

Compiler (컴파일러)

Lexical Analysis (어휘 분석)

2016 2학기
충남대학교 컴퓨터공학과
조은선

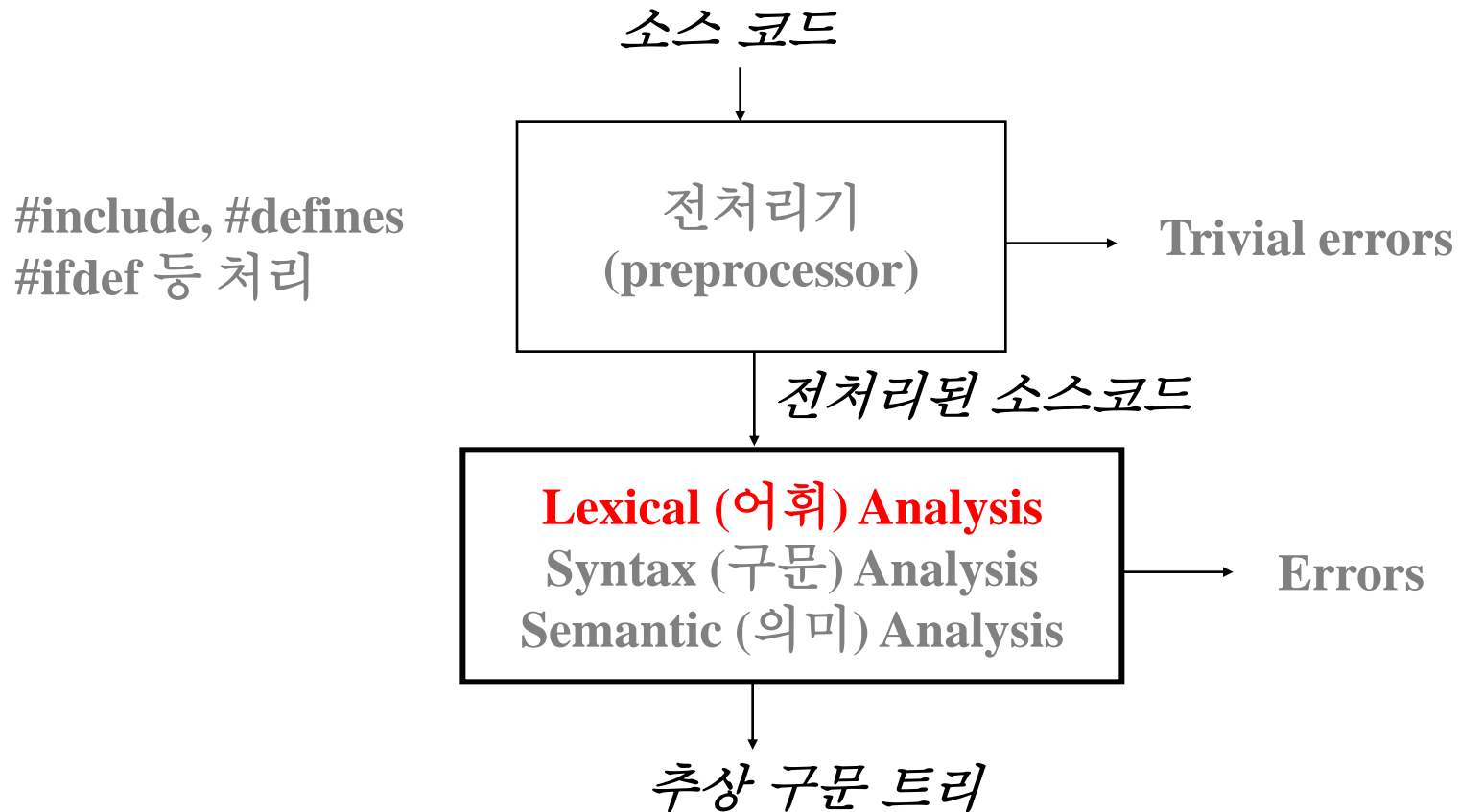
지난 시간에 한 것

- 교과목 설명

이번 시간에 할 것

- Lexical analysis

컴파일러 전반부 Structure



Lexical Analysis (어휘분석)

`if (b == 0) a = b;`

read char by char

어휘분석기
(=스캐너)

<code>if</code>	<code>(</code>	<code>b</code>	<code>==</code>	<code>0</code>	<code>)</code>	<code>a</code>	<code>=</code>	<code>b</code>	<code>;</code>
-----------------	----------------	----------------	-----------------	----------------	----------------	----------------	----------------	----------------	----------------

- 원시프로그램을 긴 문자열로 보고 차례대로 문자를 검사하여, 의미있는 최소단위들로 변환하는 것
- Space 같은 것들을 제거해서 코드의 크기도 줄임

- 토큰(Token)

“문법적으로 의미 있는 최소단위”

- 식별자 : **x y12 elsex**
- 키워드 : **if else while for break**
- 상수 : **2 1000 -20 2.0 -0.0010 .02**
- 연산자 : **+ * { ++ << < <=]**
- 문자열 : **"x"**
"He said, \" I love CSE.\""

참고) StringTokenizer in Java

```
import java.util.StringTokenizer;
public class JavaTestMain {
    public static void main(String[] args) {
        String str = "this is my string";
        StringTokenizer st
            = new StringTokenizer(str);
        System.out.println(st.countTokens());
        while(st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
        System.out.println(st.countTokens());
    }
}
```

Token 과 관련된 질문들

- 1) How to **describe tokens**?
 - 프로그래밍 언어 제작자가 토큰 하나하나를 **기술**(설명)하는 방법
- 2) How to **recognize tokens**?
 - 토큰을 **인식**해 내는 방법
- 3) How to **break up tokens**?
 - 여러 토큰이 인식되었을 때 **결정**하는 방법
- 4) How to **represent tokens**?
 - 토큰이 컴퓨터에서 **표현**되는 방법

(1) How to Describe Tokens

- 정규 표현식을 사용해서 토큰을 설명한다.
- 정규 표현식 (regular expression) 이란..
 - a 일반 문자
 - ϵ 빈 (empty) 스트링
 - $R|S$ R 이거나 S 일 때 (단, R 과 S 가 정규표현식일 때)
 - RS R 다음에 S 가 나올 때 (단, R 과 S 가 정규표현식일 때)
 - R^* R 이 0번 이상 나올 때 (단, R 이 정규표현식일 때)

사람보다는
기계한테 좋다

예

정규표현식 R	인식되는 문자열들 $L(R)$
abc	"abc"
abc^*	"ab" "abc" "abcc" "abccc" ...
$(abc)^*$	"" "abc" "abcabc" "abcabcabc" ...
$(a \epsilon)b$	"ab" "b"
$1(0 1)^*$	2진수 집합 ?

정규표현식 (Regular Expressions)



정규표현식을 위한 짧은 표현들

- **R^+** = $R(R^*)$ R이 한번이상 나타남
- **$R^?$** = $(R|\epsilon)$ R이 한번 있거나 아무것도 없음
- **$[abc]$** = $(a|b|c)$ 괄호속의 문자들 중 하나
- **$[a-z]$** = $(a|b|c|\dots|z)$ 이 범위의 문자 중 하나
- **ab** 괄호속 문자들만 제외
- **^a-z** 이 범위 내 문자들만 제외

예

정규표현식	인식되는 문자열들
abc^*	"ab" "abc" "abcc" "abccc" ...
abc^+	"abc" "abcc" "abccc" ...
$a(bc)^+$	"abc" "abcbc" "abcbcbc" ...
$a(bc)^?$	"a" "abc"
$[a-zA-Z]$	알파벳 대소문자 집합
$[0-9]$	"0", "1" "2" "3" "4" "5" .. "9"

참고: 기타 정규 표현식을 쓰는 곳 1

- Unix 명령 중 grep

grep smug files	{search files for lines with 'smug'}
grep '^smug' files	{'smug' at the start of a line}
grep 'smug\$' files	{'smug' at the end of a line}
grep '^smug\$' files	{lines containing only 'smug'}
grep '\^s' files	{lines starting with '^s', "\" escapes the ^}
grep '[Ss]mug' files	{search for 'Smug' or 'smug'}
grep 'B[oO][bB]' files	{search for BOB, Bob, BOB or BoB }
grep '^\$' files	{search for blank lines}
grep '[0-9][0-9]' file	{search for pairs of numeric digits}

<http://www.robelle.com/smugbook/regexpr.html>

참고: 기타 정규 표현식을 쓰는 곳 2

- JavaScript에서
 - exec, test, match, search 등의 메소드들이 사용함

```
1 var myRe = /d(b+)d/g;  
2 var myArray = myRe.exec("cdbbdbsbz");
```

/g는, 하나 찾고 멈추지 말고,
match 되는 것은 전부 찾으란 뜻

- 정규표현식을 사용하는 메소드들(일부)
(http://www.w3schools.com/jsref/jsref_regexp_g.asp)

exec

A RegExp method that executes a search for a match in a string. It returns an

test

A RegExp method that tests for a match in a string. It returns true or false.

참고: 기타 정규 표현식을 쓰는 곳 3

- Java에서
Package `java.util.regex`

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

Package `java.util.regex`

Classes for matching character sequences against patterns specified by regular expressions.

See: [Description](#)

Interface Summary

Interface	Description
<code>MatchResult</code>	The result of a match operation.

Class Summary

Class	Description
<code>Matcher</code>	An engine that performs match operations on a character sequence by interpreting the sequence against a regular expression.
<code>Pattern</code>	A compiled representation of a regular expression.

Exception Summary

Exception	Description
<code>PatternSyntaxException</code>	Unchecked exception thrown to indicate a syntax error in a regular-expression pattern.

Java Example

```
import java.io.Console;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class RegexTestHarness {
    public static void main(String[] args){
        Console console = System.console();
        if (console == null) ... // error.. exit!
        while (true) {
            Pattern pattern =
                Pattern.compile(console.readLine("%nEnter your regex: "));
            Matcher matcher =
                pattern.matcher(console.readLine("Enter input string to search:"));
            boolean found = false;
            while (matcher.find()) {
                console.format("I found the text" + " \"%s\" starting at "
                               + "index %d and ending at index %d.%n",
                               matcher.group(), matcher.start(), matcher.end());
                found = true;
            }
            if(!found){ console.format("No match found.%n"); }
        }
    }
}
```

Java

```
import
import
import
public
```

```
publ
```

```
Co
```

```
i
```

```
wl
```

```
Enter your regex: foo
Enter input string to search: foofoofoo
I found the text foo starting at index 0 and ending at index 3.
I found the text foo starting at index 3 and ending at index 6.
I found the text foo starting at index 6 and ending at index 9.

Enter your regex: a+
Enter input string to search: ababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "aaaa" starting at index 4 and ending at index 8.
```

```
Pattern pattern =
```

```
    Pattern.compile(console.readLine("%nEnter your regex: "));
```

```
Matcher matcher =
```

```
    pattern.matcher(console.readLine("Enter input string to search:"));
```

```
boolean found = false;
```

```
while (matcher.find()) {
```

```
    console.format("I found the text" + " \"%s\" starting at "
```

```
        + "index %d and ending at index %d.%n",
```

```
        matcher.group(), matcher.start(), matcher.end());
```

```
    found = true;
```

```
}
```

```
if(!found){ console.format("No match found.%n"); }
```

```
}
}
}
```


Class Problem

다음 을 나타내는 정규표현식을 작성해보시오. 단, 정규표현식 작성이 불가능 한 것이 있다면 추측하는 이유를 적어보시오.

1. a로 시작하는 식별자
2. 2진수중 4의 배수
3. 식별자중 a와 b의 갯수가 동일하게 나타나는 것

(2) How to Recognize Tokens

- FSA, Finite State Automata,

유한 상태 오토마타

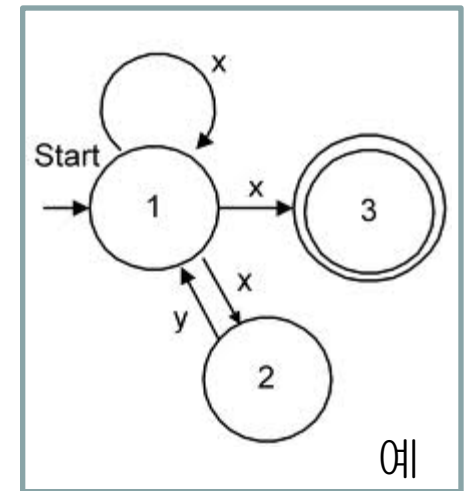
– 정의

- 상태를 유한한 갯수까지만 가짐
- 상태 간의 전이를 정의함
- 시작 상태 한 개와 끝 상태 여러 개를 가짐

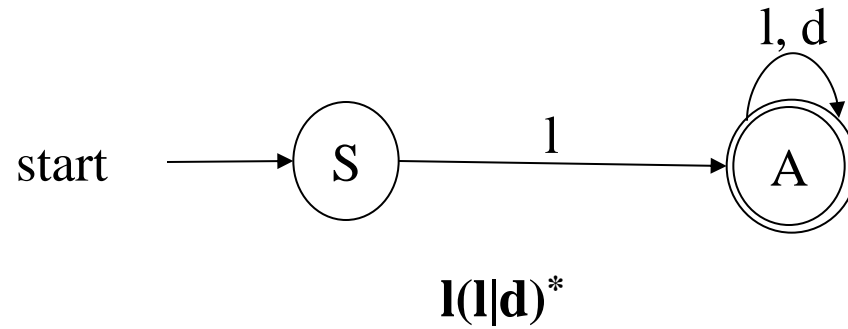
– 어휘 분석의 기초가 되는 이론적 기반임

- 이유: 정규 표현식과 동일한 표현력 (expressive power) 을 가짐

– 상태 전이도로 나타냄



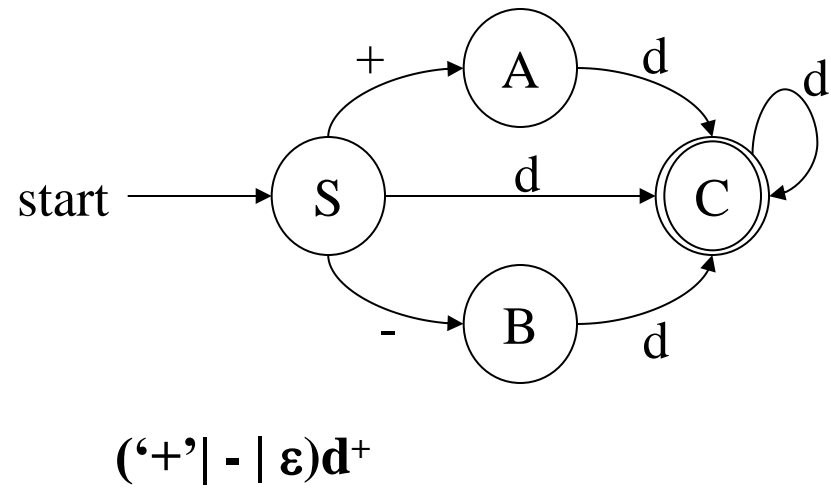
★ 식별자의 인식



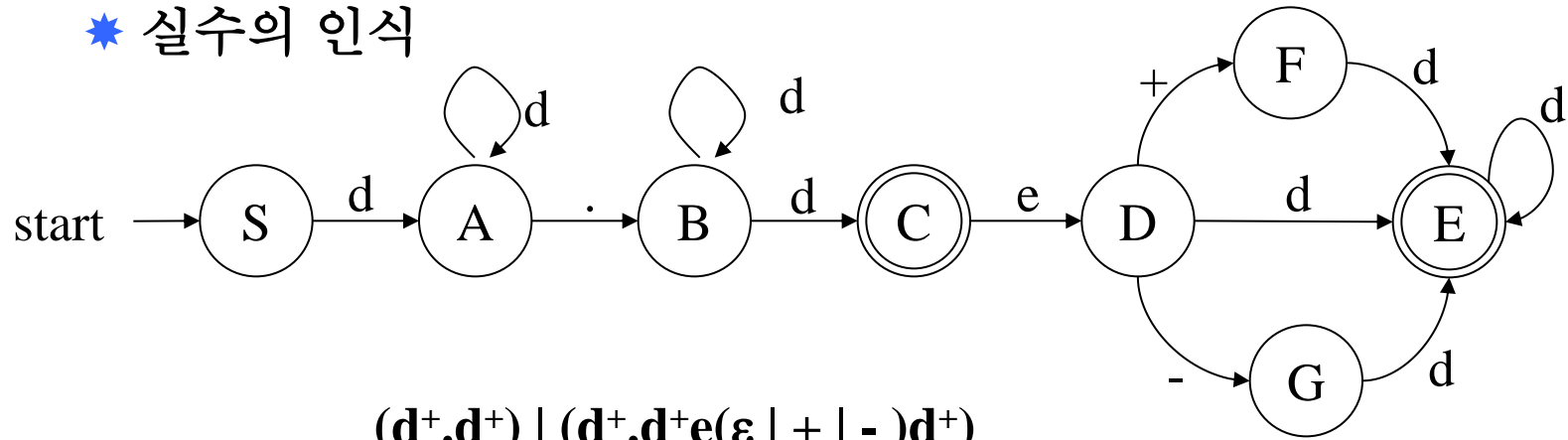
$l = [a-zA-Z]$

$d = [0-9]$ 일 때

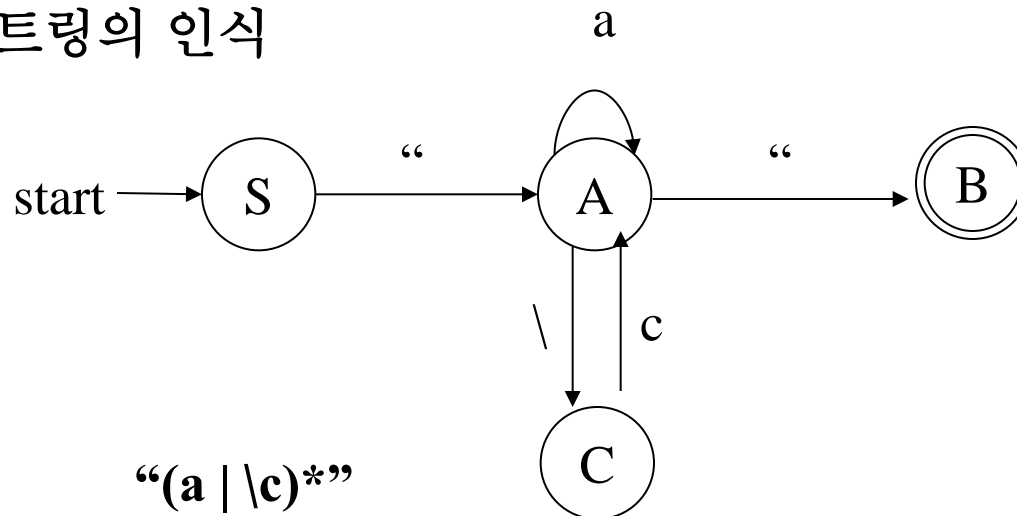
★ 정수의 인식



★ 실수의 인식



★ 스트링의 인식

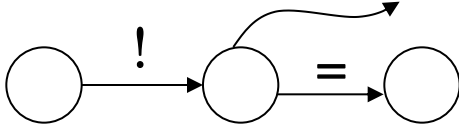


단 a 는 “와 \ 이외의 문자
c는 이스케이프문자이나, 여기서는 어떤 문자도 될 수 있다.

Token 인식을 위한 처리

- 토큰 인식을 위한 처리 절차
 - 사용자가 정의한 정규표현식을 FSA 로 변환
 - FSA대로 인식

예) 토큰 '!= '일 때,

정규표현식	!=
FSA	
인식하는 코드	<pre>while (isspace(ch = getchar())); switch (ch) { case '!' :// state 17 ch = getchar(); if (ch == '=') token.number = tnotequ; else { token.number = tnot; ungetc(ch, stdin); } }</pre>

Class Problem

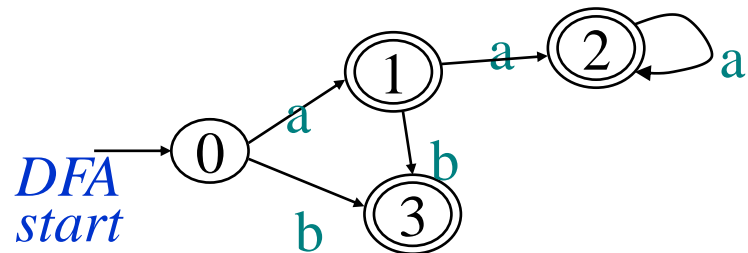
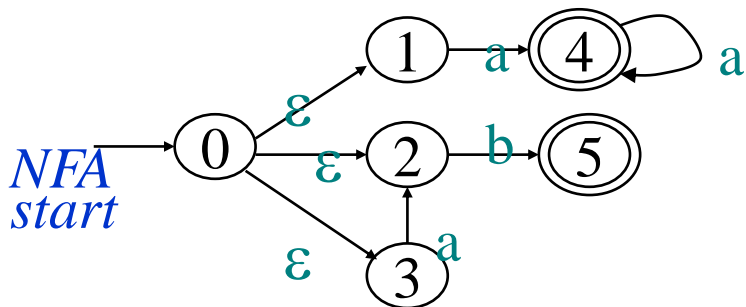
- 다음 정규 표현식을 인식하는 유한상태오토마타 (Finite State Automata)를 상태전이도로 표현 하시오.

$$(a ((b \mid c \mid a^+ c) x)^*) \mid (x^* a)$$

DFA (Deterministic finite automata)

- ‘DFA’
 - FSA의 한 종류/ 간결해서 인식기 구현이 쉽다.
 - 각 상태에서 ‘나가는 edge’ 가 레이블 문자에 의해 유일하게 결정
 - 각 상태마다, 문자 ‘a’가 붙은 edge 는 한 개만 나갈 수 있다.
 - ϵ 가 붙은 edge 도 없다.
- 그래서, 토큰 인식은
 - 정규표현식을 NFA (non-DFA) 로 변환시키고
 - 다시 이 NFA를 DFA로 변환 시키고
 - 이 DFA를 따라 인식한다

참고 예: **aa* | b | ab**



(3) How to Break up Text

<code>else</code>	<code>x</code>	<code>=</code>	<code>0</code>	<code>;</code>
<code>else</code>	<code>=</code>	<code>0</code>	<code>;</code>	

- 두 개 이상의 정규표현식이 매치될 때
 - 긴 token 우선?
 - 우선순위?

(4) How to Represent Tokens

“어휘 분석 결과 내에서 토큰들은 어떻게 표현될까?”

- 토큰번호
 - 각 토큰들을 효율적인 처리하기 위해서 고유의 내부번호
- 토큰값
 - 토큰이 프로그래머가 사용한 값을 가질 때 그 값을 말한다
 - 명칭의 토큰값은 그 자신의 스트링 값
 - 상수의 토큰값은 그 자신의 상수 값

예)

if X < Y then X :=10;

(29,0) (1,X) (18,0) (1,Y) (35,0) (1,X) (9,0) (2,10) (7,0)

lexeme

(1,10) : X
(1,20) : Y > Symbol table

Mini C 의 어휘분석기

- 미니 C에 대한 어휘분석기 구현

- 특수 심벌

=	;	,	!	&&	
+	++	+=	-	--	--=
*	*=	%	%=	/	/=
<	<=	==	>	>=	!=
[]	{	}	()

- 단어 심벌

const else if int return void while

- 여러 가지 가지 방법

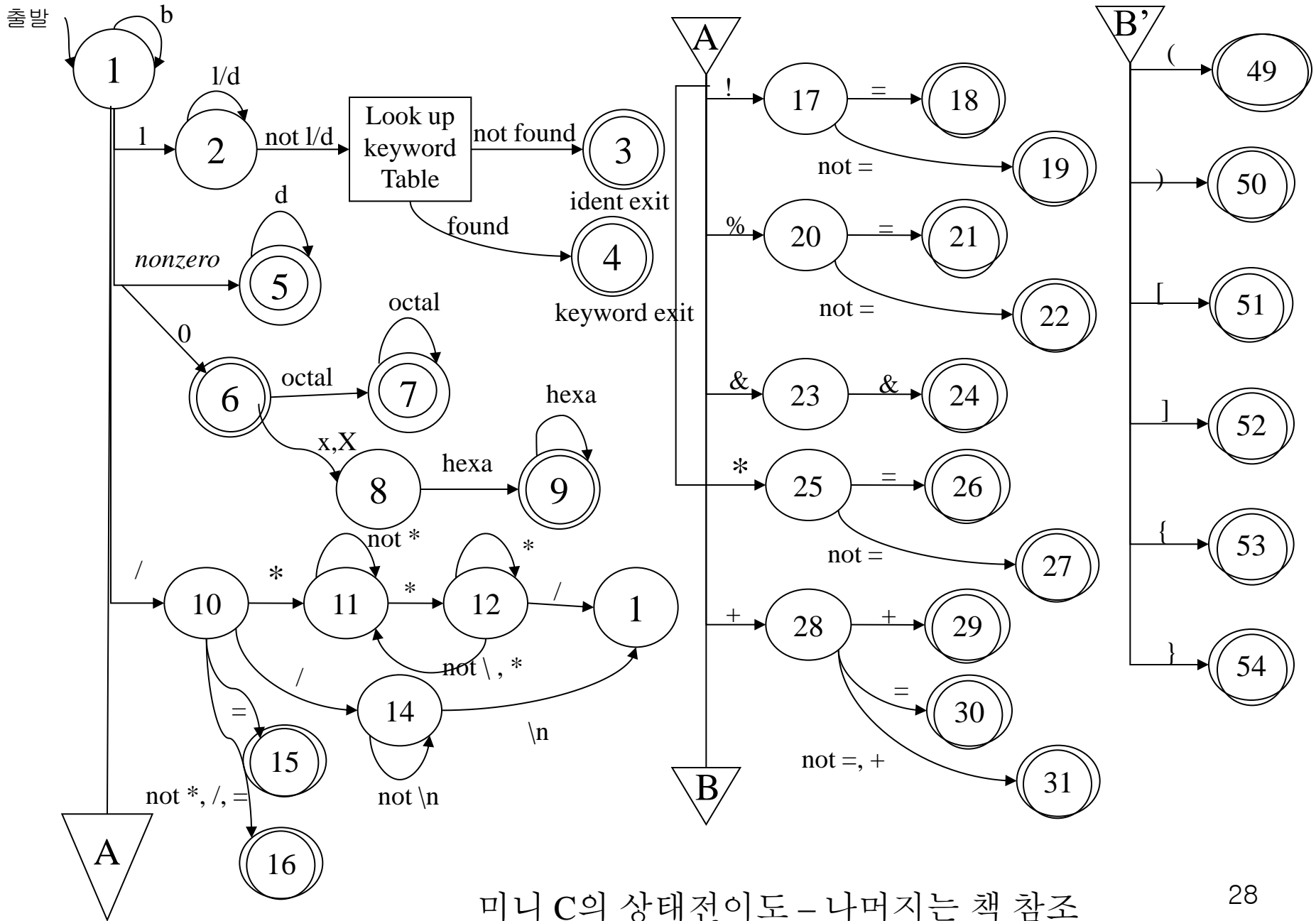
- (1) 처음부터 구현

- (2) 자동화 도구를 사용

[1] 처음부터 구현

- 앞의 심벌들을 정규표현식으로 정의
- 정의된 정규표현식을 FSA로 만들기
- 이 FSA 를 따라 인식하는 프로그램을 작성한다.

여기서는 토큰들이 간단하므로 정규표현식은 건너뛰고 FSA부터 시작한다.



인식하는 프로그램 작성

- 토큰의 자료구조 : lexeme을 표현

```
#define ID_LENGTH 12
struct tokenType {
    int number;
    union {
        char id[ID_LENGTH];
        int num;
    } value;
};
```

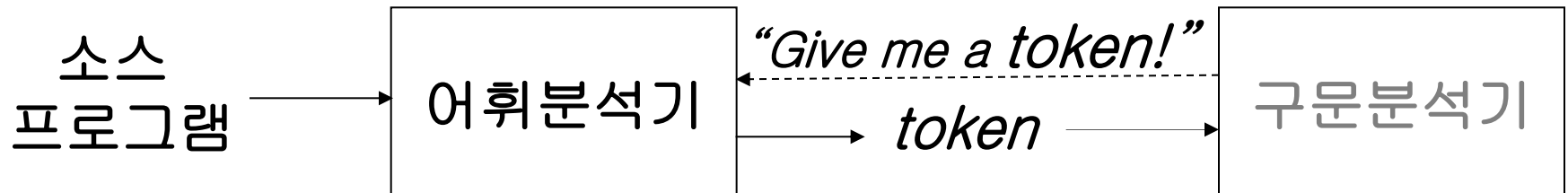
- 몇가지 유용한 함수들

- isspace(c), isdigit(c)
- superLetter(c), superLetterOrDigit(c)
- getIntNum(c), lexicalError(i)...

- 토큰의 번호의 상수화 및 키워드 갯수

```
enum tsymbol { tnull = -1,
    tnot, tnotequ, tmod, tmodAssign, tindent, ...}
#define NO_KEYWORDS 7
```

스캐너 [어휘분석기]와 파서의 관계



여기서는, 구문분석기에서
어휘분석기의 함수 `scanner()` 를 호출할 때 마다
다음 토큰 (lexeme) 이 리턴되는 구조를 취함

```
#include <stdio.h>
```

```
...
```

```
struct tokenType scanner()
```

```
{
```

```
    struct tokenType token;
```

```
    int i, index;
```

```
    char ch, id[ID_LENGTH];
```

```
    token.number = tnull;
```

```
    do {
```

```
        while (isspace(ch = getchar()));
```

```
        if (superLetter(ch)) // id or keyword
```

```
            i = 0;
```

```
            do {
```

```
                if (i < ID_LENGTH) id[i++] = ch;
```

```
                ch = getchar();
```

```
            } while (superLetterOrDigit(ch));
```

```
            if (i >= ID_LENGTH) lexicalError(1);
```

```
            id[i] = '\0';
```

```
            ungetc(ch, stdin);
```

**직접 작성한 스캐너
소스 in C (교재)**

```

    for (index=0; index < NO_KEYWORDS; index++)
        if (!strcmp(id, keyword[index])) break;
    if (index < NO_KEYWORDS)
        token.number = tnum[index];
    else {
        token.number = tindex;
        strcpy(token.value.id, id);
    }
} // end of id or keyword
else if (isdigit(ch)) { // integer..
    token.number = tnumber;
    token.value.num = getIntNum(ch);
}
else switch (ch) { // special character
    case '!' :// state 17
        ch = getchar();
        if (ch == '=') token.number = tnotequ;
        else {
            token.number = tnot;
            ungetc(ch, stdin);
        }
        break; ...

```



```
        case `}` : token.number = trbrace; break;
        case EOF : token.number = teof; break;
        default : lexicalError(4);
    } end of switch
} while (token.number == tnull);
return token;
} // end of scanner
```

Java로도 가능:

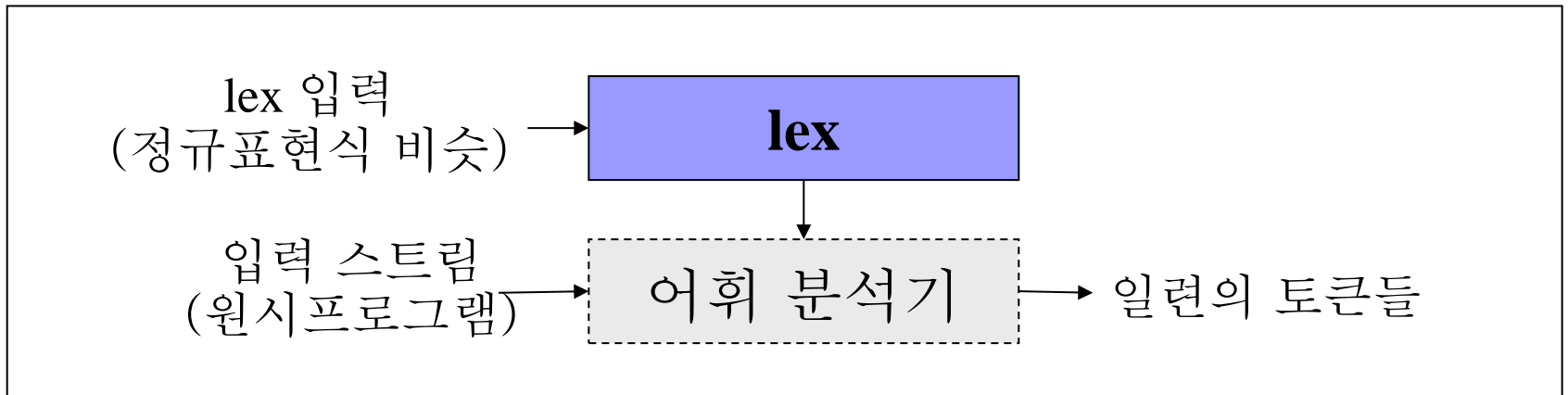
<https://inst.eecs.berkeley.edu/~cs164/sp11/lectures/lecture2/Lexer.java>

[2] 도구 사용

- Lex
 - 전통적인 방법, C 기반
 - 구문분석기 yacc과 밀접한 관련
 - 발전된 버전인 flex를 더 많이 씀
- ANTLR
 - Java 기반
 - 구문 분석할 때 다시 볼 것임
- 기타:
 - JavaCC, SableCC ...

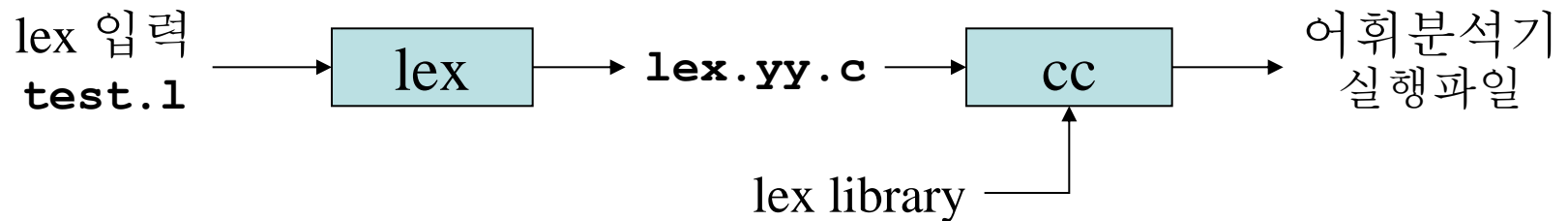
렉스(Lex)

- 1975년에 레스크에 의해 발표된 어휘분석기 생성기
- 사용자가 정의한 정규 표현과 실행코드를 입력 받아 C언어로 쓰여진 프로그램 출력



- 어휘분석기의 생성 및 동작

- 어휘분석기의 생성과정



- lex의 선택규칙

- 가장 길게 인식된 토큰을 우선으로 한다.
 - 인식될 수 있는 토큰의 길이가 같은 경우 먼저 나타난 규칙의 정규 표현으로 인식한다.

- 렉스의 입력

<정의부분>

%{

실행코드를 C언어로 기술할 때 필요한 자료구조, 변수 상수

}%

이름 정의 부분

: 특정한 정규표현을 하나의 이름으로 정의하여 그 형태의 정규표현이 필요 할 때만 쓸 수 있도록 해주는 부분

%%

<규칙부분>

:토큰의 형태를 표현하는 정규표현과 그 토큰이 인식되었을 때 처리할 행위를 기술하기 위한 부분인 실행코드

%%

<사용자 부 프로그램 부분>

: 렉스의 입력 작성시 사용되는 부 프로그램들을 정의

```

%{ /* file name : word.lex
   * 이 프로그램은 파일이름을 입력받고 파일에 들어있는 문장의 라인수 단어수
   * 문자수를 계산하고 파일이름과 같이 출력한다. */
   unsigned long charCount=0, wordCount=0, lineCount=0;
%}
word [^ \t\n]+
eol   \n

%%

{word} { wordCount++; charCount+=yyleng; }
{eol}   {charCount++; lineCount++; }
.       {charCount++;}

%%

void main() {
    FILE *file;
    char fn[20];
    printf("Type a input file Name:");
    scanf("%s",fn);
    file=fopen(fn, "r");
    if (!file) {
        fprintf(stderr,"file '%s' could not be opened. \n",fn);
        exit(1);
    }
    yyin=file;
    yylex();
    printf("%d %d %d %s \n", lineCount, wordCount, charCount,fn);
}

yywrap() { return 1; /*처리의 종료 */ }

```

- 렉스의 정규표현 (일반 정규표현식 + α)
 - * : 0번 이상 반복을 의미
예) $[a-zA-Z][a-zA-Z0-9]^*$: 변수 인식을 위한 정규표현
 - + : 한번 이상 반복 할 수 있음을 의미
예) $[a-z]^+$: 모든 소문자 문자열을 인식하는 정규표현
 - ? : 선택을 의미
예) $ab?c$: abc 또는 ac
 - | : 택일을 위한 연산자
예) $(ab | cd)$: ab 또는 cd
 $(ab | cd+)?(ef)^*$: $abefef$, $efefef$, $cdef$, $cddd$
 - [] : 문자들의 클래스를 정의하는데 사용
예) $[abc]$: a, b, c 중에서 한문자
 - - : 범위를 나타내는 연산자 문자
예) $[a-z]$: a 부터 z 사이의 문자 중에 한문자
 - ^ : 여집합을 표현
예) $[^*]$: $*$ 를 제외한 모든 문자

- 렉스의 정규표현 (일반 정규표현식 $+$ α)
 - “ : “ 사이에 있는 모든 문자는 텍스트문자로 취급
예) $a“*”b$ 와 $a*b$ 는 다르다
 - \ : [] 밖에서는 한 개의 문자를 이스케이프 하기 위하여 사용
예) $XYZ“++”, “XYZ++”, XYZ\+\+$
 - [] 속에서는 C언어의 이스케이프 문자열로 간주
예) $[\backslash t n]$: 공백, 탭, 개행 문자중의 하나
 - ^ : 라인의 시작을 인식
예) abc : 라인의 시작에서 abc 가 있을 때만 토큰으로 처리
 - \$: 라인의 끝에서만 인식
 - / : 접미 문맥을 명시할 때 사용
예) ab/cd : ab 다음에 cd 가 이어서 있을 때만 ab 를 토큰으로 처리
 - . : newline문자를 제외한 모든 문자들
예) $“- -”.*$: - -부터 한 라인의 끝까지와 부합
 - {} : 정의된 이름을 치환식으로 확장 할 때 사용

Lex Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc, abcc, abccc, abccccc, ...
a(bc)+	abc, abcbcb, abcbcbcb, ...
a(bc)?	a, abc
[abc]	one of: a, b, c
[a-z]	any letter, a through z
[a\ -z]	one of: a, -, z
[-az]	one of: - a z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	a, ^, b
[a b]	a, , b
a b	a, b

Class Problem

- 다음 lex 정규표현식들의 차이는?
 1. `[abc]` 와 `abc`
 2. `[^abc]` 와 `[abc^]` 와 `^abc`
 3. `[a-z]` 와 `[-az]`와 `[az-]`
 4. `[a|b]` 와 `(a|b)`

- 렉스 실행코드 내의 총괄변수와 함수

- `yylen` : 매칭된 토큰의 문자열의 길이를 저장하고 있는 변수
- `yytext` : 매칭된 토큰의 문자열을 담은 변수
- `yyval` : 매칭된 토큰값을 담은 변수
- `yyomore()` : 현재 매칭된 문자열의 끝에 다음에 인식될 문자열이 덧붙여 지도록 하는 함수
- `yyless(n)` : `n`개의 문자만을 `yytext`에 남겨두고 나머지는 다시 처리하기 위하여 입력 스트림으로 되돌려 보내는 함수
- `yywrap()` : 렉스가 입력의 끝을 만났을 때 호출하는 함수
정상적인 경우에 복귀값은 1이다.



<http://myweb.stedwards.edu/laurab/cosc4342/lex-examples.html>
<http://www.ibm.com/developerworks/kr/linux/library/l-lex.html>

```
% {
    /* calc.lex */
#include "global.h"
#include "calc.h"
#include <stdlib.h>
% }
white    [ \t]+
digit    [0-9]
integer  {digit}+
exponent [eE]([+-])?{integer}
real     {integer}("."{integer})?({exponent})?
%%
```

앞의 wordcount 와의 차이는?

```
%%
    {white} {}
    {real} { yylval=atof(yytext);
              return(NUMBER); }

    "+" { return(PLUS); }
    "-" { return(MINUS); }
    "*" { return(TIMES); }
    "/" { return(DIVIDE); }
    "^" { return(POWER); }
    "(" { return(LEFT_PARENTHESIS); }
    ")" { return(RIGHT_PARENTHESIS); }
    }

    "\n" { return(END); }

%%

int yywrap(void) {
    return 1;
}
```

이번 주에 한 것

- Lexical analysis
 - 프로그램을 token으로 끊어내는 과정
 - 함수 호출할 때 마다 하나씩 lexeme을 내어준다.
 - 정규표현식으로 만들고 FSA 그린 후 코딩할 수 있다.
 - 아니면, 도구를 써도 된다.

다음 시간 부터 할 것

- Syntax derivation (구문 분석)