스택을 이용하지 않는 스레드 트리 구성 알고리즘

A Threaded Tree Construction Algorithm not Using Stack

이 대 식*
Dae-Sik Lee

요 약

언어 기반 프로그래밍 환경의 발전에 따라 점진적 파싱에 대한 연구는 핵심적인 분야가 되었다. 본 논문의 목적은 파싱 속도(time)와 기억장소가 많이 요구하는 기존의 알고리즘들보다 효율적인 점진적 파싱 알고리즘을 제시하는데 있다. 본 논문에서는 스택을 이용하지 않는 스레드 트리 구성 알고리즘을 제안하였다. 또한 노드의 재파싱 과정을 없애기 위해 스택을 이용하지 않는 노드 생성 알고리즘과 점진적 스레드 트리 구성 알고리즘을 제안하였다.

Abstract

As the development of language-based programming environment, a study on incremental parsing has become an essential part. The purpose of this paper is to show the more efficient incremental parsing algorithm than earlier one that demands parsing speed and memorizing space too much. This paper suggests the threaded tree construction algorithm not using stack. In addition, to remove the reparsing process, it proposes the algorithm for creation node and construction incremental threaded tree not using stack.

Keyword : Threaded Tree, Stack, Parsing

1. 서 론

현재 소프트웨어 능력의 향상과 데이터 관리기법의 개발로 점진적 파싱이론이 활발히 연구되고 있다. 점진적 파싱은 편집기 상태에서 원시 프로그램의 변경된 부분에 대한 재번역 과정 중 변화된 부분만 다시 파싱하고 다른 부분은 재파싱하지 않으므로 프로그램 개발시 시간적 물리적 이점을 가지고자 하는 방법이다[1],[2].

Celentano는 처음으로 LR(Left Right Scanning) 기반의 점진적 파싱 알고리즘을 제안하였다[3]. 하지만 많은 기억장소와 수행시간이 길다는 단점 때문에 파싱과정을 트리구조로 변경하여 표현하였다. Ghezzi와 Mandrioli는 주어진 문법이 LR이고 RL인 문법에 대한 알고리즘을 작성하였으며, 파스 트리 상에서 점진적 파싱을 할 수 있도록 스레드 트리를 제시하였다[4]. Agrawal과 Detro는 Celentano의 연구결과를 보완하여 ε-생성 규칙이 포함된 경우라도 처리 할 수 있도록 하였다[5]. Yeh는 일반적인 상향식 파싱 방법인 shift-reduce 파서와 점진성을 지지하는 LR(0) 파서를 증가시키는 알고리즘을 제시하였다[6]. Li는 편집의 위치를 한 곳으로 제한하거나, 편집이 가능한 구문의 범주를 제한하지 않고 여러 곳의 수정 위치에 대해 편집 할 수 있는 다중 편집 위치에서 적용될 수 있는 문장형 LL(Left to right scanning, Left parse) 파서를 제시하였다[7]. Larchevêque는 Ghezzi와 Mandrioli가 제안한 SLR(Simple LR) 기반의 스레드 트리를 LALR (Lookahead LR) 형태로 이용하였으며, 특히 스레드 트리에서 스택의 push와 pop을 이용하여 노드의 재사용을 통한 최적화된 점진적 파서의 구성에 대한 개념을 제시하였다[8]. 그런데 이러한 점진적 파싱을 위해서는 그 이전의 파싱 정보를 기억하는 것이 매우 중요 했으며, 이런 파싱 정보를 스택을 이용하였기 때문에 많은 기억장소와 파싱시간을 필요로 한다.

* 정 회 원 : 관동대학교 멀티미디어공학부 겸임교수
lds@kd.ac.kr(제 1저자)

본 논문에서 점진적 파싱의 목표는 빠른 파싱과 노드 재사용을 최대화하여 기억장소의 낭비를 줄이는 것이다. 따라서 스택을 이용하지 않는 스레드 트리 구성 알고리즘을 제안하고, 스택을 이용하지 않는 노드 데이터 생성 알고리즘과 점진적 스레드 트리 구성 알고리즘을 제안한다. 특히, YACC[9]을 이용하여 단말기호를 입력 스트링으로 받아들이는 LR 파싱표를 자동으로 생성하였으며, 자동으로 생성된 LR 파싱표로 스택을 이용하지 않는 점진적 스레드 트리 구성 알고리즘을 알고리즘을 구현해 보고자 한다.

2. 이론적 배경

2.1 파싱의 기본 구조

대부분의 프로그래밍 언어 구조는 문맥 자유 문법(이하 문법이라 함)에 의하여 정의되는 하나의 본질적인 순환 구조를 갖는다[10].

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

일 때, LR 파싱표는 표 1과 같다.

〈표 1〉 G1에 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			

문장(sentence)에 대한 파싱 과정은 파서 configuration의 순서로 표현되는데 LR 파서의 configuration은 쌍(S, x$)으로 구성된다. 여기서, $S=S_0S_1\cdots S_m$ 은 톱(top)으로 S_m을 가진 스택의 내용이고 x$는 남아있는 입력이며, $는 오른쪽 끝마커(right endmaker)이다. configuration 간의 이진 관계(binary relation) ⊢는 파싱 단계를 결정한다. 문법 G와 G에 대한 LR 파서가 주어졌을 때, 각 문장 $z \in L(G)$에 대하여 $\Pi_0=(S_0, z\$)$, $\Pi_n=(S_0S_f, \$)$인 파스 순서(parse sequence) $\Pi=\Pi_0\Pi_1 \cdots \Pi_n$이라 불리는 파서 configuration이 존재한다. 여기서, S_0는 초기 상태(initial state)이고, S_f는 ACTION[S_f, $]=accept 상태인데, $0 \le i < n$인 모든 i에 대하여 $\Pi_i \vdash \Pi_{i+1}$이다.

문법 G_1에서 입력 스트링 z=(a+a)*(a+a)에 대한 파싱은 그림 1과 같다.

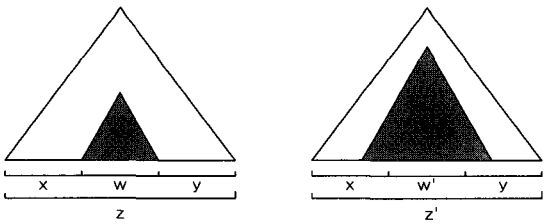
	상 태	스 택	단계
Π′ =	(S_0,	(a+a)*(a+a)$	Π_0
	(S_0S_4,	a+a)*(a+a)$	Π_1
	($S_0S_4S_5$,	+a)*(a+a)$	Π_2
	($S_0S_4S_3$,	+a)*(a+a)$	Π_3
	($S_0S_4S_2$,	+a)*(a+a)$	Π_4
	($S_0S_4S_8$,	+a)*(a+a)$	Π_5
	($S_0S_4S_8S_6$,	a)*(a+a)$	Π_6
	($S_0S_4S_8S_6S_5$,	)*(a+a)$	Π_7
	($S_0S_4S_8S_6S_3$,	)*(a+a)$	Π_8
	($S_0S_4S_8S_6S_9$,	)*(a+a)$	Π_9
	($S_0S_4S_8$,	)*(a+a)$	Π_{10}
	($S_0S_4S_8S_{11}$,	*(a+a)$	Π_{11}
	(S_0S_3,	*(a+a)$	Π_{12}
	(S_0S_2,	*(a+a)$	Π_{13}
	($S_0S_2S_7$,	(a+a)$	Π_{14}
	($S_0S_2S_7S_4$,	a+a)$	Π_{15}
	($S_0S_2S_7S_4S_5$,	+a)$	Π_{16}
	($S_0S_2S_7S_4S_3$,	+a)$	Π_{17}
	($S_0S_2S_7S_4S_2$,	+a)$	Π_{18}
	($S_0S_2S_7S_4S_8$,	+a)$	Π_{19}
	($S_0S_2S_7S_4S_8S_6$,	a)$	Π_{20}
	($S_0S_2S_7S_4S_8S_6S_5$,	)$	Π_{21}
	($S_0S_2S_7S_4S_8S_6S_3$,	)$	Π_{22}
	($S_0S_2S_7S_4S_8S_6S_9$,	)$	Π_{23}
	($S_0S_2S_7S_4S_8$,	)$	Π_{24}
	($S_0S_2S_7S_4S_8S_{11}$,	$	Π_{25}
	($S_0S_2S_7S_{10}$,	$	Π_{26}
	(S_0S_2,	$	Π_{27}
	(S_0S_1,	$	Π_{28}

〈그림 1〉 G1에서 z=(a+a)*(a+a)에 대한 파스 순서 Π

2.2 점진적 파싱의 기본 구조

점진적 파싱에서 프로그램의 변경된 부분 즉, z=xwy를 문법 G에 의해 생성되는 스트링이라 하고 z′ =xw′ y를 w에서 w′ 으로 대체하여 변경된 스트링이라 하면 w′ ≠ w, z′ ∈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, Π_0=(S_0, z$), Π_0' =(S_0, z′ $), $\Pi_n=\Pi_m'$ =(S_0 S_f, $)이다.

일반적으로, Π와 Π′ 에서 앞부분의 파싱 단계 즉, x를 파싱하기 위한 단계는 서로 같고, w와 w′ 그리고 y의 일부분에 대한 파싱 단계가 그림 2와 같이 다시 계산된다. 이것은 파스 순서에 의하여 Π′ 을 구하기 위하여 Π의 어떤 부분이 다시 계산 되는가를 찾는 것과 같다.



〈그림 2〉 z=xwy에서 z′ =xw′ y로 변경될 때의 파싱 단계

문법 G_1의 두 문장 z=(a+a)*(a+a)와 z′ =(a+a)+(a+a)를 살펴보면 x=(a+a), y=(a+a), w=*, w′ =+ 일 때 점진적으로 계산된 파스순서 Π′ 는 그림 3과 같다.

2.3 Larchevêque의 스택을 이용한 점진적 파싱

(1) 파싱을 위한 스레드 트리

스레드 트리란 기본적으로 LR 파싱 테이블을 사용하지만 파스 트리인 동시에 파스 스택을 표현할 수 있는 자료 구조이다. 스택의 상태는 스레드로 연결된 노드의 집합으로 표현되며, pop과 push 연산은 새로운 스레드를 추가함으로써 이루어진다. Larcheveque의 스레드 트리에 대한 스레드는 정의 1과 같다.

	상 태	스 택	단계
Π′ =	Π_0' ~ Π_{13}' 까지 Π_0 ~ Π_{13} 와 일치하므로 대치.		
	(S_0S_1,	+(a+a)$	Π_{14}'
	($S_0S_1S_6$,	(a+a)$	Π_{15}'
	($S_0S_1S_6S_4$,	a+a)$	Π_{16}'
	($S_0S_1S_6S_4S_5$,	+a)$	Π_{17}'
	($S_0S_1S_6S_4S_3$,	+a)$	Π_{18}'
	($S_0S_1S_6S_4S_2$,	+a)$	Π_{19}'
	($S_0S_1S_6S_4S_8$,	+a)$	Π_{20}'
	($S_0S_1S_6S_4S_8S_6$,	a)$	Π_{21}'
	($S_0S_1S_6S_4S_8S_6S_5$,	)$	Π_{22}'
	($S_0S_1S_6S_4S_8S_6S_3$,	)$	Π_{23}'
	($S_0S_1S_6S_4S_8S_6S_9$,	)$	Π_{24}'
	($S_0S_1S_6S_4S_8$,	)$	Π_{25}'
	($S_0S_1S_6S_4S_8S_{11}$,	$	Π_{26}'
	($S_0S_1S_6S_3$,	$	Π_{27}'
	($S_0S_1S_6S_9$,	$	Π_{28}'
	(S_0S_1,	$	Π_{29}'

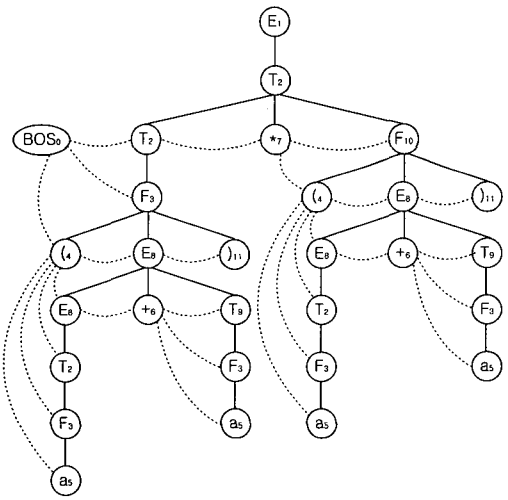
〈그림 3〉 G1에서 z′ =(a+a)+(a+a)에 대한 점진적 파스 순서 Π′

① 모든 노드는 전임자 노드에 스레드 한다.
② 전임자 노드는 왼쪽 형제 노드 또는 조상의 왼쪽 형제 노드이다.
③ 스레드는 s_1인 N_1이 심볼 X_2인 N_2의 전임자라면, goto(s_1, X_2)가 되거나 또는 action(s_1, X_2)=shift 되는 유효한 변화로 표현한다.

〈정의 1〉 스레드의 정의

문법 G_1에서 입력 스트링 z=(a+a)*(a+a)을 스택의 초기 상태인 BOS_0(Bottom Of Stack)을 시작으로 스레드 트리로 표현하면 그림 4와 같다.

스택	입력
(BOS_0,	(a+a)*(a+a)$)
(BOS_0($_4$,	a+a)*(a+a)$)
(BOS_0($_4$$a_5$,	+a)*(a+a)$)
(BOS_0($_4$$F_3$,	+a)*(a+a)$)
(BOS_0($_4$$T_2$,	+a)*(a+a)$)
(BOS_0($_4$$E_8$,	+a)*(a+a)$)
(BOS_0($_4$$E_8$+$_6$,	a)*(a+a)$)
(BOS_0($_4$$E_8$+$_6$$a_5$,	)*(a+a)$)
(BOS_0($_4$$E_8$+$_6$$F_3$,	)*(a+a)$)
(BOS_0($_4$$E_8$+$_6$$T_9$,	)*(a+a)$)
(BOS_0($_4$$E_8$,	)*(a+a)$)
(BOS_0($_4$$E_8$)$_{11}$,	*(a+a)$)
($BOS_0$$F_3$,	*(a+a)$)
($BOS_0$$T_2$,	*(a+a)$)
($BOS_0$$T_2$*$_7$,	(a+a)$)
($BOS_0$$T_2$*$_7$($_4$,	a+a)$)
($BOS_0$$T_2$*$_7$($_4$$a_5$,	+a)$)
($BOS_0$$T_2$*$_7$($_4$$F_3$,	+a)$)
($BOS_0$$T_2$*$_7$($_4$$T_2$,	+a)$)
($BOS_0$$T_2$*$_7$($_4$$E_8$,	+a)$)
($BOS_0$$T_2$*$_7$($_4$$E_8$+$_6$,	a)$)
($BOS_0$$T_2$*$_7$($_4$$E_8$+$_6$$a_5$,	)$)
($BOS_0$$T_2$*$_7$($_4$$E_8$+$_6$$F_3$,	)$)
($BOS_0$$T_2$*$_7$($_4$$E_8$+$_6$$T_9$	)$)
($BOS_0$$T_2$*$_7$($_4$$E_8$	)$)
($BOS_0$$T_2$*$_7$($_4$$E_8$)11	$)
($BOS_0$$T_2$*$_7$$F_{10}$	$)
($BOS_0$$T_2$	$)
($BOS_0$$E_1$	$)



〈그림 4〉 스택을 이용한 스레드 트리

그림 4에서 문법 G_1을 이용하면 shift는 초기노드 BOS_0의 상태 0과 lookahead (로 ACTION[0, (]=shift 4가 노드 ($_4$를 생성한다. 노드 ($_4$를 스택에 push 하고 선행자 노드 BOS_0에 스레드 한다. reduce는 노드 a_5의 상태 5와 lookahead +로 ACTION[5, +]=reduce F→a가 된다. 왼쪽 심볼로 노드 F를 생성하고 자식 노드 a_5를 연결한다. 스택에서 F의 자식노드 a_5를 pop 하고 노드 F를 push 한다. 노드 F의 스레드는 왼쪽 자식노드의 선행자 노드 ($_4$로 스레드 한다. 노드 F의 상태는 GOTO[4, F]=3 이므로 노드 F_3이 된다. 마지막으로 accept는 노드 E_1의 상태 1과 lookahead $로 ACTION[1, $]=accept으로 파싱을 종료한다.

(2) 점진적 파싱을 위한 스레드 트리

점진적 파싱이란 입력 스트링 z는 서브스트링 xwy를 포함한다. z의 스레드 트리를 T라고 한다. 변경된 스트링 z′ 는 서브스트링 xw′ y를 포함한다. z′ 를 위한 새로운 스레드 트리를 T′ 라 한다. 이때 파스 트리 중 변경된 부분 w′ 에 대한 왼쪽 부분 x와 오른쪽 부분 y의 재파싱을 피하기 위해 스레드 트리를 이용하여 점진적 파싱을 한다.

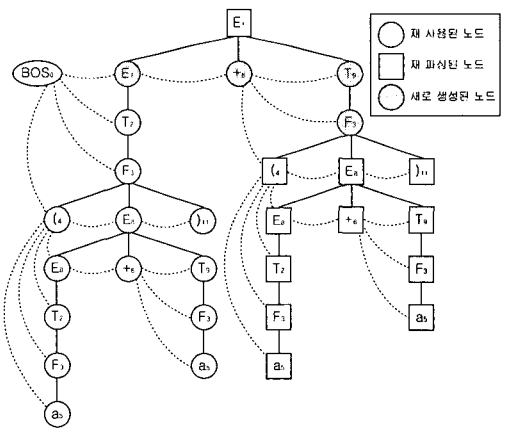
Larchevêque의 스택을 이용한 점진적 파싱 알고리즘을 제시하면 알고리즘 1과 같다.

입 력 : 입력 스트링 z와 z′, z에 대한 스레드 트리 T
출 력 : z′ 에 대한 스레드 트리 T′
방 법 : (1) last(x)로부터 스택의 top으로 시작한다.
(2) first(w′ y)를 lookahead로 놓고 T가 재사용 되는 동안 새로운 노드를 생성하지 않고 reduce 한다.
(3) 왼쪽 심벌이 z의 파서와 다르거나 first (w′ y)가 shift되면 파싱을 한다.
(4) w′ 를 포함하는 가장 가까운 조상 노드를 N′ 하고, w를 포함하는 가장 가까운 조상 노드를 N으로 한다. 만약 N과 N′ 가 심볼의 위치가 같으면 N을 N′ 로 치환하다. 그렇지 않으면 파싱을 한다.
(5) 파싱을 끝낸다.

〈알고리즘 1〉 스택을 이용한 점진적 파싱 알고리즘

알고리즘 1을 적용하여 문법 G_1의 입력 스트링 z=(a+a)*(a+a)와 변경 스트링 z′ =(a+a)+(a+a)를 살펴보면 x=(a+a), y=(a+a), w=*, w′ =+ 이다. 입력 스트링 변경시 스택을 이용한 점진적 스레드 트리를 구성하면 그림 4와 같다.

스 택	입 력	
		재사용 노드
(BOS_0E_1,	+(a+a)$)	생성된 노드
($BOS_0E_1+_6$,	(a+a)$)	
($BOS_0E_1+_6(_4$,	a+a)$)	재파싱 노드
($BOS_0E_1+_6(_4a_5$,	+a)$)	
($BOS_0E_1+_6(_4F_3$,	+a)$)	
($BOS_0E_1+_6(_4T_2$,	+a)$)	
($BOS_0E_1+_6(_4E_8$,	+a)$)	
($BOS_0E_1+_6(_4E_8+_6$,	a)$)	
($BOS_0E_1+_6(_4E_8+_6a_5$,	)$)	
($BOS_0E_1+_6(_4E_8+_6F_3$,	)$)	
($BOS_0E_1+_6(_4E_8+_6T_9$	)$)	
($BOS_0E_1+_6(_4E_8$	)$)	
($BOS_0E_1+_6(_4E_8)_{11}$	$)	
($BOS_0E_1+_6F_3$	$)	생성된 노드
($BOS_0E_1+_6T_9$	$)	
(BOS_0E_1	$)	재파싱 노드



〈그림 5〉 스택을 이용한 점진적 스레드 트리

그림 5에서 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 그림 4와 같이 파싱을 시작한다. 단계 4에서 w 전체를 지배하는 노드 N과 w′ 전체를 지배하는 노드 N′ 심볼의 위치를 찾지 못하므로 단계 3을 반복 수행하여 스레드 트리 T′ 를 구성하고 단계 5에서 파싱을 종료한다.

3. 스택을 이용하지 않은 점진적 파싱

3.1 파싱을 위한 스레드 트리

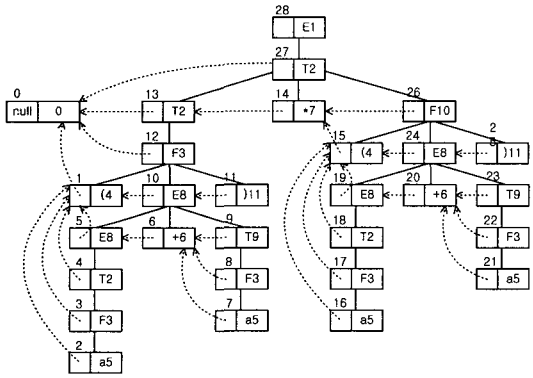
본 논문에서는 Larchevêque가 스택을 이용하여 스레드 트리 T를 구성한 것과 달리 스택을 이용하지 않고 Node를 노드 데이터(Node.data), 노드 주소(Node.address), 노드 스레드(Node.thread)로 구성한 초기 노드를 생성한다. ACTION[S, z_i]=shift이면 오른쪽 형제 노드 Node_right로 생성하고, ACTION [[S, z_i]=reduce이면 부모노드 Node_parent를 생성한다. ACTION[S, z_i] = error로 오류 회복을 위한 작업을 수행하고, ACTION[S, z_i] = accept되어 파싱을 종료한다. 따라서 입력 스트링 z_i = xwy를 스택을 제거한 스레드 트리로 구성하는 알고리즘은 제안하면 알고리즘 2와 같다.

알고리즘 2를 적용하여 문법 G_1에서 입력스트링 z=(a+a)*(a+a)에서 스택을 이용하지 않는 스레드 트리를 구성하면 그림 6과 같다.

그림 6에서 문법 G_1을 이용하면 Node.address=0, Node.data=0, Node.thread=null로 정의한 초기 노드를 생성한다. shift는 ACTION[0, (]=shift 4가 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′ =4와 결합되어 (4가 된다. 다음 노드를 생성하기 위해 Node_right를 Node로 전환한다.

```
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;
   Node.data := 0;, S := 0;, Node.thread := null;
   create Node(labeled Node.address);
(2) ACTION[S, zi] = shift
    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
    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
    Stop the parsing and signal error;
(4) ACTION[S, zi] = accept
    Terminate the parsing and signal acceptance;
```

〈알고리즘 2〉 스택을 이용하지 않는 스레드 트리 구성 알고리즘



〈그림 6〉 스택을 이용하지 않은 스레드 트리

ACTION[5, +]=reduce F→a가 되는데 Node. address=2가 증가되어 Node_parent.address= 3이 Label 된 부모노드 Node_parent를 생성한다. Node_parent의 노드 가지는 오른쪽 심볼 |α|의 길이 1만큼 가지를 연결하고, 스레드는 현재 Node로부터 |α|의 길이 1만큼 스레드를 반복하여 Node.address=0으로 스레드 한다. 다음 상태 S′ 는 Node_parent의 스레드인 노드 상태 0과 왼쪽 심볼 F가 GOTO[0, F]=3 상태로 변경된다. Node_parent.data는 왼쪽 심볼과 다음 상태가 결합되어 F3이 된다. 다음 노드를 생성하기 위해 Node_parent를 Node로 전환한다. 마지막으로 노드 E1과 $로 ACTION[1, $]= accept되어 파싱을 종료한다.

3.2 점진적 파싱을 위한 스레드 트리

Larchevêque가 스택의 push와 pop을 이용하여 점진적 파싱을 하는 것과는 달리 스택의 push와 pop을 이용하지 않고 본 논문에서는 스택을 이용하지 않는 노드 데이터 생성 알고리즘과 점진적 스레드 트리 구성 알고리즘을 제안한다.

스레드 트리 T의 노드 데이터와 비교하기 위해 점진적 스레드 트리 T′ 의 노드 데이터를 생성한다. shift이면 새로운 상태와 결합하여 노드 데이터를 생성하고, reduce이면 스레드된 노드의 상태와 결합하여 노드 데이터를 생성하는 알고리즘을 제안하면 알고리즘 3과 같다.

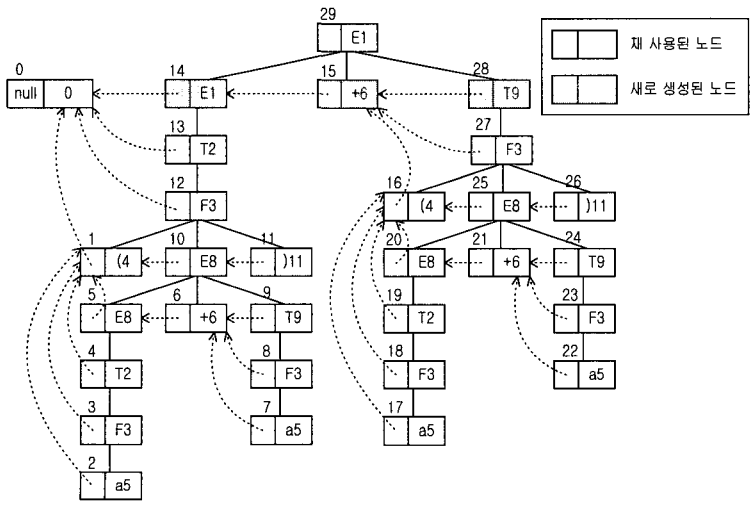
```
(1) ACTION[S, zi ′] = shift
    Node ′.address := Node ′.address + 1;
    Node ′ ^.thread := Node ′.address-1;
    Node ′ := join(zi ′, S ′);
    if Node.thread = (Node.address-1).thread then
      repeat
          Node.address := Node.address + 1;
      until shift zi;
      Nodetmp := Node
(2) ACTION[S, zi ′] = reduce A → α
    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 ′)
```

〈알고리즘 3〉 스택을 이용하지 않은 노드 데이터 생성 알고리즘

또한, 노드의 재 파싱 과정을 없애기 위해 스택을 이용하지 않는 점진적 스레드 트리 T′ 를 구성한다. 단계 1에서 스택을 제거한 스레드 트리 T가 FIRST_ REDUCE(wy$)인 노드까지 T′ 에 추가한다. 단계 2에서 z_i' =FIRST_SHIFT(y$)될 때까지 T의 노드 데이터와 알고리즘 3에 의해 생성된 T′ 의 노드 데이터를 비교하여 T의 노드 데이터와 일치하면 T′ 에 노드를 추가하고 일치하지 않으면 알고리즘 2로 T′ 의 노드를 생성한다. 단계 3에서 z_i' =LAST_SHIFT(y$)될 때까지 T의 노드와 T′ 의 노드가 일치하므로 재파싱 되는 과정 없이 노드 주소와 스레드만 변경하고 T′ 에 노드를 추가한다. 단계 4에서 z_i' =accept될 때까지 단계 2의 과정을 반복한다. 스택을 이용하지 않고 점진적 스레드 트리를 구성은 알고리즘 4와 같다.

```
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$);
     Node'.address := Node.address;
     for Node.address := 0 to Node.address do
       begin
         T' := Node;
       end;
 (2) repeat
       begin
         Node.address := Node.address + 1;
         Nodetmp := Node;
         Using Algorithm 3;
       end;
     repeat
begin
          if Nodetmp.data = Node'.data then
            begin
          if Nodetmp.data = Node'.data then
            begin
Nodetmp.address := Node'.address;
               Nodetmp.thread := Node'.thread;
               T' := Nodetmp;
               goto step 2
            end;
          else Nodetmp.address := Nodetmp.address + 1
        end;
      until shift z_i or accept;
        Node' := Node'.address-1;
        Using Algorithm 3;
      end;
   until z_i' = FIRST_SHIFT(y$) or accept;
   if z_i' = accept then go to step 5;
(3) Node.address := FIRST_SHIFT(y$);
   repeat
     begin
       Node.address := Node.address + 1;
       Nodetmp := Node;
       Using Algorithm 3;
       Nodetmp.address := Node'.address;
       Nodetmp.thread := Node'.thread;
       T' := Nodetmp
     end;
   until z_i' = LAST_SHIFT(y$);
 (4) Node.address := last_shift(y$);
    go to step 2;
 (5) Stop
```

〈알고리즘 4〉 스택을 이용하지 않은 점진적 스레드 트리 구성 알고리즘



〈그림 7〉 스택을 이용하지 않은 점진적 스레드 트리

알고리즘 3과 알고리즘 4를 적용하여 문법 G_1의 입력 스트링 z=(a+a)*(a+a)와 변경 스트링 z′ =(a+a)+(a+a)를 살펴보면 x=(a+a), y=(a+a), w=*, w′ =+ 이다. 입력 스트링 변경시 스택을 이용한 점진적 스레드 트리를 구성하면 그림 7과 같다.

그림 7에서 보면 단계 1은 그림 6의 스택을 이용하지 않는 스레드 트리 T의 FIRST_REDUCE (wy$)인 Node.address=12까지 Node를 스택을 점진적 스레드 트리 T′ 에 추가한다. 단계 2는 Node. address가 증가된 13으로부터 z′ =FIRST_SHIFT(y$)될 때까지 T의 Node.data와 알고리즘 3에 의해 생성된 T′ 의 Node′ .data를 비교하여 같으면 T의 Node를 T′ 에 추가하고 아니면 알고리즘 2로 새로운 Node를 생성한다. 단계 3은 FIRST_SHIFT(y$)인 Node.address=17로부터 z′ = LAST_SHIFT (y$)될 때까지 알고리즘 3으로 T의 Node.address와 Node. thread만 변경하여 T′ 에 추가한다. 단계 4에서 last_shift(y$)인 Node.address=26으로부터 z′ =accept 될 때까지 단계 2와 같은 방법이므로 알고리즘 2로 새로운 Node를 생성한다. 단계 4에서 ACTION[1, $]=accept이므로 단계 5에서 종료한다.

따라서, w=* 대신 w′ =+로 대체될 때 Larchevêque의 스택을 이용한 점진적 파싱 알고리즘에서는 전체 노드 30개 중에서 스택을 사용한 재파싱 노드 12개가 있으나, 스택을 이용하지 않은 점진적 스레드 트리 구성 알고리즘에서는 재파싱 노드 12개가 필요하지 않음을 알 수 있다.

3.3 성능분석

본 논문에서 제안한 스레드 트리 구성 알고리즘과 점진적 스레드 트리 구성 알고리즘의 타당성을 입증하기 위해 구현 실험해 본 결과 표 2에서 알 수 있듯이 Larchevêque의 점진적 파싱 알고리즘보다 재파싱 노드가 단축된다.

따라서 본 논문에서 제안한 점진적 스레드 트리 구성 알고리즘이 Larchevêque의 점진적 파싱 알고리즘보다 전체 파싱 노드 중 약 40%~75%의 재사용 노드와 파싱 속도는 약 44%~49%로 향상된다.

4. 결 론

점진적 파싱은 편집기 상태에서 원시 프로그램의 파싱 과정 중 변화된 부분만 재 파싱하고 같은 부분은 재 파싱을 하지 않으므로 기억 장소와 수행 시간이 많이 요구되는 프로그램 개발 시 매우 효과적일 것이다.

〈표 2〉 알고리즘의 성능분석

구분 \ 스레드 트리		스레드 트리 1	스레드 트리 2	스레드 트리 3
입력 스트링	노드 수	29	29	43
변경된 스트링	노드 수	30	28	44
스택을 이용한 알고리즘	재사용 노드	14	4	14
	재파싱 노드	12	21	26
	생성된 노드	4	3	4
	파싱 속도	0.081	0.080	0.120
스택을 이용하지 않는 알고리즘	재사용 노드	26	25	40
	재파싱 노드	0	0	0
	생성된 노드	4	3	4
	파싱 속도	0.054	0.052	0.062

(단위 : sec)

본 논문에서는 Larchevêque가 스택을 이용하여 스레드 트리를 구성한 것과는 달리 노드 재사용을 최대화하기 위해 스택을 이용하지 않는 스레드 트리 구성 알고리즘과 점진적 스레드 트리 구성 알고리즘을 제안하였다.

Larchevêque의 점진적 파싱 알고리즘과 본 논문에서 제안한 점진적 파싱 알고리즘을 비교 분석한 결과 Larchevêque의 알고리즘이 입력 스트링 변경 시 전체 파싱 노드 중 약 40%~75%의 재사용 노드와 파싱 속도는 약 44%~49%로 향상된다. 따라서 본 논문에서 제안한 점진적 스레드 트리 구성 알고리즘은 프로그램의 수정뿐만 객체를 끌어다 사용할 경우 객체지향 프로그램 환경에서 효율적으로 이용될 수 있다.

참 고 문 헌

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

[2] Morris, J. M. and Schwartz, M. D., "The Design of a Language-Directed Editor for Block-Structured Languages" , ACM SIGPLAN Notices, Vol.16, No.6, pp. 28-33, 1981.

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

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

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

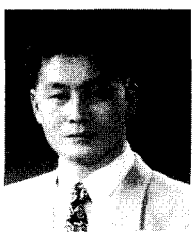
[7] Li, W. X., "A Simple and Effect Incremental LL(1) Parsing" , In SOFSEM '95: Theory and Practice of Informatics, Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 399-404, 1995.

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

[9] Johnson, S. C., "YACC - Yet Another Compiler-Compiler" , CSTR32, AT&T, Bell Labs., Murray Hill, N. J., 1975.

[10] 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.

◐ 저 자 소 개 ◑



이 대 식

1995년 관동대학교 전자계산공학과 졸업(학사)
1999년 관동대학교 대학원 전자계산공학과 졸업(석사)
2004년 관동대학교 대학원 전자계산공학과 졸업(박사)
2003~현재 관동대학교 멀티미디어공학부 겸임교수
관심분야 : 컴파일러, 프로그래밍 언어, 데이터베이스, etc.
E-mail : lds@kd.ac.kr