Assignment #2

과 목 명	형식언어 - 01분반
담당교수	손윤식 교수님
학 과	컴퓨터공학전공
학 번	2018111999
이 름	심 재 혁
제 출 일	2022.05/15

본 과제는 NFA(입실론 포함) → reduced DFA로 변환하는 과제이다.

코드는 C++로 작성하였으며, 크게 두 가지로 분류된다.

1]. NFA(입실론 포함) → DFA

2]. DFA → Reduced DFA

모두 하나의 소스 파일 (main.cpp)로 작성되었고, 아쉽게도 입출력은 진행하지 못해 콘솔 창에서 입력 받는 것으로 진행하였다. 자세한 설명을 아래에서 진행할 예정이다.

```
int main()
{
    User_Input(); // [1] 사용자로부터 입력 받는 과정

    E_NFA_TO_DFA_Converting(); // [2] E-NFA --> DFA 변환 과정

    Print_DFA(); // [3] 변환된 DFA 출력 (DFA의 결정 / 비결정 상태또한 출력)

    DFA_To_Reduced_DFA_Converting(); // [4] DFA --> Redured DFA 변환 과정

    Print_Reduced_DFA(); // [5] 변환된 Reduced DFA 출력
}
```

1. Main 서브루틴이다.

   [1]. 사용자로부터 E-NFA에 대한 정보를 입력을 받는다.

   [2]. E-NFA를 DFA로 변환한다.

   [3]. 변환된 DFA 정보를 출력한다. 또한, 결정 / 비결정 상태 또한 출력한다.

   [4]. 변환된 DFA를 Reduced DFA로 변환한다.

   [5]. 변환된 Reduced DFA를 출력한다.

각 함수에 대한 설명을 이어서 한다.

```
void User_Input()
{
    cout << "상태의 개수를 입력하세요 : ";
    cin >> S_Num;
    cout << '₩n';  // --------------- [1]

    cout << "입력의 개수를 입력하세요 : ";
    cin >> I_Num;
    cout << '₩n'; // --------------- [2]

    cout << "상태를 입력하세요 (처음에 입력한 상태는 시작 상태입니다.) : " << '₩n';
    for (int i = 0; i < S_Num; i++)
    {
        string I1;
        cin >> I1;
        if (i == S_Num - 1)
            Final_Status = I1;
        Status.push_back(I1);
    }
    cout << '₩n'; // --------------- [3]
```

```
    cout << "종결 상태를 입력하세요 : " << '₩n';
    string I1;
    cin >> I1;
    vector<string> temp = Split(I1, ',');
    for (auto e : temp)
        Final_Status_Vec.push_back(e); // --------------- [4]

    cout << "입력을 입력하세요 (입실론을 입력하고 싶으면 'epsilon'으로 입력해주세요) : " << '₩n';
    for (int i = 0; i < I_Num; i++)
    {
        string I1;
        cin >> I1;
        if (I1 == "epsilon")
            epsilon_Index = i;
        Input.push_back(I1);
    }
    cout << '₩n'; // --------------- [5]

    // 상태 + 입력을 순차적으로 입력받는다.
```

```
    for (int i = 0; i < S_Num; i++)
    {
        for (int j = 0; j < I_Num; j++)
        {
            cout << Status[i] << "에서 " << Input[j] << "로의 전이값을 입력하세요 (여러개의 상태를 입력하고 싶으시면 ','로 구분 지어주세요. 전이값이 없으면 ₩"NON₩"으로 입력해주세요) : " << '₩n';
            string I1;
            cin >> I1;
            if (I1 == "NON")
                E_NFA[{Status[i], Input[j]}].push_back("NON");
            else
            {
                vector<string> temp = Split(I1, ',');
                E_NFA[{Status[i], Input[j]}] = temp;
            }
            cout << '₩n';
        }
        if (epsilon_Index == -1)
            E_NFA[{Status[i], "epsilon"}].push_back("NON"); // E-NFA가 아닌, 일반 NFA일 때 --> 기본 프로그램 자체가 E-NFA를 기반으로 동작하기 때문에
                                                            // 입력이 입실론일 때 나오는 전이 상태가 전부 공집합임을 명시해준다.
    } // ------------ [6]
}
```

2. User_Input() 함수에서 사용자로부터 콘솔창에 입력받는다.

[1]. 상태의 개수

[2]. 입력의 개수

[3]. 상태 (상태를 여러 개로 묶은 vector<string>으로 처리. 처음 입력한 상태는 시작 상태다.)

[4]. 종결 상태 (','로 분리하여 상태를 받을 수 있도록 하였다. 종결 상태 또한 여러 개로 묶은 vector<string>으로 처리)

[5] 입력 (입력을 여러 개로 묶은 vector<string>으로 처리 E-NFA이므로 입실론을 포함할 수 있도록 epsilon 문자열이 들어올 시 따로 처리하도록 구현)

[6]. 상태 + 입력 → 전이 상태 입력 (상태(string) + 입력(string) (pair<string, string>) → 전이 상태 (vector<string>. 전이 상태는 NFA의 특성 상 없을 수도, 여러 개 있을 수도 있기 때문에 vector<string>으로 여러 개를 입력 받을 수 있도록 하였다.)

[주의] E-NFA는 일반 NFA, DFA를 모두 포함하는 개념이다. 따라서 입실론으로부터 전이 상태가 나오지 않거나 입력으로 입실론이 없어도 있는 것처럼 동작해야 한다. 따라서 기본 프로그램을 E-NFA를 기반으로 동작하도록 구현하였으며, 대신 입실론으로부터 얻어지는 전이 상태가 없으면 NON으로 처리하도록 하여 입력이 입실론일 때 나오는 전이 상태가 전부 공집합임을 명시해준다.

```
void E_NFA_TO_DFA_Converting() // E_NFA --> DFA 변환 함수
{
    for (auto i = 0; i < S_Num; i++)
    {
        E_Closure_For_S[Status[i]].push_back(Status[i]); // ---------------- [1]

        Convert_To_Epsilon_Closure(Status[i], Status[i]); // ---------------- [2]
        sort(E_Closure_For_S[Status[i]].begin(), E_Closure_For_S[Status[i]].end());
        if (i == 0)
        {
            new_Status_For_DFA[new_S_Index] = E_Closure_For_S[Status[i]];
            new_S_Index++;
            Closure_Queue.push(E_Closure_For_S[Status[i]]);
        } // ---------------- [3]
    } // 각 상태에서의 E-Closure 계산 (자기 자신 + 입실론을 보고 갈 수 있는 상태)

    if (epsilon_Index != -1)
        Input.erase(Input.begin() + epsilon_Index); // ---------------- [4]
    // 바로 위에서 입실론 클로저가 계산 됐으면 더 이상 입실론이 필요없기 때문에 입력에 입실론이 있었으면 이를 제거해줘야 한다.
```

E-NFA_TO_DFA_Converting 함수의 일부분이다. 해당 캡처 부분은 E-NFA에서의 상태에 대해 E-Closure를 계산하는 부분이다.

[1]. 모든 상태에서의 E-Closure는 항상 자기 자신을 포함한다. 따라서 이를 반영한다.

[2]. 해당 상태에서 입실론을 보고 갈 수 있는 상태를 전부 반영한다. (Convert_To_Epsilon_Closure 함수. 자세한 설명은 아래에서 할 예정)

[3]. 시작 상태에서 나온 E-Closure를 Closure_Queue에 담는다.

[4]. E-Closure를 처리했으면 더 이상의 입실론 입력이 필요 없기에 입력에서 입실론을 제거한다.

이 과정이 끝나면 본격적으로 DFA로 변환할 준비가 끝난 것이다.

```
void Convert_To_Epsilon_Closure(string Status1, string Status2)
{
    for (auto ee : E_NFA[{Status2, "epsilon"}])
    {
        if (ee == "NON")
            return; // [1] 입실론을 보고 갈 수 있는 상태가 없으면 해당 재귀 상태 빠져나옴
        if (ee == Status2)
            return; // [2] 입실론을 보고 갈 수 있는 상태가 본인이면 해당 재귀 상태 빠져나옴
        Convert_To_Epsilon_Closure(Status1, ee);
        E_Closure_For_S[Status1].push_back(ee); // -------------- [3]
    }
} // E-NFA에서의 상태의 E-Closure 변환 (top-down 형식 (재귀), 입실론을 보고 갈 수 있는 상태가 연쇄적일 수 있기 때문에 재귀 형식을 택함)
```

Convert_To_Epsilon_Closure 함수의 내용이다. E-NFA에서의 상태에서 '입실론' 입력을 보고 갈 수 있는 상태를 체크하는 과정이며, 이 때 재귀를 사용해야 한다.

[1]. 입실론을 보고 갈 수 있는 상태가 없으면 해당 재귀 상태 빠져나옴

[2]. 입실론을 보고 갈 수 있는 상태가 본인이면 해당 재귀 상태 빠져나옴

[3]. 나머지 부분에 대한 E-Closure를 배열에 담아둠 (E_Closure_For_S)

```
while (!Closure_Queue.empty()) // 큐가 빌 때까지 E-NFA --> DFA 변환을 진행 (큐가 비었다는 의미는 DFA에서의 새로운 상태가 없다는 뜻)
{
    vector<string> Poped_From_Closure_Queue = Closure_Queue.front();
    Closure_Queue.pop(); // -------------- [1]

    int new_status_temp_Index = 0;

    for (auto const& e : new_Status_For_DFA)
    {
        if (Poped_From_Closure_Queue == e.second)
            new_status_temp_Index = e.first;
    }

    for (auto e : Final_Status_Vec)
    {
        if (find(Poped_From_Closure_Queue.begin(), Poped_From_Closure_Queue.end(), e) != Poped_From_Closure_Queue.end())
            Class_Divide[1].push_back(to_string(new_status_temp_Index));
    }
    // [2] 큐에서 pop된 새로운 상태에 기존의 E-NFA에서의 종결 상태가 들어가있으면 종결 상태에 대한 부분을 먼저 파티셔닝 해준다. (종결 : 1)
```

방금 전에 시작 상태에서 나온 E-Closure를 Closure_Queue에 담았었다. DFA로 변환하면서 E-Closure(상태) + 입력 ➔ 전이 상태(E-Closure)가 지속적으로 나올 예정인데, 전이 상태가 새로운 상태일 때마다 계속 큐에 넣으면서 연산을 진행해야 한다. 따라서 큐가 빌 때까지 변환을 지속하는 것이다.

[1]. 큐의 front 원소를 임시 변수에 저장한 후 pop한다.

[2]. 큐에서 pop된 새로운 상태에 기존의 E-NFA에서의 종결 상태가 들어가 있으면 종결 상태에 대한 부분을 먼저 파티셔닝 해준다. (종결 : 1)

DFA로 변환하는 과정 중 종결 / 비종결 상태를 구분해야 하는데, 이는 Reduced DFA로 변환할 때 반드시 있어야 할 과정이다. 현재는 종결 상태만 담도록 하였는데, 비종결 상태는 종결 상태를 출

력할 때 담을 것이다.

```
for (auto input : Input)
{
    vector<string> new_status_temp;
    for (auto temp_closure : Poped_From_Closure_Queue)
    {

        for (auto e_nfa_value : E_NFA[{temp_closure, input}]) // e_nfa_value : E-NFA의 상태 + E-NFA의 입력 --> E-NFA(전이 상태)
        {
            if (e_nfa_value == "NON")
                continue;
            if (new_status_temp.size() == 0)
                new_status_temp = E_Closure_For_S[e_nfa_value];
            else
                Union_Func(new_status_temp, E_Closure_For_S[e_nfa_value]);
        } // [3] e_nfa_value에 대한 E-Closure를 계산한다. (합집합 연산)
    }
    if (new_status_temp.size() != 0)
    {
        sort(new_status_temp.begin(), new_status_temp.end()); // DFA의 상태 값이 뒤죽박죽일 수 있으니 정렬

        if (Check(new_status_temp))
        {
            new_Status_For_DFA[new_S_Index] = new_status_temp;
            new_S_Index++;
            Closure_Queue.push(new_status_temp);
        } // [4] 189~197번 줄에서의 E-Closure 변환이 새로운 상태면 DFA에서의 새로운 상태에 추가 + 큐에 추가

    int temp_closure_Index = 0;
    int new_status_temp_Index = 0;
    for (auto const& e : new_Status_For_DFA)
    {
        if (Poped_From_Closure_Queue == e.second)
            temp_closure_Index = e.first;

        if (new_status_temp == e.second)
            new_status_temp_Index = e.first;
    }
    DFA[{to_string(temp_closure_Index), input}] = to_string(new_status_temp_Index);
    // [5] 변환된 상태에 대해 DFA를 추가해준다. (집합으로 이루어진 DFA 상태는 하나의 인덱스로 치환해서 넣어줌.
    // 집합으로 이루어진 DFA 전이 상태 또한 마찬가지.)

}
```

[3]. e_nfa_value에 대한 E-Closure를 계산한다. (합집합 연산)

[4]. 189~197번 줄에서의 E-Closure 변환이 새로운 상태면 DFA에서의 새로운 상태에 추가 + 큐에 추가한다.

[5] 변환된 상태에 대해 DFA를 추가해준다. (집합으로 이루어진 DFA 상태는 하나의 인덱스로 치환해서 넣어준다. (집합으로 이루어진 DFA 전이 상태 또한 마찬가지))

[5]의 부연 설명을 하면, NFA로 얻어진 DFA의 상태는 무수히 많을 수도 있기 때문에 어느 정도 큰 범위를 가진 정수 형으로 상태를 대체한 것이다.

```
void Print_DFA()
{
    vector<string> temp;

    cout << '₩n' << "-----------------------------------------------------" << '₩n' << "NFA(E-NFA 포함) --> 최종 DFA : " << '₩n' << '₩n';
    for (auto const& e : DFA)
    {
        cout << e.first.first << " (상태)  +  " << e.first.second << " (입력)  ==>  " << e.second << '₩n';
        temp.push_back(e.first.first);
    } // ------------- [1]

    cout << '₩n' << "*************" << '₩n';

    cout << "DFA에서의 시작 상태 : " << 1 << '₩n';

    cout << "DFA에서의 종결 상태 : ";
    for (auto class_divide : Class_Divide)
    {
        if (class_divide.first == 1)
        {
            for (auto final_state : class_divide.second)
            {
                cout << final_state << " ";
                temp.erase(remove(temp.begin(), temp.end(), final_state), temp.end()); // 결정 상태 원소 지우기
            }
        }
        cout << '₩n';
    }

    for (auto e : temp)
        Class_Divide[2].push_back(e);

    sort(Class_Divide[2].begin(), Class_Divide[2].end());
    Class_Divide[2].erase(unique(Class_Divide[2].begin(), Class_Divide[2].end()), Class_Divide[2].end()); // 중복된 원소 제거
    // ----------------- [2]

    cout << "DFA에서의 비종결 상태 : ";
    for (auto class_divide : Class_Divide)
    {
        if (class_divide.first == 2)
        {
            for (auto final_state : class_divide.second)
                cout << final_state << " ";
        }
    } // ------------------ [3]
    cout << '₩n';
}
```

Print_DFA() 함수의 내용이다.

[1]. E-NFA로부터 변환된 DFA 상태를 출력한다. (상태 + 입력 ➔ 전이 상태 형태)

DFA 상태와 입력으로부터 얻어지는 전이 상태가 없을 때는 출력하지 않는데, 전이 상태가 없는 경우를 DFA에 전부 반영하게 되면 Reduced DFA 변환 과정에서 혼동이 와 그 부분은 아쉽게도 제외하였다.

[2]. 시작 상태, 종결 상태, 비 종결 상태를 출력한다. 비 종결 상태를 얻는 과정은

(1). DFA에서의 상태를 모두 담는다.

(2). DFA에서의 종결 상태를 제외한다.

의 간단한 과정이다.

[3]. 비 종결 상태를 출력한다.

```
void DFA_To_Reduced_DFA_Converting() // DFA --> Reduced DFA 변환 과정
{
    map<string, vector<int>> temporary_to_class; // 각각의 파티셔닝 그룹에서 매칭되는 그룹에 대한 임시 변수
    // ex. 임의의 파티셔닝 그룹 '1 : {A, B, C}'에서 입력 a, b ,c에서의 파티셔닝 매칭 그룹이 각각 A : {2, 1, 1}, B : {1, 1, 2}, C : {2, 1, 2}일 때 이를 저장
    int Class_Divide_Num = 0;
```

DFA_To_Reduced_DFA_Converting 함수의 내용이다.

Reduced DFA 변환 시 '파티셔닝'이라는 과정이 필요하다. 파티셔닝은 처음에 DFA의 종결 상태 / 비 종결 상태로 구분 지어야 한다.

(Ex. DFA의 종결 상태가 A, B, C, D / 비 종결 상태가 E, F이면 이를 1 : {A, B, C, D}, 2 : {E, F}로 파티셔닝 하는 과정이다. (map<int, vector<string>> 방식) 이는 Class_Divide 변수에 담는다.

또한, 각 파티셔닝 그룹에 있는 상태들에 대해 입력이 주어질 때 매칭되는 파티셔닝 그룹을 정해줘야 하는데, 이를 위한 임시 변수(temporary_to_class)를 선언하였다.

(Ex. 임의의 파티셔닝 그룹 '1 : {A, B, C}'에서 입력 a, b ,c에서의 파티셔닝 매칭 그룹이 각각 A : {2, 1, 1}, B : {1, 1, 2}, C : {2, 1, 2} (map<string, vector<int>> 방식) 일 때 이를 temporary_to_class에 저장하는 방식)

```
    while (true)
    {
        Class_Num = 0; // [1] 파티셔닝이 반복될 때마다 파티셔닝 넘버링을 초기화
        Class_Divide_Num = Class_Divide_temp.size();
        Class_Divide_temp.clear();

        for (map<int, vector<string>>::iterator j = Class_Divide.begin(); j != Class_Divide.end(); j++) // 묶여진 각각의 파티셔닝 그룹에 대해 반복문을 돌려 새로운 파티셔닝이 있는지 확인한다.
        {
            temporary_to_class.clear();

            for (auto j_second : j->second) // 묶은 문자열
            {
                for (auto input : Input) // [2] 들어오는 입력마다 파티셔닝 그룹을 매칭
                {
                    string s = DFA[{j_second, input}];
                    if (s == "")
                    {
                        temporary_to_class[j_second].push_back(-1);
                        continue;
                    } // [2 - 1] DFA의 상태 + DFA의 입력에서 오는 DFA 전이 상태가 없으면 -1를 대신 넣어줌
                    for (auto const& class_divide : Class_Divide)
                    {
                        for (auto class_divide_s : class_divide.second)
                        {
                            if (class_divide_s == s)
                                temporary_to_class[j_second].push_back(class_divide.first);
                        }
                    } // [2 - 2] 그게 아닐 시 기존의 DFA 전이 상태에 대한 파티셔닝 그룹을 매칭해준다.
                    // ex. 임의의 파티셔닝 그룹 '1 : {A, B, C}'에서 입력 a, b ,c에서의 파티셔닝 매칭 그룹을 각각 A : {2, 1, 1}로 매칭하는 방식
                }
            }
```

[1]. 파티셔닝이 반복될 때마다 파티셔닝 넘버링을 초기화한다.

[2]. 들어오는 입력마다 파티셔닝 그룹을 매칭한다.

[2 - 1]. DFA의 상태 + DFA의 입력에서 오는 DFA 전이 상태가 없으면 -1를 대신 넣어준다.

[2 - 2]. 그게 아닐 시 기존의 DFA 전이 상태에 대한 파티셔닝 그룹을 매칭해준다.

```
    vector<pair<string, vector<int>>> vec(temporary_to_class.begin(), temporary_to_class.end()); // [4] 연산을 간편하게 하기 위해 각각의 파티셔닝 그룹에서 매칭되는 그룹을 벡터화

    bool* false_ee = new bool[vec.size()]();
    for (int q = 0; q < vec.size(); q++)
    {
        if (!false_ee[q])
        {
            Class_Divide_temp[++Class_Num].push_back(vec[q].first);
            false_ee[q] = true;
        }
        for (int w = q + 1; w < vec.size(); w++)
        {
            if (vec[q].second == vec[w].second && !false_ee[w])
            {
                Class_Divide_temp[Class_Num].push_back(vec[w].first);
                false_ee[w] = true;
            }
        }
    } // [5] 브루트포스를 사용하여 새로운 파티셔닝 그룹을 만드는 과정  (ex. 1 - 2 ~ 4 / 2 - 3 ~ 4 / 3 - 4)
    // (ex. 1 : {A, B, C, D}이고, 입력 a, b에서의 파티셔닝 그룹 매칭 결과가
    //각각 A : {1, 2}, B : {1, 1}, C : {1, 1}, D : {1, 2}이면 이를 3 : {A, D}, 4 : {B, C} 형식으로 분리)
}
```

[4]. 연산을 간편하게 하기 위해 각각의 파티셔닝 그룹에서 매칭되는 그룹을 벡터화

[5]. 브루트포스를 사용하여 새로운 파티셔닝 그룹을 만드는 과정이다.

(Ex. 1 : {A, B, C, D}이고, 입력 a, b에서의 파티셔닝 그룹 매칭 결과가 각각 A : {1, 2}, B : {1, 1}, C : {1, 1}, D : {1, 2}이면 이를 3 : {A, D}, 4 : {B, C} 형식으로 분리)

```
        Class_Divide.clear();
        Class_Divide = Class_Divide_temp;
        if (Class_Divide_Num == Class_Divide_temp.size())
            // [6] 새로운 파티셔닝 그룹의 개수과 기존의 파티셔닝 그룹의 개수가 같으면 더 이상 분리할 그룹이 없으므로 무한루프 종료
            break;
    }
```

[6]. 새로운 파티셔닝 그룹의 개수과 기존의 파티셔닝 그룹의 개수가 같으면 더 이상 분리할 그룹이 없으므로 무한루프를 종료한다.

```
for (auto const& cd_1 : Class_Divide) // [7] 분리된 파티셔닝 그룹은 Reduced-DFA에서의 새로운 상태가 된다. 해당 상태들에 대해
                                      // 기존의 DFA에서의 출력 상태를 분리된 파티셔닝 그룹에 매칭하여 Reduced-DFA의 새로운 전이 상태로 추가한다.
{
    for (auto const& input : Input)
    {
        string output = "";
        for (auto const& class_divide_s : cd_1.second)
        {
            string s = DFA[{class_divide_s, input}]; // F

            for (auto const& cd_2 : Class_Divide)
            {
                for (auto class_divide_s : cd_2.second)
                {
                    if (class_divide_s == s)
                    {
                        output = to_string(cd_2.first);
                        break;
                    }
                }
            }
        }
        if (output != "")
            Reduced_DFA[{to_string(cd_1.first), input}] = output;
    }
}
```

[7] 분리된 파티셔닝 그룹은 Reduced-DFA에서의 새로운 상태가 된다. 해당 상태들에 대해 기존의 DFA에서의 출력 상태를 분리된 파티셔닝 그룹에 매칭하여 Reduced-DFA의 새로운 전이 상태로

추가한다.

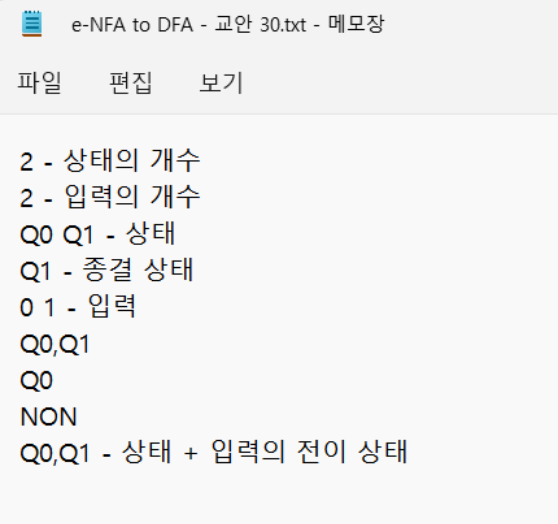
```
void Print_Reduced_DFA()
{
    cout << '₩n' << "-----------------------------------------------" << '₩n' << "DFA --> REDUCED DFA : " << '₩n' << '₩n';
    for (auto const& reduced_dfa : Reduced_DFA)
        cout << reduced_dfa.first.first << " (상태)  +  " << reduced_dfa.first.second << " (입력)  ==>  " << reduced_dfa.second << '₩n';
}
```

Pring_Reduced_DFA() 함수의 내용이다. DFA로부터 변환된 Reduced DFA를 출력한다.

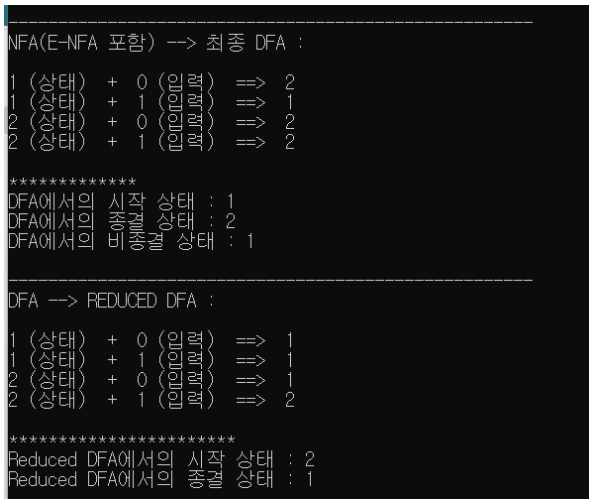
```
    for (auto const& class_divide : Class_Divide)
    {
        cout << class_divide.first << " : " << '₩n';
        for (auto const& class_divide_s : class_divide.second)
        {
            if (find(DFA_Start_Status.begin(), DFA_Start_Status.end(), class_divide_s) != DFA_Start_Status.end())
                Reduced_DFA_Start_Status.push_back(to_string(class_divide.first));
            if (find(DFA_Final_Status.begin(), DFA_Final_Status.end(), class_divide_s) != DFA_Final_Status.end())
                Reduced_DFA_Final_Status.push_back(to_string(class_divide.first));
        }
    }


    cout << '₩n' << "**********************" << '₩n';

    cout << "Reduced DFA에서의 시작 상태 : ";
    for (auto e : Reduced_DFA_Start_Status)
    {
        cout << e << " ";
    }
    cout << '₩n';

    cout << "Reduced DFA에서의 종결 상태 : ";
    for (auto e : Reduced_DFA_Final_Status)
    {
        cout << e << " ";
    }
    cout << '₩n';

}
```

Reduced DFA에 대한 시작 상태와 종결 상태를 출력한다. 새롭게 만들어진, Reduced DFA에서의 상태는 DFA에서의 상태와 완전히 다른 결과를 가진다. DFA의 상태들을 '포함'하는 파티셔닝 그룹이 Reduced DFA에서의 상태이기 때문에 Reduced DFA의 상태 중 기존 DFA의 시작 상태가 포함되면 Reduced DFA의 시작 상태가 되며, Reduced DFA의 상태 중 기존 DFA의 종결 상태가 포함되면 Reduced DFA의 종결 상태가 되는 간단한 구조이다.

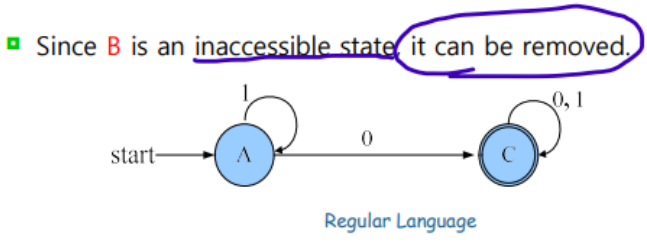
[실행 결과]



교안 p.30에 있는, 입실론이 입력으로 주어지지 않은 NFA(상태가 여러 개이기 때문)의 정보이다.



(실행 결과)

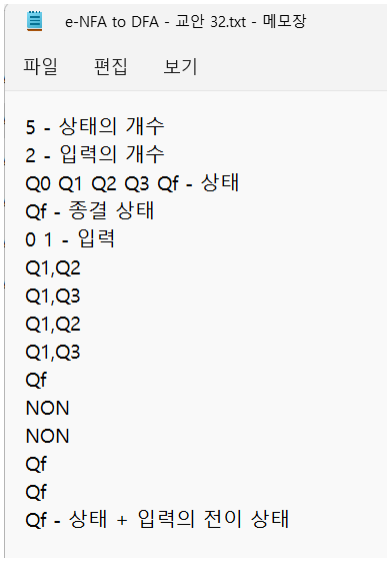


(E-NFA → DFA (교안의 정답))

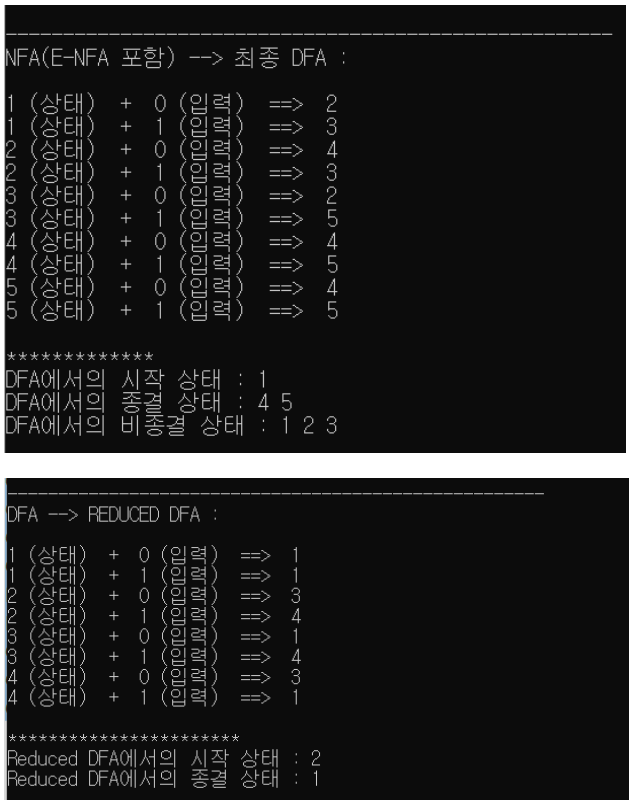
E-NFA → DFA : 출력 결과에서의 상태 4를 A, 상태 2를 B로 두고 보면 정확한 답을 얻어낼 수 있다.

DFA → Reduced DFA : 상태가 2개 뿐이며, 종결 상태 C, 비 종결 상태 A로, 파티셔닝 진행이 안되

기 때문에 상태의 넘버링만 달라지고 나머지 결과는 같다. 교안 정답 그림에서의 상태와 출력에서의 상태에 대한 매칭을 A – 2(시작 상태), C – 1(종결 상태)로 두고 보면 정확한 답을 얻어낼 수 있다.



교안 p.32에 있는, 입실론이 입력으로 주어지지 않은 NFA(상태가 여러 개이기 때문)의 정보이다.



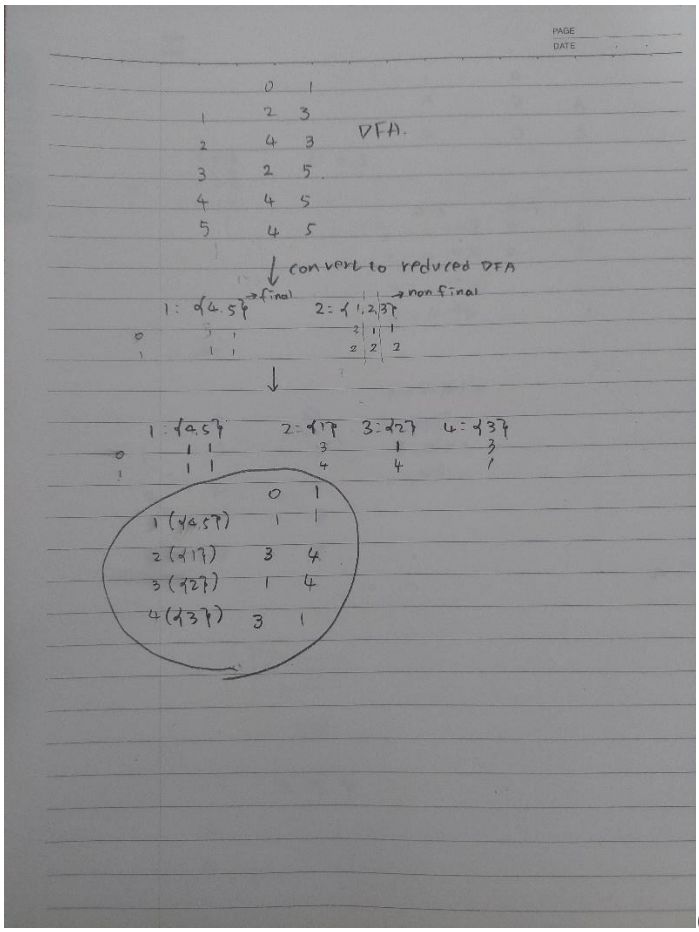
(실행 결과)

DFA :

δ'	0	1
q_0	q_1q_2	q_1q_3
q_1q_2	$q_1q_2q_f$	q_1q_3
q_1q_3	q_1q_2	$q_1q_3q_f$
$q_1q_2q_f$	$q_1q_2q_f$	$q_1q_3q_f$
$q_1q_3q_f$	$q_1q_2q_f$	$q_1q_3q_f$

(E-NFA → DFA (교안의 정답))

E-NFA → DFA : 출력 결과에서의 상태 4를 Q1Q2Qf, 상태 5를 Q1Q3Qf, 상태 1을 Q0, 상태 2를 Q1Q2, 상태 3을 Q1Q3으로 두고 보면 DFA에서의 정확한 답을 얻어낼 수 있다.



(DFA → Reduced DFA (수기 그림))

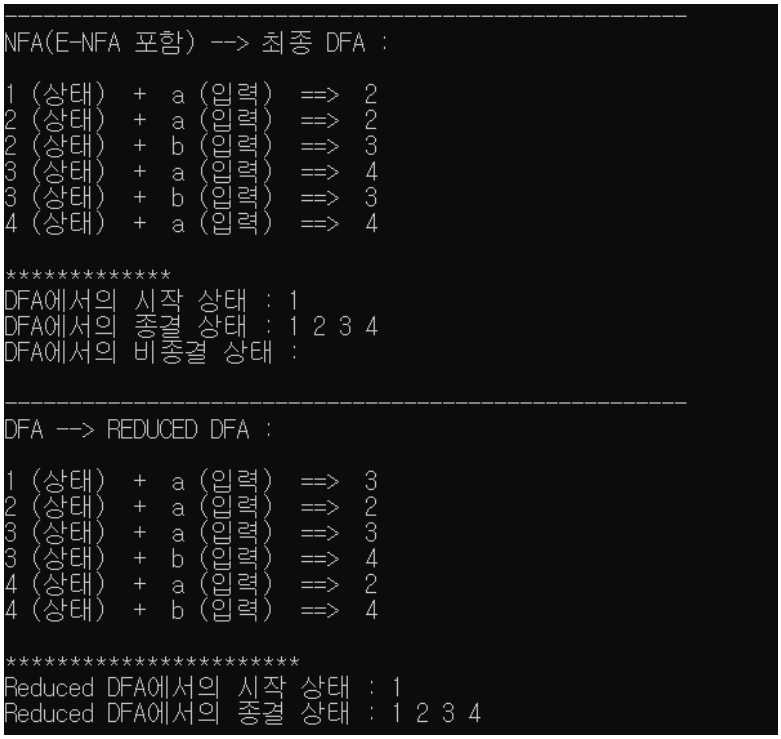
DFA → Reduced DFA : 파티셔닝을 전부 진행한 결과 수기 그림과 실행 결과의 Reduced DFA가 완벽하게 일치한다.

e-NFA to DFA - 교안 33.txt - 메모장

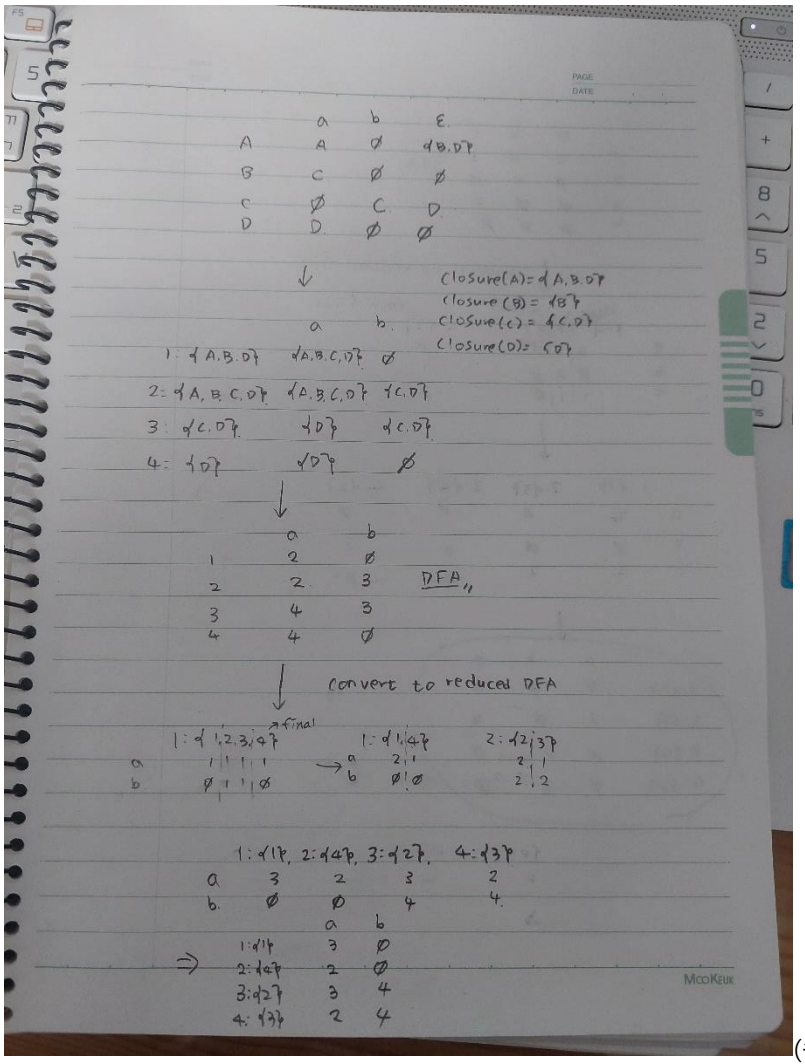
파일 편집 보기

```
4 - 상태의 개수
3 - 입력의 개수
A B C D - 상태
D - 종결 상태
a b epsilon - 입력
A
NON
B,D
C
NON
NON
NON
C
D
D
NON
NON - 상태 + 입력의 전이 상태
```

교안 p.32에 있는, 입실론이 입력으로 주어진 E-NFA(상태가 여러 개이기 때문)의 정보이다.



(출력 결과)



(수기 그림)

E-NFA → DFA : E-Closure를 계산한 후, 이에 따라 DFA를 변환하였다. 변환된 DFA에 대한 모든 상태가 본래의 E-NFA에서의 종결 상태를 포함한다. 따라서 종결 상태는 1, 2, 3, 4 전부이며, 비 종결 상태는 없다. 수기 그림과 비교한 결과 문제없이 DFA로 변환되었다.

DFA → Reduced DFA : 파티셔닝을 전부 진행한 결과 출력에 나온 Reduced DFA와 완벽하게 일치한다.

e-NFA to DFA - 교안 34.txt - 메모장

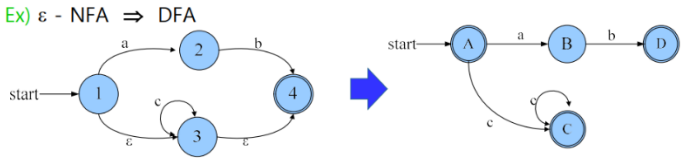
파일 편집 보기

```
4 - 상태의 개수
4 - 입력의 개수
1 2 3 4 - 상태
4 - 종결 상태
a b c epsilon
2 - 입력
NON
NON
3
NON
4
NON
NON
NON
NON
3
4
NON
NON
NON
4 - 상태 + 입력의 전이 상태
```

교안 p.34에 있는, 입실론이 입력으로 주어진 E-NFA의 정보이다.

```
------------------------------------------------------
NFA(E-NFA 포함) --> 최종 DFA :

1 (상태)  +  a (입력)  ==>  2
1 (상태)  +  c (입력)  ==>  3
2 (상태)  +  b (입력)  ==>  4
3 (상태)  +  c (입력)  ==>  3

**************
DFA에서의 시작 상태 : 1
DFA에서의 종결 상태 : 1 3 4
DFA에서의 비종결 상태 : 2

------------------------------------------------------
DFA --> REDUCED DFA :

1 (상태)  +  a (입력)  ==>  4
1 (상태)  +  c (입력)  ==>  2
2 (상태)  +  c (입력)  ==>  2
4 (상태)  +  b (입력)  ==>  3

************************
Reduced DFA에서의 시작 상태 : 1
Reduced DFA에서의 종결 상태 : 1 2 3
```

(출력 결과)

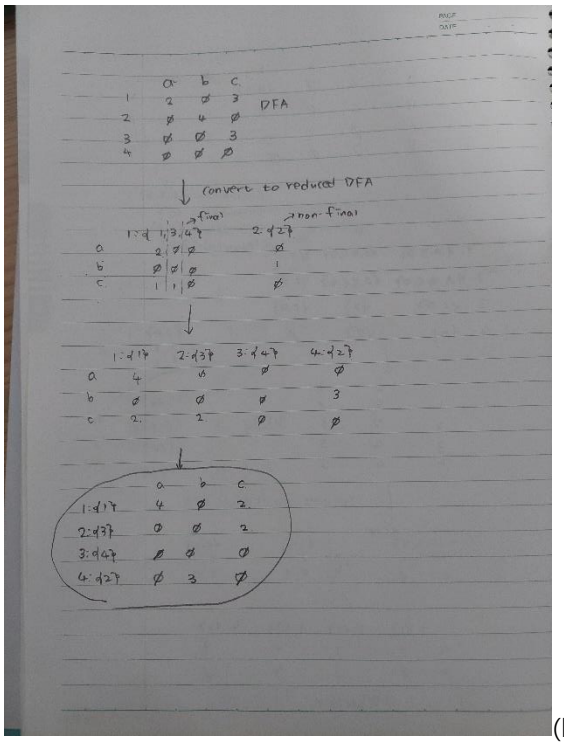


δ	a	b	c
CLOSURE(1) = {1,3,4} → [1,3,4]	CLOSURE(2) = {2} → [2]	ϕ	CLOSURE(3) = {3,4} → [3,4]
[2]	ϕ	CLOSURE(4) = {4} → [4]	ϕ
[3,4]	ϕ	ϕ	CLOSURE(3) = {3,4} → [3,4]
[4]	ϕ	ϕ	ϕ

A = [1,3,4], B = [2], C = [3,4], D = [4]

(E-NFA → DFA (교안의 정답))

E-NFA → DFA : 출력 결과에서의 상태 4를[1, 3, 4], 상태 2를 [2], 상태 3을 [3, 4], 상태 4를 [4]로 두고 보면 DFA에서의 정확한 답을 얻어낼 수 있다.



(DFA → Reduced DFA (수기 그림 정답))

DFA → Reduced DFA : 종결 상태 1, 3, 4 / 비 종결 상태 2로 처음에 파티셔닝을 진행하였고, 최종 진행 표를 동그라미로 표시하였다. 출력에 나온 Reduced DFA와 완벽하게 일치한다.

e-NFA to DFA - 교안 36.txt - 메모장

파일 편집 보기

```
6 - 상태의 개수
2 - 입력의 개수
A B C D E F - 상태
A,F - 종결 상태
a b - 입력
F
B
E
D
C
F
D
A
B
C
A
E - 상태 + 입력의 전이 상태
```

교안 p.36에서의 DFA이다.

```
------------------------------------------------------
NFA(E-NFA 포함) --> 최종 DFA :

1 (상태) + a (입력) ==> 2
1 (상태) + b (입력) ==> 3
2 (상태) + a (입력) ==> 1
2 (상태) + b (입력) ==> 4
3 (상태) + a (입력) ==> 4
3 (상태) + b (입력) ==> 5
4 (상태) + a (입력) ==> 3
4 (상태) + b (입력) ==> 6
5 (상태) + a (입력) ==> 5
5 (상태) + b (입력) ==> 1
6 (상태) + a (입력) ==> 6
6 (상태) + b (입력) ==> 2

*************
DFA에서의 시작 상태 : 1
DFA에서의 종결 상태 : 1 2
DFA에서의 비종결 상태 : 3 4 5 6

------------------------------------------------------
DFA --> REDUCED DFA :

1 (상태) + a (입력) ==> 1
1 (상태) + b (입력) ==> 2
2 (상태) + a (입력) ==> 2
2 (상태) + b (입력) ==> 3
3 (상태) + a (입력) ==> 3
3 (상태) + b (입력) ==> 1

**********************
Reduced DFA에서의 시작 상태 : 1
Reduced DFA에서의 종결 상태 : 1
```

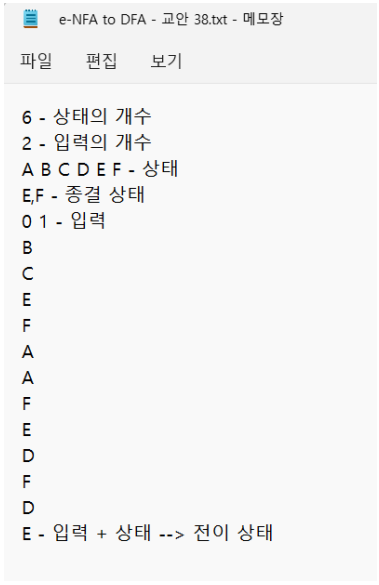
(출력 결과)

δ'	a	b
[AF]	[AF]	[BE]
[BE]	[BE]	[CD]
[CD]	[CD]	[AF]

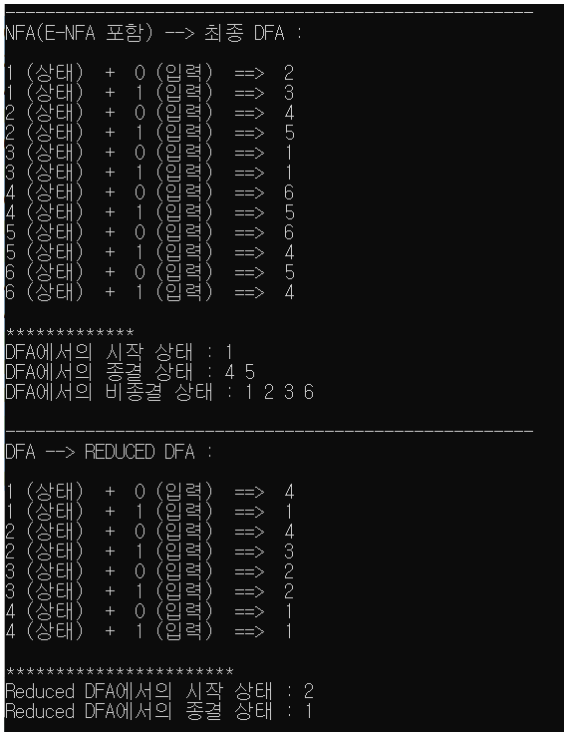
(DFA → Reduced DFA 변환 정답)

E-NFA → DFA : 본래의 입력이 DFA였기 때문에 변환하는 과정이 의미가 없어, 그대로 출력되었다. 출력 결과에서의 상태 1를 a, 상태 2를 b, 상태 3을 c, 상태 4를 d, 상태 5를 e, 상태 6을 f로 두고 보면 DFA에서의 정확한 답을 얻어낼 수 있다.

DFA → Reduced DFA : DFA → Reduced DFA 변환 정답 그림에서의 상태와 출력에서의 상태에 대한 매칭을 AF – 1(시작 & 종결 상태), BE – 2, CD – 3으로 두고 보면 정확한 답을 얻어낼 수 있다.



교안 p.36에서의 DFA이다.



(출력 결과)

δ	0	1
[A]=p	r	q
[C]=q	p	p
[B,D]=r	s	s
[E, F]=s	r	s

(DFA → Reduced DFA 변환 정답)

E-NFA → DFA : 본래의 입력이 DFA였기 때문에 변환하는 과정이 의미가 없어, 그대로 출력되었다. 출력 결과에서의 상태 1를 a, 상태 2를 b, 상태 3을 c, 상태 4를 d, 상태 5를 e, 상태 6을 f로 두고 보면 DFA에서의 정확한 답을 얻어낼 수 있다.

DFA → Reduced DFA : DFA → Reduced DFA 변환 정답 그림에서의 상태와 출력에서의 상태에 대한 매칭을 s – 1(종결 상태), p – 2(시작 상태), q – 3, r – 4로 두고 보면 정확한 답을 얻어낼 수 있다.