論文94-31B-4-2

효율적인 점진적 LR 파싱 알고리듬

(An Efficient Incremental LR Parsing Algorithm)

安 熙 學*, 劉 載 祐**, 宋 後 鳳**

(Heui Hak Ahn, Chae Woo Yoo and Hoo Bong Song)

要 約

점진적 파싱 기법은 프로그램의 점진적 구성을 허용하는 언어기반 환경의 중요한 부분이며, 프로그램의 변경된 부분에 대해서만 구문분석을 다시 함으로써 시스템의 성능을 향상 시킨다. 점진적 파싱과 점진적 파서의 구성을 위하여 수많은 방법과 알고리듬이 제안 되었다. 본 논문의 목적은 시간과 기억장소를 너무 많이 요구하는 기존의 점진적 파싱 알고리듬을 정밀히 조사하여, 기존의 알고리듬들보다 효율적인 점진적 LR 파싱 알고리듬을 제시하는데 있다. 본 논문에서는 입력으로 단말기호는 물론 비단말기호까지 허용하는 확장형 LR 파싱표를 자동으로 생성하도록 하고, 이를 효율적인 점진적 LR 파싱 알고리듬에 적용 하였다. 이 확장형 LR 파싱표를 이용하여 파싱단계는 물론 기억장소를 감소시키도록 여러 가지 방법을 제안하였다. 본 논문에 제시된 알고리듬은 SUN UNIX OS환경에서 C언어와 YACC 으로 구현되었고, 여러가지 문법과 스트링으로 시험하였다.

Abstract

The incremental parsing techniques are essential part of language-based environment which allow incremental construction of programs, as reparsing of modified part of the program alone enhances the system performance. A number of methods and algorithms have been proposed for incremental parsing and for the construction of incremental parsers. The purpose of this paper is to review the earlier incremental parsing algorithms which are too expensive in both time and space, and to present an efficient incremental LR parsing algorithm which is more efficient than the previous ones. In this paper, we generate automatically an extended LR parsing tables which allow the nonterminal symbols as well as terminal symbols for the input, and apply them to our efficient incremental LR parsing algorithm. Using these extended LR parsing tables, we suggest several methods to reduce its memory spaces and parsing steps as well. The algorithms described here were implemented in C language and YACC on a SUN UNIX OS, and were tested with several grammars and strings.

* 正會員, 關東大學校 電子計算工學科
(Dept. of Comp. Sci., Kwandong Univ.)
** 正會員, 崇實大學校 電子計算學科
(Dept. of Comp. Sci., Soongsil Univ.)
接受日字 : 1994年 1月 3日

Ⅰ. 서 론

LR 문법은 오늘날 프로그래밍 언어를 형식화하여 표기한 문법중에서 가장 강력하고 실용적인 문법으로 평가되고 있으며, LR 부류의 문법에 대한 많은 연구와 기술의 적용으로 파서 작성을 자동화하는 단계에 이르렀다. UNIX에서 제공되는 YACC등은 대표적인 파서 생성기이다.[1,3]

일괄 처리 방식의 프로그래밍 환경보다는 대화식 프로그래밍 환경에 대한 연구가 1978년 부터 시작하여 1980년대에 활발히 진행되었다.[4] 기존의 프로그램 개발은 프로그램을 편집, 컴파일, 실행의 작업을 계속하여 반복적으로 수행하는 반면, 새로운 프로그래밍 환경에서는 단말기 또는 워크스테이션(workstation)에서 프로그램 중심적인 생성(create), 편집(edit), 실행(excute), 디버깅(debugging)의 종합적인 기능을 대화식으로 행할 수 있도록 환경을 갖춘 시스템을 구상하였다. 언어 기반 편집기(language-based editors)의 개발에 따라 점진적 파싱의 문제에 많은 관심을 불러 일으키고 있다. 점진적 파싱과 점진적 파서의 구성을 위하여 수 많은 방법과 알고리듬이 제안되었다.[5,10] 언어기반 편집기는 점진적 LR(1) 파싱 알고리듬을 이용하여 확립되었다.

점진적 LR 파싱을 수행하기 위하여 파싱과정 동안 상태스택에 대한 여러 히스토리 정보와 파스결과가 이용될 수 있다. 점진적 LR 파싱 알고리듬의 계산 복잡도는 히스토리 정보를 저장하기 위해 사용된 자료구조에 의존한다. Celentano[5]는 히스토리 정보를 기록하기 위하여 스택트리를 이용한 점진적 LR 파싱 알고리듬을 제안하였다.

점진적 파싱은 원시 프로그램중에서 변경된 부분 즉 삭제, 삽입, 대치 등이 이루어진 부분으로 인하여 발생하는 재 번역 과정중 가능한한 적은 부분만을 다시 처리하고자 하는 것이다. 특히, Celentano의 논문은 최근까지도 그 가치가 인정되어 연구가 진행되고 있다. Celentano는 처음으로 점진적 LR 파서를 소개하였는데, 그의 알고리듬은 너무 많은 기억장소를 요구하고 수행시간이 길다는 단점 때문에 파싱과정을 트리 구조로 변경하여 표현하는 방법을 제시하였다.

Ghezzi와 Mandrioli[6,7]는 결정적 문맥 자유 언어(deterministic context-free languages)에 대한 점진적 파싱 개념을 소개했으며, LR이고 RL인 문법에 대한 점진적 파싱 알고리듬을 제시했다. Agrawal과 Detro[8]는 Celentano의 점진적 LR 파싱 알고리듬에 ε -생성규칙을 허용하는 문법에 대한 확장방법을 제안하였다.

Yeh[9]는 일반적인 상향식 파싱 방법인 shift-reduce파서와 점진성을 지지하는 LR(0) 파서를 증가시키는 알고리듬을 제시하였다. Yeh와 Kastens[10]은 LR(1)문법에서 ε -생성규칙은 물론 입력의 다중 변경까지 허용하는 점진적 LR(1) 파싱 알고리듬을 제시했다. Rekers와 Koorn[11]은 점진적 파싱에서 서브 스트링 파서의 이용에 대하여 연구하였다. 최근의 연구에서는 기존의 LR 파싱표와 달리 단말기호와 비단말기호 모두를 입력 스트링으로 받아들이는 확장형 LR 파싱표를 정의하고 계산하는 방법과, 이러한 확장형 LR 파싱 표를 이용하는 점진적 파싱 알고리듬을 제안 하였다.[12]

이러한 점진적 파싱 기술을 기초로한 대화식 프로그래밍 환경에 관한 대표적인 것으로 Carnegie-Mellon 대학에서 설계되고 구현된 IPE(Incremental Programming Environment)[4], Cornell 대학의 CPS (Cornell Program Synthesizer)[13,14] 등이 있다.

본 논문에서는 Celentano의 점진적 파싱 알고리듬을 분석하여 그 특징을 고찰하고, 파스칼과 같이 블럭구조의 복잡한 문법을 갖는 일반적인 프로그래밍 언어에서 효율적으로 사용될 수 있는 점진적 파싱 방법을 연구하며, Celentano의 점진적 파싱 알고리듬보다 기억장소와 파싱단계를 상당히 감소시킬 것으로 생각되는 효율적인 점진적 LR 파싱 알고리듬을 제안하였다.

특히, YACC을 이용하여 단말기호와 비단말기호 모두를 입력 스트링으로 받아들이는 확장형 LR 파싱표를 자동으로 생성하였으며, 이 자동으로 생성한 LR 파서 생성기를 이용하여 효율적인 점진적 LR 파싱 알고리듬을 검증하기 위하여 SUN UNIX OS 환경에서 C언어로 실제 구현하였다.

본 논문의 구성은 다음과 같다.

Ⅱ장에서는 여러가지 기본적인 표기방식과 정의에 대하여 설명하고, Ⅲ장에서는 기존의 점진적 파싱 알고리듬을 제시하고, Ⅳ장에서는 효율적인 점진적 파싱 알고리듬을 제안하고, 기존의 점진적 알고리듬과의 성능평가를 하여 실험결과를 논한다. 마지막으로 Ⅴ장에서 결론을 맺는다.

Ⅱ. 정의

기본적인 표기방식과 정의는 Aho와 Ullman[1] 방식에 기본을 둔다.

대부분의 프로그래밍 언어 구조들은 문맥 자유문법(context-free grammar)에 의해 정의 될 수 있는 하나의 본질적인 순환 구조를 가진다.

(정의 1) 문맥 자유문법 G=(N,T,P,S)은 다음의 4

가지 요소로 정의된다.

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 → α , A는 비단말 기호, α 는 (NUT)* 로부터의 스트링

S : 시작 기호 (start symbol)
이때, ∀ S ∈ N

LR 문법은 현재 실용적으로 사용되고 있는 다른 어떤 문법보다도 프로그래밍 언어의 파서를 구성하는데 있어서 가장 강력하고 효율적으로 사용될 수 있다.

파싱 과정은 파서 configuration의 순서로 표현될 수 있다.

LR 파서의 configuration T는 쌍(S, x$)이다. 여기서, $S = S_0\ S_1 \cdots S_m$ 은 top으로 S_m을 가진 스택의 내용이다. x$는 남아있는 입력이고, $는 오른쪽 끝 기호(right end marker)이다. LR 문법 G와 G에 대한 LR 파서가 주어졌을때, 각 문장 z∈L(G)에 대하여, $T_0 = (S_0, z\$)$, $T_n = (S_0S_f, \$)$인 파서 configuration $T = T_0\ T_1 \cdots T_n$이 존재한다. 여기서, S_0는 초기 상태, S_f는 $g(S_f, \$)$=accept인 상태이고, $0 \le i \le n$인 모든 i에 대하여 $T_i \vdash T_{i+1}$이다.

(예 1) 문맥 자유문법 G가

(1) U → with V W
(2) U → s
(3) V → a V′
(4) V′ → , V
(5) V′ → ε
(6) W → , V W
(7) W → do U

일때 LR 파싱표(parsing table)는 그림 1과 같다.

STATE	ACTION							GOTO			
	with	s	a	,		do	$	U	V	V′	W
0	s2	s14						1			
1							acc				
2			s4						3		
3					s6	s7					5
4				s11	r5	r5				10	
5							r1				
6			s4						8		
7	s2	s14						13			
8					s6	s7					9
9							r6				
10					r3	r3					
11			s4						12		
12					r4	r4					
13							r7				
14							r2				

그림 1. G에 대한 LR 파싱표
Fig. 1. LR parsing table for G.

문장 z = "with a, a do s"에 대한 파스순서(T)는 그림 2와 같다.

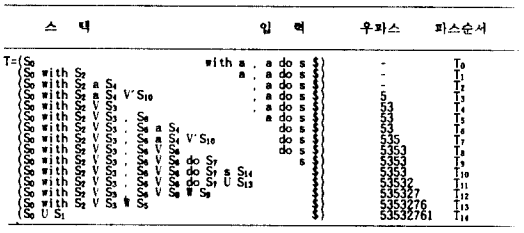


그림 2. G에 대한 파스 순서
Fig. 2. Parse sequences for G.

Ⅲ. 점진적 파싱 알고리듬

1. Celentano의 기본 알고리듬

점진적 파싱이란 프로그램의 어느 부분을 수정했을 때 전체 프로그램을 처음부터 분석하지 않고 변경된 부분(삽입, 삭제, 대치등이 이루어진 부분)에 의하여 영향받는 최소한의 부분만을 재 분석하는 파싱 방법이다.

즉, z=xwy를 문법 G에 의해 생성되는 스트링이라 하고 z′=xw′y를 w에서 w′으로 대치하여 수정된 스트링이라 하면 $w' \neq w$, $z' \in L(G)$ 이다. 파스순서 $T=T_0\ T_1 \cdots T_n$을 z와 관계있다고 하고, 파스순서 $T'=T_0'\ T_1' \cdots T_m'$을 z′과 관계있다고 하자. 여기서, $T_0=(S_0, z\$)$, $T_0'=(S_0, z'\$)$, $T_n=T_m'=(S_0S_f, \$)$이다. 대응되는 파스트리에 대하여, 점진적 파싱 알고리듬의 목적은 z′에 대한 파스트리를 구하기 위하여 새로 만들어야 하는 z에 대한 파스트리의 가장 적은 부분트리를 찾는 것이다.[5]

Celentano는 다음과 같이 점진적 파싱 알고리듬을 제시하였다.

[Algorithm 1] 점진적 LR 파싱.

입 력 : 두 스트링 z=xwy와 z′=xw′y 그리고 z에 대한 파스순서 T, $T=T_0\ T_1 \cdots T_n$

출 력 : z′에 대한 파스 순서 T′, $T'=T_0'T_1' \cdots T_m'$

방 법 : $x=x_1\ x_2 \cdots x_x$, $y=y_1\ y_2 \cdots y_y$, $w=w_1\ w_2 \cdots w_w$라 할때
p와 r을 $T_p=(S_0 \cdots S_p, wy\$)$,
$T_{p-1}=(S_0 \cdots S_{p-1}, x_xwy\$)$,
$T_r=(S_0 \cdots S_r, y\$)$,
$T_{r-1}=(S_0 \cdots S_{r-1}, w_wy\$)$인 두 인덱스라 하자.
초기화로 T′ = { } 이다.
(1) 모든 i, $0 \le i \le p$에 대하여 $T_i=(\tau_i, t\$)$

라 하자.

T′에 configuration $T_i'=(\tau_i, t\ \$)$를 추가하라.

여기서 t′은 w를 w′으로 대치함으로써 t로부터 얻어진다.

(2) T′에 $T_{p+1}'+T_{p+1}'$, $0<i\le k$인 상태 T'_{p+i}를 추가하라.

여기서 k는 $T_{p+k}'=(\tau, y\$)$인 최소 값.

(3) T_i'를 T′에 추가된 마지막 configuration이라 하자.

만약 $T_i'=T_{n+q}$, $0\le q\le n-r$인 q가 존재한다면 단계 5로 가고, 그렇지 않으면 단계 4로 가라.

(4) z′에서 파싱단계를 수행하여 얻어진 새로운 configuration을 T′에 추가하고 단계 3으로 되돌아 가라.

(5) 모든 j, $0<j\le q$ 에 대해서 $T_{t+j}'=T_{n+q+j}'$가 되도록 m=t+q라 하자.

(예 2) 문법 G에서 알고리듬 1에 따라 z="with a . a do s"가 z ="with a . a do s"로 변경될 때 점진적으로 계산된 파스순서 T′을 그림 3에 나타냈다.

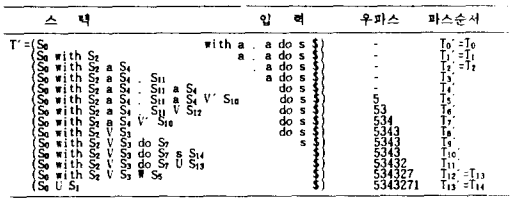


그림 3. G에 대한 점진적 파스 순서

Fig. 3. Incremental parse sequences for G.

2. 파스순서에 대한 자료구조

Celentano는 파스 히스토리를 저장하기 위하여 트리구조로 표현하는 새로운 자료 구조를 제안하였다.[5] 스택은 노드가 상태로 명칭을 부친 트리로 표현된다. 루트는 초기 상태 S_0를 포함하고, 입력은 토큰의 순서로 표현된다. 각 토큰과의 결합은 트리의 노드를 가르키는 포인터의 순서 리스트이다. 각 포인터는 configuration을 나타낸다. 파스 트리의 루트로 부터 토큰이 가르키는 노드까지의 경로는 스택 구성요소를 나타낸다.

(예 3) 문법 G에서 z="with a . a do s"에 대한 파스순서 T를 트리구조로 표현하면 그림 4 와 같다.

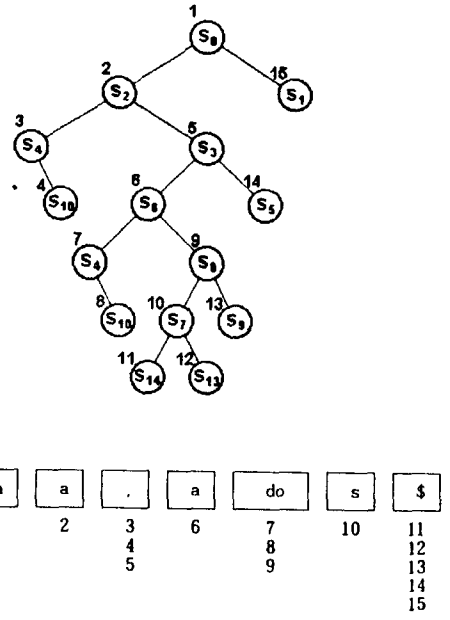


그림 4. G에 대한 트리 구조

Fig. 4. Tree structures for G.

(예 4) 문법 G에서 z="with a . a do s"가 z = "with a . a do s"로 변경될 때 점진적으로 계산된 파스순서 T′을 트리구조로 나타내면 그림 5와 같다.

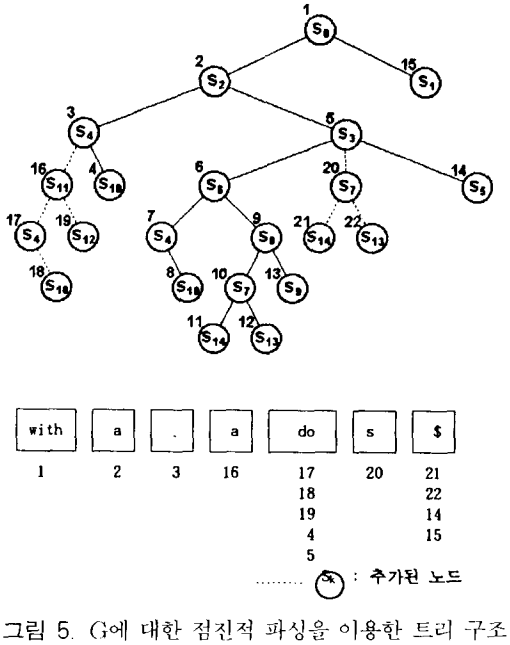


그림 5. G에 대한 점진적 파싱을 이용한 트리 구조

Fig. 5. Tree structures using incremental parsing for G.

Ⅳ. 효율적인 점진적 파싱 알고리듬

1. 확장형 파싱표

기존의 파서는 파싱표에 따라 shift, reduce, error, accept등의 일을 하는데 파싱표는 ACTION 함수와 GOTO 함수로 구성되어 있고, ACTION 함수는 $S_m \times (T \cup \{\$\})$, GOTO 함수는 $S_m \times N$에 해당하는 항목들만 존재한다. 최근의 연구에서는 기존의 파서를 확장하여 입력 기호를 받아들이는 단계에서 단말기호뿐 아니라 비단말기호까지 허용할 수 있도록 설계하 였다. [12] 즉, ACTION 함수는 $S_m \times (X_i \cup \{\$\})$, 여기서 X_i는 단말기호 또는 비단말기호가 되도록 ACTION 함수를 확장 하였다. 새로운 ACTION 함수를 구성하기 위하여 FIRST와 FOLLOW의 정의를 확장한 EFIRST와 EFOLLOW를 다음과 같이 정의한다.

(정의 2)

EFIRST(α) = {X | $\alpha \overset{*}{\Rightarrow} X\beta$ or $\alpha \overset{*}{\Rightarrow} \epsilon$ 인 경우는 ϵ }

EFOLLOW(A) = {X | $S \overset{*}{\Rightarrow} wA\gamma$ and X = EFIRST(γ)}

즉, EFIRST(α)는 α 에서 유도 될 수 있는 첫 단말기호 또는 비단말기호, EFOLLOW(A)는 문장 형태(sentential form) 중에서 A 뒤에 나올 수 있는 첫 단말기호 또는 비단말기호를 의미한다. 본 논문에서는 EFIRST와 EFOLLOW에 기본을 두고 알고리듬 2에 따라 UNIX 환경에서 제공되는 YACC과 C를 이용하여 확장형 LR 파싱표를 자동으로 생성한다.

[Algorithm 2] 확장형 LR 파싱표의 구성

입력: 첨가 문법 G′

출력: G′에 대한 확장형 LR 파싱표(ACTION 함수)

방법: (1) C={$I_0, I_1, \cdots, I_n$}을 구성한다.

(2) 상태 i는 I_i로부터 구성된다. 상태 i에 대한 구문분석 행동은 다음과 같이 정의한다.

① 만약 [A→α ·aβ] 가 I_i에 있고, GOTO(I_i,a)=I_j이면 ACTION [i,a] ="shift j" (a∈T)

② 만약 [A →α ·Bβ] 가 I_i에 있고, GOTO(I_i,B) = I_j 이면 ACTION [i,B] = "shift j"

③ 만약 [A→α ·] 가 I_i에 있다면 EFOLLOW(A)되는 모든 X에 대하여 ACTION [i,X] = "reduce A→α " (A≠S′)

④ 만약 [S →S·] 가 I_i에 있다면 ACTION [i, $] ="accept"

(3) 규칙 (2)에서 정의되지 않은 파싱표의 나머지 모든 항은 "error"

(4) 파서의 초기상태 I_0는 [S′→·S] 를 포함한다.

(예 5) 문법 G에서 EFOLLOW는 그림 6에, 확장형 LR 파싱표는 그림 7에 나타낸다.

	FOLLOW	EFOLLOW
U	{ $ }	{ $ }
V	{ , do }	{ , do W }
V′	{ , do }	{ , do W }
W	{ $ }	{ $ }

그림 6. G의 EFOLLOW

Fig. 6. EFOLLOW of G.

STATE	with	s	a	.	.	do	U	V	V′	W	$
0	s2	s14					s1				
1											acc
2			s4					s3			
3					s6	s7				s5	
4				s11	r5	r5			s10	r5	
5											r1
6			s4					s8			
7	s2	s14					s13				
8					s6	s7				s9	
9											r6
10					r3	r3				r3	
11			s4					s12			
12					r4	r4				r4	
13											r7
14											r2

그림 7. G에 대한 확장형 파싱표

Fig. 7. Extended LR parsing table for G.

2. 효율적인 알고리듬

본 논문에서는 입력스트링 z=xwy가 z′=xw′y로 변경될 경우 점진적 파싱을 하는데 단말기호뿐 아니라 비단말기호까지도 받아들이는 확장형 LR 파싱표를 이용하는 효율적인 점진적 파싱 알고리듬을 다음과 같이 제시한다.

[Algorithm 3] A new 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 to point to the node Q (labeled S_m) of the tree. X_i is the current input string , S_m is the state on top of the stack T is a

terminal symbols, N is a
nonterminal symbols
case 1 : ACTION(S_m, X_i) = **shift**
(a) if Xi = T then
S_m := ACTION(S_m, X_i);
if TOP↑.son = S_m
then TOP := TOP↑.son
else
begin
create Q' (labeled S_m)
to Q; TOP := Q'
end;
X_i X_{i+1}.
(b) if X_i = N then
S_m := ACTION(S_m, X_i);
if TOP↑.son = S_m then
begin
Q link last node(X_i)
↑.left;
replace last_node(X_i)
by node(TOP);
TOP := TOP↑.son
end
else
begin
Q link last_node(X_i)
↑.left;
replace last_node(X_i)
by node(TOP);
create Q' (labeled S_m) to Q;
TOP := Q'
end;
X_i X_{i+1}.
case 2: ACTION(S_m, X_i) = **reduce** $A \rightarrow \alpha$
Q := TOP - | ∝ | ;
S_m := ACTION(S_m, A);
if Q↑.son = S_m then TOP := Q↑.son
else
begin
create Q' (labeled S_m)
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] New 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);
Initialize y of X' by a nonterminal symbols;
TOP := first_node(X_i);
(2) node(X_j') := node(TOP);
if X_j' = FIRST(y$) then goto step 4
else repeat step 3;
(3) Using algorithm 3, perform X'
(4) P := last_node(X');
if P = TOP then go to step 6
else go to step 5;
(5) Using algorithm 3, perform X', goto step 4;
(6) Stop

(예 6) 예 1의 문법 G에서 스트링 X ="with a , a do s $"가 그림 7의 확장형 파싱표를 이용하여 X ="with a · a do s $"로 변경될 경우를 생각해 보자. 즉, X="with a" , w="," , w = "·", y="a do s" 인 경우 알고리듬 4를 수행하면 단계 1의 초기화 과정에서 X 의 y부분은 V W로 초기화 된다. 따라서, X ="with a · V W $" 이다 X = "with a , a do s $" 의 트리 구조가 그림 4와 같으므로 알고리듬 4의 단계 1을 수행하면 i=j=3 이 된다.

그리고 초기화 과정에서는 V에는 마지막 노드 6을 연결, W에는 마지막 노드 9를 연결, $에는 노드 15를 연결한다.

단계 2를 수행하면 TOP = 3으로부터 시작하여 단계 2를 수행한다.

단계 3에서 알고리듬 3의 파싱단계를 수행하여 ACTION(S_4, ·)=shift가 되고 ACTION(S_4, ·) =S_{11}이 되는데, 노드 3의 자식들을 살펴보아 S_{11}이 없으므로 새로운 노드 16(상태는 S_{11})을 노드 3에 추가하고 TOP이 노드 16을 가르키게 하고 j값이 증가하여 j=4가 된다. 단계 4에서 X_4'에 추가된 마지막 노

드값이므로, 노드 6을 P라고 한다. P와 TOP(노드 16)이 같지 않으므로 단계 5를 수행한다.

단계 5에서 알고리듬 3의 파싱단계를 수행한다. ACTION(S_{11}, V)=shift가 되고 ACTION(S_{11}, V) =S_{12}가 되는데 노드 16의 자식들을 살펴보아 S_{12}가 없으므로 노드 16이 노드 7을 연결하고 X_i'에 추가된 노드 6(상태는 S_6)을 TOP에 의해 참고된 노드 16으로 대체하고 노드 16에 노드 17을 추가시키고 TOP이 가리키도록 한다. j값이 증가되어 j=5가 된다. 단계 4로 되돌아가서 노드 9는 P가 되고 P와 TOP(노드 17)이 같지 않으므로 단계 5를 수행한다. 단계 5에서 알고리듬 3의 파싱단계를 수행한다. ACTION(S_{12}, W)=reduce" V' → · V " 이므로 2 레벨 만큼 노드 17로 부터 거슬러 올라가면 노드 3이 된다. ACTION(S_4, V')=S_{10}이 되는데 노드 3의 자식중에 S_{10}이 있으므로 TOP이 노드 4를 가리킨다. 단계 4로 되 돌아가서 P는 노드 9가 되고 P와 TOP(노드 4)이 같지 않으므로 단계 5를 수행한다.

단계 5에서 알고리듬 3의 파싱단계를 수행한다. ACTION(S_{10}, W)=reduce" V → aV' "이므로 2 레벨 만큼 노드 4로 부터 거슬러 올라가면 노드 2가 된다. ACTION(S_2, V)=S_3가 되는데 노드 2의 자식중에 S_3이 있으므로 TOP이 노드 5를 가리킨다. 단계 4로 되 돌아가서 P는 노드 9가 되고 P와 TOP(노드 5)이 같지 않으므로 단계 5를 수행한다.

단계 5에서 알고리듬 3의 파싱단계를 수행한다. ACTION(S_3, W)=shift"W"가 되고 ACTION(S_3, W)=S_5가 되는데, 노드 5의 자식중에 S_5가 있으므로 노드 5가 노드 10을 연결하고 X_5'에 추가된 노드 9를 TOP에 의해 참고된 노드 5로 대체하고 TOP이 노드 14를 가리키도록 한다. j값이 증가되어 j=6 이 된다. 단계 4로 되돌아가서 노드 15는 P가 되고 P와 TOP(노드 14)이 같지 않으므로 단계 5를 수행한다.

단계 5에서 알고리듬 3의 파싱 단계를 수행한다. ACTION(S_5, $)=reduce" U → with V W " 이므로 3 레벨 만큼 노드 14로부터 거슬러 올라가면 노드 1이 된다. ACTION(S_0, U)=S_1이 되는데 노드 1의 자식중에 S_1이 이미 존재하므로 TOP이 노드 15를 가리킨다. 단계 4로 되돌아가서 P는 노드 15가 되고 P와 TOP이 같으므로 단계 6으로 가서 파싱을 종료하게 된다. 따라서, 효율적인 점진적 파싱 알고리듬을 이용하여 계산된 트리구조는 그림 8과 같다

따라서, Celentano의 점진적 알고리듬에서는 w="."대신 w ="."로 대치될 때 전체 15 단계중 12 단계가 모두 새롭게 계산되고 추가됨을 알 수 있다. 그러나 효율적인 점진적 파싱 알고리듬에서는 configuration 초기화 과정 이후 단지 7 단계만이 수행됨을 알 수 있다.

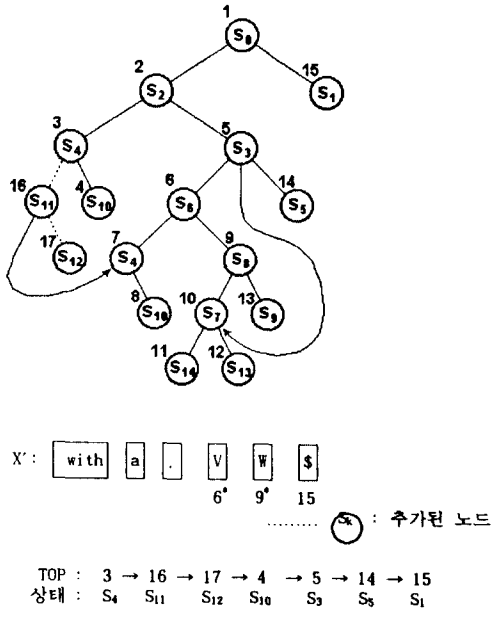


그림 8. G에 대한 효율적인 점진적 파싱을 이용한 트리구조

Fig. 8. Tree structures using efficient incremental parsing for G.

3. 실험 및 평가

SUN UNIX OS 환경에서 YACC을 이용하여 C 언어로 확장형 LR 파싱표를 자동으로 생성할 수 있는 파서생성기를 구성하고, 이 파서생성기를 이용하여 본 논문에서 제안한 효율적인 점진적 LR 파싱 알고리듬과 Celentano의 점진적 파싱 알고리듬을 구현하여 수식에 관한 문법과 파스칼 언어의 with문과 같은 여러 경우의 입력 스트링에 대하여 다음과 같이 실행하고 비교해 보았다.

실행결과에서 파싱 단계와 기억 장소는 첫 번째 입

표 1. 성능 평가

Table 1. Performance measurements.

알고리즘 / 입력 스트링	Celentano 알고리즘		본 논문에서 제시한 효율적인 알고리즘	
	파싱 단계	기억 장소	파싱 단계	기억 장소
test 1	28	528 byte	22	408 byte
test 2	43	816 byte	34	696 byte
test 3	39	720 byte	26	480 byte
test 4	64	1,152 byte	51	912 byte

력 스트링의 파싱과정과 두번째 입력 스트링(변경된 입력 스트링)의 파싱과정을 모두 합한 것이다.

※ 입력 스트링

① test 1: with a . a do s → with a . a do s

② test 2: with a . a do with a . a do s → with a . a do with a . a do s

③ test 3: a *(a + a) → a + (a + a)

④ test 4: (a + a + a) + (a *(a + a)) → (a + a + a) + (a + (a + a))

Celentano의 점진적 알고리듬보다 본 논문에서 제시한 효율적인 점진적 LR 파싱 알고리듬 이 파싱단계와 기억 장소의 요구를 상당히 축소하였음을 표 1의 성능평가로 알 수 있다. 파싱단계의 경우 test 1의 입력 스트링의 예에서 Celentano의 알고리듬에서는 28단계가 소요되고, 본 논문에서 제시한 알고리듬은 22단계만 소요된다. 기억장소의 경우에서도 Celen-tano 알고리듬은 528 byte가 필요한 반면 본 논문에서 제시한 알고리듬은 408 byte만이 요구된다.

그리고, test 1에서 두 번째 입력 스트링 "with a . a do s"의 파싱과정을 살펴보면 예 6 에서와 같이 Celentano 알고리듬에서는 12 단계가 소요되고 본 논문에서 제시한 알고리듬 에서는 7 단계만 소요된다. Agrawal과 Detro는 Celentano의 알고리듬에 바탕을 두고 ε -생성규칙을 허용하는 문법에 대한 알고리듬을 제시하였으나 [ ], Agrawal과 Detro의 알고리듬에 대한 실험결과는 Celentano 의 점진적 파싱 알고리듬에서와 같다.

그러므로, 본 논문에서 제시한 효율적인 점진적 LR 파싱 알고리듬이 실험에서 제시한 여러 입력 스트링의 예를 통하여 파싱단계와 기억장소를 상당히 감소시킬 수 있음을 보여 주었다.

Ⅴ. 결 론

본 논문에서는 Celentano가 LR 파싱에 기초한 점진적 파싱 알고리듬을 제안하며 지적했듯이 점진적 파싱을 기억장소와 수행시간이 너무 많이 요구되는 파스순서에 의한 자료구조 보다 트리구조로 표현하는 자료구조를 이용하였다.

기존의 파서를 확장하여 입력 스트링으로 단말기호와 비단말기호까지 처리하는 확장형 LR 파싱표를 자동으로 생성하도록 하였으며, 이 자동으로 생성한 LR 파서 생성기를 이용하여 효율적인 점진적 LR 파싱 알고리듬을 구현하였다.

LR 부류의 문법중에서 프로그래밍 언어에서 많이 사용되는 파스칼 언어의 with 문장이나 수식에 관한 예를 통하여 확장형 LR 파싱표를 이용하여 Celentano의 알고리듬을 이용한 점진적 파싱 과정과 본 논문에서 제안한 효율적인 점진적 LR 파싱 알고리듬을 실행하고 비교해 보았다.

이 비교과정에서 효율적인 점진적인 LR 파싱 알고리듬이 Celentano의 점진적 파싱 알고리듬보다 기억장소를 많이 감소시키고 과거의 파싱 과정에서 이미 생성된 적이 있었던 파싱 단계를 상당히 축소 시킬 수 있음을 실험결과로 알 수 있다. 이를 증명하기 위해 실제 SUN UNIX OS환경에서 C언어로 프로그래밍하여 구현하고 분석하였다.

본 연구는 점진적 파싱방법을 바탕으로하고 고급언어 프로그램에 관한 문법적, 의미적 정보를 갖고 있어서 프로그래머로 하여금 언어의 구문구조에 따라 프로그래밍 하도록 유도하고, 대화식으로 처리하는 특성을 갖는 언어기반 환경을 구현하는데 많은 기여를 할 것이다. 또한 PL/0 언어나 C 언어와 같이 문법이 크고 복잡하며 파스트리의 길이가 긴 경우에 본 논문에서 제안한 효율적인 점진적 LR 파싱 알고리듬이 더 큰 효과를 가질 것이며, 소프트웨어의 대화식 프로그래밍 환경 구축에 대한 연구가 보다 활발히 진행되고 활용될 것으로 기대된다.

參 考 文 獻

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

[2] C.N.Fischer and R.J.LeBlanc, *Crafting a Compiler with C*, the Benjamin/ Cummings Publishing Company, Inc., 1991.

[3] J.R. Levine, T. Mason, and D. Brown, *lex & yacc*, O'Reilly & Associates, Inc. 1990.

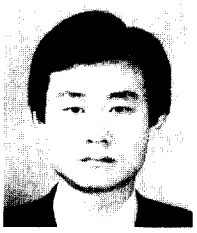
[4] R. Medina-Mora and H. Feiler, "An Incremental Programming Environment", *IEEE Trans. on SE*, Vol.7, pp.472-482, 1981.

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

[6] C. Ghezzi and D. Mandrioli, "Incremental Parsing", ACM Transactions on

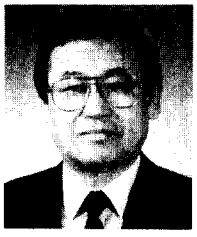
Programming Languages and Systems, Vol.1, No.1, pp.58-70, 1979.
[7] C. Ghezzi and D. Mandrioli, "Augmenting Parsers to Support Incrementality", *Journal of ACM*, Vol.27, No.3, pp.564-579, 1980.
[8] R. Agrawal and K.D. Detro, "An Efficient Incremental LR Parser for Grammars with Epsilon Productions", *Acta Informatica*, Vol.19 , pp.369-373, 1983.
[9] D. Yeh, "On Incremental Shift-Reduce Parsing", *BIT*, Vol.23, pp.36-48, 1983.
[10] D. Yeh and U. Kastens, "Automatic Construction of Incremental LR(1)-Parsers", *ACM SIGPLAN Notices*, Vol. 23, No.3, pp.33-42, 1988.
[11] J. Rekers and W. Koorn, "Substring Parsing for Arbitrary Context-Free Grammars", *ACM SIGPLAN Notices*, Vol.26, No.5, pp.59-66, 1991.
[12] 송후봉, 유재우, "구문 지향 편집기에서의 점진적 파싱 알고리즘", 한국 정보 과학회 논문지, Vol.18, No.4, pp.388-398, 1991.
[13] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Sytax-Directed Programming Environment", *Comm. ACM*, Vol.24, No.9, pp.563-573, 1981.
[14] T. Teitelbaum and T. Reps, *The Synthesizer Generator*, Springer Verlag, 1989.

著者紹介



安 熙 學(正會員)

1956年 9月 21日生. 1981年 2月 숭실대학교 전자계산학과 졸업(공학사). 1983年 2月 숭실대학교 대학원 전자계산학과 졸업(공학 석사). 1991年 2月 숭실대학교 대학원 전자계산학과 수료. 1984年 3月~ 현재 관동대학교 전자계산공학과 부교수. 주관심 분야는 컴파일러, 프로그래밍 언어론, 함수 언어, 오토마타 이론 등임.



宋 後 鳳(正會員)

1933年 3月 4日生. 1956年 6月 육군사관학교 졸업(이학사). 1986年 2月 중앙대학교 대학원 전자계산학과 졸업(이학석사). 1989年 8月 조선대학교 대학원 전산기공학과 졸업(공학박사). 1971年 ~ 현재 숭실대학교 전자계산학과 교수. 1993年 8月~ 현재 숭실대학교 부총장. 주관심 분야는 컴파일러, 프로그래밍 언어, 객체지향 운영체제, 컴퓨터 보조학습 등임.

劉 載 祐(正會員)

1954年 8月 18日生. 1976年 2月 숭실대학교 전자계산학과 졸업(공학사). 1978年 2月 한국과학기술원 전산학과 졸업(이학 석사). 1985年 2月 한국과학기술원 전산학과 졸업(이학 박사). 1986年~ 1987年 Cornell Univ.의 Visiting Scientist. 1983年 9月~ 현재 숭실대학교 전자계산학과 부교수. 한국과학기술원 인공지능 연구센터 연구교수. 주관심 분야는 컴파일러, 프로그래밍 환경, 인간과 컴퓨터 상호작용 등임.