5. Context Free Grammar and Pushdown Automata

목차 및 학습 목표

- **목차**
  - 문맥 자유문법
  - 파스 트리
  - 모호한 문법
  - 문법 변환
  - 푸쉬다운 오토마타

5.1 문맥 자유문법

- 잘 설계된 문법은 소스 프로그램을 정확한 목적 코드로 번역할 때 유용한 구조를 제공한다. 문맥자유 문법은 프로그래밍 언어의 설계에서뿐만 아니라 효율적인 컴파일러 작성 시에도 중요하게 사용된다

(non-T) (non-T, T)

- **$A \rightarrow \beta$, 단 $A \in V_N$, $\beta \in V^*$**
  - A가 β 로 치환 : A는 β로 유도(derivation). A는 β 를 생성.
  - β 가 A 로 치환 : β 는 A로 감축(reduce)

5.2 파스 트리

- **[정의 5-1] 좌단유도(leftmost derivation), 우단유도(rightmost derivation)**
  - 좌단 유도는 유도 과정의 각 단계에서 문장형태(sentential form)의 가장 왼쪽에 있는 논터미널 기호를 계속해서 대체(replacement)하는 경우로 $\underset{lm}{\Rightarrow}$ 으로 표시
  - 우단 유도는 유도 과정의 각 단계에서 문장형태의 가장 오른쪽에 있는 논터미널 기호를 계속해서 대체하는 경우로 $\underset{rm}{\Rightarrow}$ 으로 표시

  - **[예제 5-2]** 좌단 유도와 우단 유도 1
  - 다음 문법에 대해 좌단 유도와 우단 유도를 통해 문장 id + (id * id)를 유도해보자.

P : E → E + T | E - T | T

T → T * F | T / F | F

F → (E) | id

[풀이]

① 좌단 유도

$E \underset{lm}{\Rightarrow} E + T \underset{lm}{\Rightarrow} T + T \underset{lm}{\Rightarrow} F + T \underset{lm}{\Rightarrow} id + T \underset{lm}{\Rightarrow} id + F \underset{lm}{\Rightarrow} id + (E) \underset{lm}{\Rightarrow} id + (T) \underset{lm}{\Rightarrow} id + (T * F) \underset{lm}{\Rightarrow} id + (F * F) \underset{lm}{\Rightarrow} id + (id * F) \underset{lm}{\Rightarrow} id + (id * id)$

② 우단 유도

$E \underset{rm}{\Rightarrow} E + T \underset{rm}{\Rightarrow} E + F \underset{rm}{\Rightarrow} E + (E) \underset{rm}{\Rightarrow} E + (T) \underset{rm}{\Rightarrow} E + (T * F) \underset{rm}{\Rightarrow} E + (T * id) \underset{rm}{\Rightarrow} E + (F * id) \underset{rm}{\Rightarrow} E + (id * id) \underset{rm}{\Rightarrow} T + (id * id) \underset{rm}{\Rightarrow} F + (id * id) \underset{rm}{\Rightarrow} id + (id * id)$

5.2 **파스 트리**

- **[정의 5-2]** 좌파스, 우파스
  - 좌 파스(left parse)
    - 하나의 문장을 만들 때 좌단 유도에 의해서 적용된 일련의 생성규칙 순서
  - 우 파스(right parse)
    - 우단유도에 의해서 적용된 생성규칙 순서의 역순
- **[예제 5-4]** 좌파스와 우파스
  - 다음과 같은 생성 규칙을 가진 문법에 대해 좌파스와 우파스를 구해보자.

1 $E \rightarrow E + T$	2 $E \rightarrow E - T$	3 $E \rightarrow T$
4 $T \rightarrow T * F$	5 $T \rightarrow T / F$	6 $T \rightarrow F$
7 $F \rightarrow (E)$	8 $F \rightarrow id$	

  - [풀이]

① 좌파스는 좌단 유도를 할 때 적용된 생성 규칙 순서이므로 좌단 유도를 한다.

- $E \underset{1}{\Rightarrow} E + T \underset{3}{\Rightarrow} T + T \underset{6}{\Rightarrow} F + T \underset{8}{\Rightarrow} id + T \underset{6}{\Rightarrow} id + F \underset{7}{\Rightarrow} id + (E) \underset{3}{\Rightarrow} id + (T) \underset{4}{\Rightarrow} id + (T * F) \underset{6}{\Rightarrow} id + (F * F) \underset{8}{\Rightarrow} id + (id * F) \underset{8}{\Rightarrow} id + (id * id)$

∴ 좌파스는 1, 3, 6, 8, 6, 7, 3, 4, 6, 8, 8이다.

② 우파스는 우단 유도를 할 때 적용된 생성 규칙 순서의 역순이므로 우단 유도를 한다.

$E \underset{1}{\Rightarrow} E + T \underset{6}{\Rightarrow} E + F \underset{7}{\Rightarrow} E + (E) \underset{3}{\Rightarrow} E + (T) \underset{4}{\Rightarrow} E + (T * F) \underset{8}{\Rightarrow} E + (T * id) \underset{6}{\Rightarrow} E + (F * id) \underset{8}{\Rightarrow} E + (id * id) \underset{3}{\Rightarrow} T + (id * id) \underset{6}{\Rightarrow} F + (id * id) \underset{8}{\Rightarrow} id + (id * id)$

∴ 우파스는 8, 6, 3, 8, 6, 8, 4, 3, 7, 6, 1이다.

5.2 파스 트리

- **구문 분석 방법은 크게 두 종류가 존재**
  - 하향식(top-down) 구문분석
    - 좌 파스를 생성
  - 상향식(bottom-up) 구문분석
    - 우 파스를 생성

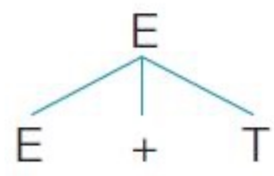
- **[정의 5-3] 문맥 자유문법 G = (V_N, V_T, P, S)에 대한 파스 트리(parse tree) 정의**
  1) 모든 노드의 이름은 문법기호이다.
  2) 루트(root) 노드의 이름은 시작기호 S이다.
  3) 만약 어떤 노드가 하나 이상의 자식(child)을 갖는다면 이 노드의 이름은 논터미널 기호이다.
  4) 왼쪽부터 순서적으로 X_1, X_2, ⋯, X_n의 n개의 자식을 갖는 어떤 노드 A가 존재한다면 생성규칙 $A \rightarrow X_1\ X_2 \cdots X_n$ 가 존재한다.
  5) 만약, 어떤 노드가 자식을 하나도 가지고 있지 않다면, 이 노드를 터미널 노드(terminal node) 혹은 잎(leaf)이라 하고 터미널 노드의 이름은 터미널 기호이다.

5.2 파스 트리

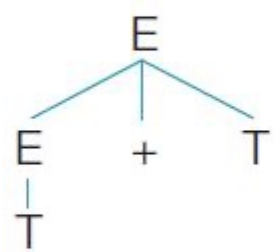
- **[예제 5-6] 파스 트리 만들기 1**
    - [예제 5-2]의 문장 id + (id * id)를 유도하는 파스 트리를 만들어보자.
    - [풀이]

① 좌단 유도로 파스 트리를 만드는 과정은 다음과 같다.

1단계 : $E \underset{lm}{\Rightarrow} E + T$에 대한 파스 트리

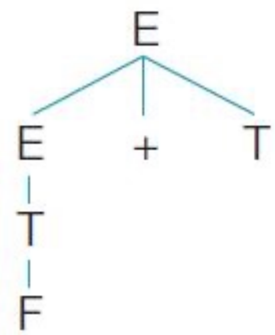


2단계 : $E + T \underset{lm}{\Rightarrow} T + T$에 대한 파스 트리

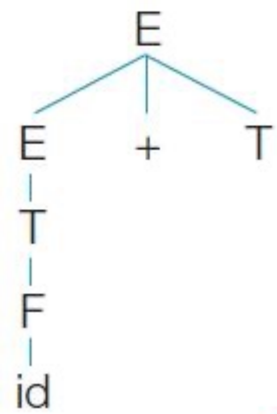


5.2 파스 트리

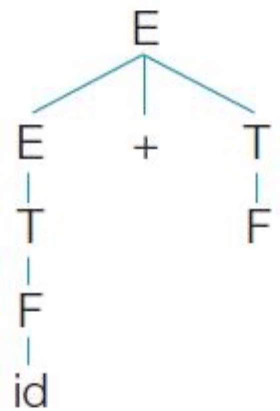
3단계 : $T + T \underset{lm}{\Rightarrow} F + T$에 대한 파스 트리



4단계 : $F + T \underset{lm}{\Rightarrow} id + T$에 대한 파스 트리

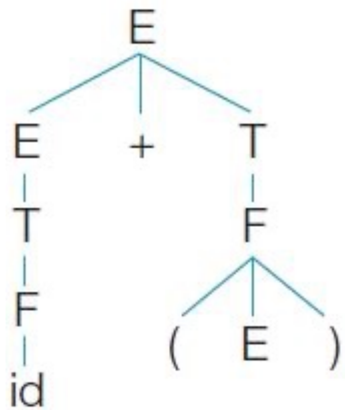


5단계 : $id + T \underset{lm}{\Rightarrow} id + F$에 대한 파스 트리

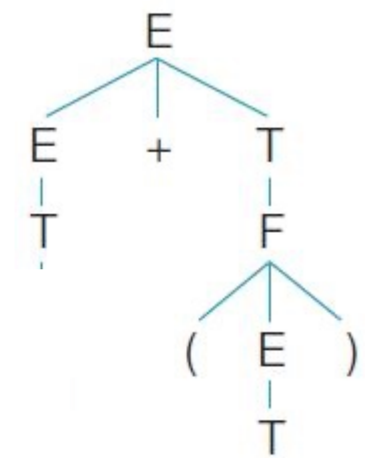


5.2 파스 트리

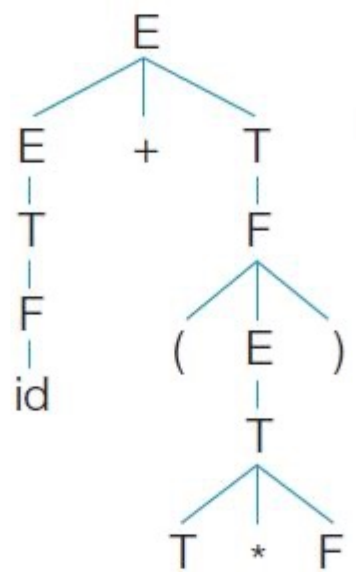
6단계 : id + F $\underset{lm}{\Rightarrow}$ id + (E)에 대한 파스 트리



7단계 : id + (E) $\underset{lm}{\Rightarrow}$ id + (T)에 대한 파스 트리

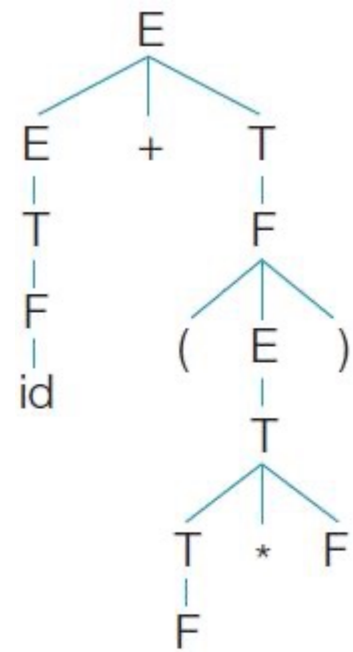


8단계 : id + (T) $\underset{lm}{\Rightarrow}$ id + (T * F)에 대한 파스 트리

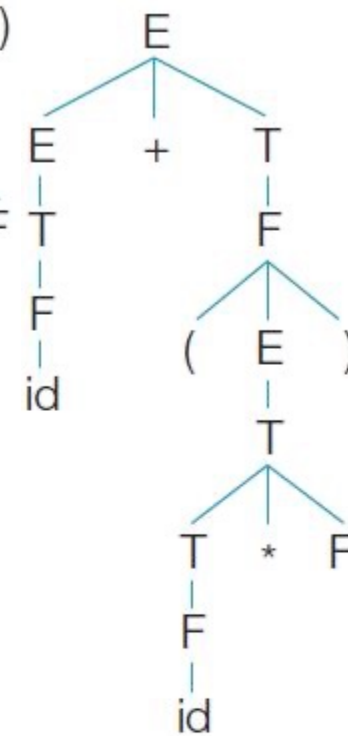


5.2 파스 트리

9단계 : $\text{id} + (T * F) \underset{lm}{\Rightarrow} \text{id} + (F * F)$에 대한 파스 트리

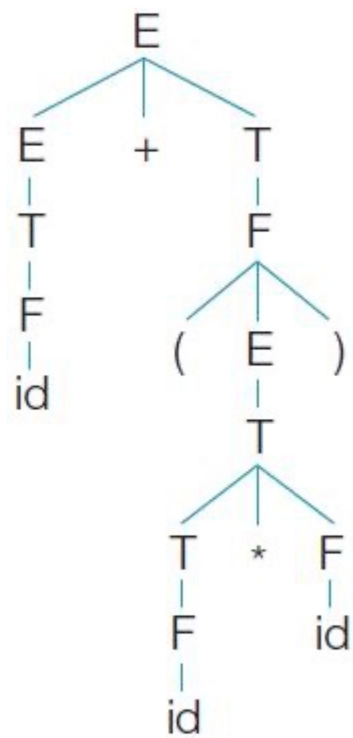


10단계 : $\text{id} + (F * F) \underset{lm}{\Rightarrow} \text{id} + (\text{id} * F)$에 대한 파스 트리



5.2 파스 트리

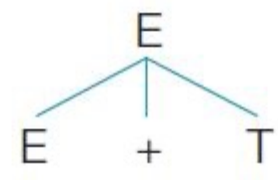
11단계 : id + (id * F) $\underset{lm}{\Rightarrow}$ id + (id * id)에 대한 파스 트리



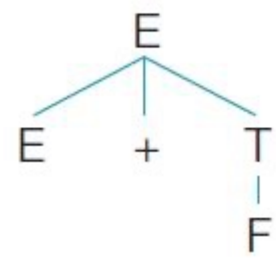
5.2 파스 트리

② 우단 유도로 파스 트리를 만드는 과정은 다음과 같다.

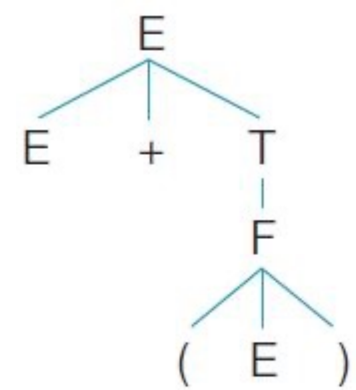
1단계 : $E \underset{rm}{\Rightarrow} E + T$에 대한 파스 트리



2단계 : $E + T \underset{rm}{\Rightarrow} E + F$에 대한 파스 트리



3단계 : $E + F \underset{rm}{\Rightarrow} E + (E)$에 대한 파스 트리

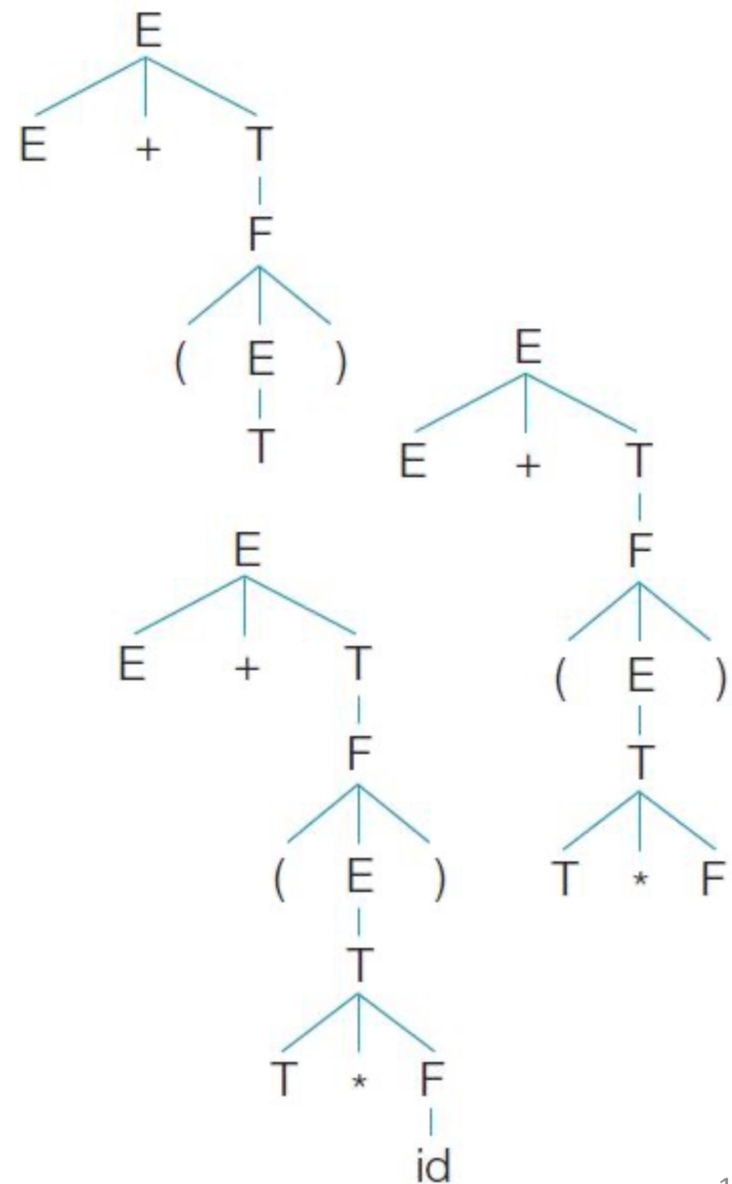


5.2 파스 트리

4단계 : $E + (E) \underset{rm}{\Rightarrow} E + (T)$에 대한 파스 트리

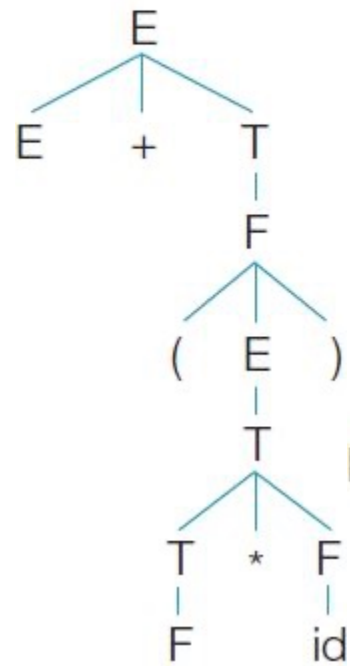
5단계 : $E + (T) \underset{rm}{\Rightarrow} E + (T * F)$에 대한 파스 트리

6단계 : $E + (T * F) \underset{rm}{\Rightarrow} E + (T * id)$에 대한 파스 트리

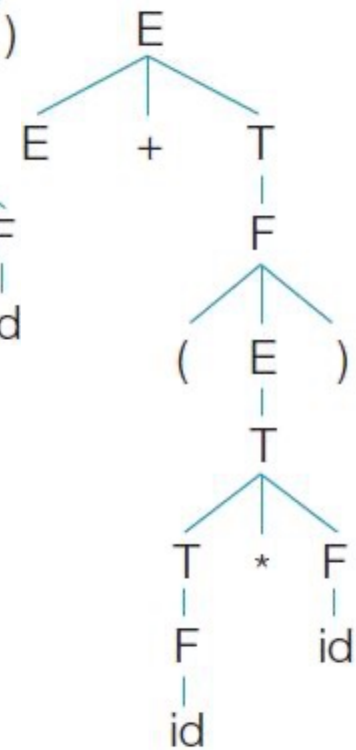


5.2 파스 트리

7단계 : E + (T * id) $\underset{rm}{\Rightarrow}$ E + (F * id)에 대한 파스 트리

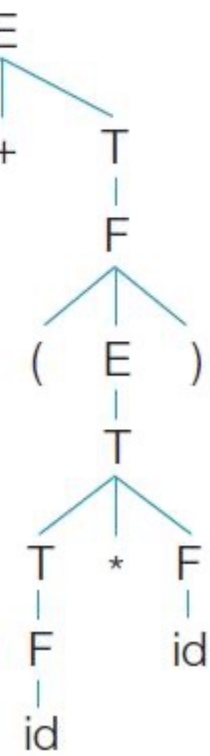


8단계 : E + (F * id) $\underset{rm}{\Rightarrow}$ E + (id * id)에 대한 파스 트리

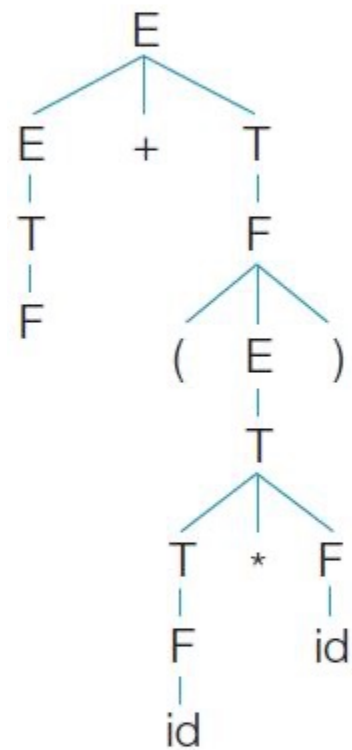


5.2 파스 트리

9단계 : E + (id * id) $\underset{rm}{\Rightarrow}$ T + (id * id)에 대한 파스 트리

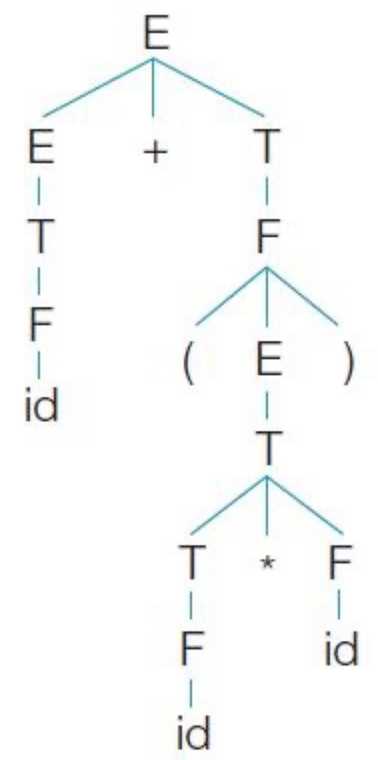


10단계 : T + (id * id) $\underset{rm}{\Rightarrow}$ F + (id * id)에 대한 파스 트리



5.2 파스 트리

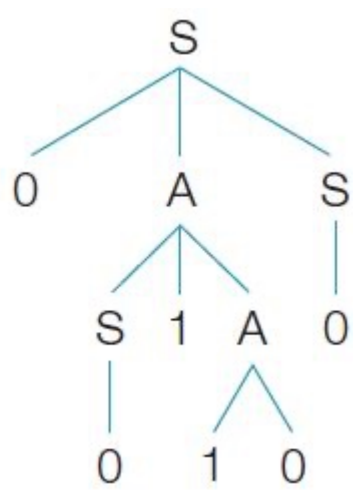
11단계 : F + (id * id) $\underset{rm}{\Rightarrow}$ id + (id * id)에 대한 파스 트리



- 이처럼 문장 id + (id * id)에 대한 파스 트리는 좌단 유도와 우단 유도 모두 같은 결과를 얻는다

5.2 파스 트리

- **[예제 5-7] 파스 트리 만들기 2**
  - [예제 5-3]과 같은 경우에 문장 001100을 유도하는 파스 트리를 만들어보자.
  - [풀이]
    - 좌단 유도와 우단 유도로 생성된 파스 트리는 다음과 같이 2개가 똑같다.



    - [예제 5-6]과 [예제 5-7]에서 파스 트리는 좌단 유도 혹은 우단 유도를 해도 똑같은 결과가 나왔다. 그러나 어떤 문법에서는 좌단 유도나 우단 유도를 할 때 생성 규칙을 어떤 순서로 적용하느냐에 따라 서로 다른 파스 트리가 만들어지는 경우도 있다. 이런 경우에 문법이 모호하다고 한다.

5.3 모호한 문법

- **[정의 5-4] 모호한 문법**
  - 하나의 문장이 서로 다른 두 개 이상의 파스 트리를 갖는다면 문법 *G*는 **모호하다** (ambiguous)고 한다.

  - **[예제 5-8]** 좌단 유도로 파스 트리 만들기 1
    - 다음 문법에서 문장 3 + 4 * 5에 대해 좌단 유도로 파스 트리를 만들어보자.

G = ({E}, { + , - , * , /, (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, P, E)

P : E → E + E | E - E | E * E | E/E | (E) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

[풀이] 여기서 문장 3 + 4 * 5를 생성하는데 좌단 유도를 해보자.

$E \Rightarrow E + E$	$E \Rightarrow E * E$
$\Rightarrow 3 + E$	$\Rightarrow E + E * E$
$\Rightarrow 3 + E * E$	$\Rightarrow 3 + E * E$
$\Rightarrow 3 + 4 * E$	$\Rightarrow 3 + 4 * E$
$\Rightarrow 3 + 4 * 5$	$\Rightarrow 3 + 4 * 5$

두 가지의 서로 다른 유도과정이 생긴다.

이에 대응하는 파스 트리를 만들어 보면 다음과 같이 두 개의 서로 다른 파스 트리가 생기므로 애매한 문법이다.

5.3 모호한 문법

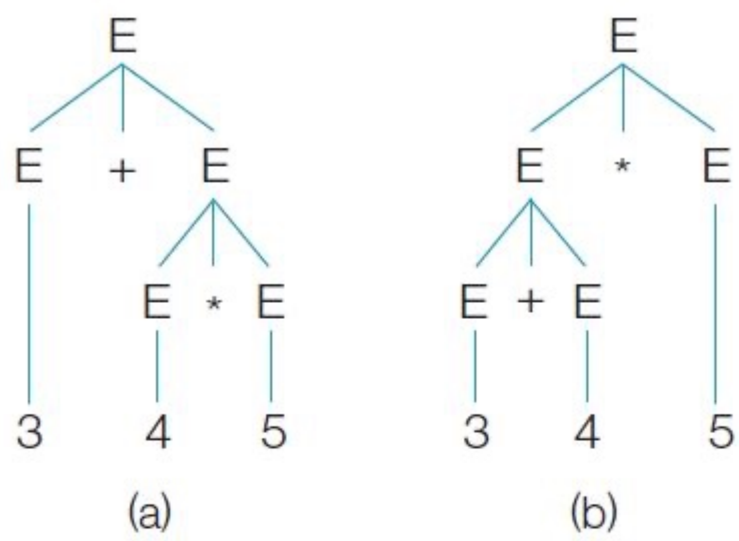


그림 5-1 **문장 3+4*5에 대한 파스 트리**

- [예제 5-8]의 문법은 하나의 문장 3 + 4 * 5에 대해 서로 다른 2개의 파스 트리가 생기므로 모호한 문법이다. 이처럼 모호한 문법은 하나의 문장에 대해 서로 다른 의미를 가진다

5.3 모호한 문법

- **[예제 5-8]은 모호한 문법**
  - 주어진 문법은 하나의 문장에 대해서 서로 다른 두 개의 파스 트리가 생기기 때문
  - 모호한 문법을 가지고 처리하기 위한 방법은 두 가지 방법이 사용
    1. 모호한 문법을 모호하지 않은 문법으로 변환시킨 후 모호하지 않은 문법을 가지고 구문 분석기를 만들어 처리하는 방법
    2. 모호한 문법을 가지고 구문 분석기를 만든 다음에 충돌이 발생한 구문 분석기에서 충돌을 없애는 방법
- **모호한 문법을 모호하지 않은 동등한 문법으로 변환**
  - 모호한 모든 문법을 동등한 모호하지 않은 문법으로 변환할 수 있는 것은 아님
  - 산술식의 경우 연산자의 우선 순위(precedence)와 결합 법칙(associativity)을 이용하여 모호하지 않은 문법으로 변환
  - 문장 3 + 4 * 5를 계산하는데
    - [그림 5-8](a)와 같이 계산하면 * 를 +보다 연산자의 우선 순위를 높게 준 경우로서(왜냐하면 * 연산자와 +연산자가 있을 때 *연산을 +연산보다 먼저 한다는 뜻) 계산된 결과 값이 23
    - [그림 5-8](b)와 같이 계산하면 +를 * 보다 연산자의 우선 순위를 높게 준 경우로서 계산된 결과 값이 35

5.3 모호한 문법

- 연산자 우선순위가 *와 /는 +, -보다 높으며 *와 /는 서로 같고 +와 -도 서로 같다고 하자. 그렇다면 문장 3 + 4 * 5에 대한 연산 순서는 [그림 5-1(a)]와 같은 파스 트리가 생성되는 3 + (4 * 5)와 같이 된다. 그런데 연산자 우선순위만으로는 모든 모호한 문법을 모호하지 않은 문법으로 변환할 수 없다. 다음 예제를 한번 살펴보자.
- **[예제 5-9]** 좌단 유도로 파스 트리 만들기 2
  - 주어진 문법은 하나의 문장에 대해서 서로 다른 두 개의 파스 트리가 생기기 때문
  - [예제 5-8]의 문법에서 문장 3 - 4 - 5에 대해 좌단 유도로 파스 트리를 만들어보자
    - 모호한 문법을 모호하지 않은 문법으로 변환시킨 후 모호하지 않은 문법을 가지고 구문 분석기를 만들어 처리하는 방법먼저 다음과 같이 유도할 수 있다.

$$
\begin{aligned}
E &\Rightarrow E - E \\
&\Rightarrow E - E - E \\
&\Rightarrow 3 - E - E \\
&\Rightarrow 3 - 4 - E \\
&\Rightarrow 3 - 4 - 5
\end{aligned}
$$

    - 이에 대한 파스 트리는 [그림 5-2(a)]와 같다.

5.3 모호한 문법

- 같은 문장을 다른 방법으로 유도하면 다음과 같다.

E ⇒ E - E

⇒ 3 - E

⇒ 3 - E - E

⇒ 3 - 4 - E

⇒ 3 - 4 - 5

- 이에 대한 파스 트리는 [그림 5-2(b)]와 같다.

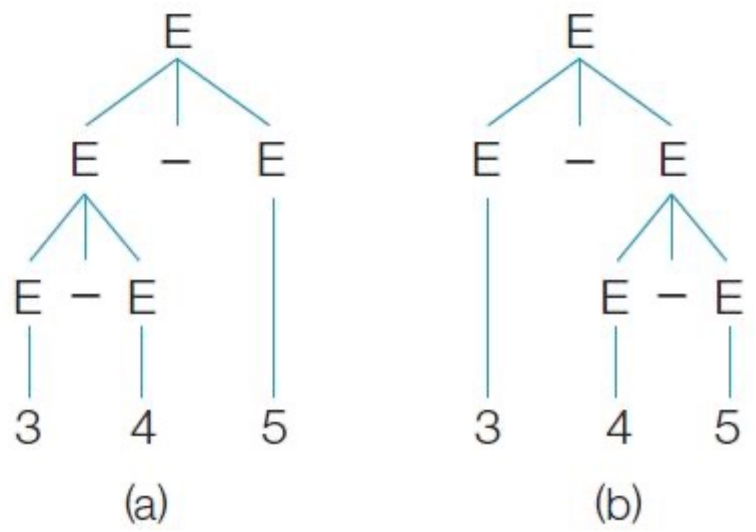


그림 5-2 3−4−5에 대한 파스 트리

5.3 모호한 문법

- 이때도 서로 다른 결과 값 2개가 생긴다. [그림 5-2(a)]의 경우에는 파스 트리를 보면 3 - 4 를 먼저 계산하므로 결과 값이 -6이다. 그러나 [그림 5-2(b)]의 경우에는 4 - 5를 먼저 계 산하므로 결과 값이 4이다. 여기서 파스 트리를 보면 -4 - 5를 계산하지 않고 4 - 5를 먼저 계산한다. 왜냐하면 -4의 -는 이항 연산자이기 때문이다.
- 이처럼 연산자의 우선순위가 같은 경우에 어느 것을 먼저 계산하는지 해결하는 방법에 결합 법칙이 사용된다. 결합 법칙은 연산자의 우선순위가 같을 때 가장 왼쪽에 있는 연산자부터 오른쪽에 있는 연산자로 계산할 것인지, 아니면 가장 오른쪽에 있는 연산자부터 왼쪽에 있는 연산자로 계산할 것인지에 대한 법칙이다. 이때 가장 왼쪽의 연산자부터 오른쪽 연산자로 계 산하는 것을 좌측 결합(left associative)이라 하고, 가장 오른쪽의 연산자부터 왼쪽 연산자로 계산 하는 것을 우측 결합(right associative)이라 한다. 대부분의 프로그래밍 언어에서는 좌측 결합 방식 을 취한다.

5.3 모호한 문법

- 이제 [예제 5-8]의 모호한 문법에 대해 연산자 우선순위를 *, / > +, -로 하고 모든 연산자 들이 좌측 결합 법칙을 취하면 [그림 5-3]과 같이 모호하지 않은 문법으로 변환할 수 있다.

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

그림 5-3 **모호하지 않은 문법**

  - [예제 5-8]의 문법과 [그림 5-3]의 문법이 동등함을 보여야 하는데, 이는 두 문법에서 생성되는 언어가 같아지는 것을 보이면 된다. 이 증명은 생략한다.

- [그림 5-3]의 문법으로 문장 3 + 4 * 5와 3 - 4 - 5에 대해 유일한 파스 트리가 만들어진다면 모호하지 않은 문법으로 변환되었음을 알 수 있다. [그림 5-4]는 문장 3 + 4 * 5와 3 - 4 - 5에 대해 만들어진 유일한 파스 트리를 보여준다

5.3 모호한 문법

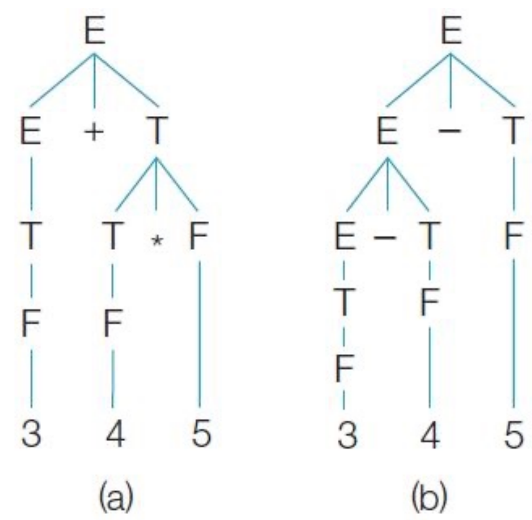


그림 5-4 **모호하지 않은 파스 트리**

  - [예제 5-8]의 문법과 [그림 5-3]의 문법이 동등함을 보여야 하는데, 이는 두 문법에서 생성되는 언어가 같아지는 것을 보이면 된다. 이 증명은 생략한다.

- [그림 5-3]의 문법으로 문장 3 + 4 * 5와 3 - 4 - 5에 대해 유일한 파스 트리가 만들어진다면 모호하지 않은 문법으로 변환되었음을 알 수 있다. [그림 5-4]는 문장 3 + 4 * 5와 3 - 4 - 5에 대해 만들어진 유일한 파스 트리를 보여준다
- 이제 모호한 문법을 모호하지 않은 문법으로 어떻게 변환하는지 알아보자

5.3 모호한 문법

- 모호한 문법을 모호하지 않은 문법으로 변환
  1. [예제 5-8]의 문법에서 가장 기초적인 피연산자를 F라 하면 이는 괄호로 묶인 산술식이나 0, 1, 2, 3, 4, 5, 6, 7, 8, 9가 될 수 있다.

     F → (E) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

     - 이 생성 규칙이 생성 규칙 중에서 가장 아래에 온다.
  2. 다음으로는 연산자를 순위가 높은 것부터 취한다.
     - *와 /의 우선순위가 가장 높으므로 다음 생성 규칙과 같이 재귀적으로 구성한다.

     T → T * F | T / F | F

     - *와 / 연산자가 좌측 결합 법칙을 취하므로 위와 같고 우측 결합 법칙을 취할 경우

     T → F * T | F / T | F와
  3. 같은 방법으로 다음 우선 순위가 +와 -이므로 이것 또한 다음과 같이 재귀적으로 생성 규칙을 만듦

     E → E + T | E - T | T

5.3 모호한 문법

- 이 생성 규칙들을 모아보면 [그림 5-3]과 똑같은 문법이 된다.

E → E + T | E - T | T

T → T * F | T / F | F

F → (E) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

5.3 모호한 문법

- **[예제 5-10] 모호한 문법을 모호하지 않은 문법으로 변환**
  - 연산자 우선순위는 단 항 연산자 > 거듭제곱 > *, / > +, -이며, 결합 법칙의 경우 거듭제곱은 우측 결합 법칙을 취하고 *, /, +, -는 좌측 결합 법칙을 취한다.
  - P : E → E + E | E - E | E * E | E / E | E ^ E | -E | (E) | id
- [풀이] E → E + T | E - T | T

  T → T * F | T / F | F

  F → P ^ F | P

  P → -P | H

  H → (E) | id

5.3 모호한 문법

- 모호한 문법 중 현수 else(dangling else)에 대해 살펴보자. 현수 else는 중첩된 if 문에서 else가 어떤 if 문에 걸리느냐 하는 것이다
- [예제 5-12] 현수 else

  다음과 같은 문법을 고려해보자.

  stat → if expr then stat

  | if expr then stat else stat

  | other

  위 문법으로부터 아래 문장이 모호한 문법임을 보여라.

  if E_1 then if E_2 then S_1 else S_2

  그리고 else 문이 가까운 if 문에 걸리도록 모호하지 않은 문법으로 변환해보자
- [풀이] 이 문장은 다음과 같은 두 가지 의미로 해석될 수 있으며, 괄호를 이용하여 이를 나타내면 다음과 같다.

① if E1 then (if E2 then S1 else S2)

② if E1 then (if E2 then S1) else S2

①은 if~then 문장이고 ②는 if~then~else 문장이다. ①번 문장과 ②번 문장에 대한 파스 트리는 [그림 5-5]와 같다.

5.3 모호한 문법

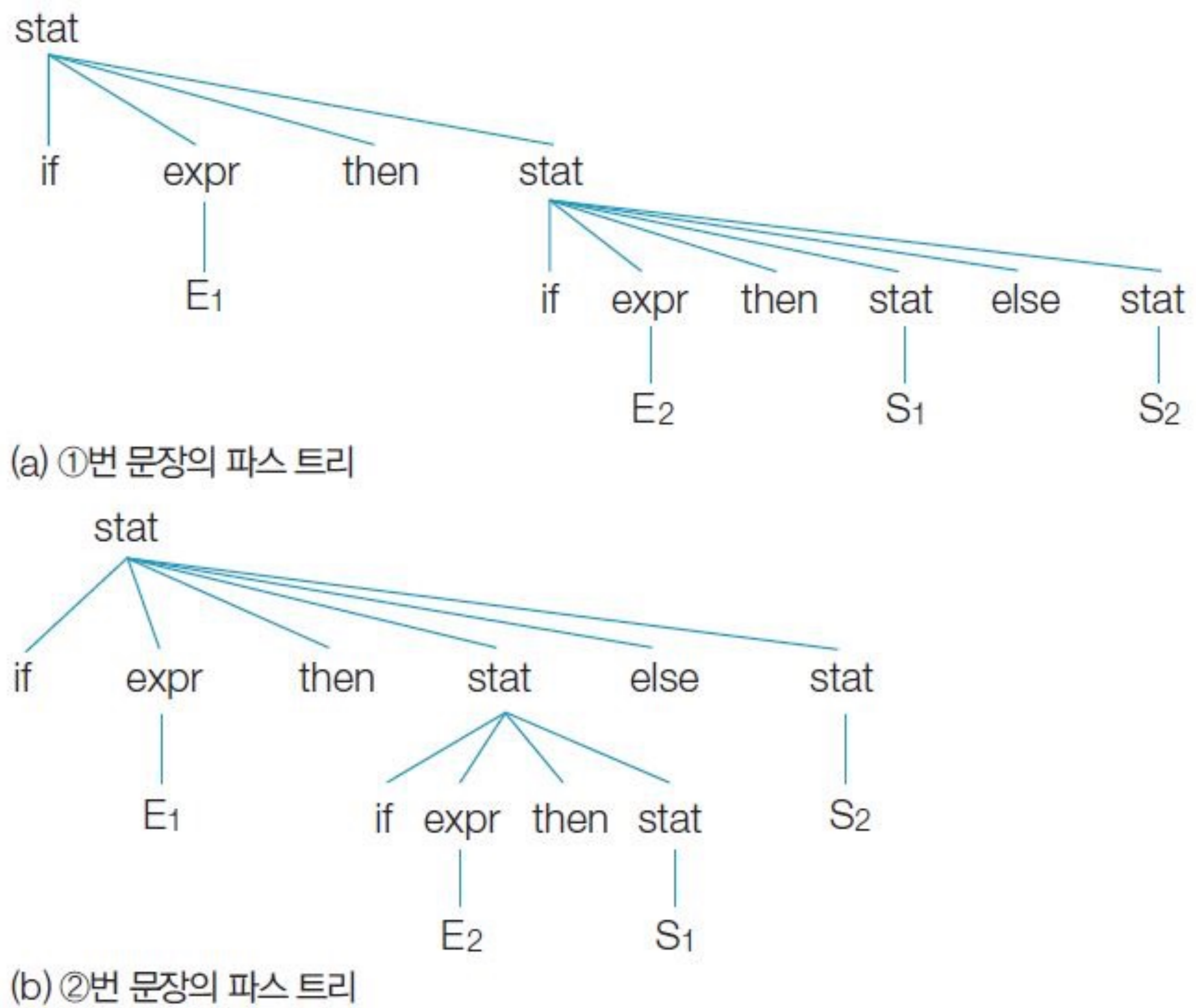


그림 5-5 ①번 문장과 ②번 문장의 파스 트리

5.3 모호한 문법

- 이와 같이 하나의 문장에 대해 서로 다른 2개의 파스 트리가 생성되므로 이 문법은 모호한 문법 이다. 현수 else도 모호한 문법인데, 현수 else를 모호하지 않은 문법으로 변환하는 방법 중 [그림 5-5(a)]와 같이 else를 그 앞에 있는 가장 가까운 if와 연결하는 것으로 일반적인 프로그래밍 언어 에서 사용하는 방법이다.
- 다음과 같은 모호하지 않은 문법으로 변환할 수 있다.

stat → matched

　　| unmatched

matched → if expr then matched else matched

　　　　| other

unmatched → if expr then stat

　　　　　| if expr then matched else unmatched

5.4 문법 변환

- **문법 변환**
  - 모호한 문법 이외에도 어떤 문법들은 구문분석을 하는데 있어서 효율을 상당히 떨어뜨리는 경우에 효율적인 구문분석이 이루어지도록 주어진 문법을 적당한 문법으로 바꾸어 주는 것
  - 변환 방법
    - 불필요한 생성규칙의 제거
    - ε -생성규칙의 제거
    - 단일 생성규칙의 제거
    - 좌인수분해(left-factoring)
    - 좌재귀(left-recursion)의 제거 등
- **불필요한 생성 규칙의 제거 (elimination of useless production)**
  - 한 문법이 생성하는 언어는 시작 기호로부터 유도할 수 있으며 모두 터미널 기호로 이루어진 문장이다. 따라서 시작 기호로 부터 도달 불가능하거나 터미널 기호들을 생성하지 못하는 기호들은 가지고 있는 생성 규칙들을 모두 제거하는 것
  - 불필요한 생성 규칙이란 불필요한 기호를 갖고 있는 생성 규칙을 말함

5.4 문법 변환

- **[정의 5-5] 불필요한 기호**
  - CFG G = (V_N, V_T, P, S)가 S $\overset{*}{\Rightarrow}$ $\alpha X \beta$ $\overset{*}{\Rightarrow}$ w, 단 w ∈ V_T^* 와 같은 유도과정이 존재하지 않는다면, 기호 X는 불필요한 기호

- **[정의 5-6] 터미널 문자열을 생성하는 논터미널 기호, 시작 기호로 부터 도달 가능한 기호**
  - 생성 규칙 A→ α에 대해서, α $\overset{*}{\Rightarrow}$ w이고 w ∈ V_T^* 일 때 A를 **터미널 문자열을 생성하는 논터미널 기호**
  - 생성 규칙 A→ α에 대해서, S $\overset{*}{\Rightarrow}$ uXw, u, w ∈ V^* 일 때 X를 **시작 기호로부터 도달 가능한 기호**

- **[알고리즘 5-1] 터미널 문자열을 생성할 수 없는 논터미널 기호를 가진 불필요한 생성규칙의 제거**

[입력] CFG G = (V_N, V_T, P, S)

단, V_N : 논터미널 기호들의 집합 V_T : 터미널 기호들의 집합

P : 생성 규칙들의 집합 S : 시작기호

[출력] CFG G = (V_N', V_T, P', S)

단, V_N' : 터미널 문자열을 생성하는 논터미널 기호들의 유한집합

V_T : 터미널 기호들의 집합 S : 시작기호

P' : 터미널 문자열을 생성할 수 없는 논터미널 기호를 가진 불필요한 생성규칙을 제거한 생성규칙의 집합

[방법] begin

$V_N' = \{A \mid A \rightarrow w \in P, w \in V_T^*\}$;

repeat

$V_N' = V_N' \cup \{A \mid A \rightarrow \alpha, \alpha \in (V_N' \cup V_T)^*\}$

until no change;

$V_N'' = V_N - V_N'$;

$P' = P - \{B \rightarrow \gamma\beta\gamma' \mid \gamma, \gamma' \in (V_N \cup V_T)^*$, 모든 B, $\beta \in V_N''\}$

end;

5.4 문법 변환

- **[알고리즘 5-1] 설명**
  - 생성 규칙의 오른쪽이 모두 터미널 기호만으로 이루어진 생성 규칙의 왼쪽 부분은 당연히 터미널 문자열을 유도하는 논터미널 기호
  - 터미널 문자열을 유도하는 논터미널 기호와 터미널 기호로 생성 규칙의 오른쪽이 이루어져 있다면 생성 규칙 왼쪽에 있는 논터미널 기호도 터미널 기호를 생성하는 논터미널 기호
  - 더 이상 새로운 논터미널이 생성되지 않을 때까지 반복해서 구한 논터미널 기호들의 집합을 구하면 이 논터미널 기호들은 터미널 기호들을 생성하는 기호
  - 주어진 생성 규칙에서 모든 논터미널 기호들에서 터미널 기호들을 생성하는 논터미널 기호들을 제외하면 주어진 논터미널 기호들의 집합은 터미널 기호들을 생성하지 못하는 논터미널 기호가 됨
  - 불필요한 기호들을 갖고 있는 생성 규칙은 불필요한 생성 규칙이 되므로 원래의 생성 규칙에서 이런 생성 규칙들을 제거

- **[예제 5-13] 터미널 문자열을 생성하지 못하는 논터미널을 가진 불필요한 생성 규칙 제거**
  - 다음 문법에 대해 터미널 문자열을 생성할 수 없는 논터미널 기호를 가진 불필요한 생성 규칙을 제거해보자.
  - 문맥 자유문법 $G = (V_N, V_T, P, S)$

    단, $V_N = \{S, A, B\}$ $V_T = \{a\}$

    $P : S \rightarrow AB \mid a$

    $A \rightarrow a$

    $S = S$

[풀이] 생성 규칙 $P : S \rightarrow a$, $A \rightarrow a$에 의하여 $V_N' = \{S, A\}$이다.

더 이상 새로운 논터미널 기호가 추가되지 않으므로

$V_N' = V_N - V_N' = \{S, A, B\} - \{S, A\} = \{B\}$

그러므로 B가 불필요한 기호가 되고 생성 규칙 중에 B를 포함하는 생성 규칙 $S \rightarrow AB$는 불필요한 생성 규칙이다.

이 생성 규칙을 제거하면

$P' = \{S \rightarrow a, A \rightarrow a\}$ 가 된다.

5.4 문법 변환

- **[알고리즘 5-2] 시작 기호로 부터 도달 불가능한 기호를 가진 불필요한 생성규칙의 제거**

[입력] CFG G = (V_N, V_T, P, S)

단, V_N : 논터미널 기호들의 집합 V_T : 터미널 기호들의 집합

P : 생성 규칙들의 집합 S : 시작기호

[출력] CFG G' = (V_N', V_T', P', S)

단, V_N' : 시작기호로 부터 도달 가능한 논터미널 기호들의 유한집합

V_T' : 시작기호로 부터 도달 가능한 터미널 기호들의 유한집합

P' : 시작기호로 부터 도달 불가능한 기호를 제거한 생성규칙의 집합

S : 시작기호

[방법] begin

V' = {S};

repeat

V' = V' ∪ {X | if A ∈ V' , then A → $\alpha X \beta$ ∈ P}

until no change;

V' = V - V';

P' = P - {B → $\gamma \beta \gamma'$ | $\gamma, \gamma' \in (V_N \cup V_T)^*$, B ∈ V', β ∈ V'};

$V_N' = V_N \cap V'$;

$V_T' = V_T \cap V'$

end;

- **[예제 5-14] 시작 기호로 부터 도달 불가능한 기호를 가진 불필요한 생성 규칙 제거**
  - [예제 5-13]의 문법에서 시작 기호로부터 도달 불가능한 기호를 가진 생성 규칙을 제거해 보자.
  - 문맥 자유문법 $G = (V_N, V_T, P, S)$

    단, $V_N = \{S, A, B\}$ $V_T = \{a\}$

    $P : S \rightarrow AB \mid a$

    $A \rightarrow a$

    $S = S$

[풀이] $V' = \{S\}$

$V' = \{S, A, B, a\}$

더 이상 변화가 없으므로

$V' = V - V' = \{S, A, B, a\} - \{S, A, B, a\} = \phi$

$\therefore P' = \{S \rightarrow AB \mid a, A \rightarrow a\}$이다.

5.4 문법 변환

- 이제 [알고리즘 5-1], [알고리즘 5-2]에 의해 불필요한 생성 규칙을 모두 제거한다. 그런 데 주어진 문법에 [알고리즘 5-1]을 먼저 적용하고 [알고리즘 5-2]를 나중에 적용하는 방법 과 [알고리즘 5-2]를 먼저 적용하고 [알고리즘 5-1]을 나중에 적용하는 방법이 있다. [예제 5-13]의 문법에 이 두 가지 방법을 모두 적용해보자.
- ① [알고리즘 5-1], [알고리즘 5-2]를 순서적으로 적용하는 경우
  - [예제 5-13]에 의해 [알고리즘 5-1]을 먼저 적용하면 다음과 같다.

$P' = \{S \rightarrow a, A \rightarrow a\}$

P'에 대해 [알고리즘 5-2]를 적용한다.

$V' = \{S\}$

$V' = \{S, a\}$

더 이상 변화가 없으므로

$V' = V - V' = \{S, A, a\} - \{S, a\} = \{A\}$

$\therefore P' = \{S \rightarrow a, A \rightarrow a\} - \{A \rightarrow a\}$

$= \{S \rightarrow a\}$

5.4 문법 변환

- ② [알고리즘 5-2], [알고리즘 5-1]을 순서적으로 적용하는 경우
  - [예제 5-14]에 의해 P′ = {S → AB | a, A → a}이다. P′에 대해 [알고리즘 5-1]을 적용 하면 [예제 5-13]과 같으므로 다음과 같이 된다.

P′ = {S → a, A → a}

- ①과 ②를 비교해보면 ②의 방법은 실제적으로 시작 기호로부터 도달 불가능한 기호를 모두 제거하지 못했다. 결국 불필요한 생성 규칙의 제거는 ①의 방법으로 해야 한다. 다시 말해 터미널 문자열을 생성할 수 없는 논터미널 기호를 가진 불필요한 생성 규칙을 제거한 다음, 시작 기호로부터 도달 불가능한 기호를 가진 생성 규칙을 제거해야만 불필요한 생성 규칙이 완전히 제거된다.

5.4 문법 변환

- **ε-생성 규칙의 제거**
  - ε-생성 규칙(ε-production rule)은 $A \rightarrow \varepsilon$ 형태의 생성 규칙을 가진 것을 말한다
- **[정의 5-7] ε-free 문법**

  CFG $G = (V_N, V_T, P, S)$ 가 다음의 조건 중 한가지 조건만을 만족할 경우

  1. P 가 ε-생성규칙을 갖지 않는다.
  2. 시작기호 S만이 $S \rightarrow \varepsilon$인 ε-생성규칙을 가질 경우, 다른 생성 규칙의 오른쪽에 S 가 나타나지 않는다.

5.4 문법 변환

- **[알고리즘 5-3] ε-생성 규칙의 제거**
  - ε-생성 규칙(ε-production rule)은 A → ε 형태의 생성 규칙을 가진 것을 말한다

[입력] 문맥자유문법 $G = (V_N, V_T, P, S)$

[출력] ε-free 문맥자유문법 $G' = (V_N', V_T, P', S')$

[방법] begin

$P' = P - \{A \to \varepsilon \mid A \in V_N\};$

$V_{N\varepsilon} = \{A \mid A \Rightarrow \varepsilon, A \in V_N\};$

for $A \to \alpha_0 B_1 \alpha_1 B_2 \cdots B_k \alpha_{k+1} \in P'$, where $\alpha_i \neq \varepsilon$ and $B_i \in V_{N\varepsilon}$ do

begin

만약 어떠한 $\alpha_j (0 \le j \le k)$도 $V_{N\varepsilon}$에 포함되어 있지 않다면 $A \to \alpha_0 X_1 \alpha_1 X_2 \cdots X_k \alpha_k + 1$에 대하여 $X_i = \varepsilon$ 혹은 $X_i = B_i$에 의해서 생성 할 수 있는 모든 생성규칙을 P'에 첨가한다.

그렇지 않은 경우,

A → ε 인 생성규칙을 P'에 첨가

end;

if $S \in V_{N\varepsilon}$ then begin $P' = P' \cup \{S' \to \varepsilon \mid S$, S'는 새로운 시작기호$\};$

$V_N' = V_N \cup \{S'\}$

end;

else begin $V_N' = V_N;$

$S' = S$

end;

end.

5.4 문법 변환

- **[알고리즘 5-3]의 의미 :** ε-생성규칙이 존재하면 이들을 제거하고 제거된 ε-생성규칙 들에 의해서 생성될 수 있는 것들을 보상해주는 방법

- **[예제 5-16]** ε-free 문법으로 변환
  - 문법 G = ({S}, {a, b}, P, S)를 ε-free 문법으로 변환해보자.

P : S → aSbS | bSaS | ε

[풀이] ε-생성규칙의 제거 방법을 적용하면

P' = {S → aSbS | bSaS} 가 되고 $V_N\varepsilon$ = {S}가 된다.

다음으로 P'에 속하는 생성규칙의 오른쪽 부분 중 논터미널 S가 있는 경우에 대해서 S 대신에 ε을 대체해서 생성될 수 있는 모든 생성규칙을 P'에 첨가하므로 P'는

S → aSbS | bSaS | aSb | abS | ab | bSa | baS | ba

또한 시작기호 S가 $V_N\varepsilon$에 속하므로 S' → S | ε을 P'에 첨가하므로 P'는

S' → S | ε

S → aSbS | bSaS | aSb | abS | ab | bSa | baS | ba

5.4 문법 변환

$\therefore G' = (V_N', V_T, P', S')$

단, $V_N' = \{S, S'\}$

$P' : S' \rightarrow S \mid \varepsilon$

$S \rightarrow aSbS \mid bSaS \mid aSb \mid abS \mid ab \mid bSa \mid baS \mid ba$

$S' = S'$

5.4 문법 변환

- **단일 생성 규칙(single production rule)의 제거** → non-Terminal 이 늘어나는 규칙. 줄이기.
  - 단일 생성 규칙이란 생성 규칙 중 A → B와 같이 생성 규칙의 오른쪽이 단 하나의 논터미널 기호로만 구성된 생성 규칙이 존재하는 경우를 말한다.

- **[알고리즘 5-4] 단일 생성 규칙의 제거**

[입력] ε-free CFG $G = (V_N, V_T, P, S)$

[출력] 단일 생성규칙을 갖지 않는 G와 동등한 ε-free 문맥자유문법 $G' = (V_N, V_T, P', S)$

[방법] begin

$\quad P' = P - \{A \rightarrow B \mid A, B \in V_N\};$

$\quad$ for each $A \in V_N$ do

$\quad\quad V_{NA} = \{B \mid A \overset{+}{\Rightarrow} B\};$

$\quad\quad$ repeat

$\quad\quad\quad V_{NA} = V_{NA} \cup \{C \mid B \rightarrow C \in P, B \in V_{NA}\}$

$\quad\quad$ until no change;

$\quad$ end for;

$\quad$ for each $B \in V_{NA}$ do

$\quad\quad$ for each $B \rightarrow \alpha \in P'$ do

$\quad\quad\quad P' := P' \cup \{A \rightarrow \alpha\}$

$\quad\quad$ end for

$\quad$ end for;

end.

- **[예 5-18] 단일 생성 규칙 제거 하기**
  - ε-free 문맥자유문법 G = (V_N, V_T, P, S)

단, P : E → E + T | T

T → T * F | F

F → (E) | a

[풀이] 단일 생성규칙의 제거방법에 의해

P' = {E → E + T, T → T * F, F → (E), F → a}

각각의 non-terminal E, T, F에 대해서 다음을 반복한다.

먼저 E에 대해서 V_{NE} = {T, F}가 되고

T → T * F, F → (E), F → a 가 존재하므로

P' = {E → E + T | T * F | (E)| a, T → T * F, F → (E), F → a}

다음으로 T 와 F(V_{NT} = {F} 와 V_{NF} = ϕ)에 대해서도 같은 방법에 의하면

E→ E + T | T * F | (E) | a

T → T * F | (E) | a

F → (E) | a

결국 G' = (V_N, V_T, P', S) 에서

P' : E → E + T | T * F | (E) | a

T → T * F | (E) | a

F → (E) | a

5.4 **문법 변환**

- **[정의 5-8] proper한 문법**
  - 문맥자유 문법 $G = (V_N, V_T, P, S)$가 어떤 $A \in V_N$에 대하여 $A \Rightarrow A$ 형태의 유도과정을 갖지 않을 때 cycle-free 하다고 한다. 만약 G가 cycle-free 하고, ε-free, 그리고 불필요한 생성 규칙을 가지지 않으면 그 문법은 proper 하다고 한다.

  - **[예제 5-20]** proper 문법 확인
    - 다음 문법이 proper 한지 확인해보자.

P' : E → E + T | T * F | (E) | a

T → T * F | (E) | a

F → (E) | a

[풀이] ε-free이고 불필요한 생성 규칙이 없으며 cycle-free이므로 이 문법은 proper 하다.

5.4 문법 변환

- **좌인수분해(left - factoring)**
  - 좌인수분해가 필요한 이유
    - 같은 기호들을 접두사로 갖는 2개 이상의 생성규칙이 있을 때, 주어진 문자열이 올바른 문장인가를 검사하기 위하여, 하향식 구문 분석기는 어떤 생성규칙을 적용해야 할지를 결정할 수 없다.
    - 만약 하나의 생성 규칙을 적용했다가 주어진 문자열이 생성되지 않으면, 다시 돌아와서 다른 생성 규칙을 적용할 수밖에 없다.
    - 이런 경우는 하향식 구문분석 방법의 전형적인 단점인 백트래킹이 나타남.
    - 이런 경우 구문 분석기는 다음 기호를 볼 때까지 그 결정을 연기함으로써 구문 분석을 효율적으로 할 수 있다.
- **[정의 5-9] 좌인수분해**
  - 같은 기호를 접두사로 가진 2개 이상의 생성 규칙이 존재할 때 공통된 접두사를 인수분해하는 것을 좌인수분해라고 한다.

5.4 문법 변환

- **[알고리즘 5-5] 좌인수분해**

  [입력] 문법 G

  [출력] 동일한 left-factoring된 문법

  [방법] begin

  repeat

  Find the production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$; (* γ 는 문자열 *)

  Find the longest prefix α,

  If $\alpha \neq \varepsilon$ then

  replace all the $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$

  by $A \rightarrow \alpha A' \mid \gamma$

  $A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ (* A'는 새로운 논터미널 *)

  until 공통된 접두사를 갖는 논터미널이 존재하지 않음

  end.

5.4 문법 변환

- **[예제 5-21] 좌인수분해 하기 1**
  - 다음 문법을 좌인수분해 해보자.

S → cAd

A → a | ab

  - [풀이] 좌인수분해하기 전에 백트래킹이 일어나는 경우를 살펴보자.

문장이 cabd인 경우, 하향식 구문 분석에서

1) c를 읽고 생성 규칙에 의해 c를 유도할 수 있는 생성 규칙 S → cAd에 의해서 cAd를 유도.

2) 문장에서 a를 읽는다.

3) 문장 형태 cAd에서 논터미널 A를 유도해야 되는데, A에 의해서 유도되는 것은 둘 다 a로 시작하기 때문에 어떤 생성 규칙을 적용해야 하는지를 결정할 수가 없다. 그래서 생성규칙 A → a를 먼저 취하면 문장 형태는 cad,

4) 다음으로 입력 문장을 읽으면 d가 되므로 주어진 문장이 아니라는 것을 알고, 다시 생성 규칙 A → ab를 취해서 문장을 유도

  - 백트래킹이 생기는 이유
    - 생성규칙 A → a | ab 에서 접두사가 **똑같이 a**이기 때문
    - 공통된 접두사 a를 인수분해 함으로써 이 문제를 해결 ➔ 좌인수분해

5.4 문법 변환

- 공통된 접두사 a를 [알고리즘 5-5]에 의해 인수분해하면 된다.

$S \rightarrow cAd$

$A \rightarrow aA'$

$A' \rightarrow \varepsilon \mid b$

5.4 문법 변환

- **좌재귀(left - recursive)**
  - 문법이 어떤 문자열 α에 대해 A ⇒ Aα의 유도과정이 존재하는 경우
  - 하향식 구문 분석시에 같은 생성 규칙이 반복적으로 적용
    - 무한 루프에 빠지게 되므로 구문 분석을 어렵게 한다.
      - 그래서 좌재귀를 제거해야 함
- **[예제 5-23]** 다음 문법에서 id + id * id를 좌단 유도로 생성해보자.

  E → E + T | T

  T → T * F | F

  F → (E) | id

  ➔ E ⇒ E + T ⇒ E + T + T ⇒ E + T + T ⇒ ...

  첫 번째 터미널 기호를 생성해야 하는데 터미널 기호를 생성하지 못하고 무한 루프에 빠짐
- **[정의 5-10] 직접 좌재귀와 간접 좌재귀**
  - 직접 좌재귀는 생성 규칙이 A → Aα의 형태이며, 간접 좌재귀는 A ⇒ Aα의 유도 과정이 존재하는 경우이다
  - 좌재귀를 제거하는 방법
    - 무한 루프에 빠지기 쉬운 좌재귀 대신에 A ⇒ αA의 형태와 같은 우재귀(right - recursion)로 변환

5.4 문법 변환

- **좌재귀를 우재귀로 변환하는 방법**
  - 직접 좌재귀가 있는 논터미널 기호의 일반적인 생성 규칙 형태

$A \rightarrow A\alpha \mid \beta \iff A = A\alpha + \beta = \beta\alpha^*$

α^* 를 생성하는 문법은 A' 도입

$A' \rightarrow \alpha A' \mid \varepsilon$

$\beta\alpha^*$ 를 생성하는 문법

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$\beta\alpha^* = \beta\alpha^*\alpha + \beta$

$= \beta(\alpha^+ + \varepsilon)$

$= \beta(\alpha^*)$

5.4 문법 변환

- **[알고리즘 5-6] 직접 좌재귀의 제거**

[입력] $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ 과 같은 좌재귀를 갖는 문법 $G = (V_N, V_T, P, S)$

[출력] 좌재귀가 제거된 문법 $G' = (\{V_N \cup \{A'\}\}, V_T, P', S)$

[방법] begin

repeat

Find $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$; / * β_i는 A로 시작하는 것이 하나도 없다 * /

replace $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

by $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon$

until not exist 좌재귀

end.

5.4 **문법 변환**

- **[예제 5-23]** 다음 문법에서 좌재귀를 제거.

  E → E + T | T

  T → T * F | F

  F → (E) | id

  [풀이]

  논터미널 E에 대해서 좌재귀를 제거하면

  E → TE'

  E' → + TE' | ε

  다시 논터미널 T에 대해서 좌재귀를 제거하면

  T → FT'

  T' → * FT' | ε

  논터미널 F에 대해서는 좌재귀가 존재하지 않으므로

  좌재구가 제거된 문법

  E → TE'

  E' → + TE' | ε

  T → FT'

  T' → * FT' | ε

  F → (E) | id

5.4 문법 변환

- **[알고리즘 5-7] 간접 좌재귀의 제거**

  [입력] 간접 좌재귀가 존재하는 문법

  [출력] 좌재귀가 제거된 동등한 문법

  [방법] begin

  생성규칙의 논터미널을 $A_1, A_2, \cdots, A_n$의 순서로 정돈 ;

  for i : = 2 to n do

  for j : = 1 to i - 1 do

  각 $A_i \rightarrow A_j\gamma$ 형태의 생성규칙을 $A_i \rightarrow \alpha_1\gamma | \alpha_2\gamma | \cdots | \alpha_k\gamma$로 대체한다;

  /* $A_j \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_k$는 모두 A_j의 생성규칙 */

  end for

  A_i-생성규칙으로부터 직접 직접 좌재귀 제거 ;

  end for

  end.

5.4 문법 변환

- [예제 5-24] 다음 문법에서 좌재귀를 제거

S → Aa | b

A → Ac | Sd | e

[풀이]

생성 규칙 S → Aa에 A → Sd를 적용하면 S → Sda가 되므로 간접 좌재귀가 존재

이를 제거하기 위해서는 간접 좌재귀의 제거 방법에 의해서 먼저 논터미널 S, A를 순서적으로 정돈

S가 A보다 정돈된 순서가 앞서 있으므로, 생성규칙 A → Sd에 S 생성 규칙을 모두 대입하면

A → Ac| Aad | bd | e

그러면 A에 대한 직접 좌재귀가 나타나므로 이것을 제거

A → bdA'| eA‘

A' → cA'| adA'| ε

그러므로 좌재귀를 제거된 최종 생성규칙

S → Aa| b

A → bdA'| eA‘

A' → cA'| adA'| ε

5.5 푸시다운 오토마타

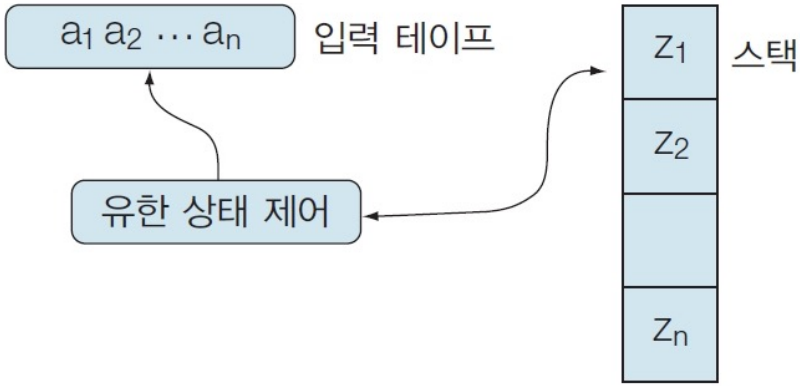


그림 5-6 푸시다운 오토마타의 구성

- **[정의 5-11] 푸시다운 오토마타는 7개의 요소로 구성**

$M = (Q, \Sigma, T, \delta, q_0, z_0, F)$

여기서 Q : 상태들의 유한집합

Σ : 입력 기호들의 유한집합

T : 스택 기호들의 유한집합

$q_0 \in Q$: 시작상태(start state)

$z_0 \in T$: 스택의 시작기호

$F \subseteq Q$: 종결상태(final state)의 집합

δ : 사상함수 $Q \times (\Sigma \cup \{\varepsilon\}) \times T \rightarrow Q \times T^*$

5.5 푸시다운 오토마타

- $\delta(q, a, z) = \{(p, r)\}$이라 하면
  - q의 상태에서 입력기호 a를 받았을 때 스택의 톱(top)에 z가 있다면 상태는 p로 이전
  - 스택의 톱인 z가 스택에서 삭제(pop)되고 스택에는 r이 삽입(push)되는 것
  - 만약 r이 ε인 경우는 스택에서 삭제만이 이루어진다.
  - 일반적인 사상함수 $\delta(q, a, z) = \{(p_1, r_1), (p_2, r_2), \cdots, (p_m, r_m)\}$이라 하면
    - 상태는 $p_i(i = 1, \cdots, m)$로 전이가 되고 스택의 톱인 z가 삭제되고 스택에는 r_i가 삽입
    - 이때 입력 제어는 한자리 오른쪽으로 이동
    - $m = 1$이면 결정적 푸시다운 오토마타(deterministic push-down automata)
    - $m \geq 2$이면 비결정적 푸시다운 오토마타(non-deterministic push-down automata)

5.5 푸시다운 오토마타

- **유한 오토마타의 입력 문자열 인식하는 방법**
  - 입력 문자열을 모두 읽은 후, 마지막 상태가 종결 상태냐 아니냐에 따라서 결정
- **푸시다운 오토마타에서 입력 문자열을 인식하는 방법**
  - FA의 경우처럼 종결 상태에 의해서 인식하는 방법
    - $L(M) = \{w \mid (q_0, w, z_0) \Rightarrow (p, \varepsilon, r), P \in F, r \in T^*\}$
  - 비어있는(empty) 스택에 의해 인식하는 방법
    - $N(M) = \{w \mid (q_0, w, z_0) \Rightarrow (p, \varepsilon, \varepsilon), P \in Q\}$
    - 푸시다운 오토마타는 입력 문자열을 모두 읽은 후에도 이동이 일어남
    - 스택이 비어있는 경우에는 이동이 일어나지 않음

5.5 푸시다운 오토마타

- **[예제 5-25] 푸시다운 오토마타에 의한 인식 1**
  - 언어 $L = \{0^n1^n \mid n \geq 1\}$을 인식하는 PDA M과 같을 때 0011과 0101이 인식되는지?

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{z, 0\}, \delta, q_0, z, \{q_0\})$$

$$\begin{aligned}\delta : \delta(q_0, 0, z) &= \{(q_1, 0z)\}\\ \delta(q_1, 0, 0) &= \{(q_1, 00)\}\\ \delta(q_1, 1, 0) &= \{(q_2, \varepsilon)\}\\ \delta(q_2, 1, 0) &= \{(q_2, \varepsilon)\}\\ \delta(q_2, \varepsilon, z) &= \{(q_0, \varepsilon)\}\end{aligned}$$

  - [풀이] 입력문자열 0011이 인식되는 과정을 살펴보자.

$$\begin{aligned}\rightarrow (q_0, 0, z) &\Rightarrow (q_1, 0, 0z)\\ &\Rightarrow (q_1, 1, 00z)\\ &\Rightarrow (q_2, 1, 0z)\\ &\Rightarrow (q_2, \varepsilon, z)\\ &\Rightarrow (q_0, \varepsilon, \varepsilon)\end{aligned}$$

위의 PDA M은 비어있는 스택에 의해서 인식되는 DPDA 이고 q_0가 종결상태에 속하므로 종결상태에 의해서 인식되는 DPDA.

➔ 다음으로 문자열 0101이 인식되는지 살펴보자.

$\delta(q0, 0, z) \Rightarrow \delta(q1, 1, 0z)$

$\Rightarrow \delta(q2, 0, z)$

더 이상 진행되지 못하므로 0101은 인식되지 않는다.

- **푸시다운 오토마타의 확장 :** 전이 함수 $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times T \rightarrow Q \times T^*$ 에서 $\delta : Q \times (\Sigma \cup \{\varepsilon\})^* \times T^* \rightarrow Q \times T^*$ 로 확장이 가능하다.

- **[예제 5-27]** 푸시다운 오토마타에 의한 인식 3

[예제 5-25]에 주어진 PDA M에 의해 입력 문자열 0011이 인식되는 과정을 확장된 PDA M에 의해 설명해보자.

$\delta(q0, 0011, z) \Rightarrow \delta(q1, 011, 0z)$

$\Rightarrow \delta(q1, 11, 00z)$

$\Rightarrow \delta(q2, 1, 0z)$

$\Rightarrow \delta(q2, \varepsilon, z)$

$\Rightarrow \delta(q2, \varepsilon, \varepsilon)$