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

3J－2

中井 央　山下義行　中田育男

筑波大学

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，開発者はコンパイラにその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，再コンパイルを変更箇所から影響を受ける範囲内だけで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はずである．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．解析木を用いた方法の一つに佐々のLR構文解析法[3]がある．佐々のアルゴリズムは通常のLR構文解析法を拡張した非常にシンプルなものである．これは，変更箇所から影響を受ける部分は別に解析（部分）木を作り，ある条件（構文照合条件）が成り立った時に元の解析木の該当する部分木と置き換えることによりインクリメンタルな構文解析を終了するものである．ところで，解析木を作りながら解析を行う解析法では，変更を受けていないトークン列からなる多くの部分木は以前の解析と同様に構成される．そのような部分木を再利用することに着目した方法としてJaliliのアルゴリズムがある[2]．こちらは，木を分割し，変更箇所の解析後，分割された木を合成していく方法である．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．

本研究では，佐々の方法における冗長な点を改善するため，木の再利用を考える．LR構文解析の自然な拡張である佐々の方法を改善するので，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．このことはJaliliのアプローチとは対照的になる．以降では，実際に佐々，Jalili，本研究の三つのアルゴリズムを実現し比較した結果を示す．

2 アルゴリズム

LR構文解析の動作はスタックを用いた状態遷移マシンで表すことができる．構文解析中の解析器の状態を次の配置で表す．

(スタックの内容，残り入力)

スタックの一つの要素は文法記号 X とLR状態 I の組 XI で表す．ただし初期状態には，文法記号が存在しない．

LR構文解析の性質として次のものがある．配置 $(\alpha XI_k, x)$ と $(\beta XI_l, x)$ 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．

この時，二つの配置の残り入力が同一であるから，この x を含む還元が起こるまではこの二つの配置からの遷移（それを $\vdash$ で表す）は同一である．すなわち，

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

となるならば，後者についても，

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

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構文規則 $Y \to \alpha XX_1'' \cdots X_p''$ による還元である．

本研究のアルゴリズムは，佐々の方法に上で述べた性質を加えて部分木の再利用を行うものである．以下にその概略を示す．

アルゴリズム

```
        配置の復元を行い,action に飛ぶ;
action ： 状態と先読み記号によって以下のいずれかへ飛ぶ;
reduce ： 通常の還元の処理（含佐々の構文照合条件）;
shift  ： 先読み記号のシフト;
          if（その記号が変更箇所のものではない){
            I1= 以前その記号をシフトした時の状態;
            I2= 現在の記号をシフトした後の状態;
            current= 元の解析木での同じ記号からなるノード;
            while(I1==I2){
              current の親をルートとする部分木の再利用;
              I1= その親への遷移があった時の状態;
              I2= 現在の解析スタックを用いた
                  その親への遷移による状態;
            }
          }
accept ： 受理;
error  ： エラー処理;
```

3 実験と考察

3.1 実験

ソースへの変更は削除と挿入の組合せからなる．構文解析は文法にしたがって行われるので，文法的に見た，さまざまな位置への変更に対するインクリメンタルな構文解析の効果を見るために実験を行った．例えば手続き型言語では，ステートメントの繰り返し，式の繰り返しというふうにある構文要素の繰り返しを用いてその多くが表されている．このことから，このような繰り返しの前後，および途中への変更に対して実験を行った．それは以下のような項目からなる．

なお，実験はSPARCStation1を用いて行い，実行時間の計測にはgprofコマンドを用いた．パーザはpascal-S[1]のyacc記述を用いて生成した．

1. ブロックの先頭への一ステートメント挿入
2. ブロックの中間位置への一ステートメント挿入
3. ブロックの末尾への一ステートメント挿入
4. 連続しない二ステートメント挿入
5. 式 (expression) の追加
6. ブロックの先頭の一ステートメント削除
7. ブロックの中間位置の一ステートメント削除
8. ブロックの末尾の一ステートメント削除
9. 連続しない二ステートメント削除
10. 式 (expression) の削除
11. あるステートメント列をブロックで囲む
12. ブロックで囲まれたあるステートメント列から囲みをとる

An Incremental LR Parsing Method and its Evaluation
Hisashi Nakai, Yoshiyuki YAMASHITA, Ikuo NAKATA
(Univ. of Tsukuba)

13. 11 のステートメント列を増加した
14. 12 のステートメント列を増加した

実行結果をまとめたものを図 1 に示す. 図で (1) は最初の解析の,(2) は再解析の時間を示し,(2′) は再解析の, 本研究のアルゴリズムを 1 とした時の比である. また,(2′) をグラフにしたものを図 2 に示す. 図 2 の棒グラフは三つ一組で, 各組は左から本研究,Jalili, 佐々のアルゴリズムに対するものである. 下の番号は先にあげた実験項目である.

3.2 考察

図 1 の (1) の項目でそれぞれの実行時間に差がある. これは佐々のアルゴリズムでは簡単なデータ構造を用いているのに対して, 本研究と Jalili のアルゴリズムの場合には実行速度が上がるようにデータ構造を少し複雑にしたためである.

Jalili のものは木を分割するので, 分割する量が増えると再解析時間も大きくなる. ここで用いた文法では, いわゆるステートメントの繰り返しは,

stmt_seq : stmt_seq SEMI stmt

で表されている. よって, 複数のステートメントの最初の方のステートメントに変更を行った場合は, 最後の方に行った場合よりも木を合成するのに多くの時間がかかる. このことは, 図 3 の 1,2,3,6,7,8 などに現れている. 3 と 8 では,Jalili のアルゴリズムもかなり速いが, 本研究および佐々のアルゴリズムが飛躍的に速い (特に 8) のは ϵ ルールのすぐあとに構文照合条件が成り立ったためである. 4 はあまり差異がない. これは, 与えたソースプログラムが小さかったため全体に対する修正箇所そのものが大きかったためと思われる. しかし, 部分木の再利用の効果は現れている. また,9 は 4 の逆 (追加したものを削除) であり, 新たに解析する部分が小さくなったために再解析時間も小さくなったと考えられる. ここでも, 部分木の再利用は有効である. 5 はステートメントよりもさらに小さな「式 (の部分) 」の追加である. Jalili が他の二つよりも遅いのは小さな構文要素の「式」に対応して, 木の分割もより細かくなったためと考えられる. 他の二つは 2 と同様である. 10 は 5 の逆であるが,Jalili のものは 5 と同様の理由でさほど速くはない. 他の二つは構文照合条件が直ちに成り立ったため大きな効果が得られた.

11 から 14 はブロックで囲むという処理とその逆である. 佐々のものは構文照合条件が成り立つまで通常の解析を行うがその間に構文照合条件のチェックが入るため, ソース全体に対して, ブロックの間が大きい場合はインクリメンタルな解析の効果はない. このような場合にブロック間の部分木を再利用できることが, 佐々の方法に比して本研究での大きな利点である.

本研究のアルゴリズムは, 他の二つのアルゴリズムと比べて, 同等以上の性能が出ている.

4 おわりに

今回の実験では, 小さなプログラムを用いて行ったが, わずか 2,30 行のプログラムへの 1,2 行の修正に対して, かなりの効果を示すことが出来た. ソースプログラム全体が 100 行,1000 行となったとき, 数行の修正に対してのインクリメンタルな構文解析は相当効果的であることが見込める.

また,[3] では,1 パスで属性評価も行う処理系でのインクリメンタルな構文意味解析のアルゴリズムについても述べられている. 本研究のアルゴリズムもその様に拡張する案はあるが, 属性に関する照合条件のチェックは思った以上にコストがかかりそうである. 今後はその様な処理系を実現し, 1 パスでのインクリメンタルな属性評価の有効を調べることが課題である.

参考文献

[1] Berry, R.E. : Programming Language Translation, Ellis Horwood limited, England(1981)

[2] Fahimeh Jalili, Jean H. Gallier: Building Friendly Parsers(1982), ACM Ninth Symposium on the Principles of Programming Languages,pp.196-206,1982

[3] 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).

		(1)	(2)	(2′)			(1)	(2)	(2′)
1	n	3.70	0.58	1.00	6	n	3.82	0.40	1.00
	j	3.91	1.51	2.59		j	4.12	1.34	3.34
	s	3.34	0.75	1.30		s	3.58	0.56	1.40
2	n	3.66	0.86	1.00	7	n	3.83	0.70	1.00
	j	3.80	1.04	1.21		j	4.10	0.84	1.21
	s	3.36	1.09	1.27		s	3.55	0.85	1.22
3	n	3.65	0.41	1.00	8	n	3.82	0.23	1.00
	j	3.85	0.71	1.75		j	3.98	0.53	2.24
	s	3.42	0.42	1.04		s	3.41	0.21	0.91
4	n	3.73	1.07	1.00	9	n	4.12	0.70	1.00
	j	3.79	1.23	1.15		j	4.24	0.93	1.32
	s	3.38	1.42	1.32		s	3.65	1.02	1.45
5	n	3.69	0.85	1.00	10	n	4.20	0.42	1.00
	j	3.78	1.81	2.12		j	4.30	1.40	3.33
	s	3.43	0.96	1.13		s	3.79	0.38	0.90
11	n	5.09	2.40	1.00	13	n	6.18	1.34	1.00
	j	5.16	2.57	1.07		j	6.10	1.68	1.26
	s	4.46	4.02	1.67		s	5.52	2.99	2.23
12	n	8.87	3.37	1.00	14	n	9.94	2.30	1.00
	j	8.65	3.90	1.16		j	9.76	3.05	1.32
	s	8.04	8.11	2.41		s	8.63	6.83	2.96

n= 本研究,j=Jalili,s= 佐々　　(1),(2) の単位は ms

図 1 インクリメンタルパーザの実行結果

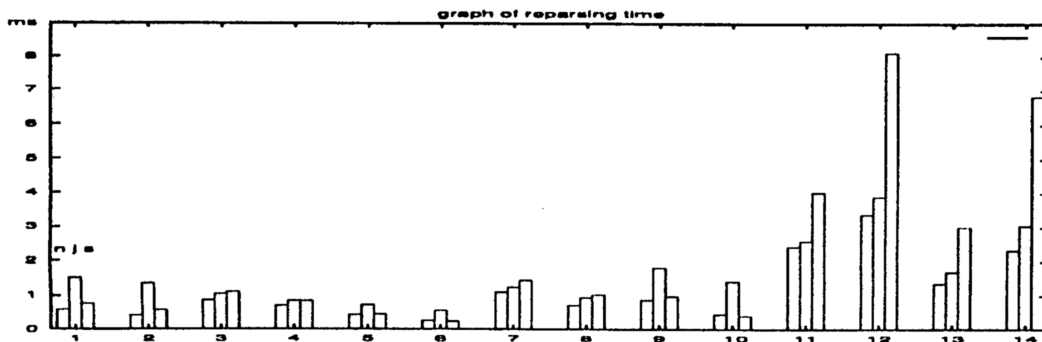

図 2 再解析時間の比較