점진적 LL(1) 구문분석에서의 재사용 시점의 계산

(Computation of Reusable Points in Incremental LL(1) Parsing)

이 경 옥 †

(Gyung-Ok Lee)

요 약 점진적 구문분석은 기존 입력 문자열에 대한 구문분석 정보를 새로운 문자열의 구문분석 시에 재사용하고자 하는 취지로 연구 개발되었다. 기존 점진적 LL(1) 구문분석에서는 미리 재사용 시점을 계산하여 이를 구문분석 시에 이용하였다. 본 논문에서는 기존의 재사용 시점 계산을 인수분해하여 불필요한 중복 계산 없이 효율적으로 수행하는 방법을 제안한다. 또한 기존의 재사용 시점 파악을 위해서 사용되었던 공통심볼 저장방법과 거리저장방법을 결합하여 공통심볼까지의 거리저장방법을 제안한다. 이에 기반한 효율적인 점진적 LL(1) 구문분석기를 생성한다.

키워드 : 점진적 구문분석, LL(1) 구문분석기, LL(1) 문법

Abstract Incremental parsing has been developed to reuse the parse result of the original string during the parsing of a new string. The previous incremental LL(1) parsing methods precomputed the reusable point information before parsing and used it during parsing. This paper proposes an efficient reusable point computation by factoring the common part of the computation. The common symbol storing method and the distance storing method were previously suggested to find the reusable point, and by combining the methods, this paper gives the storing method of the distance to common symbols. Based on it, an efficient incremental LL(1) parser is constructed.

Key words : incremental parsing, LL(1) parser, LL(1) grammar

1. 서론

점진적 구문분석은 기존 구문분석 결과를 새로운 입력 문자열의 구문분석 시에 재사용하려는 취지로 제안되었다. 본 논문은 점진적 LL 구문분석 문제를 다룬다. LL 구문분석은 LR 구문분석에 비해서 적용할 수 있는 문법 클래스가 작으나 LR 구문분석에 비해서 구현하기가 쉽고 개념적으로 이해하기 쉽다는 장점을 가진다. 점진적 LL 구문분석기에 관한 연구로는 Magpie[1]과 Galaxy[2]에서의 연구를 시작으로 Yang[3], Li[4,5], Lee[6]에 의한 방법들이 제안되었다. Yang[3]과 Li[5]는 구문분석 이전에 미리 재사용가능시점을 계산하여 이를 점진적 구문분석 시에 이용하는 방법을 제안하였다. 또한 Li[4]는 미리보기심볼(lookahead symbol)로 넌터미널 심볼을 포함시키는 것을 제안하였고, 최근에 Lee[6]는 이를 개선한 알고리즘을 제안하였다.

본 논문에서는 Yang[3]과 Li[5]의 방법에서 사용되었던 재사용가능시점 계산에 관한 효율적인 방법을 제시한다. 첫째, 기존의 문법 규칙과 문법 유도 관계에 근거한 정형식을 문법 심볼에 근거한 관계로 단순화시켰다. 둘째, 기존 작업들은 항상 터미널 심볼의 관점에서 재사용 시점을 조사하였기에 불필요한 계산을 포함시켰다. 이에 반해서 본 논문에서는 넌터미널 심볼간의 관계만으로 재사용 시점을 조사할 수가 있는 경우에는 이들만의 관계로 재사용 시점을 계산하기에 기존의 중복 계산 과정을 피한다. 셋째, Yang의 방법에서는 공통심볼을 저장함으로써 구문분석 시에 심볼 비교 과정을 포함시켰고, Li의 방법에서는 심볼간의 거리를 저장하여 구문

· 이 논문은 한신대학교 학술연구비 지원에 의해서 연구되었음

† 종신회원 : 한신대학교 정보통신학과 교수
golee@hs.ac.kr

논문접수 : 2010년 6월 23일
심사완료 : 2010년 9월 6일

분석 시에 거리 비교 과정을 포함시켰다. 또한 Li의 방법에서는 같은 거리더라도 동일 심볼이 아닌 경우가 있기에 이에 대한 심볼 비교 과정이 포함되었다. 본 논문에서는 Yang의 방법과 Li의 방법을 결합하여 공통 심볼까지의 거리를 저장하는 방법을 사용하여 기존의 비교 과정을 없앤다. 이로써 효율적인 점진적 구문분석기의 생성이 가능하다.

2장에서는 기본 정의와 표기법에 관해 언급하고, 3장에서는 효율적인 재사용 시점 계산을 위한 기본 정형식들과 그 성질을 보인다. 4장에서는 이를 이용한 점진적 LL(1) 구문분석기를 생성하고, 5장에서는 기존 방법들과의 비교한다. 끝으로 6장에서 결론을 맺는다.

2. 기본 정의와 표기법

본 논문의 기본적인 정의와 표기법은 관련 책[7,8]에 기반하며, 그 외 정의는 관련 논문[3,5]을 따른다.

문법 G = (N, Σ, P, S)로 정의되며, 여기서 N은 비단말 심볼들의 집합, Σ은 단말 심볼들의 집합, P는 문법 규칙들의 집합, S는 시작 심볼을 나타낸다. V는 N ∪ Σ을 표기한다. 본 논문에서는 G에 규칙 S'→ S$을 추가한 문법을 대상으로 하며, G가 LL(1) 문법임을 가정한다.

본 논문에서의 FIRST, FOLLOW 함수는 다음과 같다: α ∈ V*라고 하자. FIRST(α) = {X | α ⇒* Xβ, X ∈ Σ} ∪ {ε|α ⇒* ε}, FOLLOW(A) = {X | S ⇒* βAXγ, X ∈ Σ}으로 정의한다.

정의 2.1 LL-구문분석테이블 PT은 N x Σ에서 {A → α| A → α ∈ P} ∪ {오류}로의 함수이며, 다음과 같이 정의된다:

(i) A → α ∈ P, X ∈ FIRST(αFOLLOW(A))이면 PT(A, X) = A → α 생성

(ii) (i) 이외의 경우에는 PT(A, X) = 오류 □

정의 2.2 실트리(threaded tree)는 각 노드 n에 대해서 다음과 같이 링크를 갖는 트리이다:

(i) n의 우측 형제 m이 존재하는 경우에 n은 m으로의 링크를 갖는다.

(ii) (i)의 조건이 성립하지 않는 경우에 n의 가장 가까운 조상의 우측 형제 m이 존재하면 n은 m으로의 링크를 갖는다.

(iii) (i),(ii)의 조건이 모두 성립하지 않는 경우는 n은 링크 nil을 가진다. □

실트리에서 Γ(n)은 n의 링크를 나타낸다. $\Gamma^0(n) = n$, $\Gamma^i(n) = \Gamma(\Gamma^{i-1}(n))$, $i \geq 1$; $\Gamma^+(n) = \Gamma^1(n)\Gamma^2(n) \cdots \Gamma^k(n)$, $\Gamma^j(n) \neq nil$, $j \leq k$, $\Gamma^{k+1}(n) = nil$이다. 노드의 링크를 따라가면 LL 구문분석기의 스택 내용을 알 수 있다.

xyz를 기존 스트링, xy'z를 새 스트링이라하자. $S \Rightarrow_{lm}^* xA_1 \cdots A_p$가 G상에 존재한다고 하자. LL 구문분석기에서 x를 구문분석한 후에 $A_1 \cdots A_p$은 구문분석기의 스택 내용이 된다. x를 구문분석하는 동안에 생성되는 불완전한 구문분석트리를 *X-부분트리*라고 한다(x=ε인 경우에 X-부분트리는 기존 트리의 루트이다). 이 트리는 xyz와 xy'z에 의해서 공유되기에 재 구문분석함 없이 해당 트리를 재사용할 수 있다. a를 x의 마지막 터미널 심볼이라하고, n_a을 a에 대응되는 노드라고 하자. $\Gamma(n_a)$는 $n_{A_1} n_{A_2} \cdots n_{A_p}$이다.(여기서 각 i=1,2,…,p에 대해서 n_{A_i}는 A_i에 대응되는 노드이다.) 따라서 T를 $\Gamma^+(n_a)$상에 분리시켜서 T로부터 X-부분 트리를 얻을 수 있다. 한편 z에 대한 트리를 생각해보자. y의 마지막 심볼을 b라고 하고(y가 ε인 경우는 b는 a이다) n_b을 b에 대응하는 노드라고 하자. 또한 $\Gamma^+(n_b) = n_{B_1} n_{B_2} \cdots n_{B_m}$이라고 하자(여기서 각 i = 1,2,…m에 대해서 n_{Bi}는 B_i에 대응하는 노드이다). 이때 $B_1 \cdots B_m \Rightarrow_{lm}^* z$가 존재하며, $n_{B_1} \cdots n_{Bm}$은 *Z-부분트리들*이다.

3. 기본 정형식과 그 성질

본 절에서는 재사용 시점 분석을 위한 L 관계[6], d 함수, d^t 함수를 정의한다.

기본적으로 점진적 구문분석의 문제는 n_A가 기존 노드이고 n_B가 새로운 노드일 때 n_A를 루트로 하는 부분트리가 재사용 가능한가를 판단하는 것이다. n_A와 n_B가 동일하다면 명백하게 재사용 가능하다. 한편 그렇지 않는 경우에는 재사용 가능성을 좀 더 분석해야 한다. n_A와 n_B의 심볼을 각각 A, B라고 하자. B가 A를 최좌측 심볼로 생성해낸다면 A는 재사용 가능하다. 또한 이 경우가 아니더라도 B가 A로부터 생성되는 C를 최좌측 심볼로 생성하는 경우에 n_A의 후손인 n_C는 재사용 가능하다.(여기서 n_C는 C에 대응하는 노드이다.)

다음의 관계는 문법 심볼간의 좌측 의존 관계를 나타낸다.

정의 3.1 (L 관계) L은 N에서 (N ∪ Σ)로의 관계로 다음과 같이 정의된다.

A L X이기 위한 필요충분조건은 A → Xβ ∈ P이다.□

A L^1 X은 A L X를 표기하며, A L^n X, n > 1는 A L^{n-1} B, B L X를 나타낸다. L^*, L^+은 각각 L 관계의 반사적 전이(reflexively transitive) 클로져(closure), 전이 클로져이다. L 그래프는 L 관계를 그래프로 표기한 것이다.

다음은 L 관계, 문법유도관계, FIRST 집합간의 성질을 보인다.

성질 3.1 (a) A L^i X이면 A $\Rightarrow_{lm}^i$ Xβ가 G상에 존재한다. 역으로 A $\Rightarrow_{lm}^i$ Xβ가 G상에 존재하면, A L^i X이다.

(b) A L^+ a이면, a ∈ FIRST(A)이다. 역으로 a ∈ FIRST(A)이면 A L^+ a이다. □

성질 3.2 G를 LL(1)이라고 하자. 이때 다음이 성립한다.

(a) A L^+ B이면 B L^+ A는 성립하지 않는다.

(b) A L^+ B이면 L 그래프 상에 A에서 B로의 경로는 유일하다.

(증명) (a) A L^+ B, B L^+ A이면 A $\Rightarrow_{lm}^*$ Aβ가 G상에 존재하여 G가 LL(1)이 아니라는 모순이 발생한다.

(b) A에서 B로의 두 개의 경로가 존재한다고 하자. 가령, A_0 L A_1 L A_2 ⋯ A_{n-1} L A_n(A_0 = A, A_n = B)와 B_0 L B_1 ⋯ B_{m-1} L B_m(B_0 = A, B_m = B)이라고 하자. 이때 조건 A_i = B_i, i=0,1, ⋯, t, A_{t+1} ≠ B_{t+1}을 만족하는 가장 작은 t를 선택할 수가 있다. A_t = B_t이기에 A_t → A_{t+1}α, A_t → B_{t+1}δ ∈ P가 존재하며, 이는 G가 LL(1)이라는 사실에 모순이 된다. 따라서 A에서 B로의 경로는 유일하다. □

다음에서 정의되는 d, d^t 함수는 기존 연구[6]의 d^N, d^T와 비슷하나 본 논문에서는 기존의 참, 거짓 값이 아닌 L 관계의 심볼들 사이의 거리를 정의한다.

다음 d 함수는 넌터미널 간의 L 관계로부터 정의된다.

정의 3.2 (d 함수) d 함수는 N x N에서 ℵ ∪ {∞}로의 함수이다.(ℵ은 자연수 집합을 표기한다): A, B ∈ N라고 하자.

d(A, B) = i이기 위한 필요충분조건은 A L^i B인 i(>0)가 존재하는 것이다.

d(A, B) = ∞이기 위한 필요충분조건은 A L^+ B가 성립하지 않는 것이다. □

성질 3.2(b)에서 임의의 A, B에 대해서 A L^+ B일 때에 A에서 B로의 경로가 유일하기에 d 함수는 잘 정의가 되었음을 알 수 있다. (2장에서 본 논문의 G는 LL(1) 문법임을 가정하였다.)

다음은 d 함수의 성질을 보인다.

성질 3.3

(a) d(A, B) = i인 i(≠ ∞)가 존재하면 d(B, A) = ∞이다.

(b) d(A, A) = ∞

(증명)

(a) d(A, B) = i인 i(≠ ∞)가 존재하기에 A L^+ B가 성립한다. G가 LL(1)이기에 성질 3.2 (a)에 의해서 B L^+ A는 성립하지 않고, d(B, A) = ∞이다.

(b) d(A, A) ≠ ∞이면 d(A, A) = i인 i(≠ ∞)가 존재한다. (a)에 의해서 d(A, A) = ∞이 성립하며, 이는 모순이다. □

성질 3.4

(a) d(A, B) = i인 i(≠∞)가 존재하고, a ∈ FIRST(B)이면, A $\Rightarrow_{lm}^i$ Bβ $\Rightarrow_{lm}^*$ aδβ가 G상에 존재한다.

(b) A $\Rightarrow_{lm}^i$ Bβ $\Rightarrow_{lm}^*$ aδβ가 G상에 존재하면, d(A, B) = i, a ∈ FIRST(B)이다. □

다음 d^t 함수는 터미널 심볼을 고려한 넌터미널 간의 L 관계로부터 정의된다.

정의 3.3 (d^t 함수) d^t 함수는 N x N x (Σ ∪ {ε}) 에서 (ℵ ∪ {∞}) x (ℵ ∪ {∞})로의 함수이다: A, B ∈ N, a ∈ Σ ∪ {ε}라고 하자.

d^t(A, B, a) = (n_1,n_2)이기 위한 필요충분조건은 A L^{n_1} X, B L^{n_2} X, X L^* a을 만족하는 X가 존재한다. 그 외의 경우는, d^t(A, B, a) = (∞,∞)이다. □

다음은 d^t 함수의 성질을 보인다.

성질 3.5

(a) d^t(A, B, a) = (n_1,n_2), n_1(≠∞), n_2(≠∞)가 존재한다면 A $\Rightarrow_{lm}^{n_1}$ Xβ $\Rightarrow_{lm}^*$ aδβ, B $\Rightarrow_{lm}^{n_2}$ Xγ $\Rightarrow_{lm}^*$ aδγ가 G상에 존재한다.

(b) A $\Rightarrow_{lm}^{n_1}$ Xβ $\Rightarrow_{lm}^*$ aδβ, B $\Rightarrow_{lm}^{n_2}$ Xγ $\Rightarrow_{lm}^*$ aδγ가 G상에 존재하면, d^t(A, B, a) = (n_1,n_2), n_1(≠∞), n_2(≠∞)가 존재한다.

(증명)

(a) d^t 함수의 정의에 의해서 A L^{n_1} X, B L^{n_2} X, X L^* a이다. 성질 3.1(a)에 의해서 A $\Rightarrow_{lm}^{n_1}$ Xβ $\Rightarrow_{lm}^*$ aδβ, B $\Rightarrow_{lm}^{n_2}$ Xγ $\Rightarrow_{lm}^*$ aδγ가 G상에 존재한다.

(b) (a)와 유사하게 증명가능하다. □

성질 3.6 A, B ∈ N이고 d(A,B) = m(>0)이면 각 a ∈ FIRST(B)에 대해서 d^t(A, B ,a) = (m,0)이다.

(증명) d(A,B) = m이기에 성질 3.4(a)로부터 A $\Rightarrow_{lm}^m$ Bβ가 존재하고, a∈ FIRST(B)이기에 B $\Rightarrow_{lm}^*$ aδ가 G상에 존재한다. 따라서 A $\Rightarrow_{lm}^m$ Bβ $\Rightarrow_{lm}^*$ aδβ가 G상에 존재하기에 성질 3.5 (b)로부터 d^t(A, B ,a) = (m,0)이다. □

성질 3.6은 d 함수 값으로 터미널 심볼을 포함시키지 않고도 거리를 파악할 수 있음을 보인다. 이에 따라서 재사용 시점 파악을 일차적으로 d 함수를 이용하며, 터미널 심볼이 반드시 필요한 경우에는 d^t 함수를 이용함이 적합하다. 본 절에서 정의된 d^t 함수는 Yang의 break-point 테이블[3]과 유사하나, 그의 작업에서는 거리를 대신하여 공통 심볼 자체를 저장시켰다. 한편 Li의 방법[5]에서는 거리를 저장시켰으나 넌터미널 심볼간의 관계는 전혀 고려하지 않고 넌터미널 심볼과 터미널 심볼의 관계만을 고려하여 거리를 계산하였다.

4. 점진적 LL 구문분석

본 절에서는 d, d^t 함수를 이용한 점진적 LL 구문분석 알고리즘이 제시된다. 사용되는 자료구조, 함수, 알고리즘은 기본적으로 Li의 작업[5]에 기반한다. 독자의 편의를 위해서 다음에서 관련 정의들을 보인다.

실트리의 각 노드 n은 다음의 속성을 가진다:

v(n): n의 문법심볼

father(n): n의 부모로의 포인터

firstchild(n): n의 첫번째 자식으로의 포인터

Γ(n): n의 실링크

다음은 사용될 변수의 정의이다:

new: 현재 구문분석되는 노드로의 포인터

old: 재사용을 위해 고려되는 Z-부분트리들의 노드로의 포인터

token: y′z로부터의 미리보기 토큰

다음 함수가 사용된다:

empty_node(): 빈 노드를 생성하고 그 노드로의 포인터를 리턴한다.

next_input_token(): y′로부터의 다음 토큰을 받는다. 이때 토큰이 존재하지 않으면 nil을 리턴한다.

leftmost_leaf(n): Γ(n)의 최좌측 토큰을 생성한다.

breakup(n): 원래 트리 T로부터 n을 분리시킨다. 분리된 노드는 재사용을 위해서 고려된다. n의 자식을 좌에서 우로 n_1, …, n_t라고 하면 다음의 과정이 수행된다:

n′ = empty_node()

v(n′) = v(n)

Γ(n′) = Γ(n)

firstchild(n′) = firstchild(n)

firstchild(n) = nil

각 i($1 \le i \le t$)에 대해서 father(n_i) = n′이다.

build(new, p): 문법 규칙 $p = C \rightarrow C_1 \cdots C_t$를 적용하여 new로부터 자식을 생성하며 다음 과정이 수행된다:

각 i($1 \le i \le t$)에 대해서 n_{C_i} = empty_node()이다.

각 i($1 \le i \le t$)에 대해서 father(n_{C_i}) = new이다.

firstchild(new) = n_{C_1}이다.

각 i($1 \le i \le t-1$)에 대해서 $\Gamma(n_{C_i}) = n_{C_{i+1}}$이다; i = t일 때는 $\Gamma(n_{C_i})$ = Γ(new)이다.

각 i($1 \le i \le t$)에 대해서 $v(n_{C_i}) = C_i$이다.

copy(old,new): old의 모든 부분트리를 new로 복사한다. 복사 후에 실트리 링크들은 new의 실트리 링크인 Γ(new)와 동일하게 된다.

알고리즘 1 (점진적 LL 구문분석)

입력: 기존 문자열 xyz에 대한 구문분석트리, 새로운 문자열 xy′z, 구문분석 테이블 PT

출력: xy′z에 대한 구문분석트리

방법:

1. /*y′를 구문분석한다*/
   new = $\Gamma(n_a)$ (여기서 n_a은 a(a는 x의 마지막 심볼이다)에 대응하는 노드이다)
   token = new_input_token()
   while(token ≠ nil) do
   (a) if firstchild(new) ≠ nil then breakup(new)
   (b) if v(new) = token then
   new = Γ(new)
   token = next_input_token()
   else if v(new)가 넌터미널이다 then
   p = PT(v(new),token)
   if p = error then 오류를 보고하고 중단한다.
   build(new,p)
   new = firstchild(new)
   else 오류를 보고하고 중단한다.
   od
2. /* z를 구문분석한다*/
   old = n_{B_1}(여기서 n_{B_1}은 Z-부분트리들의 첫번째 노드이다)
   token = leftmost_leaf(old)
   p = PT(A,token)
   while(new ≠ old) do
   A, B를 각각 new, old의 심볼이라고 하자.
   (a) if A ≠ B, A, B ∈ Σ then 오류를 보고하고 끝낸다.
   (b) if A ∈ N then
   if p = error then 오류를 보고하고 끝낸다.
   (c) if firstchild(new) ≠ nil then breakup(new)
   (d) /* d, d^t 함수를 이용하여 재사용시점을 찾는다 */
   i.if d(A, B) = m인 m(≠∞)이 존재한다 then
   - for(i = 0; i < m; i++)
   *breakup(new)
   *build(new,p) (여기서 p=PT(v(new), token)이다)
   *new = firstchild(new)
   ii.else if d(B,A) = m인 m(≠∞)이 존재하거나 A ∈ Σ, B ∈ N이다 then
   - for(i = 0; i < m; i = i + 1) old = firstchild(old)
   iii.else if d^t(A, B, token) = (m_1,m_2)인 m_1(≠∞),m_2(≠∞)가 존재한다 then
   - for(i = 0; i < m_1; i = i + 1)
   *breakup(new)
   *build(new, p)(여기서 p = PT(v(new), token)이다.)
   *new = firstchild(new)
   - for(i = 0; i < m_2; i = i + 1) old = firstchild(old)
   (e)/*재사용한다*/
   copy(old,new)
   new = Γ(new)
   old = Γ(old)
   token = leftmost_leaf(old)

p = PT(v(new),token)

od □

알고리즘1의 단계1에서 y'는 실트리 노드를 만들면서 구문 분석되고, 단계2에서 z는 Z-부분트리들을 사용하여 구문 분석된다. 단계(i)에서는 d(A,B) = m이기에 A L^m B가 성립하고 A L^m B L* token에 대응되는 L 그래프상의 경로는 유일하다. 따라서 n_A의 m번째 자식은 n_B(v(n_B)=B이다)이고 n_B은 재사용 가능하다. 유사하게 단계(ii)에서 d(B,A) = m이기에 n_B의 m번째 후손은 재사용 가능하다. 한편 d(A,B) = ∞, d(B,A) = ∞인 경우는 토큰을 생성하는 A와 B의 공통 노드를 찾는 단계가 필요하다. 단계(iii)에서 d^t(A,B,token) = (m_1,m_2), $m_1 \neq \infty$, $m_2 \neq \infty$이면 n_A의 m_1번째 후손은 n_B의 m_2번째 후손과 일치한다. 이것은 단계(i),(ii)와 유사하게 재사용 가능함을 의미한다. 한편 $m_1 = \infty$이라면 PT(A, token) ≠ error에 모순이 되고, $m_2 = \infty$이라면 token = leftmost_leaf(old)에 모순이 되기에 본 알고리즘은 모든 경우를 다 고려하고 있음을 알 수 있다.

5. 기존 연구와의 비교

본 논문의 재사용시점 계산 작업을 기존 연구의 작업[3,5]과 비교해보면 다음과 같다. (넌터미널 심볼을 미리보기심볼로 포함시킨 기존 연구의 작업[4,6]와의 비교는 기반 축이 다르므로 본 연구와의 비교를 하지 않는다.)

첫째, d 테이블이 필요로 하는 공간은 O(|N| x |N|)이고 d^t 테이블이 필요로 하는 공간은 O((|N| x |N| x |Σ|) − (M_d x |Σ|)) (여기서 M_d는 d(A,B)≠∞,A≠B인 조건을 만족하는 $\Sigma_{A\in N}\Sigma_{B\in N}$O(1)이다.)이다. 반면에 break-point 테이블[3]은 O(|N| x |N| x |Σ|)의 공간을 필요로 한다. 따라서 Yang의 방법[3]보다 재사용시점 저장 공간이 O(M_d x |Σ|)만큼 감소했다. 한편 Li의 방법[5]에서는 LL 테이블에 재사용시점 정보를 저장하였으며 이를 위해서 O(|V| x |Σ|)의 공간이 필요하다.

둘째, L 관계는 재사용 계산 과정을 분명하게 하고 단순화시킨다. 기존 작업[3,5]에서는 문법 유도 과정과 문법 규칙으로 이를 처리한 것에 반해서 본 논문에서는 문법 심볼간의 관계로 이를 처리하고 있다. 이로써 계산 시간이 단축된다. 또한 재사용시점 계산 과정을 고려 시에 Yang과 Li의 작업에서는 항상 터미널 심볼을 포함시켜서 재사용시점 계산을 했다. 한편 본 논문에서는 재사용시점 계산을 넌터미널 심볼만으로 가능한 경우를 처리하는 d 함수와 터미널 심볼을 포함시켜서 처리하는 d^t 함수로 나누었기에 계산 시간이 단축된다.

다음으로는 본 논문에서 제안한 점진적 구문분석기를 기존 연구[3,5]의 작업과 비교해본다.

첫째, Yang의 작업[3]에서는 같은 터미널 심볼을 생성하는 심볼을 break-point 테이블에 저장하고 구문분석 시에 매번 비교를 하여서 해당 심볼인지를 가름한다. 이것에 대한 시간은 O(|y'z|)가 소모된다. 한편 제안된 방법에는 d^t 테이블이 직접 재사용시점을 알려주기에 이런 비교작업이 불필요하다. 따라서 O(|y'z|)시간이 절약된다. 한편 Li의 작업[5]에서는 넌터미널 심볼로부터 터미널 심볼까지의 거리가 저장되어서 그 정보가 LL 구문분석 테이블에 저장되었다. 점진적 구문분석 동안에 새로운 심볼과 기존 심볼의 거리가 비교되어서 같은 거리인가를 찾게 된다. 그러나 같은 거리더라도 반드시 재사용 가능한 것이 아니다. 따라서 거리가 일치하더라도 심볼의 일치 여부 판단의 작업이 요구되며, 이것에 필요한 시간은 O(|z|)이다. 한편 제안된 방법에서는 이런 작업이 불필요하기에 O(|z|)시간이 감소한다.

둘째, ε-규칙에 대해서 기존 작업[3]은 매 구문분석 시점마다 ε-규칙에 대응되는 트리를 제거하는 작업을 수행했다. 본 논문에서는 이런 작업이 불필요하다.

6. 결론

본 논문에서는 L 관계, d 함수, d^t 함수를 사용하여 효율적인 재사용시점 계산 방법을 제안하고 이를 기반으로 효율적인 점진적 LL(1) 구문분석기를 제안하였다. 본 논문은 직관적이고 이론적인 접근 방식을 취했으며, 이에 대한 실험적 분석은 차후의 과제로 남겨둔다.

참 고 문 헌

[1] M.D. Schwartz, N.M. Delisle, and V.S. Begwani, "Incremental compilation in Magpie," *Proc. SIGPLAN 84 Symp. On Compiler Construction*, Montreal, Canada, 1984, ACM SIGPLAN Notices, vol.19, pp.122-131, 1984.

[2] J.F. Beetem and A.F. Beetem, "Incremental scanning and parsing with Galaxy," *IEEE Trans. Software Engineering*, vol.17, pp.641-651, 1991.

[3] W. Yang, "An incremental LL(1) parsing algorithm," *Information Processing Letters*, vol.48, pp.67-72, 1993.

[4] W. Li, "A simple and efficient incremental LL(1) parsing," *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM '95)*. Lecture Notes in Computer Science, vol.1012, Springer, New York, pp.399-404, 1995.

[5] W. Li, "Building efficient incremental LL parsers by augmenting LL tables and threading parse trees," *Comput. Lang.*, vol.22, no.4, pp.225-235, 1996.

[6] G.O. Lee, "An improved incremental LL(1) parsing method," *Journal of KIISE : Software and Applications*, vol.37, no.6, pp.486-490, 2010.(in Korean)

[7] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and Compiling*, vols. 1 & 2, Prentice-

Hall, Englewood Cliffs, NJ, 1972, 1973.
[8] S. Sippu and E. Soisalon-Soininen. *Parsing Theory*, vols. I & II, Springer, Berlin, 1990.

이 경 옥
정보과학회논문지 : 소프트웨어 및 응용
제 37 권 제 6 호 참조