alysis.

Agrawal and Detro[5] presented the implementation of an extension to Celentano's incremantal parsing algorithm that allows epsilon rules in the grammar. The incremental compiler used in Magpie is similar in structure to a conventional compiler[6]. Their algorithms are too expensive in time and storage requirement to be of practical use.

Yeh[7] devised an incremental shift-reduce parsing algorithm which allows a single modification in the original input. Yeh and Kastens [8] presented an incremental parsing algorithm which allows not only multiple modifications in the original input, but also epsilon production rules in the underlying LR(1) grammar.

Snelting[10] described a modification to LR parsers which allows processing of incomplete input, while at the same time building of correct abstract syntax trees. Substring recognition can be useful for noncorrecting syntax error recovery and for incremental parsing[11].

Another application for substring parsing is incremental parsing. An incremental parser builds the parse tree for the current version of its input text while it reuses the parse tree generated for the previous version as much as possible.

Beetem presented the algorithms and techniques used for incremental scanning and parsing of the Galaxy language[12].

Larchevêque[13] proposed the concept of a well-formed list of threaded trees developed in the earlier works on incremental parsing.

We discuss the conventional incremental parsing algorithms which are too expensive in time and memory space, and we present the incremental LR parsing algorithm which is more efficient than the previous ones.

2. Review of LR parser

The basic definitions, notations and conventions of [1] are used in the followings.

Let $G=(N,\Sigma,P,S)$ be an augmented LR grammar with N the set of nonterminal symbols, Σ the set of terminal symbols, P the set of production rules, and S the start symbol.

LR parser can be represented by a set of states. One state, namely S_0 is distinguished as the initial state. Each state consists of a pair of function, called the action function (denoted by *action*), and the goto function (denoted by *goto*).

For each state,

(1) *action* maps $\Sigma \cup \$$ to {shift, accept, reduce, error}

(2) *goto* maps N to {a set of states} $\cup$ {error}.

The parsing can be represented by a sequence of configurations.

A configuration Π of an LR parser is a pair $(S, x\$)$,

where,

$S = S_0\ S_1 \ldots S_m$ is the stack content with S_m on the top.

$x\$$ is the unexpended input.

Given a LR grammar G and an LR parser for G, for each sentence $z \in L(G)$, there is a unique sequence of configuration, called a parse sequence $\Pi = \Pi_0 \Pi_1 \ldots \Pi_n$ such that $\Pi_0 = (S_0, z\$)$, $\Pi_n = (S_0 S_f, \$)$, where S_0 is the initial state, S_f is the state such that $action(S_f, \$) = \text{accept}$, $\Pi_i \vdash \Pi_{i+1}\ \forall i,\ 0 \le i \le n$.

Example 1. Let G be the following grammar.

(1) $E \rightarrow E - T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow n$

The LR parsing table for grammar G is shown in (Fig. 1).

STATE	action						goto		
	n	-	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

(Fig. 1) LR parsing table for G

3. An incremental parsing

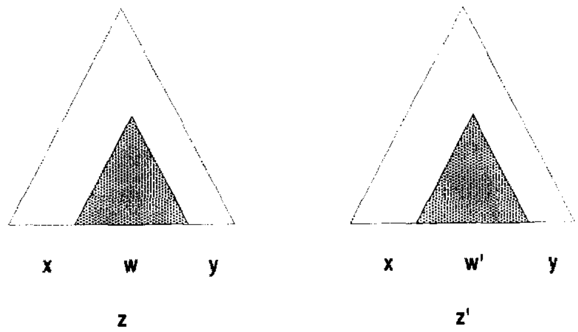
3.1 The basic data structure

Celentano[4] proposed the incremental parsing as follows.

Let z=xwy be a string generated by a grammar G, and let z′=xw′y be the string modified by substituting w′ for w (w′≠w, z′∈L(G)). Let the parse sequence $\Pi=\Pi_0\Pi_1\ldots\Pi_n$ be that associated with z, and the parse sequence $\Pi'=\Pi_0'\Pi_1'\ldots\Pi_m'$ be that associated with z′, where $\Pi_0=(S_0,z\$)$, $\Pi_0'=(S_0,z'\$)$, $\Pi_n=\Pi_m'=(S_0S_f,\$)$.

In terms of the corresponding parse trees, the purpose of the incremental parsing algorithm is to find the smallest subtree of the parse tree for z which must be reshaped in order to obtain the parse tree for z′[3].

It is clear that the terminal frontier of this subtree must include the string w; in general a reanalysis of some part of y needs to be performed, while the analysis up to the complete scanning of x remains unchanged (Fig. 2).



(Fig. 2) Parse Tree z and z′

In terms of the parse sequences this is equivalent to finding which part of Π must be recomputed to obtain Π′.

Given the two parse sequences Π and Π′, there are two indices p and q in the following algorithm;

(1) If $\Pi_i=(S_i,z\$)$ and $\Pi_i'=(S_i',z'\$)$ then $S_i=S_i'$ $\forall i,\ 0\le i\le p$

(2) $\Pi_{n-j}=\Pi_{m-j}'$ $\forall j,\ 0\le j\le q$

(3) no other indices p′>p and q′>q can satisfy the conditions (1) and (2).

The condition (1) and (2) would be too expensive to compute and store the parse sequence of configuration.

Example 2. Consider the grammar G and the two sentences z=(n-n)-(n-n) and z′=(n-n)*(n-n). We take x=(n-n), y=(n-n), w=- and w′=*. The parse sequence Π and Π′ are shown in (Fig. 3). We have p=28 and q=27; in fact $\Pi_{29}=\Pi_{28}'$, p and q are the largest values which satisfy the condition (1) and (2). Thus, the above algorithm requires 59 parsing steps.

stack	input	parse sequence	stack	input	parse sequence
S_0	(n-n)-(n-n)$	Π_0	S_0	(n-n)*(n-n)$	Π_0'
S_0S_4	n-n)-(n-n)$	Π_1	S_0S_4	n-n)*(n-n)$	Π_1'
$S_0S_4S_5$	-n)-(n-n)$	Π_2	$S_0S_4S_5$	-n)*(n-n)$	Π_2'
$S_0S_4S_3$	-n)-(n-n)$	Π_3	$S_0S_4S_3$	-n)*(n-n)$	Π_3'
$S_0S_4S_2$	-n)-(n-n)$	Π_4	$S_0S_4S_2$	-n)*(n-n)$	Π_4'
$S_0S_4S_8$	-n)-(n-n)$	Π_5	$S_0S_4S_8$	-n)*(n-n)$	Π_5'
$S_0S_4S_8S_6$	n)-(n-n)$	Π_6	$S_0S_4S_8S_6$	n)*(n-n)$	Π_6'
$S_0S_4S_8S_6S_5$	)-(n-n)$	Π_7	$S_0S_4S_8S_6S_5$	)*(n-n)$	Π_7'
$S_0S_4S_8S_6S_3$	)-(n-n)$	Π_8	$S_0S_4S_8S_6S_3$	)*(n-n)$	Π_8'

$S_0S_4S_8S_6S_9$	)-(n-n)$	Π_{9}	$S_0S_4S_8S_6S_9$	)*(n-n)$	Π_{9}'
$S_0S_4S_8$	)-(n-n)$	Π_{10}	$S_0S_4S_8$	)*(n-n)$	Π_{10}'
$S_0S_4S_8S_{11}$	-(n-n)$	Π_{11}	$S_0S_4S_8S_{11}$	*(n-n)$	Π_{11}'
S_0S_3	-(n-n)$	Π_{12}	S_0S_3	*(n-n)$	Π_{12}'
S_0S_2	-(n-n)$	Π_{13}	S_0S_2	*(n-n)$	Π_{13}'
S_0S_1	-(n-n)$	Π_{14}	$S_0S_2S_7$	(n-n)$	Π_{14}'
$S_0S_1S_6$	(n-n)$	Π_{15}	$S_0S_2S_7S_4$	n-n)$	Π_{15}'
$S_0S_1S_6S_4$	n-n)$	Π_{16}	$S_0S_2S_7S_4S_5$	-n)$	Π_{16}'
$S_0S_1S_6S_4S_5$	-n)$	Π_{17}	$S_0S_2S_7S_4S_3$	-n)$	Π_{17}'
$S_0S_1S_6S_4S_3$	-n)$	Π_{18}	$S_0S_2S_7S_4S_2$	-n)$	Π_{18}'
$S_0S_1S_6S_4S_2$	-n)$	Π_{19}	$S_0S_2S_7S_4S_8$	-n)$	Π_{19}'
$S_0S_1S_6S_4S_8$	-n)$	Π_{20}	$S_0S_2S_7S_4S_8S_6$	n)$	Π_{20}'
$S_0S_1S_6S_4S_8S_6$	n)$	Π_{21}	$S_0S_2S_7S_4S_8S_6S_5$	)$	Π_{21}'
$S_0S_1S_6S_4S_8S_6S_5$	)$	Π_{22}	$S_0S_2S_7S_4S_8S_6S_3$	)$	Π_{22}'
$S_0S_1S_6S_4S_8S_6S_3$	)$	Π_{23}	$S_0S_2S_7S_4S_8S_6S_9$	)$	Π_{23}'
$S_0S_1S_6S_4S_8S_6S_9$	)$	Π_{24}	$S_0S_2S_7S_4S_8$	)$	Π_{24}'
$S_0S_1S_6S_4S_8$	)$	Π_{25}	$S_0S_2S_7S_4S_8S_{11}$	$	Π_{25}'
$S_0S_1S_6S_4S_8S_{11}$	$	Π_{26}	$S_0S_2S_7S_{10}$	$	Π_{26}'
$S_0S_1S_6S_3$	$	Π_{27}	S_0S_2	$	Π_{27}'
$S_0S_1S_6S_9$	$	Π_{28}	S_0S_1	$	$\Pi_{28}'=\Pi_{29}$
S_0S_1	$	Π_{29}			

(Fig. 3) Parse sequences for *G*

3.2 The tree structure

An incremental parsing algorithm requires that the result of the preceding analysis of the sentence be retained[4].

Celentano[4] proposed the following tree structure to save the parse sequence.

The stack is represented by a tree, whose nodes are labeled with states, the root contains the initial state S_0. The input is represented by a sequence of tokens. Associated with each token is an ordered list of pointers that point to the nodes of the tree.

Each pointer represents a configuration. The input token to which the pointer is attached is the beginning of the unexpended input. The stack component is represented by the path through the tree from the root to the node pointed.

Algorithm 1. A parsing step.

input : a tree structure and an input list z
output : the same structure updated to include the next configuration
method : let TOP be reference to node Q of the tree such that the path from the root to Q spells out the actual stack.

Let i be a reference to the incoming symbol z_i in the input sequence; the actual configuration is then given by the last pointer in the list attached to z_i.

The cases *action*(S,z_i)=error or accept are obvious; we shall illustrate the shift and reduce cases:

case 1: *action*(S,z_i)=shift
Let S′=*goto*(S,z_i) the next state. Look at the sons of the node Q: if there is a son labeled S′, then let TOP point to it, otherwise append a new son Q′ to Q, and let TOP point to it; advance i to the next input symbol, and associate with this new symbol z_i a pointer to the same node referenced by TOP.

case 2: *action*(S,z_i)=reduce p, and production p is A→α. Back up on the tree from the node Q|α| levels, call Q′ the node so reached, and suppose it is labeled S′; let S″=*goto*(S′,A) the next state.
As in the case of shift, look at the sons of Q′; if there is one labeled S″, then let TOP point to it, otherwise append a new son to Q′, label it S″ and let TOP point to it.
Append to the list of pointers associated with z_i a new item, and let it reference the same node referenced by TOP.

Algorithm 2. Incremental LR parsing.

input : two input sequences z and z′, and a tree structure for z.
output : a tree structure representing a parse sequence for z′
method : (1) Let i and j be references to the items of the input lists for z and z′ such that z_i=FIRST(wy$) and z_j'=FIRST(w′y$). Let TOP be equal to the first pointer appen-

ded to the item containing z_i.

Repeat step 2 until j is advanced to the item containing z_j' =FIRST (y$).

Then let i reference the corresponding symbol of z, and go to step 3.

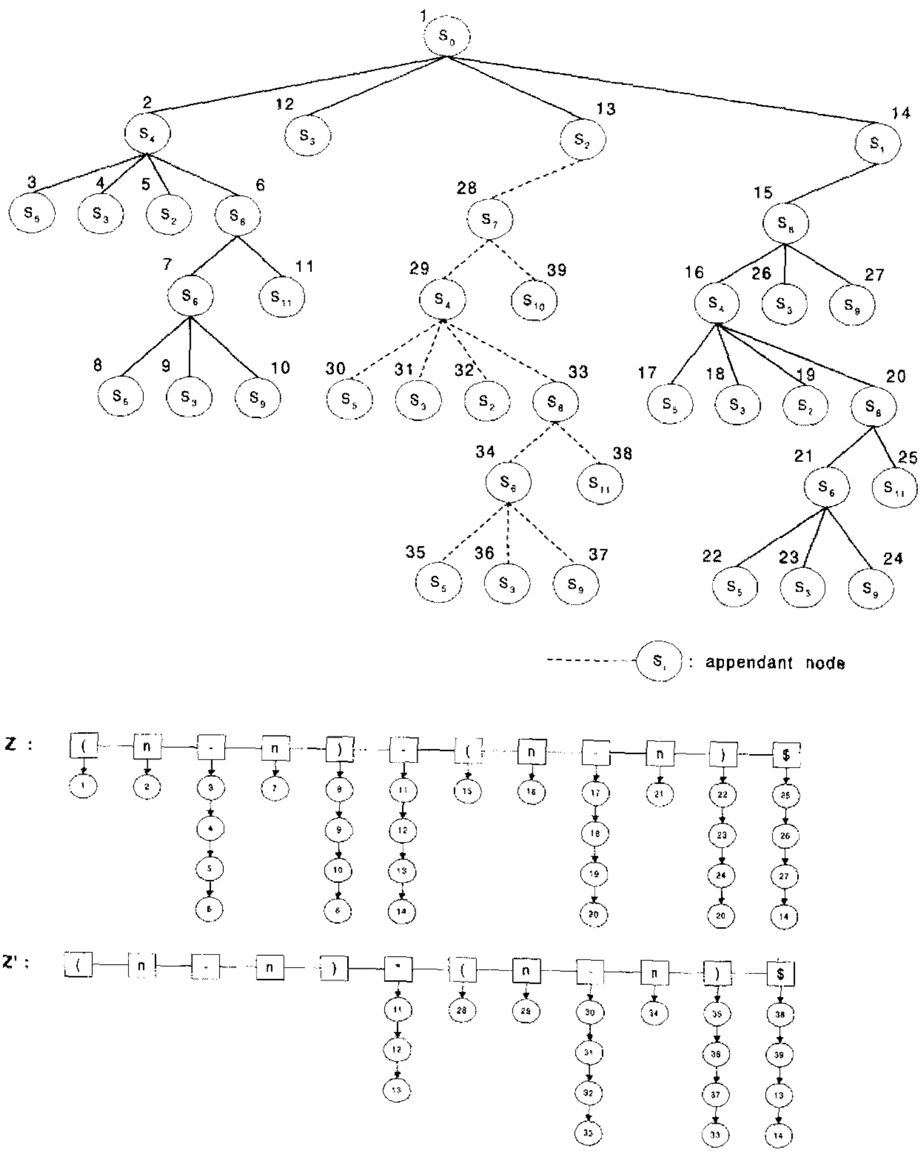
(2) Perform a parsing step on z′ as described in algorithm 1.

(3) Let P be the value of the last pointer appended to z′. If there exists a pointer associated to the item labeled z_i which is equal to P go to step 5, otherwise go to step 4.

(4) Perform a parsing step on z′, as described in algorithm 1: if j is advanced then advance i to the next item too. Go back to step 3.

(5) Report success of the match and halt the parsing.

Example 3. For Example 2, the tree structures using the Algorithm 1 and Algorithm 2 is shown in (Fig. 4)



(Fig. 4) Tree structure for G

4. An improving incremental LR parsing algorithm

4.1 Extended LR parsing table

Using the extended LR parsing table which allows grammar symbol($N \cup \Sigma$) as the input symbols, we represent the improving incremental LR parsing algorithm.

The conventional parsing table consists of two parts, a action function *action* and a goto function *goto*. The extended LR parsing table consists of one part, a action function *action* alone.

The program driving the LR parser behaves as follows. It determines S_m, the state on top of the stack, and the input string X_i which is grammar symbol. It consults *action*[S_m,X_i], the parsing action table for state S_m and input X_i.

The extended LR parsing table for G of Example 1 is shown in (Fig. 5).

STATE	*action*								
	n	+	*	(	)	E	T	F	$
0	s5			s4		s1	s2	s3	
1		s6							acc
2		r2	s7		r2				r2
3		r4	r4		r4				r4
4	s5			s4		s8	s2	s3	
5		r6	r6		r6				r6
6	s5			s4			s9	s3	
7	s5			s4				s10	
8		s6			s11				
9		r1	s7		r1				r1
10		r3	r3		r3				r3
11		r5	r5		r5				r5

(Fig. 5) Extended LR parsing table for G

We propose the efficient incremental LR parsing algorithm as follows.

Algorithm 3. An improving parsing step on the tree structure.

input : a tree structure and an input list X.
output: the same structure updated to include the next configuration.
method: Let TOP point to the node Q(labeled S_m) of the tree.

```
    Xi : the current input string, Sm : the state
    on top of the stack.
    case 1 : action( Sm,Xi ) = shift
    (a) if Xi = Σ then
         Sm := action(Sm,Xi);
         if TOP^son=Sm then TOP:=TOP^son
                else
                  begin
                  create Q' to Q;
                  TOP := Q'
                end;
        Xi := Xi+1;
    (b) if Xi ∈ N then
        Sm := action(Sm,Xi);
        if TOP^son = Sm then
                  begin
                  Q link last_node(Xi)^left;
                  replace last_node(Xi) by TOP;
                  TOP := TOP^son
                    end
                else
                  begin
                    Q link last_node (Xi)^left;
                    replace last_node(Xi) by TOP;
                    create Q' to Q;
                    TOP:= Q'
                  end;
        Xi := Xi+1.
    case 2: action(Sm,Xi ) = reduce A→β
        Q := TOP - |β|;
        Sm := action(Sm,A);
        if Q^son = Sm then TOP := Q^son
                else
                  begin
                    create Q' to Q;
```

TOP := Q'

end;

case 3: *action*(S_m, X_i) = **error**

Stop the parsing and signal error

case 4: *action*(S_m, X_i) = **accept**

Terminate the parsing and signal acceptance

Algorithm 4. An improving incremental LR parsing on the tree structure.

input : input sequence X', and a tree structure representing a parse sequence for X.
output : a tree structure representing a parse sequence for X'.
method :

(1) if X_i = FIRST(wy$) then i := i^{th}(X);
 if X_j' = FIRST(w'y$) then j := j^{th}(X);
 According to Algorithm 3, initialize y of X' by *N*;
 first_node(*N* of X') := last_node(*N* of X);
 first_node($ of X') := last_node($ of X);
 TOP := first_node(X_i);

(2) node(X_j') := TOP;
 if X_j' = FIRST(y$) then go to step 4
 else go to step 3;

(3) Using algorithm 3, perform X', go to step 4

(4) P := last_node(X');
 if P = TOP then go to step 6
 else go to step 5;

(5) Using algorithm 3, perform X', go to step 4

(6) Stop

Example 4. Using the extended LR parsing table in (Fig. 5), we suppose that X=(n-n)-(n-n) modified to X'=(n-n)*(n-n).

From step 1 in Algorithm 4, we have i=j=6, and X'=(n-n)*F. TOP points to node 11 labeled S_{11}. In step 2, since X_6'*FIRST(F$) goto step 3.

In step 3, as *action*(S_{11},*)=reduce F→(E), back up the tree from the node 11 by 3 levels. Let the node 1 labeled S_0, and *action*(S_0,F) =S_3. Since S_3 exists at the sons of node 1, TOP points to node 12 labeled S_3.

In step 4, Let node 15 be P. Since P is not equal to TOP, go to step 5. In step 5, as *action*(S_3,*)=reduce T→F, back up the tree from the node 12 by 1 level. Let the node 1 labeled S_0, and *action*(S_0,T)=S_2. Since S_2 exists at the sons of node 1, TOP points to node 13 labeled S_2.

In step 4, P points to node 15. Since P is not equal to TOP, go to step 5. In step 5, as *action*(S_2,*)=S_7. Since S_7 does not exist at the sons of node 13, the new node 28 labeled S_7 is appended to the tree, and TOP points to node 28. Then we have j=7.

In step 4, P points to node 15. Since P is not equal to TOP, go to step 5. In step 5, as *action*(S_7,F)=S_{10}. Since S_{10} does not exists at the sons of node 28, the node 28 points to root node 16 of a left subtree for node 15 appended to X_7'. The node 15 appended to X_7' is replaced by node 28 pointed by TOP. The new node 29 labeled S_{10} is appended to the tree, and TOP points to node 29. Then we have j=8.

In step 4, P points to node 14. Since P is not equal to TOP, go to step 5. In step 5, as *action*(S_{10},$)=reduce T→T*F, back up the tree from the node 29 by 3 levels. Let the node 1 labeled S_0, and *action*(S_0,T)=S_2. Since S_2 exists at the sons of node 1, TOP points to node 13 labeded S_2.

In step 4, P points to node 14. Since P is not equal to TOP, go to step 5. In step 5, as *action*(S_2,$)=reduce E→T, back up the tree from the node 13 by 1 level. Let the node 1 labeled S_0, and *action*(S_0,E)=S_1. Since S_1 exists at the sons of node 1, TOP points to node 14 labeded S_1.

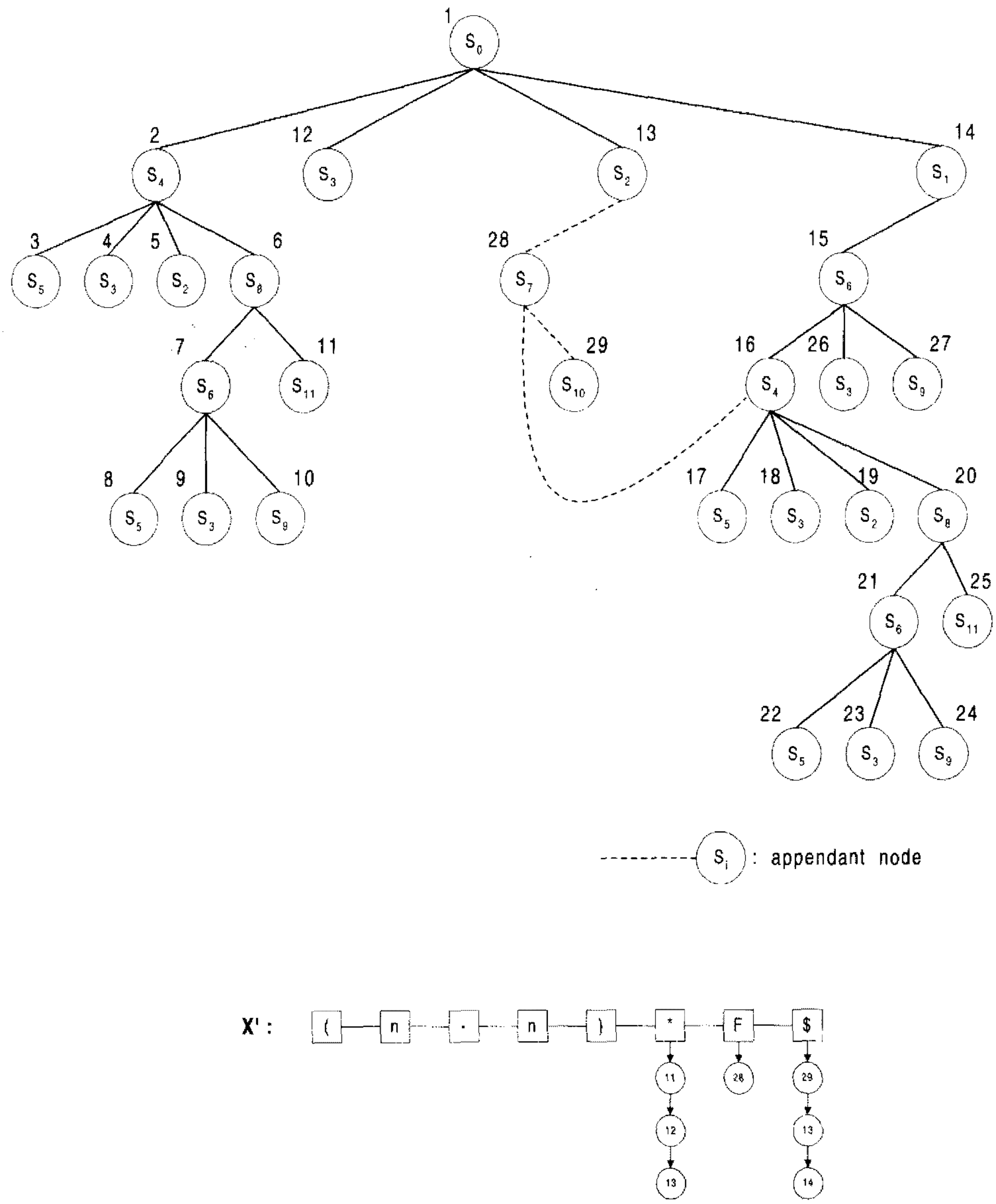
In step 4, P points to node 14. Since P is equal to TOP, go to step 6. In the step 6, the execution of the algorithm can be halted.

Therefore, using our incremental parsing algorithm 3 and algorithm 4, the tree structure for X′=(n-n)*F is shown in (Fig. 6).

Only 7 steps in our algorithm used for the incremental parsing, while Celentano's algorithm required 18 steps.

4.2 Experimental results

To evaluate the our incremental LR parsing algorithm, we implemented our algorithm and the conventional algorithm with C language on UNIX operating system.



(Fig. 6) Tree structure using our incremental parsing algorithm for *G*

〈Table 1〉 shows the input sentences of *G*.

〈Table 1〉 input sentences of *G*

input / case	input sentence
1	(n-n)-(n-n) → (n-n)*(n-n)
2	(n-n)*(n-n) → (n-n)-(n-n)
3	n-(n-n) → n*(n-n)
4	n*(n-n) → n-(n-n)
5	(n-n)-(n-(n-n)) → (n-n)-(n*(n-n))
6	(n-n)-(n*(n-n)) → (n-n)-(n-(n-n))
7	(n-(n-n))-((n-n)-(n-n)) → (n-(n-n))-((n*n)-(n-n))
8	(n-(n-n))-((n*n)-(n-n)) → (n-(n-n))-((n-n)-(n-n))
9	with a,a do s → with a,a do s
10	with a,a do s → with a,a do s
11	with a,a do with a,a do s → with a,a do with a,a do s
12	with a,a do with a,a do s → with a,a do with a,a do s

The performence measurements of the input sentences are shown in (fig. 7).

As shown in case 1 of (Fig. 7), our algorithm requires 36 parsing steps, while the conventional algorithm does 47 steps, using memory space of 672 bytes in comparison with 912 bytes in Celentano's algorithm.

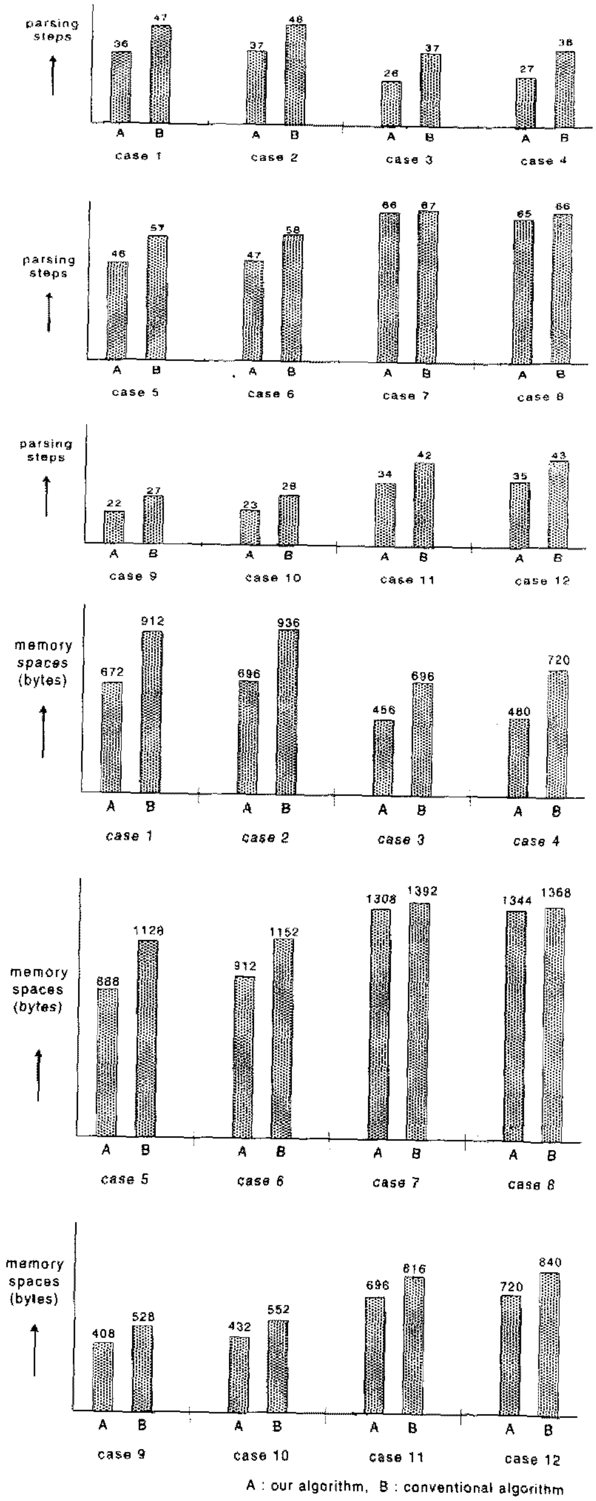
As shown in case 3 of (Fig. 7), our algorithm requires 26 parsing steps, while the conventional algorithm does 37 steps, using memory space of 456 bytes in comparison with 696 bytes in Celentano's algorithm.

Therefore, we show that the parsing steps and memory spaces in our parsing algorithm are reduced in all cases.

5. Conclusions

The incremental parsing techniques are an essential part of language-based environments which allow incremental construction of programs.

Celentano described an incremental LR parsing algorithm. Celentano suggested a possible improvement that would make his algorithm



(Fig. 7) Performence measurements

linear in time and memory space. Agrawal and Detro have shown how to extend the algorithm to accommodate epsilon production rules. However, their algorithms are too expensive in time and storage requirement to be of practical use.

We use the extended LR parsing tables which allows grammar symbols for the input, and we apply them to our incremental parsing algorithm. Using the extended LR parsing table, we suggest several methods to reduce its memory spaces and parsing steps as well. The algorithms described here were implemented in C language on a UNIX operating system, and were tested with several sentences for expressions.

As shown in case 1 of (Fig. 7), our algorithm requires 36 parsing steps, while the conventional algorithm does 47 steps, using memory space of 672 bytes in comparison with 912 bytes in Celentano's algorithm.

We show that the parsing steps and memory spaces in our algorithm are reduced in several sentences. The use of the substring parser in incremental parsing, however, has to be investigated further. One incremental parser constructed by the method in this paper is well being in our work on the implementation of an incremental evaluation algorithm for ordered attribute grammars.

In particular, our incremental LR parsing algorithm is more effective in the case of complex and large grammars, and long parse tree.

References

[1] Aho, A.V., Sethi, R. and Ullman, J.D., "Compilers: Principles, Techniques and Tools", *Addison-Wesley*, 1986.

[2] Ghezzi, C. and Mandrioli, D., "Incremental Parsing", *ACM TOPLAS*, Vol.1, No.1, pp.58-70, 1979.

[3] Ghezzi, C. and Mandrioli, D., "Augmenting Parsers to Support Incrementality", *Journal of ACM*, Vol.27, No.3, pp.564-579, 1980.

[4] Celentano, A. "Incremental LR Parsers", *Acta Informatica*, Vol.10, pp.307-321, 1978.

[5] Agrawal, R. and Detro, K.D., "An Efficient Incremental LR Parser for Grammars with Epsilon Productions", *Acta Informatica*, Vol.19, pp.369-373, 1983.

[6] Schwartz, M.D., Delisle, N.M. and Begwani, V.S.,"Incremental Compilation in Magpie", *ACM SIGPLAN Notices*, Vol.19, No.6, pp.122-131, 1984.

[7] Yeh, D., "On Incremental Shift-Reduce Parsing", *BIT*, Vol.23, No.1, pp.36-48, 1983.

[8] Yeh, D. and Kastens, U., "Automatic Construction of Incremental LR(1)-Parsers", *ACM SIGPLAN Notices*, Vol.23, No.3, pp.33-42, 1988.

[9] Bhatti, M.A., "Incremental Excution Environment", *ACM SIGPLAN Notices*, Vol.23, No.4, pp.56-64, 1988.

[10] Snelting, G., "How to Build LR Parsers Which Accept Incomplete Input", *ACM SIGPLAN Notices*, Vol.25, No.4, pp.51-58, 1990.

[11] Rekers, J., Koorn, W., "Substring Parsing for Arbitrary Context-Free Grammars", *ACM SIGPLAN Notices*, Vol.26, No.5, pp.59-66, 1991.

[12] Beetem, J.F. and Beetem, A.F., "Incremental Scanning and Parsing with Galaxy", *IEEE Transactions on Software Engineering*, Vol.SE-17, No.7, pp.641-651, 1991.

[13] Larchevêque, J.M., "Optimal Incremental Parsing", *ACM TOPLAS*, Vol.17, No.1, pp.1-15, 1995.

안 희 학

1981년 숭실대학교 전자계산학과 졸업(공학사)

1983년 숭실대학교 대학원 전자계산학과 졸업(공학석사)

1994년 숭실대학교 대학원 전자계산학과 졸업(공학박사)

1994년~1996년 관동대학교 전자계산소 소장

1984년~현재 관동대학교 전자계산공학과 교수

관심분야 : 컴파일러, 병렬컴파일러, 프로그래밍 언어, 함수 언어, 오토마타 등