

컴파일러 개요 및 어휘 분석의 역할



컴파일러와 어휘 분석에 대한 핵심 개념을 이해하고 실제 적용 방법을 배워봅시다.

컴파일러란 무엇인가?

컴파일러의 정의

컴파일러(Compiler)는 고급 언어(High-level language)로 작성된 코드를 실행 가능한 코드, 즉 기계 코드(Machine code)인 저급 언어(Low-level language)로 번역하는 프로그램입니다.

고급 언어는 사람이 작성하고 읽기 쉬우며 기계에 독립적입니다. 반면 저급 언어는 작성하고 읽기 어렵고 기계에 종속적입니다.

컴파일러가 필요한 이유는 고급 언어로 프로그램을 작성하기 위함입니다.

컴파일러의 번역 과정



분석(Analysis, Front-end)

- 어휘 분석(Lexical)
- 구문 분석(Syntactic)
- 의미 분석(Semantic)

이 단계를 거쳐 중간 코드(Intermediate code, IR)가 생성됩니다.

합성(Synthesis, Back-end)

- 최적화(Optimization)
- 기계 코드 생성(Machine code generation)

어휘 분석(Lexical Analysis)의 역할

어휘 분석(Lexical Analysis)은 컴파일러 분석 단계의 첫 번째 과정입니다.



어휘 분석기의 명칭

어휘 분석기는 렉서(Lexer) 또는 스캐너(Scanner)라고도 불립니다.



주요 역할

주요 역할은 입력 문자열, 즉 소스 코드(source code)를 읽고, 문자들을 의미 있는 단위인 "토큰(token)"이라는 '단어'들로 묶는 것입니다.



출력 형태

렉서의 출력은 파서(Syntactic Analysis)의 입력으로 사용되는 토큰들의 스트림(stream) 형태입니다.

토큰의 개념과 구성 요소

토큰(Token)의 정의

토큰(Token)은 프로그래밍 언어에서 구문적으로 가장 작은 유 효 단위입니다.

인간 언어의 명사, 동사 등과 같이, 프로그래밍 언어에서는 식별자(Identifier), 키워드(Keyword), 화이트스페이스(White-space), 숫자(Number) 등 구문적 범주(Syntactic Category)로 분류될 수 있습니다.

토큰은 일련의 문자열에 해당합니다.

예를 들어, 소스 코드 `int foo = 100;`에서 `int`는 `<keyword, int>` 토큰에 해당하는 렉시, `foo`는 `<id, "foo">` 토큰에 해당하는 렉시, `=`는 `<op, "=">` 토큰에 해당하는 렉시, `100`은 `<int, 100>` 토큰에 해당하는 렉시입니다.

렉심과 패턴의 이해

렉심(Lexeme)의 정의

렉심(Lexeme)은 소스 코드에 나타나는 토큰의 실제 인스턴스, 즉 특정 문자열입니다.

렉서는 보통 <토큰 이름, 속성 값> 형태로 토큰을 반환합니다.

패턴(Pattern)의 정의

패턴(Pattern)은 토큰을 정의하는 규칙 또는 정규 표현식입니다. 토큰은 패턴에 의해 정의되는 문자열의 집합에 해당합니다.

렉서는 프로그램의 부분 문자열들을 그 역할에 따라 분류하는 역할을 합니다.

렉서를 설계할 때는 먼저 관심 있는 모든 항목을 기술하는 토큰 집합을 정의하고, 각 토큰에 속하는 문자열을 패턴으로 기술합니다. 토큰의 선택은 언어에 따라 달라집니다.

형식 언어(Formal Language)의 기본 구성 요소

정규 표현식과 유한 상태 기계는 형식 언어 이론에 기반합니다.

알파벳(Alphabet, Σ)

언어의 문자열을 구성하는 데 사용되는 유한한 기호(문자)들의 집합입니다.

예: {a, b}, {0, 1}

문자열(String) 또는 단어(Word)

Σ 에서 가져온 기호들의 유한한 순서열입니다.

예: {a, b}에서 "abab", "aa", "aabb"

언어(Language, L)

특정 규칙을 따르는 문자열들의 집합입니다.

예: 길이 ≥ 2 인 이진 문자열 집합
{"01", "10", "110", "1010"}

문자열에 대한 연산



이어붙이기 (Concatenation, \cdot)

두 문자열 α 와 β 를 합치는 연산입니다. $\alpha \cdot \beta$ 는 $\alpha\beta$ 로 표기합니다. 결합법칙은 성립하지만 $((x \cdot y) \cdot z) = x \cdot (y \cdot z)$, 교환법칙은 성립하지 않습니다 ($x \cdot y \neq y \cdot x$).

예: "a" \cdot "b" = "ab"



빈 문자열(Empty String, ε 또는 ϵ)

길이가 0인 문자열입니다. $\varepsilon \neq \emptyset$ (빈 집합) $\neq \{\varepsilon\}$ (빈 문자열을 원소로 가지는 집합)임을 유의해야 합니다.

어떤 문자열에 빈 문자열을 이어붙여도 자기 자신입니다 ($\varepsilon \sqcup "abc" = "abc"$).



거듭제곱 (Exponentiation)

문자열 x 의 반복입니다. $x^0 = \varepsilon$ 이고, $n > 0$ 일 때 $x^n = x \cdot x^{n-1}$ 입니다.

예: $x = "hello"$ 일 때 $x^3 = "hellohellohello"$, $x^0 = \varepsilon$

언어에 대한 연산



이어붙이기 (Concatenation, \cdot)

두 언어 X 와 Y 의 이어붙이기 $X \cdot Y$ 는 X 에 속하는 문자열과 Y 에 속하는 문자열을 하나씩 이어붙인 모든 결과의 집합입니다 ($X \cdot Y = \{x \cdot y \mid x \in X, y \in Y\}$).

예: $X = \{"a", "bc"\}$, $Y = \{"x", "yz"\}$ 일 때 $X \cdot Y = \{"ax", "ayz", "bcx", "bcyz"\}$



거듭제곱 (Exponentiation)

언어 X 의 반복입니다. $X^2 = X \cdot X$ 이고, $X^0 = \{\epsilon\}$ 입니다.

예: $X = \{"a", "bc"\}$ 일 때 $X^2 = \{"aa", "abc", "bca", "bcbc"\}$.
 $X = \{\epsilon, "a"\}$ 일 때 $X^2 = \{\epsilon, "a", "aa"\}$



합집합(Union, \cup)

두 언어 X 와 Y 의 합집합 $X \cup Y$ 는 X 또는 Y 에 속하는 모든 문자열의 집합입니다 ($X \cup Y = \{u \mid u \in X \text{ or } u \in Y\}$).

예: $X = \{"a", "bc"\}$, $Y = \{"x", "yz"\}$ 일 때 $X \cup Y = \{"a", "bc", "x", "yz"\}$

클레이니 연산

클레이니 스타(Kleene's Star, L^*)

주어진 언어 L 의 문자열들을 0회 이상 이어붙여 만들 수 있는 모든 가능한 문자열의 집합이며, 빈 문자열 ϵ 를 포함합니다.

$$L^* = L^0 \cup L^1 \cup L^2 \cup \dots = \bigcup_{i=0}^{\infty} L^i \text{로 정의됩니다.}$$

예: $X = \{"a", "b"\}$ 일 때 $X^* = \{\epsilon, "a", "b", "aa", "ab", "ba", "bb", "aaa", \dots\}$

클레이니 플러스(Kleene's Plus, L^+)

L^* 와 유사하지만 빈 문자열을 제외합니다.

$$L^+ = L^1 \cup L^2 \cup \dots = \bigcup_{i=1}^{\infty} L^i \text{로 정의됩니다.}$$

($L^+ = L \cdot L^*$ 와 같습니다)

정규 표현식(Regular Expression, Regexp)

정규 표현식(Regexp)은 문자열 패턴을 형식적으로 정의하는 데 사용됩니다.



이어붙이기
(Concatenation)

두 패턴을 나란히 놓습니다. · 기호
는 흔히 생략됩니다.

예: ab (a 다음에 b)



합집합(Union, |)

주어진 패턴 중 하나를 의미합니
다.

예: a|xy ("a" 또는 "xy")



그룹화()

표현식의 일부를 묶습니다.

예: (ab)|(cd) ("ab" 또는 "cd")

정규 표현식 연산자



클레이니 스타(*)

앞의 패턴이 0회 이상 반복됨을 의미합니다.

예: a^* ("", "a", "aa", "aaa", ...)



클레이니 플러스(+)

앞의 패턴이 1회 이상 반복됨을 의미합니다.

예: $(ab)^+$ ("ab", "abab", "ababab", ...). $r^+ = rr^*$ 와 같습니다.



선택 사항(?)

앞의 패턴이 0회 또는 1회 나타남을 의미합니다.

예: $a?$ ("", "a"). $\text{color}?$ ("color" 또는 "colour"). $r? = r \mid \epsilon$ 와 같습니다.

정규 표현식 연산자 (계속)



문자 클래스(())

괄호 안의 문자 집합 중 하나가 나타남을 의미합니다.

예: [abc] ("a", "b", 또는 "c"). [0-9] (0부터 9까지의 숫자 중 하나). [a-zA-Z] = a|b|c|...|z 와 같습니다.

ASCII 코드 때문에 [a-zA-Z]는 [a-z]와 [A-Z]를 각각 지정해야 합니다.



와일드카드(.)

어떤 단일 문자를 나타낼 수 있습니다.

예: a.b ("acb", "a1b", "a_b", ...)

정규 정의(Regular Definition)

편리성을 위해 정규 표현식에 이름을 부여하고, 이 이름을 사용하여 다른 정규 표현식을 정의하는 방법입니다.

형식

$d_1 \rightarrow r_1, d_2 \rightarrow r_2, \dots, d_n \rightarrow r_n$ 과 같습니다.

d_i 는 고유한 이름이고, r_i 는 $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ 에 있는 기호들로 구성된 정규 표현식입니다.

예시

letter $\rightarrow [a-zA-Z]$

digit $\rightarrow [0-9]$

id $\rightarrow \text{letter}(\text{letter}|\text{digit})^*$

정규 표현식은 프로그래밍 언어 컴파일러의 토큰 집합을 포함하여 많은 유용한 언어를 설명하는 데 사용됩니다.

정규 언어(Regular Language)

정규 언어(**Regular Language**)는 유한 상태 기계(Finite State Machine, FSM)에 의해 인식될 수 있는 형식 언어의 한 유형입니다.

정규 표현식(Regexp)을 사용하여 표현될 수 있습니다.

정규 언어의 정의

알파벳 Σ 상의 정규 언어는 다음 규칙에 의해 정의됩니다.

- \emptyset (공집합)은 정규 언어입니다.
- $\{a\}$ ($a \in \Sigma$)는 정규 언어입니다.
- $\{\epsilon\}$ 는 정규 언어입니다.
- R 과 S 가 정규 언어이면, $R \cup S$, $R \cdot S$, R^* , R^+ 도 정규 언어입니다.



정규 표현식과 정규 언어의 관계

- 어떤 정규 표현식이 생성하는 문자열 집합은 항상 정규 언어입니다.
- 모든 정규 언어는 정규 표현식으로 표현될 수 있습니다.
- 이는 정규 표현식과 정규 언어, 그리고 이를 인식하는 유한 상태 기계 (DFA/NFA)가 모두 등가적인 표현 능력을 가짐을 의미합니다.

비정규 언어(Non-Regular Languages)

모든 언어가 정규 언어는 아닙니다. 정규 표현식이나 유한 상태 기계로 기술될 수 없는 언어도 존재합니다.

균형 잡힌 괄호들의 집합

$\{(), (), ((())), \dots\}$

이러한 중첩 구조는 문맥 자유 문법(Context-Free Grammar, CFG)으로 기술될 수 있습니다.

복잡한 패턴 매칭

w가 a와 b로 된 문자열일 때 $\{wcw \mid w \in \{a, b\}^*\}$ ($\{aca, abcab, abacaba, \dots\}$)

이 언어는 문맥 자유 문법으로도 기술될 수 없습니다.

유한 상태 기계(Finite State Machine, FSM)

유한 상태 기계(Finite Automata)는 계산의 수학적 모델이며, 정규 표현식을 실제로 구현하는 해결책 중 하나입니다.



기본 개념

입력을 받아 상태를 변경하고 잠재적으로 출력을 생성하는 추상 기계입니다.



표현 방식

알파벳 Σ 의 문자로 레이블된 에지(edge)를 가진 유한 전이 다이어그램으로 표현됩니다.



상태 구성

하나의 시작 상태를 가져야 하고, 0개 이상의 최종 상태를 가질 수 있습니다.

전이 다이어그램(Transition Diagram)

유한 상태 기계의 시각적 표현으로, 소스 코드를 스캔하고 토큰을 인식하는 데 사용됩니다.

상태(State)

원으로 표시되며 시작/중간/최종(수락) 상태가 있습니다.

- 최종(수락) 상태(Final/Accept State): 유효한 토큰이 인식되는 상태입니다.
- 실패 상태(Failure State): 유효하지 않은 입력 순서를 나타내는 상태입니다 (선택 사항).

전이(Transition)

화살표로 표시되며, 입력 문자에 따라 상태가 어떻게 변하는지를 나타냅니다.

입력 버퍼링(Input Buffering)

어휘 분석에서 입력을 효율적으로 읽고 처리하기 위한 기법입니다.



포인터 사용

`lexeme_beginning`과
`forward_pointer` 두 개의 포인터
를 사용합니다.

`lexeme_beginning`은 토큰의 시
작 위치를 표시하고,
`forward_pointer`는 문자를 스캔
하기 위해 앞으로 이동합니다 (룩
어헤드).



처리 과정

초기에는 두 포인터 모두 다음 렉
심의 첫 문자 위치를 가리킵니다.

`forward_pointer`가 스캔하고, 렉
심이 발견되면 렉심의 마지막 문자
를 가리키도록 설정됩니다.

렉심 처리 후에는 두 포인터 모두
렉심 바로 뒤의 문자를 가리키도록
설정됩니다.



실패 처리

전이 다이어그램 기반 렉서 구현
시 사용되며, 실패 시
`forward_pointer`를
`lexeme_beginning`으로 되돌리
는 등의 동작을 수행합니다.

렉심 발견 후 `retract(n)`가 필요한
경우 `forward_pointer`를 n 문자
뒤로 이동시킵니다 (룩어헤드).

결정적 유한 오토마타(Deterministic Finite Automata, DFA)

유한 상태 기계의 특정 유형입니다.

형식적 정의

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

- Σ : 알파벳
- Q : 유한한 상태 집합
- δ : 전이 함수 ($Q \times \Sigma \rightarrow Q$). 각 상태와 입력 기호 쌍에 대해 다음 상태가 유일하게 결정됩니다 (결정적)
- q_0 : 시작 상태
- F : 최종(수락) 상태 집합

특징

- DFA는 정규 언어를 인식하는 언어 인식기입니다.
- 어떤 정규 표현식에 대해서든 적어도 하나의 동등한 DFA가 존재합니다.
- 주어진 문자열이 정규 표현식에 해당하는지 여부를 수락/거부합니다.
- 구현이 NFA보다 쉽고 빠릅니다 (항상 하나의 전이).

비결정적 유한 오토마타(Nondeterministic Finite Automata, NFA)

전이 다이어그램의 특정 유형입니다. 정규 표현식은 NFA로도 변환됩니다.

형식적 정의

$$N = (Q, \Sigma, \delta, q_0, F)$$

- Q : 유한한 상태 집합
- Σ : 알파벳
- δ : 전이 함수 ($Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$). 각 상태와 입력 기호(또는 ϵ) 쌍에 대해 여러 개의 다음 상태로 전이할 수 있습니다 (비결정적). ϵ -전이(ϵ -transition, 입력 없이 전이)가 가능합니다. 2^Q 는 Q 의멱집합(power set)입니다.
- q_0 : 시작 상태
- F : 최종(수락) 상태 집합

ϵ -폐쇄(ϵ -closure)

상태 p 의 ϵ -폐쇄는 p 자신을 포함하여 p 로부터 ϵ -경로를 따라 도달할 수 있는 모든 상태들의 집합입니다.

상태 집합 S 의 ϵ -폐쇄는 S 에 속하는 각 상태의 ϵ -폐쇄들의 합집합입니다.

NFA의 확장 전이 함수와 수락 언어

확장 전이 함수 δ^*

NFA에서 문자열 w 를 따라 도달 가능한 모든 상태 집합을 나타내는 함수입니다.

NFA에 의해 수락되는 언어 $L(N)$

시작 상태 q_0 에서 문자열 w 를 따라 도달한 상태 집합에 최종 상태가 하나라도 포함되어 있으면 w 는 수락됩니다.

정규 표현식 표현은 DFA보다 쉽고 직관적입니다.

NFA와 DFA의 관계

특징	DFA (결정적 유한 오토마타)	NFA (비결정적 유한 오토마타)
전이 함수의 결과	(하나의) 상태	(여러 개의) 상태 집합
ϵ -전이 가능 여부	불가능	가능
표현의 간결성/직관성	비교적 낮음	비교적 높음 (특히 정규 표현식 변환 시)
구현의 용이성	쉬움	어려움 (비결정성 및 ϵ -전이 처리 필요)
실행 속도	빠름	느림 (여러 경로 탐색 필요)
인식 가능한 언어 집합	정규 언어	정규 언어 (DFA와 동등함)

언어 L 은 DFA D 에 의해 수락될 경우에만 NFA N 에 의해 수락됩니다 ($L(D) = L(N)$). 즉, NFA와 DFA는 동일한 언어 집합(정규 언어)을 인식합니다.

인식 능력은 같지만, 표현의 간결성(NFA 우위), 구현의 용이성(DFA 우위), 실행 속도(DFA 우위)에서 차이가 있습니다.

정규 표현식 구현 과정

정규 표현식 -> NFA 변환

Thomson의 구성 등을 사용하여 정규 표현식을 NFA로 변환합니다.

NFA -> DFA 변환

Subset Construction 등을 사용하여 NFA를 DFA로 변환합니다. 이 과정에서 DFA의 상태는 NFA 상태들의 집합이 됩니다.

DFA -> 최소 DFA 변환

최적화 과정을 통해 DFA를 최소화합니다. 최소 DFA는 주어진 언어를 수락하는 DFA 중 가장 적은 수의 상태를 가집니다.

DFA 구현

최종적으로 DFA를 코드로 구현합니다.

어휘 분석의 목표와 과정

어휘 분석의 목표는 입력 문자열을 토큰(렉심)으로 분할하고, 토큰을 식별하는 것입니다.



스캔 방향

일반적으로 왼쪽에서 오른쪽으로 스캔하지만, 때로는 토큰의 끝과 다음 토큰의 시작을 결정하기 위해 **선행 탐색(lookahead)**이 필요할 수 있습니다.



모호성 문제

모호성 문제(Ambiguity issue)가 발생할 수 있습니다.

어휘 분석의 모호성 예시

Fortran 예시

화이트스페이스가 무시되므로 "VAR1"과 "VA R1"이 같습니다.

"DO 5 I = 1,25"와 "DO 5 I = 1.25"는 "." 또는 ","가 나올 때까지 DO5I가 변수인지 DO 문인지 알 수 없어 룩어헤드가 필요합니다.

C 예시

"=" (할당 연산자)과 "==" (관계 연산자). ==를 인식하려면 = 뒤에 =가 오는지를 확인하는 룩어헤드가 필요합니다.

JavaScript 예시

키워드가 객체 속성 이름으로 사용될 수 있어 주변 문맥(선행/후행 문자)을 기반으로 토큰이 구별됩니다.

C++ 예시

>>가 스트림 연산자일 수도, 비트 시프트 연산자일 수도 있어 문맥(예: std::cin >> number; vs x >> 2;)에 따라 달라집니다.

Nested templates
Foo<Bar>에서 >>를 두 개의 >로 분할해야 합니다.

모호성 해결 전략



룩어헤드 연산자 '/'

렉서 정규 표현식에서 사용될 수 있으며, $r1/r2$ 형태로 $r1$ 이 $r2$ 가 뒤 따라올 때만 매치됨을 나타냅니다. 매치 후 `forward_pointer`는 $r1$ 의 끝으로 되돌아갑니다.



가장 긴 렉시 규칙(Longest lexeme rule)

여러 패턴이 매치될 때 가장 긴 매치를 선택합니다.

예: +, ++, +++가 유효한 경우 입력 +++에 대해 +++를 선택합니다.

그러나 이 규칙만으로는 항상 올바른 토큰화를 보장하지 못할 수 있습니다 (예: abaa 입력에 대해 ab|aba, baa 패턴이 있을 때 가장 긴 aba를 선택하면 남은 a가 언어에 속하지 않는 문제).



우선순위 규칙 (Precedence rule)

여러 패턴이 매치될 때 더 높은 우선순위를 가진 패턴의 매치를 선택합니다.

예: if와 같은 키워드가 식별자 패턴에도 매치될 수 있지만, 키워드에 더 높은 우선순위를 부여하여 키워드로 인식하도록 합니다.

렉서의 토큰 처리

렉서는 일반적으로 화이트스페이스, 주석, 세미콜론 등 구문 분석에 직접 기여하지 않는 "흥미롭지 않은" 토큰들을 폐기합니다.

Lex 도구의 이론적 기반



Lex의 역할

Lex는 어휘 분석기 생성기로, 정규 표현식과 그에 해당하는 액션을 포함하는 Lex 소스 파일을 입력받아 C 코드 어휘 분석기를 생성합니다.



내부 동작 원리

생성된 어휘 분석기는 정규 표현식(패턴)을 인식하기 위해 유한 상태 기계 이론(정규 표현식 -> NFA -> DFA 변환 및 시뮬레이션)을 활용합니다.



Lex 소스 파일 구조

Lex 소스 파일의 규칙 섹션은 "정규 표현식 {액션}" 형태이며, Lex는 이 규칙을 기반으로 코드를 생성합니다.

Lex의 주요 변수와 기능

주요 변수

Lex는 yytext에 매치된 문자열(렉심)을, yyleng에 그 길이를 저장합니다.

고급 기능

Lex는 / 연산자를 사용한 선행 탐색 및 시작 조건(Start Conditions)을 통한 다른 어휘 규칙 적용 등 어휘 분석의 다양한 요구사항을 지원합니다.

어휘 분석 이론 문제 및 해설

문제 1: 용어 정의

다음 용어들을 제공된 자료에 기반하여 간략하게 정의하시오.

1. 토큰 (Token)
2. 렉시 (Lexeme)
3. 패턴 (Pattern)
4. 알파벳 (Alphabet)
5. 정규 표현식 (Regular Expression)
6. 정규 언어 (Regular Language)
7. DFA (Deterministic Finite Automata)
8. NFA (Nondeterministic Finite Automata)

문제 1 해설: 용어 정의

1. 토큰 (Token): 프로그래밍 언어에서 구문적으로 가장 작은 유효 단위로, 입력 문자열이 분할되는 '단어'들입니다. 식별자, 키워드, 연산자 등과 같은 구문적 범주에 해당합니다.
2. 렉시 (Lexeme): 소스 코드에 나타나는 토큰의 실제 인스턴스, 즉 토큰에 해당하는 특정 문자열입니다.
3. 패턴 (Pattern): 토큰을 정의하는 규칙 또는 정규 표현식입니다.
4. 알파벳 (Alphabet): 언어의 문자열을 구성하는 데 사용되는 유한한 기호(문자)들의 집합입니다.
1. 정규 표현식 (Regular Expression): 문자열 패턴을 형식적으로 정의하는 데 사용되며, 정규 언어를 표현하는 방법 중 하나입니다.
2. 정규 언어 (Regular Language): 유한 상태 기계(FSM)에 의해 인식될 수 있는 형식 언어의 한 유형이며, 정규 표현식으로 표현될 수 있습니다.
3. DFA (Deterministic Finite Automata): 유한 상태 기계의 한 유형으로, 각 상태와 입력 기호 쌍에 대해 다음 상태가 유일하게 결정되는 결정적 기계입니다. 정규 언어를 인식하는 데 사용됩니다.
4. NFA (Nondeterministic Finite Automata): 유한 상태 기계의 한 유형으로, 각 상태와 입력 기호(또는 ϵ) 쌍에 대해 여러 개의 다음 상태로 전이할 수 있거나 ϵ -전이가 가능한 비결정적 기계입니다. 정규 언어를 인식하는 데 사용됩니다 (DFA와 동등한 인식 능력).

문제 2: 정규 표현식

다음 정규 표현식들이 각각 어떤 문자열 집합을 나타내는지 설명하시오.

1. $a|xy$
2. $(ab)^+$
3. $a?$
4. $[0-9]^+(\.[0-9]^+)?$ (여기서 digit은 정규 정의 digit -> [0-9]를 따른다고 가정합니다)
5. letter(letter|digit)* (여기서 letter -> [a-zA-Z], digit -> [0-9] 정규 정의를 따른다고 가정합니다)

문제 2 해설: 정규 표현식

1. $a|xy$: 문자열 "a" 또는 문자열 "xy".
2. $(ab)^+$: 문자열 "ab"가 1회 이상 반복되는 집합입니다. 즉, {"ab", "abab", "ababab", ...} 입니다.
3. $a^?$: 문자열 "a"가 0회 또는 1회 나타나는 집합입니다. 즉, {"", "a"} 입니다.
4. $[0-9]^+(\.[0-9]^+)?$: 숫자 패턴을 나타냅니다. $[0-9]^+$ 는 0부터 9까지의 숫자가 1개 이상 반복됨을 의미하며 정수 부분을 나타냅니다. $(\.[0-9]^+)?$ 는 소수점 이하 부분을 나타냅니다. \.는 리터럴 점(.)을 의미하며, $[0-9]^+$ 는 소수점 뒤에 숫자가 1개 이상 반복됨을 의미합니다. ?는 이 전체 부분이 0회 또는 1회 나타남을 의미하므로 소수 부분은 선택 사항입니다. 따라서 이 패턴은 1개 이상의 숫자로 이루어진 정수 또는 소수점과 1개 이상의 숫자가 뒤따르는 숫자를 나타냅니다 (예: "123", "3.14", "42.0").
5. $\text{letter}(\text{letter}| \text{digit})^*$: 식별자(Identifier) 패턴을 나타냅니다. letter -> [a-zA-Z]는 영문자 하나를 의미하며, $(\text{letter}| \text{digit})^*$ 는 영문자 또는 숫자가 0회 이상 반복됨을 의미합니다. 따라서 이 패턴은 영문자로 시작하고 그 뒤에 영문자 또는 숫자가 0개 이상 이어지는 문자열 집합을 나타냅니다.

문제 3: 형식 언어 연산

두 언어 $X = \{"ab", "c"\}$, $Y = \{"d", "e"\}$ 에 대해 다음 연산의 결과를 나타내는 문자열 집합을 구하시오.

1. $X \cdot Y$
2. X^2
3. $X \cup Y$
4. X^* (일부 원소만 나열해도 좋습니다.)

문제 3 해설: 형식 언어 연산

1. $X \cdot Y: \{x \cdot y \mid x \in X, y \in Y\}$. X 의 원소와 Y 의 원소를 하나씩 이어붙인 집합입니다.

따라서 $X \cdot Y = \{"abd", "abe", "cd", "ce"\}$ 입니다.

- "ab"와 "d" -> "abd"
- "ab"와 "e" -> "abe"
- "c"와 "d" -> "cd"
- "c"와 "e" -> "ce"

1. $X^2: X \cdot X$ 와 같습니다. X 의 원소와 X 의 원소를 하나씩 이어붙인 집합입니다.

따라서 $X^2 = \{"abab", "abc", "cab", "cc"\}$ 입니다.

- "ab"와 "ab" -> "abab"
- "ab"와 "c" -> "abc"
- "c"와 "ab" -> "cab"
- "c"와 "c" -> "cc"

1. $X \cup Y: \{u \mid u \in X \text{ or } u \in Y\}$. X 또는 Y 에 속하는 모든 문자열의 집합입니다. 따라서 $X \cup Y = \{"ab", "c", "d", "e"\}$ 입니다.
2. X^* : 주어진 언어 X 의 문자열들을 0회 이상 이어붙여 만들 수 있는 모든 가능한 문자열의 집합이며, 빈 문자열 ϵ 를 포함합니다. $X^0 = \{\epsilon\}$ $X^1 = X = \{"ab", "c"\}$ $X^2 = \{"abab", "abc", "cab", "cc"\}$ (위 2번 결과) $X^3 = X^2 \cdot X = \{"abab", "abc", "cab", "cc"\} \cdot \{"ab", "c"\} = \{"ababab", "ababc", "abcab", "abccc", "cabab", "cabc", "ccab", "ccc"\}$ 따라서 $X^* = \{\epsilon, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "ababc", "abcab", "abccc", "cabab", "cabc", "ccab", "ccc", ...\}$ 입니다.

문제 4: 정규 언어의 한계

다음은 정규 언어에 대한 설명입니다. 이 설명 중 잘못된 부분을 찾아 수정하고 그 이유를 설명하시오.

"모든 형식 언어는 정규 표현식으로 표현될 수 있으며, 이는 모든 형식 언어가 유한 상태 기계로 인식될 수 있음을 의미한다. 따라서 컴파일러의 어휘 분석 단계에서는 어떤 언어의 소스 코드든 정규 표현식으로 정의하고 유한 상태 기계로 효율적으로 토큰을 인식할 수 있다."

문제 4 해설: 정규 언어의 한계

이 설명에는 세 가지 잘못된 부분이 있습니다.

1

잘못된 내용: "모든 형식 언어는 정규 표현식으로 표현될 수 있다."

수정: "모든 정규 언어는 정규 표현식으로 표현될 수 있다."

이유: 정규 표현식으로 표현될 수 있는 언어는 형식 언어의 일부인 정규 언어에 한정됩니다. 정규 언어보다 더 복잡한 구조를 가진 형식 언어(예: 문맥 자유 언어)는 정규 표현식으로 표현할 수 없습니다. 균형 잡힌 괄호의 집합 {}, (), {...}과 같은 언어는 정규 표현식으로 표현할 수 없는 비정규 언어의 대표적인 예입니다.

2

잘못된 내용: "이는 모든 형식 언어가 유한 상태 기계로 인식될 수 있음을 의미한다."

수정: "이는 정규 언어가 유한 상태 기계로 인식될 수 있음을 의미한다."

이유: 정규 언어와 유한 상태 기계는 등가적인 인식 능력을 가집니다. 유한 상태 기계(DFA/NFA)는 오직 정규 언어만을 인식할 수 있습니다. 비정규 언어는 유한 상태 기계로는 인식할 수 없으며, 더 강력한 자동 기계(예: 푸시다운 자동 기계, 튜링 기계)가 필요합니다.

3

잘못된 내용: "따라서 컴파일러의 어휘 분석 단계에서는 어떤 언어의 소스 코드는 정규 표현식으로 정의하고 유한 상태 기계로 효율적으로 토큰을 인식할 수 있다."

수정: "따라서 컴파일러의 어휘 분석 단계에서는 토큰의 패턴과 같이 정규 언어에 해당하는 부분은 정규 표현식으로 정의하고 유한 상태 기계로 효율적으로 토큰을 인식할 수 있다."

이유: 프로그래밍 언어의 모든 구문 규칙(예: if-then-else 구조의 중첩, 괄호의 균형)이 정규 언어에 해당하는 것은 아닙니다. 특히 중첩 구조는 문맥 자유 문법으로 기술되며, 이는 구문 분석(Syntactic Analysis) 단계에서 문맥 자유 문법과 푸시다운 자동 기계를 사용하여 처리됩니다. 어휘 분석 단계에서는 키워드, 식별자, 연산자, 숫자 리터럴 등과 같이 개별 토큰의 패턴을 정의하는 데 정규 언어 및 유한 상태 기계가 사용됩니다.

문제 5: DFA와 NFA의 특징

DFA와 NFA는 모두 정규 언어를 인식할 수 있습니다. 다음 표의 빈칸을 채워 DFA와 NFA의 특징을 비교하시오.

특징	DFA (결정적 유한 오토마타)	NFA (비결정적 유한 오토마타)
전이 함수의 결과	?	?
ϵ -전이 가능 여부	?	?
표현의 간결성/직관성	?	?
구현의 용이성	?	?
실행 속도	?	?
인식 가능한 언어 집합	?	?



문제 5 해설: DFA와 NFA의 특징

특징	DFA (결정적 유한 오토마타)	NFA (비결정적 유한 오토마타)
전이 함수의 결과	(하나의) 상태	(여러 개의) 상태 집합
ϵ -전이 가능 여부	불가능	가능
표현의 간결성/직관성	비교적 낮음	비교적 높음 (특히 정규 표현식 변환 시)
구현의 용이성	쉬움	어려움 (비결정성 및 ϵ -전이 처리 필요)
실행 속도	빠름	느림 (여러 경로 탐색 필요)
인식 가능한 언어 집합	정규 언어	정규 언어 (DFA와 동등함)

문제 6: 어휘 분석의 이슈

다음은 어휘 분석 과정에서 발생할 수 있는 이슈와 해결 전략에 대한 설명입니다. 이 설명 중 잘못된 부분을 찾아 수정하고 그 이유를 설명하시오.

"어휘 분석기는 소스 코드를 왼쪽에서 오른쪽으로 스캔하며 토큰을 인식한다. 이때 모호성이 발생하면 가장 긴 매치를 선택하는 '가장 긴 렉시 규칙'만 적용하면 항상 올바른 토큰 분할이 가능하다. C++에서 >>와 같은 연산자의 모호성은 룩어헤드로는 해결할 수 없으며, 구문 분석 단계의 문맥 정보를 통해 처리해야 한다."

문제 6 해설: 어휘 분석의 이슈

이 설명에는 세 가지 잘못된 부분이 있습니다.

1

잘못된 내용: "모호성이 발생하면 가장 긴 매치를 선택하는 '가장 긴 렉시 규칙'만 적용하면 항상 올바른 토큰 분할이 가능하다."

수정: "모호성이 발생하면 '가장 긴 렉시 규칙'과 '우선순위 규칙' 등을 함께 적용하여 해결하며, 가장 긴 렉시 규칙만으로는 항상 올바른 토큰 분할을 보장할 수 없다."

이유: 가장 긴 렉시 규칙은 여러 패턴이 매치될 때 가장 긴 매치를 선택하지만, 이것만으로는 모든 모호성을 해결할 수 없습니다. 예를 들어 ab|aba 패턴과 baa 패턴이 있을 때 입력 abaa에 대해 가장 긴 aba를 선택하면 남은 a가 문제가 됩니다. 또한, 키워드(if)와 식별자(if)처럼 길이보다 의미가 중요한 경우, 키워드에 더 높은 우선순위를 부여하는 우선순위 규칙이 필요합니다. 따라서 올바른 토큰 분할을 위해서는 가장 긴 렉시 규칙과 우선순위 규칙 등이 복합적으로 사용될 수 있습니다.

2

잘못된 내용: "C++에서 >>와 같은 연산자의 모호성은 룩어헤드로는 해결할 수 없으며, 구문 분석 단계의 문맥 정보를 통해 처리해야 한다."

수정: "C++에서 >>와 같은 연산자의 모호성은 때로는 룩어헤드로 처리하기 어렵거나 불가능하며, 구문 분석 단계 또는 그 이후(의미 분석)의 문맥 정보를 통해 처리해야 하는 경우가 있다."

이유: C++의 >>는 스트림 연산자 (`std::cin >> number;`)일 수도, 비트 시프트 연산자 (`x >> 2;`)일 수도 있으며, nested template(`Foo<Bar>`)에서는 두 개의 >로 분리되어야 합니다. 이러한 모호성은 단순한 룩어헤드만으로는 완전히 해결하기 어려울 수 있습니다. 자료에서는

JavaScript에서 주변 문맥 (preceding and following characters)에 따라 토큰이 구별된다고 언급하며, C++의 복잡한 경우(nested templates)를 어휘 분석의 이슈로 제시합니다. 전통적으로 어휘 분석은 문맥 독립적인 패턴 매칭을 목표로 하지만, 일부 복잡한 언어 구조에서는 어휘 분석 단계의 룩어헤드만으로는 부족하고 구문 분석기와의 협력 또는 더 높은 단계의 문맥 정보를 활용해야 하는 경우가 발생합니다.

3

잘못된 내용 (함축된 의미): 어휘 분석 단계에서 세미콜론(;)이나 주석은 중요한 토큰이므로 반드시 파서에게 전달되어야 한다.

수정: 어휘 분석기는 일반적으로 화이트스페이스, 주석, 세미콜론 등 구문 분석에 직접적인 의미를 가지지 않는 "흥미롭지 않은" 토큰들을 폐기하고 파서에게 전달하지 않는다.

이유: 자료에서는 렉서가 보통 화이트스페이스, 주석, 세미콜론과 같이 구문 분석에 기여하지 않는 토큰들을 버린다고 명시합니다. 이러한 요소들은 문자열을 토큰으로 분할하는 어휘 분석 단계에서는 인식되지만, 프로그램 구조를 이해하는 구문 분석 단계에서는 필요 없기 때문입니다. (단, 언어에 따라 세미콜론이 문장의 끝을 명확히 표시하는 중요한 역할을 하기도 합니다. 하지만 토큰 스트림 자체에 포함되지 않고 렉서가 내부적으로 처리하거나 파서에게 특정 신호를 전달하는 방식으로 처리될 수 있습니다.)

문제 7: ϵ -폐쇄 계산

상태 집합 $S = \{p, q\}$ 가 주어졌고, NFA의 ϵ -전이 관계가 다음과 같다고 가정합니다 (여기서 $s --\epsilon--> t$ 는 상태 s 에서 ϵ 입력을 통해 상태 t 로 전이 가능함을 의미합니다).

- $p --\epsilon--> r$
- $q --\epsilon--> s$
- $r --\epsilon--> t$
- $s --\epsilon--> t$
- $t --\epsilon--> p$

S 의 ϵ -폐쇄 (ϵ -closure(S))를 계산하시오.

문제 7 해설: ϵ -폐쇄 계산

상태 집합 S 의 ϵ -폐쇄는 S 에 속하는 각 상태의 ϵ -폐쇄들의 합집합입니다. ϵ -폐쇄(p)는 p 자신을 포함하여 p 로부터 ϵ -경로를 따라 도달할 수 있는 모든 상태 집합입니다.

1

ϵ -closure(p) 계산

- p 자신: $\{p\}$
- p 에서 ϵ 로 도달 가능: r ($\{p, r\}$)
- r 에서 ϵ 로 도달 가능: t ($\{p, r, t\}$)
- t 에서 ϵ 로 도달 가능: p (이미 포함)
- p, r, t 에서 추가 ϵ -전이 경로가 없는지 확인. 현재 경로 상에서는 더 이상 새로운 상태가 없습니다.

따라서 ϵ -closure(p) = $\{p, r, t\}$ 입니다.

2

ϵ -closure(q) 계산

- q 자신: $\{q\}$
- q 에서 ϵ 로 도달 가능: s ($\{q, s\}$)
- s 에서 ϵ 로 도달 가능: t ($\{q, s, t\}$)
- t 에서 ϵ 로 도달 가능: p ($\{q, s, t, p\}$)
- p 에서 ϵ 로 도달 가능: r ($\{q, s, t, p, r\}$)
- r 에서 ϵ 로 도달 가능: t (이미 포함)

따라서 ϵ -closure(q) = $\{p, q, r, s, t\}$ 입니다.

3

ϵ -closure(S) = ϵ -closure($\{p, q\}$) 계산

ϵ -closure($\{p, q\}$) = ϵ -closure(p) \cup ϵ -closure(q).

ϵ -closure($\{p, q\}$) = $\{p, r, t\} \cup \{p, q, r, s, t\}$

ϵ -closure($\{p, q\}$) = $\{p, q, r, s, t\}$ 입니다.

따라서 상태 집합 $S = \{p, q\}$ 의 ϵ -폐쇄는 $\{p, q, r, s, t\}$ 입니다.

문제 8: 정규 표현식 및 정규 정의

다음 정규 표현식을 정규 정의를 사용하여 더 간결하게 표현하시오.

Regexp: $([a-zA-Z]_|[a-zA-Z])([a-zA-Z]_|[0-9])^*$

단, 다음 정규 정의를 활용하시오.

letter $\rightarrow [a-zA-Z]$

digit $\rightarrow [0-9]$



문제 8 해설: 정규 표현식 및 정규 정의

주어진 정규 표현식 $([a-zA-Z]_ | [a-zA-Z])([a-zA-Z]_ | [0-9])^*$ 를 분석해 봅시다.

1

첫 번째 그룹 $([a-zA-Z]_ | [a-zA-Z])$ 분석

- $[a-zA-Z]_$: 영문자 하나 뒤에 언더스코어(_)가 오는 패턴입니다.
- $[a-zA-Z]$: 영문자 하나 패턴입니다.
- $|$: 둘 중 하나를 선택합니다.
- 따라서 이 그룹은 "영문자+언더스코어" 또는 "영문자" 패턴입니다. 이는 '영문자로 시작되며, 만약 바로 뒤에 언더스코어가 온다면 그것까지 포함하는' 첫 글자 부분을 의미합니다.
- 정규 정의 letter $\rightarrow [a-zA-Z]$ 를 사용하면 $(letter_ | letter)$ 로 쓸 수 있습니다.

2

두 번째 그룹 $([a-zA-Z]_ | [0-9])^*$ 분석

- $[a-zA-Z]$: 영문자.
- $_$: 언더스코어.
- $[0-9]$: 숫자.
- $|$: 셋 중 하나를 선택합니다.
- $*$: 이 전체 패턴이 0회 이상 반복됨을 의미합니다.
- 따라서 이 그룹은 "영문자 또는 언더스코어 또는 숫자"가 0개 이상 반복되는 패턴입니다.
- 정규 정의 letter $\rightarrow [a-zA-Z]$, digit $\rightarrow [0-9]$ 를 사용하면 $(letter | digit)^*$ 로 쓸 수 있습니다.

3

정규 정의를 사용한 간결한 표현

정규 정의를 사용하여 새로운 이름을 부여하여 전체 정규 표현식을 간결하게 만들 수 있습니다.

identifier_char \rightarrow letter | _ | digit

first_char_part \rightarrow letter _ | letter

이제 이 정규 정의를 사용하여 원래 정규 표현식을 다시 표현하면 다음과 같습니다.

first_char_part
identifier_char*

또는 좀 더 직관적인 이름을 사용할 수도 있습니다.

id_start \rightarrow letter | letter_
id_char \rightarrow letter | digit | _

그러면 정규 표현식은 id_start id_char*가 됩니다.

나만의 질문 1: 개념형

토큰, 렉시, 패턴의 관계와 역할은 무엇이며, 어휘 분석 과정에서 어떻게 상호작용하는가?

모범 답안

토큰, 렉시, 패턴은 어휘 분석의 핵심 개념으로, 다음과 같은 관계와 역할을 가집니다:

- **패턴(Pattern)**: 토큰을 정의하는 규칙 또는 정규 표현식입니다. 예를 들어, 식별자를 정의하는 패턴은 "영문자로 시작하고 그 뒤에 영문자, 숫자, 언더스코어가 0개 이상 오는 문자열"입니다.
- **렉시(Lexeme)**: 소스 코드에서 패턴에 매치되는 실제 문자열입니다. 예를 들어, 'count', 'x1', 'total_sum'은 모두 식별자 패턴에 매치되는 렉시입니다.
- **토큰(Token)**: 렉시를 분류한 결과로, 보통 <토큰 타입, 속성 값> 형태로 표현됩니다. 예를 들어, 렉시 'count'는 <ID, "count">라는 토큰으로 분류됩니다.

어휘 분석 과정에서 이들은 다음과 같이 상호작용합니다:

1. 렉서는 먼저 언어에서 인식해야 할 모든 토큰 타입(키워드, 식별자, 연산자 등)에 대한 패턴을 정의합니다.
2. 소스 코드를 스캔하면서, 현재 위치에서 시작하는 문자열이 어떤 패턴에 매치되는지 확인합니다.
3. 매치되는 패턴을 찾으면, 해당 문자열(렉시)을 인식하고 적절한 토큰 타입으로 분류합니다.
4. 생성된 토큰은 구문 분석기(파서)에 전달되어 프로그램의 구조를 분석하는 데 사용됩니다.

이는 마치 자연어에서 문장을 단어로 분리하고, 각 단어가 명사, 동사, 형용사 등 어떤 품사에 해당하는지 분류하는 과정과 유사합니다.

나만의 질문 2: 과정/비교형

DFA와 NFA의 차이점은 무엇이며, 정규 표현식 구현 과정에서 어떻게 활용되는가?

모범 답안

DFA(결정적 유한 오토마타)와 NFA(비결정적 유한 오토마타)는 정규 언어를 인식하는 두 가지 유형의 유한 상태 기계입니다. 주요 차이점은 다음과 같습니다:

- **전이의 결정성:** DFA는 각 상태와 입력 기호 쌍에 대해 정확히 하나의 다음 상태만 가집니다. 반면 NFA는 여러 개의 가능한 다음 상태를 가질 수 있거나, 입력 없이도 전이(ϵ -전이)할 수 있습니다.
- **구현 복잡성:** DFA는 구현이 간단하고 실행이 빠릅니다. NFA는 여러 가능한 경로를 동시에 탐색해야 하므로 구현이 복잡하고 실행이 느립니다.
- **상태 수:** 동일한 언어를 인식하는 경우, NFA는 일반적으로 DFA보다 적은 수의 상태를 필요로 합니다. 이는 NFA가 더 간결한 표현을 제공함을 의미합니다.

정규 표현식 구현 과정에서 DFA와 NFA는 다음과 같이 활용됩니다:

1. **정규 표현식 → NFA 변환:** 정규 표현식은 먼저 Thomson의 구성 알고리즘을 사용하여 NFA로 변환됩니다. 이 과정은 정규 표현식의 각 연산자(합집합, 이어붙이기, 클레이니 스타 등)에 대한 NFA 구성 규칙을 적용하여 수행됩니다. NFA는 정규 표현식의 구조를 직관적으로 반영하므로 이 단계에서 선호됩니다.
2. **NFA → DFA 변환:** 생성된 NFA는 Subset Construction 알고리즘을 사용하여 동등한 DFA로 변환됩니다. 이 과정에서 NFA의 상태 집합(ϵ -폐쇄 포함)이 DFA의 각 상태가 됩니다. DFA는 실행 효율성이 높기 때문에 이 변환이 필요합니다.
3. **DFA 최소화:** 생성된 DFA는 최소화 알고리즘을 통해 상태 수를 줄여 최적화됩니다.
4. **코드 생성:** 최종적으로 최소화된 DFA는 어휘 분석기의 코드로 구현됩니다.

이 과정은 마치 복잡한 지도(정규 표현식)를 먼저 모든 가능한 경로를 포함하는 상세 지도(NFA)로 변환한 후, 이를 다시 명확하고 효율적인 네비게이션 시스템(DFA)으로 최적화하는 것과 유사합니다.

나만의 질문 3: 이유/목표형

어휘 분석이 컴파일러의 첫 단계로 수행되어야 하는 이유는 무엇이며, 이 과정이 없다면 어떤 문제가 발생할 수 있는가?

모범 답안

어휘 분석이 컴파일러의 첫 단계로 수행되어야 하는 이유는 다음과 같습니다:

- 구문 분석 단순화:** 어휘 분석은 소스 코드의 문자 스트림을 의미 있는 토큰으로 변환함으로써 구문 분석기(파서)의 작업을 단순화합니다. 파서는 개별 문자가 아닌 토큰 단위로 작업할 수 있어 더 효율적입니다.
- 추상화 수준 향상:** 토큰은 소스 코드의 더 높은 추상화 수준을 제공합니다. 예를 들어, 파서는 'i', 'f'라는 개별 문자가 아닌 'if'라는 키워드 토큰을 처리합니다.
- 공백 및 주석 처리:** 어휘 분석기는 공백, 주석 등 구문적으로 중요하지 않은 요소를 제거하여 파서가 핵심 프로그램 구조에만 집중할 수 있게 합니다.
- 효율적인 오류 처리:** 어휘 수준의 오류(잘못된 식별자, 알 수 없는 문자 등)를 초기에 감지하여 더 복잡한 구문 분석 단계로 넘어가기 전에 처리할 수 있습니다.

어휘 분석 과정이 없다면 다음과 같은 문제가 발생할 수 있습니다:

- 구문 분석 복잡성 증가:** 파서가 문자 수준에서 작동해야 하므로, 문법이 훨씬 더 복잡해지고 파싱 알고리즘의 효율성이 크게 저하됩니다.
- 문맥 처리 어려움:** 공백이나 주석을 포함한 모든 문자를 파서가 직접 처리해야 하므로, 실제 프로그램 구조를 인식하기 어려워집니다.
- 오류 메시지 품질 저하:** 어휘적 오류와 구문적 오류를 구분하기 어려워져, 개발자에게 덜 유용한 오류 메시지가 제공될 수 있습니다.
- 컴파일러 설계 복잡화:** 어휘 분석과 구문 분석의 관심사 분리가 없어져 컴파일러 설계가 더 복잡해지고 유지보수가 어려워집니다.
- 최적화 기회 상실:** 어휘 분석 단계에서 수행할 수 있는 최적화(예: 키워드 테이블 조회)를 활용할 수 없게 됩니다.

이는 마치 책을 읽을 때 먼저 글자를 단어로 인식하는 과정 없이 개별 글자만 보고 문장의 구조를 이해하려는 것과 같습니다. 인간의 언어 처리 방식처럼, 컴퓨터도 먼저 기본 단위(토큰)를 인식한 후 더 복잡한 구조를 분석하는 것이 효율적입니다.

어휘 분석 학습 정리

컴파일러와 어휘 분석의 기본 개념

컴파일러는 고급 언어를 기계 코드로 번역하는 프로그램이며, 어휘 분석은 소스 코드를 토큰으로 분할하는 첫 번째 단계입니다. 토큰, 렉시, 패턴의 관계를 이해하는 것이 중요합니다.

형식 언어와 정규 표현식
형식 언어의 기본 연산(이어붙이기, 합집합, 클레이니 스타 등)과 정규 표현식의 연산자를 이해하고 활용할 수 있어야 합니다. 정규 정의를 통해 복잡한 패턴을 간결하게 표현할 수 있습니다.

유한 상태 기계와 구현 과정

DFA와 NFA의 특징과 차이점을 이해하고, 정규 표현식을 NFA로 변환한 후 DFA로 최적화하는 과정을 알아야 합니다. 이 과정은 어휘 분석기 구현의 이론적 기반이 됩니다.

어휘 분석의 이슈와 해결 전략

모호성 문제를 해결하기 위한 가장 긴 렉시 규칙, 우선순위 규칙, 룩어헤드 연산자 등의 전략을 이해하고 적용할 수 있어야 합니다. Lex와 같은 도구를 활용하여 실제 어휘 분석기를 구현하는 방법도 중요합니다.