソフトウェア作法，CQ出版 (1988).
8) Jalili, F. and Gallier, J.H.: Building Friendly Parsers, *ACM Ninth Symposium on the Principles of Programming Languages*, pp.196–206 (1982).
9) 角田博保：ファイル間の相違検査法，情報処理，Vol.24, No.4, pp.514–520 (1983).
10) 中井 央，山下義行，中田育男：インクリメンタルなLR構文解析器におけるエラー処理方式の提案，第48回情報処理学会全国大会論文集，pp.99–100 (1994).
11) Sassa, M.: Incremental Attribute Evaluation and Parsing Based on ECLR-attributed Gram-

mar, Tech. Report ISE-TR-88-86 13, Inst. of Inf. Science, Univ. of Tsukuba, Tsukuba (1988).
12) Sassa, M., Ishizuka, H. and Nakata, I.: A Contribution to LR-attributed Grammars, *J. Inf. Process.*, Vol.8, No.3, pp.196–206 (1985).
13) Sassa, M. and Nakata, I.: Incremental Attribute Evaluation and Parsing Based on ECLR-attributed Grammar, 第5回日本ソフトウェア科学会全国大会論文集, pp.225–228 (1990).
14) 佐々政孝：プログラミング言語処理系，岩波書店 (1991).
15) Yeh, D.: On Incremental Evaluation of Ordered Attribute Grammars, *BIT*, Vol.23, pp.308–320 (1983).
16) Yeh, D.: On Incremental Shift-reduce Parsing, *BIT*, Vol.23, pp.36–48 (1983).
17) Yeh, D.: Automatic Construction of Incremental LR(1)-Parsers, *SIGPLAN*, Vol.23, No.3, pp.33–42 (1988).

(平成7年5月15日受付)
(平成7年12月8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．日本ソフトウェア科学会会員．

山下　義行（正会員）

1959年生．1982年大阪大学理学部物理学科卒，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．1960～79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．