Essence of Theory of Computation

서울대학교 컴퓨터공학부 이태영

July 9, 2024

서문

언제 완성될 지는 모르겠으나 일단은 시작해보고자 한다. 이 교재는 기본적으로 박근수 교수님의 오토마타 이론 과목의 강의록을 기반으로 한다.
이 교재는 필자의 '한국어 교재는 한국어 용어만, 영어 교재는 영어 용어만 사용해야 한다'는 이상한 철학에 의해 모든 용어는 한국어 용어를 기반으로 한다. 모든 번역은 이미 사용되고 있는 방식이나 한국수학회 수학용어를 기반으로 번역하였다.
이 과목을 공부하는데 있어서 조언을 하나 하고자 한다. 여러 수학적 노테이션이나 엄밀성이 익숙한 독자들도 있겠으나, 그렇지 못한 독자들도 있을 것이라 예상한다. 당연히 엄밀하면 좋겠지만, 만약 그러한 것이 익숙하지 않다면 엄밀성을 어느정도 배제하고 최대한 직관적인 방식으로 이해하는 것도 결코 나쁜 선택이 아니다. 집합론과 수리 논리학은 당연한 것을 당연하지 않다고 말하는 과목이고, 해석학과 위상수학은 당연한 걸 당연하다고 말했을 때 당연하지 않은 일이 발생하는 과목이다. 그런 과목에서는 당연히 엄밀성이 매우 강하게 요구된다. 하지만 계산 이론은 당연한 것을 당연하다고 말하는 과목이다. 이미 엄밀한 과정은 선대의 수학자들이 토대를 세워놨기 때문에 우린 어느 정도의 엄밀함은 포기하고 그 과정에서 오는 감동만 즐겨도 큰 문제가 없다.
이 책에서의 여러가지 정리의 증명과 예제의 해설은 생략되어 있다. 이를

대하는 방법을 소개한다.

- 정리의 증명이 **느슨한 증명**인 경우: 증명이 너무 길어져서 재미가 없거나, 증명의 엄밀성을 따지는게 크게 중요하지 않은 경우엔 최대한 직관적인 증명으로 적었다.

- 정리의 증명이 자명하거나 당연하다고 되어 있는 경우: 진짜로 자명해서 자명하다고 적었다. 교과서 저자들이 하는 '자명하다'의 남용을 필자는 별로 좋아하지 않는다.

- 정리의 증명이 연습문제로 넘어간 경우: 증명을 설명하기엔 책의 흐름이 너무 끊겨서 독자들이 증명의 일부분을 대신 해주길 원해서 넘긴 것이다.[1]

- 정리의 증명이 생략되어 있는 경우: 너무 어려워서 증명을 생략했지만 결과가 중요해서 적어둔 것이다. 증명하려고 애쓰지 말고 검색하자.

- 예제의 해설이 있는 경우: 해설을 적어두었지만, 이는 독자들이 답을 맞춰보기 위해 적어둔 것일 뿐 그냥 바로 해설을 읽으라는 의미는 아니다. 꼭 직접 고민해보는 시간을 가져보길 바란다.

- 예제의 해설이 생략되어 있는 경우: 독자들이 직접 풀어보길 권하는 문제들이다.

제목은 다들 무엇 무엇의 정수와 같은 방식으로 제목을 작성하길래 필자도 따라해 보았다.
아직 텍 양식이 사실상 기본 양식이다. 좋은 양식이 있으면 얼마든지 추천 바란다.

[1] 조금 어려운 경우도 있다 ㅎㅎ. 구글 검색을 애용하길 바란다.

Contents

CHAPTER 0

들어가며

0.1 이런 걸 왜 함? (진짜 모름)

대학교에서 연구하는 '학문'은 보통 어떤 '문제'를 해결하는 것이다. 예를 들어, 물리학은 수많은 자연 현상을 설명하기 위해 '문제'를 푸는 것이고, 건축공학은 집을 잘 쌓기 위한 '문제'를 해결하는 것이다.
그럼 컴퓨터 과학(Computer Science)은 어떤 학문일까? 여러가지 정의가 있겠지만 필자는 '문제'를 푸는 방법 그 자체에 대해 연구하는 학문이라 생각한다. 그 '문제'를 풀기 위해 우리는 '컴퓨터'라는 계산 기계를 활용하기 때문에 컴퓨터 과학이라 부른다.
이 책에서는 그러한 '문제'들에 대해 컴퓨터로 풀 수 있는지(solvable), 푸는 게 얼마나 복잡한지(intractable)에 대해 다룰 것이다.
이 책을 이해하기 위해서 필요한 선수지식은 거의 없다. 물론 알아두면 좋은 지식으로는 약간의 집합론 지식, 약간의 자료구조와 알고리즘 관련 지식, 약간의 통사론 관련 지식 정도가 있으나 알지 못하더라도 큰 상관은 없다.

0.2 언어

일단 언어를 정의하기 위해 알파벳과 문자열을 정의하자.

정의 0.1. 알파벳과 문자열은 다음과 같이 정의된다.

- **알파벳**(alphabet)은 어떤 글자들의 유한 집합이다. 일반적으로 Σ를 이용해 나타내며, $\{0,1\}, \{a,b,c\}$ 등이 있다. 별 다른 말이 없을 경우 이 책에서 알파벳은 $\{0,1\}$만 사용한다.
- 알파벳 Σ상에서 **문자열**(string)은 Σ 내에 있는 알파벳을 유한개 나열한 것이다. 예를 들어, 01001은 $\Sigma = \{0,1\}$ 상의 문자열이다. 주로 w로 표현한다.
- 문자열 w의 **길이**(length)는 글자들의 개수이고 주로 $\|w\|$나 $|w|$로 나타낸다.
- 길이가 0인 문자열을 ϵ이라 쓰고 **공문자열**이라고 부른다.
- ϵ을 포함해서 알파벳 Σ 상의 모든 문자열의 집합을 Σ^*로 표시한다. 예를 들어, $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, \ldots\}$이다.
- 두 문자열의 **접합**(concatenation) $x \cdot y$는 x뒤에 y를 붙인 것이다. 주로 xy라 쓴다. 예를 들어 $x = 011, y = 110$이면 $xy = 011110$이다.
- 문자열의 **역**(reverse) w^R은 w를 역순으로 나열한 문자열이다. 예를 들어, $w = 011$이면 $w^R = 110$이다.

이를 이용해 언어를 정의하자.

정의 0.2. Σ^*의 부분 집합을 **언어**(language)라 부른다. 주로 L로 표기한다. 언어의 연산들은 다음과 같이 정의된다.

- 언어 L_1, L_2의 접합 $L_1 \cdot L_2$는 다음과 같다.

$$L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}$$

  예를 들어, $L_1 = \{0, 00\}, L_2 = \{1, 11\}$일 때, $L_1 L_2 = \{01, 011, 001, 0011\}$이다. 또한 $L^0 = \{\epsilon\}, L^k = L^{k-1}L(k \geq 1)$이다.

- L^*는 L을 0번 이상 접합하여 만들어지는 모든 문자열의 집합이다. 이를 **클리니 별**(Kleene star)이라고 하고 다음과 같이 정의한다.

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$$

  $L = \{0, 1\}$ 이면 L^*는 0과 1로 이루어진 모든 문자열의 집합이다.

우리가 이 책에서 다룰 **문제**(problem)는 어떤 문자열 w가 언어 L에 속하는지 아닌지를 결정하는 문제이다.

예제 0.3. 문자열 $w = 000111$이 $L = \{0^n 1^n \mid n \geq 0\}$에 속하는지 구하시오.

0.3 표기법

이 책에서 사용할 여러가지 표기법(notation)을 소개한다.

1. 언어 L의 여집합은 $\overline{L}$로 표기한다.
2. 어떤 알파벳 x가 n개가 나열된 문자열은 x^n으로 표기한다.

Part I

오토마타와 언어

CHAPTER 1

정규 언어와 유한 오토마타

이제 계산 이론의 첫걸음이다. 일단 최대한 간단한 모델들부터 알아보자.

1.1 정규 언어

먼저 정규식(regular expression)을 정의하자.

정의 1.1. 알파벳 Σ 상의 정규식과 그것이 표현하는 집합은 다음과 같다.

- $\emptyset$은 정규식이고 공집합을 표현한다.
- ϵ은 정규식이고 $\{\epsilon\}$을 표현한다.
- $a \in \Sigma$는 정규식이고 $\{a\}$를 표현한다.
- r, s가 정규식이고 각각 R, S 집합을 표현한다면 $rs, r+s, r^*$는 각각 $RS, R \cup S, R^*$를 표시하는 정규식이다.

어떤 '조건'들을 만족시키는 언어집합을 표현하기 위해서 우리는 정규식을 사용한다.

정의 1.2. 정규식으로 표현되는 언어를 **정규 언어**(regular language)라 부른다.

예제 1.3. 1이 1개인 문자열의 집합을 표시하는 정규식을 구하라.

해설. 0^*10^*

예제 1.4. 길이가 짝수인 문자열의 집합을 표시하는 정규식을 구하라.

해설. $((0+1)(0+1))^*$

예제 1.5. $\{a, b, c\} \in \Sigma$에 대해 첫 번째 나온 알파벳이 다시 나오지 않는 문자열의 집합을 표시하는 정규식을 구하라.

예제 1.6. 111이 딱 한 번 나타나는 문자열의 집합을 표시하는 정규식을 구하라.

1.2 유한 오토마타

컴퓨터는 CPU, 메모리 등 다양한 요소로 구성된다. 이러한 컴퓨터의 이론적 모델 중에서 먼저 가장 간단한 '유한 오토마타'에 대해 알아보자.

정의 1.7. 결정 유한 오토마타(deterministic finite automata, DFA) M은 다섯 가지 요소로 구성된다.

1. 상태들의 유한 집합 Q
2. 알파벳 Σ
3. 전이함수 $\delta : Q \times \Sigma \to Q$
4. 초기상태 $q_0 \in Q$

5. 최종 상태들의 집합 $F \subseteq Q$

조금 더 구체적으로 들어가보자. 유한 오토마타는 CPU에[2] 해당하는 유한 제어기, 입력장치인 입력 테입으로 구성된다. 입력 테입에는 Σ상의 문자열이 적혀있고, 이걸 유한 제어기가 하나씩 순서대로 읽으면서 유한 제어기의 '상태'가 전이함수에 따라 변하게 된다. '상태'는 '현재 0이 짝수개인 상태', '0 다음 1이 나온 상태' 등이 있을 수 있다.
전이함수를 편리하게 사용하기 위해 전이 함수의 확장 형태인 함수 $\delta^* : Q \times \Sigma^* \to Q$를 다음과 같이 정의하자.

- $\delta^*(q, \epsilon) = q$
- $\forall w \in \Sigma^*, \forall a \in \Sigma, \delta^*(q, wa) = \delta(\delta^*(q, w), a)$

문자열 $w \in \Sigma^*$에 대해 $\delta^*(q_0, w) \in F$면, DFA M이 w를 **받아들인다**(accept)고 한다. M의 언어 $L(M)$은 다음과 같이 정의 가능하다.

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$$

DFA의 **상황**(configuration)은 $(q_1, 0011)$과 같이 현재 상태와 입력 문자열의 읽지 않은 부분으로 결정된다. 한 번의 전이에 의해 DFA M의 상황이 변화하는 과정을 $\vdash_M$로 나타낸다. 중간 전이를 생략하고 싶으면 $\vdash_M^*$로 나타낸다.

$$(q_0, 0011) \vdash_M (q_1, 011)$$
$$(q_0, 0011) \vdash_M^* (q_F, \epsilon)$$

유한 오토마타를 이해하기 쉽도록 우리는 이를 방향 그래프로 나타낸다. 그

[2] 사실 우리가 배우지도 않은 더 상위 개념을 가져왔다! 혹시 컴퓨터를 평생 본 적이 없는 사람이라면 이 설명은 무시해도 좋다.

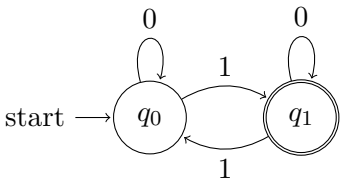


그림 1.1

림 1.1는 1이 홀수개 있는 DFA를 나타낸다. 원이 두 개 그려진 상태는 최종 상태를 의미한다. 나머지는 잘 이해할 것이라 믿는다.

예제 1.8. 010을 부분문자열으로 갖는 문자열을 받아들이는 DFA를 구하라.

해설. 그림 1.2 참조.

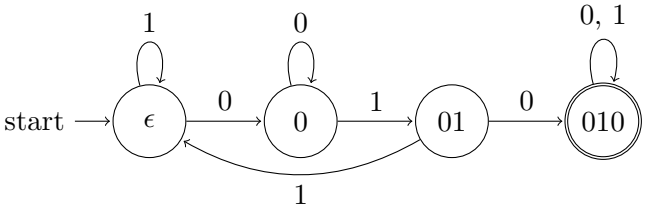


그림 1.2

예제 1.9. 010을 부분문자열으로 갖지 않는 문자열을 받아들이는 DFA를 구하라.

해설. 예제 1.8에서 F와 $Q-F$를 서로 바꿔주면 된다. 이와 같이 DFA에서는 어떤 언어 집합의 여집합을 표현하는 것이 매우 편리하다.[3]

예제 1.10. 000 또는 111을 갖지 않는 문자열을 받아들이는 DFA를 구하라.

[3] 한 번 예제 1.8, 예제 1.9를 정규식으로 표현해보면서 많이 어렵다는 것을 느끼자.

예제 1.11. 000을 부분문자열으로 가지고 111을 부분문자열으로 갖지 않는 문자열을 받아들이는 DFA를 구하라.

어떤 입력을 받아도 빠져 나올 수 없는 죽은 상태(dead state)를 활용하면 된다. 죽은 상태는 모든 입력에 대해 자기 자신으로 돌아오게 만들면 된다.

1.3 비결정론적 유한 오토마타

여기서 다룰 오토마타는 약간 우리의 상식과는 어긋난다. DFA의 경우 결정론적으로 어떤 입력을 받으면 반드시 그 다음 상태로 이동하게 된다. 즉, 다른 말로 하면 항상 '확실'하게 움직인다고 할 수 있다. 그러나 비결정론적 유한 오토마타(nondeterministic finite automata)는 다음 상태가 없거나 하나 이상일 수도 있다. 또한 아무런 입력을 받지 않고서라도 전이할 수 있다.

정의 1.12. 비결정론적 유한 오토마타(nondeterministic finite automata, NFA) M은 다음과 같은 5가지 요소로 구성된다.

1. 상태들의 유한 집합 Q
2. 알파벳 Σ
3. 전이관계 $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$
4. 초기상태 $q_0 \in Q$
5. 최종 상태들의 집합 $F \subseteq Q$

여기서 Δ를 이해하기 쉽게 $\Delta : 2^Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ 인 전이 함수로 생각할 수도 있다.

$$P \subseteq Q, a \in \Sigma \cup \{\epsilon\}, \Delta(P, a) = \bigcup_{q \in P} \Delta(q, a)$$

나머지는 모두 같으나, 전이 관계 Δ가 DFA와 다르다. DFA는 함수이므로 모든 입력값에 대해 다음으로 갈 수 있는 상태가 단 한 개만 존재하지만, NFA는 관계이므로 존재하지 않아도 되고 여러 개의 상태로 '동시에' 전이할 수 있다.
상태 q에 대해, $E(q)$를 q에서 입력을 읽지 않고 전이할 수 있는 모든 상태들의 집합이라고 하자. 이를 확장해서 상태들의 집합 P에 대해

$$E(P) = \bigcup_{q \in P} \Delta(q, a)$$

라고 할 수 있다.
함수 $\Delta^* : Q \times \Sigma^* \to 2^Q$ 는 다음과 같이 정의할 수 있다.

- $\Delta^*(q, \epsilon) = E(q)$
- $\forall w \in \Sigma^*, \forall a \in \Sigma, \Delta^*(q, wa) = E(\Delta(\Delta^*(q, w), a))$

너무 복잡하게 생각하지 말고, 알파벳 대신 문자열을 쭉 따라 갔을 때 갈 수 있는 모든 상태들의 집합으로 전이하는 함수라고 생각하면 된다.
문자열 w에 대해 $\Delta^*(q_0, w)$의 원소들 중에 F의 원소가 하나라도 있으면 M이 w을 받아들인다고 한다. 즉, $L(M)$은 M이 받아들이는 모든 문자열의 집합이다.

예제 1.13. $(010 + 01)^*$ 를 받아들이는 NFA를 만들어라.

해설. 그림 1.3 참조.

그림 1.3을 보면 알겠지만, DFA와는 다르게 어떤 입력은 전이하는 간선이 없으며, 입력을 받지 않고서도 전이가 가능하다. 또한, NFA의 최종 상태는 하나로 만들 수 있다. 모든 $q \in F$에 대해 (q, ϵ, q_F)를 넣어주면 된다.

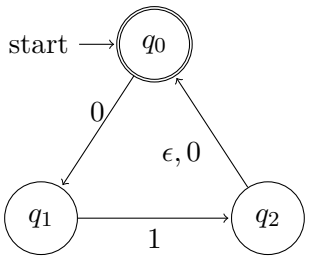


그림 1.3

예제 1.14. 끝에서 세 번째 글자가 1인 문자열을 받아들이는 NFA를 구하라.

NFA를 보다보면 대체 왜 현실에서 만들기도 힘든 비결정론적인 오토마타가 필요한지 의문이 들 것이다. 이는 여러 가지 이유가 있다.

- 비결정성을 이용하면 쉽게 문제를 해결할 수 있는 경우가 많다. 예를 들어, 그래프 내의 최장 경로(longest path)를 찾는 문제를 생각해보자. 어떤 정점에서 어떤 간선으로 가야 최장 경로가 될 수 있는지는 결정론적으로 풀 경우 모든 간선을 대해 확인해봐야 하지만, 비결정론적으로 접근할 경우 모든 간선을 '동시에' 접근하는 것이 가능하므로 쉽게 문제를 해결할 수 있다.[4]

- 우리가 풀어야 하는 문제의 도구는 결정론적이지만 문제의 세계에선 수많은 비결정성이 존재한다. 따라서 우리가 문제의 세계를 이해하기 위해서는 비결정성이라는 개념이 무조건적으로 필요하다.

[4] 멀리 안 가고 예제 2.9를 DFA로 만들려고 하면 좀 까다로울 것이다.

1.4 DFA = NFA

이제 DFA와 NFA가 표현할 수 있는 언어 집합이 같다는 것을 보이자. 일단 DFA는 NFA의 부분 집합이므로 당연하다. 그럼 반대의 경우를 보이자.

정리 1.15. 임의의 NFA에 대해 이와 동등한 DFA가 존재한다.

증명. (느슨한 증명) NFA $N = (Q_N, \Sigma, \Delta, q_0, F_N)$에 대해, 동등한 DFA $D = (Q_D, \Sigma, \Delta, q', F_D)$를 만드려고 한다. 이 증명의 핵심은 NFA에서 전이가 일어나면 다음 상태는 Q_N의 원소들로 동시에 전이되는데, 이때 이 원소들을 하나의 집합으로 취급하면 된다. 즉, DFA D의 상태는 Q_N의 멱집합의 원소이다.
$q' = E(q_0)$으로 잡고, $P \subseteq Q_N, a \in \Sigma$에 대해 $\delta(P, a) = E(\Delta(P, a))$로 잡으면 된다. 이를 의사코드(pseudocode)로 작성하면 다음과 같다.

$Q_D \leftarrow \{E(q_0)\}$
`mark` $E(q_0)$
`while` $\exists$ `marked state` $P \in Q_D$
    `unmark` P
    `for each` $a \in C$
        $R \leftarrow E(\Delta(P, a))$
        `if` R `is not in` Q_D
            `add` R `as marked state to` Q_D
        $\delta(P, a) \leftarrow R$

마지막으로 D의 상태 P가 F_N의 원소를 하나라도 포함하면 P는 F_D의 원소이다. 혹시 코드가 이해가 안된다면 예제를 보면서 따라가보는 것도 좋다.
이제 이렇게 만든 NFA N과 DFA D의 동등성을 확인해야 하는데, $\Delta^*(q_0, w) = \delta(E(q_0), w)$임을 보이면 된다. 이는 귀납법으로 충분히 '할 수 있다'. 또한

NFA의 상태 수는 유한하므로 당연히 Q_N의 멱집합도 유한하다. 따라서 이후 증명은 생략하도록 한다. □

예제 1.16. 예제 1.13의 NFA를 DFA로 만들어라.

해설. 먼저 원래의 NFA를 생각하자. 시작 상태는 q_0이다. q_0에서 0을 입력받으면 q_1로 가고, 1을 입력받으면 죽은 상태로 가게 된다. q_1에서 0을 입력받으면 죽은 상태로 가게 되고, 1을 입력받으면 빈 문자열에 의한 전이도 가능하므로 q_2, q_0으로 동시에 가게 된다. 따라서 DFA를 만들 때에는 $\{q_0, q_2\}$라는 상태를 하나 더 만들어주면 된다. q_0, q_2에서 0을 입력받으면, q_0에서는 q_1로 가고, q_2에서는 q_0, q_1로 동시에 갈 수 있으므로 $\{q_0, q_1\}$이라는 상태를 하나 더 만들어주면 된다. 1을 입력받으면 죽은 상태로 간다. 이를 계속해서 반복하고, F_N의 원소가 하나라도 있는 상태를 최종 상태로 만들어주면 그림 1.4을 얻을 수 있다.

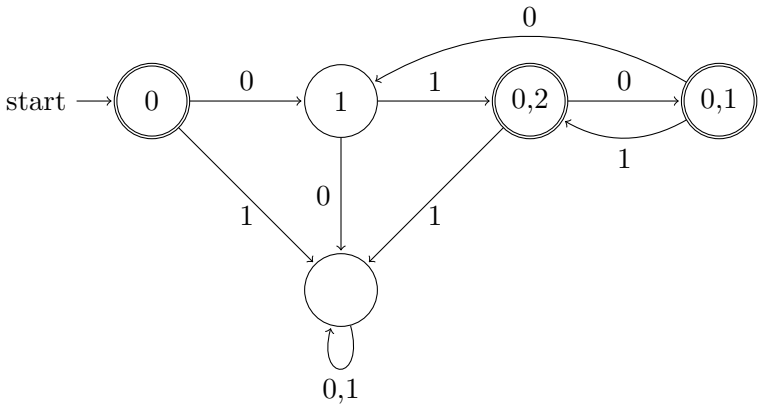


그림 1.4

예제 1.17. 예제 1.14의 NFA를 동등한 DFA로 바꾸어라.

1.5 정규 언어 = DFA = NFA

이제 정규 언어와 DFA, NFA가 나타내는 언어 집합이 같다는 것을 보일 것이다.

정의 1.18. 정규식 r에 대해 $L(r)$을 받아들이는 NFA가 존재한다.

증명. 정의 1.1의 각 요소들을 만족시키는 NFA를 만들면 된다.

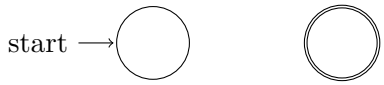


그림 1.5

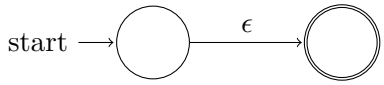


그림 1.6

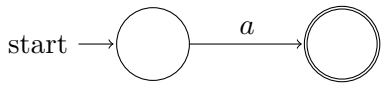


그림 1.7

그림 1.5, 1.6, 1.7이 나타내는 NFA는 각각 $\emptyset, \epsilon, \{a\}$를 의미한다. 또한, 그림

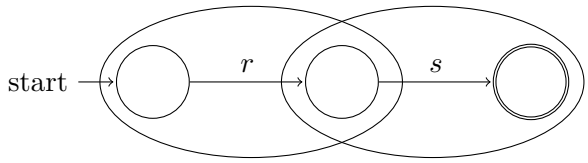


그림 1.8

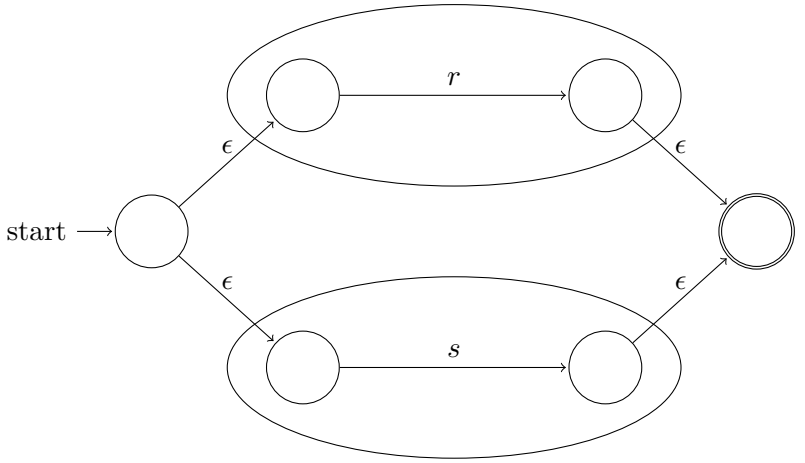


그림 1.9

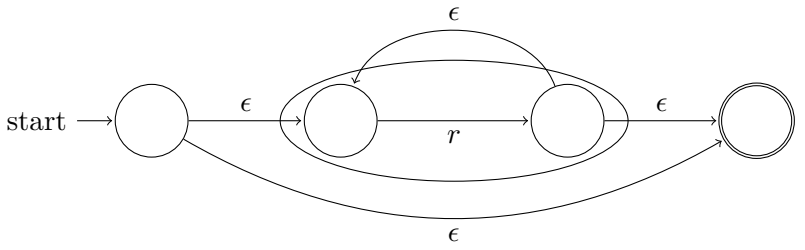


그림 1.10

1.8, 1.9, 1.10는 각각 $rs, r+s, r^*$를 의미한다. □

예제 1.19. 정규식 $(0+11)^*$를 받아들이는 NFA를 구하라.

정리 1.20. DFA M에 대해 $L(M)$을 표시하는 정규식이 존재한다.

증명. (느슨한 증명) DFA의 상태에 대해 각각 번호를 1부터 k까지 부여하고 (시작 상태는 무조건 1번), 경로 내에서 k 번호 이하의 상태만을 지나오면서 i

번에서 j 번으로 가는 모든 문자열의 집합을 $R^k_{i,j}$ 라고 하자. 이는 다음과 같이 재귀적으로 구할 수 있다.

$$R^k_{i,j} = R^{k-1}_{i,j} \cup R^{k-1}_{i,k} \left(R^{k-1}_{kk}\right)^* R^{k-1}_{k,j}$$

전부 접합, $+, *$로 이루어져 있으므로 이제 대충 귀납법 쓰면 구할 수 있다는 느낌이 온다. □

1.6 정규 언어의 특징

정리 1.21. 정규 언어는 (1) 합집합, (2) 접합, (3) 클리니 별$(*)$, (4) 여집합, (5) 교집합에 대해 닫혀있다.

증명. (1) $\sim$ (3)은 정의에 의해 당연하다.
(4) 정규 언어 L을 받아들이는 DFA는 존재하므로, DFA의 최종 상태 F를 $Q - F$로 대체하면 된다.
(5) $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ □

예제 1.22. (소속문제) 정규 언어 L에 문자열 w이 속하는지 결정하라.

해설. L을 표시하는 DFA를 만들고 이를 읽어서 결정하면 된다.[5]

예제 1.23. $L_1 \cup L_2$가 정규 언어이고 L_1이 유한 언어집합이라고 하자. L_2가 정규 언어인지 아닌지 증명하라.

[5] 조금의 스포일러를 하자면 앞으로의 소속 문제는 꽤나 어렵거나 심지어는 풀 수 없는(!) 문제라는 것을 알게 될 것이다.

1.7 정규 언어가 아닌 것들

이제 정규 언어로 모든 언어 집합을 표현할 수 있을 것만 같지만 실상은 그렇지 않다. 다음은 어떤 언어가 정규 언어가 아님을 보이기 위해서 자주 사용하는 정리이다.

정리 1.24. (펌프 정리) L을 무한 정규 언어라고 하자. 이때 다음을 만족하는 어떤 양의 정수 t가 존재한다. 길이가 t 이상인 임의의 문자열 $w \in L$에 대해 $w = xyz$로 표현되고 다음을 만족한다.

1. $|xy| \leq t$
2. $|y| \geq 1$
3. 모든 $i \geq 0$에 대해 $xy^iz \in L$이다.

마치 y를 펌프처럼 늘릴 수 있다고 해서 펌프정리(pumping lemma)라고 한다.

증명. 정규 언어 L을 받아들이는 DFA M에 대해 M의 상태의 개수를 t라 하자. $w = a_1 \dots a_n$ $(a_i \in \Sigma, n \geq t)$에 대해 $\delta^*(q_0, a_1 \dots a_i) = q_i$라고 하자. 이때 M은 t 개의 상태만을 가지므로, $q_j = q_k$ $(0 \leq j < k \leq t)$가 존재한다. 즉, $y = a_{j+1} \dots a_k$라고 하면 $xy^iz \in L$이므로 펌프 정리를 증명할 수 있다. □

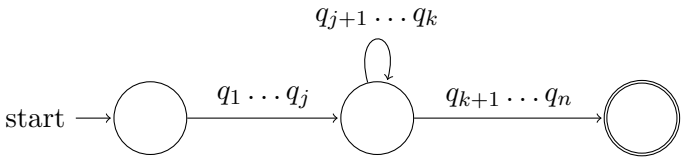


그림 1.11

펌프 정리의 역은 성립하지 않는다.

예제 1.25. $L = \{0^n 1^n \mid n \geq 0\}$은 정규 언어가 아님을 증명하라.

해설. 먼저 L을 정규 언어라고 하자. 그럼 임의의 t에 대해, $w = 0^t 1^t$라고 하자. 그럼 $w = xyz$에 대해 $y = 0^k$ $(k \leq t)$이므로, $i = 2$일 때, $xy^i z = 0^{t+k} 1^t \notin L$이므로 펌프 정리에 의해 L은 정규 언어가 아니다.

예제 1.26. $L = \{a^{n^2} \mid n \geq 0\}$은 정규 언어가 아님을 증명하라.

예제 1.27. $L = \{0^n 1^m \mid n \neq m\}$은 정규 언어가 아님을 증명하라.

예제 1.28. $L = \{ab^n c^n \mid n \geq 0\} \cup \{a^k w \mid k \neq 1, w$는 a로 시작하지 않는 문자열$\}$이 펌프 정리가 성립하지만 정규 언어가 아님을 증명하라.

정규 언어인지 아닌지 알아내는 직관적인 방법은 '이 언어를 유한한 메모리로 표현하는 것이 가능한가?'를 생각해보면 된다. 예를 들어, 예제 1.25의 경우 0의 개수를 저장해 둔 뒤 1의 개수와 비교해야 하는데, 0의 개수로 가능한 가짓수는 무한하다. 따라서 유한 개의 상태만을 가진 DFA로는 구현해내는 것이 불가능하므로 정규 언어가 아님을 알 수 있다.

1.8 연습문제

연습문제 1.1. 언어 L_n에 대해 $L_n = \{x \mid x$는 이진수이고 n의 배수이다.$\}$라고 하자. 모든 $n \geq 1$에 대해 L_n은 정규 언어임을 보여라.

연습문제 1.2. Let the **rotational closure** of language A be $RC(A) = \{yx \mid xy \in A\}$.

1. Show that for any language A, we have $RC(A) = RC(RC(A))$.

2. Show that if language A is regular then $RC(A)$ is regular.

연습문제 1.3. Consider the Language $L = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}$.

1. Show that L satisfies pumping lemma.

2. Is L regular? Verify your answer.

CHAPTER 2

문법과 오토마타

2.1 문법

정의 2.1. 문법(grammar) G는 (V, Σ, S, P)로 구성된다.

1. V는 변수들의 유한 집합
2. Σ는 알파벳
3. $S \in V$는 시작변수
4. P는 생성규칙들의 유한 집합이다.

여기서 생성규칙이란 $x \in (V \cup \Sigma)^* V (V \cup \Sigma)^*, y \in (V \cup \Sigma)^*$에 대해 $x \to y$의 형태를 가진다. 이 장에서는 약간 문법을 제한한 형태만을 다룰 것이다.

정의 2.2. 문법 중 **문맥무관 문법**(context-free grammar) G는 $A \in V, x \in (V \cup \Sigma)^*$에 대해 다음과 같은 생성규칙만을 가진다.

$$A \to x$$

이때 $A \to x, A \to y$를 줄여서 $A \to x \mid y$라고 쓴다.

일반적으로 $u, v \in (V \cup \Sigma)^*, A \in V$에 대해 uAv에 $A \to w$를 적용하면 uwv를 얻는데, 이 과정을 **유도**(derivation)라 부르고 $uAv \Rightarrow uwv$로 표시한다. 여러 유도를 생략해서 사용할 경우 $A \overset{*}{\Rightarrow} w$로 쓴다.
문맥무관 문법이 생성하는 언어를 **문맥무관 언어**(context-free language)라고 한다. 문법 G가 생성하는 언어 $L(G)$는 다음과 같이 정의된다.

$$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}$$

문맥무관 언어의 예시로는 보통의 프로그래밍 언어(Fortran, C 등)가 있다.

예제 2.3. $L = \{0^n 1^n \mid n \geq 0\}$을 만드는 문맥무관 문법을 구하라.

해설.

$$S \to 0S1 \mid \epsilon$$

예를 들어 0011을 만드는 유도 과정은 다음과 같다.

$$\begin{aligned} S &\Rightarrow 0S1 \\ &\Rightarrow 00S11 \\ &\Rightarrow 0011 \end{aligned}$$

예제 2.4. '()(())'와 같이 적절히 닫혀있는 괄호를 만드는 문맥무관 문법을 구하라.

해설.

$$S \to (S) \mid SS \mid \epsilon$$

예제 2.5. $L = \{w \mid w$의 0과 1의 개수가 같다.$\}$를 만드는 문맥무관 문법을 구하라.

해설.

$$S \to 0S1 \mid 1S0 \mid SS \mid \epsilon$$

예제 2.6. $L = \{0^n 1^m \mid n \leq m \leq 2n\}$를 만드는 문맥무관 문법을 구하라.

예제 2.7. $L = \{0^n 1^m \mid n \neq m\}$를 만드는 문맥무관 문법을 구하라. (힌트: $n > m$일 때와 $n < m$일 때를 구분하여 생각하면 된다.)

예제 2.8. $L = \{a^m b^n c^u d^v \mid m + n = u + v\}$를 만드는 문맥무관 문법을 구하라.

예제 2.9. 3+5*7와 같이 숫자, $+, *$로 구성된 수식을 만드는 문맥무관 문법을 구하라. (숫자는 id로 표시)

이처럼 문맥무관 문법은 정규식이 표현하지 못하는 언어집합을 표현할 수 있다. 이제 문맥무관 문법이 정규식보다 표현할 수 있는 언어집합이 더 넓다는 것을 알아보자.

2.2 정규문법

정의 2.10. 문법 $G = (V, \Sigma, S, P)$에 대해 다음과 같은 생성규칙만을 가지면 문법 G는 **정규문법**이다. $A, B \in V, w \in \Sigma^*$에 대해

$$A \to wB$$
$$A \to w$$

이름이 정규문법인 이유는 위와 같은 문법이 생성하는 언어는 정규 언어 종류이기 때문이다.

정리 2.11. 정규문법이 생성하는 언어는 정규 언어이다.

증명. (느슨한 증명) 정규문법이 유도되는 과정을 보자.

$$\begin{aligned} A &\Rightarrow w_1 B \\ &\Rightarrow w_1 w_2 C \\ &\Rightarrow w_1 w_2 w_3 D \\ &\vdots \end{aligned}$$

입력 $w_1 \to w_2 \to w_3 \ldots$을 받으면 상태가 $A \to B \to C \to \ldots$로 변하는 NFA를 만들 수 있을 거 같은 느낌적 느낌이 온다. 넘어가자. □

정리 2.12. 정규 언어 L에 대해 $L = L(G)$인 정규문법 G가 존재한다.

증명. (느슨한 증명) 정규 언어 L을 표현하는 DFA D의 전이함수에 있는 전이 $((p, a), q)$를 흉내내서 문법 G에 생성규칙 $p \to aq$를 추가해주면 된다. 그리고 마지막으로 $q \in F$에 대해서만 $q \to \epsilon$을 추가해주면 된다. □

예제 2.13. 예제 1.13의 NFA와 동등한 정규문법을 구하라.

해설.

$$\begin{aligned} q_0 &\to 0q_1 \mid \epsilon \\ q_1 &\to 1q_2 \\ q_2 &\to 0q_0 \mid q_0 \end{aligned}$$

예제 2.14. 다음과 같은 정규문법과 동등한 NFA를 구하라.

$$S \to aA$$
$$A \to abS \mid a$$

해설. 그림 2.1 참조.

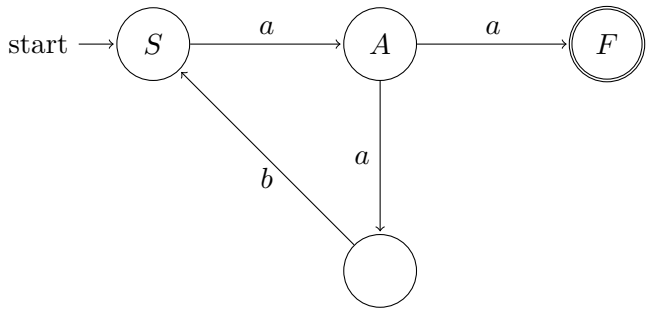


그림 2.1

문맥무관 문법은 정규문법을 통해 정규 언어를 표현할 수 있지만 정규 언어의 경우 $\{0^n1^n\}$와 같은 언어집합을 표현하지 못한다. 따라서 정규 언어는 문맥무관 언어의 진부분 집합임을 알 수 있다.

2.3 파스 트리

어떤 문장(문자열)의 구조를 잘 파악하기 위해 파스 트리를 정의하자.

정의 2.15. 파스 트리(parse tree)는 문맥무관 문법 G에 대해 다음과 같은 성질을 만족시키는 트리이다.

1. 루트 정점은 S이다.

2. 내부 정점은 V의 원소이고, A의 자식 정점이 $X_1, \ldots, X_k$ 이면 $A \to X_1 \ldots X_k$와 같은 생성규칙은 P에 있다.

3. 단말 정점은 $\Sigma \cup \{\epsilon\}$의 원소이고, 단말 정점이 ϵ이면 이 정점은 부모 정점의 유일한 자식 정점여야 한다.

예제 2.16. 예제 2.3와 같은 문맥무관 문법에 대해 다음과 같은 유도과정을 나타내는 파스 트리를 그려라.

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$$

해설. 그림 2.2 참조.

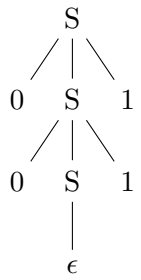


그림 2.2

예제 2.17. 예제 2.4와 같은 문맥무관 문법에 대해 다음과 같은 유도과정을 나타내는 파스 트리를 그려라.

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow (S)() \Rightarrow ()()$$

해설. 그림 2.3 참조.

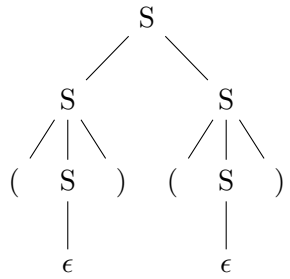


그림 2.3

유도과정에서 항상 맨 왼쪽에 있는 변수에 생성규칙을 적용하는 것을 **좌측 유도**(leftmost derivation)라 부른다. 한 개의 파스 트리에 대해서는 순서에 따라 여러가지 유도과정이 가능하다. 그러나 좌측 유도는 당연히 오직 한 가지만 존재한다. 또한 당연하게도 우측 유도도 존재한다. 단지 이 책에서는 좌측 유도를 기준으로 할 것이다.

예제 2.18. 예제 2.17에서의 유도과정을 좌측유도로 표현하고 이 파스 트리가 그림 2.3와 같다는 것을 보여라.

정의 2.19. 문맥무관 문법 G에 대해 어떤 $w \in L(G)$에 대해 2 개 이상의 파스 트리를 가지면 우리는 문법 G가 **애매하다**(ambiguous)라 한다.

예제 2.20. 다음 문법 G를 고려하자.

$$E \to E + E$$
$$E \to E * E$$
$$E \to \text{id}$$

이는 예제 2.9와 같은 문법인데, 3+5*7에 대한 파스 트리를 2개 이상 그려서 이 문법이 애매하다는 것을 보이고, 이를 애매하지 않은 문법으로 고쳐라.

예제 2.21. 프로그래밍 언어에서 if-else문을 생성하는 문법 G를 고려하자.

$$E \to \text{if } C \text{ then } S \text{ else } S$$
$$S \to \text{if } C \text{ then } S$$
$$S \to x \mid y$$
$$C \to a \mid b$$

if a then if b then x else y의 파스 트리를 2개 이상 그려서 이 문법이 애매함을 보여라.

예제 2.21의 경우 애매하지 않게 문법을 수정하는 것도 가능하나, 일반적으로 문법은 그대로 둔 채 'else는 가장 가까운 if에 붙는다'는 의미(semantics)를 부여하여 애매함을 없앨 수 있다.
자연 언어는 문맥무관 언어는 아니지만 문맥무관 언어처럼 해석한 뒤, 의미를 부여해서 애매성을 해소한다. 예를 들어, 'John said that Mary talks a lot to Tom.'과 같은 영어 문장을 생각해보자. 이 문장은 (1) 'John이 Mary가 많이 이야기한다는 것을 Tom에게 말했다'라는 뜻일 수도 있고, (2) 'John이 Mary가 Tom에게 많이 이야기한다는 것을 말했다.'라는 뜻일 수도 있다. 이에 대한 트리를 그려보면 다음과 같다.[6]
그림 2.4의 경우 1)을 의미하고, 그림 2.5의 경우 2)를 의미한다. 이렇게 단어 자체가 중의적인 것이 아니라 통사적인 구조가 중의적인 경우를 **통사적 중의성**(structual ambiguaty)라고 한다. 이러한 통사적 중의성은 문장의 맥락, 즉 의미(semantics)를 통해 해소될 수 있다.

정의 2.22. 문맥무관 언어 L을 생성하는 애매하지 않은 문법 G가 존재하면, L은 **애매하지 않다**라고 한다. L을 생성하는 모든 문법 G가 애매하면, L은

[6] 다음 트리는 아주 매우 많이 간략화되어 그려졌다.

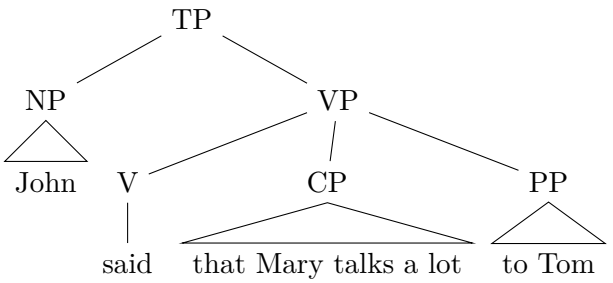


그림 2.4

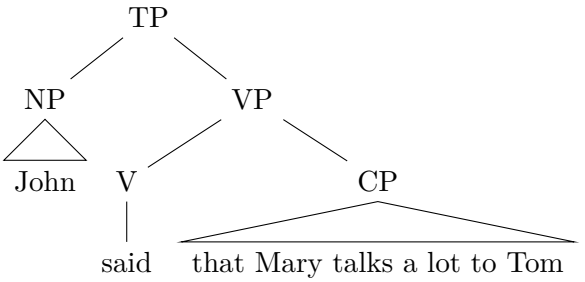


그림 2.5

본질적으로 애매하다(inherently ambiguous)고 한다.

예제 2.23. 언어 $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$ 는 본질적으로 애매함을 보여라.

해설. 이를 증명하는 것은 매우 까다롭다. 따라서 이 교재에서는 그 증명을 생략하고, 직관적으로 생각해보자. 예를 들어, $w = aabbccdd$ 에 대해 앞의 언어 집합에 해당하는 문법으로 만드는 파스 트리와 뒤의 언어 집합에 해당하는 문법으로 만드는 파스 트리는 다를 수밖에 없다. 따라서 L 은 본질적으로 애매하다.

2.4 표준형

여기서부터 다루는 문맥무관 언어 L은 ϵ을 포함하지 않는다고 가정하자.

정의 2.24. (촘스키 표준형) 문맥무관 문법 $G = (V, \Sigma, S, P)$의 모든 생성규칙이 $A, B, C \in V, a \in \Sigma$에 대해

$$A \to BC$$
$$A \to a$$

형태로만 구성되면 G를 촘스키 표준형(Chomsky Normal Form)이라고 부른다.

정의 2.25. (그레이바흐 표준형) 문맥무관 문법 $G = (V, \Sigma, S, P)$의 모든 생성규칙이 $A \in V, a \in \Sigma, x \in V^*$에 대해

$$A \to ax$$

형태로만 구성되면, G를 그레이바흐 표준형(Greibach Normal Form)이라고 부른다.

정리 2.26. 임의의 문맥무관 언어를 생성하는 촘스키 표준형이 항상 존재한다.

증명. 임의의 문맥무관 문법 $G = (V, \Sigma, S, P)$를 생각하자. 우리는 이 문법을 조금 변형한 $G = (V', \Sigma, S, P')$를 만들 것이다.

1. $A \to \epsilon$ 형태의 규칙 없애기
   먼저 $A \overset{*}{\Rightarrow} \epsilon$이면 A를 없어질 수 있는(nullable) 변수라고 하자. $B \to C_1 \dots C_k$에 대해 없어질 수 있는 변수 C_i와 ϵ을 번갈아 넣어 만들어지는

모든 생성규칙을 새로운 생성규칙 P_1에 넣어준다. 단, $B \to \epsilon$와 같은 규칙이 만들어지는 경우 넣지 않는다.

2. $A \to B$ 형태의 규칙 없애기
   $A \overset{*}{\Rightarrow} B$를 만족하는 A, B를 단일쌍이라 하자. P_1에서 $A \to B$와 같은 단위 생성규칙을 제외한 모든 규칙을 P_2에 넣는다. 그 후 단일쌍인 A, B에 대해 $B \to x$가 단위 생성규칙이 아니면 $A \to x$를 P_2에 넣는다.

3. $A \to BC, A \to a$ 형태로 바꿔주기

   (a) $A \to a$인 생성규칙은 이미 촘스키 표준형이므로 P'에 넣는다.

   (b) $r \geq 2$인 각 생성규칙 $A \to x_1 \dots x_k$에서 x_i가 알파벳 a면 새로운 변수 C_a를 V'에 만들고, $C_a \to a$를 P'에 추가하고, x_i를 C_a로 대체한다.

   (c) 그럼 이제 우변에 변수들만 있으므로 이제 두 개로 줄이기만 하면 된다. $A \to B_1 \dots B_k$에 대해 새로운 변수 $D_1 \dots D_{k-2}$를 V'에 도입하고

$$
\begin{aligned}
A &\to B_1 D_1 \\
A &\to B_2 D_2 \\
&\vdots \\
D_{k-3} &\to B_{k-2} D_{k-2} \\
D_{k-2} &\to B_{k-1} B_k
\end{aligned}
$$

   로 만들어주면 된다.

4. 이렇게 만들어진 $G' = (V', \Sigma, S, P')$은 원래 G와 동등하다.

□

예제 2.27. 다음 문맥무관 언어에 대해 동등한 촘스키 표준형을 구하라.

$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \epsilon$$

해설.

$$S \to AA_1 \mid UB \mid a \mid SA \mid AS$$
$$A \to b \mid AA_1 \mid UB \mid a \mid SA \mid AS$$
$$A_1 \to SA$$
$$U \to a$$
$$B \to b$$

촘스키 표준형으로 문법을 바꾸면 매우 비직관적으로 변하게 되는데, 그럼에도 불구하고 촘스키 표준형이 필요한 이유는 무엇일까?
첫 번째로, 촘스키 표준형으로 문법을 바꾸면 파싱을 하는데 있어서 문법이 제한되므로 애매성을 줄여준다. 촘스키 표준형으로 바꾸면 어떤 문자열 w가 문맥무관 언어 $L(G)$에 속하는지 결정하는 문제를 CYK 알고리즘을 통해 $O(n^3)$에 쉽게 해결하는 것이 가능하다.[7] [8]
두 번째로, 통사론에서 문장의 트리 구조를 그릴 때 그림 2.6과 같이 주로 두갈래 구조(자식 정점이 두개 뿐인 트리)를 그리게 되는데 이러한 생성 규칙은 촘스키 표준형의 일종이라고 할 수 있다.

[7] 본래라면 지수 시간이 걸린다.
[8] 나중에 정의 2.46에서 이게 무엇인지 다루니까 이해가 되지 않는다면 조금만 기다리자.

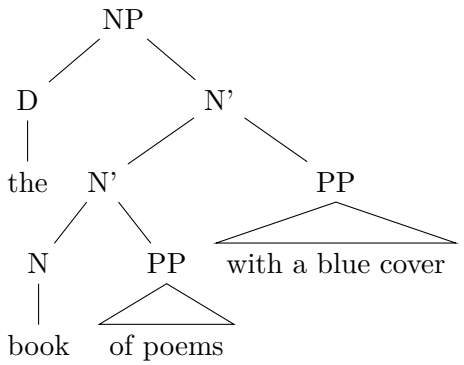


그림 2.6

정리 2.28. 임의의 문맥무관 언어를 생성하는 그레이바흐 표준형이 항상 존재한다.

증명. (느슨한 증명) 촘스키 표준형과 비슷한 형태로 새로운 문법을 만들면 된다. 크게 중요하지는 않으므로 생략한다. □

그레이바흐 표준형의 경우, 프로그래밍 언어 등을 파싱(parsing) 할[9] 때 시간 복잡도를 아낄 수 있다. 대부분의 프로그래밍 언어는 어떤 구조의 첫 번째로 그 부분이 어떤 구조인지 명시한다. 예를 들어, C 언어에서는 if, for, while 등을 통해 이 다음 어떤 구조를 가져야하는지 명시하는데, 이는 기본적으로 그레이바흐 표준형으로 파싱하기 위해서 이러한 방식을 사용한다.

2.5 내리누름 오토마타

정의 2.29. 내리누름 오토마타(Pushdown Automata, PA) M은 $(Q, \Sigma, \Gamma, \Delta, q_0, F)$로 구성된다.

[9] 파스 트리를 만드는 거라고 생각해도 무방하다.

1. Q는 상태들의 유한 집합
2. Σ는 입력 알파벳
3. Γ는 스택 알파벳(시작 글자인 #을 포함한다.)
4. Δ는 $Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times (Q \times \Gamma^*)$의 부분 집합인 전이 관계
5. $q_0 \in Q$는 초기 상태
6. $F \subseteq Q$는 최종 상태들의 집합

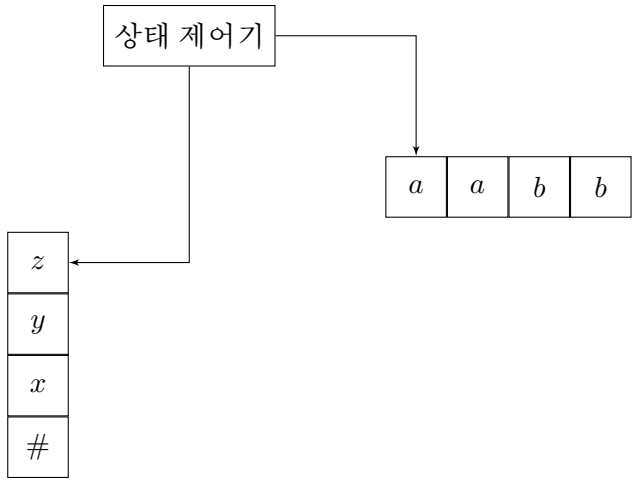


그림 2.7: 내리누름 오토마타

PA는 Δ가 전이 '관계'이므로 기본적으로 비결정론적이다. 비결정론적이므로 NFA와 마찬가지로 최종 상태는 하나라고 가정할 수 있다.
$((q, a, X), (p, Y))$에 대해, q는 현재 상태, a는 입력 알파벳, X는 현재 스택의 상단(top)에 있는 글자이다. X가 ϵ일 때는 스택의 상단을 읽지 않는다는 뜻이

다. 그리고 p는 그 다음 상태, Y는 X를 대체하는 글자이다. $((q, a, \epsilon), (p, A))$의 경우, A를 스택에 넣는 삽입(push) 연산이고, $((q, a, A), (p, \epsilon))$은 스택에서 A를 빼는 삭제(pop) 연산이다.

PA는 시작할 때, 스택의 바닥을 의미하는 #만을 가지고 있다. #을 빼는 연산은 고려하지 않는다.

PA의 상황(configuration)은 현재 상태, 아직 읽지 않은 입력 문자열, 현재 스택의 내용에 의해 결정된다. 어떤 한 번의 전이에 의해 PA M의 상황이 변화하는 과정을 $\vdash_M$로 표현할 수 있다. 중간 과정을 생략하고 싶으면 $\vdash_M^*$와 같이 쓸 수도 있다.

어떤 입력 문자열 w를 PA가 읽었을 때의 상태가 최종 상태이면 우리는 PA가 w를 받아들인다라고 한다. PA M이 받아들이는 문자열의 집합을 $L(G)$라고 표시하며 이는 다음과 같다.

$$L(M) = \{w \in \Sigma^* \mid \exists p \in F, \exists u \in \Gamma^*, (q_0, w, \#) \vdash_M^* (p, \epsilon, u)\}$$

PA를 간편하게 표현하기 위해 방향 그래프의 형태로 표시한다. $a, A/A'$에 대해 a는 입력 알파벳, A는 스택의 상단 알파벳, A'는 대체할 스택 알파벳이다.

예제 2.30. $\{0^n 1 0^n \mid n \geq 0\}$을 받아들이는 PA를 구하라.

해설. 그림 2.8 참조.

예제 2.31. $\{0^n 1^n \mid n \geq 0\}$을 받아들이는 PA를 구하라.

예제 2.32. $\{0^n 1^m \mid n \leq m \leq 2n\}$을 받아들이는 PA를 구하라.

예제 2.33. 0과 1의 개수가 같은 문자열을 받아들이는 PA를 구하라.

예제 2.34. $\{vw \mid v, w \in \{0,1\}^*, w \neq v^R, |v| = |w|\}$를 받아들이는 PA를 구하라.

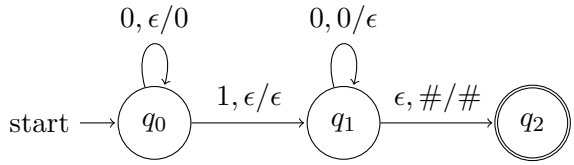


그림 2.8

2.6 PA = CFG

내리누름 오토마타는 뭔가 애매하게 제한적이라 굉장히 부자연스럽다. 이러한 이상한 오토마타를 만든 이유는 무엇일까? 이는 무려 내리누름 오토마타가 표현할 수 있는 언어집합과 문맥무관 문법이 표현할 수 있는 언어집합이 같기 때문이다.

정리 2.35. 임의의 문맥무관 언어 L에 대해 $L = L(M)$인 PA M이 존재한다.

증명. 문맥무관 문법 G에 대해 이를 흉내내는 PA를 만들면 된다. 그 PA는 다음과 같다.

$$(\{p, q, r\}, \Sigma, V \cup \Sigma \cup \{\#\}, \Delta, p, \{r\})$$

1. $((p, \epsilon, \#), (q, S\#)) \in \Delta$
2. 모든 $A \to x$에 대해 $((q, \epsilon, A), (q, x)) \in \Delta$
3. 모든 $a \in \Sigma$에 대해 $((q, a, a), (q, \epsilon)) \in \Delta$
4. $((q, \epsilon, \#), (r, \#)) \in \Delta$

□

예제 2.36. 다음 문맥무관 문법과 동등한 PA를 구하라.

$$S \to 0S1 \mid 1S0 \mid SS \mid \epsilon$$

해설. 그림 2.9 참조.

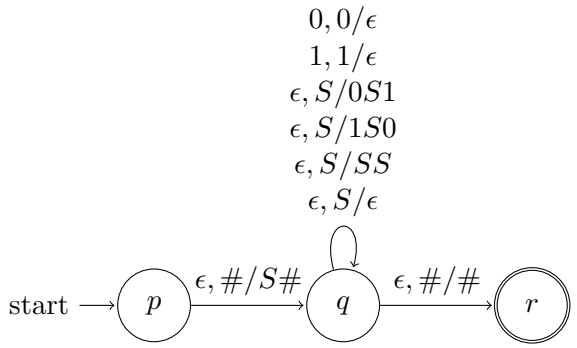


그림 2.9

정리 2.37. 임의의 PA M에 대하여, $L(M)$을 표현하는 문맥무관 문법 G가 존재한다.

증명. 먼저 일반성을 잃지 않고, $M = (Q, \Sigma, \Gamma, \Delta, q_0, F)$에 대해 다음과 같은 성질을 만족한다고 가정하자. 여기서 $a, b \in \Sigma \cup \{\epsilon\}$, $A \in \Gamma$이다.

- M은 최종상태가 q_f 한 개만 있고, 끝날 때에 스택에 #만 남기고 q_f로 간다.
- M의 전이는 $((p, a, \epsilon), (q, A))$(삽입 연산) 또는 $((p, a, A), (q, \epsilon))$(삭제 연산)만 갖는다.[10]

[10] 왜 그런가?

M과 동등한 문맥무관 무법 G는 다음과 같이 만들 수 있다.

1. 모든 $p, q \in Q$에 대해 $\langle pq \rangle \in V$
2. $S = \langle q_0 q_f \rangle$
3. 모든 $((p, a, \epsilon), (r, A)), ((s, b, A), (q, \epsilon)) \in \Delta$에 대해 $\langle pq \rangle \to a\langle rs \rangle b$를 만든다.
4. 모든 $p, q, r \in Q$에 대해 $\langle pq \rangle \to \langle pr \rangle \langle rq \rangle$를 만든다.
5. 모든 $q \in Q$에 대해 $\langle qq \rangle \to \epsilon$을 만든다.

이제 다음을 귀납법으로 보이자.

$$(p, x, \epsilon) \vdash_M^* (q, \epsilon, \epsilon) \Leftrightarrow \langle pq \rangle \stackrel{*}{\Rightarrow} x$$

($\Rightarrow$) 전이 횟수가 0인 경우에는 $p = q, x = \epsilon$이고, 5번 규칙에 의해 $\langle qq \rangle \to \epsilon$이므로 성립한다. 전이 횟수가 i 미만일 때 위 명제가 성립한다 가정하자.
전이 횟수가 $i \geq 1$일 때 스택에서 다음 둘 중 하나가 발생한다.

1. 처음에 삽입된 글자가 마지막에 삭제되는 경우: 이 경우를 $\langle pq \rangle \to a\langle rs \rangle b$로 흉내내려고 한다. 여기에서 a와 r은 처음 전이 때 읽은 글자와 다음 상태이고, b와 s는 마지막 전이 때 읽은 글자와 이전 상태이다.
2. 처음에 삽입된 글자가 중간에 삭제되는 경우: 이 경우를 $\langle pq \rangle \to \langle pr \rangle \langle rq \rangle$로 흉내내려고 한다. 여기에서 r은 처음에 삽입된 글자가 삭제되었을 때의 상태이다.

먼저 1 번이 발생하는 경우를 생각하자. A를 처음에 스택에 삽입된 글자, a와 r은 처음 전이 때 읽은 글자와 다음 상태, b와 s는 마지막 전이 때 읽은

글자와 이전 상태라고 하자. 즉 처음 전이가 $((p, a, \epsilon), (r, A))$, 마지막 전이가 $((s, b, A), (q, \epsilon))$ 이다. 이때 G는 $\langle pq\rangle \to a\langle rs\rangle b$를 갖는다. $x = ayb$라고 하자. 귀납법 가정에 의해 $(r, y, \epsilon) \vdash_M^* (s, \epsilon, \epsilon)$ 이므로 $\langle rs\rangle \overset{*}{\Rightarrow} y$ 이다. 따라서 $\langle pq\rangle \Rightarrow a\langle rs\rangle b \overset{*}{\Rightarrow} x$ 이 성립한다.
2 번이 발생하는 경우를 생각하자. r을 처음에 삽입된 글자가 삭제되었을 때의 상태라고 하고, y와 z를 각각 r 이전과 이후에 읽은 문자열이라고 하자. G는 $\langle pq\rangle \to \langle pr\rangle\langle rq\rangle$를 가지므로 귀납법 가정에 의해 $\langle pr\rangle \overset{*}{\Rightarrow} y$ 이고 $\langle rq\rangle \overset{*}{\Rightarrow} x$ 이다. 따라서 $\langle pq\rangle \overset{*}{\Rightarrow} \langle pr\rangle\langle rq\rangle \overset{*}{\Rightarrow} x$ 이다.
위와 비슷하게 ($\Leftarrow$)의 경우도 증명할 수 있다. 따라서 $w \in L(G)$ 인 경우에만 $w \in L(M)$ 이다. □

예제 2.38. 예제 2.30의 PA와 동등한 문맥무관 문법을 구하라.

2.7 문맥무관 언어의 성질

이제 문맥무관 문법으로 모든 언어 집합을 표현할 수 있을것만 같지만, 이는 그렇지 않다. 유한 오토마타 때와 비슷하게 우리는 펌프 정리를 이용하여 어떤 언어 집합이 문맥무관 언어가 아님을 보일 것이다.

정리 2.39. (펌프 정리) 문맥무관 언어 L에 대해 다음을 만족하는 양의 정수 t가 존재한다. 길이가 t 이상인 임의의 문자열 $w = uvxyz \in L$는

1. $|vxy| \le t$
2. $|vy| \ge 1$
3. 모든 i에 대해 $uv^ixy^iz \in L$

을 만족한다.

증명. 문맥무관 언어 L을 생성하는 촘스키 표준형을 G라고 하자. 문자열 $w \in L$의 파스 트리를 만들 때 G가 촘스키 표준형이므로 파스 트리의 높이가 i이면 $|w| \leq 2^{i-1}$이다. 이는 촘스키 표준형의 경우 파스 트리에서 자식 정점을 두 개 이하로만 가질 수 있기 때문이다.

$|V| = k$라고 하자. 그럼 $t = 2^k$로 잡고, $|w| \geq t$이면, 파스 트리의 높이는 $k+1$ 이상이다. 루트에서 가장 긴 경로를 P라고 하자. 그 경로 내에 $k+1$개 이상의 변수가 존재하는데, 변수의 가짓수는 k이므로 중복되는 변수가 하나 이상 존재한다. P를 맨 아래에서부터 올라갔을 때 가장 처음으로 반복되는 변수를 A라고 하자. 그 중 위에 있는 변수를 A_1, 아래에 있는 변수를 A_2라고 하자.

$A_1 \stackrel{*}{\Rightarrow} uxy$에 대해 A_1까지의 높이는 $k+1$ 이하이므로, $|vxy| \leq t$이다. 또한 $A_1 \to BC$이므로 $|vy| \geq 1$이어야 한다. 또한 $A_1 \stackrel{*}{\Rightarrow} vA_2y$이므로 이를 바꿔 말하면 $A \stackrel{*}{\Rightarrow} vAy$임을 알 수 있다. 따라서 모든 i에 대해 $uv^ixy^iz \in L$임을 알 수 있다.

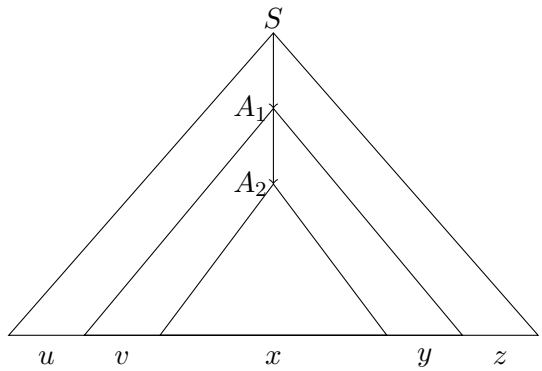


그림 2.10

예제 2.40. $L = \{a^nb^nc^n \mid n \geq 0\}$이 문맥무관 언어가 아님을 보여라.

해설. $w = a^t b^t c^t$ 라고 하자. $|vxy| \leq t$ 이므로 vy는 a, b, c 모두를 포함할 수 없다. 따라서 $uv^2xy^2z \notin L$ 이므로 문맥무관 언어가 아니다.

예제 2.41. $L = \{ww \mid w \in \{0,1\}^*\}$ 이 문맥무관 언어가 아님을 보여라.

예제 2.42. $L = \{a^n \mid n$은 소수$\}$ 이 문맥무관 언어가 아님을 보여라.

직관적으로 어떤 언어가 문맥무관 언어인지 아닌지를 판단하기 위해서는 내리누름 오토마타의 한계가 무엇인지를 생각해보면 된다. 먼저, PA는 DFA와는 달리 저장 공간이 무한하다. 하지만 저장된 내용을 한 번 읽으면 그 내용을 무조건 잊어야 한다. 또한, 그 내용을 역순으로밖에 볼 수가 없다. 따라서 예제 2.40이나 2.41이 문맥무관 언어가 아닌 것이다.

정리 2.43. 문맥무관 언어는 (1) 합집합, (2) 접합, (3) 클리니 스타 연산에 대하여 닫혀 있다.

증명.

1. $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, S, P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\})$
2. $S \to S_1 S_2$
3. $S \to SS_1 \mid \epsilon$

□

정리 2.44. 문맥무관 언어는 (1) 교집합, (2) 여집합 연산에 대해 닫혀 있지 않다.

증명.

1. $\{a^n b^n c^m\} \cap \{a^n b^m c^m\}$ 은 각각은 문맥무관 언어이지만 교집합은 예제 2.40에 의해 문맥무관 언어가 아니다.
2. $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ 이므로 여집합에 대해 닫혀있으면 교집합에 대해서도 닫혀 있어야 한다.

□

예제 2.45. (소속 문제) 문자열 w가 문맥무관 언어 L에 속하는지 결정하라.

해설. 앞의 유한 오토마타에서는 오토마타를 사용하면 쉽게 풀 수 있었으나, 여기서는 PA를 사용하지 않는다. 그 이유는 PA가 비결정론적이기 때문에 구현이 매우 어렵고, 이를 결정론적인 오토마타로 흉내내도 시간 복잡도 상으로 이득이 없기 때문이다.
따라서 우리는 문맥무관 문법 G를 촘스키 표준형으로 바꾼뒤 CYK 알고리즘을 사용해서 소속문제를 $O(n^3)$에 풀 수 있다.[11]

정의 2.46. CYK (Cocke, Younger, Kasami) 알고리즘은 다음과 같은 의사코드와 같은 알고리즘이다. 여기서 $V_{i,j} = \{A \in V \mid A \stackrel{*}{\Rightarrow} a_i \cdots a_j\}$라고 하자. 그러면 $S \in V_{i,n}$인 경우에만 $w \in L(G)$이다. 따라서 $V_{i,j}$를 $d = j - i$가 작은 값에서 큰 값으로 가면서 계산하면 된다.

```
for i = 1 to n
    V_{i,i} = {A | A → a_i}
for d = 1 to n − 1
    for i = 1 to n − d
        j = 1 + d
        V_{i,j} = d
        for k = i to j − 1
            V_{i,j} = V_{i,j} ∪ {A | A → BC, B ∈ V_{i,k}, C ∈ V_{k+1,j}}
```

예제 2.47. 다음 문맥무관 문법이 $baaba$를 생성하는지 CYK 알고리즘을 사

[11] 혹시 시간 복잡도와 $O(n^3)$이 뭘 의미하는지 모르겠다면 잠시 6.1 절과 6.2 절을 보고 오자. 별로 보고 싶지 않거나 튜링 기계가 뭔지 몰라서 이해가 되지 않는다면 CYK 알고리즘은 '적당히 빠르고 유용해서 좋다'고 이해해도 무방하다.

용해서 확인하라.

$$S \to AB \mid BC$$
$$A \to BA \mid a$$
$$B \to CC \mid b$$
$$C \to AB \mid a$$

해설. CYK 알고리즘이 만드는 표는 그림 2.11과 같다. $S \in V_{1,5}$ 이므로 $baaba$

i\j	**1**	**2**	**3**	**4**	**5**
1	B	SA	$\emptyset$	$\emptyset$	SAC
2		AC	B	B	SAC
3			AC	SC	B
4				B	SA
5					AC

그림 2.11

는 이 문법이 만드는 언어에 속한다.

2.8 결정 내리누름 오토마타

지금까지 배운 PA는 기본적으로 비결정론적이다. 그럼 결정론적인 PA를 만들면 문맥무관 언어를 모두 표현할 수 있을까? 유한 오토마타에서는 결정론적이든 비결정론적이든 계산 능력이 동일했으나, PA에서는 아쉽게도 그렇지 않다.

정의 2.48. 결정 내리누름 오토마타(deterministic pushdown automata, DPA)의 전이함수 δ는 $Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$에서 $Q \times \Gamma^*$로의 부분 함수(partial

function)로 정의한다. DPA가 입력이나 스택을 읽지 않고 전이할 수 있도록 하는 것이 편리하므로 부분 함수로 정의하는 것이다. 비결정성을 막기 위해 DPA의 전이함수는 다음과 같은 제약조건을 갖는다.

- $\delta(q, \epsilon, A)$가 정의되면, 모든 $a \in \Sigma$에 대해 $\delta(q, a, A)$가 정의되지 않는다.
- $\delta(q, a, \epsilon)$가 정의되면, 모든 $A \in \Gamma$에 대해 $\delta(q, a, A)$가 정의되지 않는다.

DPA가 받아들이는 언어를 **결정 문맥무관 언어**라고 부른다.

예제 2.49. $L = \{0^n 1^n \mid n \geq 0\}$을 받아들이는 DPA를 구하라.

정리 2.50. 결정 문맥무관 언어는 여집합에 대해 닫혀있다.

증명. (느슨한 증명) PA는 문자열을 다 읽고 났을 때 상태가 여러 개가 되는데, 그 중 하나만 최종 상태에 들어가 있어도 문자열을 받아들이게 된다. 따라서 $Q - H$와 H를 뒤집어도 문자열을 다 읽었을 때의 상태들 중 최종 상태가 없다는 보장이 없다. 하지만 DPA는 문자열을 다 읽었을 때의 상태가 1개이므로 $Q - H$와 H를 뒤집으면 최종 상태가 있거나 없거나의 보장이 확실하다. 따라서 여집합에 의해 닫혀 있다. □

정리 2.51. 결정 문맥무관 언어가 아니면서 문맥무관 언어인 언어집합이 존재한다.

증명. $L = \{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$는 문맥무관 언어이다. L이 결정 문맥무관 언어라면, $\overline{L}$도 결정 문맥무관 언어이다. $\overline{L} \cap \{a^* b^* c^*\}$도 문맥무관 언어인데, 이는 $\{a^n b^n c^n\}$이므로 모순이다. □

정리 2.52. 문맥무관 언어 L에 대해, L을 받아들이는 DPA P가 존재하는 것과 L이 애매하지 않은 문법 G를 가지는 것은 동치이다.

증명. (느슨한 증명) DPA는 결정론적이므로 상태를 따라갈 때 갈 수 있는 경로가 한 가지뿐이다. 즉, 문맥무관 문법의 기준에서 어떤 변수를 따라 추적해야 하는지가 명확하다. 따라서 이는 애매할 수 없다. □

2.9 연습문제

연습문제 2.1. 예제 2.4과 2.5의 답이 정말 맞을까? 이를 증명해라.

연습문제 2.2. Let C be a context free language and R be a regular language. Show that $C \cap R$ is a context free.

연습문제 2.3. Consider the Language $L = \{a^n b^n c^m d^m \mid n, m \geq 1\}$
$\cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$.

1. Show that L is context free.
2. Show that L is inherently ambiguous.

연습문제 2.4. Give a context free grammar that generates language L ($N_a(w)$ means the number of occurrence of a in w)

$$L = \{w \mid w \in \{0,1\}^*, N_0(w) \neq N_1(w)\}$$

CHAPTER 3

튜링 기계와 재귀 언어

3.1 튜링 기계

지금까지 우리는 정규식, 유한 오토마타를 이용해 정규 언어를 표현했고, 문맥무관 문법, 내리누름 오토마타를 이용해 그보다 더 넓은 집합인 문맥무관 언어를 표현했다. 그럼 가장 넓은 언어집합을 표현할 수 있는 오토마타는 무엇일까? '궁극적인 오토마타'인 튜링 기계에 대해 알아보자.

정의 3.1. 튜링 기계(Turing Machine, TM) M은 여섯 가지 요소 $(Q, \Sigma, \Gamma, \delta, q_0, H)$로 구성된다.

1. Q는 상태들의 유한 집합
2. Σ는 입력 알파벳
3. Γ는 테입 알파벳 (공백 문자는 #로 나타낸다.)
4. δ는 $(Q - H) \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$인 함수
5. $q_0 \in Q$는 초기 상태

6. $H \subseteq Q$는 정지상태들의 집합

튜링 기계는 한쪽으로 무한한 길이의 테입과 하나의 제어기, 그리고 테입을 가르키는 헤드로 구성된다. 테입은 **칸**(cell)으로 구성되어 있다. δ는 함수이므로 튜링 기계는 결정론적으로 움직인다. L은 헤드가 왼쪽으로 한 칸, R는 오른쪽으로 한 칸, S는 헤드가 가만히 있는다는 뜻이다. 그리고 헤드는 테입의 맨 왼쪽 끝에서 왼쪽으로 움직이지 않는다고 가정한다.
DFA나 PA와는 달리 튜링 기계는 헤드가 이리저리 움직이면서 '입력을 다 읽는다'라는 개념이 없으므로, 언제 이 기계가 동작을 멈출 것인지를 명시해 주어야 한다. 따라서 정지 상태 $h \in H$에 진입하면 튜링 기계는 정지한다.
튜링 기계의 **상황**(configuration)은 현재 제어기의 상태 q, 테입의 내용 w와 헤드가 가르키고 있는 위치로 구성된다. 헤드 왼쪽에 있는 문자열이 u, 오른쪽에 있는 문자열이 v, 가르키고 있는 문자열을 a라 할때 현재 상황은 $(q, u\underline{a}v)$로 나타낸다.

정의 3.2. TM $M = (Q, \Sigma, \Gamma, \delta, q_0, H)$이 정의하는 언어 $L(M)$은 다음과 같다.

$$L(M) = \{w \in \Sigma^* \mid (q_0, \underline{\#}w) \vdash_M^* (h, w')\}$$

이때 $h \in H$이고, w'는 임의의 문자열이다.

정의 3.3. 어떤 언어 L에 대하여, $L = L(M)$인 TM M이 존재할 때 언어 L을 **재귀 열거 언어**(recursive enumerable language) 또는 **튜링 기계가 인식 가능한 언어**(Turing-recognizable language)라고 한다.[12]

예제 3.4. $L = \{0^n \mid n \geq 1\}$을 정의하는 TM M을 구하라.

해설. 그림 3.1 참조.

[12] 이 책에서는 주로 재귀 열거 언어라는 표현을 사용할 것이다.

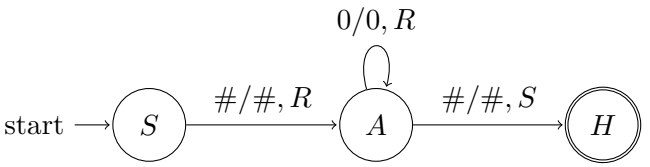


그림 3.1

예제 3.5. $L = \{0^n 1^n \mid n \geq 1\}$을 정의하는 TM M을 구하라.

해설. 그림 3.2 참조.

예제 3.6. $L = \{a^n b^n c^n \mid n \geq 1\}$을 정의하는 TM M을 구하라.

예제 3.7. $L = \{ww^R \mid w \in \{0,1\}^*\}$을 정의하는 TM M을 구하라.

예제 3.8. $L = \{ww \mid w \in \{0,1\}^*\}$을 정의하는 TM M을 구하라.

이처럼 튜링 기계의 계산 능력은 DFA, PA보다 아주 뛰어나다. 그런데 재귀 열거 언어에는 약간 문제가 있다. 어떤 문자열이 TM이 정의하는 언어에 속하면 정지하겠지만 속하지 않는 경우에는 정지하지 않고 무한히 돌아가기 때문에 우리가 무한한 시간 동안 기다리지 않는 이상 어떤 문자열이 언어 집합에 속하지 않는지 알 수 없다는 문제가 있다. 따라서 우리가 알기 쉬운 재귀 언어에 대해 알아보자.

정의 3.9. 언어 L에 대해 TM M에 대해 정지상태가 $\{y, n\}$이라 하자. $w \in L$이면 $(q_0, \underline{\#}w) \vdash_M^* (y, w')$이고 $w \notin L$이면 $(q_0, \underline{\#}w) \vdash_M^* (n, w')$를 만족하면 M이 L을 결정한다고 하고, L을 결정하는 M이 존재하면 L을 **재귀 언어**라고 부른다.

정리 3.10. 재귀 언어 종류는 재귀 열거 언어 종류의 부분 집합이다.

증명. n 상태를 무한하게 돌아가게 만들면 된다. □

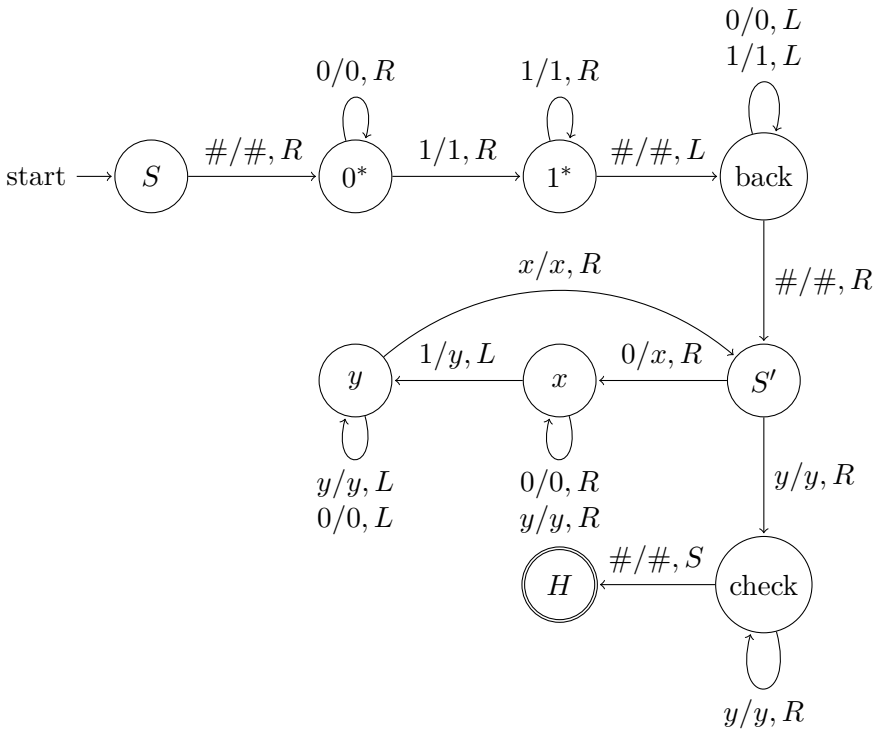


그림 3.2

예제 3.11. 예제 3.4∼3.8에서 정의한 튜링 기계가 정의하는 언어가 재귀 언어임을 보여라. (매우 간단하다.)

TM은 앞에서 다루었던 DFA, PA와는 달리 정지한 후 테이프에 문자열이 남는다. 이를 이용해 TM을 입력에서 출력을 계산하는 도구로도 생각할 수 있다.

정의 3.12. $f : \Sigma^* \to \Sigma^*$ 인 함수에 대해 TM M 이

$$(q_0, \underline{\#}w) \vdash_M^* (h, \underline{\#}f(w))$$

을 만족하면 M 이 f 를 **계산한다**(compute) 고 한다.
이때 f 를 **계산 가능한 함수**(computable function) 라고 한다.

예제 3.13. 입력 $w \in \{0,1\}$ 에 대해 이를 이진수로 해석해서 $2w$ 를 계산하는 TM M 을 구하라.

예제 3.14. 입력 $w \in \{0,1\}$ 에 대해 이를 이진수로 해석해서 $w+1$ 를 계산하는 TM M 을 구하라.

예제 3.15. 두 자연수 m, n 이 단항 표기법으로 0으로 구분되어 입력으로 주어진다. 단항 표기법이란, 예를 들어 설명하자면, 3, 5의 경우, 111011111로 주어진다. 이때 $m+n$ 을 단항 표기법으로 계산하는 TM M 을 구하라.

예제 3.16. 두 자연수 m, n 이 단항 표기법으로 0으로 구분되어 입력으로 주어진다. 단항 표기법으로 $m \times n$ 을 계산하는 TM M 을 구하라.

지금까지는 간단한 예제만 다뤘지만, 나누기, log 등등 더 복잡한 계산도 충분히 가능하다는 사실을 알았을 것이다.

3.2 튜링 기계의 확장

우리가 지금까지 다뤄 온 이 만능기계는 굉장히 계산 능력이 뛰어나다. 그럼 튜링 기계를 조금만 더 확장시켜서 더 계산 능력이 뛰어나게 할 수는 없을까? 테이프가 여러 줄인 튜링 기계를 생각하자.

정의 3.17. **k-테입 튜링 기계**는 일반적인 튜링 기계에서 테입이 k 개로 늘어난 형태다. 즉, 튜링 기계의 여섯 가지 요소가 다음과 같이 정의된다.

1. ... 생략 ...
2. δ는 $(Q-H)\times\overbrace{\Gamma\times\cdots\times\Gamma}^{k}\to Q\times\overbrace{\Gamma\times\cdots\times\Gamma}^{k}\times\{L,R,S\}^k$
3. ... 생략 ...

정리 3.18. k-테입 튜링 기계의 계산 능력은 튜링 기계와 동일하다.

증명. (느슨한 증명) 두 튜링 기계의 계산 능력이 동등하다는 것을 보이기 위해선 튜링 기계로 k-테입 튜링 기계를 흉내내면 된다. (반대는 지극히 당연하므로 보이지 않아도 된다.)
k-테입의 내용을 흉내내기 위해선 다음과 같이 k 개의 테입에서의 심볼을 하나의 별도의 심볼로 생각하면 된다. 즉, k개의 테입의 첫 번째 칸에 각각 a,b,c,d라고 적혀있으면, 튜링 기계에서는 이를 $abcd$라는 하나의 심볼로 생각하면 된다. k-테입 튜링 기계의 경우, k개의 헤드를 가지고 있으므로 이를 표시해주기 위해 다를 테입 심볼을 사용한다. 예를 들어 $abcd$라는 테입 심볼에서 두 번째 테입에 헤드가 있으면 $a\underline{b}cd$와 같은 방식으로 표시해 주면 된다. k 테입의 전이 함수는 헤드의 움직임을 한 번에 수정할 수 있지만 일반적인 튜링 기계는 이것이 불가능하므로 k번 전이함수를 쪼개서 그때 그때 헤드가 어디 있는지 탐색한 뒤 그에 맞는 전이를 하면 된다. □

3.3 비결정론적 튜링 기계

테입을 늘리는 것만으로는 튜링 기계의 계산 능력을 확장시킬 수 없는 것 같다. 그렇다면 비결정론적이게 만들면 가능할까? 아쉽게도 이렇게 해도 능력은 동등하다.

정의 3.19. 비결정론적 튜링 기계(Nondeterministic Turing Machine, NTM)는 일반적인 튜링 기계에서 전이가 비결정론적인 튜링 기계를 말한다.

1. ... 생략 ...

2. Δ는 $(Q-H)\times\Gamma\times Q\times\Gamma\times\{L,R,S\}$의 부분 집합으로써 함수가 아닌 전이 **관계**로 정의된다.

3. ... 생략 ...

정의 3.20. NTM M이 정의하는 언어 $L(M)$은 다음과 같다.

$$L(M)=\{w\in\Sigma^* \mid (q_0,\underline{\#}w)\vdash_M^* (h,*)\}$$

이때 M이 비결정론적이므로 계산 과정 중 하나라도 정지상태로 들어간다면 $w\in L(M)$이다.

정리 3.21. 비결정론적 튜링 기계와 (결정론적) 튜링 기계의 계산 능력은 동등하다.

증명. 비결정론적 튜링 기계 M과 동등한 결정론적 튜링 기계를 제시하면 된다. M은 비결정론적이므로 한 상황 C에서 여러가지 **상황** $C_1,\ldots,C_n$으로 전이할 수 있다. 이때, n은 비결정론적 튜링 기계의 정의에 의해 $|Q|\times|\Gamma|\times 3$ 이하의 값인 유한한 값을 갖는다. 이를 간단히 r이라 하자.
M의 계산 과정을 상황을 정점으로, 전이를 간선으로 표현하여 트리 T로 나타낼 수 있다. 이 트리에서 어떤 정점의 자식 정점의 개수는 r 이하로 유한하다. 즉, 이 트리의 모든 정점을 정수 $1,\ldots,r$으로 나타낼 수 있다. 루트는 초기 상태인 1이고 그 자식들은 $11,\ldots,1r$이다. 또한 11의 자식들을 $111,\ldots,11r$이다.
이제 3-테입 결정론적 튜링 기계 M'이 이를 흉내내고자 한다. M'은 트리 T의 정점을 하나하나 탐색하면서 정지하는 상태가 있는지 탐색한다. 깊이 우선 탐색(depth first search, DFS)을 하게 되면 트리가 유한하다는 보장이

없으므로 원래 M에서 정지상태가 존재함에도 끝나지 않을 수가 있기 때문에, 여기서는 너비 우선 탐색(breath first search, BFS)을 사용한다.
원래라면 큐(queue)를 이용하겠지만, 튜링 기계에서 그런 복잡한 자료구조를 구현하는 것은 어려우므로 오래걸리지만 조금 더 간단한 방식을 사용한다. 먼저 첫 번째 테입에는 입력 w를 보관한다. 두 번째 테입에는 $\{1, \ldots, r\}^*$ 인 문자열을 너비 우선 탐색의 순서로 하나씩 만들어간다. 이제 세번째 테입에서는 두 번째 테입에서 주어진 상황 문자열 $C = c_1 \ldots c_n$가 주어지면 그걸 이용해 w 입력이 어떻게 변할지 흉내낸다. 즉, 입력 w에 대해 먼저 c_1을 참고해서 M의 c_1번째 전이를 흉내내는 작업을 세 번째 타입에서 수행한다. 그 후, 다시 a_2를 적용하고 이를 계속 반복한다. 만약 해당 C가 정지 상황이면 M'도 정지하게 되고 그렇지 않으면 그러한 상황이 나올 때까지 계속 탐색하며 반복하므로 원래의 비결정론적 튜링 기계 M을 흉내낼 수 있다. □

3.4 랜덤 접근 기계

지금까지 다룬 튜링 기계는 아주 간결하면서도 계산 능력이 뛰어나지만 솔직히 말하면 '쓸모가 없다.' 무한한 길이의 테이프에서 헤드가 한 칸씩만 움직인다니 매우 비효율적이고 물리적으로 구현하기도 어렵다. 그럼 우리 현실 세계에서 구현하기 쉽고 다루기도 쉬운 튜링 기계 모델에는 어떠한 것이 있을까? 가장 대표적인 예시인 랜덤 접근 기계에 대해 알아보자.

정의 3.22. (느슨한 정의) 랜덤 접근 기계(Random Access Machine, RAM)는 여러 개의 **레지스터**(register), 한 방향으로 무한한 길이의 테이프, **프로그램 카운터**(program counter, PC)로 이루어진다. 레지스터를 각각 $R_0, R_1, \ldots$, 테이프의 각 조각을 $T[1], T[2], \ldots$, 프로그램 카운터를 κ라고 하자. 각 레지스터와 테이프 한 조각은 임의의 자연수를 저장할 수 있다.
랜덤 접근 기계는 여러개의 **명령어**(instruction)으로 이루어진 고정된 **프로**

그램(program)을 기반으로 작동한다.[13] 가능한 명령어 종류로는 그림 3.3이

명령어	피연산자	뜻
read	j	$R_0 \leftarrow T[R_j]$
write	j	$T[R_j] \leftarrow R_0$
store	j	$R_j \leftarrow R_0$
load	j	$R_j \leftarrow R_j$
loadimm	c	$R_j \leftarrow c$
add	j	$R_0 \leftarrow R_0 + R_j$
addimm	c	$R_0 \leftarrow R_0 + c$
sub	j	$R_0 \leftarrow R_0 - R_j$
subimm	c	$R_0 \leftarrow R_0 - c$
half		$R_0 \leftarrow \lfloor R_0/2 \rfloor$
jump	s	$\kappa \leftarrow s$
jpos	s	if $R_0 > 0$ then $k \leftarrow s$
jzero	s	if $R_0 = 0$ then $k \leftarrow s$
halt		$k \leftarrow 0$

그림 3.3

있다.[14] 레지스터는 0, 프로그램 카운터는 1로 초기화 된다. κ의 값은 몇 번째 명령어를 수행해야 하는지를 가르킨다. 한 명령어가 수행될 때마다 특별하게 κ의 값을 지정해주는 명령어가 아닌 이상 1씩 증가한다. κ가 0이 되면, 즉 halt 명령어를 수행하면 랜덤 접근 기계는 정지한다.

이 책에서는 랜덤 접근 기계의 좀 더 정확한 정의를 다루지 않으나, 위에서 정의한 내용만으로 충분히 이해할 수 있을거라 믿는다.

정리 3.23. 랜덤 접근 기계와 튜링 기계의 계산 능력은 동등하다.

[13] 혹시 컴퓨터 구조 과목을 공부했다면, CPU에 있는 ISA와 근본적으로 다른게 없다는 걸 느끼면 된다.

[14] 하나의 예일 뿐이며 다른 명령어들로도 충분히 다양한 연산이 가능하다.

증명. (느슨한 증명) 랜덤 접근 기계의 가장 중요한 특징은 레지스터에 저장된 주소값을 통해 바로 메모리 내의 값을 접근할 수 있다는 점인데, 3-테입 튜링 기계를 이용해 첫 번째 테입에서는 레지스터를 담당하고 두번째 테입은 메모리를 담당하고 마지막 테입은 명령어 입력을 받는 테입으로 활용하면 랜덤 접근 기계를 흉내낼 수 있다. □

3.5 무제한 문법

이제 튜링 기계와 동등한 계산 능력을 가지고 있는 언어를 알아볼 차례다.

정의 3.24. 무제한 문법(unrestricted grammar) $G = (V, \Sigma, S, P)$은 생성 규칙 $x \to y$가 임의의 형태인 $x \in (V \cup \Sigma)^* V, (V \cup \Sigma)^*, y \in (V \cup \Sigma)^*$를 가진다.

예제 3.25. $L = \{a^n b^n c^n \mid n \geq 0\}$을 생성하는 무제한 문법 G를 구하라.

해설.

$$S \to aBSc \mid \epsilon$$
$$Ba \to aB$$
$$Bc \to bc$$
$$Bb \to bb$$

예제 3.26. $L = \{a^n b^n c^n d^n \mid n \geq 0\}$을 생성하는 무제한 문법 G를 구하라.

예제 3.25를 보면 알 수 있듯이 무제한 문법은 문맥무관이 아닌 언어도 생성할 수 있다. 그럼 재귀 열거 언어를 모두 만들 수 있을까? 이는 가능하다.

정리 3.27. 무제한 문법 $G = (V, \Sigma, S, P)$에 대하여 $L(G)$를 정의하는 튜링 기계 M이 존재한다.

증명. 2-테입 비결정론적 튜링 기계 M을 만들자.[15] 먼저 M의 첫 번째 테입에서는 입력 w를 입력하고, 두 번째 테입에는 G의 문장형태를 만들어간다. 먼저 두 번째 테입에 S를 적고 시작한다.

1. 비결정적으로 두번째 테입의 임의의 위치를 선택한다.
2. 비결정적으로 G의 임의의 생성규칙 $x \to y$를 선택한다.
3. 이때 선택한 위치가 x이면 x를 y로 바꾼다. 글자 길이를 맞추기 위해 옆 글자를 이동시키는 작업이 필요하다.
4. 두 번째 테입에 있는 문장이 w와 비교해서 같으면 정지한다.
5. 이를 계속해서 반복한다.

G가 w를 생성하면 M의 한 갈래는 언젠가 정지한다. 그렇지 않으면 M은 정지하지 않으므로 $L(M) = L(G)$이다. □

정리 3.28. 튜링 기계 $M = (Q, \Sigma, \Gamma, \delta, q_0, H)$가 정의하는 언어 $L(M)$을 생성하는 무제한 문법 G가 존재한다.

증명. 우리는 G가 M이 계산하는 과정을 흉내내야 한다. 이때 일반성을 잃지 않고 M에서 $H = \{h\}$이고, M이 정지할 때 $(h, \#)$에서 정지한다고 가정할 수 있다. 이제 G의 생성규칙은 다음과 같이 만들 수 있다. 핵심은 M에서의 헤드의 위치와 상태를 표현하기 위해서 문자열 내에 상태 q를 표시하고 이를 현재 헤드가 가르키고 있는 문자의 오른쪽에 둔다는 것이다. 또한 테입의 끝은 []로 나타낸다.

- 모든 $\delta(q, a) = (p, b, S)$에 대해 G는 $bp \to aq$를 갖는다.

[15] k-테입 비결정론적 튜링 기계와 1-테입 비결정론적 튜링 기계의 계산 능력이 동등한 건 결정론적 튜링 기계에서의 증명과 유사하게 할 수 있다.

- 모든 $\delta(q,a) = (p,b,R)$에 대해 G는 모든 $c \in \Sigma \cup \{\#\}$에 대해 $bcp \to aqc$를 갖는다. 또한, $b\#p] \to aq]$를 갖는다.
- 모든 $\delta(q,q) = (p,b,L)$에 대해 $b \neq \#$이면 G는 $pb \to aq$를 갖는다. $b = \#$이면 모든 $c \in \Sigma$에 대해 $p\#c \to aqc$를 갖는다. 또한 $p] \to aq]$를 갖는다.

마지막으로 G는 처음과 끝을 위하여

$$\begin{aligned} S &\to [\#h] \\ [\#q_0 &\to \epsilon \\ ] &\to \epsilon \end{aligned}$$

를 갖는다. 그러면 M이

$$(q_0, \underline{\#}w) \vdash_M^* (h, \underline{\#})$$

전이하는 과정을, G는

$$S \Rightarrow [\#h] \stackrel{*}{\Rightarrow} [\#q_0 w] \stackrel{*}{\Rightarrow} w$$

로 유도하게 된다. □

3.6 μ-재귀 함수

튜링 기계가 받아들일 수 있는 언어는 재귀 열거 언어, 그 중에서도 튜링 기계가 항상 정지하면 계산 가능한 문제, 즉 재귀 언어라 한다. 그런데 왜 하필 '재귀'일까? 이는 재귀적인 방식으로 정의되는 함수를 통해 계산 가능한 함

수를 모두 표현할 수 있기 때문이다.

정의 3.29. $\mathbb{N}^k$에서 $\mathbb{N}$으로 가는 **기본 함수**(basic function)는 다음 세 함수를 의미한다.

1. 모든 $x_1, \ldots, x_n \in \mathbb{N}$에 대해 $zero(x_1, \ldots, x_n) = 0$이면 **영함수**(zero function)라 한다.

2. 모든 $x \in \mathbb{N}$에 대해 $succ(x) = x + 1$이면 **바로 뒤의 원소함수**(successor function)라 한다.

3. $x_1, \ldots, x_n \in \mathbb{N}, k \geq j > 0$에 대해 $id_k(x_1, \ldots, x_n) = x_k$이면 이를 **k-단위 함수**(k-ary identity function)라 한다.

그리고 이제 더 복잡한 함수를 만들기 위해 다음과 같은 함수 결합 방식을 정의하자.

1. $k, l \geq 0$, $g : \mathbb{N}^k \to \mathbb{N}$는 인자가 k개인 함수, $h_1, \ldots, h_k$는 인자가 l개인 함수라 하자. g와 $h_1, \ldots, h_k$의 **합성**(composition)은 다음과 같은 인자가 l개인 함수이다.

$$f(x_1, \ldots, x_l) = g(h_1(x_1, \ldots, x_l), \ldots, h_k(x_1, \ldots, x_l))$$

2. $k \geq 0$, g는 인자가 k개인 함수, h는 인자가 $k + 2$개인 함수라 하자. g, h에 의해 **재귀적으로 정의된 함수**(function defined recursively)는 다음과 같은 인자가 $k + 1$개인 함수이다.

$$f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$$
$$f(x_1, \ldots, x_k, m + 1) = h(x_1, \ldots, x_k, m, f(x_1, \ldots, x_k, m))$$

원시 재귀 함수(primitive recursive function)는 (1) 기본함수이거나 (2) 기본함수의 유한 번의 합성 및 재귀를 통해 얻을 수 있는 함수이다.

원시 재귀 함수는 반복 횟수가 정해져있는 for문만 사용해서 프로그래밍한 알고리즘이라고 생각하면 간단하다.

예제 3.30. 함수 $plus(m,n) = m+n$은 원시 재귀 함수임을 보여라.

해설.

$$plus(m,0) = m$$
$$plus(m,n+1) = succ(plus(m,n))$$

$succ(plus(m,n)) = succ(id_3(m,n,plus(m,n)))$이므로 id와 같은 '당연한' 함수는 편의를 위해 생략해서 사용한다.

예제 3.31. 함수 $mult(m,n) = m \cdot n$은 원시 재귀 함수임을 보여라.

해설.

$$mult(m,0) = zero(m)$$
$$mult(m,n+1) = plus(m, mult(m,n))$$

예제 3.32. 함수 $exponent(m,n) = m^n$은 원시 재귀 함수임을 보여라.

앞으로 $m+n, m \cdot n, m^n$과 같이 '당연히' 원시 재귀 함수인 함수들은 함수 표기 대신 평소에 사용하는 표기를 사용한다. 이제 편의를 위해 다양한 함수 및 개념을 미리 정의해두자.

정의 3.33. 바로 앞의 원소함수(predecessor function) $pred(n)$은 다음과 같이 정의된다.

$$pred(0) = 0$$
$$pred(n+1) = n$$

$n = 0$이면 1이고, $n > 0$이면 0인 함수 $iszero(n)$는 다음과 같이 정의된다.

$$iszero(0) = 1$$
$$iszero(m+1) = 0$$

비슷하게 $isone(n)$도 정의할 수 있다.
$iszero$와 같이 **원시 재귀 술어**(primitive recursive predicate)는 원시 재귀 함수 중 0과 1만을 값으로 가지는 함수이다.

예제 3.34. 음이 되지 않는 뺄셈 함수 $m \sim n = \max\{m - n, 0\}$가 원시 재귀 함수임을 보여라.

해설.

$$m \sim 0 = m$$
$$m \sim n + 1 = pred(m \sim n)$$

예제 3.35. $m > n$이면 1이고 이외에는 0인 원시 재귀 술어 $greater(m, n)$을 구하라.

예제 3.36. $m \geq n$이면 1이고 이외에는 0인 원시 재귀 술어 $greater\text{-}or\text{-}equal(m, n)$을 구하라.

정의 3.37. 인자가 k개인 함수 f, g, h와 원시 재귀 술어 p에 대해 **조건으로 정의된 함수**(function defined bt cases)는 다음과 같다.

$$f(n_1, \ldots, n_k) = \begin{cases} g(n_1, \ldots, n_k), & \text{if } p(n_1, \ldots, n_k); \\ h(n_1, \ldots, n_k), & \text{otherwise} \end{cases}$$

예제 3.38. 조건으로 정의된 함수가 원시 재귀 함수임을 보여라.

해설.

$$f(n_1, \ldots, n_k) = p(n_1, \ldots, n_k) \cdot g(n_1, \ldots, n_k) + (1 \sim p(n_1, \ldots, n_k)) \cdot h(n_1, \ldots, n_k)$$

예제 3.39. $m \div n$의 나머지와 몫을 각각 구하는 함수 $rem(m, n)$과 $div(m, n)$이 원시 재귀 함수임을 보여라.

해설.

$$rem(0, n) = 0$$

$$rem(m+1, n) = \begin{cases} 0 & \text{if } equal(rem(m, n), pred(n)); \\ rem(m, n) + 1 & \text{otherwise} \end{cases}$$

$$div(0, n) = 0$$

$$div(m+1, n) = \begin{cases} div(m, n) + 1 & \text{if } equal(rem(m, n), pred(n)); \\ div(m, n) & \text{otherwise} \end{cases}$$

예제 3.40. n을 p진법으로 나타냈을 때 m번째 자릿수의 숫자를 표현하는 함수 $digit(m, n, p)$이 원시 재귀 함수임을 보여라.

해설.

$$digit(m, n, p) = div(rem(n, p^m), p^{m\sim 1})$$

그럼 이제 원시 재귀 함수로 모든 계산 가능한 함수를 표현할 수 있을 것 같지만, 아쉽게도 그렇지 못하다.

정리 3.41. 원시 재귀 함수가 아닌 함수 중 계산 가능한 함수가 존재한다.

증명. 원시 재귀 함수는 기본 함수들의 유한 번의 합성 및 재귀를 통해 얻어지는 함수이므로 적절한 부호화를 통해 나열가능하다. 즉, 자연수 집합과 크기가 같다.[16] 따라서 적절히 부호화 했을 때 사전순으로 나열하는 것이 가능하다. 편의를 위해 인자가 한 개인 함수들만 생각하자. 이를 나열했을 때 다음과 같다.

$$f_0, f_1, f_2, f_3, \dots$$

모든 $n \geq 0$에 대해 함수 $g(n) = f_n(n) + 1$이라 정의하자. 이는 당연히 계산 가능하다. 만약 $g(n)$이 원시 재귀 함수라면 $g(n) = f_m(n)$인 m이 존재한다. 그럼 $g(m) = f_m(m) = f_m(m) + 1$이므로 이는 모순이다. 따라서 계산 가능하지만 원시 재귀 함수가 아닌 $g(n)$이 존재한다. □

이러한 계산 가능하지만 원시 재귀 함수가 아닌 함수의 예로는 아커만 함수가 있다.

[16] 혹시 잘 모르겠다면, 페아노 공리계를 한 번 고찰해보자.

정의 3.42. 아커만 함수(Ackermann function)는 다음과 같이 정의된다.

$$Ack(m,n) = \begin{cases} n+1, & \text{if } m = 0 \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ Ack(m-1, Ack(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

아커만 함수가 원시 재귀 함수가 아님을 보이기 위해 먼저 다음 정의를 소개한다.

정의 3.43. 다음을 만족하는 $b \in \mathbb{N}$가 존재할 때, 함수 $h : \mathbb{N}^2 \to \mathbb{N}$이 함수 $g : \mathbb{N}^k \to \mathbb{N}$를 **주요화한다**(majorize)고 한다. $a_1, \ldots, a_k \in \mathbb{N}$에 대해

$$g(a_1, \ldots, a_n) < h(a, b)$$

이때 $a = \max\{a_1, \ldots, a_k\} > 1$이다.

정리 3.44. A가 Ack에 의해 주요화되는 함수들의 집합이라 하자. 원시 재귀 함수들의 집합은 A의 부분 집합이다.

증명. 위 정리를 보이기 위해서는 기본 함수가 A에 포함되고, 합성과 재귀 연산에 대해 닫혀있음을 보이면 된다. 여기서 $x = \max\{x_1, \ldots, x_n\}, y = \max\{y_1, \ldots, y_n\}$이다.
먼저 기본 함수가 A에 포함됨을 보이자.

$$\begin{aligned} zero(n) = 0 < n+1 &= Ack(0,n) \\ succ(n) = n+1 < n+2 &= Ack(1,n) \\ id_k(x_1, \ldots, x_n) = x_k \leq x < x+1 &= Ack(0,x) \end{aligned}$$

다음으로 합성 연산에 대해 닫혀있음을 보이자. 함수 $g_1, \ldots, g_m$ 와 h 는 각각 인자가 k 개, m 개이고, A의 원소라 하자. 이는

$$g_i(x_1, \ldots, x_k) < Ack(r_i, x)$$
$$h(y_1, \ldots, y_m) < Ack(s, y)$$

를 의미한다.
이때 $f = h(g_1, \ldots, g_m)$, $g_{\max}(x_1, \ldots, x_k) = \max_i\{g_i(x_1, \ldots, x_k)\}$ 라 하자. 그럼

$$\begin{aligned} f(x_1, \ldots, x_k) &< Ack(s, g_{\max}(x_1, \ldots, x_k)) \\ &< Ack(s, Ack(r_{\max}, x)) < Ack(s + r_{\max} + 2, x) \end{aligned}$$

이다. 즉, $f \in A$ 임을 알 수 있다.
마지막으로 재귀 연산에 대해 닫혀있음을 보이자. 함수 g와 h는 각각 인자가 k 개, $k+2$ 개이고 모두 A의 원소라 하자. 즉,

$$g(x_1, \ldots, x_k) < Ack(r, x)$$
$$h(y_1, \ldots, y_{k+2}) < Ack(s, y)$$

이다. 이때 f는 g, h에 의해 재귀적으로 정의되어 있다고 하자.
먼저 x와 n과는 관계없이 다음을 만족하는 q가 있음을 증명하자.

$$f(x_1, \ldots, x_k, n) < Ack(q, n + x)$$

$q = 1 + \max\{r, s\}$ 라 하자. 먼저

$$f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k) < Ack(r, x) < Ack(q, x)$$

이다.
이때 $f(x_1, \ldots, x_k, n) < Ack(q, n+x)$ 라 하자. 그럼 $z = \max\{x, n, f(x_1, \ldots, x_k, n)\}$ 에 대해 다음이 성립한다.

$$f(x_1, \ldots, x_k, n+1) = h(x_1, \ldots, x_k, n, f(x_1, \ldots, x_k, n)) < Ack(s, z)$$

이때 수학적 귀납법에 의해 다음이 성립한다.

$$\begin{aligned} f(x_1, \ldots, x_k, n+1) &< Ack(s, z) < Ack(s, Ack(q, n+x)) \\ &\leq Ack(q-1, Ack(q, n+x)) = Ack(q, n+1+x) \end{aligned}$$

$w = \max\{x, y\}$ 라 하자.

$$\begin{aligned} f(x_1, \ldots, x_k, y) &< Ack(q, x+y) \leq Ack(q, 2w) \\ &< Ack(q, 2w+3) = Ack(q, A(2, w)) = Ack(q+4, w) \end{aligned}$$

즉, $f \in A$ 임을 알 수 있다.
따라서 원시 재귀 함수의 집합은 A의 부분 집합이다. □

정리 3.45. 아커만 함수는 계산 가능하지만 원시 재귀 함수가 아니다.

증명. Ack가 원시 재귀 함수면 정리 3.44에 의해 $Ack \in A$ 인데, 이는 불가능하므로 A는 원시 재귀 함수가 아니다. □

아커만 함수의 값은 그림 3.4에서 확인할 수 있다. 따라서 우리는 모든 계산

m\n	**0**	**1**	**2**	**3**	**4**
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	$2^{65536}-3$	$2\uparrow\uparrow 6-3$	$2\uparrow\uparrow 7-1$
5	$2\uparrow\uparrow\uparrow 3-3$	$2\uparrow\uparrow\uparrow 4-3$	$2\uparrow\uparrow\uparrow 5-3$	$2\uparrow\uparrow\uparrow 6-3$	$2\uparrow\uparrow\uparrow 7-3$

그림 3.4

가능한 함수를 표현하기 위해서 다음과 같은 방법을 정의한다.

정의 3.46. 인자가 $k+1$ 개인 함수 g에 대해 g의 **최소화**(minimalization)는 인자가 k 개인 함수 f로 다음과 같이 정의된다.

$$f(n_1,\ldots,n_k) = \{g(n_1,\ldots,n_k,m) = 1\text{이 되게 하는 최소 } m\}$$

이때 $g(n_1,\ldots,n_k,m) = 1$이 되게 하는 $m \geq 0$이 존재하면 g가 **최소화 가능하다**(minimalizable)고 하고, 만약 g가 최소화 가능하지 않으면 $f(n_1,\ldots,n_k) = 0$으로 정의한다. 이러한 g의 최소화 과정은 $\mu m\ [g(n_1,\ldots,n_k,m) = 1]$와 같이 표기한다.

정의 3.47. **μ-재귀 함수**(μ-recursive function)는 기본 함수들의 유한 번의 재귀 및 합성 또는 최소화 가능한 함수의 최소화를 통해 얻어낼 수 있는 함수이다.

어떤 함수가 최소화의 결과값은 어떻게 알 수 있을까? 이는 다음과 같은 방법을 통해 알 수 있다.

$m \leftarrow 0$
while $g(n_1,\ldots,n_k,m) \neq 1$ **do**

$m \leftarrow m + 1$

end while

이 방법의 문제점은 g가 최소화 가능하면 멈추지만 만약 최소화 가능하지 않다면 정지하지 않고 무한히 작동한다. 즉, 계산 가능한 문제라고 할 수 없다. 따라서 우리는 어떤 함수가 최소화 가능한지 판별하는 것은 '쉽지 않다.'

그럼 정리 3.41의 증명에서 원시 재귀 함수가 아닌 함수들 중 계산 가능한 함수가 있다는 것을 보일 때 사용한 논리를 μ-재귀 함수에 대해서도 동일하게 적용할 수 있을까? 이는 불가능하다. 앞의 논리를 생각해보자. 튜링 기계가 $g(n) = f_n(n) + 1$을 계산하기 위해서는 먼저 $f_0, f_1, \ldots$를 $n+1$개 나열한 뒤 $f_n(n) + 1$을 계산하면 된다. 하지만 이 '나열'하는 과정에서 우리는 μ-재귀 함수만 나열해야 하는데 이때 μ-재귀 함수의 조건 중에는 '최소화 가능한' 함수들의 최소화로 이루어진다라는 조건이 있다. 이때 우리가 아는 한 튜링 기계는 어떤 함수가 최소화 가능한지 계산할 수 없다. 따라서 μ-재귀 함수를 나열한다는 것 자체가 불가능하며 정리 3.41의 증명에서 사용한 논리를 사용할 수 없다.

예제 3.48. 로가리즘 함수 $\log(m, n) = \lceil \log_{m+2}(n+1) \rceil$가 μ-재귀 함수임을 보여라.($\log$가 범위를 벗어나지 않도록 정의했다.)

$$\log(m, n) = \mu p \left[\mathit{greater\text{-}or\text{-}equal}((m+2)^p, n+1) \right]$$

정리 3.49. 함수 $f : \mathbb{N}^k \to \mathbb{N}$가 μ-재귀 함수라는 것은 f가 튜링 기계에 의해 계산 가능하다는 것과 동치이다.

증명. 먼저 $(\Rightarrow)$를 증명하자.

f가 μ-재귀 함수라 하자. 먼저 기본 함수들이 계산 가능한 것은 자명하다. 다음으로 $f : \mathbb{N}^k \to \mathbb{N}$가 $g : \mathbb{N}^l \to \mathbb{N}$와 $h_1, \ldots, h_l : \mathbb{N}^k \to \mathbb{N}$의 합성으로 정의되었다고 하자. 이는 튜링 기계로 다음과 같이 계산할 수 있다.

$m_1 \leftarrow h_1(n_1, \ldots, n_k)$
$\vdots$
$m_l \leftarrow h_l(n_1, \ldots, n_k)$
output $g(m_1, \ldots, m_l)$

비슷하게 함수 g, h에 의해 재귀적으로 정의된 함수 $f(n_1, \ldots, n_k, m)$는 다음과 같이 계산할 수 있다.

$v \leftarrow g(n_1, \ldots, n_k)$
if $m = 0$ **then**
    output v
else
    for $i \leftarrow 1, \ldots, m$ **do**
        $v \leftarrow h(n_1, \ldots, n_k, i-1, v)$
    end for
    output v
end if

마지막으로 f가 $\mu m\,[g(n_1, \ldots, n_k, m)]$과 같이 정의되면 앞에서 봤듯이 다음과 같이 계산할 수 있다.

$m \leftarrow 0$
while $g(n_1, \ldots, n_k, m) \neq 1$ **do**
    $m \leftarrow m + 1$
end while

이때 g는 정의에 의해 최소화 가능한 함수이므로 이 알고리즘은 항상 정지한다.

이제 반대 방향($\Leftarrow$)을 증명하자.

먼저 편의를 위해 튜링 기계 $M = (Q, \Sigma, \Gamma, \delta, s, \{h\})$이 계산 가능한 함수

$f : \mathbb{N} \to \mathbb{N}$을 계산한다고 하자.[17] 이제 함수 f가 μ-재귀 함수임을 보이면 된다. 일반성을 잃지 않고 Q, Σ, Γ가 서로소라고 하자. $b = |Q| + |\Sigma| + |\Gamma|$라고 하고 함수 $E : Q \cup \Sigma \cup \Gamma \to \{0, 1 \ldots, b-1\}$를 일대일 함수라고 하자. 우리는 M의 상황을 b를 통해 표현할 것이다. 상황 $(q, a_1 a_2 \ldots \underline{a_k} \ldots a_n)$은 $a_1 a_2 \ldots a_k q a_{k+1} \ldots a_n$과 같은 형태로 b를 이용해 다음과 같이 정수로 표현한다. 이를 b진법 표기법이라 하자.

$$E(a_1)b^n + E(a_2)b^{n-1} + \ldots E(a_k)b^{n-k+1} + E(q)b^{n-k} + \cdots + E(a_n)$$

우리는 최종적으로 f를 다음과 같이 표현할 것이다.

$$f(n) = num(output(last(comp(n))))$$

num은 0과 1로 표기된 b진법 표기법을 정수로 바꿔주는 함수이다. $output$은 $\#hf(w)$ 형태로 표기된 b-정수 표기법에서 $\#, h$를 제거해주는 함수이다. 이 두 함수가 μ-재귀 함수임은 자명하다. 따라서 이 책에선 정확히 어떻게 정의되는지 기술하지 않는다.

$comp$함수는 시작 상황인 $\#sw$에서 시작해 $\#hf(w)$로 끝나는 상황을 튜링 기계의 전이 함수에 맞춰 b진법 표기법으로 순서대로 나열한다. $last$ 함수는 나열된 상황 중에서 마지막 정지 상황만을 추출한다.

먼저 $last$ 함수부터 정의하자. 이를 위해 먼저 함수 $lastpos$을 정의하자.

$$lastpos(n) = \mu m \, [equal(digit(m, n, b), E[\#]) \text{ or } equal(m, n)]$$

$lastpos$ 함수는 가장 오른쪽에 있는 # 문자를 찾는다. 이때 함수가 최소화 가능하도록 하기 위해 $equal(m, n)$을 추가하였다. 이를 통해 함수 $last$는 다

[17] 인자가 여러개인 함수도 유사하게 확장하여 증명할 수 있다. 연습문제 3.7 참고.

음과 같이 정의할 수 있다.

$$last = rem(n, b^{lastpos(n)})$$

다음으로 가장 핵심이 되는 $comp$ 함수를 정의하자. 이는 다음과 같의 정의될 수 있다.

$$comp(n) = \mu m\ [iscomp(m, n) \text{ and } halted(last(m))]$$

여기서 $iscomp$는 입력 n에 대해 m이 튜링 기계 M에 대해서 제대로 된 상황의 나열인지 검사한다. $iscomp$의 정확한 정의는 다루지 않으나, $iscomp$가 원시 재귀 함수인 것은 쉽게 알 수 있다.(모든 전이 함수의 case에 대해서 m이 제대로 상황이 나열됐는지 검사하면 된다.) 함수 $halted$의 경우 m에 정지 상태 h가 포함되는지 검사한다. 즉, m이 적절히 나열된 일련의 상황이고 m이 정지상태를 포함할 때의 m의 값을 구할 수 있다. 이때 f는 계산 가능한 함수라고 했으므로 정지상황이 항상 존재하므로 당연히 최소화 가능하다. 따라서 튜링 기계에 의해 계산 가능한 함수 f는 μ-재귀 함수이다. □

이를 통해 우리는 재귀 열거 언어 집합을 새롭게 정의할 수 있다.

정의 3.50. 부분 재귀 함수(partial recursive function)란 기본 함수들의 유한번의 재귀, 합성, 최소화를 통해 얻어낼 수 있는 부분 함수이다. (μ-재귀 함수와 달리 최소화 가능한 함수만 최소화하는 게 아니다.)

정리 3.51. 언어 L이 재귀 열거 언어인 것과 L이 부분 재귀 함수의 정의역인 것은 동치이다.

증명. 당연하다. □

이때 정의역으로 언어가 정의가 되는 이유는 부분 재귀 함수는 모든 자연수 값에 대해 정의되어 있는 **완전 함수**(total function)가 아니라 자연수 중에 값

이 결정되지 않는(즉, 계산이 불가능한) 값들이 존재하는 **부분 함수**(partial function)이기 때문이다.

3.7 람다 계산법

튜링 기계는 이름에서 알 수 있듯이 영국의 수학자 앨런 튜링이 만든 개념이다. 그럼 튜링은 이런 복잡한 기계를 왜 만들었을까? 지금의 컴퓨터처럼 무언가 많은 일을 할 수 있는 자동화된 기계를 만들기 위해서 이런 기계를 제안했을 것이라고 오해할 수 있지만, 사실은 그렇지 않다. 앨런 튜링이 하고 싶었던 것은 괴델의 불완전성 정리를 자신만의 방식으로 증명하고 싶었던 것이다. 그런데 앨런 튜링보다 먼저 알론조 처치(Alonzo Church)가 비슷한 방식으로 이러한 증명을 제안해냈는데, 이를 **람다 계산법**(lambda calculus)이라고 한다.

정의 3.52. 람다식(lambda expression)은 다음과 같이 문맥무관 언어로 정의된다.

$$\begin{array}{rcll} E & \to & x & \text{변수(variable)} \\ & | & \lambda x.E & \text{추상화(abstraction)} \\ & | & E\ E & \text{적용(application)} \end{array}$$

1. x는 어떤 변수를 의미한다.

2. $\lambda x.E$는 x를 입력으로 갖고 E를 반환하는 함수를 의미한다. 예를 들어 $\lambda x.x$의 경우 $f(x) = x$를 표현한 것이라 생각하면 된다.

3. E_1E_2는 E_1에 E_2를 넣는다는 뜻이다. 예를 들어, $\lambda x.x\ \ x$의 경우 이를 적용하면 x가 된다.

이러한 계산 모델은 아주 간단해 보이지만 실은 우리가 앞서 힘들게 정의했던 튜링 기계와 계산 능력이 동등하다. 어떻게 계산이 이루어지는지 알아보기 위해 먼저 여러가지 것들을 정의하자.

정의 3.53. 람다식 E의 **자유변수**(free variable) $FV(E)$는 E에서 λ에 의해 묶여있지 않은 변수들의 집합이다. 즉 다음과 같이 정의된다.

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.E) &= FV(E) \backslash \{x\} \\ FV(E_1\ E_2) &= FV(E_1) \cup FV(E_2) \end{aligned}$$

정의 3.54. 치환(substitution)은 변수를 다른 것으로 바꿔주는 연산이다. 치환 S는 변수 x_i를 Y_i로 바꿔주는 $x_i \mapsto Y_i$의 집합이다.

$$S = \{x_1 \mapsto Y_1, \ldots, x_n \mapsto Y_n\}$$

또는 다음과 같이 쓰기도 한다.

$$S = \{Y_1/x_1, \ldots, Y_n/x_n\}$$

정의 3.55. 치환 S를 람다식 E에 적용하는 것을 $S\ E$로 쓰고 다음과 같이 정의된다.

$$S\ x = \begin{cases} E & E/x \in S \\ x & \text{그 외} \end{cases}$$

$$S\ (\lambda x.E) = \lambda y.(S\ \{y/x\}\ E)\ (\text{완전히 새로운 } y)$$

$$S\ (E1\ E2) = (S\ E1)(S\ E2)$$

참고로, 치환 S,T를 나란히 $S\ T$라고 쓰면, T를 적용하고 S를 적용한다는 뜻이다.

정의 3.56. 람다 계산법에서의 **연산**(reduction)은 다음 2가지로 이루어진다.

- α-연산(α-reduction)은 $x' \notin FV(\lambda x.E)$에 대해 $\lambda x.E \to \lambda x'.(\{x'/x\}\ E)$를 해주는 연산이다.
- β-연산(β-reduction)은 $(\lambda x.E1)\ E2 \to \{E_2/x\}E_1$을 해주는 연산이다.

여기서 α-연산의 경우 단순히 변수명을 바꿔주는 연산이다. 람다 계산법에서 중요한 연산은 바로 β-연산이다.

정의 3.57. 레덱스(redex)란 람다식에서 β-연산이 가능한 부분, 즉 $(\lambda x.E_1)\ E_2$인 부분을 의미한다.

정의 3.58. 레덱스가 없는 람다식을 **정상식**(normal form)이라고 한다.

예제 3.59. 다음 중 정상식을 고르시오.

1. $\lambda x.x$
2. $(\lambda y.y)\ x$
3. $(\lambda x.x\ x)\ (\lambda x.x\ x)$

해설. 1은 더이상 β-연산이 불가능하므로 정상식이다. 2는 한 번 β-연산을 할 수 있으므로 정상식이 아니다. 3은 무한히 β-연산을 할 수 있으므로 정상식이 아니다.

예제 3.59의 3에서 알 수 있듯이, 정상식으로 끝날 수 없는 람다식도 존재하는 것 같다. 그러나 람다식이 만약 정상식으로 끝난다면 그 정상식은 유일하다. 이를 보이기 위해 여러가지 것들을 정의하자.

정리 3.60. (처치-로서 정리) 모든 람다식 M, N_1, N_2에 대해 M에 임의의 β-연산들을 한 결과가 각각 N_1, N_2이라고 하자. 이때, N_1, N_2에 각각 β-연산들을 해서 도출되는 공통의 람다식 X가 존재한다. 이를 좀 더 형식적으로 기술하면 다음과 같다.

$$\forall M, N_1, N_2 \in \Lambda, (M \to_\beta^* N_1) \wedge (M \to_\beta^* N_2) \to \exists X \in \Lambda, (N_1 \to_\beta^* X) \wedge (N_2 \to_\beta^* X)$$

여기서 $\to_\beta^*$는 β-연산의 반사적이고 추이적인 폐포, Λ는 람다식을 모은 집합이다.

이 정리를 증명하기 위해서는 먼저 여러가지 정의와 도움정리가 필요하다.

정의 3.61. 이항 관계 $\to$가 다음을 만족하면 **다이아몬드 성질**(diamond property)을 만족한다고 하자.

$$\forall M, N_1, N_2, (M \to N_1) \wedge (M \to N_2) \to \exists X, (N_1 \to X) \wedge (N_2 \to X)$$

도움정리 3.62. 어떤 관계 $\to$가 다이아몬드 성질을 만족하면 $\to$의 전이적이고 추이적인 폐포 $\to^*$ 또한 다이아몬드 성질을 만족한다.

증명. 자명하다. □

처치-로서 정리는 β-연산의 반사적이고 추이적인 폐포가 다이아몬드 성질을 만족한다는 것이다. β-연산이 다이아몬드 성질을 만족하면 좋겠지만 아쉽게도 이는 그렇지 않다. 대신에 증명을 위해 새로운 관계를 도입하자.

정의 3.63. 관계 $\twoheadrightarrow$는 다음과 같이 정의된다. $M, N \in \Lambda$이다.

- $M \twoheadrightarrow M$
- $M \twoheadrightarrow M' \Rightarrow \lambda x.M \twoheadrightarrow \lambda x.M'$

- $M \twoheadrightarrow M' \wedge N \twoheadrightarrow N' \Rightarrow M\ N \twoheadrightarrow M'\ N'$

- $M \twoheadrightarrow M' \wedge N \twoheadrightarrow N' \Rightarrow (\lambda x.M)\ N \twoheadrightarrow \{N'/x\}M'$

도움정리 3.64. (치환 도움정리) $M, N, P \in \Lambda$라고 하자. 만약 $x \neq y$이고 $x \notin FV(P)$이면 다음이 성립한다.

$$\{P/y\}\{N/x\}M = \{\{P/y\}N/x\}\{P/y\}M$$

증명. M을 3가지 경우로 나눠서 생각하자.

1. M이 변수인 경우: 먼저 $M = x$라고 하자. 그럼 좌항은 $\{P/y\}\{N/x\}x = \{P/y\}x$이다. 우항은 $\{\{P/y\}N/x\}\{P/y\}x = \{\{P/y\}N/x\}x = \{P/y\}N$이다. 이번엔 $M = y$라고 하자. 그럼 좌항은 $\{P/y\}\{N/x\}y = \{P/y\}y = P$이다. 우항은 $x \notin FV(P)$이므로 $\{\{P/y\}N/x\}\{P/y\}y = \{\{P/y\}N/x\}P = P$이다. $M = z$인 경우, 좌항과 우항 모두 z가 된다.

2. $M = \lambda z.M'$인 경우: 먼저 적절하게 α-연산이 이루어져서 $z \neq x, z \neq y, z \notin FV(P)$라고 하자.

$$\begin{aligned}\{P/y\}\{N/x\}(\lambda z.M') &= \lambda z.\{P/y\}\{N/x\}M' \\ &= \lambda z.\{\{P/y\}N/x\}\{P/y\}M' \text{ (귀납법)} \\ &= \{\{P/y\}N/x\}\{P/y\}\lambda z.M'\end{aligned}$$

3. $M = M_1\ M_2$ 인 경우:

$$\begin{aligned}
&\{P/y\}\{N/x\}(M_1\ M_2)\\
&= (\{P/y\}\{N/x\}M_1)\ (\{P/y\}\{N/x\}M_2)\\
&= (\{\{P/y\}N/x\}\{P/y\}M_1)\ (\{\{P/y\}N/x\}\{P/y\}M_2)\ (\text{귀납법})\\
&= \{\{P/y\}N/x\}\{P/y\}(M_1\ M_2)
\end{aligned}$$

□

사실 직관적으로는 N 치환을 먼저 하냐, P를 하고 N 치환을 P 치환에 맞게 적절히 바꿔주고 나중에 하냐의 차이이므로 당연히 같다는 것을 알 수 있다.

도움정리 3.65. $M, N \in \Lambda$에 대해, $N \twoheadrightarrow N'$이면 $\{N/x\}M \twoheadrightarrow \{N'/x\}M$이다.

증명. 이것도 앞의 증명처럼 케이스를 나누어서 생각할 수 있다.

1. $M = x$ 인 경우:

$$\{N/x\}x = N \twoheadrightarrow N' = \{N'/x\}x$$

2. $M = y\ (y \neq x)$ 인 경우:

$$\{N/x\}y = y \twoheadrightarrow y = \{N'/x\}y$$

3. $M = P\ Q$ 인 경우: 그럼 $\{N/x\}M = \{N/x\}P\ \{N/x\}Q$이다. 귀납적으로 $\{N/x\}P \twoheadrightarrow \{N'/x\}P, \{N/x\}Q \twoheadrightarrow \{N'/x\}Q$이다. 이제 $\twoheadrightarrow$의

정의에 따라

$$\begin{aligned}\{N/x\}P\ \{N/x\}Q &\twoheadrightarrow \{N'/x\}P\ \{N'/x\}Q \\ &= \{N'/x\}(PQ)\end{aligned}$$

임을 알 수 있다.

4. $M = \lambda x.P$인 경우:

$$\{N/x\}(\lambda x.P) = \{N'/x\}(\lambda x.P) = \lambda x.P$$

5. $M = \lambda y.P$ $(y \neq x)$인 경우: 적절하게 α-연산이 이루어졌다고 가정하자. $\lambda y.(\{N/x\}P)$에 대해, 귀납적으로 $\{N/x\}P \twoheadrightarrow \{N'/x\}P$임을 알 수 있다. $\twoheadrightarrow$의 정의에 의해

$$\begin{aligned}\{N/x\}(\lambda y.P) &= \lambda y.(\{N/x\}P) \\ &\twoheadrightarrow \lambda y.(\{N'/x\}P) \\ &= \{N'/x\}(\lambda y.P)\end{aligned}$$

임을 알 수 있다.

□

도움정리 3.66. $M \twoheadrightarrow M', N \twoheadrightarrow N'$이면 $\{N/x\}M \twoheadrightarrow \{N'/x\}M'$이다.

증명. $M \twoheadrightarrow M'$의 정의에 따라 케이스를 나누어 생각할 수 있다.

1. $M = M'$: 도움정리 3.65에 의해 당연하다.

2. $M = \lambda y.M_1, M' = \lambda y.M_1', M_1 \twoheadrightarrow M_1'$: 적절히 α-연산이 이루어졌다고 가정하자. $\{N/x\}(\lambda y.M_1) = \lambda y.(\{N/x\}M_1)$ 이므로 귀납법에 의해 $\{N/x\}M_1 \twoheadrightarrow M_1'\{N'/x\}$ 이다. 즉, $\{N/x\}M = \{N/x\}\lambda y.M_1 \twoheadrightarrow \lambda\{N'/x\}y.M_1' = \{N'/x\}M'$ 이다.

3. $M = P\ Q, M' = P'\ Q', M_1 \twoheadrightarrow M_1'$: $\{N/x\}M = \{N/x\}P\ \{N/x\}Q$ 이므로 귀납법에 의해 $\{N/x\}P \twoheadrightarrow \{N'/x\}P', \{N/x\}Q \twoheadrightarrow \{N'/x\}Q'$ 이다. 따라서

   $$\{N/x\}M = \{N/x\}P\ \{N/x\}Q \twoheadrightarrow \{N'/x\}P'\ \{N/x\}Q = \{N'/x\}M'$$

   임을 알 수 있다.

4. $M = (\lambda y.P)\ Q, M' = \{Q'/y\}P', P \twoheadrightarrow P', Q \twoheadrightarrow Q'$:

   $$\{N/x\}M = (\lambda y.\{N/x\}P)\ \{N/x\}Q$$

   이고 귀납법에 의해 $\{N/x\}P \twoheadrightarrow \{N'/x\}P', \{N/x\}Q \twoheadrightarrow \{N'/x\}Q'$ 임을 알 수 있다. $\lambda y.\{N/x\}P \twoheadrightarrow \lambda y.\{N'/x\}P'$ 를 이용하면

   $$\begin{aligned}\{N/x\}M &= (\lambda y.\{N/x\}P)\ \{N/x\}Q \\ &\twoheadrightarrow \{\{N'/x\}Q'/y\}\{N'/x\}P' \\ &= \{N'/x\}\{Q'/y\}P' \text{ (치환 도움정리)} \\ &= \{N'/x\}M'\end{aligned}$$

□

도움정리 3.67. $\lambda x.M \twoheadrightarrow N$ 이고 $M \twoheadrightarrow M'$ 이면 $N = \lambda x.M'$ 이다.

증명. 케이스를 나누어서 생각하자.

1. $\lambda x.M = N$: $M \twoheadrightarrow M'$ 이므로 $N = \lambda x.M \twoheadrightarrow \lambda x.M'$ 이다.
2. $N = \lambda x.M'$: 자명하다.

□

도움정리 3.68. $M\ N \twoheadrightarrow L$ 이면 다음 중 하나를 만족한다.

1. $M \twoheadrightarrow M', N \twoheadrightarrow N'$ 에 대해 $L = M'\ N'$ 이다.
2. $M = \lambda x.P, L = \{N'/x\}P', P \twoheadrightarrow P', N \twoheadrightarrow N'$ 이다.

증명. 케이스를 나누어서 생각하자.

1. $M\ N = L$: 자명하다.
2. $L = M'\ N', M \twoheadrightarrow M', N \twoheadrightarrow N'$: 자명하다.
3. $M = \lambda x.M_1, L = \{N'/x\}M_1', M_1 \twoheadrightarrow M_1', N \twoheadrightarrow N'$: 이것도 자명하다...

□

도움정리 3.69. $\twoheadrightarrow$는 다이아몬드 성질을 만족한다.

증명. $M \twoheadrightarrow M_1, M \twoheadrightarrow M_2$ 인 람다식 M, M_1, M_2을 생각하자. 다이아몬드 성질은 $M_1 \twoheadrightarrow M_3, M_2 \twoheadrightarrow M_3$ 인 M_3이 존재한다는 건데, 이를 증명하기 위해 $M \twoheadrightarrow M_1$의 케이스를 분류하자.

1. $M = M_1$: $M_3 = M_2$라고 두면 당연히 성립한다.
2. $M = P\ Q, M_1 = P'\ Q', P \twoheadrightarrow P', Q \twoheadrightarrow Q'$: 도움정리 3.68를 생각하면 다음과 같이 두 케이스로 나눌 수 있다.

(a) $P \twoheadrightarrow P'', Q \twoheadrightarrow Q'', M_2 = P''\ Q''$: 먼저 $P \twoheadrightarrow P', P \twoheadrightarrow P''$ 이고, $Q \twoheadrightarrow Q', Q \twoheadrightarrow Q''$ 임을 알 수 있다. 귀납법에 의해 $P' \twoheadrightarrow P''', P'' \twoheadrightarrow P''', Q' \twoheadrightarrow Q''', Q'' \twoheadrightarrow Q'''$ 인 P''', Q''' 가 존재함을 알 수 있다. 이 때, $M_1 = P'\ Q' \twoheadrightarrow P'''\ Q'''$ 이고 $M_2 = P''Q'' \twoheadrightarrow P'''Q'''$ 이므로 $M_3 = P'''\ Q'''$ 이 된다.

(b) $P = \lambda x.P_1, M_2\{Q''/x\}P_1'', P_1 \twoheadrightarrow P_1'', Q \twoheadrightarrow Q''$: 먼저 $P_1 \twoheadrightarrow P_1', P' = \lambda x.P_1'$ 이다. 이제 $P_1 \twoheadrightarrow P_1'$ 이고 $P_1 \twoheadrightarrow P_1''$ 이고 $Q \twoheadrightarrow Q', Q \twoheadrightarrow Q''$ 임을 알 수 있다. 따라서 귀납법에 의해 $P_1' \twoheadrightarrow P_1''', P_1'' \twoheadrightarrow P_1''', Q' \twoheadrightarrow Q''', Q'' \twoheadrightarrow Q'''$ 인 P_1''', Q''' 이 존재함을 알 수 있다. 따라서 $M_1 = P'\ Q' = (\lambda x.P_1')\ Q' \twoheadrightarrow \{Q'''/x\}P_1''', M_2 = \{Q_1''/x\}P_1'' \twoheadrightarrow \{Q_1'''/x\}P_1'''$ 이므로 $M_3 = \{Q_1'''/x\}P_1'''$ 를 얻을 수 있다.

3. $M = (\lambda x.P)\ Q, M_1 = \{Q'/x\}P', P \twoheadrightarrow P', Q \twoheadrightarrow Q'$: 이것도 똑같이 도움정리 3.68를 생각하면 다음과 같이 두 케이스로 나눌 수 있다.

(a) $M_2 = (\lambda x.P''')\ Q'', P \twoheadrightarrow P'', Q \twoheadrightarrow Q''$: 먼저 $P \twoheadrightarrow P', P \twoheadrightarrow P''$ 이다. 그리고 $Q \twoheadrightarrow Q', Q \twoheadrightarrow Q''$ 도 알 수 있다. 따라서 귀납법에 의해 $P' \twoheadrightarrow P''', P'' \twoheadrightarrow P''', Q' \twoheadrightarrow Q''', Q'' \twoheadrightarrow Q'''$ 인 P''', Q''' 가 존재한다. 이제 $M_1 = \{Q'/x\}P' \twoheadrightarrow \{Q'''/x\}P'''$ 이고 $M_2 = (\lambda x.P'')\ Q'' \twoheadrightarrow \{Q'''/x\}P'''$ 임을 알 수 있다. 따라서 $M_3 = \{Q'''/x\}P'''$ 을 얻을 수 있다.

(b) $M_2 = \{Q''/x\}P'', P \twoheadrightarrow P'', Q \twoheadrightarrow Q''$: 먼저 $P \twoheadrightarrow P', P \twoheadrightarrow P''$ 이다. 그리고 $Q \twoheadrightarrow Q', Q \twoheadrightarrow Q''$ 도 알 수 있다. 따라서 귀납법에 의해 $P' \twoheadrightarrow P''', P'' \twoheadrightarrow P''', Q' \twoheadrightarrow Q''', Q'' \twoheadrightarrow Q'''$ 인 P''', Q''' 가 존재한다. 이제 $M_1 = \{Q'/x\}P' \twoheadrightarrow \{Q'''/x\}P''', M_2 = \{Q''/x\}P'' \twoheadrightarrow \{Q'''/x\}P'''$ 임을 알 수 있다. 따라서 $M_3 = \{Q'''/x\}P'''$ 를 얻을 수 있다.

4. $M = \lambda x.P, M_1 = \lambda x.P', P \twoheadrightarrow P'$: 먼저 도움정리 3.67에 의해 $P \twoheadrightarrow P''$ 이고 $M_2 = \lambda x.P''$ 인 P'' 이 존재함을 알 수 있다. 귀납법에 의해 $P' \twoheadrightarrow P''', P'' \twoheadrightarrow P'''$ 인 P''' 가 존재하므로, $M_1 = \lambda x.P' \twoheadrightarrow P'''$ 이고 $M_2 = \lambda x.P'' \twoheadrightarrow P'''$ 이다. 따라서 $M_3 = \lambda x.P'''$ 를 얻을 수 있다.

□

이제 정리 3.60의 증명을 할 수 있다.

증명. (처치-로서 정리의 증명) 먼저 도움정리 3.69에 의해 $\twoheadrightarrow$ 이 다이아몬드 성질을 만족한다는 것은 알 수 있다. 이때, $\rightarrow_\beta \subseteq \twoheadrightarrow \subseteq \rightarrow_\beta^*$ 이므로 $\rightarrow_\beta^*$ 가 $\twoheadrightarrow$ 의 전이적이고 추이적인 폐포임을 알 수 있다. $\twoheadrightarrow$ 가 다이아몬드 성질을 만족하므로 도움정리 3.62에 의해 $\rightarrow_\beta^*$ 도 다이아몬드 성질을 만족한다. □

그럼 이제 계산이 끝난다면 그 값이 유일함을 알게 되었다. 그렇다면 어떤 방식으로 해야 정상식에 다다를 수 있을까?

정의 3.70. 정상 순서 연산(normal-order reduction) 이란 가장 왼쪽(left-most) 이면서 가장 바깥쪽(outermost) 의 레덱스부터 베타 연산을 수행하는 연산이다.

예제 3.71. 다음 람다식을 정상 순서 연산 방식을 통해 정상식으로 만들어라.

1. $\lambda u.(\lambda u.((u\ y)\ \lambda v.x)\ (\lambda x.x\ \lambda u.u))$
2. $(\lambda z.x)((\lambda x.x\ x)\ (\lambda x.x\ x))$

2의 경우, $(\lambda x.x\ x)\ (\lambda x.x\ x)$ 부터 연산을 진행할 경우 영원히 반복된다.

정리 3.72. (표준화 정리, standardization theorem) 모든 계산이 끝나는 람다식에 대해, 정상 순서 연산을 하면 정상식으로 결정된다.

증명. 일단은 생략한다... □

이제 이러한 간단한 연산을 이용해 아주 다양한 계산을 할 수 있음을 보이자.

정의 3.73. 처치 부호화(Church encoding)는 다음과 같이 정의된다.

$$\begin{aligned}
0 &= \lambda f.\lambda x.x \\
1 &= \lambda f.\lambda x.f\ x \\
n &= \lambda f.\lambda x.f^n\ x \\
\text{true} &= \lambda x.\lambda y.x \\
\text{false} &= \lambda x.\lambda y.y \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 &= e_1\ e_2\ e_3 \\
\text{not} &= \lambda b.\lambda x.\lambda y.b\ y\ x \\
\text{and} &= \lambda p.\lambda q.p\ q\ p \\
\text{iszero} &= \lambda n.\lambda x.\lambda y.n\ (\lambda z.y)\ x \\
\text{plus} &= \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x) \\
\text{mult} &= \lambda m.\lambda n.\lambda f.n\ (m\ f) \\
\text{pred} &= \lambda n.\lambda f.\lambda x.\lambda n.(\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u)
\end{aligned}$$

이것만으론 아쉽게도 아직 계산 가능한 모든 함수를 표현할 수는 없다.[18] 어떻게 하면 재귀함수를 잘 정의할 수 있을까?

정의 3.74. Y-결합자(Y-combinator)는 다음과 같이 정의된다.

$$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

[18] 왜 그럴까?

정리 3.75. 람다식 g에 대해 Y-결합자는 다음을 만족한다.

$$Y\ g = g\ Y\ g$$

증명.

$$\begin{aligned} Y\ g &= \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))\ g \\ &= (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x)) \\ &= g\ (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x)) \\ &= g\ Y\ g \end{aligned}$$

□

예제 3.76. 다음과 함수 fac를 정의하자.

```
def fac n =
    if iszero n
    then 1
    else mult n (f (pred n))
```

함수 fac는 람다 식으로는 간단히 다음과 같이 정의된다.

$$fac = \lambda f.\lambda n.\text{if iszero } n \text{ then } 1 \text{ else mult } n\ (f\ (\text{pred } n))$$

$fac\,2$를 연산하는 과정을 Y-결합자를 이용해 표현해라. ($Y\ (fac\ 2)$를 계산하라는 뜻이다.)

해설.

$$
\begin{aligned}
& Y\ fac\ 2 \\
&= fac\ (Y\ fac)\ 2 \\
&= (\lambda f.\lambda n.\text{if iszero } n \text{ then 1 else mult } n\ (f\ (\text{pred } n)))(Y\ fac)\ 2 \\
&= (\lambda n.\text{if iszero } n \text{ then 1 else mult } n\ ((Y\ fac)\ (\text{pred } n)))\ 2 \\
&= \text{if iszero 2 then 1 else mult 2 } ((Y\ fac)\ (\text{pred 2})) \\
&= \text{if iszero 2 then 1 else mult 2 } ((Y\ fac)\ (1)) \\
&= \text{mult 2 } ((Y\ fac)\ (1)) \\
&= \text{mult 2 } (fac\ (Y\ fac)\ (1)) \\
&\vdots \\
&= \text{mult 2 } (\text{mult 1 } ((Y\ fac)\ 0)) \\
&= \text{mult 2 } (\text{mult 1 } (fac\ (Y\ fac)\ 0)) \\
&\vdots \\
&= \text{mult 2 } (\text{mult 1 1}) \\
&= 2
\end{aligned}
$$

예제 3.77. 정의 3.46에서 정의한 최소화 함수 min을 람다식을 이용해 정의하라.

람다식을 이용해 최소화를 구현할 수 있으므로 μ-재귀 함수와 계산 능력이 동등하다.

정리 3.78. 람다 계산법와 튜링 기계의 계산 능력은 동등하다.

증명. (느슨한 증명) 먼저 람다 계산법은 μ-재귀 함수를 통해 튜링 기계를

흉내낼 수 있다. 튜링 기계가 람다 계산법를 흉내내는 과정은 자세히 다루진 않으나, 충분히 할 수 있을 거라고 믿자... □

3.8 연습문제

연습문제 3.1. Construct a 1-tape Turing Machine which accepts the given language L.

$$L = \{w \mid w \in \{0,1\}^*, N_0(w) > N_1(w)\}$$

($N_a(w)$ means the number of a in w.)

연습문제 3.2. Construct a 1-tape Turing Machine which converts a binary number to a unary number.(ex. $1011 \rightarrow 11111111111$)

연습문제 3.3. Let's think about a **restricted Turing Machine**. In the definition of Turing Machine in the textbook, the movement of head could have three choices(left, right, stay). Is it equivalent to the Turing Machine in the textbook if the head cannot stay and always have to move left or right?

연습문제 3.4. A pushdown automata only have one stack. A **k-stack machine** is a deterministic PA with k stacks. In one movement, the k-stack machine can (1) change to a new state and (2) replace the top symbol of each stack with ϵ or several stack symbols. The transition rule of k-stack machine looks like $\delta(q, a, X_1, \ldots, X_n) = (p, Y_1, \ldots, Y_n)$.
Show that if a language L is accepted by a Turing machine, then L is accepted by a two-stack machine.

연습문제 3.5. The **counter machine** has the same structure as the multistack machine, but each stack is a 'counter'. A counter can hold any nonnegative integer, but we can only distinguish the value of counter between zero and nonzero. In one move, counter machine can (1) change state and (2) add or substract 1 from any of its counters, independently. We do not think subtracting 1 from a counter that is currently 0.

1. Show that every recursive enumerable language is accepted by a three-counter machine.

2. Show that every recursive enumerable language is accepted by a two-counter machine.

연습문제 3.6. 최소화를 정의할 때 정의 3.46에서는 상수를 1로 두었다. 꼭 1이어야 할까? 다음과 같이 어떤 음이 아닌 정수 k여도 괜찮을까?

$$f(n_1, \ldots, n_k) = \{g(n_1, \ldots, n_k, m) = k\text{가 되게 하는 최소 } m\}$$

위와 같은 정의여도 μ-재귀 함수는 계산 가능한 함수와 동등한가?

연습문제 3.7. 정리 3.49의 증명에서 함수의 인자가 여러 개일 때의 증명을 완성하여라.

Part II

계산 가능성

CHAPTER 4

계산 가능성

우리는 지금까지 튜링 기계의 뛰어난 계산 능력에 대해 알아보았다. 튜링 기계는 정말 다양한 일을 할 수 있으며, 이러한 기초적인 개념에서 확장되어 현재 우리가 사용하는 컴퓨터가 되었다. 그런데 튜링 기계는 정말 모든 일을 할 수 있을까? 혹시 튜링 기계가 못하는 일이 있을까? 정말 아쉽게도 튜링 기계가 하지 못하는 일은 존재한다. 자, 이제 컴퓨터 과학의 시작이자 하이라이트를 맛볼 준비가 되었다.

4.1 튜링 기계의 부호화

먼저 튜링 기계를 0과 1만으로 잘 표현하는 방법에 대해서 알아보자. 다음 절에서 튜링 기계의 설계도를 입력으로 넣기 위해서 이런 작업을 한다. 혹시 이런 세세하고 귀찮은 과정에 관심이 없다면, '아니 튜링 기계는 테입 빼면 적당히 유한한데 당연히 알잘딱하게 되는거 아님???'라고 생각하고 넘어가도 무방하다.

일단 $\Sigma = \{0, 1\}, \Gamma = \{0, 1, \#\}$라고 하자. 제한된 언어로 계산 불가능함을

보이면 어차피 더 일반적인 문제는 당연히 계산이 불가능하기 때문에 문제는 없다.[19]

정의 4.1. 튜링 기계 $M = (Q, \{0,1\}, \{0,1,\#\}, \delta, q_1, H)$를 다음과 같은 방식으로 $\{0,1\}$ 상의 문자열으로 만들자.

1. $Q = \{q_1, \ldots, q_n\}$에 대해 q_i는 각각 0^i로 표시한다.
2. $\Sigma = \{0,1\}$에서 0과 1은 각각 0과 00으로 표시한다.
3. $\Gamma = \{0,1,\#\}$에서 0, 1, #는 각각 0, 00, 000으로 표시한다.
4. L, R, S는 각각 0, 00, 000으로 표시한다.
5. $\delta(q,a) = (p,b,d)$는 각 q, a, p, b, d의 부호들 사이에 1을 두고 차례로 나열한다. 즉, $\delta(q_1, 0) = (q_2, 1, L)$은 01010010010이다.
6. TM M의 부호는 전이의 부호들 사이에 11을 두고 사전 순서로 나열하고, 앞뒤에 111을 붙인것이다.
7. 초기 상태는 항상 q_1이다.
8. 정지 상태는 전이의 첫째 원소에 나타나지 않는 상태이다. 만약 무조건 결정되게 하고 싶다면 $H = \{y, n\}$인데, 이때 y는 두 정지 상태 중 사전 순서가 작은 것이고 n은 큰 것이다.

우리는 M을 적당한 문자열으로 대응시킨 것을 M의 **부호화**(encoding)라고 하고, $\langle M \rangle$으로 표시한다.

[19] 다른 알파벳이 있다고 하더라도 ASCII 코드처럼 적당히 부호화를 하면 된다.

예제 4.2. TM $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, \#\}, \delta, q_1, \{q_3\})$ 이고

$$\delta(q_1, \#) = (q_2, \#, R)$$
$$\delta(q_2, 0) = (q_2, 1, R)$$
$$\delta(q_2, 1) = (q_2, 0, R)$$
$$\delta(q_2, \#) = (q_3, \#, S)$$

이다. $\langle M \rangle$을 구하라.

$$11101000100100010011001010010010011$$
$$00100100101001100100010001000100111$$

튜링 기계 M에 들어가는 입력 문자열 w도 0과 1을 각각 0과 00으로 표시하고 글자 사이에 1을 넣어 부호 $\langle w \rangle$을 만들 수 있다. 즉, $\langle 0110 \rangle = 010010010$ 이다. $\langle M \rangle$과 $\langle w \rangle$를 접합한 것을 $\langle M, w \rangle$로 표시한다.

정의 4.3. 튜링 기계의 상황 $(q_0, u\underline{x}v)$는 다음과 같이 부호화 될 수 있다.

1. u와 v는 0과 1을 각각 0과 00으로 표시하고 글자 사이에 1을 넣어 부호화 할 수 있다.

2. q_0와 $\underline{x}$는 $\langle x \rangle 1 \langle q_0 \rangle$와 같이 표현하여 이를 11로 감싸고 q_i는 0^i, x는 0 또는 00으로 표현한다.

4.2 보편 만능 기계

컴퓨터가 지금까지 인류가 만들어온 기계와 다른 점은 무엇일까? 형광등은 불을 밝힐 수만 있다. 세탁기는 세탁만 할 수 있다. 하지만 컴퓨터는 이렇

게 책도 쓸 수 있고, 게임도 할 수 있고, 계산도 할 수 있다. 즉, 어디에서나 사용가능한 **보편 만능 기계**(universal computing machine)다. 이러한 보편 만능성은 튜링 기계가 다른 튜링 기계를 흉내낼 수 있다는 점에서 출발한다. 보편 만능 튜링 기계 M_u는 $\langle M, w\rangle$를 입력으로 받아서 M이 w를 가지고 돌린 후 그 결과를 출력하는 튜링 기계다.

정의 4.4. 보편 만능 튜링 기계 M_u는 3-테입 튜링 기계로 다음과 같이 정의된다.

1. 첫 번째 테입은 $\langle M, w\rangle$을 저장한다. 먼저, M이 제대로 된 튜링 기계의 부호화인지 점검한다.

2. 첫 번째 테입에 $\langle M\rangle$만을 남기고 $\langle w\rangle$를 두 번째 테입에 옮겨 적는다. 그리고 세 번째 테입에는 M의 시작 상태인 0을 적는다.

3. 세 번째 테입의 내용이 0^a이고 둘째 테입의 현재 글자가 0^b이면 첫 번째 테입에서 그에 해당하는 전이 (110^a10^b1)가 있는지를 확인한다. 그런 전이가 없으면 M은 정지하고, 있으면 그 전이를 이용해 두, 세 번째 테입의 내용을 적절히 바꿔준다.

4.3 정지 문제

이제 우리가 정말로 풀지 못하는 문제를 마주할 차례다. 그 전에 재귀 언어에 대해 알아보자.

정리 4.5. 재귀 언어 L의 여집합은 재귀이다.

증명. $L(M) = L$인 튜링 기계 M에 대해 y 상태와 n 상태를 서로 뒤집어 주면 된다. □

정리 4.6. 재귀 열거 언어 L에 대해 그 여집합 $\overline{L}$이 재귀 열거 언어이면 L은 재귀 언어이다.

증명. L과 $\overline{L}$가 둘 다 재귀 열거 언어이므로 각각 $L(M) = L, L(M') = \overline{L}$가 존재한다. 이때 입력 w에 대해 M과 M'을 동시에 돌려서 M이 멈추면 $w \in L$, M'이 멈추면 $w \notin L$임을 알 수 있으므로 L은 재귀 언어이다. □

이제 정지 문제를 정의해보자.

정의 4.7. 정지 문제(halting problem)란 프로그램 P와 입력 w가 주어졌을 때 P에 w를 입력했을 때 프로그램 P가 정지할지 묻는 문제이다. 이를 형식화해서 서술하면 다음과 같다.

$$L_h = \{\langle M, w\rangle \mid w \in L(M)\}$$

우리는 정지 문제가 재귀 언어가 아님을 보일 것이다. 이를 위해 다음을 정의하자.

정의 4.8. 언어 L_d는 다음과 같이 정의된다.

$$L_d = \{\langle M\rangle \mid \langle M\rangle \in L(M)\}$$

정리 4.9. $\overline{L_d}$는 재귀 열거 언어가 아니다.

증명. $\overline{L_d}$가 재귀 열거 언어이기 위해서는 $L(M) = \overline{L_d}$인 튜링 기계 M이 필요하다.

1. 먼저 $\langle M\rangle \in L(M)$이라고 하자. 이때 $\overline{L_d}$의 정의 상 $\langle M\rangle \notin \overline{L_d}$이어야 한다. 즉, 모순이다.

2. 이번엔 $\langle M \rangle \notin L(M)$ 이라고 하자. 이때는 반대로 $\overline{L_d}$의 정의 상 $\langle M \rangle \in \overline{L_d}$ 이어야 한다. 즉, 모순이다.

따라서 $L(M) = \overline{L_d}$ 인 튜링 기계 M은 존재하지 않는다. 즉, $\overline{L_d}$는 재귀 열거 언어가 아니다. □

왜 재귀 열거 언어가 아닌 언어가 존재할까? 이는 튜링 기계 자체가 유한하기 때문이다. 튜링 기계는 부호화가 가능하고, 이를 사전 순으로 나열하는 것이 가능하다. 즉, 자연수 집합과 일대일 대응이 가능하므로 크기가 자연수 집합과 같다. 그러나 언어는 $2^{\{0,1\}^*}$의 원소이다. 즉, 모든 언어를 모은 집합의 크기는 실수 집합과 같다. 칸토어의 대각선 논법으로 두 집합의 크기는 같지 않다는 것을 보일 수 있다.
그럼 이젠 세상에 튜링 기계로 푸는 시늉조차 못하는 문제가 있다는 것을 알게 되었다. 그럼 재귀 열거 언어지만 재귀 언어는 아닌 언어도 존재할까?

정리 4.10. L_h는 재귀 열거 언어다.

증명. 범용 튜링 기계를 이용하면 L_h를 판정할 수 있다. □

정리 4.11. L_d는 재귀 열거 언어다.

증명. 이 또한 범용 튜링 기계에 M을 입력할 때, $\langle M, M \rangle$을 입력하면 된다. □

정리 4.12. L_d와 L_h는 재귀 언어가 아니다.

증명. 먼저 L_d가 재귀 언어가 아님을 증명하자. L_d가 재귀 언어면 $\overline{L_d}$도 재귀 언어가 되므로 모순이다. 따라서 L_d는 재귀 언어가 아니다.
이제 L_h가 재귀 언어가 아님을 증명하자. L_h가 재귀 언어라면 $L(M) = L_h$이고 y, n을 판정할 수 있는 M이 존재한다는 건데, 그러한 M이 존재하면 M의 입력으로 $\langle M, M \rangle$을 넣으면 L_d를 판정할 수 있다. 즉, L_d가 재귀 언어라는 결론이 나오므로 모순이다. 즉, L_h는 재귀 언어가 아니다. □

4.4 정지 문제의 응용

정지 문제는 우리가 해결하고 싶어하는 컴퓨터 과학의 문제들에 대해서 '그건 불가능하다.'라고 말할 수 있게 해주는 아주 좋은 증명 도구이다. 우리는 정지 문제를 이용해서 다양한 문제를 증명할 것이기 때문에 그 문제를 적절히 변환시켜줄 필요가 있다.

정의 4.13. 언어 L_1에서 언어 L_2로의 **변환**(reduction)이란

$$f(w) \in L_2 \rightarrow w \in L_1$$

를 만족하는 계산 가능한 함수 f이다.

L_2라는 문제를 통해 L_1를 풀 수 있도록 입력 w를 적절히 바꿔주는 과정이 존재하면 이를 변환 가능하다고 한다. 즉, 우리는 이미 계산 불가능하다는 것을 알고 있는 L_1를 통해 L_2도 계산 불가능하다는 것을 보이고 싶을 때 변환을 사용한다.

정리 4.14. 다음과 같이 정의된 언어 L_ϵ이 계산 불가능함을 보여라.

$$L_\epsilon = \{\langle M \rangle \mid \epsilon \in L(M)\}$$

증명. 먼저 우리가 알고 있는 계산 불가능한 언어는 앞에서 정의한 L_h이다. 이제 L_ϵ을 계산할 수 있으면 L_h 또한 계산 가능함을 보일 것이다.
L_h의 입력 $\langle M, w \rangle$에 대해 $\langle M_w \rangle$을 출력하는 함수 f를 생각하자. M_w는 입력 ϵ이 주어지면 테입에 w를 쓰고 M을 흉내내는 튜링 기계다. 이러한 변환 f는 충분히 계산 가능한 함수다. 만약 $L(M_\epsilon) = L_\epsilon$인 M_ϵ이 존재하면 이를 통해 L_h를 계산할 수 있다. 그러나 L_h는 계산 불가능하므로 M_ϵ은 존재하지 않는다. □

언어 간의 변환을 조금 더 간단하게 생각하면 L_2을 계산하는 튜링 기계 M_2를 하나의 부품으로 생각해서 L_1을 계산하는 M_1을 만들어 낼 수 있다는 것을 보이면 된다.

정리 4.15. 다음과 같이 정의된 언어 $L_\emptyset$이 계산 불가능함을 보여라.

$$L_\emptyset = \{\langle M \rangle \mid L(M) = \emptyset\}$$

증명. 증명을 위해 우리가 알고 있는 계산 불가능한 언어 L_ϵ를 변환한다. L_ϵ의 입력 $\langle M \rangle$에 대해 $\langle M' \rangle$을 출력하는 함수 f를 생각하자. 이때 M'은 어떤 입력이 들어와도 이를 지우고 입력을 ϵ으로 만든 뒤 M을 실행시킨다. 이러한 변환 f는 충분히 계산 가능하다. 만약 M이 ϵ에 대해 정지하면 M'은 모든 입력에 대해 정지하고, 반대로 M이 ϵ에 대해 정지하지 않으면 M'은 모든 입력에 대해 정지하지 않는다. 만약 $L(M_\emptyset) = L_\emptyset$인 $M_\emptyset$이 존재하면 이를 통해 L_ϵ을 통해 계산할 수 있다. 그러나 L_ϵ은 계산 불가능하므로 $M_\emptyset$은 존재하지 않는다. □

예제 4.16. 다음과 같이 정의된 언어 L_r은 계산 불가능함을 보여라.

$$L_r = \{\langle M \rangle \mid L(M)\text{은 정규 언어}\}$$

이번에는 조금 더 실용적인 예제를 보자.

예제 4.17. 프로그래밍을 하다 보면 가끔 다음과 같은 오류를 마주한다. 다음 Python 코드를 보자.

```
x = 1
y = True
if x == y:
    print("Hello!")
```

위 코드를 컴파일 해보면 타입 오류(type error)가 발생할 것이다. Int형 변수인 x와 bool형 변수인 y를 비교하였기 때문이다. 물론 우리는 이러한 코드를 컴파일하기 전에 IDE에서 빨간줄 등을 통해 코드가 잘못되었다는 사실을 알려주기 때문에 이를 굳이 컴파일하지는 않는다. 그럼 과연 코드를 돌리기 전에 타입을 검사해주는 타입 검사기(type checker)는 항상 맞을까? 언제나 옳은 답[20]을 내놓는 타입 검사기는 존재할까? 이와 같은 타입 검사기가 존재하지 않는다는 사실을 증명해라.

해설. 다음 코드를 생각하자.

```
// M is a turing machine
// terminates(M, x) returns true
// when M halts for input x
// or returns false
if terminates(M, x):
    1 == True
else:
    print("Hello!")
```

만약 M이 정지한다면 이 코드는 타입 오류가 발생한다. 만약 정지하지 않는다면, 이 코드는 타입 오류가 발생하지 않을 것이다. 만약 완벽한 타입 검사기가 존재한다면 타입 오류를 검증할 수 있고, 즉 이를 통해 M이 x에 대해 정지하지 않는지를 알 수 있다. 하지만 이는 불가능하므로 완전한 타입 검사기는 존재하지 않는다.

그래서 요즘 사용하는 타입 검사기는 '안 되는 것 같아 보이지만 사실은 맞는 프로그램'을 검증하는 것을 포기한다. 위 증명에서 사용된 코드의 경우, 아마

[20] 타입 오류가 없으면 없다고 해주고, 있으면 있다고 해주는

python IDE에서 사용하는 타입 검사기의 경우 빨간 줄을 표시할 것이다. 즉, 최소한 타입 검사기가 된다고 했으면 무조건 타입에 있어서는 오류를 내뿜지 않는 프로그램이다.[21]
지금까지 알아본 예제들을 보면 뭔가 튜링 기계에 관련된 문제는 전부 계산 불가능한 것 같다. 이는 실제로도 맞다는 것을 보이자.

정리 4.18. (라이스 정리) C를 모든 재귀 열거 언어를 모은 집합의 공집합이나 전체 집합이 아닌 부분 집합이라고 하자. $L_C = \{\langle M\rangle \mid L(M) \in C\}$는 계산 불가능하다.

증명. 먼저 일반성을 잃지 않고 $\emptyset \notin C$라고 하자. 만약 그렇지 않다면 $\overline{C}$를 고려하면 된다. C는 공집합이 아니므로 $L \in C$인 언어 L을 고려하자. L은 재귀 열거 언어이므로 $L(M_L) = L$인 튜링 기계 M_L이 존재한다. L_C가 만약 재귀 언어라면 항상 정지하는 M_C가 존재한다. 이제 L_ϵ을 L_C로 변환하자. L_ϵ의 입력 $\langle M\rangle$을 $\langle M'\rangle$으로 바꿔주는 계산 가능한 함수 f를 생각하자. M'는 다음과 같은 일을 한다.

1. 먼저 주어진 입력 w를 따로 보관한다.

2. ϵ에 대해 M을 돌리고 M이 정지하면 w에 대해 M_L을 돌린다.

그러면 만약 M이 ϵ에 대해 정지하면 M'은 M_L과 동일한 결과를 가지게 된다. 즉, $L(M') = L$이다. 만약 M이 ϵ에 대해 정지하지 않으면 M'은 항상 정지하지 않는다. 즉, $L(M') = \emptyset$이다. $L \in C$이고 $\emptyset \notin C$이므로 M_C가 $L(M') = L$인지 아닌지 알려준다. 이를 통해 M이 ϵ에 대해 정지하는지 안 하는지 알 수 있으므로 모순이다. □

[21] 그렇다면 syntax error는 어떨까? 궁금하면 3장으로 돌아가자.

라이스 정리가 함의하는 바는 우리가 사실 과제로 제출해온 코드가 정말로 스펙에 맞는 코드인지 알 길이 없다는 뜻이다. 그래서 우리는 수많은 테스트 케이스를 넣어 '이만하면 맞겠지'를 검증하는 것 뿐이다.

4.5 계산 가능한 것들

그럼 우리가 지금까지 다뤄왔던 정규 언어와 문맥무관 언어는 결정가능할까? 다행히도 이는 웬만하면 계산 가능하다.

정리 4.19. 다음과 같이 정의된 언어 L_{DFA}는 계산 가능하다.

$$L_{\text{DFA}} = \{\langle D, w\rangle \mid \text{DFA } D\text{는 } w\text{를 받아들인다.}\}$$

증명. 다음과 같은 튜링 기계 M을 구성하자.

1. D에 입력 w를 넣고 이를 흉내낸다.
2. 최종 상태에 진입하면 받아들이고, 진입하지 않으면 받아들이지 않는다.

구체적인 구현은 적어두지 않았으나, 충분히 흉내낼 수 있다는 사실을 알 것이라 믿는다. □

NFA와 정규식에 대해서는 따로 적어두진 않으나, 자명하게 가능하다.

정리 4.20. 다음과 같이 정의된 언어 E_{DFA}는 계산 가능하다.

$$E_{\text{DFA}} = \{\langle D\rangle \mid D\text{는 DFA이고 } L(D) = \emptyset\}$$

증명. DFA D는 최종 상태에 진입하는 것이 가능하면 $L(D) \neq \emptyset$임을 알 수 있다. 즉, 튜링 기계 M은 너비 우선 탐색이나 깊이 우선 탐색과 같은 그래프 전체 탐색 알고리즘을 이용하면 최종 상태에 진입 가능한지 알 수 있다. □

정리 4.21. 다음과 같이 정의된 언어 EQ_{DFA}는 계산 가능하다.

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A, B\text{는 DFA이고 } L(A) = L(B)\}$$

증명. DFA A, B로부터 새로운 DFA C를 정의하자. $L(C)$는 다음과 같이 정의된다.

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

정리 1.21에 의해 $L(C)$는 정규 언어이다. 이때 $L(C) = \emptyset$인 것과 $L(A) = L(B)$는 동치인 것을 알 수 있다. 정리 4.20에 의해 $L(C)$는 계산 가능하므로, EQ_{DFA}는 계산 가능하다. □

즉, 정규 언어에 관련된 것들은 전부 계산 가능하다. 그럼 이제 문맥무관 언어에 대해서도 알아보자.

정리 4.22. 다음과 같이 정의된 언어 L_{CFG}는 계산 가능하다.

$$L_{\text{CFG}} = \{\langle G, w\rangle \mid G\text{는 문맥무관 문법이고 문자열 } w\text{를 생성한다.}\}$$

증명. 다음과 같은 튜링 기계 M을 구성하자.

1. G를 동등한 촘스키 표준형 문법 C로 변환한다.
2. CYK 알고리즘을 이용해 w가 생성되는지 확인한다.
3. 생성되면 받아들이고, 생성되지 않으면 받아들이지 않는다.

□

정리 4.23. 다음과 같이 정의된 언어 E_{CFG}는 계산 가능하다.

$$E_{\text{CFG}} = \{\langle G, w\rangle \mid G\text{는 문맥무관 문법이고 } L(G) = \emptyset\}$$

증명. 다음과 같이 튜링 기계 M을 구성하자.

1. G에 대해 알파벳 Σ에 마킹해둔다.
2. $A \to x_1 \dots x_k$와 같은 생성 규칙에 대해 $x_1, \dots x_k$가 전부 표시되면 변수 A도 마킹해둔다.
3. 2를 반복해서 전부 탐색했을 때 시작 변수가 마킹되면 받아들이지 않고, 마킹되지 않으면 받아들인다.

□

이제 문맥무관 문법 G_1, G_2에 대해 $L(G_1) = L(G_2)$임을 판정하는 것 또한 정규 언어처럼 계산 가능할 것 같지만, 실제로는 그렇지 않다. 정리 4.21에서 사용한 논리를 그대로 사용할 수 없는게 정규 언어는 여집합 연산에 대해 닫혀있지만, 문맥무관 언어는 그렇지 않기 때문이다. 이를 증명하기 위해 먼저 다른 유사한 정리를 알아보자.

정리 4.24. 다음과 같이 정의된 언어 A_{CFG}는 계산 불가능하다.

$$A_{\text{CFG}} = \{\langle G\rangle \mid G\text{는 문맥무관 문법이고 } L(G) = \Sigma^*\}$$

증명. 먼저 $L(M_A) = A_{\text{CFG}}$인 튜링 기계 M_A가 존재한다고 가정하자. 이 튜링 기계를 이용해서 L_h를 결정하고자 한다. $L(M_h) = L_h$인 튜링 기계 M_h와 그 입력 $\langle M, w\rangle$에 대해 M이 w를 받아들이지 않을 때에만 문맥무관

언어 G가 모든 문자열을 생성하도록 G를 만들자. G는 문맥무관 언어이므로 $L(P) = L(G)$인 내리누름 오토마타 P가 존재한다.
여기서 우리는 P가 M의 상황의 나열 $C_1, \ldots, C_n$에 대해 받아들여지는 상황의 나열, 즉 제대로 M의 전이 규칙에 따라 전이하여 정지 상태에 들어가는 상황이 아닌 문자열을 전부 받아들이도록 구성할 것이다. 이는 비결정론적으로 다음과 같이 4 갈래로 구성된다.

1. 첫번째 갈래는 입력이 제대로 형식을 갖추고 있는지 확인한다. 만약 형식이 맞지 않는다면(##의 꼴이 나온다던가) 받아들인다.

2. 두번째 갈래는 C_1이 제대로 $\langle M, w\rangle$의 상황인지를 확인한다. 만약 제대로 된 상황이 아니면 받아들인다.

3. 세번째 갈래는 C_n이 정지상태에 들어가는지 확인한다. 정지상태가 아니라면 받아들인다.

4. 네번째 갈래는 C_i에서 C_{i+1}로 제대로 전이 규칙을 따라 전이되었는지 확인한다. C_i를 읽을 때는 이를 스택에 삽입하고, 이를 C_{i+1}^R를 읽을 때 꺼내면서 상황을 비교한다.[22] 헤드 근처가 아니면 테입의 내용은 동일하므로 같은지만 검사하다가 헤드에 도달하게 되면 전이가 제대로 이루어졌는지 확인한다.

$$\# \underset{\rightarrow}{C_1} \# \underset{\leftarrow}{C_2^R} \# \underset{\rightarrow}{C_2} \# \underset{\leftarrow}{C_3^R} \# \ldots \# C_n \#$$

따라서 P를 통해 $\langle M, w\rangle$에서의 받아들여지는 상황의 나열이 존재성을 확인할 수 있으므로 L_h도 결정할 수 있다. 이는 모순이므로 A_{CFG}는 계산 불가능하다. □

[22] 확인을 용이하게 하기 위해 상황 문자열의 역으로 비교한다.

정리 4.25. 다음과 같이 정의된 언어 EQ_{CFG}는 계산 불가능하다.

$$EQ_{CFG} = \{\langle G_1, G_2\rangle \mid G_1, G_2\text{는 문맥무관 문법이고 } L(G_1) = L(G_2)\}$$

증명. (느슨한 증명) 먼저 $L(M_{EQ}) = EQ_{\text{CFG}}$인 튜링 기계 M_{EQ}가 존재한다고 가정하자. G'을 $L(G') = \Sigma^*$인 문맥무관 문법이라고 할 때, M_{EQ}에 입력을 $\langle G, G'\rangle$으로 주면 $L(G) = \Sigma^*$인지 계산 가능하다. 즉, EQ_{CFG}가 계산 가능하면, A_{CFG}도 계산 가능하다. 그러나, 정리 4.24에 의해 이는 모순이므로 EQ_{CFG}는 계산 불가능하다. □

4.6 촘스키 위계

잠시 계산에서 벗어나 우리가 지금까지 다뤄왔던 언어로 돌아와보자. 지금까지 다뤄왔던 많은 모델들 중 언어 모델은 다른 모델들과는 굉장히 이질적이다. 이 모델들은 모두 언어학자 **노암 촘스키**(Noam Chomsky)에 의해 제시되었다. 촘스키는 언어를 보다 형식적으로 기술하기 위해서 언어 생성 모델을 제시하였고, 이러한 모델은 수학자와 컴퓨터 과학자가 제시해 온 여러가지 계산 모델들과는 독립적으로 제시되어 왔었다. 촘스키는 이러한 모델을 이용해 인간의 언어 생성 방식을 기술하려고 하였고, 실제로도 이는 언어학에서 현대 통사론의 근간을 이루고 있다. 이후에 이러한 언어 생성 방식이 계산 모델이 계산하는 능력과 동일하다는 것이 밝혀졌고, 언어의 계산 능력적 위계가 제시되었다. 이는 표 4.1에서 확인할 수 있다. 여기서 우리가 아직 다루지 않았었던 문맥 유관 언어와 선형 유한 자동 기계는 다음과 같이 정의된다.

정의 4.26. 문맥 유관 문법(context sensitive grammar) $G = (V, \Sigma, S, P)$은

언어	오토마타	예시
정규 언어	유한 오토마타	$\{a^n \mid n \geq 0\}$
문맥무관 언어	내리누름 오토마타	$\{a^n b^n \mid n \geq 0\}$
문맥유관 언어	선형 유한 자동 기계	$\{a^n b^n c^n \mid n \geq 0\}$
재귀 열거 언어	튜링 기계	정지 문제

표 4.1

생성 규칙 $x \to y$가 다음과 같은 형식만을 만족한다.

$$\alpha A \beta \to \alpha \gamma \beta$$

여기서 $A \in V, \alpha, \beta \in (V \cup \Sigma \setminus \{S\}), \gamma \in (V \cup \Sigma \setminus \{S\})^+$이다. 이때 문맥무관 문법으로 정의된 언어 $L(G)$를 **문맥 유관 언어**(context sensitive grammar)라고 한다.

예제 4.27. $L = \{a^n b^n c^n \mid n \geq 1\}$를 만드는 문맥 유관 문법을 구하라.

해설. 예제 3.25 참조.

예제 4.28. $L = \{a^n b^n c^n d^n \mid n \geq 1\}$를 만드는 문맥 유관 문법을 구하라.

정의 4.29. 선형 유한 오토마타(linear bounded automata, LBA)란 테입의 길이가 입력 문자열의 길이 만큼 제한된 비결정론적 튜링 기계이다. 즉, 길이가 n인 입력 문자열에 대해 테이프 길이가 n인 것이다.

정리 4.30. 선형 유한 오토마타의 계산능력과 문맥 유관 언어의 계산 능력은 동일하다.

증명. (느슨한 증명) 문맥 유관 언어의 가장 큰 특징은 생성 규칙에 의해 문자열이 생성될 때 계속해서 문자열의 길이가 늘어난다는 점이다. 바꿔서

말하면, 생성되는 중간에 결국 최종 문자열의 길이보다 긴 공간을 사용할 일이 없다는 뜻이다. 따라서 직관적으로 선형 유한 오토마타로 이를 흉내낼 수 있다. 반대의 경우도 비슷하게 할 수 있다. □

예제 4.31. $L = \{a^n b^n c^n \mid n \geq 1\}$를 만드는 선형 유한 오토마타를 구하라.

예제 4.32. $L = \{a^n b^n c^n d^n \mid n \geq 1\}$를 만드는 선형 유한 오토마타를 구하라.

4.7 처치-튜링 논제

지금까지 현대 수학과 컴퓨터 과학의 가장 중요한 근간인 계산 가능성 이론에 대해 다루어보았다. 이러한 다양한 계산 모델이 나오게 된 배경에는 1931년에 발표된 괴델의 불완정성 정리에서 출발한다. 이 정리를 통해 더 이상 우리는 수학의 모든 문제를 풀 수 없다는 것이 증명되었다. 따라서 우리가 풀 수 없는 문제가 무엇인지를 알고자 하는 것은 매우 당연한 수순이었다. 이를 위해 1931년에 괴델과 에르브랑이 제시했고 우리가 3.6 절에서 다루었던 재귀 함수, 1933년에 처치가 제시했고 우리가 3.7 절에서 다루었던 람다 계산법, 1936년에 튜링이 제시했고 우리가 3.1 절에서 다루었던 튜링 기계가 각각 제시되었다.

재귀 함수는 수학에서 다루었던 함수의 측면에서 우리가 계산하지 못하는 함수가 무엇인지를 최소화의 근본적인 한계를 통해 제시하였다. 람다 계산법은 최대한 간단한 모델을 통해서 최소한의 규칙만으로 우리가 다룰 수 있는 계산을 제시하였다. 튜링 기계는 계산하는 과정에 집중하여 우리가 계산하지 못하는 것이 무엇인지를 제시하였다. 그리고 각각 모델에서 제시하는 계산 가능성은 서로 동일하다는 것 또한 증명되었다.

이러한 모델들과 계산 가능성이 시사하는 바는 처치-튜링 논제에 의해 기술될 수 있다.

논제 4.33. (처치-튜링 논제, Church-Turing Thesis) 다음 명제는 서로 동치다.[23]

- 자연수 상에서의 함수가 **효과적 방법**(effective method)으로 산술될 수 있다.
- 튜링 기계에 의해 계산가능한 함수다.

먼저 위 논제는 왜 '논제'일까? 첫 번째로, 이는 증명되지 않았다. 두 번째로, 사실 증명이 불가능할 수 있다. 왜냐하면 '효과적 방법'이 무엇인지 정확히 정의되지 않았다. 이 논제가 함의하는 바는 바로 우리 인간이 사고하는 방식은 결국 튜링 기계가 사고하는 방식과 같다는 것을 의미한다. 튜링은 앞에서 서술했듯이, 괴델의 불완전성 정리를 자신만의 방식으로 증명하기 위해 튜링 기계를 제시했다. 이는 결국 인간이 기술할 수 있는 수학은 결국 튜링 기계가 계산하는 방식과 동일하다는 것을 내포하고 있다. 혹자는 이를 '정의'로 취급하여, 아예 '알고리즘'이란 튜링 기계에 의해 계산가능한 함수로써 정의하기도 한다.
우리는 정말 많은 '상식적인 계산 모델'을 배웠지만 전부 튜링 기계의 한계를 벗어나지 못했다. 어쩌면 이러한 계산 모델들의 한계가 전부 동일하다는 것은 우리 인간의 사고의 한계가 여기까지라는 것을 보여주는 것이 아닐까?

4.8 바쁜 비버 함수

튜링 기계를 이용해 정의한 한 재미있는 함수에 대해 알아보자.

정의 4.34. 바쁜 비버 함수(busy beaver function)는 다음과 같은 튜링 기계에 의해 정의된다.

[23] 참고로 이 논제는 처치와 튜링 둘 다 딱히 말한 적이 없다...

1. 이 튜링 기계는 정지 상태를 포함하여 n 개의 상태를 가진다.
2. 이때 튜링 기계의 테입은 양방향으로 무한하다. (우리가 앞에서 정의한 튜링 기계와는 약간 다르다.)
3. 테입 알파벳 $\Gamma = \{0, 1\}$ 이다. 여기서 0은 공백문자이다.
4. 전이 함수 $\delta : (Q - H) \times \Gamma \to Q \times \Gamma \times \{L, R\}$

위와 같은 상태의 개수가 n 개인 튜링 기계 중 정지하는 것만 생각했을 때, 정지하고 난 뒤 테입에 적혀있는 1의 개수가 가장 많은 튜링 기계를 n 번째 **바쁜 비버**(busy beaver) 라고 한다. 이때 함수 $BB : \mathbb{N} \to \mathbb{N}$에 대해 $BB(n)$은 n 번째 바쁜 비버가 정지하고 난 뒤 테입에 적혀 있는 1의 개수로 정의된다. $BB(n)$를 **바쁜 비버 함수**라고 한다.

예제 4.35. 가능한 n-상태 튜링 기계의 개수는 $(4n + 4)^{2n}$ 이다.

정리 4.36. $BB(n)$은 계산 불가능하다.

증명. '정지하는' 튜링 기계만 생각해야 하는데, 어떤 튜링 기계가 정지하는지 안 하는지는 정지문제에 의해 자명하게 계산 불가능하기 때문이다. □

$BB(n)$은 매우 빠르게 증가하는 함수이며, 몇 개의 작은 값들을 제외한 $BB(n)$의 값은 밝혀지지 않았다. 지금까지 밝혀진 $BB(n)$의 값들은 그림 4.1에서 확인할 수 있다.

정리 4.37. ZF 공리계가 무모순이면 $BB(748)$가 얼마인지 증명할 수 없다.

증명. (느슨한 증명) 간단하게 증명하자면, ZF 공리계에서 만들어질 수 있는 모든 명제를 하나하나 적으면서, 이를 모두 증명하고 있는 748-상태 튜링 기계를 만들면 된다. 궁금하면 [14]를 확인해보자. □

n	$BB(n)$
1	1
2	6
3	21
4	107
5	≥ 47176870
6	$> 7.4 \times 10^{36534}$
7	$> 10^{10^{10^{10^{7}}}}$

그림 4.1

정리 4.38. 골드바흐 추측이 거짓이면 정지하는 27-상태 튜링 기계가 존재한다.

정리 4.39. 리만 가설이 거짓이면 정지하는 744-상태 튜링 기계가 존재한다.

즉, 바꿔 말하면 $BB(27)$이 얼마인지 구하는 것은 골드바흐 추측을 증명하는 것보다 어려운 일이라는 것이다.

정의 4.40. 정의 4.34에서 정지한 뒤 테입에 적혀 있는 1의 개수가 아니라 스텝 횟수로 생각하자. 이를 **미친 개구리**(fast frog)라고 하고, 함수 $FF : \mathbb{N} \to \mathbb{N}$에 대해 $FF(n)$은 n 번째 미친 개구리가 정지할 때까지의 스텝 수로 정의된다. $FF(n)$을 **미친 개구리 함수**라고 한다.

정리 4.41. $FF(n)$은 계산 불가능하다.

증명. 자명하다. □

정의 4.42. 차분한 오리너구리 함수(placid platypus function) $pp(n)$은 정의 4.34에서의 튜링 머신이 n 개 이상의 1을 출력하기 위한 최소한의 상태 갯수로 정의된다. 즉, 바쁜 비버 함수의 역함수이다.

정리 4.43. $pp(n)$은 계산 불가능하다.

증명. $BB(n)$이 계산 불가능하므로 당연히 $pp(n)$도 계산 불가능하다. □

$BB(n), FF(n)$, 아커만 함수는 너무 빠르게 증가해서 계산 불가능한 거라면, $pp(n)$은 오히려 너무 느리게 증가해서 계산 불가능하다.

정리 4.44. $\lim_{n\to\infty} pp(n) = \infty$이다.

증명. 자명하다. □

4.9 연습문제

연습문제 4.1. Let's define **Post Correspondence Problem**(PCP) precisely. An instance of the PCP is a collection P of dominos.

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\}$$

A match is a sequence $i_1, \ldots, i_n$, where $t_{i_1} \ldots t_{i_n} = b_{i_1} \ldots b_{i_n}$. The problem is to determine whether P has a match.

$$PCP = \{\langle P\rangle \mid P \text{ is an instance of the PCP with a match}\}$$

1. To prove that PCP is undecidable, we need to define a new problem, **Modified Post Correspondence Problem**(MPCP).

   $$MPCP = \{\langle P\rangle \mid P \text{ is an instance of PCP with a match that starts with the first domino}\}$$

   Show that MPCP is undecidable.

2. Show that PCP is undecidable.

CHAPTER 5

튜링 위계와 산술적 위계

지금까지는 풀기 가능한 문제(언어)들의 위계를 다루어보았으니, 이제는 풀기 불가능한 문제에 대한 위계를 세울 차례다. 지금까지 많은 '풀기 불가능한 문제'들을 알아보았다. 예를 들어, 어떤 프로그램이 멈출지 말지 결정하는 것은 불가능한 문제이고, 또 어떤 프로그램이 우리가 의도한 동작을 수행하고 끝마칠지 말지 결정하는 것 또한 불가능한 문제다. 만약의 후자의 문제를 풀 수 있는 가상의 무언가가 있으면 당연히 전자도 풀 수 있게 되므로 후자가 아마 전자보다 더 어려운 문제일 것이다. 그렇다면 이런 불가능한 프로그램들 사이에서도 위계가 존재할까? 이를 어떻게 표현할 수 있을까?

5.1 괴델 수

이 장에서는 오직 자연수에 대해서만 다룰 것이다. 그러나 현재까지 다뤄온 문자열이나 앞으로 다룰 여러 가지 정형 논리식(well-formed formula)은 자연수가 아니므로 이를 부호화 하는 과정이 필요하다. 정의 4.1처럼 적당히 잘 부호화 될 것이라 믿는다면 이 절은 넘겨도 좋다.

정의 5.1. 괴델 수(Gödel numbering)는 다음과 같이 어떤 논리식을 하나의 자연수로 나타내는 방식이다. 각 기호는 표 5.1 같이 하나의 자연수에 각각 대응된다.[24] 그럼 모든 논리식 ϕ는 수열 $(x_1, x_2, \ldots, x_n)$에 대응될 수 있다.

	기호	숫자
	0	1
	S	3
	$\neg$	5
	$\vee$	7
	$\forall$	9
	(	11
	)	13
숫자 변수	x_1	17
	x_2	19
	$\vdots$	$\vdots$
특성 변수	P_1	289
	P_2	361
	$\vdots$	$\ldots$

표 5.1

이때 수열 $(x_1, x_2, \ldots, x_n)$은 함수 enc에 의해 하나의 자연수에 대응된다.

$$enc(x_1, x_2, \ldots, x_n) = 2^{x_1} \cdot 3^{x_2} \ldots p_n^{x_n}$$

이때 p_n은 n번째 소수이다. 어떤 논리식 ψ의 괴델 수는 간단하게 $G(\psi)$로 나타낸다.

예제 5.2. $1 = 1$의 괴델 수를 구하라. (단, $=$의 괴델 수는 17으로 한다.)

[24] 이는 괴델이 사용한 부호화 방식이지만 당연하지만 다른 방식을 사용해도 전혀 문제는 없다.

해설. $1 = 1$은 $S\ 0 = S\ 0$과 같이 나타낼 수 있으므로 수열 $(3, 1, 17, 3, 1)$에 대응된다. 즉 괴델수는

$$enc(3, 1, 13, 3, 1) = 2^3 \cdot 3^1 \cdot 5^{17} \cdot 7^3 \cdot 11^1 = 69085693359375000$$

임을 알 수 있다.

정리 5.3. 논리식 ψ_1, ψ_2에 대하여

$$\psi_1 \neq \psi_2 \Rightarrow G(\psi_1) \neq G(\psi_2)$$

이 성립한다.

증명. 소인수분해의 유일성에 의해 자명하다. □

5.2 부분 재귀 함수

여기서 잠시 재귀 언어와 재귀 열거 언어를 함수의 관점에서 재정의하자.[25]

정의 5.4. 집합 A에 대한 **특성 함수**(characteristic function) χ_A는 다음과 같이 정의된다.

$$\chi_A(x) = \begin{cases} 0 & (x \notin A) \\ 1 & (x \in A) \end{cases}$$

정의 5.5. 집합 A에 대해 특성 함수 χ_A가 존재하면 A를 **재귀적**(recursive)이라고 한다.

[25] 4 장에서 정의했던 것들도 몇 개 재정의하였다.

정의 5.6. 어떤 집합 $A \subseteq \mathbb{N}$에 대해, A가 어떤 부분 재귀 함수의 정의역이면 A를 **재귀 열거**(recursive enumerate, r.e.)라고 한다.

정의 5.7. 튜링 기계를 괴델 수로 부호화하여 나열했을 때 e번째 튜링 기계를 P_e라고 한다. 여기서 e를 **색인**(index)이라고 한다.

정의 5.8. P_e에 의해 계산되는 부분 함수를 φ_e라고 한다. 변수가 n개면 $\varphi_e^{(n)}$이라고 한다.

정리 5.9. 모든 $\varphi_x^{(n)}$에 대해 $\varphi_x^{(n)} = \varphi_y^{(n)}$인 색인 y가 $\aleph_0$개 존재한다.[26]

증명. 튜링 기계에 의미없는 명령어를 계속해서 추가하면 된다. □

정리 5.10. (정상 표현 정리, Normal Form Theorem) 모든 φ_e에 대해 다음을 만족하는 술어 $T(e,x,y)$와 원시 재귀 함수 $U(y)$가 존재한다.

$$\varphi_e(x) = U(\mu y\,[T(e,x,y)])$$

여기서 $T(e,x,y)$를 **클리니 T-술어**(Kleene T-predicate)라고 한다.

정리 5.11. (열거 정리, Enumeration Theorem) 모든 $\varphi_e(x)$에 대해 $\varphi_z^{(2)}(e,x) = \varphi_e(x)$인 부분 함수 $\varphi_z^{(2)}$가 존재한다.

정리 5.12. (s-m-n 정리, s-m-n Theorem) 모든 $m, n \geq 1$에 대해 다음을 만족시키는 일대일 원시 재귀 함수 s_n^m가 존재한다.

$$\forall x, y_1, \ldots, y_m, \varphi^{(n)}_{s_n^m(x,y_1,\ldots,y_m)} = \lambda z_1, \ldots, z_n\,[\varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n)]$$

정의 5.13. 편의를 위해 $\langle x, y\rangle = \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$라고 하자.[27] 유사하게 $\langle x_1, \ldots, x_{n+1}\rangle = \langle\langle x_1, \ldots, x_n\rangle, x_{n+1}\rangle$로 정의할 수 있다.

[26] 자연수 집합의 크기만큼 존재한다와 같은 말이다.
[27] 연습문제 5.1 참조.

정의 5.14. $x, y, e < s$ 이고 y 가 튜링 기계 P_e 에 대해 s 번의 스텝 미만으로 결정되어 출력되는 결과값이면 $\varphi_{e,s}(x) = y$ 라고 하자. 이러한 y 가 존재하면 $\varphi_{e,s}(x)$ 는 **수렴한다**(converge) 고 하고, $\varphi_{e,s}(x) \downarrow$ 라고 쓴다. 그렇지 않으면 **발산한다**(diverge) 고 하고, $\varphi_{e,s}(x) \uparrow$ 라고 쓴다.

정의 5.15. 만약 $\varphi_{e,s}(x) \downarrow = y$ 를 만족시키는 s 가 존재하면 $\varphi_e(x) \downarrow = y$ 라고 쓴다.

예제 5.16. 다음 두 집합이 재귀임을 보여라.

1. $L_1 = \{\langle e, x, s\rangle \mid \varphi_{e,s}(x) \downarrow\}$

2. $L_2 = \{\langle e, x, y, s\rangle \mid \varphi_{e,s}(x) = y\}$

정의 5.17. 집합 A 가 어떤 부분 재귀 함수의 정의역이면 A 를 재귀 열거라고 한다.

정의 5.18. 집합 W_e 를 다음과 같이 정의하자.

$$W_e = \mathrm{dom}\varphi_e = \{x \mid \varphi_e(x) \downarrow\} = \{x \mid (\exists y)T(e, x, y)\}$$

이와 비슷하게 $W_{e,s}$ 는 다음과 같의 정의할 수 있다.

$$W_{e,s} = \mathrm{dom}\varphi_{e,s}$$

정의 5.19. 향후를 위해 여러가지 집합을 정의해두자.

$$\begin{aligned}
K &= \{x \mid x \in W_x\} \\
K_0 &= \{\langle x, y\rangle \mid x \in W_y\} \\
K_1 &= \{x \mid W_x \neq \emptyset\} \\
\mathrm{Fin} &= \{x \mid |W_x| < \infty\} \\
\mathrm{Inf} &= \{x \mid |W_x| \geq \aleph_0\} \\
\mathrm{Tot} &= \{x \mid W_x = \mathbb{N}\} \\
\mathrm{Con} &= \{x \mid W_x = \mathbb{N} \wedge \forall i, j, \varphi_x(i) = \varphi_x(j)\} \\
\mathrm{Cof} &= \{x \mid |\overline{W_x}| < \infty\}
\end{aligned}$$

정리 5.20. K는 재귀 열거이다.

정리 5.21. K는 재귀가 아니다.

정리 5.22. K_0는 재귀가 아니다.

정의 5.23. 집합 A, B에 대해 다음을 만족하는 재귀적인 함수 f가 존재하면 A는 B로 **다대일 환원 가능**(many-one reducible) 하다고 하거나 m-환원 가능하다고 한다.

$$\begin{aligned}
f(A) &\subseteq B \\
f(\overline{A}) &\subseteq \overline{B}
\end{aligned}$$

이를 $A \leq_m B$라고 적는다.

정의 5.24. $A \leq_M B$에 대해 재귀 함수 f가 일대일 함수면 A는 B로 **일대일 환원 가능**(one-one reducible) 하다고 한다. 이는 $A \leq_1 B$라고 적는다.

예제 5.25. $K \leq_1 K_0$ 임을 보여라.

정의 5.26. $\leq_m$와 $\leq_1$에 관련된 동치 관계들을 정의하자.

$$A \equiv_m B \Leftrightarrow A \leq_m B \wedge B \leq_m A$$
$$A \equiv_1 B \Leftrightarrow A \leq_1 B \wedge B \leq_1 A$$
$$\deg_m(A) = \{B \mid A \equiv_m B\}$$
$$\deg_1(A) = \{B \mid A \equiv_1 B\}$$

예제 5.27. $A \leq_m B$이고 B가 재귀적이면 A는 재귀적이다.

예제 5.28. $K \leq_1 \mathrm{Tot}$이다.

해설. 다음과 같은 함수 $\psi(x, y)$를 정의하자.

$$\psi(x,y) = \begin{cases} 1 & x \in K, \\ \text{정의되지 않음} & \text{o.w.} \end{cases}$$

$\psi(x, y)$는 자명하게 재귀 열거이다. s-m-n 정리에 의해 $\varphi_{f(x)}(y) = \psi(x, y)$인 일대일 재귀 함수 f가 존재한다.

정의 5.29. $A \subseteq \mathbb{N}$에 대해 모든 x, y에 대해 다음이 성립하면 A를 **색인 집합**(index set)이라고 한다.

$$x \in A \wedge \varphi_x = \varphi_y \Rightarrow y \in A$$

정의 5.19에서 정의한 집합들은 전부 색인 집합이다.

정리 5.30. 색인 집합 A가 공집합이나 $\mathbb{N}$가 아니면 $K \leq_1 A$이거나 $K \leq_1 \overline{A}$이다.

증명. 모든 y에 대해 φ_{e_0}가 정의되지 않도록 하는 e_0를 고르자. 만약 $e_0 \in$ 흠 p21인데 할까말까 □

정의 5.31. 재귀 열거 집합 A에 대해 모든 재귀 열거 집합 W_e에 대해 $W_e \leq A$이면 **1-완전**(1-complete)하다고 한다.

예제 5.32. K_0, K, K_1는 1-완전함을 보여라.

아쉽게도 위에서 정의한 다른 색인 집합들은 1-완전하지 않다는 것을 좀 뒤에 보일 예정이다.

정의 5.33. 재귀적인 일대일 함수 $f : \mathbb{N} \to \mathbb{N}$를 **재귀적 순열**(recursive permutation)이라고 한다.

정의 5.34. 재귀적 불변량(recursively invariant)이란 재귀적 순열 안에서 변하지 않는 집합의 성질을 말한다.

예제 5.35. 다음 중 재귀적 불변량인 것과 아닌 것을 골라라.

1. A는 재귀 열거다.
2. $|A| = n$
3. A는 재귀적이다.
4. $2 \in A$
5. A는 색인 집합이다.

해설. 1, 2, 3은 재귀적 불변량이고 4, 5는 아니다.

정의 5.36. 집합 A, B에 대해 $p(A) = B$인 재귀적 순열 p가 존재하면 A와 B는 **재귀적 동형**(recursively isomorphic)이라고 한다. 이는 $A \equiv B$라고 쓴다.

정리 5.37. (마이힐 동형 정리, Myhill Isomorphism Theorem) $A \equiv B \Leftrightarrow A \equiv_1 B$ 이다.

증명. 먼저 $(\Rightarrow)$는 자명하므로 넘어가자.
$(\Leftarrow)$를 생각하자. 함수 f를 통해 $A \leq_1 B$, g를 통해 $B \leq_1 A$를 안다고 하자. □

5.3 튜링 위계

지금까지는 '상식적으로' 다룰 수 있는 것들에 대해 알아보았다. 이제 그 너머로 나아가보자. 직관적으로, $y \in A$인지 아닌지를 이용해 $x \in B$를 계산할 수 있으면 우리는 B가 A 안에서 계산 가능하다고 하고, 이를 $B \leq_T A$라고 쓴다. 이를 엄밀하게 정의하고 이를 이용한 다양한 튜링 완전성을 알아보자.

정의 5.38. $A \subseteq \mathbb{N}$라고 하자. 부분 함수 ψ가 χ_A와 함께 정의 3.29와 정의 3.46에서 사용되는 함수 구성 방식을 사용해서 구성되면 A에 대해 **부분 재귀적**(partial recursive)이다라고 한다. 이를 줄여서 ψ가 A-부분 재귀라고 표기한다.

위와 같은 정의는 튜링 머신을 통해 새롭게 정의될 수 있다.

정의 5.39. 집합 A에 대해 A-**신탁 튜링 기계**(A-oracle turing machine) M은 **신탁**(oracle)이 붙어있는 튜링 기계이다. 여기서 신탁이란 자연수 x에 $\chi_A(x)$를 반환해주는 가상의 기계이다. 여기서 $L(M)$은

$$L(M) = \{x \mid M\text{이 입력 } x\text{에 대해 정지한다.}\}$$

으로 정의된다.

정의 5.40. 부분 함수 ψ에 대해 다음과 같은 정의를 알아보자.

1. 집합 A에 대해 다음과 같은 성질을 만족하는 A-신탁 튜링 기계 M이 존재하면 부분 함수 ψ가 A에 대해 **튜링 완전**(Turing complete)하다고 한다.

$$\forall x \forall y, \psi(x) = y \Leftrightarrow x \in L(M) \wedge M\text{의 출력값은 } y$$

이는 $\psi \leq_T A$와 같이 표기한다. 대응되는 튜링 기계로 ψ를 표현하고 싶을 경우 $\psi = \Phi_M^A$ 또는 $\psi = \{\ e\ \}^A$와 같이 표기하기도 한다.

2. M이 x에 대해 정지하지 않으면 $\psi(x)$는 **발산**(diverge)한다고 하고 $\psi(x)\uparrow$로 표기한다.

정리 5.41. 부분 함수 ψ가 집합 A에 대해 튜링 완전한 것과 A에 대해 부분 재귀적인 것은 동치이다.

증명. 책 [7]에 나와있다고 하는데 안 읽어봐서 모른다... □

이를 통해 논제 4.33에서의 처치-튜링 논제를 더 확장해서 A-부분 재귀적이라는 것은 A를 통해 효과적 방법으로 기술되는 것과 동치라고 말할 수 있다.

정리 5.42. (상대화된 열거 정리, Relativized Enumeration Theorem)

정리 5.43. (상대화된 s-m-n 정리, Relativized s-m-n Theorem)

정리 5.44. (상대화된 재귀 이론, Relativized Recursion Theorem)

정의 5.45. $x, y, e < s, s > 0$이고 $\{\ e\ \}^A(x) = y$가 $\hat{P}_e$에서 s 스텝보다 작은 스텝으로 계산되고 계산에서 s보다 작은 숫자들인 $z < s$만 사용되면 $\{\ e\ \}_s^A(x) = y$라고 적는다.

정의 5.46. 1. $W_e^A = \mathrm{dom}\{\ e\ \}^A$로 정의한다. 비슷하게 $W_{e,s}^A$도 정의할 수 있다.

2. $\Phi_e^A(x) = \{\ e\ \}^A(x)$로 정의한다. 비슷하게 $\Phi_{e,s}^A(x)$도 정의할 수 있다.

3. $\{\ e\ \}(x) = \{\ e\ \}^{\emptyset}(x)$로 정의한다.

정의 5.47. 1. 만약 $\{\ e\ \}^A$가 B의 특성 함수가 되도록 하는 e가 존재하면 B가 A에서 **재귀적**(recursive) 또는 A로 **튜링-환원가능**(Turing-reducible)하다고 한다. 이는 $B \leq_T A$라고 적는다.

2. 만약 $B = W_e^A$인 e가 존재하면 B가 A에서 **재귀 열거**라고 한다.

정리 5.48. $B \leq_T A$인 것과 B와 $\overline{B}$가 A에서 재귀 열거인 것은 동치이다.

정의 5.49. 1. 만약 $A \leq_T B$이고 $B \leq_T A$이면 $A \equiv_T B$이다.

2. A의 **튜링 차수**(Turing degree)는 다음과 같이 정의된다.

$$\deg(A) = \{B \mid B \equiv_T A\}$$

재귀 열거가 아닌 것들 중 K를 가장 처음 다루었는데, 이를 이용해서 튜링 기계를 더 확장해보자.

정의 5.50. 1. K^A는 다음과 같이 정의된다.

$$K^A = \{x \mid \Phi_x^A(x)\downarrow\} = \{x \mid x \in W_x^A\}$$

K^A를 A의 **널뛰기**(jump)라고 하고, 간단히 A'로 쓴다.

2. $A^{(n)}$는 A의 n 번째 널뛰기로 $A^{(0)} = A, A^{(n+1)} = \left(A^{(n)}\right)'$와 같이 정의된다.

정리 5.51. (널뛰기 정리, Jump Theorem)

1. A'는 A에서 재귀 열거이다.
2. $A' \not\leq_T A$
3. $B \leq_1 A'$이면 B는 A에서 재귀 열거이다.
4. A가 B에서 재귀 열거이고 $B \leq_T C$이면 A는 C에서 재귀 열거이다.
5. $B \leq_T A \Leftrightarrow B' \leq_1 A$
6. $B \equiv_T A$이면 $B' \equiv_1 A'$이다.
7. A가 B에서 재귀 열거인 것과 A가 $\overline{B}$에서 재귀 열거인 것은 동치이다.

증명. 흠 시발 언제 적지 p54 □

정의 5.52. $\mathbf{0}^{(n)}$는 다음과 같이 정의된다.

$$\mathbf{0}^{(n)} = \deg(\emptyset^{(n)})$$

예제 5.53. 다음과 같이 무한한 차수의 위계를 세울 수 있다는 것을 증명하라.

$$\mathbf{0} < \mathbf{0}' < \mathbf{0}'' < \cdots < \mathbf{0}^{(n)} < \cdots$$

예제 5.54. 다음을 증명하라.

$$\mathbf{0} = \deg(\emptyset) = \{B \mid B\text{는 재귀적이다.}\}$$

5.4 산술적 위계

정의 5.55. 집합 B가 재귀적이면 $\Sigma^0_0(\Pi^0_0)$ 안에 있다고 한다. 이를 $B \in \Sigma^0_0(\Pi^0_0)$이라고 쓴다.

정의 5.56. 1. $n \geq 1$에 대해 다음을 만족하는 재귀적 관계 R이 존재하면 B가 Σ_n^0 안에 있다고 한다.

$$x \in B \Leftrightarrow (\exists y_1)(\forall y_2)(\exists y_3) \dots (Qy_n)\ R(x, y_1, \dots, y_n)$$

n이 홀수면 $Q = \exists$, n이 짝수면 $Q = \forall$이다.

2. $n \geq 1$에 대해 다음을 만족하는 관계 R이 존재하면 B가 Π_n^0 안에 있다고 한다.

$$x \in B \Leftrightarrow (\forall y_1)(\exists y_2)(\forall y_3) \dots (Qy_n)\ R(x, y_1, \dots, y_n)$$

n이 홀수면 $Q = \forall$, n이 짝수면 $Q = \exists$이다. 각각에 대해 $B \in \Sigma_n^0, \Pi_n^0$과 같이 쓴다.

3. $B \in \Sigma_n^0 \cap \Pi_n^0$이면 $B \in \Delta_n^0$이다.

4. $B \in \cup_n(\Sigma_n^0 \cup \Pi_n^0)$이면 B를 산술적이라고 한다.

여기선 앞으로 $\Sigma_n^0, \Pi_n^0, \Delta_n^0$은 간단히 $\Sigma_n, \Pi_n, \Delta_n$으로 쓴다.[28]

정리 5.57. (양화사 축약 정리, Quantifier Contraction Theorem)

정리 5.58. (정규 표현 정리, Normal Form Theroem) 집합(언어) A가 재귀 열거인 것과 $A \in \Sigma_1$인 것은 동치이다.

증명. ($\Leftarrow$) A가 재귀 열거 언어이므로 A는 어떤 μ-재귀 함수 ϕ의 정의역이다. M

($\Rightarrow$) $A =$ □

[28] 저 위에 0 말고 다른 것이 붙는다면 상당히 '위험'해진다...

예제 5.59. $\mathrm{Fin} \in \Sigma_2$ 임을 보여라.

해설.

예제 5.60. $\mathrm{Cof} \in \Sigma_3$ 임을 보여라.

해설.

5.5 포스트 정리와 위계 정리

정의 5.61. Σ_n-완전

정리 5.62. (포스트 정리, Post's Theorem) 모든 $n \geq 0$에 대해 다음이 성립한다.

1. $B \in \Sigma_{n+1} \Leftrightarrow B$는 어떤 Π_n에서 재귀 열거이다.
2. $n > 0$에 대해 $\emptyset^{(n)}$는 Σ_n-완전이다.
3. $B \in \Sigma_{n+1} \Leftrightarrow B$는 $\emptyset^{(n)}$에서 재귀 열거이다.
4. $B \in \Delta_{n+1} \Leftrightarrow B \leq_T \emptyset^{(n)}$

증명. 1.

□

정리 5.63. (위계 이론, Hierarchy Theorem) 모든 $n > 0$에 대해 $\Delta_n \subset \Sigma_n$이고 $\Delta_n \subset \Pi_n$이다.

위계 이론은 그림 5.1과 같이 도식화 할 수 있다.

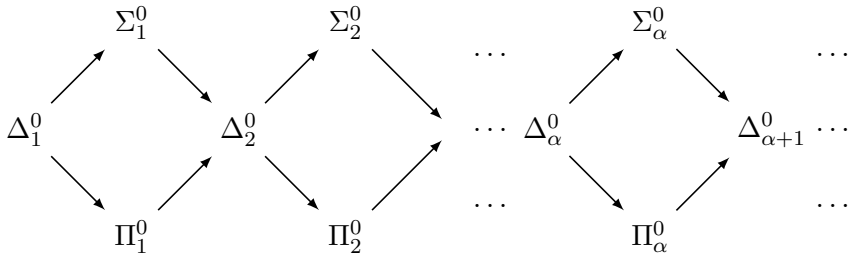


그림 5.1

5.6 그 너머의 계산 모델들

ㅋㅋ

5.7 연습문제

연습문제 5.1. 함수 $f(x,y) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$이 다음과 같이 정의된다고 하자.

$$f(x,y) = \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$$

함수 $f(x,y)$가 일대일 대응임을 보여라.

Part III

계산 복잡도

CHAPTER 6

시간 복잡도

6.1 점근적 표기법

지금까지는 어떤 문제가 계산할 수 있는 문제인지 알아봤다. 이제는 조금 더 현실적인 문제로 돌아올 차례다. 그럼 이제 풀 수 있는 문제들은 모두 풀 수 있을까? 다르게 말하면 '현실적으로' 푸는 것이 가능할까? 이에 대해 알아보기 위해 우리는 문제가 얼마나 어려운지 다룰 것이다.

정의 6.1. 항상 정지하는 결정론적 TM M에 대해 M의 **시간 복잡도**(time complexity)는 함수 $f : \mathbb{N} \to \mathbb{N}$로 나타낸다. 이때 $f(n)$은 입력 문자열의 길이 n에 대해 M의 최대 스텝(step)을 의미한다.

우리가 어떤 알고리즘(튜링 머신)의 시간 복잡도를 계산할 때 입력(입력 문자열)의 크기 n에 따라 정확히 몇 스텝인지 아는 것은 매우 어렵고, 크게 의미있지 않다. 따라서 우리는 개략적으로만 알고리즘의 복잡도를 분석하는데 이를 **점근적 분석**(asymptotic analysis)이라 한다. 우리는 아주 큰 입력에 대해서만 알고리즘을 고려할 것이므로 사사로운 것들에 구애받지 않아도

된다.

정의 6.2. 함수 $f, g : \mathbb{N} \to \mathbb{R}^+$에 대해 다음과 같은 성질을 만족하는 양수의 정수 c, n_0가 존재하면 $f(n) = O(g(n))$이라 한다. $n \geq n_0$에 대해

$$f(n) \leq cg(n)$$

이를 빅-O 표기법(Big-O notation)이라 하며, 이때 $g(n)$을 $f(n)$의 **점근적 상한** (asymptotic upper bound)이라 한다.

수학적으로 어렵게 적어놨지만, 시간 복잡도 $f(n)$에 대해 가장 압도적인 (dominate) 항만 계수를 빼고 생각하겠다는 뜻이다. 예를 들어 $f(n) = 3n^3 + 2n^2 + 72n + 3000$에 대해 n이 커지면 $\lim_{n\to\infty} \frac{f(n)}{n^3} = 3$으로 수렴가능하므로 $f(n) = O(n^3)$임을 알 수 있다.[29]

정의 6.3. 함수 $f, g : \mathbb{N} \to \mathbb{R}^+$에 대해 다음 성질을 만족하면 $f(n) = o(g(n))$이라 한다.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

이를 스몰-o 표기법(Small-o notation)이라 한다.

빅-O의 경우, 어떤 함수보다 작거나 같다는 걸 표현하기 위해 사용하고, 스몰-o의 경우, 어떤 함수보다 같지 않고 작다는 것을 표현하기 위해 사용된다.

예제 6.4. 다음 점근적 표기법이 올바르다는 것을 증명하라.[30]

1. $\frac{1}{100}n^2 + 1000000n + 20000000000 = O(n^2)$

[29] 정말 그럴까? 연습문제 6.1 참고.
[30] 사실 증명할 필요없고 그냥 맞다는 것을 느끼면 된다.

2. $\sqrt{n} = o(n)$

3. $n \log n = o(n^2)$

4. $3n \log n + 2n \log \log n = O(n \log n)$

5. $2^n = o(n!)$

6.2 복잡도 분석

정의 6.5. 함수 $t : \mathbb{N} \to \mathbb{R}^t$에 대해 **시간 복잡도 모임**(time complexity class)[31] $\mathrm{TIME}(t(n))$은 시간 복잡도가 $O(t(n))$인 튜링 기계로 결정할 수 있는 언어들의 모임이다.

예제 6.6. 언어 $L = \{0^n 1^n \mid n \geq 0\}$을 받아들이는 튜링 기계 M_1에 대한 점근적 상한을 구하라.

1. 입력 문자열 전체를 스캔하면서 1 오른쪽에 0이 있으면 받아들이지 않는다.

2. 0과 1이 테입에 남아있지 않을 때까지 3.을 반복한다.

3. 0이 나오면 이를 지우고, 1이 나올때까지 오른쪽으로 가다가 1이 나오면 이를 지우고 다시 왼쪽으로 가서 가장 왼쪽 0이 나올때까지 움직이다가 이를 지우고 다시 반복한다.

4. 만약 3 이후에 0만 남아 있거나 1만 남아 있으면 문자열을 받아들이지 않고, 만약 다 지워지면 이를 받아들인다.

[31] 모임이라고 적어뒀지만 결국엔 $2^{\{0,1\}^*}$의 부분 집합이므로 사실은 집합이 맞다. 그러나 $\{0,1\}^*$와 같은 언어 집합의 복잡도를 다루는 것이 아닌 더 깊고 이상한 복잡도를 다룰 경우, 집합이 아닐 수도 있다고는 하는데 필자도 잘은 모른다...

해설. 먼저 1의 경우 $O(n)$ 번의 스텝이 필요하다. 3의 경우, 각 문자열을 지우는데 약 $\frac{n}{2}$ 번의 스텝(즉, $O(n)$ 번)이 필요하는데 이를 n 번 반복해야 하므로, $O(n^2)$ 번의 스텝이 필요함을 알 수 있다. 4의 경우, $O(n)$ 번의 스텝이 필요하다. 따라서 $L(M_1) \in \mathrm{TIME}(n^2)$ 이다.

그럼 이를 더 향상시킬 수는 없을까? 다음 튜링 기계 M_2를 생각해보자.

예제 6.7. 언어 $L = \{0^n 1^n \mid n \geq 0\}$을 받아들이는 튜링 기계 M_2에 대한 점근적 상한을 구하라.

1. 입력 문자열을 전체를 스캔하면서 1 오른쪽에 0이 있으면 받아 들이지 않는다.
2. 0과 1이 남아있지 않을 때까지 3~4를 반복한다.
3. 입력을 스캔하면서 0과 1의 개수를 세서 홀수면 받아들이지 않는다.
4. 입력을 스캔하면서 0 오른쪽에 0이 있으면 오른쪽에 있는 0을 지우고, 1에 대해서도 동일한 작업을 한다. 이를 계속해서 반복한다.
5. 0, 1이 남아있지 않으면 받아들이고, 아니면 받아들이지 않는다.

해설. 여기서 가장 핵심이 되는 부분은 3~4 부분인데 한 번씩 수행할때마다 0, 1의 개수가 절반으로 줄어든다. 즉, $3 \sim 4$를 $O(\log n)$ 번 반복하고, 각각에 대해 입력을 스캔하므로 $O(n)$ 번의 스텝을 진행한다. 따라서 M_2의 수행시간은 $O(n \log n)$ 이다.

대신에 2-테입 튜링 기계를 사용하면 더 적은 스텝으로 판별할 수 있다.

예제 6.8. 언어 $L = \{0^n 1^n \mid n \geq 0\}$을 받아들이는 2-테입 튜링 기계 M_3에 대한 점근적 상한을 구하라.

1. 입력 문자열을 전체를 스캔하면서 1 오른쪽에 0이 있으면 받아 들이지 않는다.

2. 처음부터 읽으면서 0이 나올 때마다 두 번째 테입에 하나씩 0을 복사해 둔다. 이를 1이 나올때까지 반복한다.

3. 1이 나오면 1을 읽을 때마다 두 번째 테입에 있는 0을 하나씩 지운다. 이때 1을 다 읽기 전에 0이 지워지면 받아들이지 않는다.

4. 두 번째 테입에 0이 남아있으면 받아들이지 않고, 0이 없으면 이를 받아들인다.

해설. 각 과정에 대해 $O(n)$ 번 스텝을 수행하므로 M_3의 수행시간은 $O(n)$이다.

이처럼 앞 장에서 계산 가능성을 다룰 때는 어떤 튜링 기계던지 수행능력은 동일했으나, 계산 복잡도 이론에서는 그렇지 않다. 어떤 튜링 기계를 사용하는지에 따라 우리가 알고자 하는 복잡도가 차이나게 된다. 그럼에도 불구하고 결정론적인 튜링 기계들 사이에서 계산 복잡도는 '큰 차이'가 없다는 것을 보일 것이다.

정리 6.9. 함수 $t(n) \geq n$에 대해, 수행 시간이 $t(n)$인 k-테입 튜링 기계 M과 동등하고 수행시간이 $O\left(t^2(n)\right)$인 1-테입 튜링 기계 M'이 존재한다.

증명. 이는 우리가 앞 장에서 동등한 튜링 기계를 만드는 과정을 생각하면 된다. 먼저 적절한 k-테입 형태로 바꾸는 과정은 $O(n)$번의 스텝이 필요하다. 그 후 테입을 스캔하면서 헤드를 찾고 M의 전이를 흉내내야 하는 것을 반복하는데, 이것의 상한은 $O(t(n))$이다. 왜냐하면 원래 M에서 $t(n)$번 스텝을 밟게 되므로 테입에 적혀있는 문자열의 길이의 상한은 $t(n)$이기 때문이다. 따라서 M의 전이를 한번 흉내내는데에 걸리는 스텝은 $O(t(n))$이고, M의

수행시간은 $O(t(n))$ 이므로 M'의 전체 수행시간은 $O(n) + O(t^2(n))$ 이다. 이때 $t \geq n$ 이므로[32] M'의 수행시간은 $O(t^2(n))$ 이다. □

정리 6.10. 수행시간이 $t(n)$ 이고 테입 알파벳이 Γ 인 TM M에 대해 이와 동등하고 테입 알파벳이 $\Gamma' = \{0, 1, a, b\}$ 인 M'이 존재한다. 이때 M'의 수행시간은 $O(\log|\Gamma| t(n))$ 이다.

증명. 증명의 핵심 아이디어는 Γ의 알파벳을 a, b를 통해 부호화 시키는 것이다. 최소 $\log|\Gamma|$ 개의 알파벳을 통해 부호화 하는 것이 가능하므로, 이로 인해 어떤 정보를 읽는데 $\log|\Gamma|$의 스텝이 필요하다. 따라서 수행시간은 $O(\log|\Gamma| t(n))$ 이다. □

예제 6.11. 수행시간이 $t(n)$ 인 양방향 튜링 기계 M에 대해 이와 동등하고 수행시간이 $O(t(n))$ 인 1-테입 튜링 기계가 존재한다는 것을 증명해라.

위 정리들에 의해 우리가 알고 있는 '상식적인' 결정론적 튜링 기계의 수행시간이 다항시간이라면 가장 기본적인 형태인 동등한 1-테입 튜링 기계가 수행시간이 다항시간이라는 사실을 알 수 있다.
그럼 실제 알고리즘을 분석하는 데에 있어서는 어떤 모델을 사용할까? 현실의 컴퓨터는 정의 3.22에서 정의한 랜덤 접근 기계에 기반하므로 이 모델을 사용한다. 1-테입 튜링 기계와 가장 큰 차이라면 테입에 해당하는 레지스터(register)를 바로 접근할 수 있다는 것이다. 엄밀히 증명하지는 않으나, 이 또한 결정론적 튜링 기계의 한 종류이므로 RAM으로 다항시간내에 해결할 수 있는 문제는 1-테입 튜링 기계로도 다항시간 내에 해결이 가능하다.

정의 6.12. 항상 정지하는 NFA N에 대해 N의 **수행시간**(running time)은 함수 $f : \mathbb{N} \to \mathbb{N}$로 나타낸다. 여기서 $f(n)$은 N이 가질 수 있는 모든 경우에 대한 최대 스텝을 의미한다.

[32] 입력 테입을 다 읽지 않는 튜링 기계는 의미가 없으므로 $t \geq n$이라 가정해도 무방하다.

정리 6.13. 함수 $t(n) \geq n$에 대해 수행시간이 $t(n)$인 비결정론적 1-테입 튜링 기계와 동등하고 수행시간이 $2^{O(t(n))}$인 1-테입 튜링 기계가 존재한다.

증명. 정리 3.21에서 비결정론적 튜링 기계를 흉내내는 결정론적 튜링 기계를 만드는 과정을 생각하면 된다. 비결정론적 튜링 기계 N의 수행시간이 $t(n)$이라 하자. 그럼 우리는 앞 장에서 했듯이 N을 흉내내는 결정론적 튜링 기계 M을 만들 수 있다.
입력 문자열의 길이 n에 대하여 N이 가질 수 있는 비결정론적 계산 트리는 최대 $t(n)$이다. 트리의 각 정점에 대해서 가질 수 있는 최대 자식의 수가 b라고 하자. N의 전이 관계에서 전이하는 최대 상태의 개수를 잡으면 된다. 따라서 트리의 잎 정점의 수는 최대 $b^{t(n)}$이다. M이 N을 흉내낼 때 너비 우선 탐색을 하게 된다. 이때 트리의 총 정점 수는 $2b^{t(n)}$ 이하 이므로 $O(b^{t(n)})$이라 할 수 있다. 모든 정점을 탐색하는 시간은 $O(t(n))$이므로 M의 수행시간은 $O(t(n)b^{t(n)}) = 2^{O(t(n))}$이다.[33] □

6.3 P

n^3과 2^n을 생각해보자. $n = 1000$일 때, n^3의 경우 10억 정도로 그럭저럭 큰 숫자이지만, 2^n의 경우 우주의 원자수보다도 크다. 이처럼 지수적인 수행시간을 가지고 있는 알고리즘은 적당히 큰 입력에 대해서도 현실적인 시간 내에 해결하지 못한다는 문제가 있다. 따라서 우리는 다항시간 내에 풀 수 있는 문제들이 어떤 것인지 알아볼 필요가 있다.

정의 6.14. **P**는 1-테입 튜링 기계로 다항시간 내에 풀 수 있는 언어들의 모임

[33] 정말로? 연습문제 6.3로 가자.

이다. 이는 다음과 같이 나타낼 수 있다.

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k)$$

P를 정의하면서 우리는 다음과 같은 이점을 얻을 수 있다.

- 어떤 튜링 기계를 사용하던지 사사로운 것에 신경쓰지 않고 '대충' 다항시간 내에 풀리는 문제를 구분할 수 있다.
- 우리가 컴퓨터를 이용해 현실적으로 풀 수 있는 문제들에는 어떠한 것들이 있는지 알 수 있다.

예제 6.15. 문제(언어) **RELPRIME**은 다음과 같이 정의된다.

$$\mathrm{RELPRIME} = \{\langle x, y\rangle \mid x\text{와 } y\text{는 서로소이다.}\}$$

이때 RELPRIME $\in$ P임을 보여라.

해설. 유클리디안 알고리즘을 사용하면 당연하게 다항시간 내($O(n)$)에 해결이 가능하다.

예제 6.16. 문제 **PATH**는 다음과 같이 정의된다.

$$\mathrm{PATH} = \{\langle G, s, t\rangle \mid \text{방향 그래프 } G\text{에 대해 } s\text{에서 } t\text{로 가는 경로가 존재한다.}\}$$

이때 PATH $\in$ P임을 보여라.

해설. DFS 또는 BFS를 이용해 탐색하면 다항시간 내($O(|V| + |E|)$)에 해결이 가능하다.

예제 6.17. 문제 **SHORTPATH**는 다음과 같이 정의된다.

SHORTPATH = $\{\langle G, s, t, k\rangle \mid$ 가중치가 있는 그래프 G에 대해 s에서 t까지 가중치의 합이 k이하인 경로가 존재한다.$\}$

SHORTPATH $\in$ P임을 보여라.

해설. 벨만-포드 알고리즘을 사용하면 $O(n^3)$에 해결이 가능하다.[34]

6.4 NP

정의 6.18. 언어 A의 **검증자**(verifier)는 다음을 만족시키는 알고리즘(튜링 기계) V이다.

$$A = \{w \mid V\text{가 } \langle w, c\rangle\text{를 어떤 문자열 } c\text{에 대해 받아들인다.}\}$$

이때 w의 길이에 대하여 다항시간 내에 알고리즘이 종료하면 우리는 V를 **다항시간 검증자**(polynomial time verifier)라고 한다. 다항시간 검증자가 존재하면 우리는 A를 **다항시간 내에 검증 가능하다**(polynomially verifiable)라 한다.

예제 6.19. 헤밀토니안 경로(hamiltonian path)는 방향 그래프 G에서 모든 정점을 한 번씩만 방문하는 경로이다. 이때 문제 **HAMPATH**는 다음과 같

[34] 지금까지 나온 알고리즘들이 무슨 알고리즘인지 모르겠다면 구글에 검색해보고, 이해가 되지 않는다면 '적당히 잘 된다'라고 믿고 넘어가도 된다. 이 과목은 알고리즘 과목이 아니다.

이 정의된다.

$$\text{HAMPATH} = \{\langle G, s, t\rangle \mid G\text{는 방향 그래프이고 } s\text{에서 } t\text{까지 헤밀토니안 경로가 존재한다.}\}$$

HAMPATH는 다항시간 내에 검증 가능함을 보여라. 이때 $\overline{\text{HAMPATH}}$는 다항시간 내에 검증 가능할까? 예상해보라.[35]

정의 6.20. **NP**는 다항시간 내에 검증 가능한 언어들의 집합이다.

예제 6.21. 문제 **COMPOSITE**은 다음과 같이 정의된다.

$$\text{COMPOSITE} = \{n \mid \exists p, q > 1, n = pq\}$$

$\text{COMPOSITE} \in \text{NP}$임을 보여라.

예제 6.22. 문제 **LONGPATH**는 다음과 같이 정의된다.

$$\text{LONGPATH} = \{\langle G, s, t, k\rangle \mid \text{가중치가 있는 그래프 } G\text{에 대해 } s\text{에서 } t\text{까지 가중치의 합이 } k\text{이상인 경로가 존재한다.}\}$$

$\text{LONGPATH} \in \text{NP}$임을 보여라.

참고로 $\text{LONGPATH} \in \text{P}$인지는 아직 모른다.

정의 6.23. 비결정론적 시간복잡도 모임 **NTIME**은 다음과 같이 정의된다.

$$\text{NTIME}(t(n)) = \{L \mid L\text{은 비결정론적 튜링 기계로 } O(t(n)) \text{ 내에 결정되는 언어이다.}\}$$

[35] 증명하라는 뜻이 아니다.

정리 6.24. 어떤 언어가 NP라는 것은 그 언어가 어떤 비결정론적 튜링 기계로 다항시간 내에 계산 가능하다는 것과 동치이다. 즉, $\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$ 이다.

증명. $A \in \mathrm{NP}$라 하자. V는 A의 다항시간 검증자라고 하자. 먼저 어떤 비결정론적 TM N에 대해 다항시간 내에 A가 결정됨을 보이고 싶다. 다음과 같이 N을 구성하자.

1. 비결정론적으로 길이 n^k 이하의 문자열 c를 고른다.
2. V를 $\langle w, c\rangle$에 대해 돌린다.
3. V가 받아들이면 받아들이고, 아니면 받아들이지 않는다.

V는 다항시간 검증자이므로 N은 다항시간 내에 정지함을 알 수 있다.
이제 반대 방향을 검증하자. A가 비결정론적 TM N에 의해 다항시간 내에 결정된다고 할때, 다항시간 검증자 V는 다음과 같이 구성할 수 있다.

1. 앞에서 V를 흉내내는 과정을 생각하면 된다. 입력 $\langle w, c\rangle$에 대해 N을 w에 대해 c를 참고하면서 비결정론적 튜링 기계에서 다양한 상황 중 하나를 선택해 가면서 흉내낸다.[36]
2. 만약 이 N이 받아들이면 받아들이고, 아니면 받아들이지 않는다.

□

예제 6.25. 문제 **SUBSET-SUM**은 다음과 같이 정의된다.

$$\mathrm{SUBSET\text{-}SUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ 에 대해 } \sum y_i = t\text{를 만족하는 } \{y_1, \ldots, y_l\} \subseteq S\text{가 존재한다.}\}$$

[36] 정리 3.21에서 비결정론적 튜링 기계를 흉내내는 결정론적 튜링 기계를 생각하면 된다.

이때 SUBSET-SUM $\in$ NP임을 보여라.

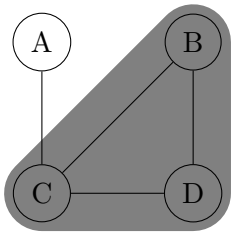
해설. 다항시간 검증자 V는 다음과 같이 만들 수 있다.

1. c의 원소의 합이 t인지 확인한다.
2. c가 S의 부분 집합인지 확인한다.
3. 만약 1, 2 둘 다 만족하면 받아들이고, 그렇지 않으면 받아들이지 않는다.

다른 증명으로는 다항시간 내에 작동하는 비결정론적 튜링 기계를 다음과 같이 만들면 된다.

1. $\langle S, t\rangle$에 대해 S의 부분 집합 c를 비결정론적으로 선택한다.
2. c의 원소들의 합이 t인지 검사한다.
3. 만약 통과하면 받아들이고, 그렇지 않다면 받아들이지 않는다.

예제 6.26. 클릭(clique)은 무방향 그래프의 부분 그래프 중 모든 정점이 서로 연결되어 있는 그래프를 의미한다. k-클릭(k-clique)은 정점이 k개인 클릭을 의미한다. 문제 **CLIQUE**는 다음과 같이 정의된다.



3-clique

그림 6.1

$$\text{CLIQUE} = \{\langle G, k\rangle \mid \text{무방향 그래프 } G\text{에 } k\text{-클릭이 존재한다.}\}$$

CLIQUE $\in$ NP임을 보여라.

이제 수학계 7대 난제인 P-NP 문제를 드디어 이해할 수 있게 됐다. 일단 P $\subseteq$ NP인 것은 당연히 자명하다. 그럼 P $\supseteq$ NP도 성립하는가? 아니면 P $\neq$ NP인가? 수많은 컴퓨터 과학자들의 노력이 있었으나 아쉽게도 이는 밝혀지지 않았다. 참고로 필자는 당연히 $P \neq NP$라고 믿는다.[37]

추측 6.27. (만인의 추측) $P \neq NP$이다.

6.5 NP-완전

우리가 상사에게 아주 어려운 문제를 풀라고 지시받았다. 아무리 풀려고 노력해봐도 답을 구할 수가 없을 때 상사에게 뭐라고 말해야 할까?

1. 그냥 자기가 멍청하다고 말한다.
2. 이 문제는 답이 없다고 말한다.
3. 다른 천재들도 이 문제를 풀어봤으나 그들도 구하지 못했다고 말한다.

1의 경우 멍청해보이고, 2의 경우 답이 없다는 걸 보이기 전까지는 무례한 대답이다. 3이 가장 설득력 있는 답안이라고 할 수 있는데, 이러한 관점에서 NP-완전라는 것을 정의하기로 한다.[38]

정의 6.28. 함수 $f : \Sigma^* \to \Sigma^*$에 대해 다항시간 내에 작동하고, $f(w)$를 테입에 적고 정지하는 튜링 기계 M이 존재하면 f를 **다항시간 내 계산 가능한 함수**(polynomial time computable function)라고 한다.

[37] 물리학자들이었다면 이미 이 명제는 공리화가 되었을 것이다.

[38] M. Garey와 D.Johnson의 책인 Computers and Intractability: A Guide to the Thoery of NP-Completeness에 있는 유명한 예제이다.

정의 6.29. 언어(문제) A에 대해 다음과 같은 조건을 만족시키면 A를 B로 **다항시간 내에 변환 가능**(polynomial time reducible)하다고 하고 $A \leq_p B$라고 표기한다.
모든 w에 대해 다음을 만족시키는 다항시간 내 계산 가능한 함수 $f : \Sigma^* \to \Sigma^*$이 존재한다.

$$w \in A \iff f(w) \in B$$

이러한 f를 A에서 B로의 **다항시간 변환**(polynomial time reduction)이라고 한다.

정리 6.30. $A \leq_p B$이고 $B \in P$이면 $A \in P$이다.

증명. M을 B를 다항시간 내에 결정하는 알고리즘, f를 A에서 B로의 다항시간 변환이라고 하자. 우리는 다음과 같이 다항시간 내에 A를 결정하는 알고리즘 N을 구성할 수 있다.

1. 입력 w에 대해 $f(w)$를 계산한다.
2. M을 입력 $f(w)$에 돌려서 받아들이면 받아들이고, 받아들이지 않으면 받아들이지 않는다.

□

예제 6.31. 리터럴(literal)이란 불 변수나 불 변수의 역을 의미한다. 이때 리터럴들이 $\vee$로 연결 된 것을 **절**(clause)이라고 한다. 이러한 절들이 논리곱 $\wedge$로 연결된 것을 **논리곱 정규형**(Conjunctive Normal Form, CNF)이라 한다. 각 절이 정확히 k개의 리터럴로 되어 있는 CNF를 k-CNF라 한다. 만약 부울식의 변수 값을 잘 할당하여 참이 될 수 있으면 **만족 가능**(satisfiable)하다고

한다. 예를 들어 $(a \vee b \vee \bar{c}) \wedge (d \vee \bar{e} \vee f)$는 3-CNF이고 만족 가능하다. 이때 문제 **3-SAT**는 다음과 같이 정의된다.

$$\text{3-SAT} = \{\langle \phi \rangle \mid \text{3-CNF } \phi\text{가 만족 가능하다.}\}$$

3-SAT $\leq_p$ CLIQUE임을 보여라.

정의 6.32. 모든 언어 $L \in$ NP에 대해 언어 A가 $L \leq_p A$이면 A를 **NP-하드**(NP-Hard)라 한다.

정의 6.33. 언어 A가 다음 두 가지 조건을 만족시키면 A를 **NP-완전**(NP-Complete)라 한다.

1. $A \in$ NP
2. $A \in$ NP-하드

정리 6.34. $A \in$ NP-완전이고 $A \in$ P면, P = NP이다.

즉, 어떤 NP-완전인 언어 A를 해결하는 다항시간 알고리즘을 발견한다면 무려 P=NP임을 증명할 수 있게 된다.[39]
그런데 어떤 언어가 NP-완전임을 보이는 것은 쉽지 않은 일이다. 왜냐하면 세상에 존재하는 모든 NP 문제를 어떤 언어 A로 다항시간 내에 환원 가능함을 보여야 하기 때문이다. 그러나 아주 놀랍게도 쿡과 레빈이 이 문제를 해결해 냈다.

정리 6.35. (쿡-레빈 정리) 언어 **SAT**는 다음과 같이 정의된다.

$$\text{SAT} = \{\langle \phi \rangle \mid \text{부울식 } \phi\text{가 만족 가능하다.}\}$$

[39] 당연하지만 어떤 NP 문제의 다항시간 알고리즘이 존재하지 않는다면 NP-완전이 아니어도 알아서 P $\neq$ NP임을 알 수 있다.

이때 SAT는 NP-완전이다.

증명. 먼저 SAT가 NP임은 자명하다. 모든 변수들의 T/F를 선택하여 만족하는지 확인하면 되기 때문이다.
이제 모든 언어 $A \in \text{NP}$에 대해 A에서 SAT로 다항시간 내에 변환 가능함을 보이자. A를 결정하는 비결정론적 튜링 기계 N에 대해 일반성을 잃지 않고 수행시간이 $n^k - 3$이라 가정하자.
이때 입력 w에 대한 N의 **도표**(tableau)는 $n^k \times n^k$ 크기의 표로써 각 행이 N

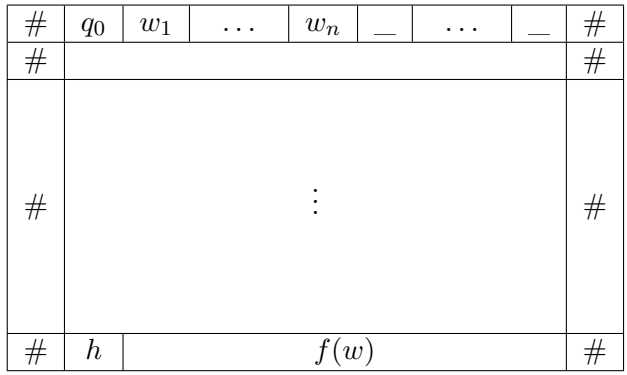


그림 6.2

의 계산의 갈래 중 하나의 상황을 보여준다. 첫 번째 행은 N에 입력 w이 들어왔을 때의 상황이다. 그 다음 행부터는 N의 그 전 행의 상황과 전이 관계에 의해 결정된다. 만약 도표의 한 행이라도 받아들이는 상황($\#h$와 같은 상황)이 된다면 도표가 **받아들여지는 도표**(accepting tableau)라고 한다. 따라서 N이 w를 받아들이는지에 대한 문제는 N에 대해 받아들여지는 도표가 존재하는지에 대한 문제와 같다. 이제 A를 SAT로 변환하는 다항시간 변환 f를 찾아보자. 입력 w에 대해 변환 f는 CNF ϕ를 만들어 낸다. $C = Q \cup \Gamma \cup \{\#\}$라 하자. $1 \le i, j \le n^k$와 $s \in C$에 대해 변수 $x_{i,j,s}$를 생각하자. 표의 한 칸을

뜻하는 **칸**(cell)에 대해 i행 j열에 있는 칸을 $cell[i,j]$라 하자. $x_{i,j,s} = 1$인 것은 $cell[i,j]$이 s를 갖는다는 뜻이다.
이제 ϕ를 정의해보자. ϕ는 다음과 같이 4가지 부분의 AND 연산으로 이루어진다.

$$\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

먼저 ϕ_{cell}에 대해 알아보자. ϕ_{cell}의 역할은 각 칸이 제대로 된 하나의 원소만을 담고 있는지 확인한다. 이는 다음과 같이 만들 수 있다.

$$\phi_{cell} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigvee_{s,t \in C, s \neq t} (\overline{x_{i,j,s}} \wedge \overline{x_{i,j,t}}) \right) \right]$$

$\bigvee_{s \in C} x_{i,j,s}$는 칸이 C의 원소를 담고 있는지 확인하고, $\bigvee_{s,t \in C, s \neq t} (\overline{x_{i,j,s}} \wedge \overline{x_{i,j,t}})$는 칸이 C의 원소를 하나만 담고 있는지 확인한다.
ϕ_{start}는 첫번째 행이 제대로 시작 상황인지 확인한다. 이는 다음과 같다.

$$\begin{aligned} \phi_{start} = & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\_} \wedge \ldots \wedge x_{1,n^k-1,\_} \wedge x_{1,n^k,\#} \end{aligned}$$

ϕ_{accept}는 받아들이는 상태가 있는지 확인한다. 이는 다음과 같다.

$$\phi_{accept} = \bigwedge_{1 \le i,j \le n^k} x_{i,j,q_{accept}}$$

마지막으로 ϕ_{move}는 도표가 N의 전이 관계에 따라 적절하게 생성되었는지 확인한다. 이를 확인하기 위해서 우리는 **창문**(window)을 활용한다. 우리는

2×3 창문을 이용해 전이 관계 규칙을 따르고 있는지, 즉 **적법**(legal) 한지 확인한다.
예를 들어, N의 전이관계 중 다음과 같은 규칙이 존재한다고 하자.

1. $\Delta(q_1, a) = \{(q_1, b, R)\}$
2. $\Delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$

a	q_1	b
q_2	a	c

(a)

a	q_1	b
a	a	q_2

(b)

a	a	q_1
a	a	b

(c)

$\#$	b	a
$\#$	b	a

(d)

a	b	a
a	b	q_2

(e)

b	b	b
c	b	b

(f)

표 6.1

표 6.1는 적법한 창문의 예시이다. (a)와 (b)는 전이 관계를 따르고 있으므로 적법하다. (c)는 q_1이 가르키고 있는 테입의 무슨 내용인지 모르나, 적절하게 움직이고 있으므로 적법하다. (d)는 내용이 같으므로 당연히 적법하다. (e)는 헤드가 보이지 않는 곳에서 왼쪽으로 이동하면서 q_2로 바뀌는 것이 전이 관계에 의해 가능하므로 적법하다. (f)는 오른쪽에 헤드가 보이지 않는 곳에서 b를 c로 바꾸는 것이 가능하므로 적법하다.
다음 표 6.2는 적법하지 않은 창문들의 예시이다. 왜 안 되는지 느끼면 된다. 어떤 두 붙어있는 행을 생각하자. 위 행의 칸 중 상태(헤드)와 붙어있지 않은 칸은 값이 변하지 않아야 한다. 그리고 상태(헤드)와 붙어있는 칸은 전이 관계에 따라 적절하게 값이 변해야 한다. 이 둘은 전부 창문을 통해 적법한지 확인할 수 있다. 따라서 귀납적으로 만약 첫 행이 시작 상황이고 모든 창문이 적법하다면 적절한 도표를 이룬다는 사실을 알 수 있다.

a	c	a
a	a	a

(a)

a	q_1	b
q_2	a	a

(b)

a	a	q_1
q_2	a	q_1

(c)

표 6.2

이제 ϕ_{move}를 만들어보자. ϕ_{move}는 창문이 적법한지 검사해서 도표가 적절히 전이 관계에 따라 구성되는지 확인한다. 각 창문은 총 6개의 칸로 이루어지는데 이는 고정된 숫자이므로 복잡도에는 영향을 주지 않는다.

$$\phi_{move} = \bigwedge_{1 \le i < n^k, 1 < j < n^k} ((i,j)\text{-창문은 적법하다.})$$

(i,j)-창문은 $cell[i,j]$가 창문에서 가운데 위에 있는 창문이다. 적법한 창문인지 검사하는 과정을 조금 더 엄밀하게 쓰면 다음과 같다.

$$\bigvee_{(a_1,\ldots,a_6)\text{는 적법}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

이제 이러한 변환이 다항 시간 내에 이루어짐을 보이자. 그러기 위해서 ϕ의 크기를 구해야 한다. 표는 n^{2k}개의 칸들로 이루어지고, 각 칸에서 가능한 문자의 개수는 튜링 기계 N에 의해 결정되므로[40] 변수의 개수는 $O(n^{2k})$이다. 이제 각 4가지 부분에 대해 크기를 구해보자. ϕ_{cell}는 각 칸에 대해 검사하므로 크기가 칸의 개수에 비례한다. 즉 크기는 $O(n^{2k})$이다. ϕ_{start}는 첫 행만 검사하므로 크기는 $O(n^k)$이다. ϕ_{move}와 ϕ_{accept}는 이와 비슷한 이유로 크기가 $O(n^{2k})$임을 알 수 있다. 따라서 ϕ의 전체 크기는 $O(n^{2k})$이므로 이는 다항시간 변환임을 알 수 있다.

[40] 상수이므로 무시할 수 있다는 뜻이다.

즉, 모든 NP 문제는 다항 시간 내에 SAT 문제로 환원가능하므로 SAT는 NP-완전이다. □

예제 6.36. 3-SAT는 NP-완전임을 보여라.

해설. 우리가 이미 알고 있는 NP-완전 문제인 SAT 문제를 적절히 변환하면 된다. 예를 들어, $(a_1 \vee a_2 \vee \ldots \vee a_n)$와 같은 절은 다음과 같이 새로운 변수 $z_1, \ldots, z_{n-3}$를 도입해 변형하면 된다.

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge \ldots \wedge (\overline{z_{n-3}} \vee a_{n-1} \vee a_n)$$

이는 당연히 다항시간 변환이다.

예제 6.37. CLIQUE는 NP-완전임을 보여라.

6.6 coNP

정의 6.38. 언어 A에 대해 $\overline{A} \in \text{NP}$이면 A는 **coNP**라고 한다.

예제 6.39. $\overline{\text{SAT}}$는 coNP 문제임을 보여라.

정의 6.40. 모든 언어 $L \in \text{coNP}$에 대해 언어 A가 $L \leq_p A$이면 A를 **coNP-하드**(coNP-Hard)라 한다.

정의 6.41. 언어 A가 다음 두 가지 조건을 만족시키면 A를 **coNP-완전**(coNP-Complete)라고 한다.

1. $A \in \text{coNP}$
2. $A \in$ coNP-하드

예제 6.42. 문제 **TAUTOLOGY**를 다음과 같의 정의하자.

$$\text{TAUTOLOGY} = \{\phi \mid \phi\text{는 항상 참인 명제이다.}\}$$

TAUTOLOGY는 coNP-완전임을 보여라.

예제 6.43. P = NP이면 P = coNP = NP임을 보여라.

6.7 최적화 문제

현재까지 다룬 모든 문제는 Yes/No만이 답인 문제였다. 그러나 우리가 다루는 문제가 항상 이런 것은 아니다. 현실세계에서 문제의 종류는 매우 다양하지만 이번 절에서는 '최적해'를 찾는 **최적화 문제**(optimization problem)에 대해 다룬다.

정의 6.44. (다항 시간 변환의 확장) 문제 A의 사례 α를 문제 B의 사례 β

6.8 연습문제

연습문제 6.1. 우리는 $O(g(n))$을 정의할 때 정의 6.2와 같이 극한을 사용하지 않고 정의하였다. 그런데 $o(g(n))$을 정의할 때는 정의 6.3와 같이 극한을 사용해서 정의했다. 그럼 $O(g(n))$을 정의할 때 다음과 같이 정의하면 안 되는 걸까?

$$f(n) = O(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \; (c \geq 0)$$

위 정의가 원래 정의 6.2과 동치인지 구하라.

연습문제 6.2. 예제 6.7에서 $L = \{0^n 1^n \mid n \geq 0\}$ 인 언어를 받아들이는 $O(n \log n)$ 인 튜링 기계를 제시했다. 그럼 이보다 더 빠른 수행시간을 가지는 알고리즘은 존재할까? 아쉽게도 불가능하다. 이를 증명하기 위해 $\mathrm{TIME}(o(n \log n))$의 원소는 정규 언어임을 증명해라.
이는 어려운 문제이므로 논문 [8]를 참고해라.

연습문제 6.3. $b \geq 1$에 대해 $O(f(n)b^{f(n)}) = 2^{O(f(n))}$ 임을 증명하라.

연습문제 6.4. 어떤 튜링 기계 M과 함수 $f(n) : \mathbb{N} \to \mathbb{R}$에 대해 $M = O(f(n))$ 임을 결정하는 것은 계산 불가능하다는 것을 증명해라.

연습문제 6.5. 정리 6.35인 쿡-레빈 정리의 증명에서 우리는 2×3 크기의 창문을 사용했다. 더 작은 창문을 사용하면 안 되는 걸까? 2×2 크기의 창문을 사용하면 증명이 가능할지 서술하라.

CHAPTER 7

공간 복잡도

우리는 지금까지 시간 복잡도에 대해 알아보았다. 시간도 중요한 자원이지만, 우리에게는 시간만 부족한 것이 아니다. 우리는 아쉽지만 한정된 돈으로 인해 제한된 메모리를 갖고 프로그램을 작성해야 하므로 우리가 만든 알고리즘이 얼마나 많은 공간을 차지하는지를 알아보자.

7.1 복잡도 분석

정의 7.1. 항상 정지하는 결정론적 TM M에 대해, M의 **공간 복잡도**(space complexity)는 함수 $f : \mathbb{N} \to \mathbb{N}$로 나타낸다. 이때 $f(n)$은 길이가 n인 입력에 대해 헤드가 테입에서 접근하는 칸의 최대 개수를 의미한다.

정의 7.2. 모든 상황에 대해 정지하는 비결정론적 TM M에 대해, M의 **공간 복잡도**는 M의 모든 상황 중 헤드가 접근하는 테입의 최대 개수를 의미한다.

공간 복잡도 또한 시간 복잡도처럼 점근적 표기로 표현하는 것이 유용하다.

정의 7.3. 함수 $f : \mathbb{N} \to \mathbb{R}^+$에 대해 **공간 복잡도 모임**(space complexity class)은 다음과 같이 정의된다.

- SPACE($f(n)$)은 공간 복잡도가 $O(f(n))$인 결정론적 튜링 기계로 결정할 수 있는 언어들의 모임이다.
- NSPACE($f(n)$)은 공간 복잡도가 $O(f(n))$인 비결정론적 튜링 기계로 결정할 수 있는 언어들의 모임이다.

예제 7.4. 정리 6.35에서 정의한 언어 **SAT**가 SPACE($O(n)$) 임을 보여라.

해설. 우리는 **SAT** 정도는 당연히 다항 시간 내에 풀지 못할 것이라고 믿고 있기 때문에 공간 복잡도도 어려울 것이라고 예상하지만, 공간은 시간과 달리 재활용해서 사용할 수 있다. 다음과 같은 튜링 기계 M을 구상하자.

1. 입력 $\langle\phi\rangle$에 대해 ϕ의 변수 $x_1, \ldots, x_k$의 진리 부여치를 테입에 적는다.
2. 1에서 적은 진리 부여치에 따라 ϕ의 값을 계산하고 만약 1이 나오면 받아들이고, 0이 나오면 다른 진리 부여치 값을 적은 뒤 이를 반복한다.
3. 만약 전부 0이 나오면 받아들이지 않는다.

M은 당연히 $O(k)$의 공간을 사용하고 $k < n$이므로 M은 $O(n)$에 속한다는 것을 알 수 있다.

예제 7.5. 다음과 같은 언어 ALL_{NFA}를 생각하자.

$$ALL_{\text{NFA}} = \{\langle A\rangle \mid A\text{는 NFA이고 } L(A) = \Sigma^*\}$$

$\overline{ALL_{\text{NFA}}} \in \text{NSPACE}(n)$ 임을 보여라.

시간 복잡도를 다룰 때는 비결정론적 튜링 기계를 결정론적 튜링 기계로 다루는 일은 매우 어려운 일이었다. 하지만 공간 복잡도의 관점에서는 훨씬 적은 공간만으로도 이를 다룰 수 있다.

정리 7.6. (사비치 정리, Savitch's theorem) $f(n) \geq n$인 함수 $f : \mathbb{N} \to \mathbb{R}$에 대하여 다음이 성립한다.

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}((f(n))^2)$$

증명. A를 결정하는 비결정론적 튜링 기계 N에 대해 $N \in \mathrm{NSPACE}(f(n))$이라고 하자. 이제 A를 결정하는 결정론적 튜링 기계 M을 구성하자. M을 만드는 데 있어서 가장 핵심적인 알고리즘인 CANYIELD을 구성하자. CANYIELD는 N이 어떤 상황에서 다른 상황까지 정해진 스텝 안에 진행될 수 있는지 판단하는 알고리즘이다. 즉, $\mathrm{CANYIELD}(c_1, c_2, t)$는 N의 상황 c_1에서 t번의 스텝 안에 c_2까지 진행되는 것이 가능한지 판단하는 알고리즘이다. 편의성을 위해 우리는 $t = 2^k$라고 가정하자. CANYIELD는 다음과 같이 구성될 수 있다.

1. 만약 $t = 1$이라면 $c_1 = c_2$이거나 N의 전이 관계 중 $c_1 \vdash_N c_2$가 되는 규칙이 있는지 찾고 있으면 받아들이고 없다면 받아들이지 않는다.

2. 만약 $t > 1$이라면 N의 각 상황 c_m에 대해 공간 $f(n)$을 사용하여 3을 계산한다.

3. $\mathrm{CANYIELD}(c_1, c_m, \frac{t}{2}), \mathrm{CANYIELD}(c_m, c_2, \frac{t}{2})$를 계산한 뒤, 둘 다 받아들이면 받아들인다.

4. 만약 3에서 모든 상황에 대해 받아들여지지 않으면 받아들이지 않는다.

이제 CANYIELD를 이용해 N을 흉내내는 M을 정의하자. 일단 먼저 편의성을 위하여 N의 계산이 끝나면 테입의 내용을 지우고 가장 왼쪽으로 헤드를 이동한 뒤 정지하는 상황 c_{accept}에 들어간다고 하자. 또한, 입력 w에 대해 N의 시작 상태를 c_{start}라고 하자. $n = |w|$라고 할 때, N이 $f(n)$ 만큼의 테입을 쓸 때 가질 수 있는 상황의 개수가 $2^{df(n)}$을 넘지 않도록 상수 d를 적절히 결정하자.[41] 그럼, N이 최대 $2^{df(n)}$ 스텝만큼만 움직이므로 M을 다음과 같이 구상할 수 있다.

1. 입력 w에 대해 CANYIELD$(c_{\text{start}}, c_{\text{accept}}, 2^{df(n)})$을 계산한다.

CANYIELD는 재귀적으로 구성되므로 이를 (c_1, c_2, t)를 저장하기 위한 스택이 필요하다. N은 최대 $O(f(n))$ 만큼의 테입을 사용하므로 (c_1, c_2, t)의 크기는 $O(f(n))$이다. $t = 2^{df(n)}$이라 했고, t는 재귀 한 번마다 절반이 되므로 스택의 최대 길이는 $O(\log 2^{df(n)}) = O(f(n))$이다. 따라서 총 필요한 스택의 크기는 $O((f(n))^2)$이다. [42] □

7.2 PSPACE

정의 7.7. PSPACE는 다음과 같의 정의된다.

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k)$$

NPSPACE도 정의하고 싶지만, 어차피 사비치 정리에 의해 NPSPACE = PSPACE이므로 따로 정의하지 않는다.

정리 7.8. P ⊆ PSPACE이다.

[41] 이는 테입 알파벳과 상태의 개수에 의존한다.

[42] 근데 문제가 있다. 여기서 우리는 이미 N이 $f(n)$ 만큼의 공간을 사용한다는 사실을 알고 있다고 가정하고 시작했다. 만약 이를 모르면 어떻게 할까? 연습문제 7.1로 가자.

증명. 시간보다 많이 공간을 사용할 수 없기 때문에 자명하다. □

정리 7.9. NP ⊆ PSPACE이다.

증명. 자명하다. □

현재까지 우리가 알아낸 바를 정리하면 다음과 같다.

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE}$$

7.3 PSPACE-완전

우리는 6.5 절에서 NP-완전을 정의함으로써 아마도 NP와 P가 같지 않을 것이라는 강력한 증거를 주었다. 여기서도 똑같이 PSPACE-완전을 정의하자.

정의 7.10. 다음을 만족할때 언어 B를 **PSPACE-완전**(PSPACE-complete)이라고 한다.

1. $B \in \mathrm{PSPACE}$
2. 모든 $A \in \mathrm{PSPACE}$에 대해 A는 B로 다항시간 변환 가능하다.

여기서 B가 2만 만족하면 B를 **PSPACE-하드**(PSPACE-hard)라고 한다.

여기서 왜 다항 공간 변환이 아니라 다항 시간 변환을 이용해서 정의하는 것일까? 완전 문제들은 어떤 복잡도 클래스에서 가장 어려운 문제들의 예시이다. 같은 복잡도 클래스 내 다른 문제들이 '쉽게' 그 문제로 변환될 수 있기 때문이다. 따라서 우리가 완전 문제를 '쉽게' 풀 수 있으면 다른 문제들도 '쉽게' 풀 수 있어야 한다. 따라서 변환은 원래 복잡도 클래스 그 자체보다 쉬워야 한다. 앞으로 완전 문제들을 몇 개 더 정의할 것인데, 완전 문제에서의 변환은 항상 그 복잡도 클래스와 비슷하거나 그보다 더 쉬울 것이다.

예제 7.11. 어떤 부울식 ϕ에 대해 부울식 내 모든 변수가 양화사 $\forall, \exists$에 의해 양화되면 이를 **전부 양화된 부울식**(fully quantified Boolean formula)이라고 한다. 예를 들어 다음 부울식은 전부 양화된 부울식이다.

$$\phi = \forall x \exists y[(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$

이때 문제 **TQBF**는 다음과 같이 정의된다.

$$\text{TQBF} = \{\langle \phi \rangle \mid \phi\text{는 전부 양화된 부울식이고 참이다.}\}$$

TQBF는 PSPACE임을 보여라.

정리 7.12. TQBF는 PSPACE-완전이다.

증명. 귀찮구나... □

7.4 L과 NL

우리는 지금까지 입력 스트링의 길이로 인해 $f(n) \geq n$인 경우만 봤다. 그러나 우리가 꼭 이러한 계산 모델만 생각할 이유는 없다. 예를 들어, 컴퓨터가 블루레이 디스크로 입력을 받는다고 하자. 그럼 컴퓨터의 메인 메모리보다 더 큰 크기의 입력이 들어오지만, 실제로 메인 메모리를 다 사용하지는 않을 것이다. 또한, 우리는 그 블루레이 디스크에 새로운 내용을 덧쓸 수 없고 오직 읽을 수밖에 없다. 이러한 경우를 고려하기 위해, 이제 계산 모델을 조금 수정해서 $f(n) < n$인 경우도 고려해보도록 하자.

정리 7.13. 읽기 전용 2-테입 튜링 기계는 다음과 같이 정의된다.

1. 첫 번째 테입에 입력 스트링이 들어오고 이 테입은 수정할 수 없다.

2. 두 번째 테입은 비어있고, 수정할 수 있다.

정의 7.14. 읽기 전용 2-테입 튜링 기계의 **공간 복잡도**(space complexity) $f(n)$은 길이가 n인 입력에 대해 헤드가 두 번째 테입에서 접근하는 칸 최대 개수를 의미한다.[43]

예제 7.15. 어떤 1-테입 튜링 기계 M과 이와 동등한 읽기 전용 2-테입 튜링 기계 M'에 대해 M이 PSPACE인 것과 M'이 PSPACE인 것이 동치임을 보여라.

앞으로 이 장에서 사용하는 계산 모델은 읽기 전용 2-테입 튜링 기계이다. $f(n) \geq n$인 경우는 별 다를 바가 없다는 것을 알았으니, $f(n) < n$인 경우를 알아보기 위해 다음을 정의하자.

정의 7.16. **L**은 다음과 같이 정의된다.

$$\mathrm{L} = \mathrm{SPACE}(\log n)$$

정의 7.17. **NL**은 다음과 같이 정의된다.

$$\mathrm{NL} = \mathrm{NSPACE}(\log n)$$

왜 $\sqrt{n}, (\log n)^2$ 같은 함수가 아니라 굳이 $\log n$으로 정의했는지 궁금할 수 있는데, 꽤나 많은 컴퓨터 과학적으로 의미있는 문제들이 $O(\log n)$ 공간에 풀리기 때문이다. 그 예들을 알아보자.

예제 7.18. 언어 $A = \{0^n 1^n \mid n \geq 0\}$은 L이다.

해설. 0과 1을 읽으면서 0과 1의 개수를 두 번째 테입에 이진수로 기록하면 된다. 그럼 $O(\log n)$만을 이용해서 결정할 수 있으므로, $A \in \mathrm{L}$이다.

[43] 첫 번째 테입은 어차피 읽기 전용이므로 고려하지 않는다.

예제 7.19. 예제 6.16에서의 PATH 문제에 대해, PATH $\in$ NL 임을 보여라.

해설. 귀찮구나.

7.5 NL-완전

추측 7.20. (만인의 추측) L $\neq$ NL 이다.

7.6 NL = coNL

7.7 연습문제

연습문제 7.1. 정리 7.6의 증명을 완성하라. (힌트: 튜링 기계가 시간을 완전히 포기하고 $f(n) = 1, 2, 3, \cdots$ 를 하나하나 차근차근 해보면 된다.)

CHAPTER 8

복잡도 위계

우리는 지금까지 $L \stackrel{?}{=} NL \stackrel{?}{=} P \stackrel{?}{=} NP$와 같은 아직 풀리지 않은 문제들에 대해 알아보았다. 이렇게 보면 복잡도 클래스끼리 서로 비교하는 것은 매우 어려운 일로 보인다. 하지만 분명히 직관적으로 $O(n^3)$에 소속되는 언어가 $O(n^2)$에 소속되는 언어보다 많을 걸로 예상되는데, 이 장에서는 그 직관이 맞다는 것을 증명한다.

8.1 공간적 위계

정리 8.1. (공간적 위계 이론, spcae hierarchy theorem)

증명. □

8.2 시간적 위계

정리 8.2. (시간적 위계 이론, time hierarchy theorem)

증명. □

APPENDIX A

잡다한 수학적 배경지식

여기선 중간에 필요한 수학적 내용들을 나열한다. 이미 내용을 알고 있다면 넘어가도 좋다. 또는 모르는 내용이나 용어가 나올 때마다 여기를 봐도 된다.

A.1 관계

정의 A.1. 집합 A, B, R에 대해 $R \subseteq A \times B$이면 R을 **관계**(relation)라고 한다. 이때, $R \subseteq A \times A$이면 R을 집합 A 위의 관계라고 한다. $(a, b) \in R$이면 $a \ R \ b$라고 쓴다.

예제 A.2. 집합 $A = \{1, 2, 3\}$에 대해 관계 $<$를 정의하라.[44]

$$\{(1, 2), (2, 3), (1, 3)\}$$

정의 A.3. 다음과 같이 집합 A위의 관계 R의 성질을 정의하자.

1. $\forall a \in A, (a, a) \in R$이면 R을 **반사적**(reflexive)이라고 한다.

[44] 우리가 아는 그 부등호다.

2. $\forall a, b \in A, (a, b) \in R \rightarrow (b, a) \in R$이면 R을 **대칭적**(symmetric)이라고 한다.

3. $\forall a, b, c \in A, (a, b) \in R \wedge (b, c) \in R \rightarrow (a, c) \in R$이명 R을 **추이적**(transitive)이라고 한다.

4. R이 반사적이고 대칭적이고 추이적이면 R을 **동치 관계**(equivalent relation)라고 한다.

정의 A.4. 관계 R의 **폐포**(closure) C란 $R \subseteq C$이면서 성질 P를 만족하는 최소의 관계이다. 예를 들어, 반사적 폐포, 대칭적 폐포, 추이적 폐포 등이 있다.

예제 A.5. 집합 $A = \{1, 2, 3\}$과 A 위에서 정의되는 관계 R은 다음과 같다.

$$R = \{(1, 2), (2, 2), (2, 3)\}$$

R의 반사적 폐포, 대칭적 폐포, 추이적 폐포, 반사적이고 추이적인 폐포를 각각 구하시오.

해설. 1. 반사적 폐포는 $\{(1, 1), (1, 2), (2, 2), (2, 3), (3, 3)\}$이다.

2. 대칭적 폐포는 $\{(1, 2), (2, 1), (2, 2), (2, 3), (3, 2)\}$이다.

3. 추이적 폐포는 $\{(1, 2), (1, 3), (2, 2), (2, 3)\}$이다.

4. 반사적이고 추이적인 폐포는 $\leq$이다.

A.2 기수

정의 A.6. $f : X \to Y$ 인 일대일 대응이 존재하면 두 집합 X, Y 의 **기수**(cardinality)(또는 크기)가 같다고 한다. 이는 다음과 같이 적는다.

$$|X| = |Y|$$

정의 A.7. $f : X \to Y$ 인 일대일 함수가 존재하면 집합 Y 의 기수가 X 보다 크거나 같다고 한다. 이는 다음과 같이 적는다.

$$|X| \leq |Y|$$

만약 $|X| \leq |Y|$ 인데 $|X| \neq |Y|$ 이면 $|X| < |Y|$ 라고 적는다.

정리 A.8. (칸토어 정리) 모든 집합 X 에 대해, $|X| < |2^X|$ 이다.

증명. 먼저 $f : X \to 2^X$ 인 함수라고 하자. 집합 $Y = \{x \mid x \in X \wedge x \notin f(x)\}$ 은 f 의 치역이 아니다. 만약 $f(z) = Y$ 인 $z \in X$ 가 존재하면, $z \notin Y$ 이면서 $z \in Y$ 이 되므로 모순이다. 즉, $|2^X| \neq X$ 이다.
이때, 함수 $g(x) = \{x\}$ 는 $g : X \to 2^X$ 인 일대일 함수이므로 $|X| \leq |2^X|$ 이다.
따라서 $|X| < |2^X|$ 이다. □

A.3 페아노 공리계

정의 A.9. 다음 성질들을 만족하는 집합 $\mathbb{N}$ 을 가리켜 **자연수**(natural number) 집합이라고 한다.

1. $0 \in \mathbb{N}$

2. $n \in \mathbb{N} \Rightarrow n^+\mathbb{N}$

3. $\nexists n \in \mathbb{N}, n^+ = 0$

4. $\forall m, n \in \mathbb{N}, m^+ = n^+ \Rightarrow m = n$

5. $\forall S \subseteq \mathbb{N}, 0 \in S$ 이고 $\forall n \in S, n^+ \in S \Rightarrow n^+ \in S$ 면, $S = \mathbb{N}$ 이다.

공리 A.10. 정의 A.9와 같은 자연수 집합이 존재한다는 공리계가 **페아노 공리계**(Peano's axioms) 다.

A.4 ZFC 공리계

공리 A.11. (선택 공리, axiom of choice) 임의의 집합 S에 대해 $\emptyset \notin S$ 이면 S의 원소인 각각의 집합에서 그 집합의 원소를 하나씩 뽑는 선택 함수(choice function) 가 존재한다. 이는 다음과 같이 적을 수 있다.

$$\forall S[(\emptyset \notin S) \Rightarrow (\exists f \in (\cup S)^S, \forall A \in S\ (f(A) \in A))]$$

A.5 괴델의 불완정성 정리

정리 A.12. (제 1 괴델의 불완전성 정리) 페아노 공리계를 포함하는 어떠한 공리계도 무모순인 동시에 완전할 수 없다. 즉 자연수 체계를 포함하는 어떤 체계가 무모순이라면, 그 체계에서는 참이면서도 증명할 수 없는 명제가 적어도 하나 이상 존재한다.

정리 A.13. (제 2 괴델의 불완전성 정리) 페아노 공리계가 포함된 어떠한 공리계가 무모순일 경우, 그 공리계로부터 그 공리계 자신의 무모순성을 도출할 수 없다.

매우 어렵게 들리지만, 간단하게 이야기하면 참임에도 불구하고 증명할 수 없는 명제는 항상 존재한다는 것이다. 그 예시로는 **연속체 가설**(continuum hypothesis) 등이 있다. 튜링은 수학의 추론 규칙들을 튜링 기계라는 기계적인 방식으로 환원하였고, 이를 통해 정지 문제와 같은 증명할 수 없는 문제들을 제시해냈다.

Bibliography

[1] Scott Aaronson. The busy beaver frontier. *ACM SIGACT News*, 51(3):32–54, 2020.

[2] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[3] Hendrik Pieter Barendregt. The lambda calculus, its syntax and semantics. revised edition edn. *Studies in Logic and the Foundations of Mathematics*, 103, 1984.

[4] Petr Hájek. Arithmetical hierarchy and complexity of computation. *Theoretical Computer Science*, 8(2):227–237, 1979.

[5] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[6] Thomas Jech. *Set theory: The third millennium edition, revised and expanded*. Springer, 2003.

[7] Stephen Cole Kleene. Introduction to metamathematics. 1952.

[8] Kojiro Kobayashi. On the structure of one-tape nondeterministic turing machine time hierarchy. *Theoretical computer science*, 40:175–193, 1985.

[9] Charles Eric Leiserson, Ronald L Rivest, Thomas H Cormen, and Clifford Stein. *Introduction to algorithms*, volume 3. MIT press Cambridge, MA, USA, 1994.

[10] Harry R Lewis and Christos H Papadimitriou. Elements of the theory of computation. *ACM SIGACT News*, 29(3):62–78, 1998.

[11] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

[12] Robert I Soare. A study of computable functions and computably generated sets. *Perspectives in Mathematical Logic, Springer-Verlag, Berlin*, 1987.

[13] Robert I Soare. *Recursively enumerable sets and degrees: A study of computable functions and computably generated sets*. Springer Science & Business Media, 1999.

[14] Adam Yedidia and Scott Aaronson. A relatively small turing machine whose behavior is independent of set theory. *arXiv preprint arXiv:1605.04343*, 2016.

[15] 문병로. 쉽게 배우는 알고리즘. 한빛미디어, 2018.

[16] 박근수. 오토마타 이론 강의록. `https://ocw.snu.ac.kr/node/2251`, 2008.

[17] 이광근. Snu 4190.310 programming languages lecture notes, 2006.