유한 오토마타 예제 작성하기

컴파일러 3번째 수업

경민기, 2025-09-16

학습 목표

- DFA의 5-튜플 정의를 이해한다.
- 전이 테이블을 코드로 옮기는 절차를 익힌다.
- C 언어로 간단한 DFA 인식기를 구현하고 실행해본다.

DFA (Deterministic Finite Automata)

- DFA는 “유한한 상태”를 가지며, 현재 상태와 다음에 읽은 1개의 입력기호가 주어지면 다음 상태가 하나로 결정되는 (=결정적) 오토마타이다.
- DFA = (Q, Σ, δ, q0, F)
  - Q: 유한한 상태 집합
  - Σ: 유한한 입력 알파벳
  - δ: 전이함수
    - $\delta : Q \times \Sigma^* \rightarrow Q$
  - q0: 시작 상태, $q0 \in Q$
  - $F \subseteq Q$: 종결 상태들의 유한 집합

어떻게 동작하나

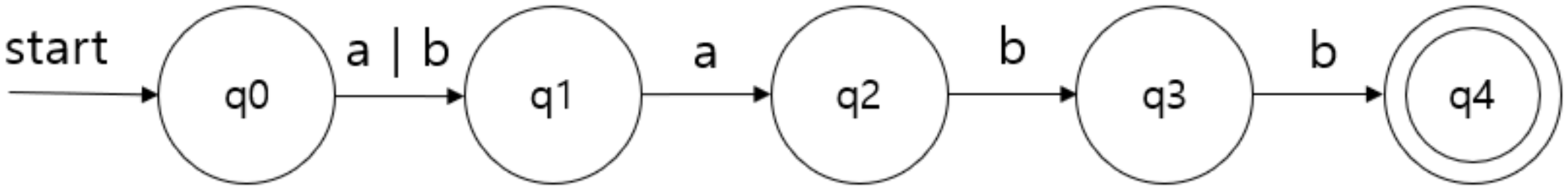
- 입력 문자열을 왼쪽 → 오른쪽으로 한 글자씩 읽으면서 δ를 따라 상태를 갱신함
- 문자열을 끝까지 읽은 후 현재 상태가 F에 속하면 수용, 아니면 거부(reject)
- 확장 전이함수 $\hat{\delta}(q, w)$를 쓰면,
  - $\hat{\delta}(q, \varepsilon) = q$ (빈 문자열에서는 상태 변화 없음)
  - $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ (접미 한 글자를 더 읽어 갱신)

예제 설명

예제 문제 정의

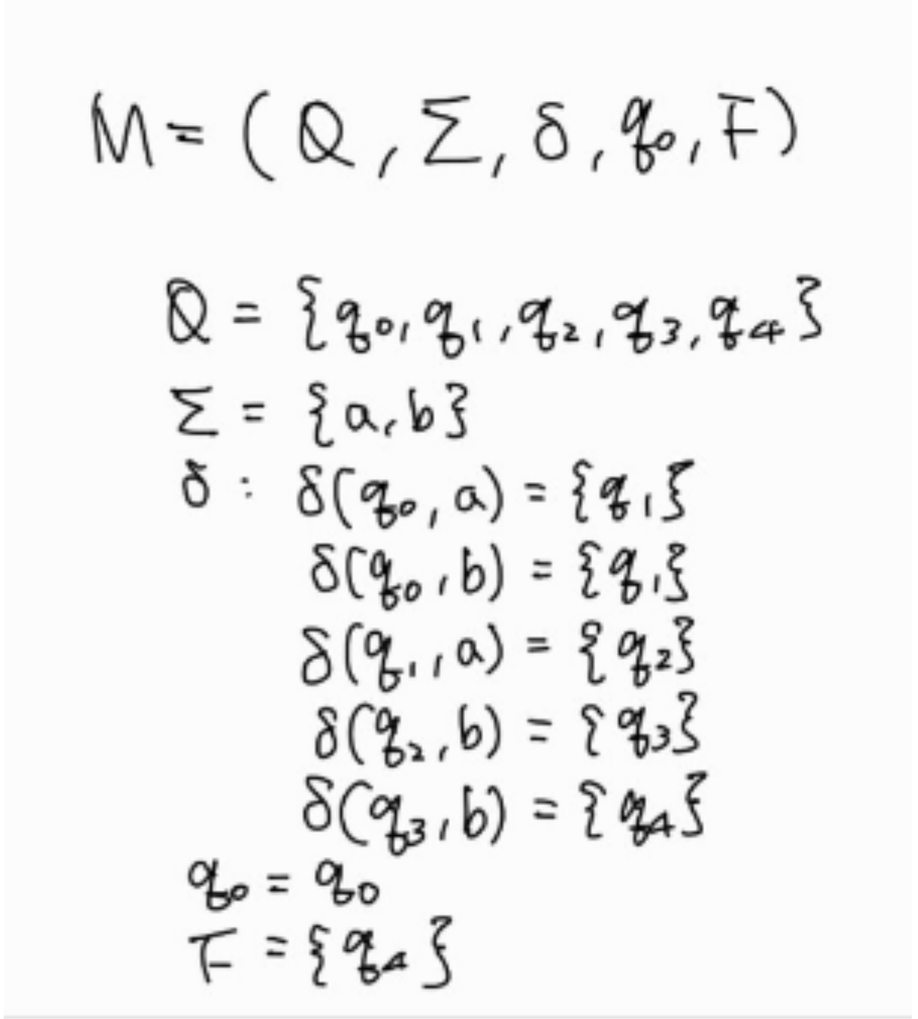
- 입력 알파벳 Σ = {'a','b'}
- 언어 L = { w ∈ {a,b}* | w가 "abb"로 끝난다 }
- 수용 조건: 입력 전체를 처리한 뒤 상태 ∈ F면 ACCEPT

상태전이도



DFA에 따른 형식적 표현방법

- DFA = (Q, Σ, δ, q0, F)
  - Q: 유한한 상태 집합
  - Σ: 유한한 입력 알파벳
  - δ: 전이함수
    - $\delta : Q \times \Sigma_* \rightarrow Q$
  - q0: 시작 상태, $q0 \in Q$
  - $F \subseteq Q$: 종결 상태들의 유한 집합



상태전이표

현재 상태 \ 입력	a	b
q0	q1	q0
q1	q1	q2
q2	q1	q3
q3	q1	q0

코드 설명

dfa_endswith_abb.c (1)

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// 알파벳: { 'a', 'b' } -> 인덱스 {0, 1}
static int sym_to_idx(int c) {
    if (c == 'a') return 0;
    if (c == 'b') return 1;
    return -1; // 알파벳 이외
}

enum { Q0=0, Q1, Q2, Q3, NUM_STATES };

// 전이 테이블: next_state[current_state][symbol_index]
static const int next_state[NUM_STATES][2] = {
    /* from Q0 */
      { Q1, Q0 }, // 'a'->Q1, 'b'->Q0
    /* from Q1 */
      { Q1, Q2 }, // 'a'->Q1, 'b'->Q2
    /* from Q2 */
      { Q1, Q3 }, // 'a'->Q1, 'b'->Q3
    /* from Q3 */
      { Q1, Q0 }, // 'a'->Q1, 'b'->Q0  (끝이 abb였는데 추가 입력에 따라 이동)
};
```

dfa_endswith_abb.c (2)

```
static int is_accept_state(int q) {
    return q == Q3;
}
```

```
int main(void) {
    char buf[1024];

    printf("Enter strings over {a,b}. Ctrl+D/Ctrl+Z to end.\n");
    while (fgets(buf, sizeof(buf), stdin)) {
        // 개행 제거
        size_t n = strlen(buf);
        while (n && (buf[n-1] == '\n' || buf[n-1] == '\r')) buf[--n] = '\0';

        int state = Q0;
        int ok = 1;
```

dfa_endswith_abb.c (3)

```
    for (size_t i = 0; i < n; ++i) {
        int idx = sym_to_idx(buf[i]);
        if (idx < 0) { // 유효하지 않은 문자
            ok = 0;
            break;
        }
        state = next_state[state][idx];
    }
    if (!ok) {
        printf("input=\"%s\" => REJECT (invalid symbol)\n",
buf);
    } else if (is_accept_state(state)) {
        printf("input=\"%s\" => ACCEPT\n", buf);
    } else {
        printf("input=\"%s\" => REJECT\n", buf);
    }
}
    return 0;
}
```

dfa_endswith_abb 실행

컴파일

gcc -O2 -Wall -Wextra -o dfa_endswith_abb dfa_endswith_abb.c

실행 (여러 줄 입력 가능)

./dfa_endswith_abb

abba # REJECT

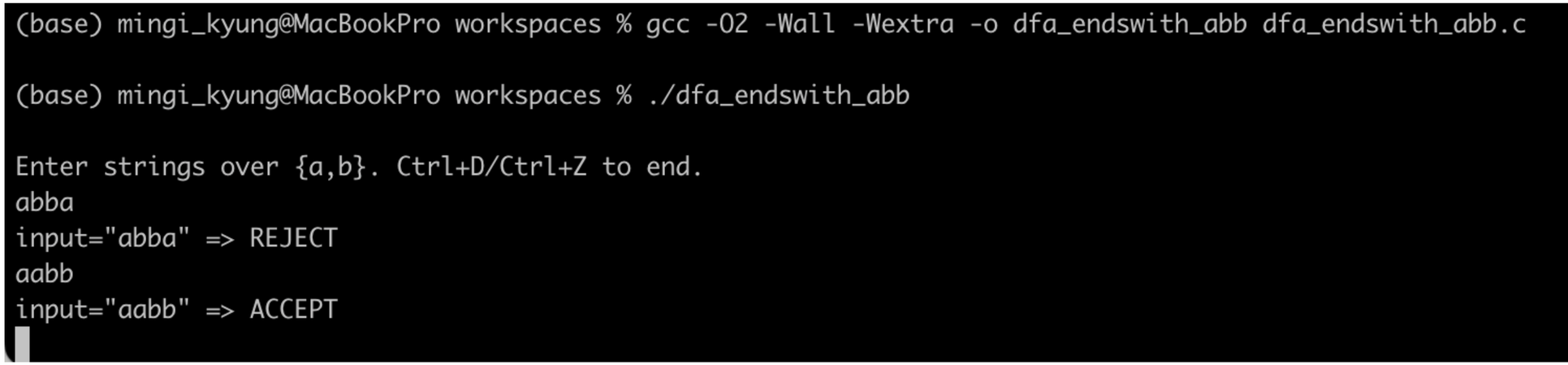
aabb # ACCEPT

abb # ACCEPT

babbab # REJECT

aaabbb # REJECT

xx # REJECT (invalid symbol)



다음에 수업할 내용

지금까지의 문제는 무엇인가

- 글자를 일일히 표현/검증하는 것이 큰 문제가 됨
- 그래서 flex / bison 같은 도구를 씀

Flex

정규표현식을 써서 입력한 문자가 제대로 된 것인지를 검증

```
%{
#include "parser.tab.h"      /* bison이 만든 토큰 선언 포함 */
%}

%%              /* --- 패턴 → 동작 --- */
[0-9]+    { yylval = atoi(yytext); return NUM; }
[ \t\r\n]+     ;                  /* 공백 무시 */
.         { return yytext[0]; }   /* + - * / ( ) 등 단일문자 토큰 */

%%
```

정규표현식

Bison

```
%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char *s){ fprintf(stderr, "error: %s\n", s); }

int yylex(void);

%}
```

```
%token NUM

%left '+' '-'

%left '*' '/'


%%

input   : /* empty */

        | input expr '\n'   { printf("%d\n", $2); }

        ;

expr    : expr '+' expr     { $$ = $1 + $3; }

        | expr '-' expr     { $$ = $1 - $3; }

        | expr '*' expr     { $$ = $1 * $3; }

        | expr '/' expr     { $$ = $1 / $3; }

        | '(' expr ')'      { $$ = $2; }

        | NUM               { $$ = $1; }

        ;

%%
```

정규표현식

정규표현식

- 정규표현식(Regular Expression, 줄여서 Regex)은 문자열에서 특정한 패턴을 찾거나, 검사하거나, 변환하기 위해 사용하는 일종의 형식 언어
  - 패턴 정의: 단순한 문자열 검색이 아니라, "규칙"을 정의해서 문자열 집합을 표현할 수 있음
  - 검색/매칭: 텍스트에서 특정 규칙을 만족하는 부분 문자열을 찾아냄
  - 변환/치환: 찾아낸 문자열을 다른 문자열로 바꾸거나 가공 가능

정규표현식 규칙

구분	표현식	의미	예시
문자 매칭	.	임의의 한 문자	a.c → abc, axc
	[]	문자 집합 중 하나	[abc] → a, b, c
	[^]	괄호 안에 없는 문자	`[^0-9]` → 숫자가 아닌 문자
	-	범위 지정	`[a-z]` → 소문자 알파벳
수량(반복)	*	0회 이상 반복	`a*` → `""`, `a`, `aa`
	+	1회 이상 반복	`a+` → `a`, `aa`, `aaa`
	?	0 또는 1회	`a?` → `""`, `a`
	{n}	정확히 n회 반복	a{3} → aaa
	{n,}	n회 이상	a{2,} → aa, aaa, ...
	{n,m}	n회 이상 m회 이하	a{2,4} → aa, aaa, aaaa
앵커(위치)	^	문자열의 시작	^abc → "abc..."
	$	문자열의 끝	xyz$ → "...xyz"
	\b	단어 경계	\bcat\b → "cat"만 매칭
	\B	단어 경계 아님	\Bcat\B → "concatenate" 속 cat
그룹/선택	(...)	그룹 지정	(abc)+ → "abc", "abcabc"
	`	`	OR (선택)
이스케이프	\	특수문자 그대로 사용	`\.` → `"."`
문자 클래스	\d	숫자 (0–9)	\d\d → "12", "99"
	\D	숫자가 아닌 문자	\D+ → "abc", "@"
	\w	단어 문자 (알파벳, 숫자, _)	\w+ → "hello_123"
	\W	단어 문자가 아닌 것	`\W` → `"  "`, `"!"`
	\s	공백 문자 (스페이스, 탭, 개행)	\s+ → " "
	\S	공백이 아닌 문자	\S+ → "word"

예시

- abc → 문자열 "abc"가 그대로 있는 경우 찾음
- [0-9]+ → 하나 이상의 숫자(예: "123", "98765")
- ^Hello → 문자열이 "Hello"로 시작하는 경우
- world$ → 문자열이 "world"로 끝나는 경우

이메일 주소의 정규표현식

- 이메일 주소 ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-z]{2,}$
  - ^[...] → 문자열 시작
  - [a-zA-Z0-9._%+-]+ → 로컬 파트 (알파벳, 숫자, 특수문자 허용)
  - @ → 반드시 있어야 함
  - [a-zA-Z0-9.-]+ → 도메인 이름
  - \. → 점(.)
  - [a-zA-Z]{2,} → 최상위 도메인(최소 2자 이상)
  - $ → 문자열 끝

인터넷 주소의 정규표현식 찾기

- 의외로 자주 쓰는 내용이니 찾아보시기 바랍니다.
- https://regexr.com/