컴파일러의 front end 쪽은
새롭게 개발할 일이 적다. back end 쪽은 frontend 에 비해 복잡하다.
(IR)

4. Lexical Analysis

목차 및 학습 목표

- **목차**
  - 어휘분석의 개요
  - 토큰의 인식
  - 어휘분석기의 설계 및 구현

- **학습 목표**
  - 어휘분석기를 이해할 수 있다.
  - 각각의 토근들에 대한 인식기를 다룬다.
  - 어휘분석기를 설계하고 구현에 관한 사항들을 이해할 수 있다.

4.1 어휘분석의 개요

- **어휘 분석(lexical analysis)**이란 소스 프로그램을 읽어 토큰(token)이라는 문법적으로 의미있는 최소의 단위로 분리하는 것으로 토큰 스트림(stream)을 출력.
- 어휘 분석하는 도구를 어휘 분석기(lexical analyzer) 또는 스캐너(scanner)라 함

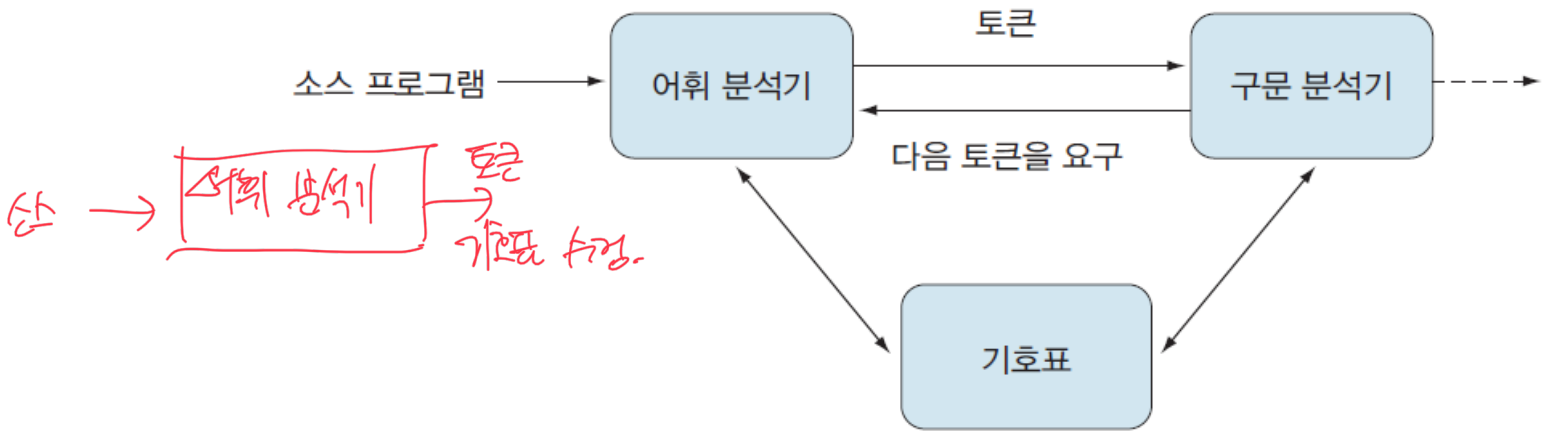


그림 4-1 어휘 분석기와 구문 분석기의 상호 작용

- **어휘 분석기 기능**
  - 토큰을 인식
  - 전처리 과정(preprocessing) : 코멘트, white space, tab, new line, macro 등을 처리
  - 기호표(symbol table) 구성
  - 에러 처리에 대한 진단

4.1 어휘분석의 개요

- **용어들**
  - 토큰(Token) : 문법에 있는 터미널 기호들로 구성된 **문법적으로 의미 있는 최소 단위**
  - 패턴(Pattern) : 토큰을 서술하는 규칙들
  - 렉심(Lexeme) : 패턴에 의해 매칭된 문자열

- **일반적인 프로그래밍 언어에서 사용하는 토큰들**
  - **Special form** - ***language designer***

    1. 지정어(**Keyword**) --- for, if, int 등의 언어에 이미 정의된 단어
    2. 연산자 기호(**Operator symbol**) --- +, -, *, /, <, =, ++ 등의 연산기호
    3. 구분자(**Delimiter**) --- ;, ,, (,), [,] 등 단어와 단어의 구분하거나 문장과 문장들을 구분
  - **General form** - ***programmer***

    4. 식별자(**identifier**) --- abc, b12, sum, 등 프로그래머가 정의하는 변수
    5. 상수(**constant**) --- 526, 3.0, 0.1234e-10, 'string' 등 실수형, 정수형, 문자형 상수

4.1 어휘분석의 개요

- **패턴 :**
  - 패턴들은 정규 문법에 의해 서술
    - C 언어의 식별자 : 첫 번째 기호가 영문자 소문자나 대문자 혹은 언더바(_)로 시작하고 두 번째 기호 부터는 영문자 소문자나 대문자, 숫자, 언더바가 오는 것
    - <ident> ::= (<letter> | _){<letter | <digit> | _ }
    - <letter> ::= a | b | c | ... | z | A | B | C | ... | Z
    - <digit> ::= 0 | 1 | 2 | ... | 9
    - 주의할 점은 <ident>에서 괄호와 중괄호의 의미 생각

4.1 어휘분석의 개요

- **[예제 4-1] 토큰 스트림 구하기**
  - [그림 2-4]의 문법을 이용하여 다음 문장의 토큰 스트림을 구해보자.
  - ni = ba * po - 60 + ni / (abc + 50);
    - **[풀이]** 토큰 스트림은 다음과 같이 16개로 되어 있으며, 순서대로 토큰을 분리한다.

❶ ⟨Sub C⟩ ::= ⟨assign-st⟩
❷ ⟨assign-st⟩ ::= ⟨lhs⟩ = ⟨exp⟩;
❸ ⟨lhs⟩ ::= ⟨variable⟩
❹ ⟨exp⟩ ::= ⟨exp⟩+⟨term⟩ | ⟨exp⟩-⟨term⟩ | ⟨term⟩
❺ ⟨term⟩ ::= ⟨term⟩*⟨factor⟩ | ⟨term⟩/⟨factor⟩ | ⟨factor⟩
❻ ⟨factor⟩ ::= ⟨variable⟩ | ⟨number⟩ | (⟨exp⟩)
❼ ⟨variable⟩ ::= ⟨ident⟩
❽ ⟨ident⟩ ::= (⟨letter⟩ | _){⟨letter⟩ | ⟨digit⟩ | _}
❾ ⟨number⟩ ::= {⟨digit⟩}
❿ ⟨letter⟩ ::= a | ··· | z
⓫ ⟨digit⟩ ::= 0 | 1 | ··· | 9

그림 2-4 아주 간단한 C 문법

Token	Lexime
식별자	ni
치환 연산자	=
식별자	ba
곱셈 연산자	*
식별자	po
뺄셈 연산자	-
상수	60
덧셈 연산자	+

Token	Lexime
식별자	ni
나눗셈 연산자	/
구분자	(
식별자	abc
덧셈 연산자	+
상수	50
구분자	)
구분자	;

4.1 어휘분석의 개요

- **[예제 4-2] 토큰, 렉심, 패턴 구하기**
  - [예제 4-1]에 주어진 문장에서 토큰, 렉심, 패턴 구하기
  - ni = ba * po - 60 + ni / (abc + 50);
    - **[풀이]**

토큰	렉심	패턴
식별자	ni, ba, po, abc	첫 자가 영문자나 언더바이고 둘째 자부터는 영문자나 숫자, 언더바
연산자	치환 연산자(=), 덧셈 연산자(+), 뺄셈 연산자(–), 곱셈 연산자(*), 나눗셈 연산자(/)	치환 연산자(=), 산술 연산자(+, –, *, /)
상수	60, 50	첫 번째와 두 번째 문자가 모두 숫자 0부터 9
구분자	(,), ;	문장 사이를 구분해주는 구분자

4.1 어휘분석의 개요

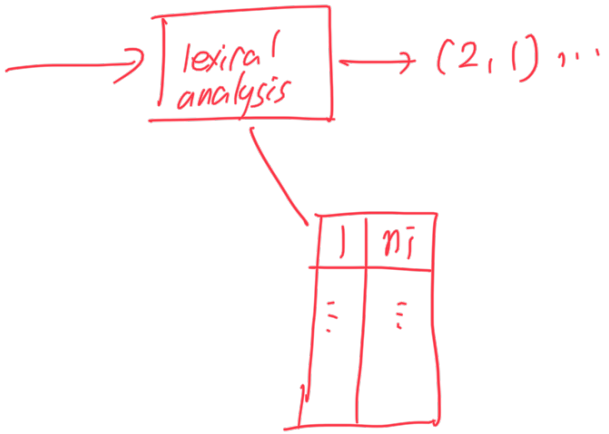
- **토큰의 표현 :**
  - 효율적인 구문분석을 할 수 있도록 토큰을 토큰번호(token number)와 토큰 값(token value)의 순서쌍으로 표현
  - **(토큰 번호, 토큰 값)**
    - 토큰번호 - 각각의 토큰들을 구분하기 위해서 각각의 토큰들에게 고유의 내부번호를 부여한 정수코드
    - 토큰 값 - 토큰들 중에 식별자나 상수는 하나의 이름으로 여러 번 사용될 수 있으므로 여러 번 사용될 때마다 다르게 표현하는 것이 아니라 프로그래머가 사용한 값으로 구별하기 위한 값
      - 식별자나 상수의 토큰 값은 기호표의 식별자나 상수가 위치하는 포인터 값

4.1 어휘분석의 개요

- **[예제 4-3] 토큰 스트림을 순서쌍으로 표현하기**
- [예제 4-1]의 토큰 스트림을 토큰 번호와 토큰 값으로 구성된 순서쌍으로 표현해보자.
- 먼저 각각의 토큰에 대한 토큰 번호가 필요하다. [그림 2-6]에 주어진 토큰 번호를 사용하고 식별자의 토큰 값은 식별자가 저장된 기호표의 위치라고 가정하자. 그러면 다음과 같이 나타낼 수 있다.
- (2, 1), (10, -), (2, 2), (8, -), (2, 3), (7, -), (4, 60), (6, -), (2, 1), (9, -), (13, -), (2, 4), (6, -), (4, 50), (14, -), (12, -)
- (단, 토큰 값 -는 토큰 값이 없음을 나타내고, 식별자의 토큰 값은 기호표에 저장된 위치의 값이며, 상수의 토큰 값은 그 자신의 상수 값이다.)

식별자의 (토큰 번호, 토큰 값)

식별자	ni	식별자	ni
치환 연산자	=	나눗셈 연산자	/
식별자	ba	구분자	(
곱셈 연산자	*	식별자	abc
식별자	po	덧셈 연산자	+
뺄셈 연산자	–	상수	50
상수	60	구분자	)
덧셈 연산자	+	구분자	;



4.2 토큰의 인식

- **토큰의 인식**
  - 토큰의 구조 : 정규 표현을 사용해서 표현
  - 인식기는 정규 표현으로부터 정규 표현을 받아들이는 유한 오토마타를 구성함으로써 달성
- **토큰 인식을 위한 영문자와 숫자**

  letter → a | b | c | … | z | A | B | C | … | Z

  digit → 0 | 1 | 2 | … | 9

  letter : l 로, digit : d로 표현

- **Identifier(식별자) 인식**
  - C 언어의 식별자는 첫 글자가 letter이거나 밑줄 문자로 시작하고, 두 번째 기호 부터는 letter, 밑줄 문자, digit가 오며, 길이에 상관없이 인식
  - 정규문법으로부터 정규표현으로 변환하고, 정규 표현을 받아들이는 상태전이도를 만듦

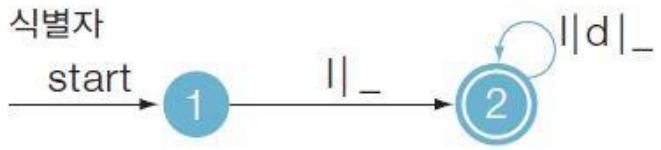


그림 4-2 식별자에 대한 상태 전이도

4.2 토큰의 인식

- [그림 4-2]에 대해 정규 문법을 거쳐 정규 표현으로 변환해보자

  S → lA | _A

  A → lA | dA | _A | ε

  3장에서 설명한 방법으로 정규 문법을 정규 표현으로 변환한다.

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

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

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

4.2 토큰의 인식

- **예약어 인식**
  - 예약어 if와 for에 대한 상태 전이도는 [그림 4-3]과 같다.

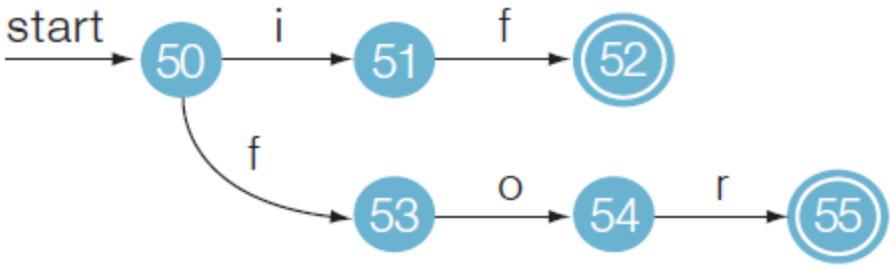


그림 4-3 예약어에 대한 상태 전이도

4.2 토큰의 인식

- **정수의 인식**

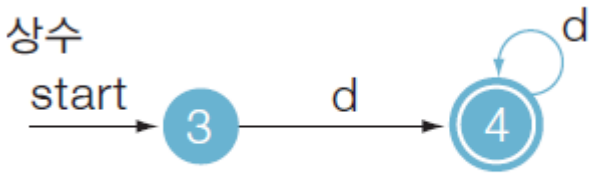


그림 4-4 10진법 양의 정수에 대한 상태 전이도

- [그림 4-4]에 대해 정규 문법을 거쳐 정규 표현으로 변환해보자. 정규 문법은 다음과 같다.

$S \rightarrow dC$

$C \rightarrow dC \mid \varepsilon$

정규 문법을 정규 표현으로 변환한다.

$C = dC + \varepsilon = d^*$

$S = dC = dd^* = d^+$

4.2 토큰의 인식

- **실수 상수의 인식**

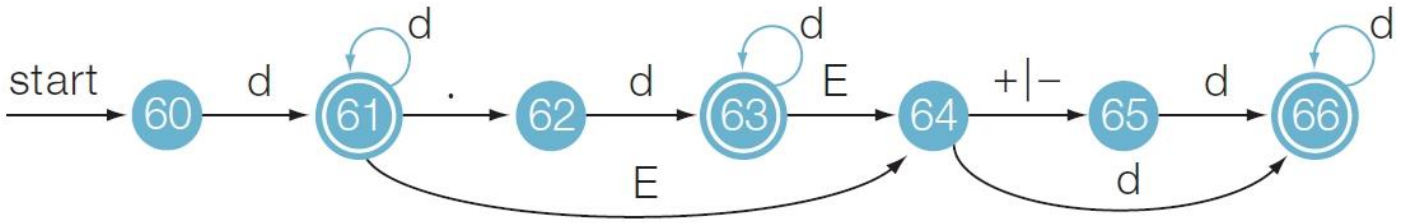


그림 4-5 부호 없는 실수에 대한 상태 전이도

- 정규 문법

S → dA
A → dA | .B
B → dC
C → dC | eD |ε
D → dE | +F | -G
E → dE |ε
F → De
G → dE

4.2 토큰의 인식

- 정규 표현

$E = dE + \varepsilon = d^*$ $F = dE = dd^* = d^+$ $G = dE = dd^* = d^+$

$D = dE + '+'F + -G = dd^* + '+'d^+ + -d^+$

$= d^+ + '+'d^+ + -d^+ = (\varepsilon + '+' + -)d^+$

$C = dC + eD + \varepsilon = dC + e(\varepsilon + '+' + -)d^+ + e$

$= d^*(e(\varepsilon + '+' + -)\, d^+ + \varepsilon)$

$B = dC = dd^*(e(\varepsilon + '+' + -)d^+ + \varepsilon)$

$= d^+ + (e(\varepsilon + '+' + -)\, d^+ + \varepsilon)$

$A = dA + .B$

$= d^*.d + (e(\varepsilon + '+' + -)d^+ + \varepsilon)$

$S = dA$

$= dd^*.\, d^+(e(\varepsilon + '+' + -)\, d^+ + \varepsilon)$

$= d^+.d^+(e(\varepsilon + '+' + -)\, d^+ + \varepsilon)$

$= d^+.d^+ + d^+.d^+e(\varepsilon + '+' + -)\, d^+$

참고 터미널 기호 +를 '+'로 표기.

4.2 토큰의 인식

- 주석의 처리

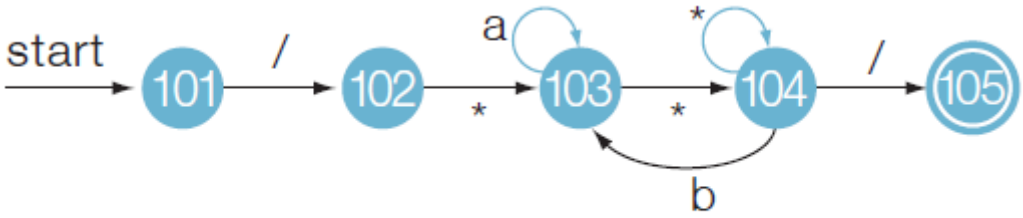


그림 4-6 주석에 대한 상태 전이도

- 이때 a는 * 이외의 모든 문자이고 b는 *와 /을 제외한 모든 문자를 나타낸다. 이에 대해 정규 문법을 거쳐 정규 표현으로 변환해보자. 정규 문법은 다음과 같다.

S → /A

A → *B

B → aB | *C

C → *C | bB | /D

D → ε

4.2 토큰의 인식

- 정규 문법을 정규 표현으로 변환한다.

C = *C + bB + /D
  = *C + bB + /
  = **(bB + /)
B = aB + *C
  = aB + ***(bB + /)
  = aB + *+bB + *+/
  = (a + *+b)B + *+/
  = (a + *+b)*(*+/)
A = *B
  = *(a + *+b)*(*+/)
S = /A
  = /*(a + *+b)*(*+/)

4.3 어휘분석기의 설계 및 구현

- **어휘분석기의 구현 방법 : 크게 두 가지**
  - **1. 이론적인 방법들을 프로그래밍을 통하여 구현** C/C++
  - **2. 렉스(플렉스)를 통해서 구문 분석기를 생성하는 방법**
  - **3. 프로그래밍 기법**
    - 문법으로 부터 토큰 추출하고 토큰표 작성어휘 분석기를 설계하려면 문법이 있어야 함. [그림 2-4]의 문법을 예로 들면, 먼저 [그림 2-4]의 문법에서 사용하는 토큰과 패턴을 찾아야 한다. 토큰과 패턴을 찾아서 토큰표를 만들고 그것을 정규 문법으로 표현한 다음, 이 정규 문법을 정규 표현으로 나타낸다. 그리고 다시 그 정규 표현을 받아들이는 NFA를 구현하고 NFA를 DFA로 변환한 뒤, 마지막 으로 DFA를 상태수가 최소화된 DFA로 변환한 것이 바로 어휘 분석기이다. 이 과정을 [그림 4-7]에 나타냈다.

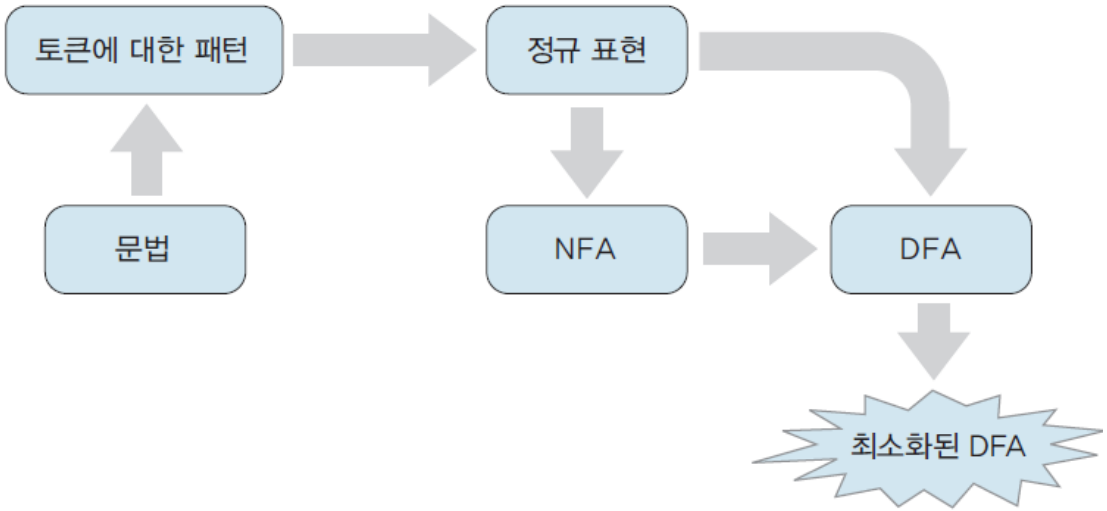


그림 4–7 어휘 분석기의 설계

4.3 어휘분석기의 설계 및 구현

- 문법으로부터 어떠한 토큰들이 사용되는지 확인한다

(1) <Sub C> ::= <assign-st>

(2) <assign-st> ::= <lhs> = <exp>;

(3) <lhs> ::= <variable>

(4) <exp> ::= <exp>+<term> | <exp>-<term> | <term>

(5) <term> ::= <term>*<factor> | <term>/<factor> | <factor>

(6) <factor> ::= <variable> | <number> | (<exp>)

(7) <variable> ::= <ident>

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

(9) <number> ::= {<digit>}

(10) <letter> ::= a |…| z

(11) <digit> ::= 0 | 1 | …| 9

- [그림 2-4]로부터 토큰을 찾기 위해 일반적인 언어에서 사용하는 다섯 가지 토큰을 가지고 검사한다.

4.3 어휘분석기의 설계 및 구현

- 토큰 표 작성

표 4-1 [그림 2-4]에 대한 토큰표

구분	토큰 이름	토큰 번호	토큰 값
식별자	식별자	2	기호표의 포인터
상수	상수	4	상수 값
연산자	+	6	
	-	7	
	*	8	
	/	9	
	=	10	
구분자	;	12	
	(	13	
	)	14	

4.3 어휘분석기의 설계 및 구현

- 각 토큰들을 받아들이는 유한 오토마타 작성
- (1) 식별자에 대한 유한 오토마타
  - 정규 표현

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

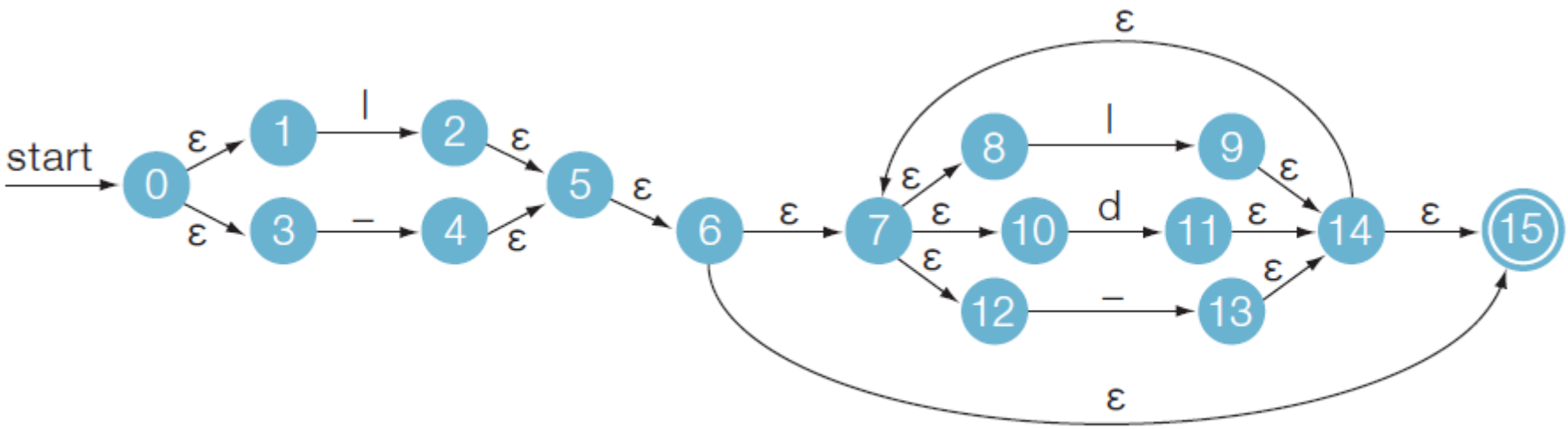


그림 4-8 식별자를 인식하는 NFA

4.3 어휘분석기의 설계 및 구현

- [그림 4-8]을 DFA로 변환

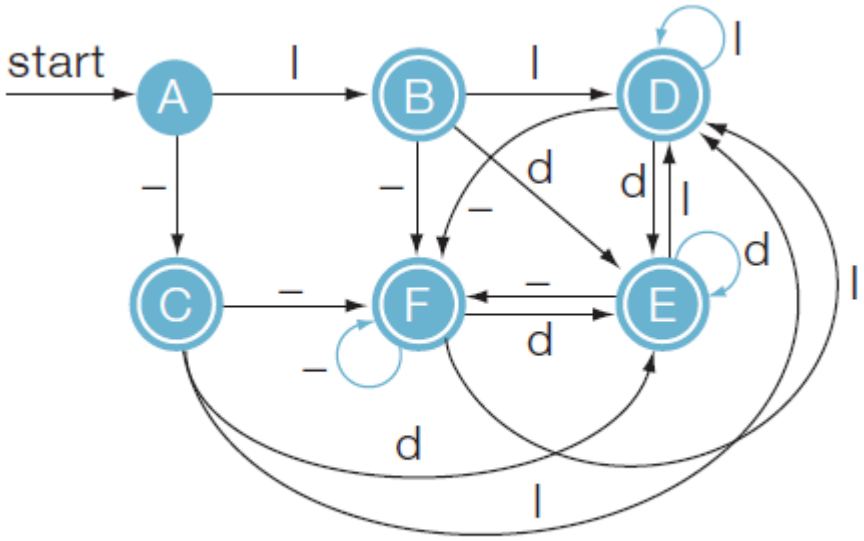


그림 4-9 식별자를 인식하는 DFA

- 상태수를 최소화하면 [그림 4-2]와 같다.

4.3 어휘분석기의 설계 및 구현

- (2) 상수에 대한 어휘분석기

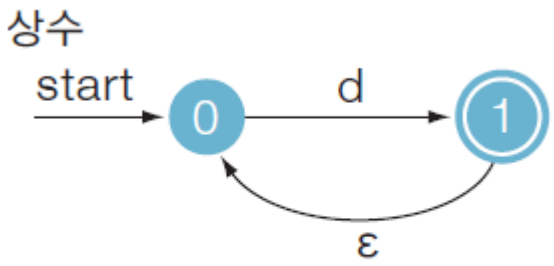


그림 4–10 양의 정수를 인식하는 NFA

- 상태수 최소화하면 [그림 4-4]와 같다.

4.3 어휘분석기의 설계 및 구현

- 같은 방법으로 연산자의 상태 전이도는 [그림 4-11]과 같다.

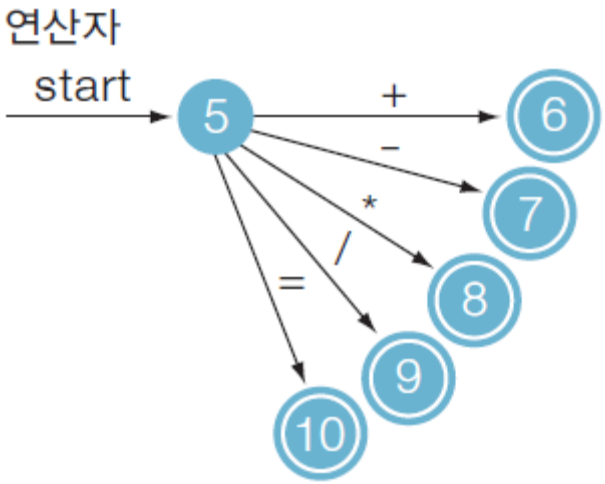


그림 4-11 연산자에 대한 상태 전이도

4.3 어휘분석기의 설계 및 구현

- 같은 방법으로 구분자의 상태전이도는 [그림 4-12]와 같다.

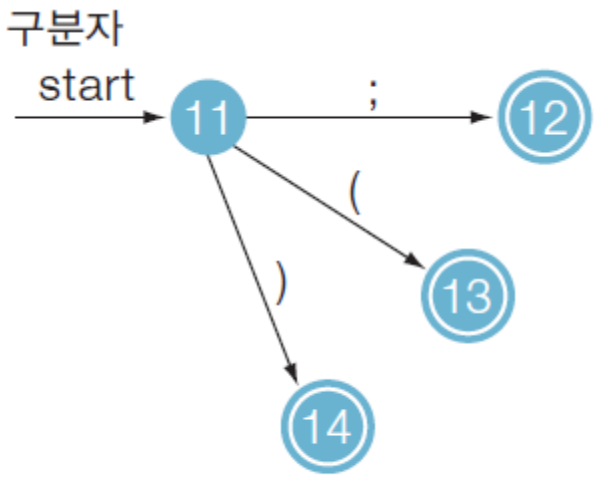


그림 4-12 구분자에 대한 상태 전이도

4.3 어휘분석기의 설계 및 구현

- [그림 2-4]에 대한 상태 전이도

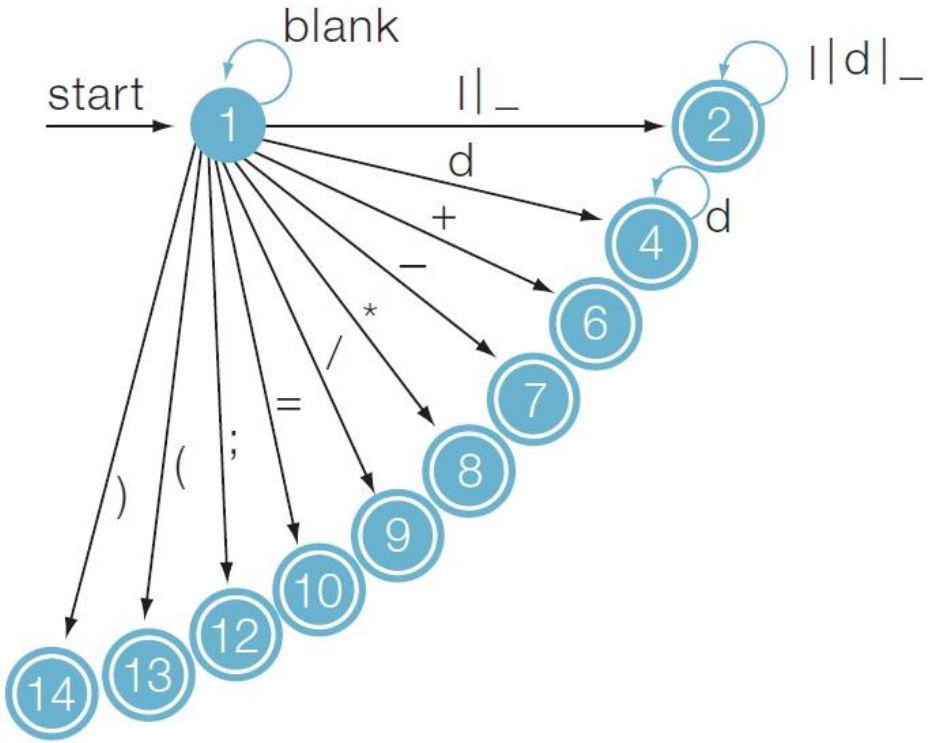


그림 4-13 [그림 2-4]에 대한 상태 전이도