공학 연구논문집 제24권 제2호 pp. 125~139, 울산대학교
Journal of Engineering Research Vol. 24, No. 2, pp. 125~139, 1993.
University of Ulsan.

# 생략 가능한 특수 터미날 심볼의 오류보정

정영필* · 이명준** · 박양수**
*컴퓨터공학과 · **전자계산학과

〈요 약〉

본 논문에서는 프로그램 작성시 주어진 특수 터미널 심볼의 생략 가능성을 분류하기 위한 형식론을 제안한다. 제안된 방법을 만족하는 문법의 경우, 생략된 심볼을 어휘 분석단계(lexical analysis phase)나 구문분석단계(syntax analysis phase)와 같은 컴파일하는 각 단계에서 보정할 수 있게 한다. 그리고, 제시된 방법에 근거하여, 각 컴파일 단계에서 복구할 수 있는 심볼이 포함된 프로그래밍 언어들의 특성을 논의한다. 또한, 제안된 방법론을 실제 구현하였으며 Ada나 Modula-2와 같은 주요 프로그래밍 언어들에 적용 하였다. 이들 언어들에 대해서는 심볼을 생략함으로써 야기되는 오류를 만족할만하게 보정해 준다. 제안된 방법을 적용하기 어려운 프로그래밍 언어들에 대한 특성 또한 논의 되었다.

---

# Error Repair for Omittable Special terminal Symbols

Young-Phil Cheung* · Myung-Joon Lee** · Yang-Su Park**
* Dept. of Computer Engineering · ** Computer Science

〈Abstract〉

This paper presents a formalism for classifying the possibility of omitting a given special terminal symbol when writing programs. For the grammars satisfying the proposed formalism, the formalism enables to repair the symbol during phases of compilation such as lexical analysis or syntax analysis. Based on the method, this paper also discusses the characteristics of programming languages in which the symbol is repaired during such phases. The method was implemented and applied to major programming languages such as Ada and Modula-2, repairing satisfactorily the errors caused by omitting the symbol. In

addition, the characteristics of the programming languages to which the presented method is hardly applicable, are discussed.

## 1. 서 론

컴파일러의 역할 중 하나는 입력 프로그램에 대하여 가능한 많은 문법적인 오류를 발견하는 일이다. 구문분석을 진행하는 도중에 오류가 발견되면 오류의 원인을 알리고 더 이상 구문분석이 진척될 수 없으므로 종료하는 것이 일반적이나, 프로그램 개발의 능률을 올릴 수 있도록 하기 위해서는 프로그램에 산재된 오류를 가능한 많이 지적 할 수 있게 하는 것이 바람직하다. 그러므로 이를 위한 오류 복구(error recovery) 및 오류 보정(error repair)이 필요하다. 오류 복구는 입력 프로그램의 구문오류에 대하여 구문분석이 계속 진행될 수 있도록 파서를 복구시키는 기능이고, 오류 보정은 구문적으로 틀린 문장에 대하여 정정하는 기능으로 이들에 관한 연구가 많은 주목을 받아왔다[3,4,10]. 그러나 이미 알려진 방법들은 다소 심각한 단점을 가지고 있다. [2,5]의 방법은 hand-coded 복구루틴의 사용을 요구하는 근본적으로 임기응변식의 방법으로 해결하고 있으며, [8,11]에서는 어떤 문법적인 오류를 만났을 때 입력의 특정 부분을 완전히 무시해버리도록 강요하고 있다. [17]의 방법은 요구되는 공간과 처리시간에 있어서 많은 부담을 가지기 때문에 비 현실적이다. [13,16]에서는 복구의 방법에 대하여 전혀다른 문법 구조로 인식도록하는 부당한 선택을 취하기도 한다. 그리고 [3,9]에서는 문법에서 사용되는 터미널 심볼의 삽입 및 삭제비용에 기초하여 오류가 발생된 시점에서 오류심볼을 삭제하거나 최소비용 삽입 스트링을 찾아내어 그것을 오류 심볼 앞에 삽입하여 보정하는 방식을 제안하였다.

본 논문에서는 근원적인 오류로 기인한 복구에 따른 문제가 아니라 프로그래밍 언어에서 생략될 수 있는 특수 터미널 심볼의 오류 보정에 대해서 논의하고자 한다. 프로그래밍 언어에서 분명하고도 빈번히 나타나는 터미널 심볼에 대하여 생략될 수 있도록 표기법에 융통성을 제공한다면, 그 터미널에 연루된 사소한 구문적인 오류에 프로그래머가 관심을 두지 않아도 되므로 프로그램 개발에 소요되는 노력을 절감할 수 있다. 그러므로 특수 터미널 심볼에 대하여 생략할 수 있게 하고 이를 컴파일러가 언어 문법에 위배되지 않도록 자동적으로 보정하는 것은 의미가 있다. 이를 목적으로 본 논문에서는 생략된 특수 터미널 심볼을 어휘분석이나 구문분석 단계에서 그 터미널 심볼을 보정해 줄 수 있는지 여부에 근거하여, 그 언어에서의 특수 터미널 심볼의 생략 가능성을 적절한 형식론을 이용하여 분류하였다.

어떠한 유형의 언어[12,19]는 본 논문에서 고안된 알고리즘을 사용하면 어휘분석 단계에서 생략된 특수 터미널 심볼에 대하여 오류보정이 가능하다. 어휘분석 단계에서 보정할 수 없는 언어[Ada, Modula-2, 등] 중에는 구문분석 단계 즉 구문분석 상태 상에서 보정을 행할 수 있는 것도 있다. 따라서 보정을 행할 수 있는 단계와 생략 가능한 특수 터미널 심볼과는 상호 연관성이 있다. 본 논문에서는 이들의 관계에 대하여서도 논의하며, 그 관계에 따라 특수 터미널 심볼을 분류하기 위한 형식론을 제안한다. 그리고 제안된 형식론에 따라 생략 가능한 특수 터미널 심볼들이 어떻게 분류되는 지와 그 터미널 심볼들을 보정하기 위

한 어휘분석 단계와 구문분석 단계에서의 보정방법이 고안 되었으며 각각의 단계에서 보정될 수 있는 주요 언어를 보인다. 생략된 특수 터미널 심볼을 구문분석 단계에서 자연스럽게 보정하기 어려운 언어 부류에 대해서는 이들이 안고 있는 문제점을 설명하고, 이와 관련되어 다른 언어(Ada나 Modular-2와 같이 구문분석 단계에서 보정 될 수 있는 언어)에서는 어떤 문법적인 구조의 도입으로 해결이 되었는지 예를 들어 설명한다.

2장에서는 LR에 근거한 구문분석[14, 15, 18, 20]을 위한 기본 정의 및 용어가 먼저 서술되고 어휘분석단계 또는 구문분석 단계에서 생략된 특수 터미널 심볼을 보정하기 위한 방법론도 제시된다. 3장에서는 제안된 형식론에 입각하여 생략된 특수 터미널 심볼에 대한 오류보정기를 2장에서 언급될 방법론에 근거를 두고 실제 구현하여 몇몇 주요 프로그래밍 언어에 적용한 결과가 제시된다.

## 2. 생략된 특수 심볼의 보정방법

### 2.1 기본정의 및 용어

이 절에서는 본 논문의 형식론에서 사용하게 될 문맥자유 문법 및 LR 구문분석을 위한 기본 정의와 용어를 서술한다. 여기에 기술되지 않은 기본적인 표기법은 Aho와 Ullmann[2]의 표기법을 따랐다.

**정의 1** 문맥 자유문법 (Context-Free-Grammar:CFG)

문맥 자유문법 G는 4개의 튜플로 이루어진다. 즉

$G=(N, T, P, S)$

여기서, $N$=넌터미널(nonterminal) 심볼들의 유한집합.

$T$=터미널(terminal) 심볼들의 유한집합. 단, $N \cap T = \phi$

$P = N \times (N \cup T)^*$

$S$=문법의 시작 심볼. 단, $S \in N$

본 논문에서 심볼에 대한 각 표기는 다음과 같은 의미로 사용된다.

- a, b, c ... (알파벳 소문자 시작부): $T$에 있는 심볼들
- A, B, C ... (알파벳 대문자 시작부): $N$에 있는 심볼들
- u, v, w ... (알파벳 소문자 종료부): $T^*$에 있는 스트링들
- U, V, W ... (알파벳 대문자 종료부): $V$에 있는 심볼들
- α, β, γ... (그리스문자 시작부): $V^*$에 있는 스트링들

  단, $V = N \cup T$

- P의 각 요소 (A,α)에 대하여 production이라 부르며 A->α로 표기한다.

**정의 2** 확장(Augmented) CFG

주어진 CFG $G=(N, T, P, S)$에 대한 확장 CFG $G'$는 $N$에 있지 않는 새로운 시작심볼 $S'$과 새로운 production $S' \rightarrow S$을 가지는 문법이다. 확장문법 $G'$에는 쓸모없는 심볼이 없다고 가정한다.

**정의 3** LR(1) 상태 및 LR(1) 아이템(item)

LR(1) 상태 q는 LR(1) 아이템들의 집합이며, [A->α.β,a]의 형태를 가진다. 여기서, A->α.β는 production이며, a는 production A->α.β의 lookahead 문자로 $a \in T \cup \{\$\}$이다. 만약 $\alpha \neq \varepsilon$이거나 $A = S'$이면 LR(1) 아이템 [A->α.β,a]을 커널 아이템이라 하며 나머지 경우는 클로즈(closure) 아이템이라 부른다.

**정의 4** IEDP(Immediate Error Detection Property)

오류 토큰이 lookahead에 나타나자마자

구문분석기가 즉시 문법 오류임을 감지할 수 있을 때 그 구문분석기는 IEDP 성질을 만족한다.

**정의 5** Prev, Next

$Prev(t)=\{a|S=\rangle uatv,\ u,v\in T^*,\ t,a\in T\}$

$Next(t)=\{a|S=\rangle utav,\ u,v\in T^*,\ t,a\in T\}$

## 2.2 어휘분석 단계에서의 보정

구문분석기 전반부의 일반적인 동작은 어휘분석기로부터 토큰들을 받아 파서 트리(parse tree)를 형성해 감과 아울러 문법적인 오류를 감지하는 것이다. 정상적인 토큰들의 스트림을 가정하면 문법오류의 감지는 구문분석의 진행 중에 이루어진다. 하지만 입력 언어의 종류에 따라 어휘분석기에서 하나의 토큰을 생성해 내었을 때 즉시 문법적인 오류임을 감지하고 보정해줄 수 있는 경우가 있다. 이는 잘못된 입력 문장에 대한 오류보정의 문제가 아니고 생략될 수 있는 특수 터미널 심볼에 대한 보정의 문제가 바로 그런 경우이다.

생략된 특수 터미널 심볼에 대하여 어휘분석 단계에서 오류보정을 할 수 있다면 언어 표기에 있어서 융통성을 제공할 수 있음과 동시에 구문분석기에서 가져야하는 터미널 심볼의 오류보정에 따른 부담을 덜 수 있다. 다음의 CFG를 예로 그 가능성을 생각해 보자.

P: S -> aAB
A -> C | AtC
B -> b | ε
C -> c | d

그림 2.1 어휘분석 단계에서 보정 가능한 문법

위 문법에 맞는 문장을 쓸 때에 터미널 심볼 t를 생략하여 보자. 즉 아래에 나타낸 것과 같이 그림 2.2(1)의 입력 대신에 그림 2.2(2)와 같이 입력 하였을 때 터미널 심볼 t가 보정될 수 있을 지가 문제가 된다.

| actdtcb | acdcb |
|---|---|
| (1) | (2) |

그림 2.2 터미널 심볼 t의 생략

이러한 문제를 어휘분석 단계에서 해결하기 위해 다음 정의 6을 정의한다.

**정의 6** strongly-omittable

t∈T에 대하여

t is *strongly-omittable* iff

$Next(t)\cap Next(Prev(t))=\phi$

그림 2.3에서 Saved_Token은 Token_t 문법규칙 "A->αtβ"의 형태에 대하여 t 다음에 올 수 있는 심볼들과 t 앞에 올 수 있는 심볼들의 다음 심볼에 대하여 공통된 심볼이 존재하지 않으면 이때 심볼 t는 생략 가능하다. 왜냐하면 t의 이전에 나타날 수 있는 심볼들에 대한 다음에 올 수 있는 심볼들이 t 이후에 올 수 없으므로, 이전 토큰심볼의 Next가 t가 될 수 있고 현재 토큰심볼의 Prev가 t가 될 수 있다면 특수 터미널 심볼 t가 생략 되었다고 볼 수 있기 때문이다. 이 조건을 만족하는 생략 가능한 특수 터미널 심볼 부류를 strongly-omittable로 분류한다. 임의의 언어에서 정의 6을 만족하는 특수 터미널 심볼 t는 생략될 수 있으며 그 심볼 t의 Prev들에 대한 Next와 t의 Next들간에 공통요소를 가지지 않기 때문에 생략 되어도 자연스럽게 어휘분석 단계에서 보정해 줄 수 있다. 어휘분석기는 보정을 행하기 위해서 생략 가능한 특수 터미널 심볼 t에 대하여 Prev(t)와 Next(t)를 가지면 된다. 따라서 주어진 언어에서 t가 정의 6을 만족한다면 어휘분석기가 현재의 토큰을 구문분석기에게

되돌리기 전에 다음의 알고리즘 Next_Token()에 의해 생략된 심볼 t를 보정해

```
Initialize:
   Previous_token := Null
   Saved_Token := Null

ALGORITHM Next_Token()
BEGIN
      Previous_Token := Current_Token
      IF Saved_Token THEN
            Current_Token := Saved_Token
            Saved_Token := Null
            RETURN Current_Token
      ENDIF
      Current_Token  := get token from Scanner
      IF (Previous_Token is in Prev(Token_t)) AND
            (Current_Token is in Next(Token_t)) THEN
            Saved_Token := Current_Token
            Current-Token := Token_t
      ENDIF
      RETURN Current_Token
END Next_Token.
```

그림 2.3 어휘분석 단계의 보정을 위한 알고리즘

에 대하여 보정이 일어날 때 현재 생성된 토큰(Current_Token)과 다음번 Next_Token() 수행시 순서적으로 우선 되돌리기 위하여 도입 되었다.

실제적인 예를 보기위해 그림 2.1의 문법에서 Prev(t)와 Next(t) 값을 구하면 다음과 같다.

Prev(t)={'c', 'd'}
Next(t)={'c', 'd'}

그리고 Next(Prev(t))={'b', 't'}이므로 정의 6을 만족한다. 따라서 그림 2.1의 문법에서 터미널 심볼 t는 strongly-omittable 이며 입력 문장에서 생략될 수 있고 어휘분석 단계에서 보정할 수 있다.

만약 주어진 언어에서 어떤 특수 터미널 심볼 t가 정의 6을 만족한다면 Prev(t)와 Next(t)의 값을 기초로 해서 어휘분석기가 입력문장의 다음 토큰 심볼을 생성하는 시점에서 이전의 토큰이 Prev(t)에 속해 있고 현재의 토큰이 Next(t)에 속해 있다면 현재 위치에 터미널 심볼 t가 보정 되어야 함을 알 수 있다. 이러한 범주의 언어에 대하여 실제적인 예를 3장을 통하여 보이게 되며 그림 2.3의 알고리즘을 기초로 보정 시스템을 구현하게 된다.

## 2.3 구문분석 단계에서의 보정

주어진 언어에 대하여 어떤 터미널 심볼 t가 생략 가능한 지를 살펴 볼 때, 터미널

심볼 t는 정의 6을 만족해야 하지만 일반적으로 이를 만족하는 특수 터미널 심볼은 매우 드물다. 예를 들어 다음에 주어진 CFG 문법을 생각해 보자.

P: S -> aAB
A -> C | AtC
B -> b | ε
C -> b | c

그림 2.4 어휘분석 단계에서 터미널 심볼 t가 보정 불가능한 문법

위 문법에 대하여 생략 가능한 터미널 심볼인 t의 Prev(t)와 Next(t)의 값은 다음과 같다.

Prev(t) = { 'b', 'c' }
Next(t) = { 'b', 'c' }

그림 2.4의 문법에 대하여 가능한 입력 스트링 그림 2.5(1)에 대하여 그림 2.5(2)와같이 표기하고 Next_Token() 알고리즘으로 보정을 수행한다면 그림 2.5(3)과 같은 보정이 일어난다. 하지만 이 보정은 문법적으로 위배가 된다.

actbtcb (1) acbcb (2) actbtctb (3)

그림 2.5 어휘분석 단계에서의 잘못된 보정 시나리오

그림 2.4 문법의 경우 Next(Prev(t))= { 'b', 't' }이므로 근본적으로 정의 6을 만족하지 않는다. 그러므로 2.4의 문법에서 터미널 심볼 t는 입력문장에서 생략될 경우 자연스럽게 보정할 수 없다.

임의의 LR 상태 q에 존재하는 모든 서로 다른 LR(1) 커널 아이템 쌍 〔A->α.tβ, a〕, 〔B->γ.δ, b〕에 대하여 입력 문장에서 터미널 심볼 t가 생략되고 LR 상태 q에서 보정될 수 있는 조건을 위하여 다음과 같이 정의 7을 정의한다.

**정의 7** *weakly-omittable*
t is *weakly-omittable* iff
$First(\beta)\oplus\{a\} \cap First(\delta)\oplus\{b\} = \phi$

정의 7이 나타내는 의미는 LR(1) 상태 q에서 t가 생략될 경우 β의 First들과 δ의 First들간에 공통 부분이 없다면 t가 생략되어도 상태 q에서 현재 t를 진행할 순서에 있고, 다음 토큰 심볼이 (First(β)⊕{a})에 있다면 t를 보정해 줄 수 있다. 그리고 β나 δ가 nullable하다면 lookahead 심볼(a, b)을 대상으로 고려하면 되므로 위와 같이 정의될 수 있으며, 주어진 언어에서 이러한 조건을 만족하는 특수 터미널 심볼 t를 weakly-omittable로 분류한다. 그러나 α가 nullable하다면 t가 생략됨으로 인해 β에서 파생될 터미널 심볼이 α에 의해 생성될 수 있는 심볼로 인식되어 shift 혹은 reduce가 일어나 원래의 문법적 의미와는 다르게 구문분석이 진행될 수 있다. 따라서 이 조건을 부가하여 다음을 정의한다.

**정의 8** *omittable*
t is *omittable* iff
$First(\beta)\oplus\{a\} \cap First(\delta)\oplus\{b\} = \phi$
and $\alpha \neq\!\!\!\!\!\!\cdot\;\; \varepsilon$

이상의 omittable들에 대한 정의(정의 6, 7, 8)로부터 다음의 정리들이 유도될 수 있다.

**정리 1**
어떤 터미널 심볼이 strongly-omittable이면 omittable이다.

**정리 2**
어떤 터미널 심볼이 omittable이면 weakly-omittable이다.

주어진 언어에서 정의 8을 만족하는 특수 터미널 심볼 t는 상태 q에서 보정할 수 있다. 그리고 정의 6,7,8의 범주에 들지 않는 터미널 심볼들은 unomittable 심볼로 정의한다. 3장 후반부에서 이 단계의 심볼을 포함하는 언어들에 대해서 발생되는 문제점을 분석하고, 다른 언어(보정 될 수 있는 언어)에서는 어떠한 문법적인 구조체에 의해서 해결 되었는지를 비교를 통해 보이게 될 것이다.

## 2.4 생략 가능한 특수 터미널 심볼의 분류와 구문분석기의 오류보정

이상의 정의로부터 생략된 특수 터미널 심볼에 대하여 보정할 수 있는 어휘분석 혹은 구문분석기의 존재 여부에 따라 분류할 수 있으며 아래와 같이 정리할 수 있다.

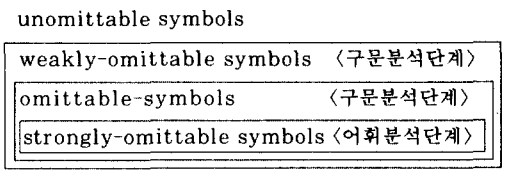



그림 2.4 생략 가능한 특수 터미널 심볼의 분류

기존하는 대부분의 언어에서 정의 8을 만족하는 심볼은 좀처럼 보이지 않는다. 언어 설계에 있어서도 특수 터미널 심볼을 생략할 수 있도록 하기 위해 문법 내에 존재하는 모든 터미널 심볼들에 대하여 정의 8을 만족하도록 문법규칙의 서술에 제약을 가한다는 것은 타당하지 않는 것으로 보인다.

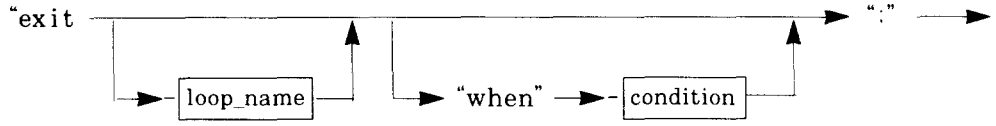
구문분석 단계에서 생략된 특수 터미널 심볼의 보정 문제를 고려 할 때, weakly-omittable 심볼의 경우 구문분석 단계에서 오류 복구기법을 이용하면 보정 가능할 수 있다. 터미널 t가 생략되고 α가 nullable 함으로써 발생 될 수 있는 상황에 대하여 두가지 전략을 가질 수 있는데 편이상 이들을 우선보정 원칙과 정상진행우선 원칙이라 지칭한다. 우선보정 원칙은 LR(1) 상태 q에서의 문법규칙 A->α.tβ에 대하여 α가 nullable하고 다음 문법적 심볼이 생략 가능한 심볼 t라면 t를 보정함에 우선권을 주는 것이다. 즉 지금 처리해야될 입력 토큰은 β에서 유도될 심볼임을 가정하는 것이다. 그리고 t를 우선 보정함으로써 오류가 발생된다면 이 오류는 t를 보정함으로써 발생된 오류로 간주할 수 있으므로 t에 관계된 오류 복구를 시켜준다. 정상진행우선 원칙은 다음 토큰 심볼을  에서 유도된 것으로 간주하는 것이다. 즉 정상적인 구문분석 동작에 우선권을 주어 입력 토큰에 대하여 타당한 구문분석 상태로의 전이가 가능하다면 계속 구문분석이 진행 되도록 하는 것이다. 그러나 이 경우는 특수 터미널 심볼을 생략함으로 인한 오류 상황에 대하여 복구하는데 따른 문제점을 내포하고 있다. 왜냐하면 이미 많은 입력 토큰에 대하여 shift 혹은 reduce를 통하여 구문분석이 진행된 이후에 오류가 판명될 경우도 있기 때문이다. 그러한 상황에서의 복구 문제는 많은 처리의 부담을 가져온다. 예를 들어 Ada 프로그램에서 exception 구조에 대한 다음 문장을 그 예로 생각해 보자(여기서 생략 가능한 심볼을 세미콜론이라 가정하자).

문제는 오류가 발견되는 시점이다. 위 예

```
...
exception
   when NUMERIC_ERROR =>
        MACHINE_OVERFLOWS := TRUE;
```

```
        exit;
    when CONSTRAINT_ERROR =>
        put ("Overflow raises CONSTRAINT_ERROR.");
    ...
```

위의 Ada 프로그램 부분에서 exit 다음의 세미콜론을 생략할 경우 이어지는 문장인 when 이하 절을 exit 구문에 이어지는 구문구조로 인식하여 구문분석을 계속 진행할 수 있다. 참고로 Ada의 exit 구문에 대한 구문도는 다음과 같다.




제의 경우 구문분석의 동작은 "when"에 대하여 shift, "COSTRAINT_ERROR"에 대하여 shift, "=>"에 대하여 부등호 심볼로 인식하여 reduce를 수행하고 "put" 토큰을 보는 순간 오류를 발생 시킨다. 이와 같이 이미 상당한 shift와 reduce동작을 거쳐서 진행한 이후에 오류를 발견하기 때문에 정상진행우선 원칙을 적용하면 복구에 따른 부담이 많다. 그러나 문법이 정의 8을 만족한다면 정의에서 나타내 주듯이 비 정상적인 shift는 일어날 수 없으므로 이 기법이 유리할 것이다. 첫번째 기법의 경우는 복구에 대한 비용을 생각해 볼 때 다소 부담이 적은 접근방법이다. 왜냐하면 미리 토큰을 보정해 주고 오류가 발생할 경우에 이를 복구해 주기 때문이다. 이 경우는 세미콜론의 보정에 의한 오류만 고려하면 된다. 또한 매번 토큰이 발생될 때마다 보정해야 하는 문제점을 생각할 수 있는데 3장에서 언급되겠지만 프로그래머가 제공하는 구조정보를 이용하면 보다 현실적으로 적용할 수 있다. 본 논문에서는 첫번째 방법인 우선보정 원칙을 통하여 적용 가능한 예와 함께 reduce 스택을 사용해서 구문분석 단계에서 어떻게 보정할 수 있는지를 3장에서 구현한다.

정형적(canonical) LR(1) 구문분석기의 경우는 IEDP의 특성을 지니고 있다[1]. 그러나 SLR(1)이나 LALR(1)과 같이 LR(1)에 근거한 기법들은 lookahead들을 추출하기 위해 근사(approximation) 시키기 때문에 부당한 심볼이 lookahead에 나타났을 때 잘못된 shift는 일어날 수는 없지만 잘못된 reduce는 계속적으로 일어날 수 있다[6]. 따라서 SLR(1)이나 LALR(1)과 같은 경우는 IEDP를 만족하지 않으며 복구를 위해 reduce 스택이 유지 되어야 한다[3,9]. Reduce 스택은 reduce한 상태를 저장하게 되며, 터미널 심볼에 대한 shift 동작이 일어날 때 초기화 된다. 오류를 만나게 되면 reduce 스택을 참조하여 오류 심볼이 처음으로 사용된 reduce가 일어나기 전의 상태로 되돌아 갈 수 있게 하는 것이다.

다음 장을 통하여 생략된 특수 심볼의 실제 보정 시스템을 몇몇 주요 언어에 대하여 구현한 결과를 보인다.

## 3. 응 용

본 장에서는 2장을 통하여 제시된 형식론에 입각하여 실제 프로그래밍 언어에서의 적용 여부를 분석하고 이를 실현한다. 생략

가능한 특수 터미널 심볼의 적용 예제는 세미콜론(;)이며 end-of-line에서 세미콜론을 생략할 수 있도록 하고 이를 어휘분석 단계 구문분석 단계에서 보정할 수 있는 시스템을 만들고, 이 시스템을 주요 언어들에 적용하여 그 실용성을 입증한다.

일반적으로 어휘분석 단계에서 화이트 스페이스로 다루어지는 end-of-line은 고려할 필요가 있다. 왜냐하면 프로그램 작성시 end-of-line의 사용은 문법적인 구문구조에 있어서 분리를 의미하며 프로그래머가 자주 사용하는 매우 강력한 구조정보이기 때문이다. 이를 무시하기 보다는 컴파일러 입장에서 수용하는 것이 바람직하다. 더욱이 end-of-line에서 세미콜론은 대부분의 언어에서 문장의 종료자로써 사용된다. 어떤 의미에서는 end-of-line과 세미콜론은 중복 되어서 사용되는 심볼로 생각할 수 있다. 따라서 생략할 수 있는 강력한 심볼이며 이것의 표기에 대한 융통성을 제공하는 것이 바람직하다. 또한 세미콜론을 문장의 종료자(terminator)로서가 아니라 단지 문장의 구분자(separator)로서 사용 될 수 있도록 해준다. 이러한 응용의 중요성은 [7]에 잘 나타나 있다. 실제로 학생들의 프로그램을 대상으로 실험 해 본 결과 문법 오류의 15% 정도가 세미콜론으로 인한 오류인 것으로 감안해 볼 때 고려할 가치가 있다.

구현 시에 사용한 특수 터미널 심볼에 대한 보정 기능을 가진 구문분석기 생성 시스템은 UNIX 환경하의 구문분석기 생성기인 yacc를 확장하여 설계 하였다.

## 3.1 Lex와 Yacc

Yacc(Yet Another Compiler Compiler)는 UNIX O/S 환경 하에서 잘 알려진 구문분석기 생성 시스템으로 입력 문법을 받아들여 해당 구문분석 테이블과 그 구동(driver) 프로그램을 C 언어로 된 원시 프로그램(y.tab.c)으로 출력한다. Lex (Lexical analyzer)는 입력 토큰들을 위한 정규식(regular expression)으로 구성된 입력 명세(input specification)를 받아들여 토큰분석 루틴들이 포함된 lex.yy.c를 출력한다. 이 C 프로그램들을 컴파일하면 수행가능한 구문분석기와 어휘분석기가 각각 만들어 진다. 본 논문에서의 접근방법은 yacc 원시 프로그램을 수정하여 2장에서 언급한 보정에 필요한 테이블 정보를 추가로 원시 프로그램에 출력 시키고 구동 프로그램이 이에 반응할 수 있도록 수정한다. 구문분석기 구동 루틴인 yyparse()는 토큰을 가져오기 위하여 토큰분석기인 yylex()를 호출한다. 어휘분석 단계에서 오류보정을 할 경우는 Yacc의 원시 프로그램이나 Lex의 원시 프로그램에 대하여 수정을 가할 필요가 없으며 yylex()와 yyparse() 사이에서 보정을 위해 작성된 루틴(편의상 루틴의 이름을 Next_Token()이라 가정)이 개입할 수 있도록 #define 문을 사용한다. 다음 절에서 이에 대한 구체적인 알고리즘을 보인다.

## 3.2 SR, ICON[12,19]

이 범주에 속하는 언어들은 세미콜론에 대하여 정의 5를 만족하므로 어휘분석 단계에서 세미콜론을 보정할 수 있다. 어휘분석 단계의 보정 시스템을 구현하기 위해서 다음 사항을 고려 해야 한다.

1) lex의 입력에서 end-of-line을 입력 토큰으로 인식할 수 있도록 다음과 같은 정규식을 Lex의 입력 명세 화일에 포함 시켜야 한다. 또한 편의상 세미콜론의 토큰을 TK_OMITABLE로 정의한다.

```
\n    return TK_EOLN;
 :    return TK_OMITABLE;
```

2) TK_EOLN을 다루고 보정을 수행하는 루틴인 Next_Token()를 작성하고 yyparse()와 yylex() 사이에서 보정 루틴이 개입할 수 있도록 yacc의 입력 명세의 C 언어 기술 부분에 다음과 같이 정의한다.

```
# define yylex   Next_Token
```

3) 생략 가능한 특수 터미널 심볼(본 논문의 예제인 세미콜론)에 대하여 2장에서 언급한 Prev(t)와 Next(t)를 구하여 저장한다(편의상 그 정보를 저장할 테이블을 omit_table[ ]이라 한다). 따라서 이 테이블의 크기는 터미널 심볼의 개수와 동일하며 다음과 같은 정의를 가진다.

문법내의 모든 터미널 심볼 t에 대하여 end-of-line에서

omit_table[i] =
- PREV : 세미콜론 앞에 t가 올 수 있다.
- NEXT : 세미콜론 뒤에 t가 올 수 있다.
- BOTH : 세미콜론 앞에도, 뒤에도 t가 올 수 있다
- NONE : 앞과 뒤에 올 수 없다. 즉 문장의 중간에 나타나는 터미널 심볼이다.

여기서, i는 터미널 심볼 t에 대한 토큰 값
PREV = 1, NEXT = 2, NONE = 0
BOTH = PREV & NEXT

omit_table의 값은 주어진 문법에서 Prev(t)와 Next(t)를 계산하여 얻어진다. 이상의 정보로부터 end-of-line에서 세미콜론을 보정하는 루틴인 Next_Token() 루틴의 상세한 알고리즘은 FUNCTION Next_Token()과 같다.

SR, ICON 언어에서의 세미콜론은 정의 6을 만족하여 strongly-omittable 심볼이다. 따라서 어휘분석 단계에서 알고리즘 Next_Token()를 이용하여 end-of-line의 세미콜론을 보정할 수 있다.

```
FUNCTION Next_Token()
/* cur_tok  : current token or next token                    */
/* prev_tok : previous token                                 */
/* sav_tok  : saved token to repairing the semicolon         */
BEGIN
    prev_tok := cur_tok
    IF sav_tok != Null THEN      /* there exist a saved token */
        /* make the saved token to the current */
        cur_tok = sav_tok
        RETURN cur_tok
    END
    cur_tok = yylex()   /* get next token from yylex() */
```

```
IF cur_tok != TK_EOLN THEN    /* just return it */
      RETURN cur_tok
IF cur_tok = TK_EOLN  THEN
     cur_tok = yylex ()    /* get next token from yylex() */
     IF ((omit_table[prev_tok] = PREV) .AND.
         (omit_table[cur_tok] = NEXT))
         /* make it to a saved token */
         sav_tok = cur_tok
         cur_tok = TK_OMITABLE      /* repair semicolon */
         RETURN cur_tok
     END
  END
  RETURN cur_tok
END Next_Token.
```

### 3.3 Ada, Modulo-2

Ada와 modulo-2의 문법은 세미콜론에 대하여 정의 7을 만족하며 α가 nullable한 경우가 발생한다. 그러나 이 언어들은 우선 보정 원칙을 적용하면 구문분석의 단계에서 세미콜론 보정을 수행할 수 있는 언어들이다. 따라서 세미콜론 보정으로 인한 오류복구를 고려해야 하며 이를 위해 reduce 스택이 유지 되어야 한다[3,9].

그러므로 생략된 터미널 심볼 t의 잘못된 보정을 복구하기 위해 구문분석 루틴 전반에 걸쳐 오류처리 및 reduce 스택 관리를 위한 코드가 반영되어야 한다. Reduce 동작에 대하여 현재 상태를 reduce 스택에 저장하며, shift 동작에 대해서는 reduce 스택을 초기화하게 된다. 오류가 발생허다면 reduce 스택을 참조하여 오류심볼이 처음으로 사용된 reduce가 일어나기 전의 상태로 돌아가며 다시 reduce 스택을 초기화시키는 이와같은 일련의 과정이 구문분석기 생성기(Yacc)에 의해 생성될 구문분석기 루틴에 반영되어 포함되어야 한다.

토큰이 발생될 때마다 매번 보정 여부를 검사하여 보정해야 하는 부담을 덜기 위해 프로그래머가 제공해 주는 구조정보를 이용하면 보다 실제적으로 적용할 수 있다. 그 구조정보는 3장 서론부에서 언급했듯이 end-of-line 정보이다. 즉 end-of-line에서 세미콜론을 보정 하는 문제를 고려하면 현실적인 응용을 생각할 수 있다. 아울러 이러한 구조정보를 이용하는 입장에 있어서 정상적인 구문분석의 진행에 우선권을 부여하기 보다는 end-of-line에서 세미콜론을 우선보정 하는 것이 보다 의미가 있다. 왜냐하면 대부분의 언어에서 세미콜론은 문법적인 구문구조를 닫아주는 역할을 한다. end-of-line 문자가 명시 되었다는 의미 역시 이러한 맥락과 일치한다. 이는 대부분의 프로그래머가 취하는 공통의 프로그램 스타일이다. 이러한 구조정보의 수용은 프로그램 작성시의 유연성에 영향을 줄 것이다.

### 3.4 C, Pascal

이 부류에 해당되는 언어의 문법은 LR 상태 q에 대하여 정리 7을 만족하지 않는다. 이러한 경우에 대해서는 분석을 통하여 경험적인 정보를 사용함으로써 구문분석 단계에서 보정기를 만들 수 있다. 다음 절에

서 이들 언어에서의 보정에 따른 문제점을 분석한다.

### 3.4.1 C

C언어는 근원적으로 부당한 입력 문장을 오류나 경고 없이 구문분석한다. 한가지 예로 다음과 같이 선언문으로만 된 문장을 허용한다.

```
int;
```

또한 쓸모 없는 널 문장의 표기를 허용한다.

```
;;;;;;
```

이러한 부가적인 문장들의 허용이 세미콜론의 자동 보정 문제를 어렵게 만들며 경험적인 정보를 적용해야 근본적으로 해결할 수 있다. 예를 들면 구문분석 진행의 우선순위를 부여한다던가 하는 등의 부가 정보를 사용할 수 있다. 좀더 심각한 예로 end-of-line에서 세미콜론이 생략된 다음의 if-else 문장을 고려해 보자.

```
if ( a > 100 )
    acc = acc + a * 200
else
    acc = acc + a
```

위 문장의 경우 첫번째 if 문장에 대하여 end-of-line 에서 세미콜론을 보정하게 된다. 왜냐하면 null 문장이 허용 되기 때문에 우선 보정하게 되고 타당한 구문분석을 계속 진행하고 결국 else 문장을 만났을 때 오류를 감지하게 된다. 이 경우 복구의 문제는 많은 부담을 가진다. 왜냐하면 구문분석 상태 상에서 shift와 reduce를 통해 이미 많은 진행하기 때문이다. 이 문제는 Ada나 Modula-2의 경우는 if문장 구조에 then이라는 키워드를 명시 하도록 문법 구조가 되어 있기 때문에 자연스럽게 해결된다. 앞에서 예시한 널 문장의 경우도 문법에서 근본적으로 허용하지 않아 당연히 해결해 주고 있다.

근본적으로 C 문법의 경우는 세미콜론에 대하여 정의 7을 만족하지 않는다. 또한 경험적인 정보를 사용할 수 있는 방법에 있어서도 위에서 예시한 문제점 이외에도 상당수가 존재하여 세미콜론의 생략에 따른 보정 처리에 따른 비용이 많이 들기 때문에 현실적이지 못하다.

### 3.4.2 Pascal

Pascal의 경우는 C와는 달리 문법적인 구조가 명확하기 때문에 한가지를 제외하고는 문제점이 발생되지 않았다. 그 문제점은 case문장의 레이블에 관한 경우이다. 다음에 주어진 Pascal 문장을 생각해 보자.

```
......
    case choice of
        -1: acc := acc + 256;
        -2: acc := acc - 257;
        -3: ...
......
```

(1) 프로그래머의 의도

```
......
    case choice of
        -1: acc := acc + 256
        -2: acc := acc
         - 257
        -3: ...
......
```

(2) 프로그래머가 표기한 내용

위에서 주어진 Pascal문장 (1)에 대하여 (2)와 같이 표기할 경우 오류가 발견되는 시점이 문제가 된다. 즉 (2)에서 acc :=

acc 다음에 ; 보정이 이루어 지고 -257을 타당한 case 레이블로 인식하여(원래 의도는 - 심볼을 보는 순간 오류를 발생시키고 세미콜론 보정에 의한 오류복구를 하여 acc := acc - 256로 구문분석을 진행해야 한다) 구문분석을 계속 진행한다. 그리고 그 다음 case 레이블인 -3을 보는 시점에서 오류를 감지한다. 이 경우에 있어서 복구에 따른 부담을 가진다. 왜냐하면 이미 정상적인 shift와 reduce동작을 수행했기 때문이다. 이러한 문제에 대하여 Ada의 경우는 case 레이블 앞에 when 이라는 키워드를 명시 하도록 문법구조가 이루어져 있으며 modular-2의 경우 터미널 심볼인 "|"를 명시하게 함으로써 문법적으로 자연스럽게 해결하고 있다.

Pascal의 경우도 근본적으로 세미콜론에 대하여 정의 7을 만족하지 않는다. 그러나 이 경우는 C에 비하여 예외 사항이 적으며 (case문에서만 예외) 경험정보를 사용하여 예외처리를 통해 실제적인 보정이 가능하였다.

## 4. 결 론

어떤 특수 터미널 심볼을 프로그램 작성시 생략할 수 있게 하고 이를 컴파일러가 언어 문법에 위배되지 않도록 자동적으로 보정해 주는 것은 프로그래머에게 융통성 있는 표기법을 제공하여 줌으로써 사소한 오류로부터 해방될 수 있게 해 준다. 이를 목적으로 본 논문에서는 어휘분석과 구문분석 단계에서 생략된 특수 터미널 심볼의 보정 가능 여부에 따라 생략 가능한 심볼을 분류하는 형식론을 제안하였다. 또한 이 형식론에 입각하여 기존의 언어들에 대한 적용 가능성 여부를 조사하고 end-of-line에서 세미콜론을 보정할 수 있는 시스템을 어휘분석 단계와 구문분석 단계에서 동작하도록 만들었으며, 이를 주요 프로그래밍 언어들에 적용하였다. C나 Pascal은 제안된 형식론만 입각해서는 자연스럽게 세미콜론이 생략될 수 없는 언어로 나타났다. 그러나 이들 언어가 가진 문제점을 다른 언어들에서는 구문구조상의 적절한 키워드에 의해서 자연스럽게 해결된 것을 분석된 결과가 보여주었다. 이러한 특성을 고려해 볼 때 특수 터미널 심볼의 생략 가능성에 대한 특성이 언어 설계시에 특수 터미널 심볼과 관계된 구문 구조가 가져야할 키워드 특성의 권고사항이 될 수 있을 것으로 기대된다.

실제로 Ada, SR, Modulo-2, Pascal 등의 언어에 대하여 세미콜론 보정의 예에 적용하여 표 1의 환경 하에서 원시프로그램에 대하여 end-of-line의 세미콜론을 모두 제거하고 수행을 해본 결과 C와 Pascal을 제외한 언어에서는 생략된 세미콜론을 아주 성공적으로 보정해 주었다. 이 결과는 본 논문에서 제안된 방법론들이 실제적으로 의미가 있음을 보여주고 있으며, 또한 이들 프로그램 언어에서 세미콜론 표기에 따른 융통성을 제공하여 세미콜론으로 인한 오류 발생 가능성을 줄일 수 있도록 하여준다.

표 1은 실험에 사용한 각 언어별 문법에 대한 구문분석의 특성을 나타낸다. t상태 수는 전체 구문분석 상태 가운데 생략 가능한 특수 터미널 t를 구문분석할 차례에 있는 상태의 수이며 test한 라인 수는 보정 시스템 검정에 사용된 각 언어별 테스트 목적으로 사용한 설명문장을 제외한 원시 프로그램 라인 수를 의미한다.

| 특성 \ 문법 | Ada | SR | Modulo-2 | C | Pascal |
|---|---|---|---|---|---|
| 문법규칙의 수 | 459 | 349 | 246 | 234 | 196 |
| 터미널의 수 | 97 | 133 | 75 | 86 | 64 |
| 넌터미널 수 | 238 | 153 | 134 | 81 | 97 |
| 상태 수 | 860 | 559 | 387 | 381 | 339 |
| t 상태 수 | 75 | 12 | 28 | 17 | 16 |
| test한 라인수 | 3321 | 5252 | 4089 | 1514 | 3750 |

여기서 t = semicolon(;)

표 1. 실험에 사용된 언어의 특성

## 참고문헌


[1] Aho, A. V., and Ullman, J.D., The theory of Parsing, Translation and Compiling, Vols.1 and 2, Prentice Hall, 1973.

[2] Aho, A. V., and Ullman, J.D., Principles of Compiler Design, Addison-Wesley, 1977.

[3] Choe, K.M., Chang, C.H., "Efficient Computation of the Locally Least-Coast Insertion String for the LR Error Repair," Information Processing Letters, Vol.23, No.6, pp.311~316, December 1986.

[4] Ciesinger, J., "A Bibliography of Error-Handling," ACM SIGPLAN Notices, Vol.14, No.1, pp.16-26, ACM, January 1979.

[5] Conway, R.W., WilCox, T.R., "De-sign and Implememtation of a diagnostic compiler for PL/I," Comm, ACM, Vol.14, pp.169-179, 1973.

[6] DeRemer, F.L., "Simple LR(k) gram-mars," Comm. ACM, Vol.14, pp.453-460, 1971.

[7] Ellis Horowitz, Fundamentals of Programming languages, second edition, Computer Science Press Inc., 1984.

[8] Feyock,S., Lazarus, P. "Syntax-directed correction of syntax errors," Software Practice and Experience 6, pp.207-219, 1976.

[9] Fischer, C.N., Dion B.A., Mauney, J., "A locally Least-coast LR-error Correction and recovery using only insertions," Acta Informatica 13 (3), pp.141-154, 1980.

[10] Fischer, C.N., LeBlanc, R.J., Craft-ing A Compiler with C, The Benjamin/Cummings Pub. Co., Chapter 17, 1991.

[11] Graham, S.L., Rhodes, S.P. "Prac-tical syntactic error recovery," Comm. ACM, Vol.18, pp.639-650, 1975.

[12] Gregory R. Andrews, Concurrent Programming: Principles and Practice, Benjamin/Cummings, 1991.

[13] Irons, E.T., "An error-correcting parse algorithm," Comm. ACM, Vol.7, pp.669-673, 1963.

[14] Knuth, D.E., "On the translation of languages from left to right," Inf. Control, 8 , pp. 607-639, 1965.

[15] Lee, M.J. and Choe, K.M., "SLR(k) covering for LR(k) gram-

mars," Information Processing Letters, 37, pp.337-347, 1991.

[16] Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E. Compiler Design Theory, Reading Mass, Addison-Wesley 1973.

[17] Lyon, G. "Syntax-directed least-errors analysis for context-free languages: a practical approach," Comm. ACM, Vol.17, pp.3-14, 1974.

[18] Park, J.C., Choe, K.M., and Chang, C.H., "A new analysis of LALR formalisms," ACM Trans. Program. Lang. Syst., 7, pp. 159-175, 1985.

[19] Ralph E. Griswold., "String Scanning in the Icon Programming Language," The Computer Journal, Vol.33 No.2, April 1990.

[20] 이명준, "LR(k)-Colored 문법과 그 응용에 관한 연구," 박사학위논문, 전산학과, 한국과학기술원, 1991.