

어휘 분석이란?

어휘 분석은 소스 코드를 문자열 스트림으로 읽어들여 의미 있는 최소 단위인 토큰(token)들로 분할하는 과정입니다.

예를 들어, int main()과 같은 코드 라인은 다음과 같이 분할됩니다:

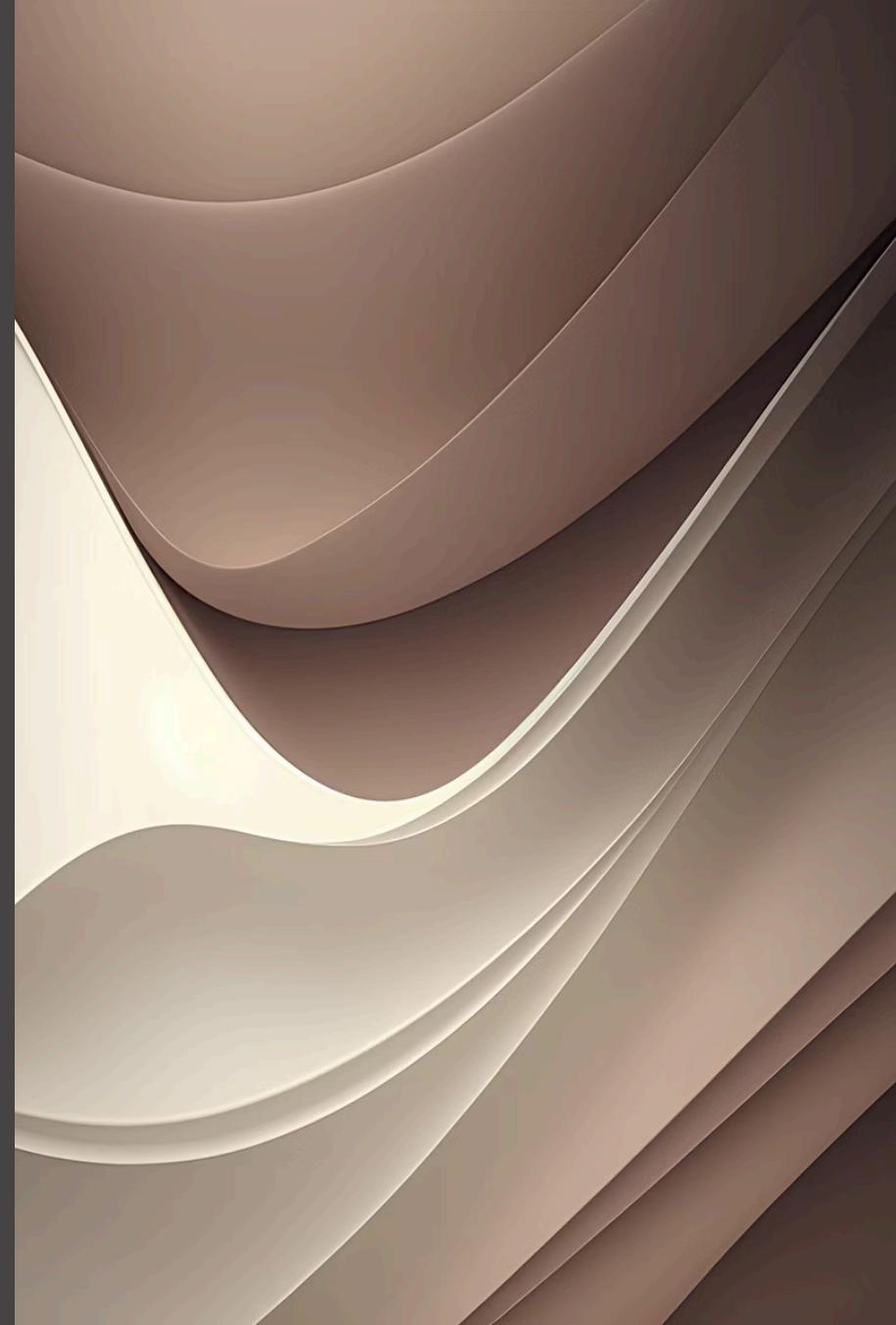
- int (키워드 토큰)
- main (식별자 토큰)
- ((괄호 토큰)
-) (괄호 토큰)

어휘 분석의 이론적 기반

어휘 분석 단계의 이론적 기반은 **정규 언어(Regular Language)**입니다. 정규 언어는 키워드, 식별자, 숫자 상수, 연산자 등 프로그래밍 언어의 유효한 토큰 패턴을 정의하는 데 사용됩니다.

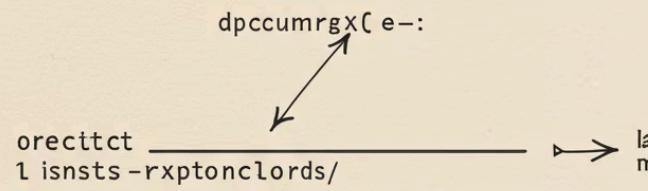
정규 언어는 두 가지 주요 방식으로 표현되거나 인식될 수 있습니다:

1. **정규 표현식(Regular Expression, Regexp)**을 사용하여 패턴을 정의하는 것
2. **유한 상태 기계(Finite State Machine, FSM)**를 사용하여 패턴을 인식하는 것

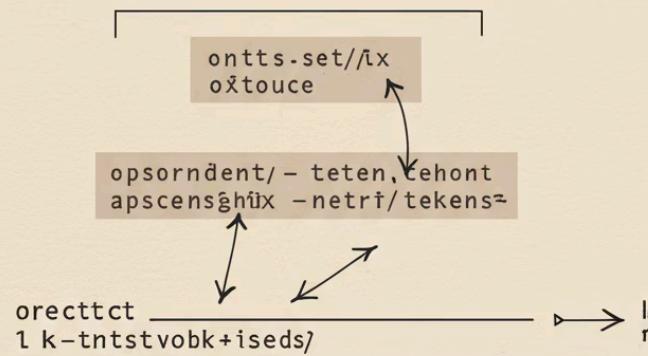


Regular e. Exgesive

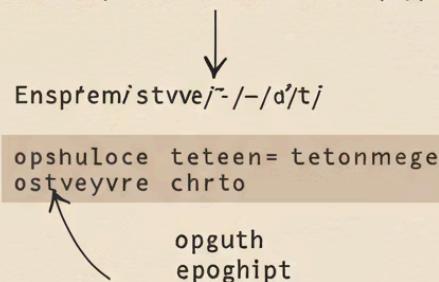
Matuncion mattemcus gpr= tattens



Eregatr exturons/=:



Vithtove//xtrber-/nstwtnk/
matherp soumeskst/ipesrte/x; j



Maccat storaxsion

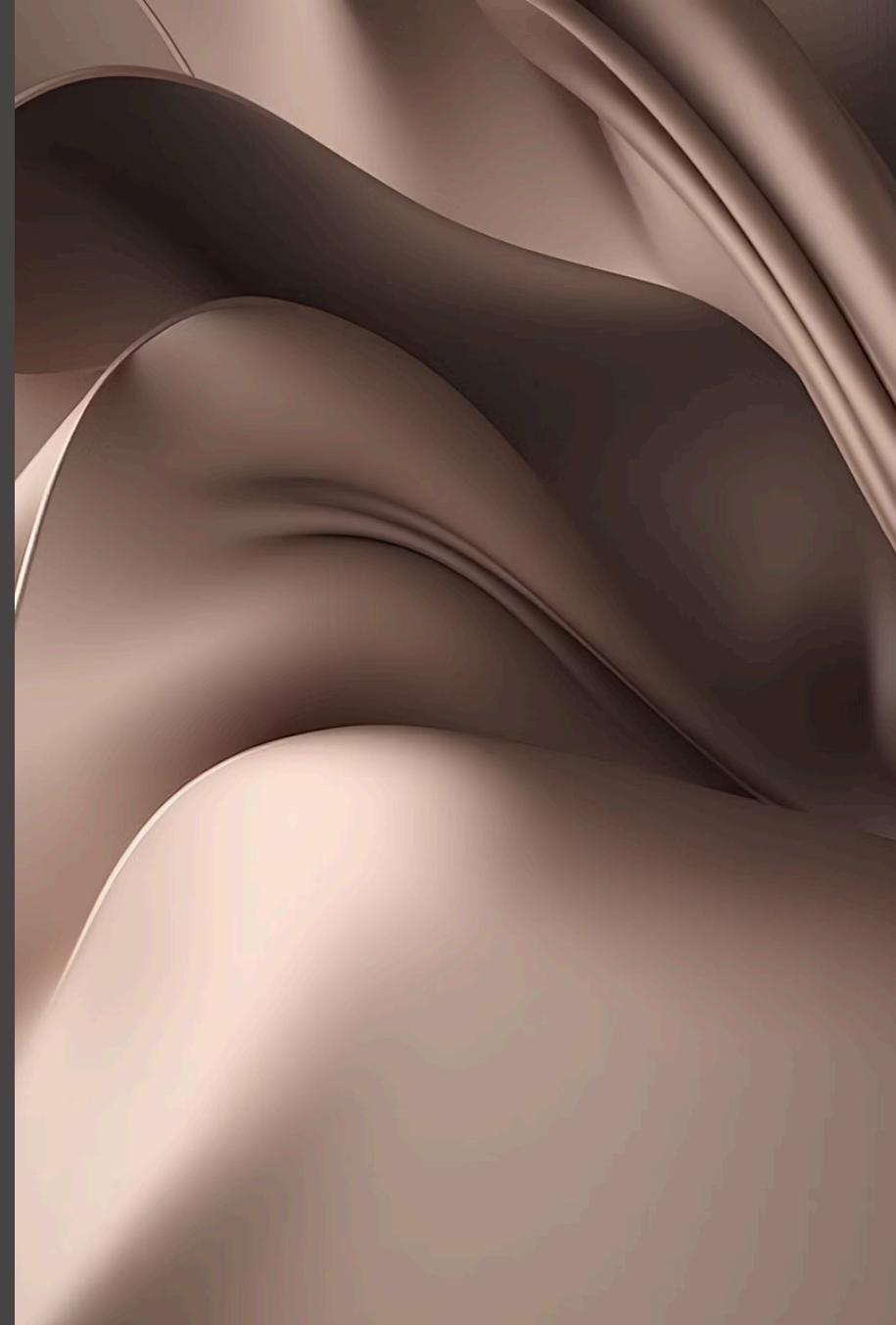
정규 표현식 (Regular Expression, Regexp)

정규 표현식은 문자열 패턴을 형식적으로 명세하는 강력한 도구입니다. Lex는 이러한 정규 표현식을 사용하여 입력 스트림에서 인식해야 할 토큰의 패턴을 정의합니다.

정규 표현식의 기본 구성 요소 및 연산에 대해 알아보겠습니다.

정규 표현식의 기본 구성 요소

- 알파벳 (Σ): 문자열을 구성하는 데 사용되는 유한한 기호 집합입니다. 예: $\{a, b\}$, $\{0, 1\}$
- 문자열 (string) / 단어 (word): 알파벳에서 가져온 기호들의 유한한 순서열입니다. 예: "abab", "aa", "aabb"
- 빈 문자열 (ε): 길이가 0인 문자열로, 어떤 문자열에 이어붙여도 원래 문자열 그대로입니다



정규 표현식의 기본 연산자



이어붙이기 (Concatenation, \cdot)

두 문자열이나 언어를 단순히 연결하는 연산입니다. $x \cdot y$ 는 xy 로 표기하며, 결합법칙은 성립하나 교환법칙은 성립하지 않습니다. 정규 표현식에서는 문자를 나란히 쓰는 것으로 표현하며, 기호는 생략됩니다.



거듭제곱 (Exponentiation)

문자열 x 의 n 제곱은 x 를 n 번 이어붙인 것입니다. $x^0 = \epsilon$ (빈 문자열)입니다. 언어 X 의 n 제곱은 X 의 언어에서 가져온 n 개의 문자열을 이어붙여 만들 수 있는 모든 결과의 집합입니다.



합집합 (Union, $|$)

두 언어 또는 정규 표현식의 합집합은 둘 중 하나에 속하는 모든 문자열의 집합입니다. Lex에서는 $|$ 기호를 사용합니다. 예: $ab|cd$ 는 "ab" 또는 "cd"를 매치합니다.





정규 표현식의 특수 연산자



클레이니 스타 (Kleene's Star, *)

주어진 언어의 문자열을 0회 이상 반복하여 만들 수 있는 모든 문자열의 집합입니다. 빈 문자열(ϵ)을 포함합니다. Lex에서는 * 기호를 사용합니다. 예: A*는 "", "A", "AA", ...를 매치합니다.



클레이니 플러스 (Kleene's Plus, +)

클레이니 스타와 유사하나, 1회 이상 반복을 의미하며 빈 문자열을 제외합니다. Lex에서는 + 기호를 사용합니다. 예: A+는 "A", "AA", AAA", ...를 매치합니다.



선택 사항 (?)

앞의 패턴이 0회 또는 1회 나타남을 의미합니다. Lex에서는 ? 기호를 사용합니다. 예: a?는 "" 또는 "a"를 매치합니다. colou?r는 "color" 또는 "colour"를 매치합니다.

정규 표현식의 그룹화와 문자 클래스



(): 그룹화

표현식의 일부를 그룹화하는 데 사용됩니다. 예: $(ab|cd+)?(ef)^*$



[]: 문자 클래스

괄호 안의 문자 집합 중 하나를 매치합니다. 예: [abc]는 "a", "b", 또는 "c"를 매치합니다.



-: 범위 지정

문자 클래스 내에서 범위를 지정합니다. 예: [a-zA-Z]는 소문자 알파벳과 숫자를 매치합니다.

ASCII 코드 상에서 연속되는 집합이어야 함을 주의!

정규 표현식의 특수 문자



\wedge : 부정 및 라인 시작

문자 클래스 내에서 사용 시
'not'의 의미로, 괄호 안의 문자를
제외한 모든 문자를 매치합니다.
예: $[^\wedge abc]$ 는 a, b, c를 제외한 모든
문자를 매치합니다. Lex에서는 \wedge
가 라인의 시작을 의미하기도 합니
다.



. : 와일드카드

개행 문자($\backslash n$)를 제외한 어떤 단일
문자든지 매치합니다. 예: a.b는
"acb", "a1b", "a_b" 등을 매치합
니다.



\$: 라인의 끝

라인의 끝을 매치합니다. ab\$는
ab $\backslash n$ 과 동일하며, 뒤따라오는 개
행 문자에 대한 lookahead입니다.

정규 정의 (Regular Definition)

편리성을 위해 복잡한 정규 표현식에 이름을 부여하고 재사용할 수 있습니다.

예시:

```
DIGIT [0-9]
LETTER [a-zA-Z]
ID    {LETTER}({LETTER} | {DIGIT})*
```

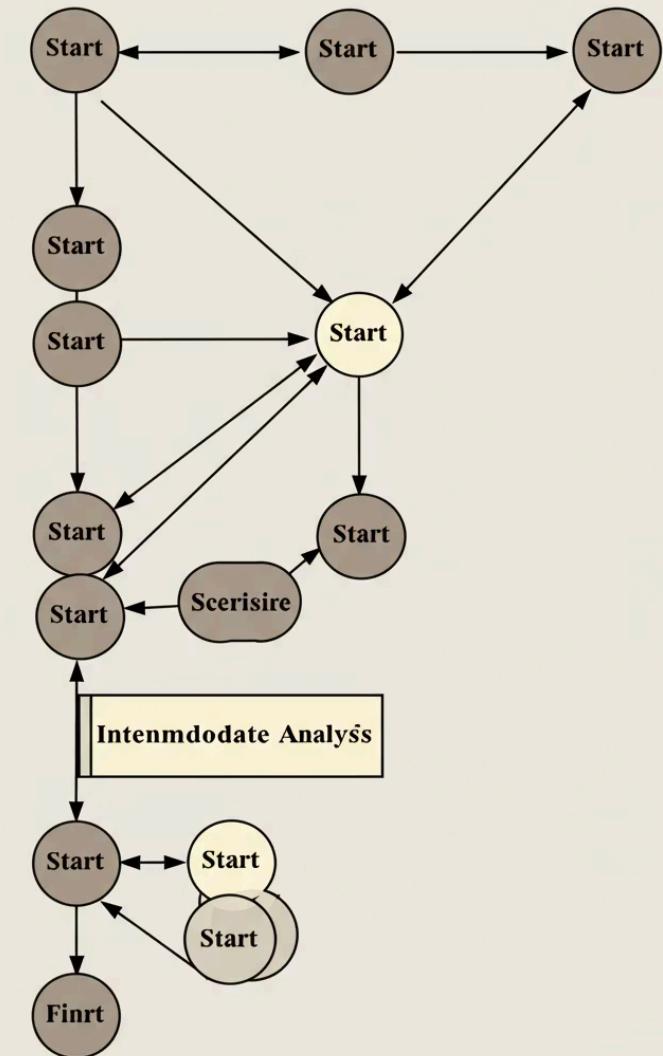
이렇게 정의된 이름은 다른 정규 표현식에서 {DIGIT}, {LETTER}와 같이 사용할 수 있습니다.

유한 상태 기계 (Finite State Machine, FSM) / 오토마타 (Automata)

정규 언어는 유한 상태 기계에 의해 인식될 수 있습니다. Lex가 생성하는 어휘 분석기 프로그램은 기본적으로 유한 상태 기계로 구현됩니다.

자동 기계 (Automata)는 계산의 수학적 모델로, 입력을 받아 상태를 변경하고 출력을 생성하는 추상 기계입니다. 소프트웨어 시스템의 동작을 모델링하는 데 사용됩니다.

Finite State Machine
Lexical Analysis



유한 자동 기계 (Finite Automata, FA)

유한한 개수의 상태와 상태 간의 전이(transition)를 가진 기계입니다. 입력 기호에 따라 현재 상태에서 다음 상태로 이동하며, 입력 문자열의 끝에 도달했을 때 최종 상태(final state, accepting state)에 있으면 해당 문자열을 수락(accept)합니다.



결정적 유한 오토마타 (Deterministic Finite Automata, DFA)

각 상태와 입력 기호 쌍에 대해 다음 상태가 유일하게 하나로 결정되는 유한 자동 기계입니다. 전이 함수 δ 는 $Q \times \Sigma \rightarrow Q$ 형태입니다.



비결정적 유한 오토마타 (Nondeterministic Finite Automata, NFA)

각 상태와 입력 기호 쌍에 대해 다음 상태가 여러 개일 수 있으며, 입력 기호 없이 상태를 이동하는 ϵ -전이(ϵ -transition)가 허용됩니다. 전이 함수 δ 는 $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ (상태 집합의멱집합) 형태입니다.

ϵ -폐쇄 (ϵ -closure)

ϵ -전이가 있는 NFA에서 특정 상태 또는 상태 집합으로부터 ϵ 전이만을 따라 도달할 수 있는 모든 상태들의 집합입니다.

이 개념은 NFA를 DFA로 변환하는 과정에서 중요한 역할을 합니다.

NFA와 DFA의 관계

NFA와 DFA는 정규 언어를 인식하는 등가적인 능력을 가집니다. 즉, 어떤 언어가 NFA에 의해 인식되면 반드시 그 언어를 인식하는 DFA가 존재하며, 그 역도 성립합니다.

NFA의 장점

정규 표현식으로부터 쉽게 구성할 수 있습니다.

DFA의 장점

구현이 더 쉽고 효율적입니다 (항상 하나의 전이만 따라가면 되므로).

정규 표현식 구현 과정

Lex와 같은 어휘 분석기 생성기는 일반적으로 다음 단계를 따릅니다:



정규 표현식을 NFA로 변환

Thomson의 구성 등의 알고리즘 사용



NFA를 DFA로 변환

Subset Construction 등의 알고리즘 사용



DFA를 최소 상태 DFA로 최적화

Minimal DFA 생성



최종 DFA를 기반으로 인식기 구현

Lex는 C 코드를 생성

Lex 도구 (Lexical Analyzer Generator)

Lex는 어휘 분석 과정을 자동화하기 위한 프로그램 생성기입니다. Lex 소스 파일(.l)을 입력받아 C 언어로 작성된 어휘 분석기 프로그램 (lex.yy.c)을 생성합니다.

Lex의 역할은 문자 입력 스트림에 대한 어휘 처리를 수행하는 것입니다. 소스 코드에서 정의된 정규 표현식(토큰 패턴)에 해당하는 문자열(렉심, lexeme)을 인식하고, 해당 패턴이 발견되었을 때 수행할 액션(Action)을 실행합니다.

Lex 소스 파일 (.l) 구조

Lex 소스 파일은 크게 세 섹션으로 나뉩니다:

```
{definitions}  
%%  
{rules}  
%%  
{user subroutines}
```



선언 섹션 (Declarations)

정규 정의(예: DIGIT), 변수, 상수, C 헤더 파일(%{...%} 안에 포함) 등을 선언합니다. 이 내용은 생성된 lex.yy.c 파일의 헤더 부분에 복사됩니다.



규칙 섹션 (Rules)

Lex의 핵심 부분으로, 정규 표현식 {액션} 형태의 규칙들로 구성됩니다.



사용자 서브루틴 섹션 (User Subroutines)

규칙 섹션의 액션이나 전체 Lexer 프로그램에서 사용할 C 함수들을 정의합니다. 일반적인 C 코드 함수들이며, 보통 main 함수를 포함합니다. 이 내용도 lex.yy.c 파일의 끝 부분에 복사됩니다.

Lex 규칙 섹션 상세

규칙 섹션은 Lex의 핵심 부분으로, 정규 표현식과 그에 대응하는 액션으로 구성됩니다:

- **정규 표현식 (왼쪽 열)**: 인식하고자 하는 토큰의 패턴을 정의합니다.
- **액션 (오른쪽 열)**: 해당 정규 표현식이 입력 스트림에서 매치되었을 때 실행될 C 언어 코드 조각입니다. 중괄호 {} 안에 작성합니다.

예시:

```
{integer} {printf("found keyword INT");}
```

이 규칙은 입력에서 {integer} 정규 정의에 해당하는 패턴을 발견할 때마다 메시지를 출력하는 액션을 정의합니다.

Lex 파일 컴파일 과정

Lex 소스 파일(lex.l)을 Lex (또는 Flex) 도구에 입력하면 C 소스 파일 lex.yy.c가 생성됩니다. 이 C 소스 파일을 C 컴파일러(gcc 등)로 컴파일하면 최종 어휘 분석기 프로그램이 생성됩니다.

```
$ flex lex.l  
$ gcc -o Lexer lex.yy.c -l||
```

생성된 Lexer는 yylex() 함수를 통해 호출되며, 이 함수가 입력을 읽고 토큰을 반환합니다.

Lex 예약 변수 (Reserved Variables)

액션 내에서 사용 가능한 주요 예약 변수들:



yytext

현재 매치된 렉심(입력 문자열 조각)을 가리키는 char* 포인터입니다. 예: [a-z]+ { printf("%s\n", yytext); } 는 매치된 소문자 문자열을 출력합니다.



yy leng

현재 매치된 렉심의 길이(문자 수)를 저장하는 정수형 변수입니다. 예: [a-z]+ { printf("%d\n", yy leng); } 는 매치된 문자열의 길이를 출력합니다.



yyin

입력 파일을 지정하는 FILE* 포인터입니다. 기본적으로 표준 입력 (stdin)입니다.

Lex의 모호성 해결 (Ambiguity Resolution)

입력 스트림에서 여러 개의 정규 표현식이 동시에 매치될 수 있는 경우, Lex는 다음과 같은 규칙에 따라 모호성을 해결합니다:



가장 긴 렉심 규칙 (Longest lexeme rule)

가장 긴 길이의 문자열을 매치하는 규칙이 우선적으로 선택됩니다.



우선순위 규칙 (Precedence rule)

매치된 문자열 길이가 동일한 경우, Lex 소스 파일에서 먼저 나타나는 규칙이 우선권을 가집니다.



선행 탐색 (Lookahead, /)

r1/r2 형태로 사용하여, r1이 r2에 의해 뒤따라올 때만 r1을 매치하도록 명시적으로 지정할 수 있습니다. 이는 Fortran의 DO 루프와 같은 특정 문맥에서만 키워드를 인식하는 경우에 유용합니다.

예를 들어, DO 키워드가 숫자와 변수 다음에 올 때만 루프 키워드로 인식하도록 할 수 있습니다.

시작 조건 (Start Conditions)

Lex는 규칙에 시작 조건(mode)을 지정하여 다른 문맥(상태)에서 다른 어휘 규칙을 적용할 수 있도록 합니다.

- **%s condition_name:** 사용자 정의 시작 조건을 선언합니다 (%s는 Exclusive start condition, %x는 Inclusive start condition)
- **regular_expression {action}:** 특정 시작 조건에서만 활성화되는 규칙을 정의합니다
- **BEGIN condition_name:** 현재 시작 조건을 변경하는 매크로입니다. BEGIN 0은 초기 상태로 돌아갑니다

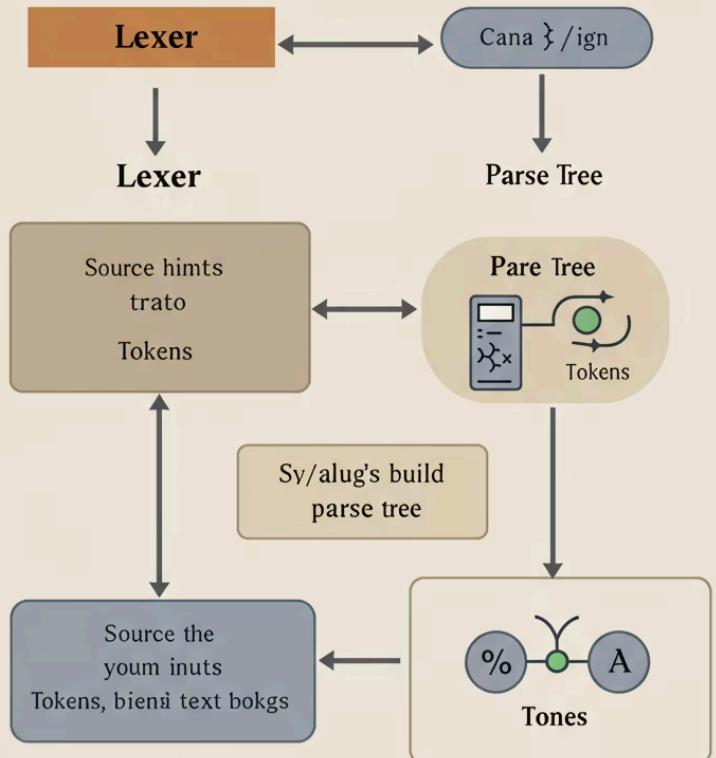
Lex와 Yacc의 연동

Lex는 종종 Yacc(또 다른 컴파일러 도구, 파서 생성기)와 함께 사용됩니다.

Lex(Lexer)는 입력 소스 코드를 스캔하여 토큰 스트림을 생성하고, Yacc(Parser)는 이 토큰 스트림을 입력받아 문법 규칙에 따라 프로그램의 구조(파스 트리 등)를 분석합니다.

Yacc가 토큰이 필요할 때 Lex가 생성한 `yylex()` 함수를 호출하여 다음 토큰을 얻어옵니다. Parser가 Lexer 위에 있는 구조라고 설명됩니다.

Pegnoutil



Compluteer Front-end

어휘 분석의 중요성

어휘 분석은 컴파일러 프런트엔드의 필수 단계입니다. 소스 코드를 의미 있는 단위(토큰)로 나누어 구문 분석(Syntax Analysis)이 처리하기 쉬운 형태로 만듭니다.

또한, 이 단계에서 주석이나 공백 문자를 제거하고(보통 액션에서 무시함으로써), 기본적인 오류(예: 인식할 수 없는 문자 시퀀스)를 탐지할 수 있습니다.

정규 표현식과 유한 상태 기계 이론을 통해 효율적이고 정확한 어휘 분석기를 구현할 수 있으며, Lex와 같은 도구는 이 과정을 자동화하여 컴파일러 개발을 용이하게 합니다.

정규 표현식 해석 연습

다음 Lex 스타일 정규 표현식이 매치할 수 있는 문자열들의 예시를 살펴보고, 각 표현식에서 사용된 연산자들의 의미를 이해해 봅시다.

1 $[A-Za-z]^+$

대문자 A-Z 또는 소문자 a-z 중 하나를 매치하는 문자 클래스에 + 연산자를 적용한 것으로, 알파벳 대소 문자로만 구성된 길이가 1 이상인 문자열을 매치합니다.

매치 예시: hello, Compiler, a, Variable

2 $a(b|c)^*d$

'a'로 시작하고, 그 뒤에 'b' 또는 'c'가 0회 이상 반복되며, 마지막에 'd'로 끝나는 문자열을 매치합니다.
매치 예시: ad, abd, acd, abbcd, acbcd

3 num: digit+.(digit+)?
 $(E(+|-)?digit+)?$

표준적인 숫자 상수(정수, 소수, 과학적 표기법) 패턴을 매치합니다.
정수로 시작하며, 소수점 부분은 선택 사항이고, 지수 부분도 선택 사항입니다.

매치 예시: 123, 45.67, 8e3, 9.0E-2, 1e+5

Lex 규칙과 액션 연습

다음 Lex 규칙들이 입력 스트림 "abc 123 xyz 45"에 대해 순서대로 적용될 때의 결과를 살펴봅시다.

```
%%
[a-z]+ { printf("Identifier: %s (len %d)\n", yytext, yyleng); }
[1, 2, 7, 8, 14, 17-20]+ { printf("Integer: %s (len %d)\n", yytext, yyleng); }
. { /* ignore other characters like space */ }
%%
```

Lex는 가장 긴 매치를 우선하고, 길이 같으면 먼저 나오는 규칙을 우선합니다.

Lex 규칙 적용 과정 분석

입력 스트림 "abc 123 xyz 45"에 대한 Lex 규칙 적용 과정을 단계별로 살펴봅시다:

1. 입력 시작: "abc"
 - 규칙 1 [a-z]+는 "abc"를 매치할 수 있습니다. 길이 3.
 - 규칙 2 [1, 2, 7, 8, 14, 17-20]+는 "abc"를 매치할 수 없습니다.
 - 규칙 3 .는 "a"를 매치할 수 있습니다. 길이 1.
 - 가장 긴 매치는 "abc" (길이 3)이므로 규칙 1이 선택됩니다.
 - 결과: "Identifier: abc (len 3)" 출력
2. 다음 문자는 공백: 규칙 3이 선택되어 무시됩니다.
3. 다음 입력: "123"
 - 규칙 2가 선택되어 "Integer: 123 (len 3)" 출력
4. 다음 문자는 공백: 규칙 3이 선택되어 무시됩니다.
5. 다음 입력: "xyz"
 - 규칙 1이 선택되어 "Identifier: xyz (len 3)" 출력
6. 다음 문자는 공백: 규칙 3이 선택되어 무시됩니다.
7. 다음 입력: "45"
 - 규칙 2가 선택되어 "Integer: 45 (len 2)" 출력

최종 출력: "Identifier: abc (len 3)", "Integer: 123 (len 3)", "Identifier: xyz (len 3)", "Integer: 45 (len 2)"

NFA 동작 추적 연습

다음 간단한 NFA (ϵ -전이 없음)가 있다고 가정합니다:

- 상태 집합 $Q = \{q_0, q_1, q_2, q_3\}$
- 알파벳 $\Sigma = \{a, b\}$
- 시작 상태 q_0
- 최종 상태 집합 $F = \{q_3\}$
- 전이 함수 δ :
 - $\delta(q_0, a) = \{q_0, q_1\}$
 - $\delta(q_0, b) = \{q_0\}$
 - $\delta(q_1, b) = \{q_2\}$
 - $\delta(q_2, b) = \{q_3\}$
 - 그 외 $\delta(\text{상태}, \text{기호}) = \emptyset$ (공집합)

입력 문자열 "aab"에 대해 이 NFA가 수락하는지 여부를 판단해 봅시다.

NFA 동작 추적 과정

입력 문자열 "aab"에 대한 NFA 동작 추적:

1. 초기 상태: $\{q_0\}$
2. 첫 번째 입력 'a': $\delta(q_0, a) = \{q_0, q_1\}$. 현재 상태 집합은 $\{q_0, q_1\}$ 입니다.
3. 두 번째 입력 'a':
 - 상태 q_0 에서 'a': $\delta(q_0, a) = \{q_0, q_1\}$
 - 상태 q_1 에서 'a': $\delta(q_1, a) = \emptyset$
 - 합집합: $\{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$. 현재 상태 집합은 $\{q_0, q_1\}$ 입니다.
4. 세 번째 입력 'b':
 - 상태 q_0 에서 'b': $\delta(q_0, b) = \{q_0\}$
 - 상태 q_1 에서 'b': $\delta(q_1, b) = \{q_2\}$
 - 합집합: $\{q_0\} \cup \{q_2\} = \{q_0, q_2\}$. 현재 상태 집합은 $\{q_0, q_2\}$ 입니다.
5. 입력 문자열 "aab"를 모두 소비했습니다. 최종 도달 상태 집합은 $\{q_0, q_2\}$ 입니다.

이 집합에 최종 상태 q_3 이 포함되어 있지 않으므로, 입력 문자열 "aab"는 이 NFA에 의해 수락되지 않습니다.

NFA 구현의 어려움과 DFA 변환의 필요성

NFA를 직접 프로그램으로 구현하기 어려운 이유:

- NFA는 비결정적입니다. 즉, 특정 상태에서 특정 입력 기호에 대해 다음 상태가 여러 개일 수 있습니다.
- ϵ -전이가 있을 수 있어 입력 기호 소비 없이도 상태가 변할 수 있습니다.
- 이를 직접 구현하려면 여러 가능한 경로를 동시에 추적하거나 모든 가능성을 탐색하는 복잡한 알고리즘이 필요합니다.

DFA 변환 (Subset Construction)의 목적:

- NFA와 DFA는 동일한 언어를 인식하므로, NFA를 동일한 언어를 인식하는 DFA로 변환할 수 있습니다.
- DFA는 각 상태-입력 쌍에 대해 다음 상태가 유일하므로, 이를 프로그램으로 구현하기 매우 쉽고 효율적입니다.
- 정규 표현식을 효율적인 어휘 분석기(DFA)로 구현하기 위해 NFA를 중간 단계로 거쳐 DFA로 변환합니다.

질문 1: 토큰의 개념과 구성 요소

책을 덮고 내가 교수라면 여기서 어떤 문제를 낼까?를 상상하며 만든 질문입니다.

토큰이란 무엇이며, 어떤 요소로 구성되는가?

토큰(Token)은 프로그래밍 언어에서 의미를 가지는 최소 단위의 어휘 요소입니다. 소스 코드를 어휘 분석 과정에서 분할한 결과물로, 컴파일러가 구문 분석 단계에서 처리하는 기본 단위가 됩니다.

토큰은 다음과 같은 요소로 구성됩니다:

1. **토큰 유형(Token Type)**: 식별자, 키워드, 연산자, 상수, 구분자 등의 분류
2. **렉시م(Lexeme)**: 토큰에 해당하는 실제 문자열 (예: "main", "123", "+", "if")
3. **속성 값(Attribute Value)**: 토큰의 추가 정보 (예: 숫자 상수의 실제 값, 식별자의 심볼 테이블 인덱스)

질문 2: 렉시컬 분석기와 파서의 차이점

렉시컬 분석기(Lexical Analyzer)와 파서(Parser)의 차이점을 설명하시오.

렉시컬 분석기(Lexical Analyzer)와 파서(Parser)는 컴파일러의 프런트엔드에서 서로 다른 역할을 수행합니다:

렉시컬 분석기 (Lexer)

- 소스 코드를 문자 단위로 읽어 토큰으로 분할합니다
- 정규 언어(Regular Language)를 기반으로 합니다
- 유한 상태 기계(FSM)로 구현됩니다
- 공백, 주석 등을 제거합니다
- 토큰 스트림을 출력합니다

파서 (Parser)

- 토큰 스트림을 입력받아 구문 구조를 분석합니다
- 문맥 자유 문법(Context-Free Grammar)을 기반으로 합니다
- 푸시다운 오토마타(PDA)로 구현됩니다
- 구문 오류를 검출합니다
- 파스 트리 또는 추상 구문 트리(AST)를 생성합니다

비유하자면, 렉시컬 분석기는 문장을 단어로 나누는 작업을, 파서는 그 단어들의 문법적 관계를 파악하는 작업을 수행합니다.

질문 3: 컴파일러가 렙시컬 분석 단계를 거쳐야 하는 이유

컴파일러가 렙시컬 분석 단계를 거쳐야 하는 이유는 무엇인가?

컴파일러가 렙시컬 분석 단계를 거쳐야 하는 이유는 다음과 같습니다:



구문 분석 단순화

소스 코드를 의미 있는 토큰 단위로 나누어 구문 분석기(Parser)가 처리하기 쉬운 형태로 변환합니다. 이는 마치 책을 읽을 때 문자 하나하나를 보는 것보다 단어 단위로 인식하는 것이 더 효율적인 것과 같습니다.



불필요한 요소 제거

공백, 주석, 들여쓰기 등 구문 분석에 불필요한 요소를 제거하여 처리해야 할 데이터의 양을 줄입니다.



초기 오류 검출

잘못된 문자나 유효하지 않은 토큰 패턴을 조기에 발견하여 오류 보고를 할 수 있습니다.



모듈화 및 분업

컴파일러를 여러 단계로 나누어 각 단계가 특정 작업에 집중할 수 있게 합니다. 이는 컴파일러의 설계, 구현, 유지보수를 용이하게 합니다.



효율성 향상

정규 표현식과 유한 상태 기계를 사용하여 효율적으로 토큰을 인식할 수 있습니다. 이는 전체 컴파일 과정의 성능을 향상시킵니다.

비유하자면, 렙시컬 분석은 요리를 시작하기 전에 재료를 손질하고 준비하는 과정과 같습니다. 이 과정을 거치지 않으면 이후 요리 과정(구문 분석, 의미 분석 등)이 훨씬 복잡하고 어려워집니다.

나만의 모범 답안: 토큰의 개념과 구성 요소

토큰은 프로그래밍 언어에서 의미를 가지는 최소 단위의 어휘 요소로, 어휘 분석기가 소스 코드를 분석하여 생성하는 기본 단위입니다.

토큰의 구성 요소를 일상생활의 예로 설명하자면, 신분증과 비슷합니다:



토큰 유형 (Token Type)

신분증의 종류(주민등록증, 운전면허증, 여권)와 같이 토큰이 어떤 범주에 속하는지 나타냅니다. 예: 키워드(if, while), 식별자(변수명), 연산자(+, -), 구분자(;, {}, {}), 상수(123, "hello") 등



렉시م (Lexeme)

신분증에 적힌 실제 이름처럼, 소스 코드에서 토큰에 해당하는 실제 문자열입니다. 예: "main", "123", "+", "if"



속성 값 (Attribute Value)

신분증의 추가 정보(생년월일, 주소)처럼, 토큰에 대한 부가 정보입니다. 예: 숫자 상수 "123"의 실제 정수값 123, 식별자 "count"의 심볼 테이블 인덱스

나만의 모범 답안: 렉시컬 분석기와 파서의 차이점

렉시컬 분석기와 파서의 차이점을 음식 준비 과정에 비유해 설명해 보겠습니다:

렉시컬 분석기 (Lexer)

요리사가 재료를 씻고, 껍질을 벗기고, 적당한 크기로 자르는 **재료 준비 과정**과 같습니다.

- 문자 스트림을 의미 있는 토큰으로 분할
- 공백, 주석 등 불필요한 요소 제거
- 정규 표현식과 유한 상태 기계 사용
- 문맥에 독립적인 처리 (각 토큰은 독립적으로 인식)

렉시컬 분석기가 "무엇이 있는지" 식별한다면, 파서는 "어떻게 구성되어 있는지" 분석합니다. 렉시컬 분석기는 단어를 인식하고, 파서는 문장의 문법을 이해하는 것과 같습니다.

파서 (Parser)

준비된 재료를 레시피에 따라 조합하고 요리하는 **실제 조리 과정**과 같습니다.

- 토큰 스트림을 구문 구조로 조직화
- 문법 규칙에 따라 토큰 간의 관계 분석
- 문맥 자유 문법과 푸시다운 오토마타 사용
- 문맥에 의존적인 처리 (토큰의 위치와 관계가 중요)

나만의 모범 답안: 컴파일러가 렙시컬 분석 단계를 거쳐야 하는 이유

컴파일러가 렙시컬 분석 단계를 거쳐야 하는 이유를 도서관 책 정리 과정에 비유해 설명해 보겠습니다:

도서관에 수천 권의 책이 무작위로 쌓여 있다고 상상해 보세요. 이 책들을 효율적으로 분류하고 정리하려면 어떻게 해야 할까요?



1단계: 기본 분류 (렙시컬 분석)

먼저 책들을 소설, 과학, 역사, 예술 등 기본 카테고리로 분류합니다. 이는 소스 코드를 토큰으로 분할하는 렙시컬 분석과 같습니다.

2단계: 세부 정리 (구문 분석)

각 카테고리 내에서 저자, 출판년도 등에 따라 책을 정리합니다. 이는 토큰들의 관계를 파악하는 구문 분석과 같습니다.

3단계: 카탈로그 작성 (의미 분석)

책의 내용, 주제, 키워드 등을 분석하여 카탈로그를 만듭니다. 이는 프로그램의 의미를 분석하는 의미 분석과 같습니다.

렙시컬 분석 없이 바로 구문 분석을 시도하는 것은, 수천 권의 책을 한꺼번에 저자별, 주제별로 정리하려는 것과 같습니다. 이는 매우 비효율적이고 오류가 발생하기 쉽습니다.

렙시컬 분석은 복잡한 문제를 작은 단위로 나누어 해결하는 '분할 정복(divide and conquer)' 전략의 첫 단계입니다. 이를 통해 컴파일러의 설계가 더 모듈화되고, 각 단계가 더 효율적으로 작동할 수 있습니다.

정규 표현식과 유한 상태 기계의 관계

정규 표현식과 유한 상태 기계는 동전의 양면과 같습니다. 같은 정규 언어를 다른 방식으로 표현하는 것입니다.

이 관계를 교통 시스템에 비유해 보겠습니다:

- 정규 표현식은 목적지까지 가는 경로를 설명하는 **여행 가이드**와 같습니다. "A 거리를 따라가다가 B 또는 C 거리로 갈 수 있고, 그 후 D 거리로 가세요."
- 유한 상태 기계는 같은 정보를 **도로 지도**로 표현한 것입니다. 각 교차로(상태)와 도로(전이)가 명확하게 표시되어 있습니다.

정규 표현식은 패턴을 간결하게 정의하기 좋고, 유한 상태 기계는 패턴을 효율적으로 인식하기 좋습니다. Lex와 같은 도구는 정규 표현식으로 정의된 패턴을 유한 상태 기계로 변환하여 효율적인 어휘 분석기를 생성합니다.

NFA와 DFA의 차이점

NFA(비결정적 유한 오토마타)와 DFA(결정적 유한 오토마타)의 차이점을 일상생활에 비유해 설명해 보겠습니다:

NFA (비결정적 유한 오토마타)

여러 가능성을 동시에 탐색하는 모험가와 같습니다.

- 하나의 입력에 대해 여러 가능한 경로 존재
- 입력 없이도 상태 변경 가능 (ϵ -전이)
- 설계가 직관적이고 간단함
- 구현이 복잡하고 비효율적일 수 있음

DFA (결정적 유한 오토마타)

정해진 규칙만 따르는 체계적인 여행자와 같습니다.

- 하나의 입력에 대해 정확히 하나의 경로만 존재
- 모든 전이는 입력 기호 필요
- 설계가 복잡할 수 있음
- 구현이 간단하고 효율적

NFA는 "이 길도 가보고, 저 길도 가보자"라는 접근 방식이라면, DFA는 "이 상황에서는 반드시 이 길로만 가야 한다"는 접근 방식입니다. 두 방식 모두 같은 목적지(정규 언어 인식)에 도달할 수 있지만, 접근 방식이 다릅니다.

Lex 파일의 구조와 작동 원리

Lex 파일의 구조와 작동 원리를 요리 레시피에 비유해 설명해 보겠습니다:

1

선언 섹션 (Declarations)

요리 재료와 준비물 목록과 같습니다. 필요한 도구(헤더 파일), 재료(정규 정의), 사전 준비 사항(변수, 상수 선언) 등을 명시합니다.



규칙 섹션 (Rules)

요리 과정 설명과 같습니다. "이런 재료가 나오면 이렇게 조리하라"는 지시사항으로, 정규 표현식과 그에 대응하는 액션으로 구성됩니다.



사용자 서브루틴 섹션 (User Subroutines)

요리사의 특별한 기술과 노하우와 같습니다. 규칙 섹션에서 사용할 수 있는 사용자 정의 함수들과 메인 함수를 포함합니다.

Lex는 이 "레시피"를 바탕으로 C 코드(`lex.yy.c`)를 생성하고, 이 코드는 컴파일되어 어휘 분석기 프로그램이 됩니다. 이 프로그램은 입력 텍스트를 스캔하며 정의된 패턴을 찾고, 매치될 때마다 해당 액션을 실행합니다.



Lex의 모호성 해결 규칙

Lex에서 여러 규칙이 동시에 매치될 때 어떤 규칙을 선택할지 결정하는 모호성 해결 규칙을 살펴봅시다:



가장 긴 렉심 규칙 (Longest lexeme rule)

가장 긴 길이의 문자열을 매치하는 규칙이 우선적으로 선택됩니다.

예: "if"와 "identifier"([a-z]+) 패턴이 모두 "ifelse"를 매치할 수 있다면, "ifelse" 전체를 매치하는 identifier 규칙이 선택됩니다.

2

우선순위 규칙 (Precedence rule)

매치된 문자열 길이가 동일한 경우, Lex 소스 파일에서 먼저 나타나는 규칙이 우선권을 가집니다.

예: "if"가 키워드 규칙과 식별자 규칙 모두에 매치된다면, Lex 파일에서 먼저 정의된 규칙이 선택됩니다.

이는 마치 경주에서 1) 가장 멀리 간 선수가 이기고, 2) 동점이면 먼저 출발한 선수가 이기는 규칙과 같습니다.

선행 탐색(Lookahead)의 활용

선행 탐색(Lookahead, /)은 현재 패턴 뒤에 특정 패턴이 따라올 때만 매치하도록 하는 Lex의 기능입니다.

이를 일상생활에 비유하면, "앞사람이 우산을 들고 있을 때만 비가 온다고 판단하는 것"과 같습니다.

예를 들어, Fortran에서 DO 문은 문맥에 따라 다르게 해석될 수 있습니다:

```
DO/[0-9]+[ \t]+{id}[ \t]*= { return DO_LOOP; }
{id}                  { return ID; }
```

이 규칙은 "DO" 뒤에 숫자, 공백, 식별자, 공백, 등호가 따라올 때만 DO_LOOP 토큰으로 인식하고, 그렇지 않으면 일반 식별자로 인식합니다.

선행 탐색은 토큰의 의미가 뒤따르는 문맥에 따라 달라질 때 유용하게 사용됩니다.

시작 조건(Start Conditions)의 활용

시작 조건(Start Conditions)은 Lex에서 다른 상태(문맥)에 따라 다른 규칙 집합을 적용할 수 있게 하는 기능입니다.

이를 교통 신호등에 비유해 보겠습니다:

- 초기 상태(INITIAL): 평소 신호등 - 일반적인 토큰 규칙 적용
- 특수 상태(예: COMMENT): 공사 중 특별 신호등 - 특수한 규칙 적용
- BEGIN 매크로: 신호등 모드 전환 버튼 - 상태 전환

C 언어 주석 처리를 위한 시작 조건 예제:

```
%s COMMENT
%%
"/**"      { BEGIN COMMENT; /* 주석 모드로 전환 */ }
"*/" { BEGIN 0; /* 초기 상태로 복귀 */ }
.   { /* 주석 내용 무시 */ }
\n { /* 주석 내 개행 무시 */ }
[a-zA-Z]+ { return ID; /* 일반 식별자 인식 */ }
%%
```

이 방식으로 주석 내부와 외부에서 서로 다른 규칙을 적용할 수 있습니다.

Lex와 Yacc의 협력 관계

Lex와 Yacc의 협력 관계를 공장 생산 라인에 비유해 설명해 보겠습니다:

Lex (어휘 분석기)

원자재 가공 공정과 같습니다.

- 소스 코드(원자재)를 토큰(부품)으로 분할
- yylex() 함수를 통해 토큰을 하나씩 생산
- 토큰 유형을 반환하고 yytext에 렉시 저장

Yacc (구문 분석기)

부품 조립 공정과 같습니다.

- Lex가 생산한 토큰(부품)을 조립
- 필요할 때마다 yylex()를 호출하여 다음 토큰 요청
- 문법 규칙에 따라 파스 트리(완제품) 생성

이 두 도구의 협력은 컴파일러의 프런트엔드를 효율적으로 구현하는 데 필수적입니다. Lex는 "무엇이 있는지" 식별하고, Yacc는 "어떻게 구성되어 있는지" 분석합니다.

어휘 분석기 설계 시 고려사항

효율적인 어휘 분석기를 설계할 때 고려해야 할 주요 사항들을 알아봅시다:



성능 최적화

어휘 분석은 컴파일러에서 반복적으로 수행되는 작업이므로, DFA 기반의 효율적인 구현이 중요합니다. 최소 상태 DFA를 사용하여 메모리와 처리 시간을 절약할 수 있습니다.



오류 처리

인식할 수 없는 문자나 패턴을 만났을 때 적절한 오류 메시지를 제공하고, 가능하면 복구하여 분석을 계속할 수 있어야 합니다.



문맥 처리

일부 언어에서는 토큰의 의미가 문맥에 따라 달라질 수 있습니다. 선행 탐색(/)이나 시작 조건을 활용하여 이러한 상황을 처리할 수 있습니다.



파서와의 통합

어휘 분석기는 파서와 효율적으로 통합되어야 합니다. 토큰 유형, 속성 값 등을 파서가 이해할 수 있는 형태로 전달해야 합니다.

어휘 분석기 설계는 컴파일러의 전체 성능과 사용성에 큰 영향을 미치므로, 이러한 고려사항들을 균형 있게 반영하는 것이 중요합니다.

정규 표현식의 한계

정규 표현식은 강력한 도구이지만, 표현할 수 있는 언어의 범위에 한계가 있습니다:

중첩 구조 표현 불가

정규 표현식은 중첩된 괄호의 균형을 맞추는 것과 같은 중첩 구조를 표현할 수 없습니다. 예: $\{a^n b^n \mid n \geq 0\}$ (a가 n번, b가 n번 나타나는 패턴)

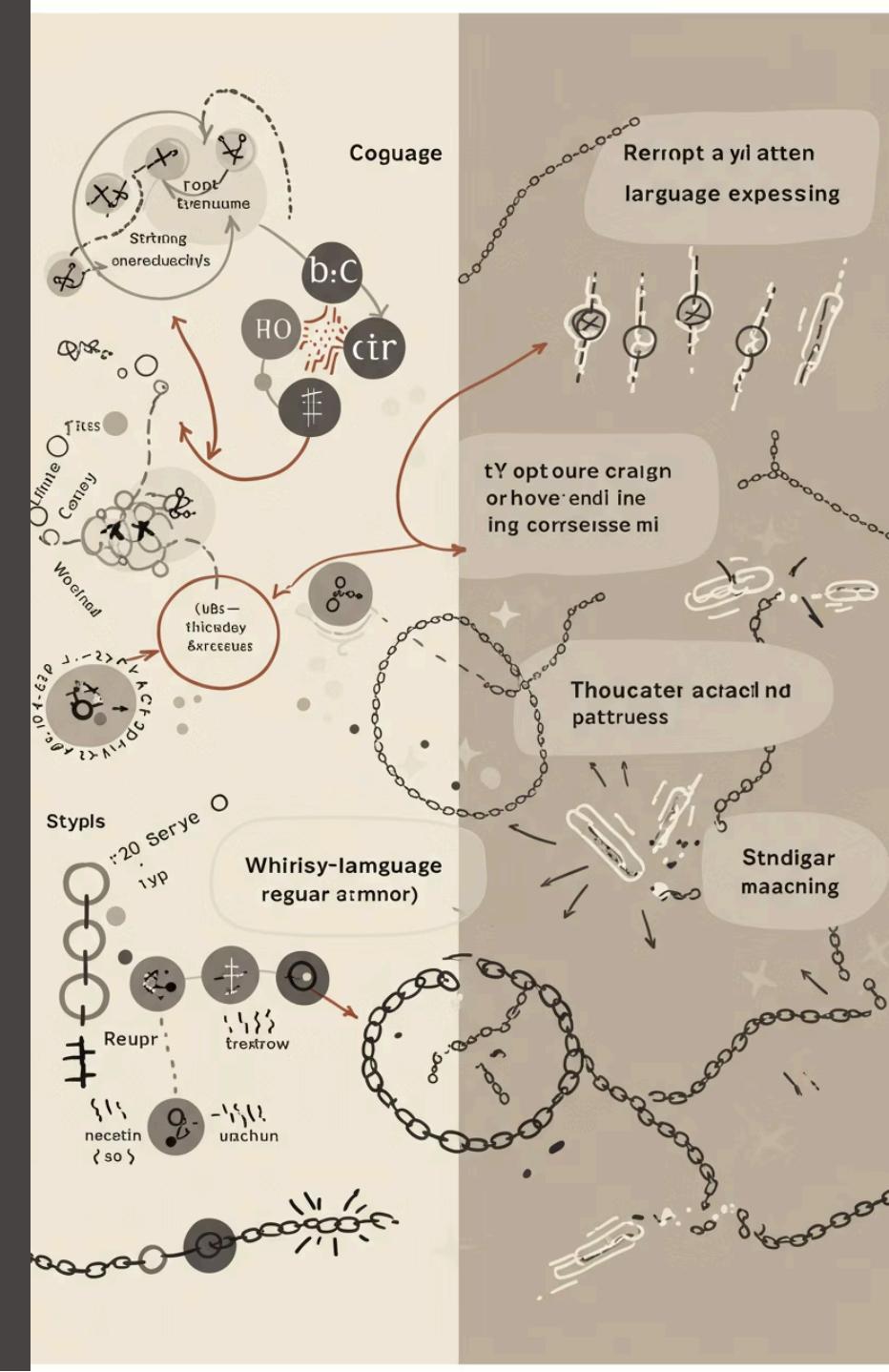
문맥 의존성 처리 제한

정규 표현식은 기본적으로 문맥 독립적입니다. 이전에 나온 패턴에 따라 다른 패턴을 매치해야 하는 경우 한계가 있습니다.

재귀적 패턴 표현 불가

재귀적으로 정의되는 패턴(예: 프로그래밍 언어의 표현식)은 정규 표현식으로 완전히 표현할 수 없습니다.

이러한 한계 때문에, 프로그래밍 언어의 구문 분석에는 정규 표현식보다 더 강력한 문맥 자유 문법(Context-Free Grammar)이 사용됩니다. 어휘 분석은 정규 표현식으로 충분하지만, 구문 분석은 더 강력한 형식 체계가 필요합니다.



어휘 분석과 정규 표현식의 실제 응용

어휘 분석과 정규 표현식은 컴파일러 외에도 다양한 분야에서 활용됩니다:



텍스트 검색 및 치환

grep, sed, awk와 같은 유닉스 도구들은 정규 표현식을 사용하여 텍스트 파일에서 패턴을 검색하고 치환합니다.



데이터 유효성 검사

이메일 주소, 전화번호, 우편번호 등의 형식을 검증하는 데 정규 표현식이 널리 사용됩니다.



코드 에디터의 구문 강조

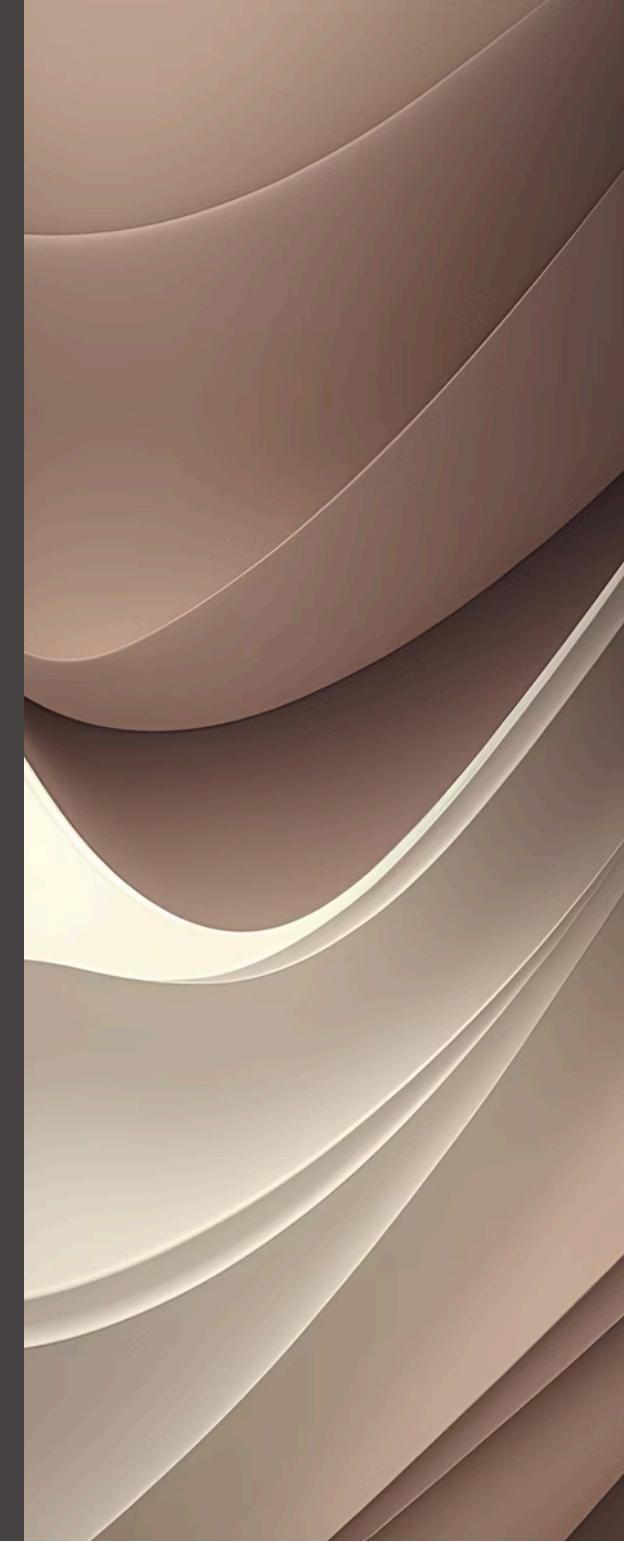
프로그래밍 에디터의 구문 강조 기능은 어휘 분석과 유사한 과정을 통해 코드의 다른 부분(키워드, 문자열, 주석 등)을 식별하고 다르게 표시합니다.



자연어 처리

토큰화(tokenization)는 자연어 처리의 기본 단계로, 텍스트를 단어나 문장으로 분할하는 과정입니다.

이처럼 어휘 분석과 정규 표현식의 개념은 컴퓨터 과학의 여러 분야에서 기본적이고 중요한 역할을 합니다.



어휘 분석기 구현 예제

간단한 계산기 언어를 위한 Lex 파일 예제를 살펴봅시다:

```
%{  
#include "y.tab.h" /* Yacc/Bison이 생성한 헤더 파일 */  
int line_num = 1;  
%}  
  
%%  
[0-9]+ { yyval = atoi(yytext); return NUMBER; }  
[0-9]+\.[0-9]+ { yyval = atof(yytext); return FLOAT; }  
"+" { return PLUS; }  
"-" { return MINUS; }  
"*" { return TIMES; }  
"/" { return DIVIDE; }  
 "(" { return LPAREN; }  
 ")" { return RPAREN; }  
[ \t]+ { /* 공백 무시 */ }  
\n { line_num++; }  
. { printf("Unknown character: %s\n", yytext); }  
%%  
  
int yywrap() {  
    return 1; /* 입력 파일의 끝 */  
}
```

이 Lex 파일은 숫자, 연산자, 괄호 등을 인식하여 적절한 토큰 유형을 반환합니다. 공백은 무시하고, 개행 문자는 라인 번호를 증가시키며, 인식할 수 없는 문자는 오류 메시지를 출력합니다.

어휘 분석의 미래 동향

어휘 분석과 관련된 최신 동향과 미래 방향을 살펴봅시다:



기계 학습 기반 접근법

전통적인 규칙 기반 어휘 분석 외에도, 기계 학습을 활용하여 더 유연하고 적응력 있는 어휘 분석기를 개발하는 연구가 진행되고 있습니다.



증분식 어휘 분석

대화형 개발 환경에서는 코드가 변경될 때마다 전체를 다시 분석하는 대신, 변경된 부분만 효율적으로 재분석하는 증분식 어휘 분석 기법이 중요해지고 있습니다.



다국어 및 특수 언어 지원

다양한 문자 집합과 언어 특성을 지원하는 어휘 분석기의 필요성이 증가하고 있습니다. 특히 비 라틴 문자 기반 언어나 특수 도메인 언어에 대한 지원이 중요해지고 있습니다.



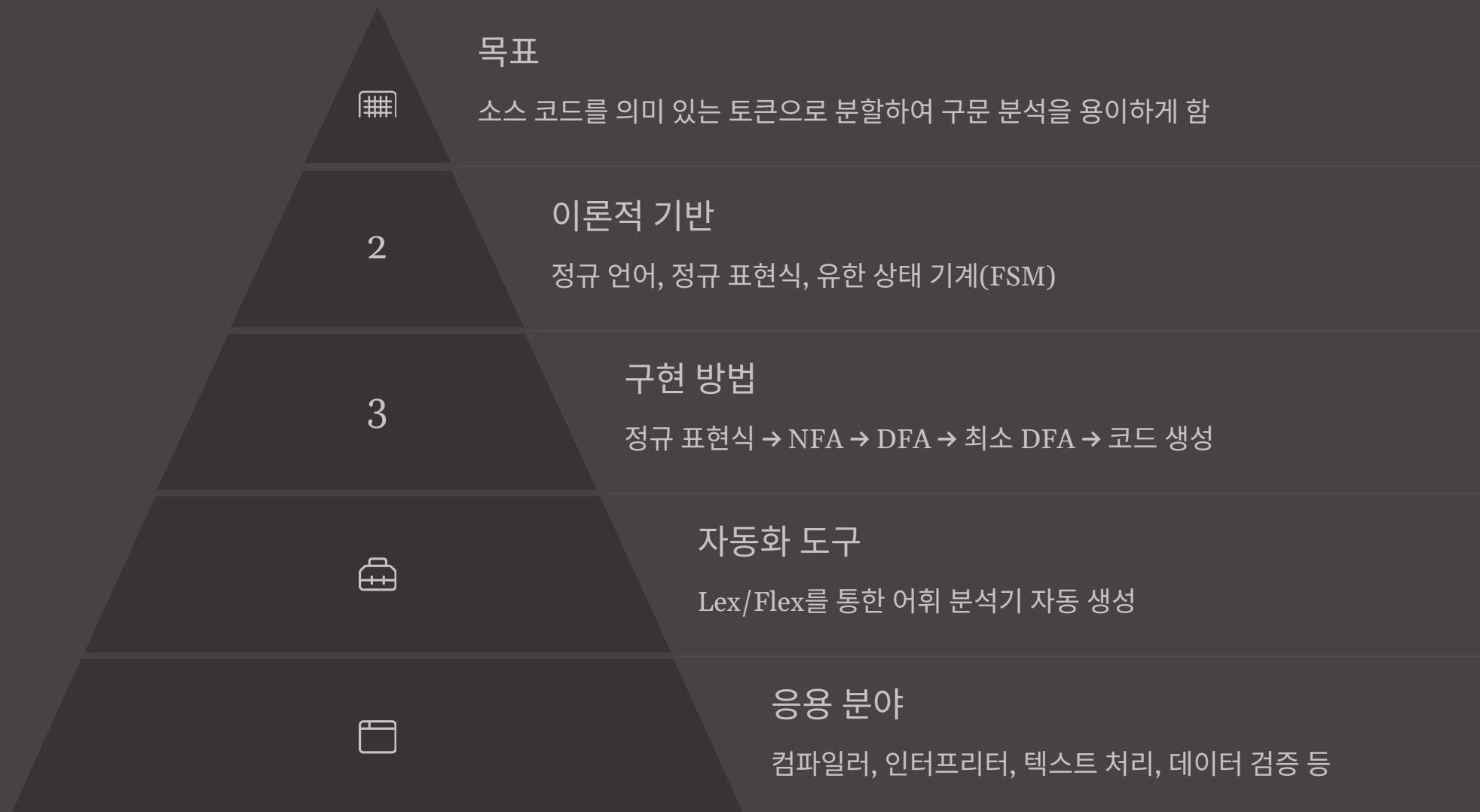
향상된 오류 복구 및 제안

단순히 오류를 보고하는 것을 넘어, 가능한 수정 방법을 제안하거나 자동으로 복구하는 지능형 어휘 분석기의 개발이 진행되고 있습니다.

이러한 동향은 프로그래밍 언어 처리의 효율성과 사용자 경험을 향상시키는 데 기여할 것으로 예상됩니다.

어휘 분석 학습의 핵심 요약

어휘 분석과 Lex 도구에 대한 학습 내용을 최종적으로 요약해 봅시다:



어휘 분석은 컴파일러의 첫 단계이자 기초로, 이후의 모든 분석 단계에 영향을 미치는 중요한 과정입니다. 정규 표현식과 유한 상태 기계의 이론적 이해를 바탕으로, Lex와 같은 도구를 활용하여 효율적인 어휘 분석기를 구현할 수 있습니다.