점진적 스레드 트리를 구성하기 위한 파싱 알고리즘

A Parsing Algorithm for Constructing Incremental Threaded Tree

이 대 식*
Dae Sik Lee

요 약

점진적 파싱 기법은 프로그램의 점진적 구성을 허용하는 언어기반 환경의 중요한 부분이며, 프로그램의 변경된 부분에 대해서만 구문분석을 다시 함으로써 시스템의 성능을 향상 시킨다. 기존의 점진적 파싱은 파싱 정보를 저장하기 위해 스택 자료구조를 사용한다. 본 논문에서는 스택 자료구조를 사용하지 않고 노드 주소로 스레드를 추가하여 스레드 트리 구성 알고리즘을 제안한다. 또한 구성된 스레드 트리를 사용하여 5단계의 점진적 파싱 과정으로 나누어 점진적 스레드 트리 구성 알고리즘을 제안한다.

Abstract

The incremental parsing technique plays an important role in language-based environment which allows the incremental construction of a program. It improves the performance of a system by reanalyzing only the changed part of a program. The conventional incremental parsing uses the stack data structure in order to store the parsing information. In this paper, we suggest a threaded tree construction algorithm which parse by adding the threaded node address instead of using a stack data structure. We also suggest an incremental threaded tree construction which has incremental parsing process of five steps using the constructed threaded tree.

☞ Keyword : 점진적 스레드 트리(Incremental Threaded Tree), 파싱 알고리즘(Parsing Algorithm), 점진적 파싱 알고리즘(Incremental Parsing Algorithm)

1. 서 론

컴퓨터 하드웨어 기술의 급속한 발달과 소프트웨어 규모의 증가로 인하여 프로그램 개발은 소프트웨어의 재사용에 사용되는 점진적 파싱(incremental parsing) 기법이 매우 중요시되어 왔다[1].

점진적 파싱은 편집기 상태에서 원시 프로그램의 변경된 부분(삽입, 삭제, 대체등이 이루어진 부분)에 대한 재번역 과정중 변화된 부분만 파싱하고 다른 부분은 재파싱하지 않으므로 프로그램 개발시 파싱속도를 향상시킨다[2,3].

점진적 파싱 알고리즘에 관해서는 많은 연구가 있었으며, Celentano는 점진적 LR 파서[4], Ghezzi와 Mandrioli는 파스 트리 상에서 점진적 파싱을 할 수 스레드 트리[5], Degano와 Mannucci는 LR 기법을 확장하여 파싱표를 여러 개로 분리하는데 기초를 둔 일반적인 파서 기법[6], Yeh은 토큰 중심의 표현방법을 구성하는 점진적 파서[7] 등이 있다.

Larchevêque는 Ghezzi와 Mandrioli가 제안한 스레드 트리를 LALR(Lookahead LR) 형태로 사용하였으며, 특히 스레드 트리에서 스택의 push와 pop을 사용하여 노드의 재사용을 통한 최적화된 점진적 파서의 구성에 대한 개념을 제안하였다[8].

본 논문에서는 스택의 push와 pop을 사용하지 않고 노드 주소로 스레드를 추가하여 스레드 트리 구성 알고리즘을 제안한다. 또한 구성

* 정회원 : 안동과학대학 사이버테러대응과 전임강사
dslee@andong-c.ac.kr
[2005/02/20 투고 - 2006/03/21 심사 - 2006/05/09 심사완료]

된 스레드 트리를 사용하여 5단계의 점진적 파싱 과정을 나누어 점진적 스레드 트리 구성 알고리즘을 제안한다. 특히, Larchevêque의 점진적 스레드 트리와 제안한 점진적 스레드 트리 구성 알고리즘을 비교 분석해 보고자 한다.

2. 점진적 파싱

2.1 파싱의 정의

기본적인 표기방식은 Aho와 Ullman 방식을 기본으로 대부분의 프로그래밍 언어 구조는 문맥 자유 문법에 의하여 정의되는 하나의 본질적인 순환 구조를 갖는다[9,10].

문법 G_0 : (1) $S \rightarrow fFf$
(2) $F \rightarrow GH$
(3) $G \rightarrow g$
(4) $G \rightarrow Gg$
(5) $H \rightarrow h$

일 때, LR 파싱표는 표 1과 같다.

〈표 1〉 G_0에 대한 LR 파싱표

STATE	ACTION				GOTO			
	f	g	h	$	S	F	G	H
0	s2				1			
1				acc				
2		s5				3	4	
3	s6							
4		s8	s9					7
5	r3	r3	r3	r3				
6	r1	r1	r1	r1				
7	r2	r2	r2	r2				
8	r4	r4	r4	r4				
9	r5	r5	r5	r5				

문법 G_0에서 입력 스트링 z=fgghf에 대한 파싱은 그림 1과 같다.

	스택	입력	파스순서
Π =	(S_0,	fgghf$)	Π_0
	(S_0S_2,	gghf$)	Π_1
	($S_0S_2S_5$,	ghf$)	Π_2
	($S_0S_2S_4$,	ghf$)	Π_3
	($S_0S_2S_4S_8$,	hf$)	Π_4
	($S_0S_2S_4$,	hf$)	Π_5
	($S_0S_2S_4S_9$,	f$)	Π_6
	($S_0S_2S_4S_7$,	f$)	Π_7
	($S_0S_2S_3$,	f$)	Π_8
	($S_0S_2S_3S_6$,	$)	Π_9
	(S_0S_1,	$)	Π_{10}

〈그림 1〉 파스 순서 Π

2.2 Celentano의 점진적 파싱

점진적 파싱은 프로그램의 변경된 부분 즉, z=xwy를 문법 G에 의해 생성되는 스트링이라 하고 z′=xw′y를 w에서 w′으로 대체하여 변경된 스트링이라 하면 $w' \neq w$, $z' \in L(G)$ 이다. z의 파스 순서를 $\Pi=\Pi_0\ \Pi_1 \cdots \Pi_n$ 이라 하고, z′의 파스 순서를 $\Pi'=\Pi_0'\ \Pi_1' \cdots \Pi_m'$ 이라고 하자. 여기서, $\Pi_0=(S_0,\ z\$)$, $\Pi_0'=(S_0,\ z'\ \$)$, $\Pi_n=\Pi_m'=(S_0\ S_f,\ \$)$이다.

문법 G_0의 두 문장 입력 스트링 z=fgghf와 변경된 스트링 z′=fghf를 살펴보면 x=fg, y=hf, w=g, $w'=\varepsilon$ 이다. 점진적 파스순서 Π′는 그림 2와 같다.

	스택	입력	단계
T′ =	(S_0,	fghf$)	Π_0'
	(S_0S_2,	ghf$)	Π_1'
	($S_0S_2S_5$,	hf$)	Π_2'
	($S_0S_2S_4$,	hf$)	$\Pi_3'=\Pi_5$
	($S_0S_2S_4S_9$,	f$)	$\Pi_4'=\Pi_6$
	($S_0S_2S_4S_7$,	f$)	$\Pi_5'=\Pi_7$
	($S_0S_2S_3$,	f$)	$\Pi_6'=\Pi_8$
	($S_0S_2S_3S_6$,	$)	$\Pi_7'=\Pi_9$
	(S_0S_1,	$)	$\Pi_8'=\Pi_{10}$

〈그림 2〉 입력 스트링 삭제시 점진적 파스 순서 Π′

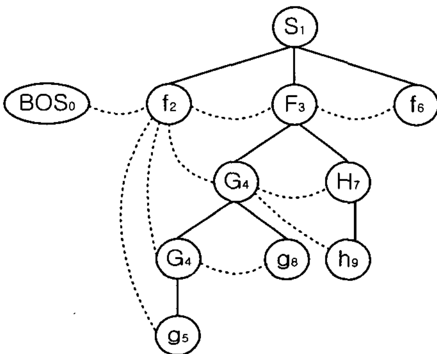
2.3 Larchevêque의 점진적 스레드 트리 구성을 위한 파싱

(1) 스레드 트리 구성

스레드 트리란 기본적으로 LR 파싱표를 사용하고 파스 트리인 동시에 파스 스택을 표현할 수 있는 자료구조이다. 스택의 상태는 스레드로 연결된 노드의 집합으로 표현되며, 스택의 push와 pop 연산은 새로운 스레드를 추가함으로써 이루어진다.

문법 G_0에서 입력 스트링 z=fgghf에 대한 파스 순서로 스택의 초기 상태인 BOS_0(Bottom Of Stack)을 시작으로 스레드 트리를 표현하면 그림 3과 같다.

스택	입력
(BOS_0,	fgghf$)
(BOS_0f_2,	gghf$)
($BOS_0f_2g_5$,	ghf$)
($BOS_0f_2G_4$,	ghf$)
($BOS_0f_2G_4g_8$,	hf$)
($BOS_0f_2G_4$,	hf$)
($BOS_0f_2G_4h_9$,	f$)
($BOS_0f_2G_4H_7$,	f$)
($BOS_0f_2F_3$,	f$)
($BOS_0f_2F_3f_6$,	$)
(BOS_0S_1,	$)



〈그림 3〉 Larchevêque의 스레드 트리 T

그림 3에서 표 1을 사용하여 shift, reduce, accept 단계를 보면 초기노드 BOS_0의 상태 0과 lookahead f로 ACTION[0, f]=shift 2가 노드 f_2를 생성한다. 노드 f_2를 스택에 push 하고 선행자 노드 BOS_0에 스레드 한다. 노드 g_5의 상태 5와 lookahead g로 ACTION[5, g]=reduce G→g가 된다. 왼쪽 심볼로 노드 G를 생성하고 자식 노드 g_5를 연결한다. 스택에서 G의 자식노드 g_5를 pop 하고 노드 G를 push 한다. 노드 G의 스레드는 왼쪽 자식노드의 선행자 노드 f_2로 스레드 한다. 노드 G의 상태는 GOTO[2, G]=4 이므로 노드 G_4가 된다. 마지막으로 노드 S_1의 상태 1과 lookahead $로 ACTION[1, $]=accept으로 파싱을 종료한다.

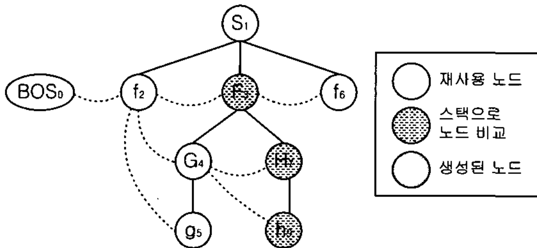
(2) 점진적 스레드 트리 구성

Larchevêque의 점진적 스레드 트리를 구성하면 알고리즘 1과 같다.

```
Given a threaded tree T for xwy, xw′y is
parsed in the following way.
 (1) Start with last(x) as the top of the
     stack.
 (2) With first(w′y) as the lookahead,
     perform reductions without creating
     new nodes as nodes long as of T can
     be reused. (When the left-hand side
     of a production to be applied
     matches the symbol of the parent of
     the top of stack, this node can be
     reused.)
 (3) As soon as a left-hand symbol
     differs from the previous parse or
     first(w′y) has been shifted, start
     parsing exhaustively.
 (4) When a node N′ just created domi
     nates the whole of w′, find the
     nearest common ancestor N of all
     terminal nodes that represent w (the
     deleted string) and of all nodes of
     T that are now dominated by N′, and
     if their symbols and positions
     match, substitute N′ for N.
```

(알고리즘 1) Larchevêque의 점진적 스레드 트리 구성 알고리즘

알고리즘 1을 적용하여 문법 G의 입력 스트링 z=fgghf와 변경 스트링 z′=fghf를 살펴보면 x=fg, y=hf, w=g, w′=ε 이다. 점진적 스레드 트리를 구성하면 그림 4와 같다.



〈그림 4〉 Larchevêque의 점진적 스레드 트리 T′

그림 4에서 보면 단계 1로부터 last(x)인 g를 스택의 top으로 한다. 단계 2에서는 first(w′y)=h를 lookahead로 놓고 스레드 트리 T가 재사용 되는 노드 G_4까지 노드를 생성하지 않고 reduce한다. 단계 3에서 first(wy)와 first(w′y)의 파싱이 다르므로 그림 3과 같이 파싱을 시작한다. 노드 h_9와 H_7, F_3은 스택을 사용한 스레드 트리 T에 있으므로 노드를 생성하지 않는다. 단계 4에서 w 전체를 지배하는 노드 $N(F_3)$과 w′ 전체를 지배하는 노드 $N'(F_3)$의 위치가 같으므로 N을 N′로 치환하고 점진적 스레드 트리 구성을 위한 파싱을 종료한다.

따라서 Larchevêque의 점진적 스레드 트리 T′를 구성하기 위하여 그림 3의 스레드 트리 T와 스택의 push와 pop을 사용하여 3개의 노드를 비교하였고, 새로 생성된 노드는 없다.

3. 제안한 점진적 스레드 트리 구성을 위한 파싱

3.1 제안한 스레드 트리 구성

기본적인 파싱은 LR 문법을 사용하여 스택

```
input  : Input string z
output : Threaded tree T
method : Node (Node.data, Node.address, Node.thread)
         S : current state, S′ : next state
(1) Node.address :=0; /* start node create */
    Node.data := 0;, S := 0;, Node.thread := null;
    create Node(labeled Node.address);
(2) ACTION[S, zi] = shift /* right brother node create */
    Node_right.address := Node.address + 1;
    create Node_right(labeled Node_right.address) to Node;
    Node_right↑.thread := Node.address;
    Node_right.data := join(zi, S′);
    Node := Node_right
(3) ACTION[S, zi] = reduce A → α /* parent node create */
    Node_parent.address := Node.address+1;
    create Node_parent(labeled Node_parent.address) to Node;
    Nodetmp := Node;
    for i := 1 to |α| do
      begin
        Node_parent↑.thread := Nodetmp↑.thread;
        Nodetmp := Nodetmp↑.thread
      end;
    S′ := GOTO[Node_parent↑.thread.S, A];
    Node_parent.data := join(A, S′)
    Node := Node_parent
(3) ACTION[S, zi] = error /* error string */
    Stop the parsing and signal error;
(4) ACTION[S, zi] = accept /* correct string */
    Terminate the parsing and signal acceptance;
```

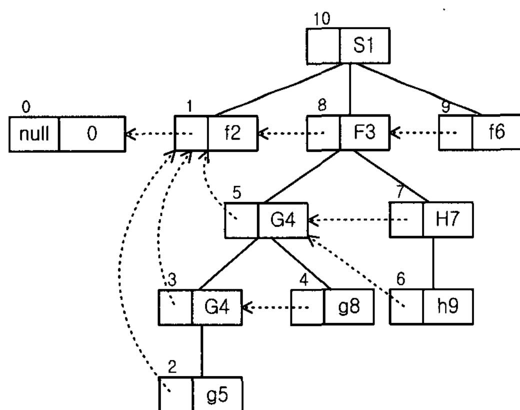
(알고리즘 2) 제안한 스레드 트리 구성 알고리즘

자료구조로 프로그래밍 언어의 파서를 구성한다. Celentano는 파싱 정보를 스택에 저장하였고[4], Larchevêque는 스택의 push와 pop을 사용하여 파싱 정보를 스레드 트리로 표현하였다[8].

본 논문에서는 스택의 push와 pop을 사용하지 않고 노드 주소로 스레드를 추가하였다. 따라서 노드 주소, 노드 데이터, 노드 스레드로 하나의 노드를 구성하고, 스레드 트리 T를 구성하였다.

입력 스트링 z_i = xwy를 파싱 정보를 스레드 트리로 표현하는 알고리즘을 제안하면 알고리즘 2와 같다.

알고리즘 2를 적용하여 입력스트링 z=fgghf에서 스레드 트리를 표현하면 그림 5와 같다.



〈그림 5〉 제안한 스레드 트리 T

그림 5에서 표 1을 사용하여 shift, reduce, accept 단계를 보면 Node.address=0, Node.data=0, Node.thread=null로 정의한 시작 노드를 생성한다.

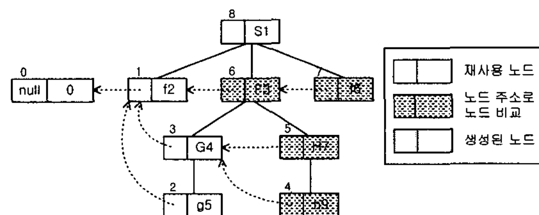
ACTION[0, f]=shift 2가 되는데 Node.address =0이 증가되어 Node_right. address=1이 Label된 오른쪽 형제 노드 Node_right로 생성한다. Node_right의 스레드는 현재 Node의 Node.address=0으로 스레드 하고, 노드 데이타는 입력 스트링이 다음 상태 S′ =2와 결합되어 f2가 된다. 다음 노드를 생성하기 위해 Node_right로 생성된 노드 1을 Node로 전환한다. ACTION[5, g]=reduce G→g가 되는데 Node.address=2가 증가되어 Node_parent.address= 3이 Label된 부모노드 Node_parent를 생성한다. Node_parent의 노드 가지는 오른쪽 심볼 |α|의 길이 1만큼 가지를 연결하고, 스레드는 이전 Node로부터 |α|의 길이 1만큼 스레드를 반복하여 Node.address=1로 스레드 한다. 다음 상태 S′ 는 Node_parent의 스레드인 노드 상태 2와 왼쪽 심볼 G가 GOTO[2, G]=4 상태로 변경된다. Node_parent.data는 왼쪽 심볼과 다음 상태가 결합되어 G4가 된다. 다음 노드를 생성하기 위해 Node_parent로 생성된 노드 3을 Node로 전환한다. 마지막으로 노드 S1과 $로 ACTION[1, $]= accept되어 스레드 트리 구성을 위한 파싱을 종료한다.

3.2 제안한 점진적 스레드 트리 구성

Celentano는 스택에 저장된 파싱 정보를 사용하여 점진적 파싱을 하였고[4], Larchevêque는 파싱 정보를 스레드 트리로 표현하는 과정에서 스택 개념을 사용하여 점진적 스레드 트리를 구성하였다[8].

본 논문에서는 스레드 트리만으로 표현된 파싱 정보를 사용하여 5단계로 나누어 점진적 스레드 트리를 구성하였다. 점진적 스레드 트리 알고리즘을 제안하면 알고리즘 3과 4와 같다.

알고리즘 3을 적용하여 문법 G_0에서 변경후 스트링 z′ =fghf일 때 점진적 스레드 트리 T′ 를 구성하면 그림 6과 같다.



〈그림 6〉 제안한 점진적 스레드 트리 T′

```
      if ACTION[S, zi′] = shift then
         begin
           Node′.address := Node′.address + 1;
           Node′↑.thread := Node′.address-1;
           Node′ := join(zi′, S′);
           if Node.thread = (Node.address-1).thread then
              repeat
                 begin
                   Node.address := Node.address + 1
                 end;
              until shift zi;
              Nodetmp := Node
         end;
         else if ACTION[S, zi′] = reduce A → α then
           begin
              Node′tmp := Node′;
              Node′.address := Node′.address+1;
              for i := 1 to |α| do
                 begin
                   Node′↑.thread := Node′tmp↑.thread;
                   Node′tmp := Node′tmp↑.thread
                 end;
              S′ := GOTO[Q↑.thread.S, A];
              Node′.data := join(A, S′)
              if Node.thread = (Node.address-1) then
                 begin
                   Node.address := Node.address - 1;
                   Nodetmp := Node
                 end;
```

(알고리즘 3) 노드 주소 및 스레드 변경 알고리즘

```
input : Changed input string z′, Threaded tree T of z
output : Incremental thread tree T′ of changed input string z′
method : Node′ (Node′.data, Node′.address, Node′.thread)
(1) Node.address := FIRST_REDUCE(wy$); /* first reduce part of string z=wy$ */
    Node′.address := Node.address;
    for Node.address := 0 to Node.address do
       begin /* insert node of T to T′ */
         T′ := Node;
       end;
(2) repeat
      begin /* ready for compare to Node and Node′ */
        Node.address := Node.address + 1;
        Nodetmp := Node;
        using Algorithm 3;
        /* change Node′.address and Node′.thread of T′ */
      end;
      repeat
         begin
           if Nodetmp.data = Node′.data then
              begin /* if Node and Node′ equal, insert node of T to T′ */
                Nodetmp.address := Node′.address;
                Nodetmp.thread := Node′.thread;
                T′ := Nodetmp;
                goto step 2
              end;
           else Nodetmp.address := Nodetmp.address + 1
         end;
      until shift zi or accept;
         Node′ := Node′.address-1;
         using Algorithm 3;
      end;
    until zi′ = FIRST_SHIFT(y$) or accept;
    if zi′ = accept then go to step 5;
(3) Node.address := FIRST_SHIFT(y$); /* first shift part of string z′=y$ */
    repeat
      begin
         Node.address := Node.address + 1;
         Nodetmp := Node;using Algorithm 3;
         Nodetmp.address := Node′.address;
         Nodetmp.thread := Node′.thread;
         T′ := Nodetmp
      end;
    until zi′ = LAST_SHIFT(y$);
(4) Node.address := LAST_SHIFT(y$); /* last shift part of string z′=y$ */
    go to step 2;
(5) Stop
```

(알고리즘 4) 제안한 점진적 스레드 트리 구성 알고리즘

그림 6에서 보면 단계 1은 그림 5의 스레드 트리 T에서 FIRST_REDUCE(wy$)인 Node.address=3까지 Node를 점진적 스레드 트리 T′ 에 추가한다. 단계 2는 Node.address가 증가된 4로부터 z′ =FIRST_SHIFT(y$)될 때까지 T의 Node.data와 T′ 의 Node′ .data를 비교하여 같으면 T의 Node를 T′ 에 추가하고 아니면 알고리즘 2로 새로운 Node를 생성하는데 w′ ="ε"이므로 단계 2를 실행하지 않는다. 단계 3은 FIRST_SHIFT(y$)인 Node.address=4로부터 z′ = LAST_SHIFT (y$)될 때까지 알고리즘 3으로 T의 Node.address와 Node.thread만 변경하여 T′ 에 추가한다. 단계 4에서 last_shift(y$)인 Node.address=7로부터 z′ =accept될 때까지 T의 Node.data와 T′ 의 Node′ .data를 비교하여 같으면 T의 Node를 T′에 추가하고 아니면 알고리즘 2로 새로운 Node를 생성한다. 단계 4에서 ACTION[1, $]=accept이므로 단계 5에서 점진적 스레드 트리 구성을 종료한다.

따라서 제안한 점진적 스레드 트리 T′ 를 구성하기 위하여 그림 5의 스레드 트리 T의 노드를 비교한 것은 4개이고, 새로 생성된 노드는 없다.

4. 실험 및 분석

Larchevêque의 점진적 스레드 트리 구성과 본 논문에서 제안한 점진적 스레드 트리 구성 알고리즘의 재사용 노드 성능을 비교 분석한다. SUN Blade-2000 시스템에서 C언어를 사용하여 실험하였다.

실험)

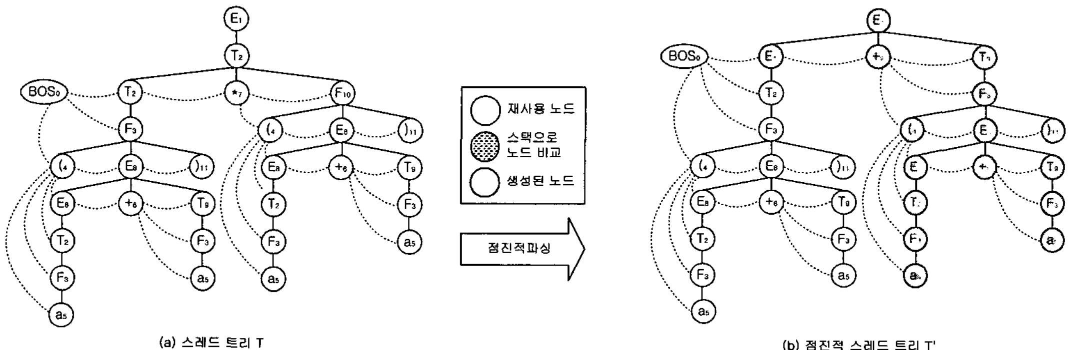
문법 G_1 :
(1) $E \rightarrow E+T$
(2) $E \rightarrow T$
(3) $T \rightarrow T*F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow a$

일 때, LR 파싱표는 표 2와 같다.

〈표 2〉 G_1에 대한 LR 파싱표

STATE	ACTION						GOTO		
	a	+	*	(	)	$	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			

문법 G_1을 만족하는 입력 스트링 z=(a+a)*(a+a)가 z′ =(a+a)+(a+a)로 변경될 경우 Larchevêque의 알고리즘으로 점진적 스레드 트리를 구성하



(a) 스레드 트리 T
(b) 점진적 스레드 트리 T'

〈그림 7〉 Larchevêque의 점진적 스레드 트리

면 그림 7과 같고, 제안한 알고리즘으로 점진적 스레드 트리를 구성하면 그림 8과 같다.

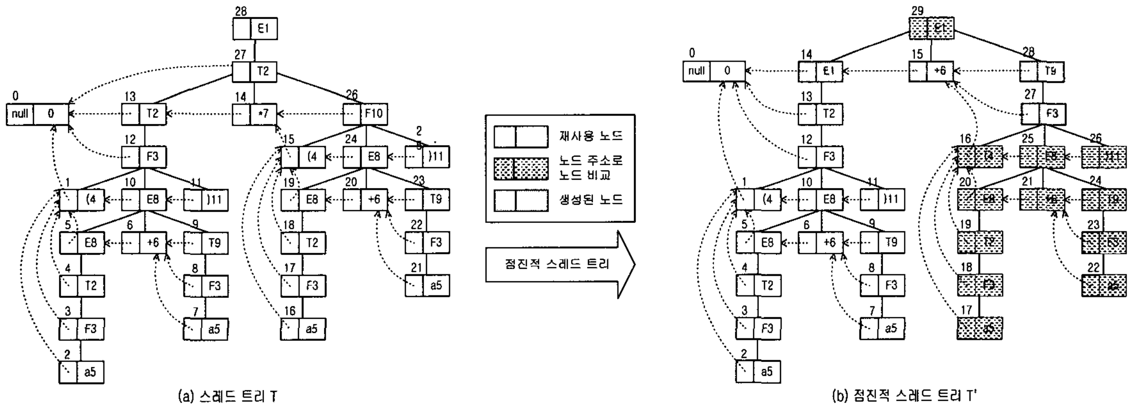
알고리즘 1에 따라 그림 7을 보면 단계 1로부터 last(x)인)를 스택의 top으로 한다. 단계 2에서는 first(w′ y)=*를 lookahead로 놓고 스레드 트리 T가 재사용 되는 노드 T_2까지 노드를 생성하지 않고 reduce한다. 단계 3에서 first(wy)와 first(w′ y)의 파서와 다르므로 first(w′ y)의 파싱을 시작한다. 그러나 단계 4에서 w 전체를 포함하는 노드 N과 w′ 전체를 포함하는 노드 N′ 심볼의 위치를 찾지 못하므로 계속 파싱을 수행하여 노드를 생성한다. 따라서 전체 30개의 노드 중 재사용 노드는 14개이고, 생성된 노드는 16개이다.

그림 8에서 보면 알고리즘 4의 3단계에서 스트링 z′ =y$의 LAST_SHIFT인 노드 주소 26까지 E_1, $(_4$, a_5(노드 주소 17), F_3(노드 주소 18), T_2, E_8(노드 주소 20), $+_6$, a_5(노드 주소 22), F_3 (노드 주소 23), T_9, E_8(노드 주소 25), $)_{11}$은 스레드 트리에 노드 주소를 추가하여 비교한다. 따라서 전체 30개의 노드 중 재사용 노드는 26개이고, 생성된 노드는 4개이다.

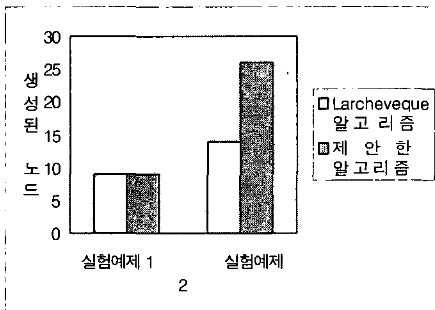
본 논문에서 제안한 점진적 스레드 트리 구성 알고리즘의 타당성을 입증하기 위해 그림 4와 그림 6을 실험 예제 1, 그림 7와 그림 8을 실험 예제 2로 하면 전체 파싱 노드 중 약 40%의 새로 생성된 노드가 감소되고, 점진적 파싱속도는 약 9% ~ 45% 가량 향상된다. 따라서 실험예제 1과 실험예제 2를 나타내면 그림 9와 같다.

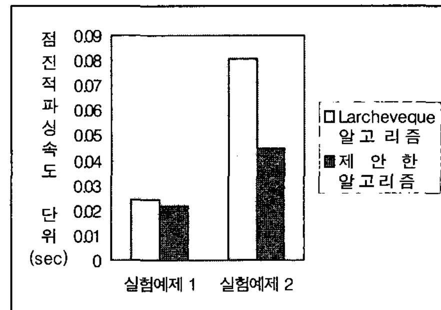
5. 결 론

점진적 파싱이란 프로그램의 어느 부분을 수정했을 때 프로그램 전체를 처음부터 분석하지 않고 변경된 부분에 의하여 영향을 받는 최소



〈그림 8〉 제안한 점진적 스레드 트리





〈그림 9〉 점진적 스레드 트리의 성능 분석

한의 부분만을 다시 분석하는 파싱 방법이다.

본 논문에서는 스택의 push와 pop을 사용하여 스레드 트리를 구성한 것과는 달리 노드 주소로 스레드를 추가하는 스레드 트리 구성 알고리즘과 점진적 스레드 트리 구성 알고리즘을 제안하였다. 제안한 점진적 스레드 트리 구성 알고리즘은 입력 스트링 z=xwy의 y 부분에 따라 다르지만 스레드 트리 T의 입력 스트링 변경시 스택을 사용한 점진적 스레드 트리 구성보다 전체 파싱 노드 중 약 40%의 새로 생성된 노드가 감소되고, 점진적 파싱속도는 약 9% ~ 45% 가량 향상된다.

따라서 본 논문에서 제안한 점진적 스레드 트리 구성 알고리즘은 프로그램 재사용을 위해 프로그램 편집 환경에서 효율적으로 사용될 수 있다.

참 고 문 헌

[1] Wagner, T.A., and Graham, S.L., "Efficient and Flexible Incremental Parsing", *ACM Transactions on Programming Languages and Systems*, Vol.20, No.5, pp.980-1013, 1998.

[2] Dykes, L. R. and Cameron, R. D. "Toward High-Level Editing in Syntax- Bsaed Editors", *Software Eng.* J., Vol.5, pp.237-244, 1990.

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

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

[5] Ghezzi, C. and Mandrioli, D., "Incremental Parsing", ACM Transactions on Programming Languages and Systems, Vol.1, No.1, pp. 58- 70, 1979.

[6] Degano, P., Mannucci, S. and Mojana, B., "Efficient Incremental LR Parsing for Syntax-Directed Editor", *ACM Transactions on Programming Languages and Systems,* Vol.10, No.3, pp.345-373. 1988.

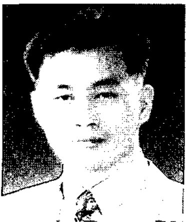
[7] Yeh, D. and Kastens, U., "Automatic Construction of Incremental LR(1) - Parsers", *ACM SIGPLAN Notices*, Vol.23, No.3, pp. 33-42, 1988.

[8] Larchevêque, J. M., "Optimal Incremental Parsing", *ACM Transactions on Programming Languages and Systems*, Vol.17, No.1, pp.1-15, 1995.

[9] Aho, A. V., Sethi. R. and Ullman. J. D., "*Compilers : Principles, Techniques, and Tools*", Addison-Wesly Publishing Company, 1986.

[10] 안희학 "LR 파서를 위한 효율적인 점진적 파싱", *한국정보처리학회 논문지*, 제 5권, 제 6호, pp. 1660-1669, 1998.

◐ 저 자 소 개 ◑



이 대 식 (Dae Sik Lee)

1995년 관동대학교 전자계산공학과(공학사)
1999년 관동대학교 전자계산공학과(공학석사)
2004년 관동대학교 전자계산공학과(공학박사)
2005년~현재 안동과학대학 사이버테러대응학과 전임강사
E-mail : dslee@andong-c.ac.kr