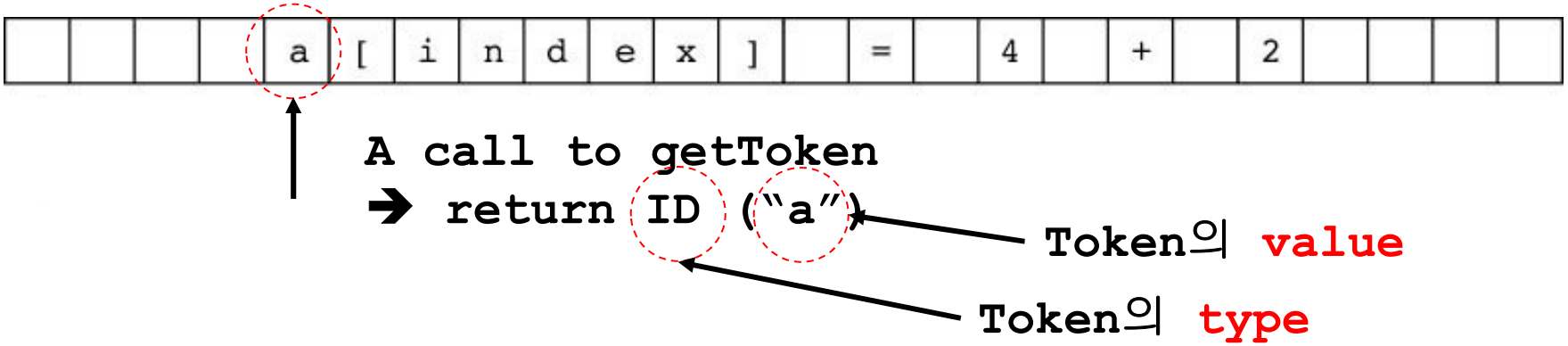
Chapter 4

어휘 분석

목차

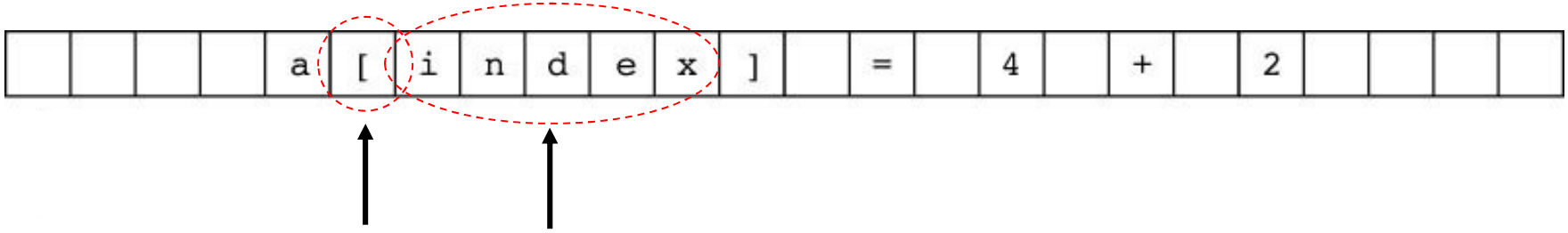
어휘 분석 과정

- **Scanner는 원시 프로그램에서 모든 토큰을 한꺼번에 다 찾는 게 아니라**
  - Parser 가 요청할 때마다
    - **`getToken`** 함수 호출 : **`TokenType getToken(void);`**
  - 토큰을 찾아 parser 에게 전달



어휘 분석 과정

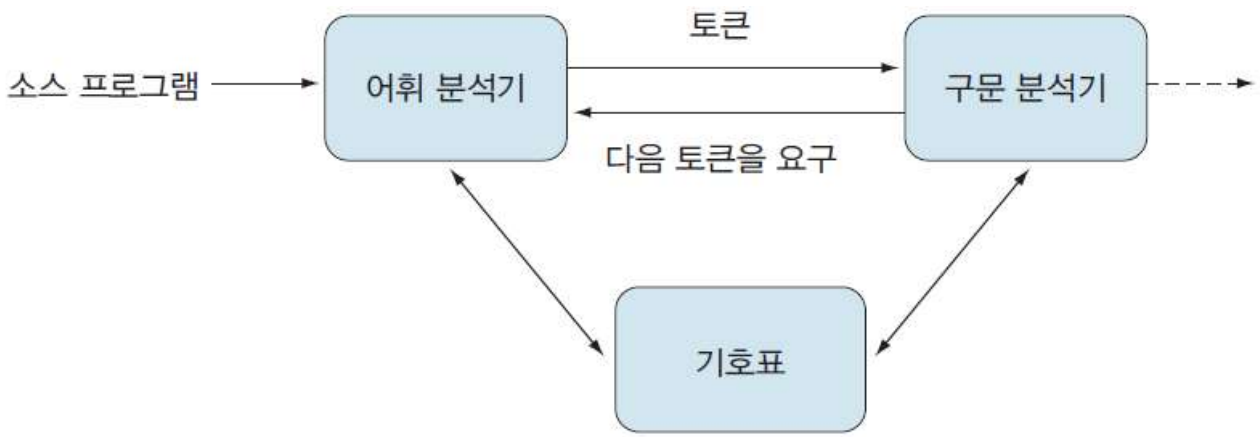
- **Scanner는 원시 프로그램에서 모든 토큰을 한꺼번에 다 찾는 게 아니라**
  - Parser 가 요청할 때마다
    - **`getToken`** 함수 호출
  - 토큰을 찾아 parser 에게 전달



Lexical Analysis (어휘분석) (1/2)

정의

- 소스 프로그램에서 토큰(token)을 찾음



기능

- 구문 분석기(parser) 요청이 있을 때마다 토큰을 찾아 전달
- 주석 및 white space 제거
- 기호표 (symbol table) 구성

Lexical Analysis (2/2)

■ 용어

- 토큰(Token) : terminal 기호로만 구성. **문법적으로 의미 있는 최소 단위.**
- 패턴(Pattern) : token 특징을 표현하는 규칙(rule).

  규칙 ➔ 정규 표현 (즉, 정규 문법)
- 렉심(Lexeme) : 지정한 pattern과 일치하는(matching) 문자열.

Delimiter (구분기호)

정의

- Token에 속한 문자열이 이 기호 왼쪽 문자까지임을 알려주는 역할

예

- `xtemp=ytemp`
  - '=' 이 구분기호
- `while x . . .`
  - 공백(*blank*)이 구분기호

  whitespace = (newline | blank | tab | comment)+
- `do/* */if`
  - 주석(*comment*)이 구분기호

Lookahead

- **Delimiters *end token strings* but they are *not part of the token itself*.**
  - 예 1: `xtemp=ytemp`
    - = 는 delimiter이면서 다음 token의 일부
  - 예 2: `x+=2;`
    - +는 delimiter이면서 `+=`가 다음 token
- **현재 token에 속하지 않는다고 (입력 버퍼에서) 읽은 문자를 버리지 않음**
  - 먼저 살펴보고(*lookahead*)
    - Token 의 일부이면 지금 처리
    - 아니면 *delimiter* ➔ 다음 token 을 찾을 때 사용

유한 오토마타(FA)를 정규 문법으로 변환

[입력] 유한 오토마타 $M = (Q, \Sigma, \delta, q_0, F)$

Q : 상태들의 유한집합

Σ : 입력 기호들의 유한집합

δ : 상태전이 함수

q_0 : 종결기호

[출력] 정규 문법 $G = (V_N, V_T, P, S)$

V_N : Nonterminal 기호들의 집합

V_T : Terminal 기호들의 집합

P : 생성규칙의 집합

S : 시작기호

[방법]

$V_N = Q$

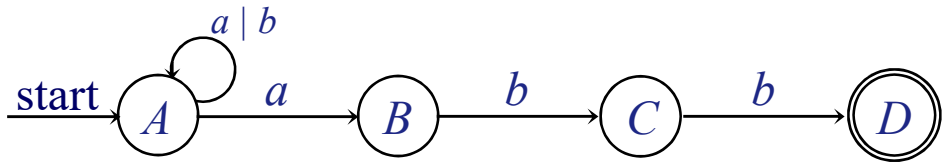
$V_T = \Sigma$

$S = q_0$

P : if $\delta(q, a) = r$, then $q \rightarrow ar$;

if $q \in F$, then $q \rightarrow \varepsilon$;

유한 오토마타(FA)를 정규 문법으로 변환 : 예



➔ G = (V_N, V_T, P, S)

V_N = {A, B, C, D}

V_T = {a, b}

S = A

P : A → aA | bA | aB

B → bC

C → bD

D → ε

정규 문법을 정규 표현으로 변환

[입력] 정규 문법 $G = (V_N, V_T, P, S)$

V_N : Nonterminal 기호들의 유한 집합

V_T : Terminal 기호들의 유한 집합

P : 생성 규칙들의 집합

$A \rightarrow tB$, $A \rightarrow t$ 혹은 $A \rightarrow Bt$, $A \rightarrow t$

단, $t \in V_T^*$, $A, B \in V_N$

S : 시작 기호. 단, $S \in V_N$.

[출력] 정규 문법 G가 생성하는 언어 $L(G)$를 나타내는 정규 표현

[방법]

(1) 정규 문법의 규칙을 정규 표현 방정식으로 바꾼다.

(2) 정규 표현 방정식 중 $\mathbf{X = \alpha X + \beta}$ 형태를 찾아 $\mathbf{X = \alpha^* \beta}$ 로 변환한다.

(3) 시작 기호로부터 출발하여 다른 방정식들을 차례대로 대입하면서 $X = \alpha X + \beta$ 형태의 식이 나타나면 $X = \alpha^* \beta$ 로 변환한다.

정규 문법을 정규 표현으로 변환 : 예 1

정규 문법 G = ({S, A, B}, {a, b}, P, S)

$S \rightarrow aA \mid bS$

$A \rightarrow aS \mid bB$

$B \rightarrow aB \mid bB \mid \varepsilon$

각 생성 규칙을 정규 표현 방정식으로 변환

$S = aA + bS$

$A = aS + bB$

$B = aB + bB + \varepsilon$ ➔ $B = (a+b)B + \varepsilon$ ➔ $B = (a+b)^* \varepsilon = (a+b)^*$

시작 기호 S 에 대한 식에서

$S = aA + bS = a(\mathbf{aS + bB}) + bS$

$= aaS + abB + bS = aaS + ab\,\mathbf{(a+b)^*} + bS$

$= (aa+b)S + ab(a+b)^*$

위 식은 $X = \alpha X + \beta$ 형태의 식이므로 $S = (aa+b)^* ab(a+b)^*$

$\therefore L(G) = \mathbf{(aa+b)^* ab(a+b)^*}$

정규 문법을 정규 표현으로 변환 : 예 2

정규 문법 G = ({S, A, B}, {a, b}, P, S)

S → aA | bS

A → aA | bA | b

각 생성 규칙을 정규 표현 방정식으로 변환

S = aA + bS

A = aA + bA + b ➔ A = (a+b)A + b ➔ A = (a + b)*b

시작 기호 S 에 대한 식에서

S = aA + bS

= a(a + b)*b + bS

= bS + a(a + b)*b

위 식은 X = αX + β 형태의 식이므로 S = b*a(a + b)*b

∴ L(G) = **b*a(a + b)*b**

정규 표현 방정식으로부터 해를 구하는 방법

- **α, β 가 정규 표현이고, α가 ε을 포함하지 않는다면**
- **X= αX+ β 의 유일한 해(unique solution)는 X = α^* β 이다.**
  - **[증명]** 정규 표현 방정식 X = αX + β 의 해가 X = $\alpha^*\beta$ 임을 증명

$$X = \alpha X + \beta = \alpha(\alpha^* \beta) + \beta$$

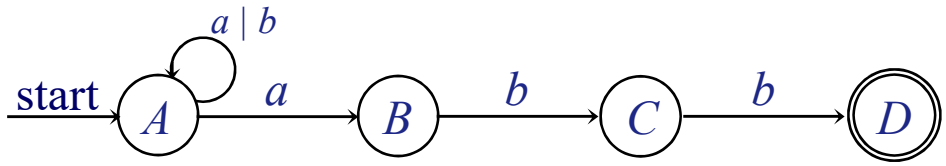
$$= \alpha^+ \beta + \beta$$

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

$$= \alpha^* \beta$$

∴ X= αX + β 의 해는 α^* β 이다.

앞의 유한 오토마타(FA)를 정규 표현으로 변환 : 예



➔ G = (V_N, V_T, P, S), V_N = {A, B, C, D}, V_T = {a, b}, S = A

A → aA | bA | aB ➔ A = aA + bA + aB

B → bC ➔ B = bC = bb

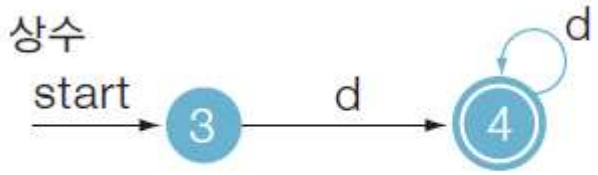
C → bD ➔ C = bD = bε = b

D → ε ➔ D = ε

A = aA + bA + aB = (a+b)A + abb

∴ **L(A) = (a+b)*abb**

정수 인식



- **DFA를 정규 문법으로 변환**

  S → dC

  C → dC | ε

- **정규 정규 문법을 정규 표현으로 변환**

  S = dC

  C = dC + ε

- **DFA가 인식하는 언어 : L(G) = d+**

  C = d*, S = dC = dd* = d+

토큰 인식

- **토큰의 인식**
  - 토큰의 구조 : 정규 표현(즉 정규 문법)을 사용해서 표현
  - 토큰 인식기 : 정규 표현을 인식하는 유한 오토마타를 구성
- **정규 문법으로 나타냄**

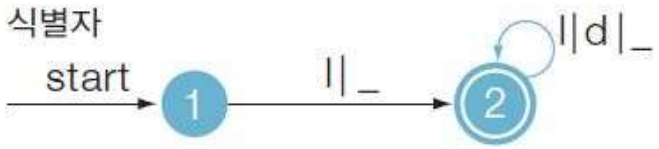
  <ident> ::= (<letter> | _) { <letter> | <digit> | _ }

  <letter> ::= a | b | c | ... | z | A | B | C | ... | Z

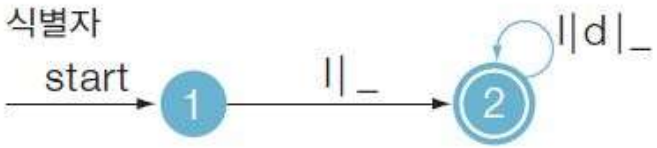
  <digit> ::= 0 | 1 | 2 | ... | 9

  letter : l 로, digit : d로 표현

- **Identifier(식별자) 인식 유한 오토마타**



Identifier(식별자) 인식



- **DFA를 정규 문법으로 변환**

  상태(state)는 Nonterminal에 해당

  상태 1 ➔ Nonterminal S, 상태 2 ➔ Nonterminal A

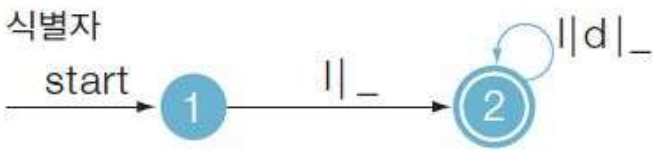
  S → l A | _ A

  A → l A | d A | _ A

  종결 상태일 경우 A → ε **추가**

  A → l A | d A | _ A | ε

Identifier(식별자) 인식



- **DFA를 정규 문법으로 변환**

  S → l A | _ A

  A → l A | d A | _ A | ε

- **정규 문법을 정규 표현으로 변환**

  S = lA + _A = (l + _)A

  A = lA + dA + _A + ε = (l + d + _)A + ε

- **DFA가 인식하는 언어 : L(G) = (l + _)(l + d + _)***

  A = (l + d + _)A + ε = (l + d + _)*

  ∴ S = (l + _)A = **(l + _)(l + d + _)***

  X= αX + β 의 해는 α^* β

어휘분석기의 설계 및 구현