

문맥 자유 문법 (Context Free Grammar)

문맥 자유 문법은 컴파일러 설계의 핵심 개념으로, 프로그래밍 언어의 구문 구조를 정의하는 형식적인 방법입니다.

문맥 자유 문법의 구성 요소

- 종단 기호 (Terminal symbols): 언어의 기본 요소들
- 비종단 기호 (Non-terminal symbols): 구문 구조를 나타내는 변수들
- 생성 규칙 (Production rules): 비종단 기호가 어떻게 다른 기호들로 대체될 수 있는지 정의
- 시작 기호 (Start symbol): 문법의 시작점

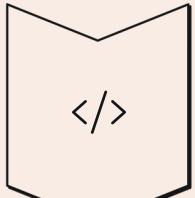
문맥 자유 문법의 중요성

문맥 자유 문법은 다음과 같은 이유로 컴파일러 개발에 필수적입니다:

- 프로그래밍 언어의 구문을 명확하게 정의
- 구문 분석기(Parser) 생성의 기반
- 언어의 모호성 식별 및 해결
- 코드의 구조적 정확성 검증

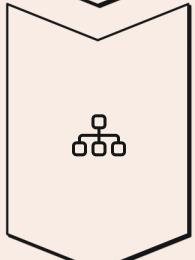
문맥 자유 문법은 특히 컴파일러의 구문 분석 단계에서 중요한 역할을 합니다. 이를 통해 소스 코드의 구조를 파악하고, 이후 단계인 의미 분석과 코드 생성을 위한 기반을 마련합니다. 파서는 이 문법을 바탕으로 입력된 토큰 스트림이 언어의 구문 규칙을 따르는지 확인합니다.

분석 단계 (Front-end)



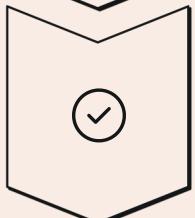
어휘 분석 (Lexical Analysis)

입력 문자열을 토큰(tokens)이라 불리는 "단어"로 분리합니다. 이 단계를 담당하는 것이 렉서(Lexer)입니다.



구문 분석 (Syntactic Analysis)

토큰으로부터 구조를 구성하고 이를 파스 트리(parse tree)로 표현합니다. 이 단계를 담당하는 것이 파서(Parser)입니다.



의미 분석 (Semantic Analysis)

"의미"를 결정합니다(예: 타입 체킹). 코드 생성을 준비하고 심볼 테이블(symbol table)과 상호작용합니다.

컴파일러의 프론트엔드 부분은 소스 코드를 분석하여 중간 표현(IR)으로 변환하는 역할을 합니다. 이 과정에서 코드의 구조와 의미를 파악하고 오류를 검출합니다. 각 단계는 순차적으로 진행되며, 이전 단계의 결과물이 다음 단계의 입력으로 사용됩니다. 이러한 분석 과정을 통해 컴파일러는 고급 언어로 작성된 코드를 기계가 이해할 수 있는 형태로 변환할 준비를 합니다.

어휘 분석 (Lexical Analysis)

어휘 분석은 컴파일러의 첫 번째 단계로, 소스 코드를 의미 있는 단위(토큰)로 분류합니다.



렉서 (Lexer)

문자 시퀀스를 의미 있는 단위(토큰)로 수집합니다.



토큰 (Token)

식별자(사용자 정의 이름), 예약어, 구분자, 연산자, 특수 기호 등을 포함합니다.



토큰화 과정

소스 코드를 스캔하여 패턴을 인식하고 적절한 토큰 유형으로 분류합니다.

예를 들어, `a[index] = 4 + 2` 라는 소스 코드는 다음과 같이 8개의 토큰으로 분리됩니다:

식별자	구분자	식별자	구분자	연산자	숫자	연산자	숫자
a	[index]	=	4	+	2

이러한 토큰화 과정은 구문 분석을 위한 준비 단계로, 코드의 기본 구성 요소를 식별합니다. 토큰화가 완료되면 이 토큰들은 파서(Parser)로 전달되어 구문 분석 단계로 넘어갑니다.

토큰 명세: 정규 언어 (Regular Language)

정규 언어는 유효한 토큰 패턴을 정의하는 데 사용됩니다.

정규 언어의 특징

- 유효한 토큰 패턴을 정의하는 데 사용됨
- 유한 상태 기계(Finite State Machine)로 인식 가능한 형식 언어
- 정규 표현식(Regular Expression)으로 표현 가능

정규 언어 예시

- letter → [a-zA-Z]
- digit → [0-9]
- id → letter(letter|digit)*

유한 상태 기계 종류

- 결정적 유한 오토마타 (DFA)
- 비결정적 유한 오토마타 (NFA)

컴파일러는 정규 언어를 사용하여 소스 코드에서 유효한 토큰을 식별합니다. 이는 어휘 분석의 핵심 메커니즘입니다.

정규 언어는 컴파일러 설계에서 중요한 역할을 하며, 문법적으로 간단하지만 토큰 인식에 충분한 표현력을 제공합니다. 유한 상태 기계는 이러한 정규 언어를 구현하는 효율적인 방법입니다.

언어와 오토마타 (Language and Automata)



언어란 무엇인가?

형식 언어(Formal language)는 특정 규칙에 따라 구성된 문자열의 집합입니다. 이는 알파벳(Alphabet)으로부터 생성된 문자열 중 특정 문법 규칙을 만족하는 부분집합입니다.



언어를 정의하는 방법

정규 언어의 경우 정규 표현식(Regular expression)을 사용하여 정의합니다. 문맥-자유 언어(Context-free language)는 문맥-자유 문법(CFG)으로 정의됩니다.



언어의 단어를 인식하는 방법

정규 언어는 유한 오토마타(Finite automata)로, 문맥-자유 언어는 푸시다운 오토마타(Pushdown automata)로 인식합니다. 각 오토마타는 해당 언어 클래스의 인식 능력에 맞게 설계되었습니다.

언어 이론에서 오토마타는 언어를 인식하는 추상적인 기계입니다. 정규 언어는 유한 오토마타로 인식할 수 있는 가장 단순한 형태의 형식 언어입니다. 이러한 이론적 기반은 컴파일러 설계와 형식 언어 분석에 필수적입니다.

형식 언어 (Formal Language)

알파벳 (Alphabet, Σ)

언어에서 단어를 구성하는 데 사용되는 기호(문자)의 유한 집합입니다.

예: {a, b}, {0, 1}

문자열 (String, word)

알파벳에서 가져온 기호의 유한 시퀀스입니다.

예: {a, b}에서 "abab", "aa", "aabb"

언어 (Language, L)

특정 규칙(문법)을 따르는 문자열의 집합입니다.

알파벳 Σ 에 대한 언어 L 은 Σ^* 의 부분 집합입니다. ($L \subseteq \Sigma^*$)

예: $L = \{anbn \mid n \geq 1\} = \{ab, aabb, aaabbb, \dots\}$ (무한 언어)

형식 언어는 컴퓨터 과학에서 언어를 수학적으로 정의하고 분석하는 데 사용됩니다. 컴퓨터 설계에서 프로그래밍 언어의 구문을 정의하는 기초가 됩니다.

인식 문제 (Recognition Problem)

인식 문제는 주어진 문자열이 특정 언어에 속하는지 여부를 결정하는 문제입니다.

주어진 문자열 $w \in \Sigma^*$ 와 알파벳 Σ 에 대한 언어 L 에 대해, w 가 L 에 속하는지 여부를 결정하는 것입니다.

예시:

- $w_1 = 100011$, $w_2 = 1000111$
- L 은 0과 1의 개수가 동일한 문자열의 언어
- $w_1 \in L$? (예, 0과 1이 각각 3개씩 있음)
- $w_2 \in L$? (아니오, 0이 3개, 1이 4개 있음)

인식 문제는 컴파일러가 소스 코드의 토큰이나 구문이 프로그래밍 언어의 규칙에 맞는지 확인하는 과정과 직접적으로 관련됩니다.

형식 언어의 동기

인식 관점

문자열 w 를 입력으로 받아 두 가지 가능한 답변을 반환하는 "블랙 박스"를 가정합니다:

- 예, w 는 **수락됨** → w 는 언어 L 에 속함
- 아니오, w 는 **거부됨** → w 는 언어 L 에 속하지 않음



오토마타 (Automata)

언어를 인식하는 추상적 기계입니다.

예: 정규 언어를 위한 유한 오토마타

생성 관점

합법적인 문자열을 생성할 수 있는 규칙의 관점에서 언어를 지정합니다.



문법 (Grammar)

언어를 생성하는 규칙 체계입니다.

예: 문맥 자유 문법(CFG)은 프로그래밍 언어의 구문 구조를 정의합니다.

형식: 생성 규칙 집합으로 $V \rightarrow W$ 형태로 표현됩니다.

형식 언어 이론은 컴파일러가 프로그래밍 언어의 구문을 정의하고 인식하는 데 필요한 이론적 기반을 제공합니다.

정규 언어 (Regular Language)

정규 언어는 컴파일러의 어휘 분석 단계에서 중요한 역할을 하지만, 모든 언어 구조를 표현하기에는 한계가 있습니다.

정규 언어의 특징

- 유효한 토큰 패턴을 정의하는 데 사용됨
- 유한 오토마타로 인식 가능한 형식 언어
- 정규 표현식으로 표현 가능

정규 언어의 한계

복잡한 언어 구조를 표현하기 어렵습니다.

예: $L = \{(), (), ((())), (((()))), \dots\}$

(*)*와 같은 중첩된 구조는 정규 표현식으로 표현할 수 없습니다.

이러한 한계 때문에 컴파일러의 구문 분석 단계에서는 더 강력한 문법 체계인 문맥 자유 문법(CFG)을 사용합니다.

문맥 자유 문법 (Context-Free Grammar)

문맥 자유 문법(CFG)은 정규 언어의 한계를 극복하고 더 복잡한 언어 구조를 표현할 수 있는 형식 문법입니다. 이는 프로그래밍 언어의 구문 구조를 정의하는 데 사용됩니다.

CFG의 구성 요소

- 비단말 기호 (Non-terminal symbols)
- 단말 기호 (Terminal symbols)
- 생성 규칙 (Production rules)
- 시작 기호 (Start symbol)

문맥 자유 문법의 특징

- 중첩된 구조 표현 가능
- 재귀적 정의 지원
- 푸시다운 오토마타로 인식 가능

정규 언어와의 차이점

정규 언어가 표현할 수 없는 " $anbn$ ($n \geq 1$)"과 같은 언어를 표현할 수 있습니다.

예: $S \rightarrow aSb \mid ab$

문맥 자유 문법은 컴파일러의 구문 분석 단계에서 핵심적인 역할을 하며, 프로그래밍 언어의 중첩된 구조(예: 괄호, 블록, 함수 호출 등)를 정의하는 데 필수적입니다.

문법 예시: 반복 패턴

문법은 반복적인 패턴을 가진 언어도 간결하게 표현할 수 있습니다.

문법 규칙

- $S \rightarrow \text{dingdongA}$
- $A \rightarrow \text{dengS} \mid \text{deng}$

유도 예시

$S \Rightarrow \text{dingdongA} \Rightarrow \text{dingdongdeng}$
 $S \Rightarrow \text{dingdongA} \Rightarrow$
 $\text{dingdongdengS} \Rightarrow$
 $\text{dingdongdengdingdongA} \Rightarrow$
 $\text{dingdongdengdingdongdeng}$

생성되는 언어

$L = \{(\text{dingdongdeng})^n \mid n \geq 1\}$
또는 정규 표현식으로:
 $(\text{dingdongdeng})^+$

이 예시는 재귀적인 규칙을 통해 반복 패턴을 가진 언어를 생성하는 방법을 보여줍니다. 이러한 패턴은 프로그래밍 언어에서 반복 구조를 표현하는 데 유용합니다.

문법 예시: 조건부 패턴

문법은 특정 조건을 만족하는 문자열 집합을 정의할 수 있습니다.

문법 규칙

- $S \rightarrow 0S \mid 0B$
- $B \rightarrow 1C$
- $C \rightarrow 0C \mid 0$

생성되는 언어

$$L(G) = \{0^n 1 0^m \mid n, m \geq 1\}$$

예시 문자열

010, 0010, 00100, 0000010,
0010000, ...

이 문법은 하나 이상의 0으로 시작하고, 중간에 1이 하나 있으며, 하나 이상의 0으로 끝나는 문자열 집합을 정의합니다. 이러한 조건부 패턴은 프로그래밍 언어의 특정 구문 구조를 정의하는 데 사용될 수 있습니다.

형식 문법 (Formal Grammar)

형식 문법 G 는 (VN , VT , P , S)로 정의됩니다.



VN : 비단말 기호 (non-terminal symbols)

문법의 규칙에서 다른 기호로 대체될 수 있는 기호들입니다.



VT : 단말 기호 (terminal symbols)

알파벳 또는 토큰으로, 더 이상 대체되지 않는 기호들입니다.

$$VN \cap VT = \emptyset, VN \cup VT = V$$



P : 생성 규칙 (productions)

문자열 쌍의 유한 집합으로, $\alpha \rightarrow \beta$ 형태를 가집니다.

$$\alpha \in V^+, \beta \in V^*$$

α : 좌변(lhs), β : 우변(rhs)

\rightarrow : 좌측 기호가 우측 기호로 대체됨을 의미



S : 시작 기호 (start symbol)

특별한 비단말 기호로, 모든 유도가 시작되는 기호입니다.

형식 문법은 프로그래밍 언어의 구문을 정의하는 데 사용되며, 컴파일러의 구문 분석 단계에서 중요한 역할을 합니다.

문법 표기법

문법을 표현할 때 사용하는 추가적인 표기법입니다.



비단말 기호

대문자로 표기합니다.



단말 기호

소문자로 표기합니다.



시작 기호

일반적으로 S로 표기합니다.

문법을 제시할 때는 종종 생성 규칙의 집합만 표시하며, VN과 VT는 생성 규칙에서 유추할 수 있습니다. 또한 같은 좌변을 가진 생성 규칙은 그룹화하여 표현합니다.

S → A | a (S → A S → a 대신)

예시:

- G = ({S,A}, {0,1}, P, S)
- P: S → 0AS | 0
- A → S1A | 10 | SS

이 예시에서 비단말 기호는 {S, A}, 단말 기호는 {0, 1}이며, 생성 규칙의 수는 5개입니다.

유도 과정 (Derivation)

유도는 생성 규칙을 적용하는 단계별 과정입니다.

- 시작 기호 s 에서 시작
- 언어의 문자열을 생성하기 위해 생성 규칙을 적용

만약 $\gamma\alpha\delta$ 가 VT^* 의 문자열이고 $\alpha\rightarrow\beta$ 가 G 의 생성 규칙이라면, $\gamma\alpha\delta$ 는 $\gamma\beta\delta$ 를 직접 유도한다고 말하며 $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$ 로 표기합니다.

- $\Rightarrow^*:$ 전체 유도 과정(0단계 이상)
- 0단계? $s \Rightarrow^* s$ (규칙 적용 없음)
- $\Rightarrow^+:$ 1단계 이상의 유도

유도 과정은 문법이 어떻게 언어의 문자열을 생성하는지 보여주는 중요한 개념입니다.

유도 예시

다음 문법에 대한 유도 예시를 살펴보겠습니다:

$$G = (\{S, A\}, \{0, 1\}, P, S)$$

$$P: S \rightarrow 0AS \mid 0$$

$$A \rightarrow S1A \mid 10 \mid SS$$

첫 번째 유도:

$$S \Rightarrow 0 \text{ (} S \rightarrow 0 \text{ 규칙 적용)}$$

두 번째 유도:

$$S \Rightarrow 0AS \text{ (} S \rightarrow 0AS \text{ 규칙 적용)}$$

$$\Rightarrow 0SSS \text{ (} A \rightarrow SS \text{ 규칙 적용)}$$

$$\Rightarrow 00SS \text{ (첫 번째 } S \text{에 } S \rightarrow 0 \text{ 규칙 적용)}$$

$$\Rightarrow 000S \text{ (두 번째 } S \text{에 } S \rightarrow 0 \text{ 규칙 적용)}$$

$$\Rightarrow 0000 \text{ (마지막 } S \text{에 } S \rightarrow 0 \text{ 규칙 적용)}$$

세 번째 유도:

$$S \Rightarrow 0AS \text{ (} S \rightarrow 0AS \text{ 규칙 적용)}$$

$$\Rightarrow 0S1AS \text{ (} A \rightarrow S1A \text{ 규칙 적용)}$$

$$\Rightarrow 001AS \text{ (} S \rightarrow 0 \text{ 규칙 적용)}$$

$$\Rightarrow 00110S \text{ (} A \rightarrow 10 \text{ 규칙 적용)}$$

$$\Rightarrow 001100 \text{ (} S \rightarrow 0 \text{ 규칙 적용)}$$

이러한 유도 과정을 통해 문법이 생성하는 언어의 문자열을 확인할 수 있습니다. 각 단계에서 어떤 생성 규칙이 적용되었는지 명시하여 유도 과정의 정확한 흐름을 파악할 수 있습니다.

문법에 의해 생성된 언어

문법 $G = (VN, VT, P, S)$ 가 주어졌을 때, G 에 의해 생성된 언어 $L(G)$ 는 다음과 같이 정의됩니다:

$$L(G) = \{w \mid w \in VT^* \text{ and } S \xrightarrow{*} w\}$$

즉, 시작 기호 S 에서 시작하여 생성 규칙을 적용해 얻을 수 있는 모든 단말 기호로만 구성된 문자열의 집합입니다.

예시:

- $G = (\{S, A\}, \{0, 1\}, P, S)$
- $VT^* = \{\epsilon, 0, 1, 01, 10, 111, 0000, 110101010, \dots\}$
- $P: S \xrightarrow{} 0AS \mid 0$
- $A \xrightarrow{} S1A \mid 10 \mid SS$
- $L(G) = \{0, 0000, 001100, \dots\}$

이 예시에서 $L(G)$ 는 문법 G 에 의해 생성될 수 있는 모든 문자열의 집합입니다. 이는 VT^* 의 부분집합으로, 특정 규칙에 따라 생성된 문자열만 포함합니다.

중요한 성질:

- 동일한 언어는 여러 다른 문법에 의해 생성될 수 있습니다.
- 모든 문맥 자유 문법은 정규 표현식으로 표현할 수 없는 언어도 생성할 수 있습니다.
- 특정 문법이 생성하는 언어를 이해하기 위해서는 해당 문법의 유도 과정을 분석해야 합니다.

문법의 유도를 통해 언어 $L(G)$ 의 모든 문자열을 생성할 수 있으며, 이는 컴파일러와 파서 설계의 기본 원리가 됩니다. 다음 섹션에서 유도 예시를 통해 문법이 어떻게 언어를 생성하는지 더 자세히 살펴보겠습니다.

문법 예시: 홀수 길이 문자열

다음 문법을 살펴보겠습니다:

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

$$P: S \rightarrow a \mid aE$$

$$E \rightarrow aS$$

이 문법의 유도 예시:

- $S \Rightarrow a$ ($S \rightarrow a$)
- $S \Rightarrow aE$ ($S \rightarrow aE$) $\Rightarrow aaS$ ($E \rightarrow aS$) $\Rightarrow aaa$ ($S \rightarrow a$)
- $S \Rightarrow aE$ ($S \rightarrow aE$) $\Rightarrow aaS$ ($E \rightarrow aS$) $\Rightarrow aaaE$ ($S \rightarrow aE$) $\Rightarrow aaaaS$ ($E \rightarrow aS$) $\Rightarrow aaaaa$ ($S \rightarrow a$)

이 문법이 생성하는 언어:

$$L(G) = \{a^{2n+1} \mid n \geq 0\} = a(aa)^* = (aa)^*a$$

이 문법은 홀수 길이의 a로만 구성된 문자열을 생성합니다. 이는 a로 시작하고 끝나며, 중간에 짝수 개의 a가 있는 형태입니다.

각 단계에서 S는 a 하나로 대체되거나 aE로 확장되며, E는 항상 aS로 대체됩니다. 이 패턴으로 인해 항상 홀수 개의 a가 생성되는 것을 확인할 수 있습니다. 이러한 형태의 문법은 특정 패턴을 가진 문자열 집합을 정확히 정의하는 데 매우 유용합니다.

문법 예시: 조건부 패턴

다음 문법을 살펴보겠습니다:

$$G = (\{S, B, C\}, \{0, 1\}, P, S)$$

$$P: S \rightarrow 0S \mid 0B$$

$$B \rightarrow 1C$$

$$C \rightarrow 0C \mid 0$$

이 문법이 생성하는 언어:

$$L(G) = \{0^n 1 0^m \mid n, m \geq 1\}$$

이 언어는 하나 이상의 0으로 시작하고, 중간에 1이 하나 있으며, 하나 이상의 0으로 끝나는 모든 문자열의 집합입니다.

유도 예시:

- $S \Rightarrow 0B \quad (S \rightarrow 0B) \Rightarrow 01C \quad (B \rightarrow 1C) \Rightarrow 010 \quad (C \rightarrow 0)$
- $S \Rightarrow 0S \quad (S \rightarrow 0S) \Rightarrow 00B \quad (S \rightarrow 0B) \Rightarrow 001C \quad (B \rightarrow 1C) \Rightarrow 0010 \quad (C \rightarrow 0)$
- $S \Rightarrow 0S \quad (S \rightarrow 0S) \Rightarrow 00S \quad (S \rightarrow 0S) \Rightarrow 000B \quad (S \rightarrow 0B) \Rightarrow 0001C \quad (B \rightarrow 1C) \Rightarrow 00010C \quad (C \rightarrow 0C) \Rightarrow 000100 \quad (C \rightarrow 0)$

이 문법은 0이 한 개 이상 나오고, 1이 정확히 한 번 나온 후, 0이 한 개 이상 나오는 패턴을 정의합니다. 이러한 조건부 패턴은 프로그래밍 언어에서 특정 구문 구조를 정의하는 데 유용합니다.

촘스키 계층 구조 (Chomsky Hierarchy)

미국의 언어학자 노암 촘스키(Noam Chomsky)가 제안한 형식 문법의 네 가지 유형입니다.

유형	문법	인식 기계	예시
0	무제한 문법 (Unrestricted Grammar)	튜링 머신 (Turing Machine)	모든 계산 가능한 언어
1	문맥 의존 문법 (Context-Sensitive Grammar, CSG)	선형 제한 오토마타 (Linear-Bounded Automata, LBA)	제약이 있는 구문 (예: 태입 체킹)
2	문맥 자유 문법 (Context-Free Grammar, CFG)	푸시다운 오토마타 (Pushdown Automata, PDA)	중첩 구조 파싱
3	정규 문법 (Regular Grammar, Regexp)	유한 오토마타 (Finite Automata, DFA/NFA)	키워드, 식별자, 식별자

각 유형은 이전 유형보다 더 제한적인 규칙을 가지며, 더 효율적인 인식 기계로 처리할 수 있습니다. 컴파일러 설계에서는 각 단계에 적합한 문법 유형을 사용합니다.

촘스키 계층 구조: 유형 0과 유형 1

유형 0 - 무제한 문법

모든 $\alpha \rightarrow \beta$, $\alpha \in V^+$, $\beta \in V^*$

가장 일반적인 형태의 문법으로, 어떤 제약도 없습니다.

튜링 머신으로 인식 가능합니다.

유형 1 - 문맥 의존 문법 (CSG)

모든 $\alpha \rightarrow \beta$ 에 대해, $|\alpha| \leq |\beta|$ (길이), $\alpha \in V^+$, $\beta \in V^*$

우변의 길이가 좌변의 길이보다 크거나 같아야 합니다.

예: $S \rightarrow A$

$A \rightarrow abC \mid aABC$

$bB \rightarrow bbb$

$bC \rightarrow bb$

유형 0 문법은 가장 강력하지만 인식하기 가장 어려운 문법입니다. 유형 1 문법은 좌변이 우변으로 대체될 때 문맥을 고려하며, 문자열의 길이가 줄어들지 않는 제약이 있습니다.

촘스키 계층 구조: 유형 2와 유형 3



유형 2 - 문맥 자유 문법 (CFG)

모든 $\alpha \rightarrow \beta$ 에 대해, $\alpha \in VN$, $\beta \in V^*$ (즉, $A \rightarrow \beta$)

좌변이 단일 비단말 기호여야 합니다.

예: $S \rightarrow 0AS \mid 0$, $A \rightarrow S1A \mid 10 \mid SS$

유형 3 - 정규 문법 (Regular Grammar)

$\alpha, \beta \in VN$, $t \in VT^*$

$\alpha \rightarrow t\beta \mid t$ (우선형)

$\alpha \rightarrow \beta t \mid t$ (좌선형)

정규 언어와 동등합니다.

예: $(a|b)^*abb$

$S \rightarrow aS \mid bS \mid aA$, $A \rightarrow bB$, $B \rightarrow b$

유형 2 문법(CFG)은 컴파일러의 구문 분석에 주로 사용되며, 중첩된 구조를 표현할 수 있습니다. 유형 3 문법(정규 문법)은 어휘 분석에 사용되며, 정규 표현식과 동등한 표현력을 가집니다.

정규 표현식과 정규 문법의 관계

정규 표현식	정규 문법
a	$S \rightarrow a$
a^*	$S \rightarrow aS \mid \epsilon$
ab	$S \rightarrow aA, A \rightarrow b$
$a \mid b$	$S \rightarrow a \mid b$
$(ab)^*$	$S \rightarrow aA, A \rightarrow bS \mid \epsilon$

정규 표현식과 정규 문법은 동등한 표현력을 가지며, 서로 변환이 가능합니다. 이는 어휘 분석기가 정규 표현식으로 정의된 토큰 패턴을 인식할 수 있는 이론적 기반이 됩니다.

예를 들어, 'a'라는 단일 문자를 인식하는 정규 표현식은 ' $S \rightarrow a$ '라는 정규 문법으로 표현할 수 있습니다. 마찬가지로, ' a^* '(0개 이상의 a)는 ' $S \rightarrow aS \mid \epsilon$ '로 표현할 수 있습니다.

DFA에서 정규 문법으로의 변환

결정적 유한 오토마타(DFA)는 우선형 정규 문법으로 변환할 수 있습니다. 변환 방법은 다음과 같습니다:

1. DFA의 각 상태를 비단말 기호로 변환합니다.
2. 시작 상태를 문법의 시작 기호로 설정합니다.
3. 각 전이 $\delta(q, a) = p$ 에 대해, 생성 규칙 $q \rightarrow ap$ 를 추가합니다.
4. 각 수락 상태 q 에 대해, 생성 규칙 $q \rightarrow \epsilon$ 을 추가합니다.

이 변환을 통해 DFA가 인식하는 언어와 정확히 동일한 언어를 생성하는 정규 문법을 얻을 수 있습니다.



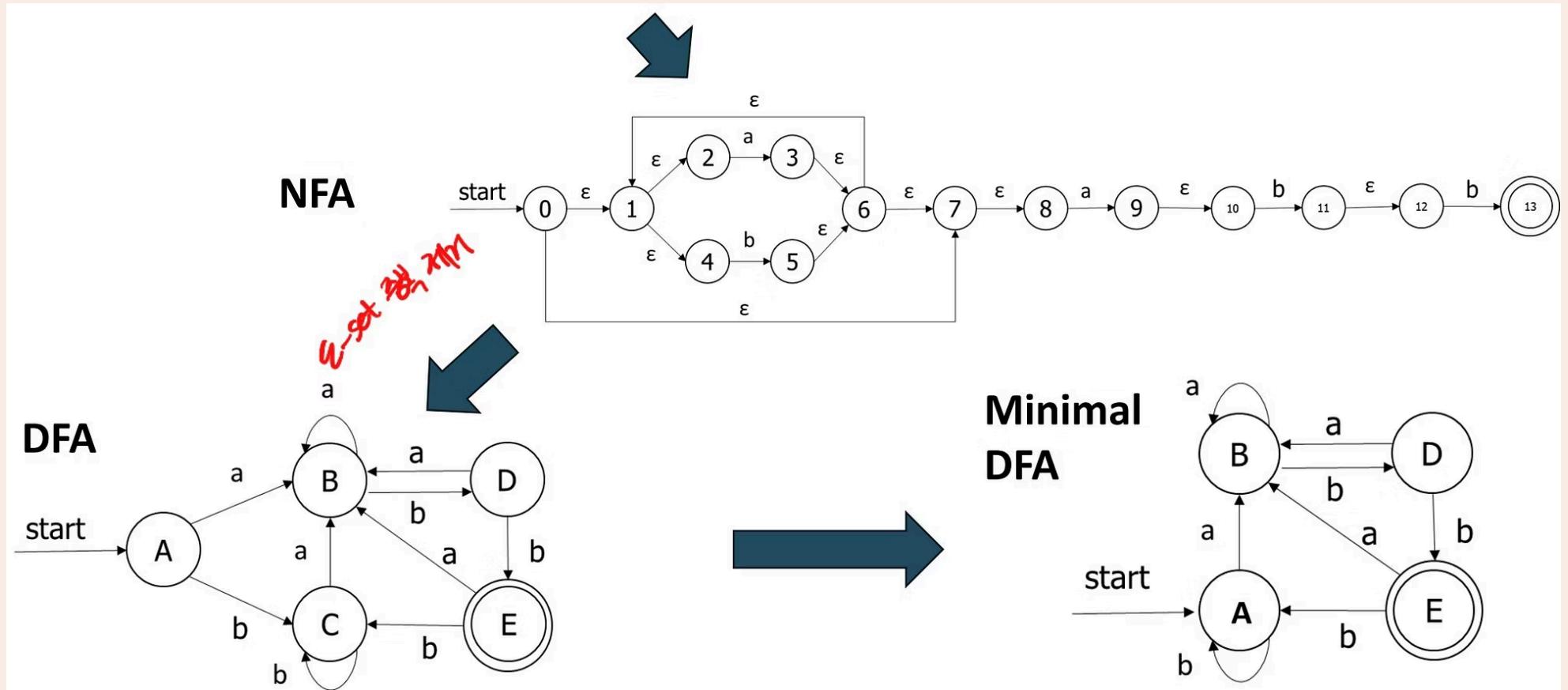
촘스키 계층 구조 요약

문법	생성 규칙
무제한 (Unrestricted)	$\alpha \rightarrow \beta, \alpha \in V^+, \beta \in V^*$
문맥 의존 (Context-Sensitive)	$\alpha \rightarrow \beta, \alpha \leq \beta , \alpha \in V^+, \beta \in V^*$
문맥 자유 (Context-Free)	$\alpha \rightarrow \beta, \alpha \in VN, \beta \in V^*$
정규 (Regular)	$\alpha \rightarrow t\beta \mid t$ (우선형) 또는 $\alpha \rightarrow \beta t \mid t$ (좌선형), $\alpha, \beta \in VN, t \in VT^*$

각 문법 유형은 특정 제약 조건을 가지며, 이러한 제약은 문법의 표현력과 인식 복잡성에 영향을 미칩니다. 컴파일러 설계에서는 각 단계에 적합한 문법 유형을 선택하여 사용합니다.

촘스키 계층 구조는 형식 언어를 분류하는 중요한 체계로, 상위 계층의 언어는 하위 계층의 언어를 포함합니다. 정규 언어는 가장 제한적이지만 인식하기 가장 쉬우며, 무제한 문법은 가장 표현력이 풍부하지만 일반적으로 결정 불가능한 문제를 포함합니다.

정규 언어 표현: $(a|b)^*abb$



정규 표현식 $(a|b)^*abb$ 는 a와 b로 구성된 문자열 중에서 abb로 끝나는 모든 문자열을 나타냅니다. 이 정규 표현식은 다음과 같이 해석됩니다:

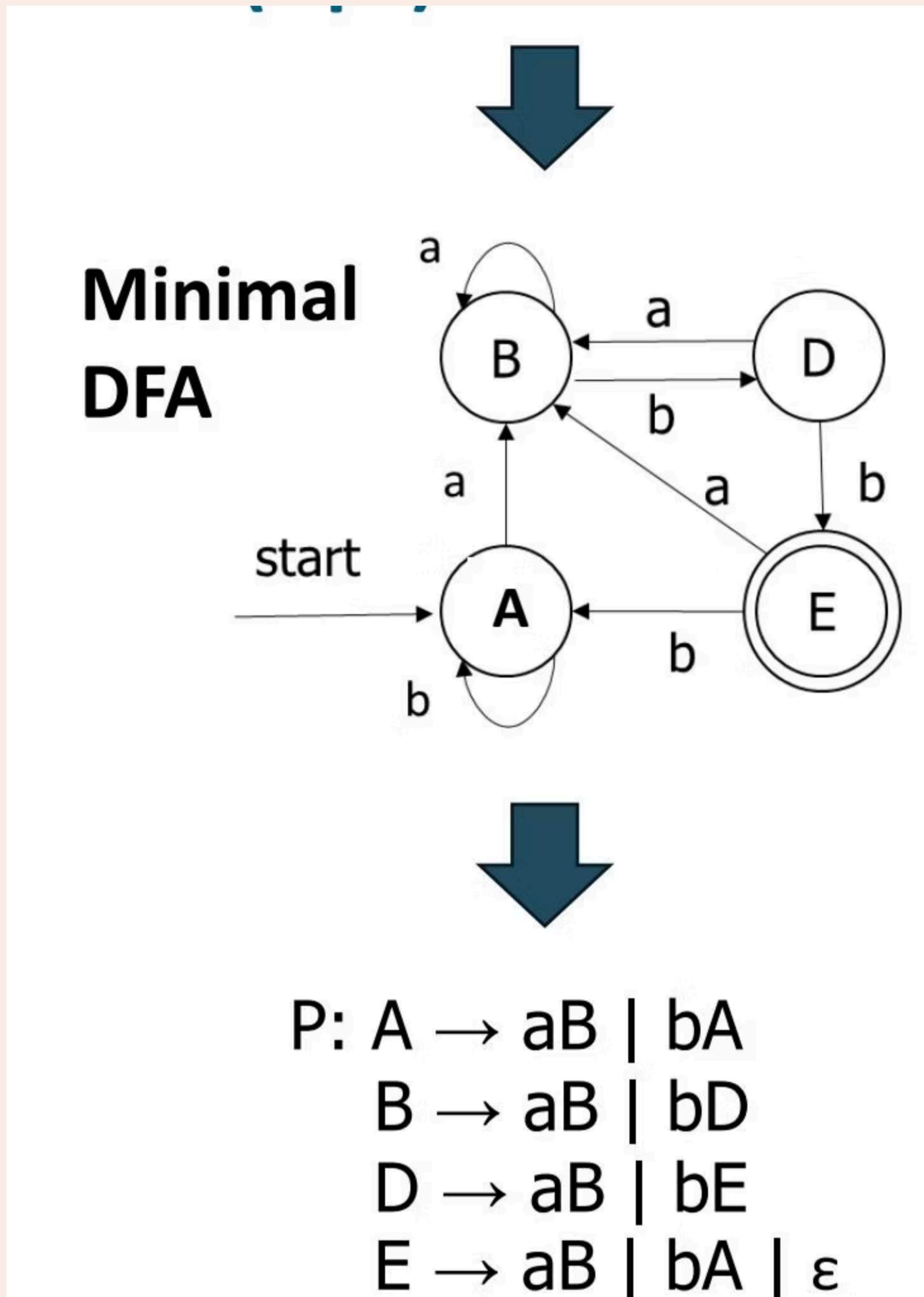
- $(a|b)^*$: a 또는 b가 0번 이상 반복됨
- abb: 그 뒤에 abb가 따옴

예시 문자열: abb, aabb, babb, aaabb, ababb, ...

이 정규 표현식은 유한 오토마타(DFA 또는 NFA)로 표현할 수 있으며, 이는 정규 문법으로도 변환 가능합니다. 정규 문법은 촘스키 계층 구조에서 가장 제한적인 형태로, 컴파일러의 어휘 분석(Lexical Analysis) 단계에서 토큰을 인식하는 데 사용됩니다.

유한 오토마타로 변환할 때, $(a|b)^*abb$ 는 상태 전이를 통해 abb로 끝나는 패턴을 인식하게 됩니다. 이런 오토마타는 컴퓨터 과학에서 패턴 매칭, 텍스트 처리, 프로그래밍 언어 파싱 등 다양한 응용 분야에서 활용됩니다.

정규 언어 표현: NFA



위 이미지는 정규 표현식 $(a|b)^*abb$ 에 대한 비결정적 유한 오토마타(NFA)를 보여줍니다. NFA는 다음과 같은 특징을 가집니다:

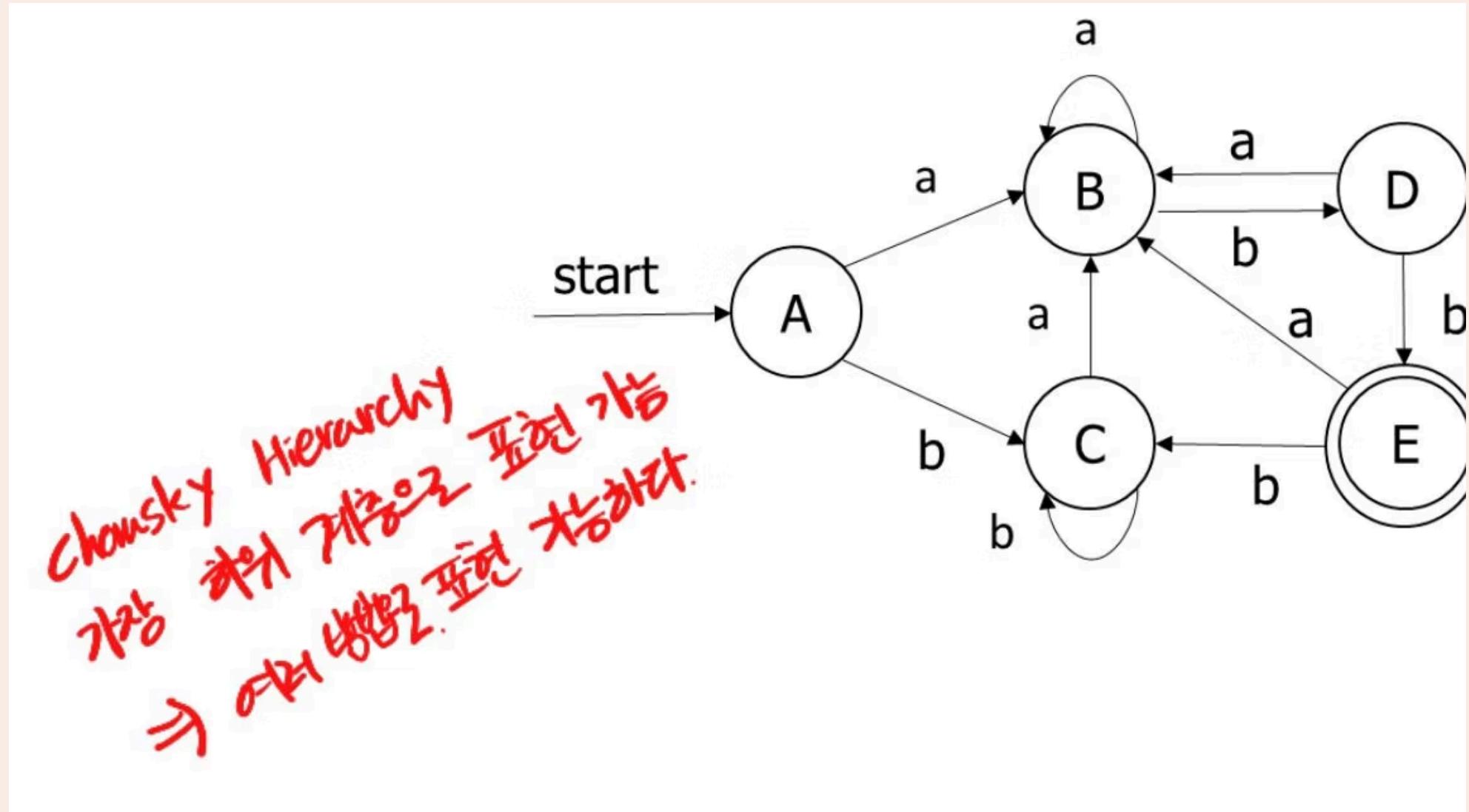
- 상태 0에서 a 또는 b를 읽으면 자기 자신으로 돌아옵니다 ($(a|b)^*$ 부분).
- 상태 0에서 a를 읽으면 상태 1로 이동할 수 있습니다.
- 상태 1에서 b를 읽으면 상태 2로 이동합니다.
- 상태 2에서 b를 읽으면 상태 3(수락 상태)으로 이동합니다.

이 NFA는 abb로 끝나는 모든 문자열을 인식합니다. NFA는 동시에 여러 상태에 있을 수 있는 가능성을 허용하기 때문에, 같은 입력 기호에 대해 여러 전이가 가능합니다.

NFA와 DFA(결정적 유한 오토마타)의 주요 차이점은 비결정성에 있습니다. NFA는 동일한 입력에 대해 여러 경로를 탐색할 수 있는 반면, DFA는 각 상태와 입력 기호에 대해 정확히 하나의 전이만 허용합니다. 모든 NFA는 등가의 DFA로 변환될 수 있으며, 이 과정을 '부분집합 구성법'이라고 합니다.

정규 표현식 $(a|b)^*abb$ 에 대한 NFA는 정규 문법으로도 표현 가능하며, 이는 컴파일러의 어휘 분석기 설계에서 중요한 역할을 합니다. 정규 언어의 세 가지 표현 방법(정규 표현식, 유한 오토마타, 정규 문법)은 서로 변환 가능하고 동일한 언어를 표현할 수 있습니다.

정규 언어의 다양한 표현 방법



정규 언어 $(a|b)^*abb$ 는 다양한 방법으로 표현할 수 있습니다:



정규 표현식 (Regular Expression)

$(a|b)^*abb$



유한 오토마타 (Finite Automata)

DFA(결정적 유한 오토마타) 또는
NFA(비결정적 유한 오토마타)



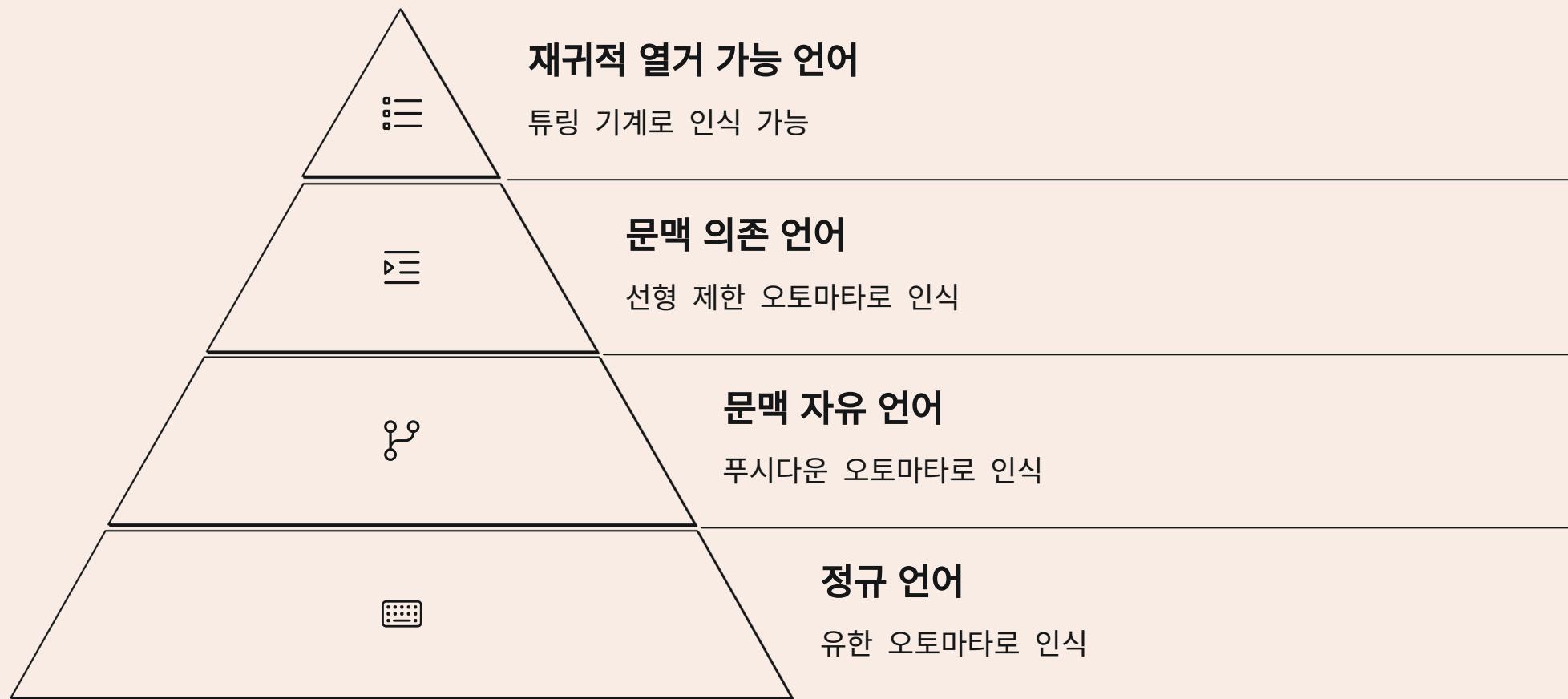
정규 문법 (Regular Grammar)

$S \rightarrow aS \mid bS \mid aA$
 $A \rightarrow bB$
 $B \rightarrow b$

이 세 가지 표현 방법은 모두 동등한 표현력을 가지며, 같은 언어를 정의합니다. 컴파일러의 어휘 분석 단계에서는 이러한 표현 방법을 사용하여 토큰을 인식합니다.

촘스키 계층 구조 도식

촘스키 계층 구조는 형식 언어를 포함 관계에 따라 분류합니다:



이는 모든 정규 언어가 문맥 자유 언어이지만, 그 역은 성립하지 않음을 의미합니다. 마찬가지로, 모든 문맥 자유 언어는 문맥 의존 언어이지만, 그 역은 성립하지 않습니다.

컴파일러 설계에서는 각 단계에 적합한 언어 클래스를 사용합니다:

- 어휘 분석: 정규 언어
- 구문 분석: 문맥 자유 언어
- 의미 분석: 문맥 의존 언어의 일부 특성 사용

문맥 자유 문법 (Context Free Grammar, CFG)

문맥 자유 문법은 컴파일러의 구문 분석 단계에서 중요한 역할을 합니다.



정의

문맥 자유 문법 $G = (VN, VT, P, S)$ 에서 생성 규칙 P 는 $\alpha \rightarrow \beta$ 형태이며, $\alpha \in VN$, $\beta \in V^*$ 입니다.



특징

좌변이 단일 비단말 기호여야 합니다. 이는 문맥에 관계없이 대체가 이루어짐을 의미합니다.



인식 기계

푸시다운 오토마타(Pushdown Automata, PDA)로 인식 가능합니다.



응용

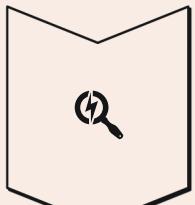
중첩 구조 파싱에 적합하며, 프로그래밍 언어의 구문을 정의하는 데 주로 사용됩니다.

문맥 자유 문법은 정규 문법보다 더 강력한 표현력을 가지며, 중첩된 괄호와 같은 구조를 표현할 수 있습니다. 이는 프로그래밍 언어의 복잡한 구문 구조를 정의하는 데 필수적입니다.

문맥 자유 문법의 한계점으로는 문맥 의존적인 제약 조건(예: 변수 선언 전 사용 금지)을 표현하기 어렵다는 점이 있습니다. 이러한 제약은 문맥 의존 문법이나 의미 분석 단계에서 처리됩니다.

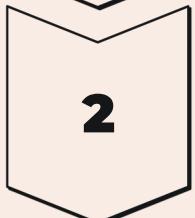
컴파일러의 분석 단계 (Front-end)

컴파일러의 분석 단계는 소스 코드를 분석하여 중간 표현으로 변환하는 과정입니다.



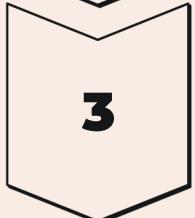
어휘 분석 (Lexical Analysis)

입력 문자열을 토큰(tokens)이라 불리는 "단어"로 분리합니다.



구문 분석 (Syntactic Analysis)

토큰으로부터 구조를 구성하고 이를 파스 트리(parse tree)로 표현합니다.

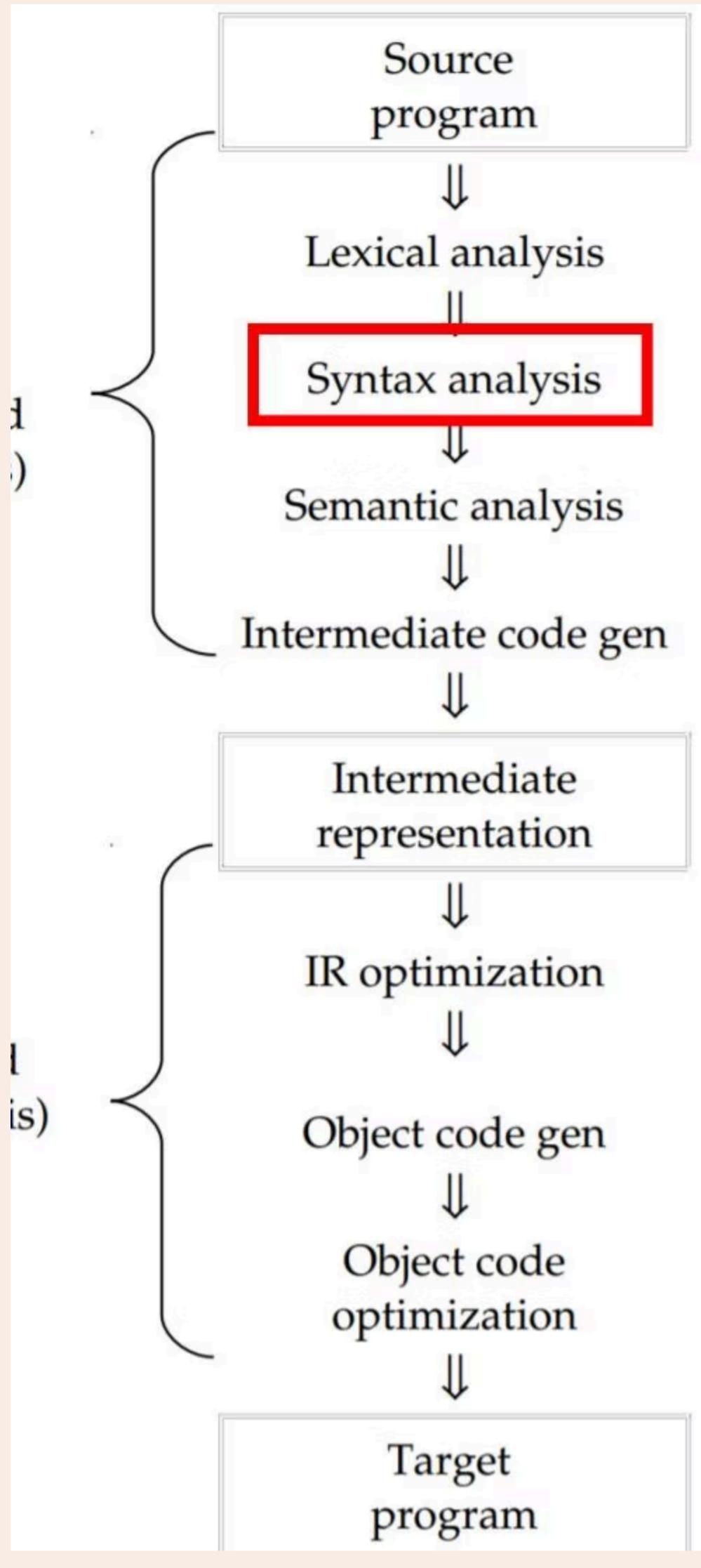


의미 분석 (Semantic Analysis)

"의미"를 결정합니다(예: 타입 체킹). 코드 생성을 준비하고 심볼 테이블(symbol table)과 상호작용합니다.

각 단계는 서로 다른 형식 언어 이론을 기반으로 합니다. 어휘 분석은 정규 언어를, 구문 분석은 문맥 자유 언어를, 의미 분석은 일부 문맥 의존 언어의 특성을 사용합니다.

구문 분석 (Syntax Analysis)



구문 분석은 컴파일러의 두 번째 단계로, 프로그램의 구조를 결정합니다.



파서 (Parser)

문법을 통해 토큰을 그룹화하여 프로그램의 구조를 결정합니다.



문법 (Grammar)

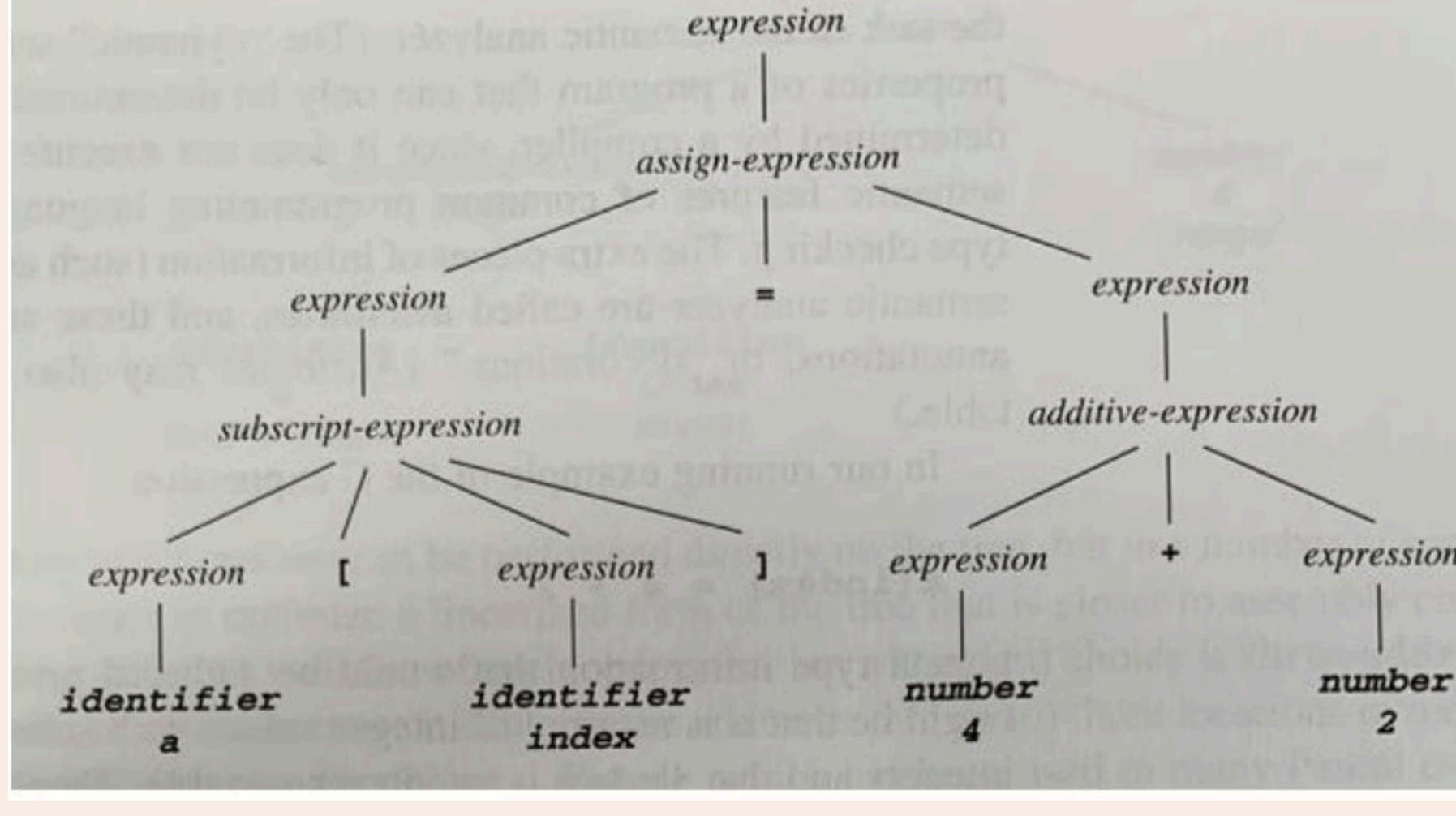
프로그래밍 언어에서 유효한 구조를 정의하는 규칙 집합입니다.



구조적 요소

프로그램의 구조적 요소와 그들 간의 관계를 파악합니다.

예를 들어, `a[index] = 4 + 2`라는 코드는 다음과 같은 파스 트리/구문 트리로 표현될 수 있습니다:



내부 노드는 그들이 나타내는 구조의 이름으로 레이블이 지정되고, 리프 노드는 스캐너에서 가져온 토큰입니다.

구문 분석의 주요 접근법은 하향식(Top-down) 파싱과 상향식(Bottom-up) 파싱으로 나뉩니다. 하향식은 시작 기호에서 시작하여 입력 문자열을 파생하는 방식이며, 상향식은 입력 문자열에서 시작하여 시작 기호로 축소해 나가는 방식입니다.

구문 분석은 푸시다운 오토마타(PDA)로 구현되며, 문맥 자유 문법(CFG)을 기반으로 합니다. 이 단계에서 구문 오류가 감지되면 컴파일러는 적절한 오류 메시지를 생성합니다.

문맥 자유 문법 (CFG)

문맥 자유 문법 $G = (VN, VT, P, S)$ 는 다음과 같이 정의됩니다:



비단말 기호 집합 **VN**

문법의 변수로, 다른 문자열로 대체될 수 있는 기호들입니다.



단말 기호 집합 **VT**

입력 언어의 알파벳으로, 더 이상 대체되지 않는 기호들입니다.



생성 규칙 **P**

문자열 쌍의 유한 집합으로, $\alpha \rightarrow \beta$ 형태를 가집니다.

$\alpha \in VN$, $\beta \in V^*$ (즉, 좌변이 단일 비단말 기호)



시작 기호 **S**

$S \in VN$ 으로, 모든 유도가 시작되는 특별한 비단말 기호입니다.

유형	문법	인식 기계	예시
0	무제한 문법	튜링 머신	모든 계산 가능한 언어
1	문맥 의존 문법 (CSG)	선형 제한 오토마타 (LBA)	제약이 있는 구문 (예: 태입 체킹)
2	문맥 자유 문법 (CFG)	푸시다운 오토마타 (PDA)	중첩 구조 파싱
3	정규 문법 (RG)	유한 상태 오토마타 (FSA)	어휘 분석, 간단한 패턴 매칭

문맥 자유 문법은 프로그래밍 언어의 구문을 정의하는 데 가장 널리 사용되는 문법입니다. 이는 중첩된 구조를 표현할 수 있으며, 효율적인 파싱 알고리즘이 존재하기 때문입니다.

문맥 자유 언어 (Context Free Language, CFL)

문맥 자유 언어는 문맥 자유 문법에 의해 생성된 언어입니다.

문맥 자유 문법 $G = (VN, VT, P, S)$ 가 주어졌을 때,

$$L(G) = \{w \mid w \in VT^* \text{ and } S \Rightarrow^* w\}$$

언어 $L \subseteq \Sigma^*$ 가 문맥 자유 언어인 것은 어떤 문맥 자유 문법 G 에 대해 $L = L(G)$ 인 경우입니다.

문맥 자유 언어의 예시:

- 균형 잡힌 괄호: $\{(), (), ((())), \dots\}$
- 회문(palindrome): $\{a, aa, aba, abba, \dots\}$
- $a^n b^n$: $\{ab, aabb, aaabbb, \dots\}$

이러한 언어들은 정규 언어로는 표현할 수 없지만, 문맥 자유 문법으로는 표현 가능합니다.

문맥 자유 언어는 프로그래밍 언어의 구문 분석, 자연어 처리, XML 구조 분석 등 다양한 분야에서 활용됩니다. 특히 중첩된 구조를 가진 언어를 표현하는 데 적합합니다.

문맥 자유 언어의 인식기로는 푸시다운 오토마타(PDA)가 사용됩니다. 푸시다운 오토마타는 유한 오토마타에 스택을 추가한 계산 모델로, 문맥 자유 언어를 인식할 수 있는 능력을 갖추고 있습니다.

모든 정규 언어는 문맥 자유 언어이지만, 그 역은 성립하지 않습니다. 즉, 문맥 자유 언어는 정규 언어보다 더 표현력이 높은 언어 클래스입니다.

문맥 자유 문법 예시

예시 1: $a^i b^i$

$G1 = (\{S\}, \{a,b\}, P, S)$

$P: S \rightarrow aSb \mid ab$

$L(G1) = \{a^i b^i \mid i \geq 1\}$

유도 예시:

- $S \quad ab$
- $S \quad aSb \quad aabb$
- $S \quad aSb \quad aaSbb \quad aaabbb$

예시 2: 간단한 산술 표현식

$G2 = (\{E\}, \{+, *\}, \{(\), a\}, P, E)$

$P: E \rightarrow E + E \mid E * E \mid (E) \mid a$

유도 예시:

- $E \quad E + E \quad a + E \quad a + a$
- $E \quad E * E \quad (E) * E \quad (E + E) * E \quad (a + a) * a$

이러한 문맥 자유 문법은 중첩된 구조를 표현할 수 있어 프로그래밍 언어의 구문을 정의하는 데 적합합니다.

파스 트리 (Parse Tree / Derivation Tree)



파스 트리는 유도를 그래픽으로 표현한 것입니다.



특징

특정 CFG의 기호로 레이블이 지정된 트리입니다.



리프 노드

단말 기호 또는 ϵ 로 레이블이 지정됩니다.



내부 노드

비단말 기호로 레이블이 지정됩니다.

자식 노드는 부모에 대한 생성 규칙의 우변으로 레이블이 지정됩니다.



루트

시작 기호로 레이블이 지정되어야 합니다.

파스 트리는 문자열이 문법에 따라 어떻게 구성되는지 시각적으로 보여주며, 컴파일러의 구문 분석 단계에서 중요한 역할을 합니다.

CFG의 유도 방법

문맥 자유 문법에서는 두 가지 주요 유도 방법이 있습니다:

최좌 유도 (Leftmost Derivation)

가장 왼쪽에 있는 비단말 기호에 먼저 생성 규칙을 적용합니다.

최우 유도 (Rightmost Derivation)

가장 오른쪽에 있는 비단말 기호에 먼저 생성 규칙을 적용합니다.

예시: $G = (\{E\}, \{+, *, (,), a\}, P, E)$ 에서 $P: E \rightarrow E + E \mid E * E \mid (E) \mid a$

문자열 "a + a * a"의 최좌 유도:

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$$

문자열 "a + a * a"의 최우 유도:

$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * a \Rightarrow E + E * a \Rightarrow a + a * a$$

두 유도 방법 모두 같은 문자열을 생성하지만, 생성 규칙을 적용하는 순서가 다릅니다. 이러한 유도 방식의 차이는 파서 구현에 중요한 영향을 미칩니다.

최좌 유도는 하향식 파서(Top-down parser)에서 사용되고, 최우 유도의 역순은 상향식 파서(Bottom-up parser)에서 활용됩니다. 이 유도 과정은 다음 섹션에서 다룰 파스 트리(Parse Tree)를 통해 시각적으로 표현할 수 있습니다.

C 언어 문법

C 언어의 문법은 문맥 자유 문법으로 정의됩니다. 다음은 C 언어 문법의 주요 구성 요소입니다:

- 선언문 (declarations)
- 문장 (statements)
- 표현식 (expressions)
- 함수 정의 (function definitions)
- 전처리기 지시문 (preprocessor directives)

예를 들어, C 언어의 if 문은 다음과 같은 BNF 표기법으로 표현할 수 있습니다:

```
| if-statement ::= "if" "(" expression ")" statement [ "else" statement ]
```

C 언어의 전체 문법은 매우 복잡하며, 컴파일러는 이 문법을 사용하여 C 프로그램의 구문을 분석합니다. 문맥 자유 문법으로 표현할 수 없는 일부 C 언어의 특성(예: 변수 선언 전 사용 금지)은 의미 분석 단계에서 처리됩니다.

Python 문법

Python 언어의 문법도 문맥 자유 문법으로 정의됩니다. Python 공식 문서 (<https://docs.python.org/3/reference/grammar.html>)에서 전체 문법을 확인할 수 있습니다.

```
statements: statement+
statement: compound_stmt | simple_stmts
statement newline: compound_stmt NEWLINE | simple_stmts NEWLINE | ENDMARKER
simple_stmts:
    simple_stmt !;" NEWLINE # Not needed, there for speedup
    ';' . simple_stmt+ [';'] NEWLINE
# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
    assignment
    | type_alias
    | star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | 'pass'
    | del_stmt
    | yield_stmt
    | assert_stmt
    | "break"
    | 'continue'
    | global_stmt
    | nonlocal_stmt
```

Python 문법의 특징 중 하나는 들여쓰기를 사용하여 코드 블록을 구분한다는 점입니다. 이는 문맥 자유 문법으로 표현하기 어려운 부분이지만, 어휘 분석 단계에서 특별히 처리됩니다.

파싱과 인식 문제

문맥 자유 문법 $G = (VN, VT, P, S)$ 가 주어졌을 때, 문자열 w 를 파싱한다는 것은 $w \in L(G)$ 인지 확인하고, 그렇다면 w 에 대한 유도를 생성하는 것을 의미합니다.



인식 문제

문자열 $w \in \Sigma^*$ 와 알파벳 Σ 에 대한 문맥 자유 언어 L 이 주어졌을 때, w 가 L 에 속하는지 여부를 결정하는 문제입니다.

파싱 알고리즘은 주어진 문자열이 문법에 따라 유효한지 확인하고, 유효하다면 그 구조를 파스 트리 형태로 표현합니다. 이는 컴파일러의 구문 분석 단계에서 핵심적인 작업입니다. 문맥 자유 문법의 파싱에는 CYK 알고리즘, Earley 알고리즘, LL 파서, LR 파서 등 다양한 방법이 사용됩니다.



파싱 문제

문자열 w 가 문법 규칙을 사용하여 시작 기호에서 유도될 수 있는지 확인하고, 가능하다면 해당 유도 과정이나 파스 트리를 생성하는 문제입니다.

모호한 문맥 자유 문법

문맥 자유 문법 $G = (VN, VT, P, S)$ 가 모호하다는 것은 $L(G)$ 에 속하는 어떤 문자열 w 가 둘 이상의 유도(파스 트리)를 가지는 경우를 의미합니다.



모호한 문법

$L(G)$ 에 속하는 어떤 문자열이 둘 이상의 유도를 가지는 문법입니다.



비모호 문법

$L(G)$ 에 속하는 모든 문자열이 유일한 유도를 가지는 문법입니다.

예시: $P: E \rightarrow E + E \mid E * E \mid (E) \mid a$

문자열 "a + a * a"의 유도:

1. $E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$
2. $E \Rightarrow E * E \Rightarrow E * a \Rightarrow E + E * a \Rightarrow E + a * a \Rightarrow a + a * a$

이 문법은 모호합니다. 같은 문자열 "a + a * a"에 대해 두 가지 다른 유도가 가능하기 때문입니다. 이는 연산자 우선순위가 명확하게 정의되지 않았기 때문입니다.

모호한 문법은 프로그래밍 언어 설계에서 피해야 할 특성입니다. 왜냐하면 컴파일러가 코드를 해석할 때 어떤 의미를 선택해야 할지 불분명해지기 때문입니다. 따라서 실제 프로그래밍 언어의 문법은 보통 비모호하게 설계되며, 연산자 우선순위와 결합 법칙을 명확히 정의합니다.

모호한 문법 예시



다음 문법을 살펴보겠습니다:

```
G = ({S}, {a, b}, P, S)
P: S → aSb | SS | ε
```

이 문법이 모호하다는 것을 증명하기 위해 문자열 "aabb"에 대한 두 가지 다른 파스 트리를 그릴 수 있습니다:

1. $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\epsilon bb \Rightarrow aabb$
2. $S \Rightarrow SS \Rightarrow aSbS \Rightarrow aa\epsilon bbS \Rightarrow aabbS \Rightarrow aabb\epsilon \Rightarrow aabb$

위 이미지는 이 두 가지 다른 파스 트리를 보여줍니다. 첫 번째 유도에서는 S 를 aSb 로 두 번 대체한 후 마지막으로 S 를 ϵ (공문자열)로 대체합니다. 두 번째 유도에서는 S 를 SS 로 먼저 대체한 후, 첫 번째 S 를 aSb 로 대체하고 나머지 과정을 진행합니다. 같은 문자열 "aabb"에 대해 두 가지 다른 구조적 해석이 가능하므로, 이 문법은 모호합니다.

이러한 모호성은 프로그래밍 언어 설계에서 중요한 문제로, 컴파일러가 소스 코드를 명확하게 해석할 수 없게 만듭니다. 따라서 실제 프로그래밍 언어의 문법은 대부분 비모호하게 설계됩니다.

모호한/비모호 문법



문자열 "a + a * a"에 대한 예시를 살펴보겠습니다:

모호한 문법

$$P: E \rightarrow E + E \mid E * E \mid (E) \mid a$$

유도 1: $E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$

유도 2: $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow E + E * a \Rightarrow E + a * a \Rightarrow a + a * a$

비모호 문법

$$P: E \rightarrow E + T \mid T$$

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

$$F \rightarrow (E) \mid a$$

이 문법은 연산자 우선순위를 명시적으로 정의하여 모호성을 제거합니다.

비모호 문법은 각 문자열에 대해 유일한 파스 트리를 생성합니다. 이는 프로그래밍 언어의 구문을 명확하게 정의하는 데 중요합니다. 문자열 "a + a * a"는 모호한 문법에서는 두 가지 다른 해석이 가능하지만, 비모호 문법에서는 하나의 유일한 파스 트리만 생성됩니다.

본질적으로 모호한 언어

문맥 자유 언어 L 이 본질적으로 모호하다는 것은 L 에 대한 모든 CFG G 가 모호하다는 것을 의미합니다.



정의

언어를 생성하는 모든 가능한 문맥 자유 문법이 모호한 경우, 그 언어는 본질적으로 모호합니다.



예시

$L = \{0i1j2k \mid i = j \text{ 또는 } j = k; i, j, k \geq 0\}$

이 언어를 생성하는 문법:

$$S \rightarrow AB \mid CD$$

$$A \rightarrow 0A \mid \epsilon$$

$$B \rightarrow 1B2 \mid \epsilon$$

$$C \rightarrow 0C1 \mid \epsilon$$

$$D \rightarrow 2D \mid \epsilon$$



모호성의 원인

$\{0i1j2k \mid i = j = k\}$ 를 유도 할 때 모호성이 발생합니다.

본질적으로 모호한 언어는 어떤 방식으로도 비모호 문법으로 표현할 수 없습니다. 이는 언어 자체의 특성이며, 문법의 설계 방식과는 무관합니다.

본질적 모호성의 증명

언어가 본질적으로 모호한지 증명하는 것은 형식 언어 이론의 중요한 연구 분야입니다.



증명 방법

언어 L 이 본질적으로 모호함을 증명하기 위해서는 모든 가능한 문맥 자유 문법 G 에 대해 G 가 L 을 생성할 때 항상 모호함을 보여야 합니다.

일반적으로 귀류법을 사용하여 L 에 대한 비모호 문법이 존재한다고 가정한 후 모순을 도출합니다.



중요한 예시

다음은 본질적으로 모호한 것으로 증명된 대표적인 언어들입니다:

- $\{aibjck \mid i=j \text{ 또는 } j=k, i, j, k \geq 0\}$
- $\{ww \mid w \in \{a, b\}^*\}$ (같은 문자열의 반복)
- 여러 프로그래밍 언어의 표현식 부분집합

본질적으로 모호한 언어의 존재는 형식 언어 처리의 한계를 보여줍니다. 이는 모든 언어가 효율적으로 파싱될 수 없음을 의미하며, 이러한 이론적 한계는 프로그래밍 언어 설계 및 컴파일러 구현에 중요한 영향을 미칩니다.

파서 생성과 구문 분석

컴파일러의 구문 분석 단계는 문법 이론을 실제 코드 분석에 적용합니다.



파서 생성 도구

LL, LR, LALR 등의 파싱 알고리즘을 기반으로 합니다.

Yacc, Bison, ANTLR과 같은 도구가 문법 명세로부터 자동으로 파서를 생성합니다.



구문 분석 트리

소스 코드의 구조적 표현을 생성합니다.

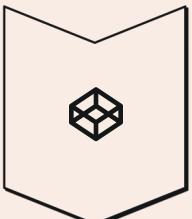
추상 구문 트리(AST)는 의미 분석과 코드 생성의 기초가 됩니다.

파서는 문맥 자유 문법을 기반으로 하지만, 실제 프로그래밍 언어는 문맥 의존적 특성도 가지고 있습니다. 이러한 문맥 의존적 측면은 구문 분석 이후의 의미 분석 단계에서 처리됩니다.

효율적인 파서 설계는 컴파일러의 성능에 직접적인 영향을 미치며, 문법의 모호성을 최소화하는 것이 중요합니다. 구문 분석 단계가 완료되면 다음 단계인 의미 분석과 최적화 단계로 넘어갑니다.

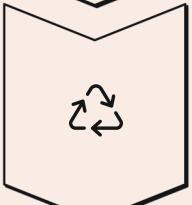
컴파일러 최적화 기법

컴파일러는 소스 코드를 기계어로 변환하는 과정에서 다양한 최적화 기법을 적용합니다.



상수 폴딩 (Constant Folding)

컴파일 시간에 계산 가능한 상수 표현식을 미리 계산합니다. 예: $3 + 4$ 는 7로 대체됩니다.



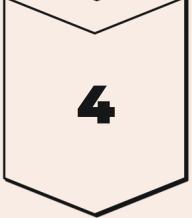
공통 부분식 제거 (Common Subexpression Elimination)

동일한 표현식이 여러 번 계산되는 경우, 한 번만 계산하고 그 결과를 재사용합니다.



루프 최적화 (Loop Optimization)

루프 언롤링, 루프 융합, 루프 불변 코드 이동 등의 기법을 사용하여 루프의 효율성을 향상시킵니다.



인라인 확장 (Inlining)

함수 호출을 함수 본문으로 대체하여 함수 호출 오버헤드를 제거합니다.

이러한 최적화 기법은 컴파일러의 중간 단계와 후반 단계에서 적용되며, 생성된 코드의 성능을 크게 향상시킬 수 있습니다. 최적화 수준은 컴파일러 옵션을 통해 조절할 수 있습니다.

컴파일러 백엔드 (Back-end)

컴파일러의 백엔드는 중간 표현(IR)을 목표 기계어로 변환하는 역할을 합니다.

코드 최적화 (Code Optimization)

중간 표현을 분석하고 최적화하여 더 효율적인 코드를 생성합니다. 이 단계에서는 다양한 최적화 기법이 적용됩니다.

코드 생성 (Code Generation)

최적화된 중간 표현을 목표 기계의 명령어로 변환합니다. 이 과정에서 레지스터 할당, 명령어 선택, 명령어 스케줄링 등이 수행됩니다.

목적 코드 생성 (Object Code Generation)

기계어 코드를 목적 파일 형식으로 출력합니다. 이 파일은 링커에 의해 다른 목적 파일과 결합되어 실행 파일이 됩니다.

컴파일러의 백엔드는 목표 기계의 아키텍처에 따라 달라지며, 같은 프론트엔드를 사용하더라도 다른 백엔드를 사용하여 다양한 플랫폼에 대한 코드를 생성할 수 있습니다.

컴파일러 구현 도구

렉서 생성기

- Lex/Flex: 정규 표현식을 기반으로 어휘 분석기를 생성하는 도구
- JFlex: Java 기반의 렉서 생성기

파서 생성기

- Yacc/Bison: 문맥 자유 문법을 기반으로 구문 분석기를 생성하는 도구
- ANTLR: 다양한 언어에 대한 파서 생성기
- JavaCC: Java 기반의 파서 생성기

컴파일러 프레임워크

- LLVM: 모듈식 컴파일러 인프라 스트럭처
- GCC: GNU 컴파일러 컬렉션

이러한 도구들은 컴파일러 개발을 크게 단순화하며, 특히 렉서와 파서 생성기는 정규 표현식과 문맥 자유 문법을 직접 구현하는 대신 자동으로 코드를 생성해줍니다. 이를 통해 개발자는 언어의 의미론적 측면에 더 집중할 수 있습니다.

컴파일러 설계 실습: 간단한 계산기

간단한 계산기 언어를 위한 컴파일러를 설계해 봅시다.

어휘 규칙 (정규 표현식)

NUMBER: [0-9]+(\.[0-9]+)?

PLUS: \+

MINUS: -

TIMES: *

DIVIDE: /

LPAREN: \(

RPAREN: \)

WHITE SPACE: [\t\n]+

문법 규칙 (CFG)

expr: term ((PLUS | MINUS) term)*

term: factor ((TIMES | DIVIDE) factor)*

factor: NUMBER | LPAREN expr RPAREN

이 문법은 사칙연산의 우선순위를 올바르게 처리합니다. 곱셈과 나눗셈이 덧셈과 뺄셈보다 우선순위가 높으며, 괄호를 사용하여 우선순위를 변경할 수 있습니다.

예를 들어, "3 + 4 * 2"는 "3 + (4 * 2)" = "3 + 8" = "11"로 계산되고, "(3 + 4) * 2"는 "(7) * 2" = "14"로 계산됩니다.

컴파일러 설계 실습: 파스 트리 생성

앞서 정의한 계산기 언어의 문법을 사용하여 표현식 "3 + 4 * 2"의 파스 트리를 생성해 봅시다.

어휘 분석

입력 문자열 "3 + 4 * 2"를 토큰으로 분리합니다:

```
[NUMBER(3), PLUS, NUMBER(4),  
TIMES, NUMBER(2)]
```

구문 분석

토큰 시퀀스를 파스 트리로 변환합니다:

```
expr  
|  
+-- term (3)  
| |  
| +-- factor (3)  
| |  
| +-- NUMBER(3)  
  
|  
+-- PLUS  
  
|  
+-- term (4 * 2)  
|  
+-- factor (4)  
| |  
| +-- NUMBER(4)  
  
|  
+-- TIMES  
  
|  
+-- factor (2)  
|  
+-- NUMBER(2)
```

계산

파스 트리를 순회하며 표현식을 계산합니다:

$\text{term}(4 * 2) = 8$

$\text{expr}(3 + 8) = 11$

이 예시는 문맥 자유 문법이 어떻게 표현식의 구조를 정의하고, 파서가 어떻게 이 구조를 파스 트리로 표현하는지 보여줍니다. 파스 트리는 연산자 우선순위를 고려하여 곱셈이 먼저 수행되도록 구조화되었으며, 이는 문법 규칙에 의해 자연스럽게 처리됩니다.

컴파일러 설계 실습: 의미 분석

간단한 프로그래밍 언어에서 의미 분석의 역할을 살펴봅시다.

타입 체킹

변수와 표현식의 타입을 확인하고, 타입 호환성을 검사합니다.

```
int a = 5;  
float b = 3.14;  
a = b; // 타입 오류: float을 int에 할당
```

심볼 테이블

변수, 함수, 타입 등의 식별자 정보를 저장하고 관리합니다.

```
int foo(int x) {  
    return x + 1;  
}  
  
void main() {  
    int a = foo(5); // 심볼 테이블에서 foo 함수 검색  
    int b = bar(a); // 오류: bar 함수가 정의되지 않음  
}
```

의미 분석은 문맥 자유 문법으로 표현할 수 없는 언어의 제약 조건을 검사합니다. 예를 들어, 변수는 사용하기 전에 선언해야 한다는 규칙이나, 함수 호출 시 인자의 수와 타입이 일치해야 한다는 규칙 등이 있습니다.

이러한 제약 조건은 문맥 의존적이며, 문맥 자유 문법만으로는 표현할 수 없습니다. 따라서 컴파일러는 구문 분석 이후에 별도의 의미 분석 단계를 통해 이러한 제약 조건을 검사합니다.

의미 분석의 결과물로 추상 구문 트리(AST)는 더 풍부한 정보를 갖게 되며, 이는 이후의 중간 코드 생성 단계에서 활용됩니다. 의미 분석 단계에서 오류가 발견되지 않으면, 컴파일러는 다음 단계인 중간 코드 생성으로 진행됩니다.

아래는 의미 분석 과정에서 추가로 수행되는 검사들입니다:



스코프 분석

변수와 함수의 가시성 범위를 검사하고, 중첩된 블록에서의 이름 충돌을 해결합니다.



흐름 제어 검사

break, continue 문이 반복문 내에서만 사용되는지, return 문이 함수 내에서 적절하게 사용 되는지 확인합니다.



초기화 검사

변수가 사용되기 전에 초기화되었는지 확인하여 미정의 동작을 방지합니다.

컴파일러 설계 실습: 코드 생성

간단한 표현식에 대한 코드 생성 예시를 살펴봅시다.

소스 코드

```
int a = 3;  
int b = 4;  
int c = a + b * 2;
```

중간 표현 (IR)

```
t1 = 3  
t2 = 4  
t3 = t2 * 2  
t4 = t1 + t3
```

어셈블리 코드 (x86)

```
mov eax, 3 ; a = 3  
mov ebx, 4 ; b = 4  
mov ecx, 2 ; 상수 2  
imul ecx, ebx ; ecx = b * 2  
add eax, ecx ; eax = a + (b * 2)  
mov c, eax ; c = eax
```

코드 생성 단계에서는 파스 트리나 중간 표현을 목표 기계의 명령어로 변환합니다. 이 과정에서 레지스터 할당, 명령어 선택, 최적화 등이 수행됩니다.

위 예시에서는 표현식 "a + b * 2"가 어떻게 x86 어셈블리 코드로 변환되는지 보여줍니다. 곱셈이 덧셈보다 우선순위가 높기 때문에, 먼저 "b * 2"를 계산한 후 그 결과를 "a"와 더합니다.

코드 생성기는 중간 표현(IR)의 각 명령어를 대상 아키텍처에 맞는 하나 이상의 기계어 명령으로 변환합니다. 이 과정에서 다음과 같은 요소를 고려해야 합니다:

- 레지스터 할당: 변수와 임시 값을 저장할 레지스터를 효율적으로 할당
- 명령어 선택: 주어진 연산을 수행하는 가장 효율적인 기계어 명령 선택
- 주소 지정 모드: 메모리 접근을 위한 최적의 주소 지정 방식 결정
- 제어 흐름: 조건문, 반복문 등의 제어 구조를 기계어로 변환

최종적으로 생성된 기계어 코드는 실행 파일로 패키징되어 운영 체제에서 로드되고 실행될 수 있습니다. 코드 생성은 컴파일러의 마지막 단계로, 앞선 단계들(어휘 분석, 구문 분석, 의미 분석, 최적화)의 결과물을 활용하여 실행 가능한 코드를 만들어냅니다.

컴파일러 최적화 예시

컴파일러가 적용할 수 있는 다양한 최적화 기법의 예시를 살펴봅시다.

최적화 전 코드

```
int sum = 0;  
for (int i = 0; i < 100; i++) {  
    sum = sum + i * 2;  
}
```

상수 폴딩 및 루프 불변 코드 이동

```
int sum = 0;  
for (int i = 0; i < 100; i++) {  
    int temp = i * 2; // 루프 내에서  
    매번 계산  
    sum = sum + temp;  
}
```

루프 언롤링

```
int sum = 0;  
for (int i = 0; i < 100; i += 4) {  
    sum = sum + (i << 1);  
    sum = sum + ((i+1) << 1);  
    sum = sum + ((i+2) << 1);  
    sum = sum + ((i+3) << 1);  
}
```

↓

```
int sum = 0;  
for (int i = 0; i < 100; i++) {  
    sum = sum + (i << 1); // i * 2를  
    비트 시프트로 최적화  
}
```

이러한 최적화 기법은 코드의 실행 속도를 향상시키거나 코드 크기를 줄이는 데 도움이 됩니다. 컴파일러는 최적화 수준에 따라 다양한 기법을 적용하며, 개발자는 컴파일러 옵션을 통해 최적화 수준을 조절할 수 있습니다.

컴파일러 설계 문제: 모호한 문법 해결

다음 모호한 문법을 비모호 문법으로 변환해 봅시다.

```
E → E + E | E * E | (E) | id
```

이 문법은 연산자 우선순위가 명확하지 않아 모호합니다. 예를 들어, "id + id * id"는 두 가지 다른 방식으로 파싱될 수 있습니다.

비모호 문법으로 변환:

```
E → E + T | T
T → T * F | F
F → (E) | id
```

이 비모호 문법은 다음과 같은 우선순위를 명시적으로 정의합니다:

- 괄호 내의 표현식이 가장 높은 우선순위를 가집니다.
- 곱셈(*)이 두 번째로 높은 우선순위를 가집니다.
- 덧셈(+)이 가장 낮은 우선순위를 가집니다.

이제 "id + id * id"는 유일한 파스 트리를 가지며, 이는 "(id + (id * id))"로 해석됩니다.

이 문법 변환의 핵심은 각 비단말 기호(E, T, F)가 특정 우선순위 수준을 나타내도록 하는 것입니다. 이렇게 하면 파서가 연산자 우선순위 규칙에 따라 표현식을 올바르게 평가할 수 있습니다. 또한, 좌측 연관성(left-associativity)을 갖도록 문법을 구성하여 "a + b + c"와 같은 표현식이 "((a + b) + c)"로 해석되도록 합니다.

컴파일러 설계 문제: 파스 트리 비교

좌 재귀를 제거한 후, 원래 문법과 변환된 문법이 동일한 언어를 생성하는지 파스 트리를 통해 확인해 봅시다.

원래 문법: $E \rightarrow E + T \mid T$

변환된 문법: $E \rightarrow T E'$
 $E' \rightarrow + T E' \mid \epsilon$

입력 문자열 "id + id + id"에 대한 파스 트리 비교:

- 원래 문법의 파스 트리는 좌측 결합(left-associative)을 보여줍니다:

$E \rightarrow E + T \rightarrow E + T + T \rightarrow T + T + T \rightarrow id + id + id$

이는 "(id + id) + id"로 해석됩니다.

- 변환된 문법의 파스 트리:

$E \rightarrow T E' \rightarrow id E' \rightarrow id + T E' \rightarrow id + id E' \rightarrow id + id + T E' \rightarrow id + id + id E' \rightarrow id + id + id \epsilon$

이 역시 "(id + id) + id"로 해석되어 원래 문법과 동일한 결과를 보여줍니다.

이처럼 좌 재귀 제거는 문법의 구조를 변경하지만, 생성되는 언어와 표현식의 의미는 보존합니다. 다만 파스 트리의 모양은 달라질 수 있습니다.

컴파일러 설계 문제: 파스 트리 구성

다음 문법과 입력 문자열에 대한 파스 트리를 구성해 봅시다.

```
S → if E then S else S | if E then S | id  
E → id
```

입력 문자열: "if id then if id then id else id"

이 문법은 모호합니다. "else" 절이 어떤 "if"와 연관되는지 명확하지 않기 때문입니다. 이를 "dangling else" 문제라고 합니다.

가능한 두 가지 파스 트리:

1. 첫 번째 해석 (else가 가장 가까운 if와 연관):

```
S  
/|\  
/ | \  
/ | \  
if E then  
| \  
id S  
/|\  
/ | \  
/ | \  
if E then  
| \  
id S  
/|\  
/ \  
S S  
| |  
id id
```

2. 두 번째 해석 (else가 첫 번째 if와 연관):

```
S  
/|\  
/ | \  
/ | \  
if E then  
| \  
id S  
/|\  
/ \  
S S  
/|\ |  
/ | \ id  
if E then  
| \  
id S  
|  
id
```

대부분의 컴파일러는 "else"가 가장 가까운 이전 "if"와 연관된다는 규칙을 사용하여 첫 번째 해석을 선택합니다. 이 규칙은 "가장 가까운 if와 매치" 또는 "else-matching" 규칙이라고 합니다.