개선된 점진적 LL(1) 파싱 방법
(An Improved Incremental LL(1) Parsing Method)

이 경 옥 †
(Gyung-Ok Lee)

요 약 점진적 파싱은 기존의 입력 문자열에 대한 파싱 정보를 새로운 문자열의 파싱시에 이용하고자 하는 취지로 연구 개발되었다. 본 논문은 기존에 제안된 비단말 심볼을 미리보기(lookahead) 심볼로 포함시킨 점진적 LL(1) 파서를 개선시킨 방법을 제안한다. 기존 연구에서는 오류가 발생한 상황에서 불필요한 작업을 반복적으로 수행하기에 시간적으로 비효율적이다. 본 논문에서는 이에 대한 해결 방안을 제공한다.
키워드 : 점진적 파싱, LL(1) 파서, LL(1) 문법

Abstract Incremental parsing has been researched in the intention that the parse result of the original string is reused in the parsing of a new string. This paper proposes an improved method of the previous incremental LL(1) parser with nonterminal lookahead symbols. The previous work is time-inefficient because it repeatedly performs unnecessary steps when an error occurs. This paper gives a solution for the problem.
Key words : incremental parsing, LL(1) parser, LL(1) grammar

1. 서 론

점진적 파싱은 기존의 파스 결과를 새로운 입력 문자열의 파싱 시에 재사용하려는 취지로 제안된 후에 LR 파싱과 LL 파싱에 기반을 두고 연구개발되었다. LR 파싱은 결정적으로 파싱 가능한 문법을 파싱할 수 있는 대표적인 상향식 방법이며, LL 파싱은 LR 파싱에 대응하는 대표적인 하향식 방법이다. 본 논문은 점진적 LL 파싱 문제를 다루며, LL 파싱은 LR 파싱에 비해서 적용할 수 있는 문법 클래스가 작으나 LR 파싱에 비해서 구현하기가 쉽고 개념적으로 이해하기 쉽다는 장점을 가진다.

점진적 LL 파서에 관한 연구로는 Magpie[1]과 Galaxy [2]에서의 연구를 시작으로 Yang[3]과 Li[4,5]에 의해 개선된 방법들이 제안되었다. Yang과 Li의 연구[3,5]에서는 단말 심볼을 미리보기 심볼로 사용하는 점진적 파싱 방법을 제안하였고, Li의 또 다른 연구[4]에서는 단말 심볼 뿐만 아니라, 비단말 심볼을 미리보기 심볼로 포함한 점진적 파싱 방법을 제안하였다.

본 논문에서는 비단말 심볼을 미리보기 심볼로 포함한 기존 연구[4]에서 제안된 방법의 불필요한 시간 소모의 문제점을 제기하고, 이에 대한 해결 방안을 제시하고자 한다. 기존 연구[4]에서 제안된 방법은 파싱 테이블의 구성 요소로 오류의 항목이 명시되었더라도 재사용이 가능하기에 단말 심볼에 도달할 때까지 반복적으로 파싱을 수행한다. 그러나 이런 경우에 실제로 오류가 발생한 경우에는 불필요한 작업을 수행하게 되어서, 시간적으로 비효율적이다. 본 논문에서는 문법 분석을 통해서 실제로 오류가 발생한 상황을 미리 파악하여, 불필요한 파싱 작업을 막는 방법을 제안한다.

2장에서는 기본 정의와 표기법에 관해 언급하고, 3장에서는 기존의 알고리즘과 동기 예제를 제시한다. 4장에서는 해결 방안을 위한 정형식과 개선된 알고리즘을 기술하며, 5장에서 결론을 맺는다.

2. 기본 정의와 표기법

본 논문의 기본적인 정의와 표기법은 관련 기존 연구들[4-7]을 따른다.

· 이 논문은 한신대학교 학술연구비 지원에 의하여 연구되었음
† 종신회원 : 한신대학교 정보통신학과 교수
golee@hs.ac.kr
논문접수 : 2010년 3월 2일
심사완료 : 2010년 4월 2일

정보과학회논문지: 소프트웨어 및 응용 제37권 제6호(2010.6)

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

본 논문에서의 FIRST, FOLLOW 함수는 일반적인 LL(1)문법에서의 FIRST, FOLLOW을 확장한다: $\alpha \in (N \cup \Sigma)^*$라고 하자. $FIRST(\alpha) = \{X \mid \alpha \Rightarrow^* X\beta, X \in (N \cup \Sigma)\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}$, $FOLLOW(A) = \{X \mid S \Rightarrow^* \beta AX\gamma, X \in (N \cup \Sigma)\} \cup \{\varepsilon \mid S \Rightarrow^* \beta A\}$으로 정의한다.

LL-파싱 테이블 M은 N x (N ∪ Σ)에서 {A → α| A → α ∈ P} ∪ {오류}로의 함수이며, 다음과 같이 정의된다[4]:

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

(ii) (i) 이외의 경우에는 M(A, X) = 오류

단말 노드 t에서 파스 트리 T를 나눈다[4]는 다음 과정의 수행을 지칭한다: (i) 초기화로 n을 t라하고 과정 (ii)와 (iii)을 n이 더 이상 존재하지 않을 때까지 반복한다 (ii) n을 우측형제로 이동시킨다; 이때 우측형제가 존재하지 않는 경우에는 우측형제를 갖는 가장 가까운 조상 노드의 우측형제로 이동시킨다. (iii) T로 부터 n을 루트로 하는 부분 트리를 분리한다.

xyz가 기존 문자열이고 xy′z가 새로운 문자열이라 하자. xyz에 대한 *X-부분트리*는 x의 마지막 심볼 노드에서 파스 트리 T를 나누는 작업을 수행한 후에 남은 파스 트리이다(단, x가 ε인 경우에는 루트 노드를 지칭한다). $S \Rightarrow^*_{lm} xA_1 \cdots A_n$의 유도 과정에 대응하여 x를 파싱한 후에 파싱 스택의 내용은 $A_1 \cdots A_n$이 된다. X-부분트리로부터 이 스택의 내용을 얻을 수가 있으며, 파스 스택의 내용을 *X-문자열*이라 한다. xyz에 대한 *Z-부분트리들*은 y의 마지막 심볼 노드(y가 ε인 경우에는 x의 마지막 심볼 노드)에서 파스 트리를 나눈 후에 분리되어 나온 트리로 정의된다(단, 이때 ε이거나 모든 자식이 ε인 트리는 배제된다). 이 트리의 루트 심볼들을 *Z-문자열*이라 한다. 용어의 이해를 돕기 위해서 다음의 예제를 도입한다.

예제 1. G_1을 다음 규칙을 가진 문법이라고 하자: S → bAc, S → cB, A → C, C → D, D → a, D → c, B → b. 입력 문자열이 bac에서 bcc로 변경되었다고 하자. 이때 위의 설명에 따르면, x = b, y = a, z = c, y′ = c이며, 그림 1은 bac의 파스트리, X-부분트리, Z-부분트리들을 나타낸다. 이때 X-문자열은 Ac, Z-문자열은 c이다.

3. 기존 알고리즘과 동기 예제

본 절에서는 독자의 편의를 위해서 기존의 알고리즘[4]을 기술하며 본 논문의 동기가 되는 예제가 주어진다.

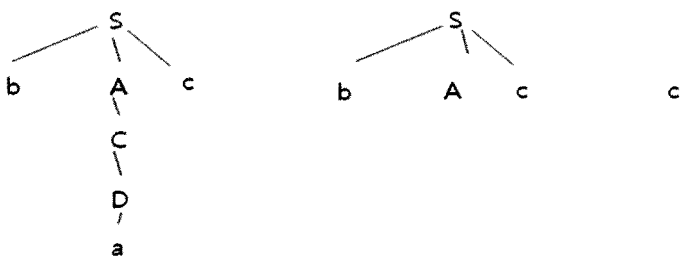
알고리즘 1 [4] (점진적 LL 파싱)

입력: 기존 문자열 xyz에 대한 파스 트리, 새로운 문자열 xy′z, 파싱 테이블 M

출력: xy′z에 대한 파스 트리

방법:

1. (a) 트리 T를 X-부분트리와 Z-부분트리들로 분리한다.
   (b) X-문자열 $X_1 \cdots X_n$은 P-Stack에 $X_n, \cdots, X_1$순으로 삽입(push)하며, Z-문자열 $Z_1 \cdots Z_m$은 I-Stack에 $Z_m, \cdots, Z_1$순으로 삽입한다.
   /*P-Stack과 I-Stack은 각각 파싱스택과 입력심볼스택을 의미한다.*/
   (c) y′을 I-Stack의 최상위(top)에 삽입한다.
2. **repeat**
   X와 Y를 각각 P-Stack의 최상위 심볼과 I-Stack의 최상위 심볼이라고 하자.
   (a) **If** X = Y **then**
   /* Y를 루트로 한 부분트리가 재사용된다. */
   - P-Stack으로부터 X, I-Stack으로부터 Y를 삭제(pop)한다.
   (b) **else if** X ≠ Y, X ∈ Σ, Y ∈ Σ **then**
   - 오류를 보고하고 끝낸다.



(a) bac의 파스트리 (b) X-부분트리 (c) Z-부분트리들

그림 1 bac로부터 bbc로의 변경시의 트리들

(c) **else if** M(X,Y) = p **then**

p = X → $X_1 \cdots X_n$이라 하자.

- P-Stack으로부터 X를 삭제하고 P-Stack에 $X_n,\cdots,X_1$순으로 삽입한다.

/* X노드에 대한 $X_1,..,X_n$의 자식노드를 만든다. */

(d) **else if** M(X,Y) = 오류 **then**

/* 다음의 (d-1)와 (d-2)는 본 논문에서 수정될 부분이다. */

(d-1) **If** *$Y \in N$ then*

트리상에 Y에 연결된 규칙을 $Y \rightarrow Y_1 \cdots Y_m$이라고 하자.

(i) I-Stack에서 Y를 삭제한다.

(ii) $Y_m,\cdots,Y_1$순으로 I-Stack에 삽입한다(이때 $Y_j \rightarrow \varepsilon$, $1 \le j \le m$인 경우는 제외한다).

/ Y_1은 새로운 Y가 된다. */*

(d-2) ***else** 오류를 보고하고 끝낸다. /* $Y \in \Sigma$ */*

until X= $

□

알고리즘 1에서 M(X,Y) = 오류인 경우에 대해서 실제로는 오류 상황이 아니라 Y의 부분 트리를 재사용할 수가 있기에 반복적으로 Y의 자식들에 대한 재사용 가능성을 시도하게 된다. 그러나 입력 문자열이 실제로 오류인 상황에서는 이런 시도가 행해지는 것은 불필요한 시간 낭비이다. 다음은 이런 경우에 대한 예제이다.

예제 2. 예제 1의 G_1에 대해서, 입력 문자열이 bac에서 cac로 변경되었다고 하자. 이때 x = ε, y = b, z = ac, y′ = c가 되며 이에 대한 X-부분트리와 Z-부분트리들은 그림 2에서 보여준다. 이때 X-문자열은 S, Z-문자열은 Ac이다. 그림 3은 알고리즘 1에 따른 P-Stack과 I-Stack의 변화를 나타낸다. 그림 3의 파싱 형상 (c)에서 문법 분석을 통해서 B와 A는 공통 심볼을 생성하지 않음을 미리 알 수가 있다. 따라서 파싱 형상 (d)-(f)의 과정은 불필요하며, 이 과정들 없이 바로 오류의 상태로 이동할 수가 있다. □

4. 정형식과 개선된 방법

점진적 LL 파싱의 근본 문제는 A가 기존 심볼이고 B가 새로운 심볼인 경우에 A를 재사용할 수가 있는지를 판단하는 문제이다. A = B인 경우에는 명백하게 재사용 가능하다. A ≠ B인 경우의 재사용 여부는 동일 심볼을 생성하는가에 달려있다. 동일 심볼 생성 여부의 파악을 위해서 본 논문에서는 다음의 L 관계를 이용한다.

정의 2. (L 관계) L은 N에서 (N ∪ Σ)로의 관계로 다음과 같이 정의된다.

A L X이기 위한 필요충분조건은 A → Xβ ∈ P이다. □

L^*는 L 관계의 재귀적 전이(reflexively transitive) 클로저(closure)이다.

다음에서 정의되는 d^T, d^N 함수는 L 관계를 이용한 심볼간의 동일 심볼 생성 여부 판정의 함수들이다.

정의 3. (d^T 함수) d^T 함수는 N × N × Σ에서 {true, false}로의 함수이다.

d^T(A, B, a) = true이기 위한 필요충분조건은 A L^* X, B L^* X, X L^* a을 만족하는 X가 존재한다. 그외의 경우는, d^T(A, B, a) = false이다. □

정의 4. (d^N 함수) d^N 함수는 N x N에서 {true, false}로의 함수이다.

d^N(A, B) = true이기 위한 필요충분조건은 어떤 a ∈ Σ에 대해서 d^T(A, B, a) = true이다. 그 외의 경우는 d^N(A, B) = false이다. □

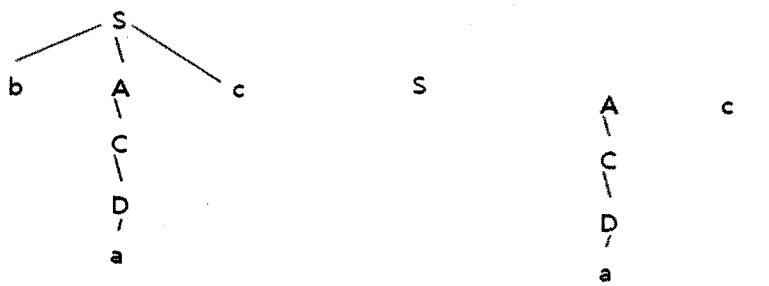
알고리즘 1의 단계 (d-1)과 (d-2)는 d^N, d^T 함수를 이용해 다음의 알고리즘 2와 알고리즘 3으로 개선할 수가 있다.

알고리즘 2. (개선된 방법 1)

입력: 기존 문자열 xyz에 대한 파스 트리, 새로운 문자열 xy′z, 파싱 테이블 M

출력: xy′z에 대한 파스 트리

방법:



(a) bac의 파스트리 (b) X-부분트리 (c) Z-부분트리들

그림 2 bac로부터 cac로의 변경시의 트리들

/* 알고리즘 1의 단계 (d-1)과 (d-2)는 아래로 대치되며, 나머지는 알고리즘 1과 동일하다. */

(d-1) **If** Y ∈ N **then**

If $d^N(X,Y)$ = true **then**

트리 상에 Y에 연결된 규칙을 $Y \rightarrow Y_1 \cdots Y_m$이라고 하자.

(i) I-Stack에서 Y를 삭제한다.

(ii) $Y_m,\cdots,Y_1$순으로 I-Stack에 삽입한다(이때 $Y_j \rightarrow \varepsilon$, $1 \le j \le m$인 경우는 제외한다).

/* Y_1은 새로운 Y가 된다. */

else 오류를 보고하고 끝낸다.

(d-2) **else** 오류를 보고하고 끝낸다 /*Y ∈ Σ */

□

알고리즘 3. (개선된 방법 2)

입력: 기존 문자열 xyz에 대한 파스 트리, 새로운 문자열 xy'z, 파싱 테이블 M

출력: xy'z에 대한 파스 트리

방법:

/* 알고리즘 1의 단계 (d-1)과 (d-2)는 아래로 대치되며, 나머지는 알고리즘 1과 동일하다. */

(d-1) **If** Y ∈ N **then**

if $d^T(X,Y,a)$ = true (a는 Y를 루트로 한 트리의 최좌측 단말 심볼이다) **then**

트리 상에 Y에 연결된 규칙을 $Y \rightarrow Y_1 \cdots Y_m$이라고 하자

(i) I-Stack에서 Y를 삭제한다

(ii) $Y_m,\cdots,Y_1$순으로 I-Stack에 삽입한다(이때 $Y_j \rightarrow \varepsilon$, $1 \le j \le m$인 경우는 제외한다).

/* Y_1은 새로운 Y가 된다. */

else 오류를 보고하고 끝낸다

(d-2) **else** 오류를 보고하고 끝낸다 /*Y ∈ Σ */

□

알고리즘 1에서는 M(X,Y) = 오류일 때에 Y ∈ N인 경우에 대해서 반복적으로 관련 규칙을 스택에 넣는 작업을 수행하였다. 한편 $d^N(X,Y)$ = false이거나 $d^T(X,Y,a)$ = false인 경우에는 명백한 오류상황이므로 이런 작업이 불필요하다. 이에 따라서, 알고리즘 2와 알고리즘 3에서는 각각 d^N 함수와 d^T 함수를 통해서 공통 심볼의 생성이 불가한 경우에는 바로 오류 처리를 하였다. 이로써 불필요한 시간을 줄일 수가 있다. 예제 2의 경우에 알고리즘 3을 적용 시에 그림 3의 파싱 형상 (c)에서 d^N(B, A) = false이기에 바로 오류상태로 처리될 수 있다.

알고리즘 2와 알고리즘 3은 d^N 함수와 d^T 함수 정보를 이용하고 있으며, 이를 위한 계산이 요구된다. 그러나 이는 파싱 시간 이전에 한번의 계산만을 요구하므로 실제 파싱 시간에는 영향을 주지 못한다. 별도로 필요한 파싱 시간을 고려해보면 알고리즘 2의 (d-1) 단계에서 d^N 함수 값의 회수 시간, 알고리즘 3의 (d-1) 단계에서 d^T 함수 값의 회수 시간이 필요하며, 이를 위해서는 O(1) 시간이 요구된다. 한편 (d-1) 단계는 M(X,Y) = error, Y ∈ N인 경우에 수행되며, (d-1) 단계의 수행 시에 실제로 오류가 발생한 상황일 수도 있고 아닐 수도 있다. 만일 오류가 발생하지 않는 상황이라면 O(1) 시간을 낭비한 결과가 된다. 한편 알고리즘 1의 경우에는 오류가 발생한 상황에서 단계 (i)과 (ii)을 수행하며, 각각에 대한 O(1) 시간과 O(m) 시간이 요구된다(m은 문법규칙의 우변의 최대길이이다). 따라서 이 경우에는 O(m) 시간을 낭비한 결과가 된다. 보다 정교한 분석을 위해서 실제로 오류가 발생할 확률을 r이라 가정하자. 이 때 알고리즘 1의 (d-1) 단계는 O(r*m)의 시간이 낭비되며 알고리즘 3의 (d-1) 단계는 O(1-r)의 시간이 낭비된다. 따라서 $r > 1/(m+1)$인 경우에는 알고리즘 3이 알고리즘 1보다 시간적인 면에서 개선됨을 알 수 있다. 한편 대부분의 문법 규칙의 우변은 2 이상임을 감안할 때 알고리즘 3이 시간적인 면에서 우월함을 알 수 있다.

별도로 필요한 공간으로는 알고리즘 2에서는 d^N 함수를 위한 O(N x N)의 공간이 요구되며 알고리즘 3에서는 d^T 함수를 위한 O(N x N x Σ)의 공간과 각 노드에 대한 최좌측단말 노드의 공간이 요구된다. 한편 알고리즘 3은 알고리즘 2에 비해 최좌측단말 심볼을 고려하여 정교하게 분석되었다. 이에 따라 알고리즘 2와 알고리즘 3은 응용 분야에 따라서 선택적으로 적용될 수 있다.

5. 결 론

본 논문에서는 기존 연구[4]에서 제안된 방법에 대한 실제로 오류가 발생한 상황 처리시에 수행되는 불필요한 작업 시간을 줄이는 방법을 제시하였다. 제안된 방법은 학생 프로그램과 같이 오류 발생이 빈번한 컴파일 환경에 유용하게 사용될 수 있다.

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] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and Compiling*, vols. 1 & 2, Prentice-Hall, Englewood Cliffs, NJ, 1972, 1973.
[7] S. Sippu and E. Soisalon-Soininen. *Parsing Theory*, vols. I & II, Springer, Berlin, 1990.

이 경 옥

1990년 2월 서강대학교 전자계산학과 졸업(학사). 1992년 8월 한국과학기술원 전산학과 졸업(석사). 2000년 2월 한국과학기술원 전산학과 졸업(박사). 2000년 8월부터 한신대학교 정보통신학과 근무. 현재 한신대학교 정보통신학과 부교수. 관심분야는 프로그래밍 언어와 컴파일러 등