KNU 4471.043 컴파일러 설계

고상기

5주차

2022 Spring

강원대학교 컴퓨터공학과

4주차 요약

- 어휘 분석lexical analysis
- 토큰token 인식하기
- 어휘 분석기 설계하기
- 어휘 분석기 구현하기

5주차 개요

- 문맥-자유 문법context-free grammar
- 파스 트리
- 모호한 문법ambiguous grammar
- 문법 변환하기

문맥-자유 문법context-free grammar

- 잘 설계된 문법은 소스 프로그램을 정확한 목적 코드로 번역할 때 유용한 구조를 제공한다.
- 문맥-자유 문법은 프로그래밍 언어를 설계하거나 컴파일러를 구현할 때 중요한 이론적 기반을 제공한다.

- 문맥-자유 문법의 생성 규칙 $A \to \beta$에 대해,
  - A를 β로 치환하는 과정을 A를 β로 유도$_{\text{derivation}}$한다고 말한다.
  - 반대로 β를 A로 치환하는 과정을 β를 A로 감축$_{\text{reduce}}$한다고 말한다.

왼쪽 유도와 오른쪽 유도

Definition

왼쪽 유도$_{\text{leftmost derivation}}$는 유도 과정의 각 단계에서 문장 형태의 가장 왼쪽에 있는 논터미널 기호를 계속해서 대체하는 경우이며 $\Rightarrow_{lm}$로 표시한다.

반대로, 오른쪽 유도$_{\text{rightmost derivation}}$는 가장 오른쪽에 있는 논터미널 기호를 계속해서 대체하는 경우이며 $\Rightarrow_{rm}$로 표시한다.

- 다음 문법에 대해 각각 왼쪽 유도와 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도해보자.

$$E \to E + T \mid E - T \mid T$$

$$T \to T * F \mid T / F \mid F$$

$$F \to (E) \mid id$$

파스 트리

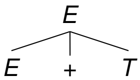
Definition

문맥-자유 문법 $G = (V_N, V_T, P, S)$에 대해, 파스 트리는 다음과 같이 정의된다.

- 모든 노드의 이름은 문법 기호이다.
- 루트 노드의 이름은 시작 기호 S이다.
- 만약 어떤 노드가 하나 이상의 자식을 가지고 있다면 이 노드의 이름은 논터미널 기호이다.
- 왼쪽부터 순서대로 $X_1, X_2, \ldots, X_n$의 n개 자식을 가진 어떤 노드 A가 존재한다면 생성 규칙 $A \rightarrow X_1 X_2 \cdots X_n \in P$이 성립한다.
- 만약 어떤 노드가 자식을 하나도 가지고 있지 않다면 이 노드를 단말 노드$_{\text{terminal node}}$ 또는 잎 노드$_{\text{leaf node}}$라 하고 노드의 이름은 터미널 기호이다.

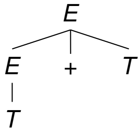
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



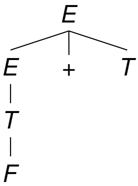
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



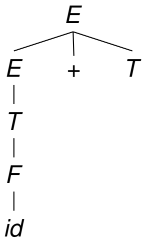
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



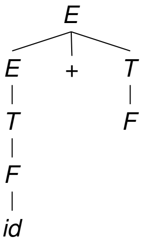
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



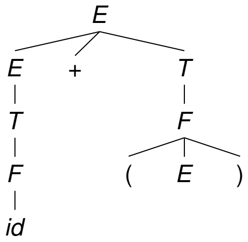
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



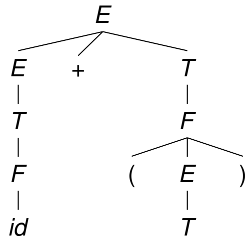
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



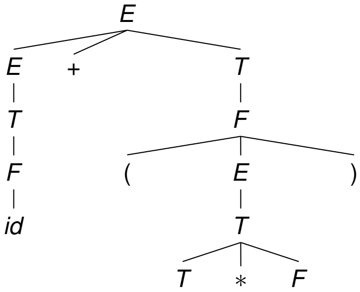
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



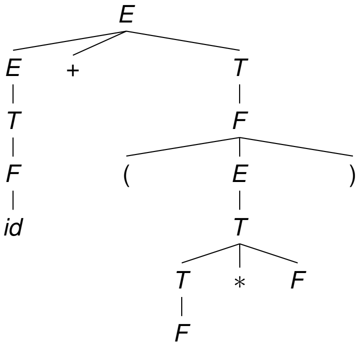
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



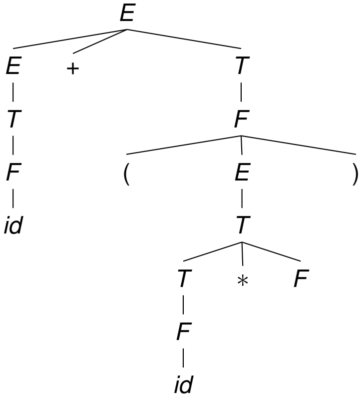
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



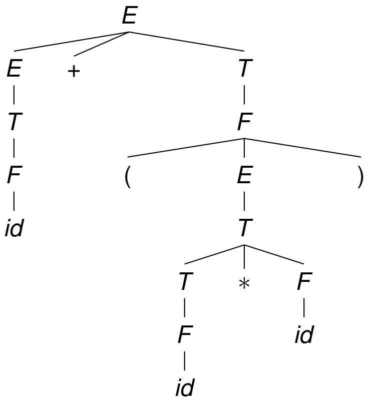
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



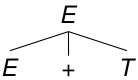
파스 트리 예

이전 슬라이드에서 왼쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



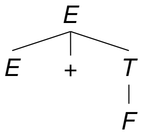
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



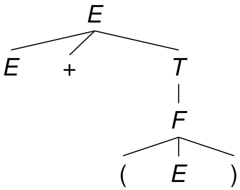
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



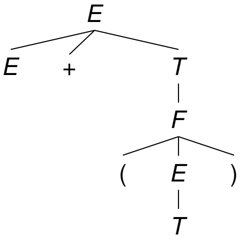
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



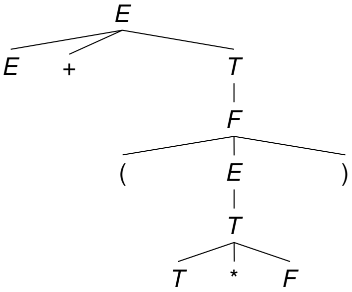
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



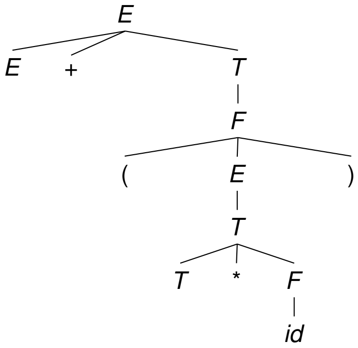
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



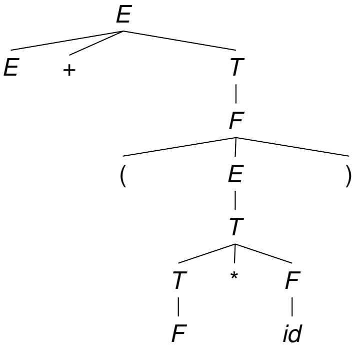
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



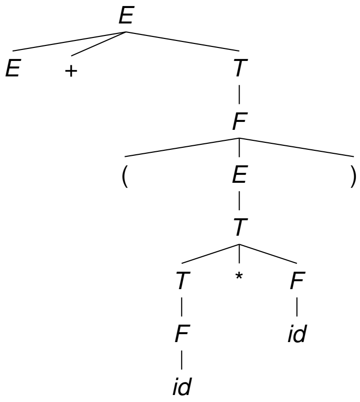
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



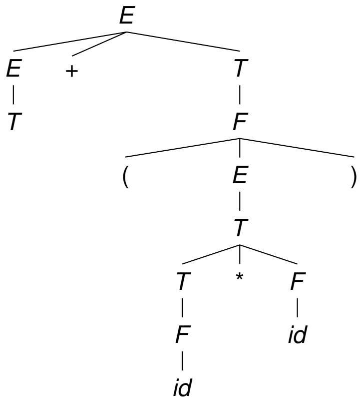
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



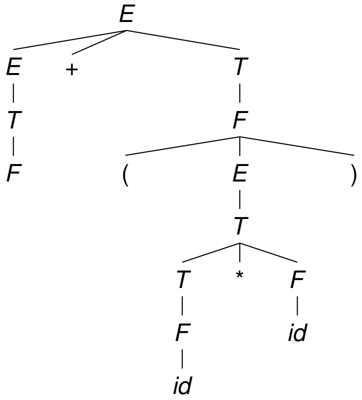
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



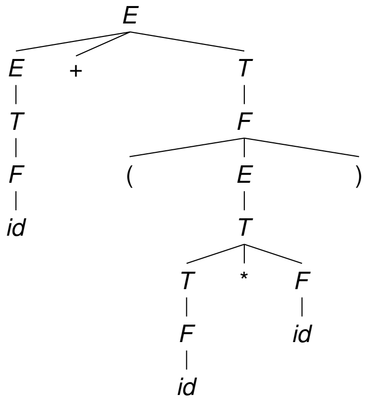
파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



파스 트리 예

이번엔 오른쪽 유도를 통해 문장 $id + (id * id)$를 유도한 과정을 파스 트리로 나타내보자.



모호한 문법$_{\text{ambiguous grammar}}$

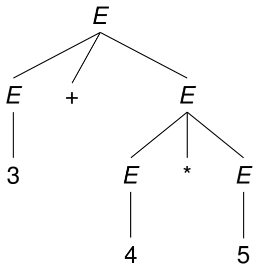
Definition

하나의 문장에 대해 서로 다른 2개 이상의 왼쪽 유도를 생성하는 문법 G를 모호하다$_{\text{ambiguous}}$고 한다.

아래의 문법으로부터 문장 $3 + 4 * 5$를 왼쪽 유도해보자.

$$E \to E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid 0 \mid 1 \mid 2 \mid \cdots \mid 9$$

모호한 문법ambiguous grammar 예



문법의 모호성 판단

- 문맥-자유 문법의 모호성을 판단하는 알고리즘은 존재하지 않는다$_{\text{undecidable}}$[1].
- 단지 몇 가지 예를 통해 문법의 모호성을 확인할 수 있다.
  - 어떤 문장이 두 개 이상의 왼쪽 유도가 존재할 때
  - 어떤 문장이 두 개 이상의 오른쪽 유도가 존재할 때
- 모호한 문장이 왜 문제가 될까?
- 어떤 파싱 알고리즘은 모호한 문법에도 작동한다.
  - 알고리즘이나 언어의 설계자가 제공하는 비문법적인 정보를 사용한다.
- 대부분의 경우에는 모호한 문법을 모호하지 않게 바꿀 수 있다.

[1] J. Hopcroft, R. Motwani, J. Ullman. Introduction to automata theory, languages, and computation, 2001, Addision-Wesley

연산자 결합법칙

- 식이 동일한 우선순위를 갖는 두 개의 연산자를 포함할 때, 어느 연산자가 우선순위를 갖는지 명세하는 결합 규칙$_{\text{associativity}}$이 필요하다.
  - 예) $A/B*C$에서 '$/$'와 '$*$' 연산자는 왼쪽 결합법칙$_{\text{left-associative}}$을 따른다.
- 덧셈과 곱셈은 결합의 방향에 무관하다.
- 지수 연산자$_{\text{exponentiation}}$는 보통 오른쪽 결합$_{\text{right-associative}}$을 따른다.

$$\begin{aligned} \langle\text{factor}\rangle &::= \langle\text{exp}\rangle ** \langle\text{factor}\rangle \mid \langle\text{exp}\rangle \\ \langle\text{exp}\rangle &::= (\ \langle\text{exp}\rangle\ ) \mid \langle\text{id}\rangle \end{aligned}$$

모호하지 않은 문법unambiguous grammar

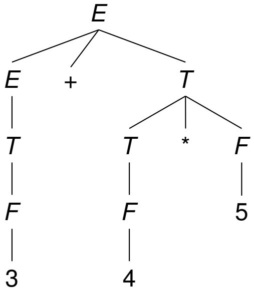
- 아래 문법은 9페이지의 모호한 문법을 모호하지 않은 문법으로 변환한 것이다.

$$E \to E + T \mid E - T \mid T$$

$$T \to T * F \mid T/F \mid F$$

$$F \to (E) \mid 0 \mid 1 \mid 2 \mid \cdots \mid 9$$

- 위 문법은 9페이지의 문법과 동일한 언어를 생성한다.
- 다시 한번 문장 $3 + 4 * 5$를 유도해보자.



모호하지 않은 문법으로의 변환

- 문법에서 가장 기초적인 피연산자를 F라 하면 이는 괄호로 묶인 산술식이나 숫자가 될 수 있다.

$$F \to (E) \mid 0 \mid 1 \mid 2 \mid \cdots \mid 9$$

- 다음으로는 연산자를 순위가 높은 것부터 취한다.
    - *와 /의 연산자 우선순위가 가장 높으므로 다음 생성과 같이 재귀적으로 구성한다.

$$T \to T * F \mid T/F \mid F$$

    - *와 /가 만약 오른쪽 결합을 따른다면?

$$T \to F * T \mid F/T \mid F$$

- 그 다음 우선순위를 갖는 +와 -에 대해 재귀적으로 생성 규칙을 만든다.

$$E \to E + T \mid E - T \mid T$$

- 파스 트리를 구성했을 때 우선순위가 낮은 연산자가 위에 오도록 문법을 구성한다.

모호하지 않은 문법으로의 변환 예

- 아래의 문법은 이전 문법에서 거듭제곱 연산자 '^' 와 단항 연산자$_{\text{unary operator}}$인 부호 연산자 '+'와 '-' 가 추가된 것이다.

$$E \to E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid E\hat{\ }E \mid -E \mid +E \mid 0 \mid 1 \mid 2 \mid \cdots \mid 9$$

- 위 문법을 모호하지 않은 문법으로 변환해보자.

모호한 문법 예 2

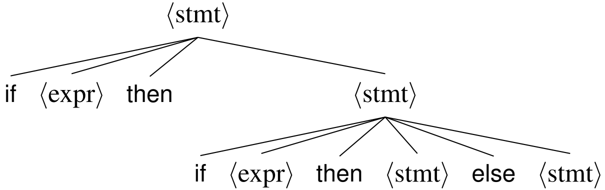
- 매달린 else 문제$_{\text{dangling else problem}}$: 중첩된 if문에서 else가 어떤 if문에 걸리는지 모호해지는 문제
- 아래의 문법은 모호한가? (아래 문법으로부터 if $\langle\text{expr}\rangle$ then if $\langle\text{expr}\rangle$ then $\langle\text{stmt}\rangle$ else $\langle\text{stmt}\rangle$를 유도해보자)

$$\begin{aligned}\langle\text{stmt}\rangle \to\ & \textit{if}\ \langle\text{expr}\rangle\ \textit{then}\ \langle\text{stmt}\rangle \\ & \mid \textit{if}\ \langle\text{expr}\rangle\ \textit{then}\ \langle\text{stmt}\rangle\ \textit{else}\ \langle\text{stmt}\rangle \\ & \mid \cdots\end{aligned}$$

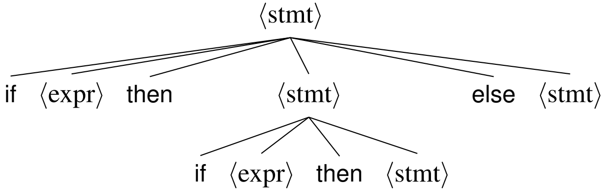
- 그렇다면 어떻게 위 문법을 모호하지 않은 문법으로 바꿀 수 있을까?

매달린 else 문제

else 절이 두 번째 if 문에 걸린 형태



else 절이 첫 번째 if 문에 걸린 형태



매달린 else 문제의 해결

- 일반적인 프로그래밍 언어에서는 else를 그 앞에 있는 가장 가까운 if와 연결하도록 한다.
- 기존 문법을 아래와 같이 변환한다.

$$
\begin{aligned}
\langle \text{stmt} \rangle &\rightarrow \langle \text{matched} \rangle \mid \langle \text{unmatched} \rangle \\
\langle \text{matched} \rangle &\rightarrow \textit{if} \ \langle \text{expr} \rangle \ \textit{then} \ \langle \text{matched} \rangle \ \textit{else} \ \langle \text{matched} \rangle \\
&\quad \mid \cdots \\
\langle \text{unmatched} \rangle &\rightarrow \mid \textit{if} \ \langle \text{expr} \rangle \ \textit{then} \ \langle \text{stmt} \rangle \\
&\quad \mid \textit{if} \ \langle \text{expr} \rangle \ \textit{then} \ \langle \text{matched} \rangle \ \textit{else} \ \langle \text{unmatched} \rangle \\
&\quad \mid \cdots
\end{aligned}
$$

- if $\langle \text{expr} \rangle$ then if $\langle \text{expr} \rangle$ then $\langle \text{stmt} \rangle$ else $\langle \text{stmt} \rangle$를 다시 한번 유도해보자.

문맥-자유 문법의 변환

- 모호한 문법 이외에도 어떤 문법들은 구문 분석 작업의 효율을 상당히 떨어뜨리는 요인을 가지고 있다.
- 이 경우 효율적인 구문 분석이 가능하도록 주어진 문법을 효율적인 문법으로 변환할 수 있다.

- 아래의 방법들을 통해 문법을 변환한다.
  - 불필요한 생성 규칙$_{\text{useless production rule}}$의 제거
  - ε-생성 규칙의 제거
  - 단일 생성 규칙$_{\text{unit production rule}}$의 제거
  - 좌인수분해$_{\text{left-factoring}}$
  - 좌재귀$_{\text{left-recursion}}$ 제거 등

불필요한 생성 규칙 제거

- 불필요한 기호는 다음과 같의 정의한다.

Definition

문맥-자유 문법 $G = (V_N, V_T, P, S)$에서 $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$와 같은 유도 과정이 존재하지 않는다면 논터미널 기호 X는 불필요한 기호$_{\text{useless symbol}}$라고 한다.

- 불필요한 기호는 다시 말해,
  1. 터미널 문자열을 생성할 수 없는 기호이거나,
  2. 시작 기호로부터 도달할 수 없는 기호를 말한다.
- 불필요한 기호를 가지고 있는 생성 규칙은 제거해도 문법이 생성하는 언어를 변경하지 않는다.

터미널 문자열을 생성할 수 없는 기호 제거하기

문맥-자유 문법 $G = (V_N, V_T, P, S)$이 주어졌을 때,

1. 논터미널 기호의 집합 $V_N' = \{A \mid A \rightarrow w \in P,\ \ w \in V_T^*\}$를 구한다.
2. 아래와 같이 집합 V_N'을 변경한다.

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

3. V_N'이 더 이상 변경되지 않을 때까지 2번 작업을 반복한다.
4. 새로운 논터미널 기호의 집합 $V_N'' = V_N - V_N'$을 구한다.
5. 새로운 생성 규칙 집합 $P' = P - \{B \rightarrow \gamma C \gamma' \mid \gamma, \gamma' \in (V_N \cup V_T)^*,\ \ B \in V_N''$ 또는 $C \in V_N''\}$를 구한다.

위의 절차를 통해 생성된 새로운 문맥-자유 문법 $G' = (V_N', V_T, P', S)$은 여전히 기존 문법 G와 동일한 언어를 생성한다.

터미널 문자열을 생성할 수 없는 기호 제거 예

- 다음 문법으로부터 터미널 문자열을 생성할 수 없는 기호를 제거해보자.

$$G = (\{S, A, B\},\ \{a\},\ P,\ S),$$

$$P = \{S \to AB \mid a,\ A \to a\}.$$

시작 기호로부터 도달 불가능한 기호 제거하기

문맥-자유 문법 $G = (V_N, V_T, P, S)$이 주어졌을 때,

1. 문법 기호의 집합 $V' = \{S\}$을 정의한다.
2. 아래와 같이 집합 V'을 변경한다.

$$V' = V' \cup \{X \mid A \in V',\ \ A \to \alpha X \beta \in P\}$$

3. V'이 더 이상 변경되지 않을 때까지 2번 작업을 반복한다.
4. 새로운 문법 기호의 집합 $V'' = V - V'$을 구한다.
5. 새로운 생성 규칙 집합 $P' = P - \{B \to \gamma\beta\gamma' \mid \gamma, \gamma' \in (V_N \cup V_T)^*,\ \ B \in V''$ 또는 $\beta \in V''\}$을 구한다.
6. 새로운 논터미널 기호 집합 $V_N' = V_N \cap V'$과 터미널 기호 집합 $V_T' = V_T \cap V'$을 정의한다.

위의 절차를 통해 생성된 새로운 문맥-자유 문법 $G' = (V_N', V_T', P', S)$은 여전히 기존 문법 G와 동일한 언어를 생성한다.

시작 기호로부터 도달 불가능한 기호 제거 예

- 다음 문법에서 시작 기호로부터 도달 불가능한 기호를 제거해보자.

$$G = (\{S, A, B\},\ \{a\},\ P,\ S),$$

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

불필요한 생성 규칙 제거하기

- 주어진 문법으로부터 불필요한 생성 규칙을 제거하는 방법은 다음과 같다.
  1. 터미널 문자열을 생성할 수 없는 기호를 제거한다.
  2. 시작 기호로부터 도달 불가능한 기호를 제거한다.

- 순서를 반대로 하면 어떻게 될까?

$$G = (\{S, A, B\},\ \{a\},\ P,\ S),$$

$$P = \{S \to AB \mid a,\ A \to a\}.$$

불필요한 생성 규칙 제거하기 예

- 다음 문법에서 불필요한 생성 규칙을 제거해보자.

$$G = (\{S, A, B, C\},\ \{a, b\},\ P,\ S),$$

$$P = \{S \to aS \mid A \mid C,\ A \to a,\ B \to aa,\ C \to aCb\}.$$

ε-자유 문법

Definition

문맥-자유 문법 $G = (V_N, V_T, P, S)$가 다음을 만족할 때 ε-자유 문법이라고 한다.

- P가 ε-생성 규칙을 가지고 있지 않거나,
- 시작 기호 S만이 $S \to \varepsilon$인 ε-생성 규칙을 가질 경우, 다른 생성 규칙의 오른쪽에 S가 나타나지 않는다.

ε-생성 규칙 제거하기

문맥-자유 문법 $G = (V_N, V_T, P, S)$이 주어졌을 때,

1. 문법 기호의 집합 $V_\varepsilon = \{A \mid A \stackrel{+}{\Rightarrow} \varepsilon,\ \ A \in V_N\}$을 정의한다.
2. 새로운 생성 규칙 집합 $P' = P - \{A \rightarrow \varepsilon \mid A \in V_N\}$을 구한다.
3. $\alpha_i \neq \varepsilon$과 $B_j \in V_\varepsilon$를 만족하는 생성 규칙 $A \rightarrow \alpha_0 B_1 \alpha_1 B_2 \cdots B_k \alpha_{k+1} \in P'$에 대해,
   - $A \rightarrow \alpha_0 X_1 \alpha_1 X_2 \cdots X_k \alpha_{k+1}$에서 $X_i = \varepsilon$ 또는 $X_i = B_i$에 의해 생성할 수 있는 모든 생성 규칙을 P'에 추가한다.
4. 만약 $S \in V_\varepsilon$인 경우, 생성 규칙 $S' \rightarrow S \mid \varepsilon$을 P'에 추가한다.

위의 절차를 통해 생성된 새로운 문맥-자유 문법 $G' = (V_N \cup \{S'\}, V_T, P', S')$은 여전히 기존 문법 G와 동일한 언어를 생성한다.

ε-생성 규칙 제거하기 예

- 다음 문법에서 ε-생성 규칙을 제거해보자.

$$G = (\{S\},\ \{a, b\},\ P,\ S),$$
$$P = \{S \to aSbS \mid bSaS \mid \varepsilon\}.$$

- 다음 문법에서 ε-생성 규칙을 제거해보자.

$$G = (\{S, A, B, C, D\},\ \{a, b, d\},\ P,\ S),$$
$$P = \{S \to ABaC,\ A \to BC,\ B \to b \mid \varepsilon,\ C \to D \mid \varepsilon,\ D \to d\}.$$

단일 생성 규칙unit production rule 제거

- 생성 규칙 중 $A \rightarrow B$와 같이 생성 규칙의 오른쪽이 단 하나의 논터미널 기호로만 구성된 생성 규칙을 단일 생성 규칙unit production rule이라 한다.

- 효율적인 구문 분석을 위해 단일 생성 규칙을 제거하는 것이 좋다.

- 먼저 단일 생성 규칙을 모두 제거하고, 제거된 단일 생성 규칙에 의해 생성될 수 있는 모든 생성 규칙을 추가한다.

단일 생성 규칙 제거하기

문맥-자유 문법 $G = (V_N, V_T, P, S)$이 주어졌을 때,

1. 새로운 생성 규칙 집합 $P' = P - \{A \to B \mid A \to B \in P,\ \ A, B \in V_N\}$을 구한다.
2. 모든 논터미널 기호 $A \in V_N$에 대해,
   2.1 논터미널 기호 집합 $V_A = \{A\}$를 정의한다.
   2.2 $V_A = V_A \cup \{C \mid B \to C \in P,\ \ B \in V_A\}$를 통해 V_A에 새로운 변경이 없을 때까지 기호를 추가한다.
3. 모든 논터미널 기호 $A \in V_N$와 $B \in V_A$에 대해, $P' = P' \cup \{A \to \alpha \mid B \to \alpha \in P'\}$를 통해 생성 규칙 집합 P'를 업데이트한다.

위의 절차를 통해 생성된 새로운 문맥-자유 문법 $G' = (V_N, V_T, P', S)$은 여전히 기존 문법 G와 동일한 언어를 생성한다.

단일 생성 규칙 제거 예

- 다음 문법에서 단일 생성 규칙을 제거해보자.

$$G = (\{S, A, B\},\ \{a, b, c\},\ P,\ S),$$
$$P = \{S \to aA \mid A,\ A \to bB \mid B,\ B \to c\}.$$

- 다음 문법에서 단일 생성 규칙을 제거해보자.

$$G = (\{E, T, F\},\ \{(,), +, *, a\},\ P,\ E),$$
$$P = \{E \to E + T \mid T,\ T \to T * F \mid F,\ F \to (E) \mid a\}.$$

순환-자유$_{\text{cycle-free}}$ 문법과 proper한 문법

Definition

문맥-자유 문법 $G = (V_N, V_T, P, S)$가 어떤 $A \in V_N$에 대해 $A \overset{+}{\Rightarrow} A$ 꼴의 유도 과정을 가지지 않을 때, 순환-자유$_{\text{cycle-free}}$라 한다.

Definition

문맥-자유 문법 $G = (V_N, V_T, P, S)$가 순환-자유이고, ε-자유이며 불필요한 생성 규칙을 갖지 않을 때, 그 문법을 proper하다고 한다.

문법이 proper한지 확인하기

- 다음 문법이 proper한지 확인해보자.

$$G = (\{E, T, F\},\ \{(,), +, *, a\},\ P,\ E),$$

$$P = \{E \to E + T \mid T * F \mid (E) \mid a,\ T \to T * F \mid (E) \mid a,\ F \to (E) \mid a\}.$$

좌인수분해left-factoring

- 같은 기호를 접두사로 가진 2개 이상의 생성 규칙이 존재할 때, 공통된 접두사를 인수분해하는 과정을 좌인수분해라고 한다.

- 같은 기호를 접두사로 갖는 생성 규칙이 여러 개 존재하면, 하향식 구문 분석 방법에서는 어떤 생성 규칙을 적용해야 할지 결정할 수 없다.

- 예를 들어 아래와 같은 문법에서 문장 *caaabbb*를 유도해보자.

$$G = (\{S, A\},\ \{a, b, c, d\},\ P,\ S),$$

$$P = \{S \to cAd,\ A \to a \mid ab\}$$

좌인수분해 알고리즘

문맥-자유 문법 $G = (V_N, V_T, P, S)$이 주어졌을 때,

1. 아래와 같은 생성 규칙이 있다고 하자.

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma.$$

2. 생성 규칙의 오른쪽에서 가장 긴 공통 접두사인 α를 구한다.
3. 만약 $\alpha \neq \varepsilon$이라면, 위 생성 규칙을 아래의 꼴로 대체한다.

$$A \to \alpha A' \mid \gamma$$

$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

4. 공통된 왼쪽 논터미널과 오른쪽 진접두사를 갖는 생성 규칙이 없을 때까지 위 과정을 반복한다.

좌인수분해 예

다음 문법을 좌인수분해해보자.

$$
\begin{aligned}
\langle \mathrm{stmt} \rangle \rightarrow\ & \textit{if}\ \langle \mathrm{expr} \rangle\ \textit{then}\ \langle \mathrm{stmt} \rangle \\
\mid\ & \textit{if}\ \langle \mathrm{expr} \rangle\ \textit{then}\ \langle \mathrm{stmt} \rangle\ \textit{else}\ \langle \mathrm{stmt} \rangle \\
\mid\ & \textit{statement} \\
\langle \mathrm{expr} \rangle \rightarrow\ & \textit{expression}
\end{aligned}
$$

좌재귀$_{\text{left-recursion}}$ 제거하기

- 문법에서 어떤 문자열 α에 대해 $A \overset{*}{\Rightarrow} A\alpha$의 유도 과정이 존재하는 경우를 좌재귀$_{\text{left-recursion}}$라 한다.
- 좌재귀 문법의 경우 하향식 구문 분석을 할 때 같은 생성 규칙이 순환 적용되어 무한 루프에 빠지게 된다.
- $A \to A\alpha$ 꼴의 생성 규칙이 존재하는 경우를 직접 좌재귀$_{\text{direct left-recursion}}$라 하고 $A \overset{*}{\Rightarrow} A\alpha$ 형태의 유도가 존재하는 경우를 간접 좌재귀$_{\text{indirect left-recursion}}$라 한다.
- 직접 좌재귀는 다음과 같은 방법으로 제거한다.
  1. 각 논터미널 A에 대해 $A \to A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$와 같이 오른쪽이 A로 시작하는 규칙들과 아닌 규칙들로 묶는다.
  2. 기존 문법에서 A에 대한 생성 규칙들을 다음과 같이 대체한다.

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_m A' \mid \varepsilon$$

직접 좌재귀 제거 예

- 아래의 예제 문법에서 직접 좌재귀 규칙을 제거해보자.

$$E \to E + T \mid T, \;\; T \to T * F \mid F, \;\; F \to (E) \mid id$$

- E의 생성 규칙에 대해, $\alpha_1 = +T, \beta_1 = T$가 되므로, E의 규칙을 다음으로 대체한다.

$$E \to TE', \;\; E' \to +TE' \mid \varepsilon$$

- T의 생성 규칙에 대해, $\alpha_1 = *F, \beta_1 = F$가 되므로, T의 규칙을 다음으로 대체한다.

$$T \to FT', \;\; T' \to *FT' \mid \varepsilon$$

- 결과 문법은 다음과 같다.

$$E \to TE', \;\; E' \to +TE' \mid \varepsilon, \;\; T \to FT', \;\; T' \to *FT' \mid \varepsilon, \;\; F \to (E) \mid id$$

간접 좌재귀 제거하기

- 간접 좌재귀를 제거하기 전에 먼저 직접 좌재귀를 제거해야 한다.
- 직접 좌재귀가 없는 문법에 대해,
  1. 문법의 전체 논터미널 기호의 순서를 $A_1, A_2, \ldots, A_n$ 꼴로 정렬한다.
  2. i를 2부터 n까지 늘려가면서, i보다 작은 j에 대해 $A_i \to A_j\gamma$ 꼴의 생성 규칙을 $A_i \to \alpha_1\gamma \mid \alpha_2\gamma \mid \cdots \mid \alpha_k\gamma$로 대체한다. 이 때, $A_j \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$이다.
  3. A_i의 생성 규칙이 직접 좌재귀를 갖는다면 제거한다.

- 아래 문법에서 좌재귀를 제거해보자.

$$S \to Aa \mid b$$

$$A \to Ac \mid Sd \mid e$$

Any questions?

참고문헌

- A. Aho, J. Ullman, R. Sethi, M. S. Lam, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, 2006
- R. Sebesta, Concepts of Programming Languages, 5th Edition, Addison-Wesley, 2001
- K. C. Louden, Compiler Construction: Principles and Practice, Cengage Learning, 1997
- K. C. Louden and K. A. Lambert, Programming languages : Principles and Practice, 3rd Edition, Cengage Learning, 2012
- 박두순, 컴파일러의 이해, 한빛아카데미, 2016
- 김종훈, 김종진, 프로그래밍 언어론 : 쉽게 배우는 언어의 원리와 구조, 한빛미디어, 2013