변화된 스레드 트리를 이용한 점진적 LR 파싱 알고리즘 구현 및 설계

이 대 식*

요 약

스레드 트리란 LR 파싱표를 사용하여 파스 트리인 동시에 파스 스택을 표현 할 수 있는 자료 구조이다. Larchevêque는 스택을 사용하여 스레드 트리를 구성하고 점진적 파싱을 한다. 본 논문에서는 재 파싱 노드와 파싱속도를 줄이기 위해 스택을 사용하지 않는 변화된 스레드 트리를 구성하는 알고리즘을 제안한다. 또한 노드의 재 파싱 과정을 없애기 위해 변화된 스레드 트리와 LR 파싱표를 사용하는 점진적 파싱 알고리즘을 제안한다.

On Design and Implementation of Incremental LR Parsing Algorithm Using Changed Threed Tree

Dae Sik Lee*

ABSTRACT

Threaded Tree is the data structure that can express parse stack as well as parse tree with LR parsing table. Larchevêque makes Threaded Tree and Incremental Parsing with stack. This paper suggests the algorithm consisting of changed threaded tree without stack in order to reduce reparsing node and parsing speed. Also, it suggests incremental parsing algorithm to get rid of the reparsing process in node.

Key words : Parsing, Threaded Tree

* 안동과학대학 사이버테러대응과

1. 서 론

프로그래밍 환경들은 복잡한 소프트웨어 시스템을 개발하기 위해 프로그래머의 입력에 따라 분석과 정보의 제공이 즉시 이루어지고, 생산성을 향상시키는 중요한 구성 요소는 점진적 파싱(incremental parsing)이다[1-3].

점진적 파싱에 대한 기존의 연구는 Celentano의 히스토리(history) 정보를 기록하기 위한 스택 트리[4], Ghezzi와 Mandrioli는 Celentano의 스택 트리를 개선하여 파스 트리 상에서 점진적 파싱을 할 수 있도록 스레드 트리를 제시하였다[5].

Larchevêque는 Ghezzi와 Mandrioli가 제안한 SLR(Simple LR) 기반의 스레드 트리를 LALR (Lookahead LR) 형태로 이용하였으며, 특히 스레드 트리에서 스택의 push와 pop을 사용하여 노드의 재사용을 통한 최적화된 점진적 파서의 구성에 대한 개념을 제시하였다[6].

본 논문에서 점진적 파싱의 목표는 빠른 파싱과 노드 재사용을 최대화하여 재 파싱 노드와 기억장소의 낭비를 줄이는 것이다. 따라서 편집기 상태에서 원시 프로그램의 파싱 과정 중 프로그램의 변화된 부분만 재 파싱하고 같은 부분은 재 파싱을 하지 않기 위하여 파싱 스택을 사용하지 않는 스레드 트리(이하 변환된 스레드 트리라 함)를 구성하는 알고리즘을 제안하고, 수정된 스레드 트리를 사용한 점진적 파싱 알고리즘을 제안한다. 특히, 효율적으로 점진적 파싱하기 위하여 변화된 스레드 트리를 사용한 점진적 파싱 알고리즘을 구현하여 비교 분석해 보고자 한다.

2. 이론적 배경

2.1 일반적인 파싱

대부분의 프로그래밍 언어 구조는 문맥 자유 문법에 의하여 정의되는 하나의 본질적인 순환 구조를 갖는다[7, 8].

> 문맥 자유 문법(이하 문법이라 함)은 G=(N, T, P, S)로 표현된다.
> 여기서,
> N : 비단말 기호(nonterminal symbol)의 유한 집합
> T : 단말 기호(terminal symbol)의 유한 집합
> P : 생성 규칙(production rule)의 유한 집합
> $A \rightarrow a$ 단, $A \in N$, $a \in (N \cup T)^*$
> S : 시작 기호(start symbol)
> 단, $S \in N$

(정의 1) 문맥 자유 문법

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

일 때, LR 파싱표[9]는 <표 1>과 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				

문장(sentence)에 대한 파싱 과정은 파서 configuration의 순서로 표현되는데 LR 파서의 configuration은 쌍(S, x$)으로 구성된다. 여기서, $S=S_0S$

1… S_m은 톱(top)으로 S_m을 가진 스택의 내용이고 x$는 남아있는 입력이며, $는 오른쪽 끝마커(right endmaker)이다. 문법 G와 G에 대한 LR 파서가 주어졌을 때, 각 문장 z ∈ 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 상태이다.

문법 G_0에서 입력 스트링 fgghf에 대한 파싱은 (그림 1)과 같다.

스 택	입 력	단 계
$\Pi = (S_0,$	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) G_0에서 z=fgghf에 대한 파스 순서 Π

2.2 Celentano의 기본 알고리즘

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, $\Pi_0=(S_0, z\$)$, $\Pi_0'=(S_0, z'\$)$, $\Pi_n=\Pi_m'=(S_0\ S_f, \$)$이다.

문법 G_0의 두 문장 z=fgghf와 z′=fghf를 살펴보면 x=fg, y=hf, w=g, w′=ε 일 때 점진적으로 계산된 파스순서 Π′는 (그림 2)와 같다.

스 택	입 력	단 계
$\Pi' = (S_0,$	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) G_0에서 z′=fghf에 대한 점진적 파스 순서 Π와 Π′

따라서, 점진적 파싱을 수행하기 위하여 파싱 과정 동안 상태 스택에 대한 여러 히스토리 정보와 파스 결과가 사용될 수 있다. 그러나 Celentano의 점진적 파싱 알고리즘은 파스 순서를 계산하고 저장하는데 너무나 비경제적이므로 이 알고리즘을 직접 구현하는 것은 적합하지 않다.

2.3 Larchevêque의 파싱

2.3.1 파스 순서에 따른 스레드 트리

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

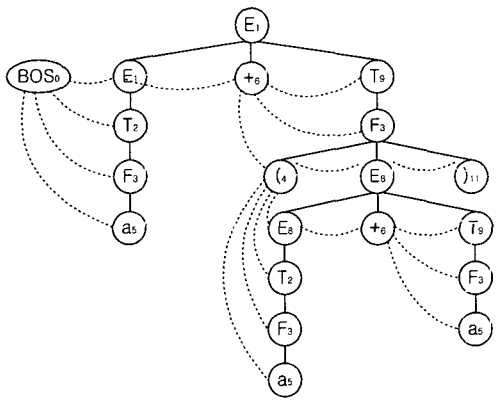
문법 G_1 :
(1) E → E+T
(2) E → T
(3) T → T*F
(4) T → F
(5) F → (E)
(6) F → a

일 때, LR 파싱표는 <표 2>와 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			

스 택	입 력
(BOS_0,	a+(a+a)$)
(BOS_0a_5,	+(a+a)$)
(BOS_0F_3,	+(a+a)$)
(BOS_0T_2,	+(a+a)$)
(BOS_0E_1,	+(a+a)$)
($BOS_0E_1+_6$,	(a+a)$)
($BOS_0E_1+_6(_4$,	a+a)$)
($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$,	$)
($BOS_0E_1+_6F_3T_9$,	$)
(BOS_0E_1,	$)



(그림 3) G_1에서 z=a+(a+a)에 대한 스레드 트리

문법 G_1에서 입력 스트링 z=a+(a+a)에 대한 파스 순서로 스택의 초기 상태인 BOS_0(Bottom Of Stack)을 시작으로 스레드 트리를 표현하면 (그림 3)과 같다.

(그림 3)에서 shift, reduce, accept 단계를 보면 초기노드 BOS_0의 상태 0과 lookahead a로 ACTION[0, a]=shift 5가 노드 a_5를 생성한다. 노드 a_5를 스택에 push 하고 선행자 노드 BOS_0에 스레드 한다. 노드 a_5의 상태 5와 lookahead +로 ACTION[5, +]=reduce F→a가 된다. 왼쪽 심볼로 부모 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 BOS_0로 스레드 한다. 노드 F의 상태는 GOTO[0, F]=3 상태이므로 노드 F는 F_3이 된다. 마지막 노드 E_1의 상태 1과 lookahead $로 ACTION[1, $]=accept으로 파싱을 종료한다.

2.3.2 스레드 트리를 사용한 파싱

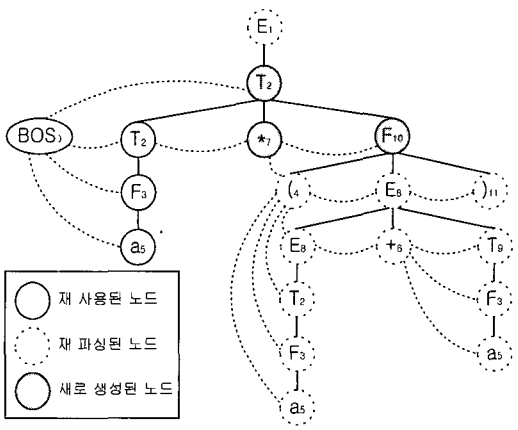
Larchevêque의 스레드 트리를 사용한 점진적 파싱 알고리즘을 제시하면 (알고리즘 1)과 같다.

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 다시 시작한다.
(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 종료한다.

(알고리즘 1) 스레드 트리에서 점진적 파싱 알고리즘

(알고리즘 1)을 적용하여 문법 G_1의 두 문장 z=a+(a+a)와 z′=a*(a+a)를 살펴보면 x=a, y=(a+a), w=+, w′=*일 때 점진적으로 계산된 파스순서에 대한 스레드 트리는 (그림 4)와 같다.

스 택	입 력
(BOS_0,	a*(a+a)$)
(BOS_0a_5,	*(a+a)$)
(BOS_0F_3,	*(a+a)$)
(BOS_0T_2,	*(a+a)$)
($BOS_0T_2*_7$,	(a+a)$)
($BOS_0T_2*_7(_4$,	a+a)$)
($BOS_0T_2*_7(_4a_5$,	a+a)$)
($BOS_0T_2*_7(_4F_3$,	a)$)
($BOS_0T_2*_7(_4T_2$,	+a)$)
($BOS_0T_2*_7(_4E_8$,	+a)$)
($BOS_0T_2*_7(_4E_8+_6$,	+a)$)
($BOS_0T_2*_7(_4E_8+_6a_5$,	)$)
($BOS_0T_2*_7(_4E_8+_6F_3$,	)$)
($BOS_0T_2*_7(_4E_8+_6T_9$,	)$)
($BOS_0T_2*_7(_4E_8$,	)$)
($BOS_0T_2*_7(_4E_8)_{11}$,	$)
($BOS_0T_2*_7F_3$,	$)
(BOS_0T_2,	$)
(BOS_0E_1,	$)



(그림 4) G_1에서 z=a·(a+a)에 대한 스레드 트리

(그림 4)에서 보면 단계 1로부터 last(x)인 a를 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 3)과 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. 변화된 스레드 트리를 사용한 파싱

3.1 변화된 스레드 트리 구성을 위한 파싱

본 논문에서는 Larchevêque가 스택의 push와 pop을 사용하여 스레드 트리 T를 구성한 것과 달리 입력 스트링 z_i = xwy로 생성되는 변화된 스레드 트리 T의 노드는 노드 데이터(Node.data), 노드 주소(Node.address), 노드 스레드(Node.thread)로 구성한다. start_node는 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 변화된 스레드 트리를 구성하는 알고리즘은 (알고리즘 2)와 같다.

```
input  : input string z
output : threaded tree T
method : Node
         S : current state
         S′ : next state
         zi = xwy : input string
if start_node then
  begin
    Node.address := 0;
```

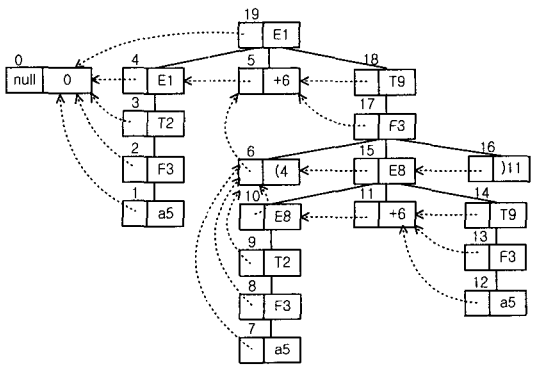
```
    create Node(labeled Node.address);
    node.data := 0;
    S := 0;
    Node.thread := null
  end;
  else if ACTION[S, zi] = shift then
    begin
     Node_right.address := Node.address + 1;
     create Node_right(labeled Node_right.address)
     to Node;
     Node_right↑.thread := Node.address;
     Node_right.data := join(zi, S′);
     Node := Node_right
   end;
   else if ACTION[S, zi] = reduce A → α then
      begin
        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
    end;
    else if ACTION[S, zi] = error then
      Stop the parsing and signal error;
     else if ACTION[S, zi] = accept then
   Terminate the parsing and signal acceptance;
```

(알고리즘 2) 변화된 스레드 트리 사용한 파싱 알고리즘

(알고리즘 2)를 적용하여 입력스트링 z=(a+a)*(a+a)에 대한 변화된 스레드 트리를 표현하면 (그림 5)와 같다.

입력스트링 z=(a+a)*(a+a)에 대한 변화된 스레드 트리를 표현하면 (그림 5)와 같다.

(그림 5)에서 shift, reduce, accept 단계를 보면 start_node는 Node.address=0이 Label된 노드를 생성한다. Node. data=0, Node.thread=null로 정의한 초기 노드를 생성한다.



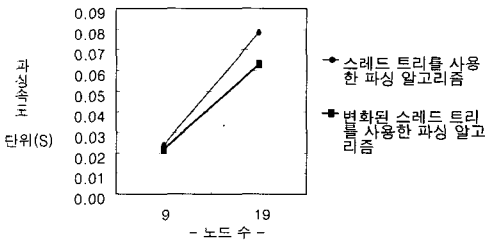
(그림 5) G_1에서 z=a+(a+a)에 대한 변화된 스레드 트리 T

ACTION[0, a]=shift 5가 되는데 Node.address =0 이 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 a_5가 된다. 다음 노드를 생성하기 위해 Node_right로 생성된 노드 1을 Node로 전환한다.

ACTION[5, *]=reduce F→a가 되는데 Node. address=1이 증가되어 Node_parent.address=2가 Label된 부모노드 Node_parent를 생성한다. Node_parent의 노드 가지는 오른쪽 심볼 $|\alpha|$의 길이 1만큼 가지를 연결하고, 스레드는 현재 Node로부터 $|\alpha|$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_parent로 생성된 노드 3을 Node로 전환한다. 마지막으로 ACTION[1, $]= accept되어 파싱을 종료한다.

3.2 성능 평가

본 논문에서 제시한 변화된 스레드 트리를 사용한 파싱 알고리즘의 타당성을 입증하기 위해 구현 실험해 본 결과 (그림 6)에서 알 수 있듯이 변화된 스레드 트리를 사용한 파싱 알고리즘이 Larchevêque의 스레드 트리를 사용한 파싱 알고리즘보다 파싱속도가 단축되는 것을 알 수 있다.



(그림 6) 파싱속도

4. 결 론

Larchevêque가 스택을 사용하여 스레드 트리를 구성한 것과는 달리 파싱속도를 줄이기 위해 스택을 사용하지 않는 변화된 스레드 트리를 사용한 파싱 알고리즘을 제안하였다.

스택을 사용하여 스레드 트리를 구성한 것과는 달리 변화된 스레드 트리를 사용한 파싱 알고리즘은 파싱 속도는 약 9%~38%로 향상된다.

참 고 문 헌

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

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

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

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

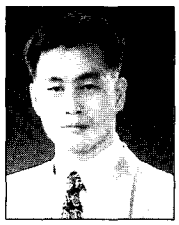
[5] C. Ghezzi, and D. Mandrioli, "Incremental Parsing", ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, pp. 58-70, 1979.

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

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

[8] 안희학, "LR 파서를 위한 효율적인 점진적 파싱", 한국정보처리학회 논문지, 제5권, 제6호, pp. 1660-1669, 1998.

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



이 대 식

1995년 관동대학교 전자계산공학과(공학사)

1999년 관동대학교 전자계산공학과(공학석사)

2004년 관동대학교 전자계산공학과(공학박사)

2003년~현재 안동과학대학 사이버테러대응학과 교수