Theroy of Computation

서울대학교 컴퓨터공학부 이태영

June 30, 2023

Preface

이걸 쓰게 된 이유는 여러 이유가 있다.

1. 겨울방학에 스터디용으로 활용하기 위해서
2. 서울대학교 박근수 교수님의 강의록이 있기는 하나, '감동'적이지 않은 부분을 빼고 궁금증을 해소하기 위해 '감동'적인 내용을 보충하기 위해서
3. 계산 복잡도에 관련된 한국어 자료가 아예 전무해서 (물론 영어를 많이 섞을 것 같기는 하다.)
4. 언어학 내용도 함께 첨가하고 싶어서
5. 가장 근본적인 이유로, 뭔가 내 강의록? 책?을 써보고 싶어서

언제 완성될지는 모르겠으나 일단은 시작해보고자 한다.
참고 자료는

- 서울대학교 컴퓨터공학부 박근수 교수님의 오토마타 이론 강의록
- John E. Hopcroft의 Introduction to Automata Theory, Languages and Computation
- Michael Sipser의 Introduction to the Theory of Computation
- Sanjeev Arora의 Computational Complexity: A Modern Approach
- Harry R. Lewis의 Elements of the Theory of Computation
- Charles E. Leiserson의 Introduction to Algorithms
- 문병로 교수님의 쉽게 배우는 알고리즘

이 있다.

연습문제에 관하여

이 책의 문제들은 본문 중간의 '예제'와 각 장 끝의 '연습문제'로 구성된다.
예제는 본문의 내용을 이해하는 데 있어서 도움이 되는 문제들로 구성했다. 답이 적혀있는 경우는 답을 먼저 보고 내용을 이해하도록 도와주는 문제이다. 답이 없는 경우는 문제가 (1) 문제를 풀면서 고민하는 과정이 도움되거나 (2) 답이 너무 길어서 답을 작성하기 귀찮은 경우이다.
이 책의 연습문제의 난이도 표기법은 Donald E. Knuth의 The Art of Computer Programming에 기반한다. 이는 다음과 같다.

- 00 - 본문의 내용을 이해했다면 즉시 답을 낼 수 있는 아주 쉬운 문제.
- 10 - 본문의 내용을 좀 생각해야 하나 그렇다고 어렵지는 않은 간단한 문제.
- 20 - 본문 내용의 기본적인 이해를 시험하는 문제. 시험 문제에서 난이도 있게 출제될 수 있다.
- 30 - 꽤나 어려운 문제. 만족스럽게 풀려면 한 시간 이상 투자해야 할 수 있다.
- 40 - 매우 어려워서 학기말 과제나 텀 프로젝트로 적절하다.
- 50 - 필자를 포함한 많은 사람들이 답을 잘 모르는 문제. 혹시 풀게 된다면 먼저 답을 필자에게 알려주길 바란다.

그 사이의 17과 같은 등급은 '로그' 축적으로 보간한 것에 해당한다. 물론 이러한 난이도는 오직 필자의 주관적인 해석에 의해서만 평가된 것이므로 절대적인 난이도로 착각하지 않았으면 한다.

Contents

CHAPTER 1

Introduction

1.1 Motivation

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), 푸는게 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.

1.2 언어

일단 언어를 정의하기 위해 알파벳과 스트링을 정의하자.

- 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 앞의 예시만 사용한다.

- 알파벳 Σ상에서 스트링은 Σ 내에 있는 알파벳을 유한개 나열한 것이다. 예를 들어, 01001은 $\Sigma = \{0,1\}$ 상의 스트링이다. 주로 w로 표현한다.

- 스트링 w의 길이는 글자들의 개수이고 주로 $\|w\|$로 나타낸다.

- 길이가 0인 스트링을 ϵ이라 쓴다.
- 공스트링을 포함해서 알파벳 Σ 상의 모든 스트링의 집합을 Σ^*로 표시한다. 예를 들어, $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, \cdots\}$이다.
- 두 스트링의 접합(concatenation) $x \cdot y$는 x뒤에 y를 붙인 것이다. 주로 xy라 쓴다. 예를 들어 $x = 011, y = 110$이면 $xy = 011110$이다.
- 스트링의 역(reverse) w^R은 w를 역순으로 나열한 스트링이다. 예를 들어, $w = 011$이면 $w^R = 110$이다.

우리는 Σ^*의 부분집합을 **언어**(language)라 부른다.
그럼 이제 언어의 연산을 정의하자.

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

$$\{uv : u \in L, v \in L\}$$

  예를 들어, $L_1 = \{0, 00\}, L_2 = \{1, 11\}$ 일 때, $L_1L_2 = \{01, 011, 001, 0011\}$이다. 또한 $L^0 = \{\epsilon\}, L^k = L^{k-1}L(k \geq 1)$이다.

- L^*는 L을 0번 이상 접합하여 만들어지는 모든 스트링의 집합이다.

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

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

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

예제 1.1. 스트링 $w = 000111$이 $L = \{0^n1^n | n \geq 0\}$에 속하는지 구하시오.

CHAPTER 2

정규 언어와 유한 오토마타

2.1 정규 언어

먼저 정규식(regular expression)을 정의하자.

정리 2.1. 알파벳 Σ 상의 정규식과 그것이 표시하는 집합은 다음과 같다.

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

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

정리 2.2. 정규식으로 표현되는 언어를 **정규언어**(regular language)라 부른다.

예제 2.1. 길이가 짝수인 스트링의 집합을 표시하는 정규식은 $((0+1)(0+1))^*$ 이다.

예제 2.2. 1이 1개인 스트링의 집합을 표시하는 정규식은 0^*10^* 이다.

예제 2.3. $\{a, b, c\} \in \Sigma$에 대해 첫 번째 나온 알파벳이 다시 나오지 않는 스트링의 집합을 표시하는 정규식을 구하라.

예제 2.4. 111이 딱 한 번 나타나는 스트링의 집합을 표시하는 정규식을 구하라.

2.2 유한 오토마타

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

정의 2.1. 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. 최종상태들의 집합 $F \subseteq Q$

조금 더 구체적으로 들어가보자. 유한 오토마타는 CPU에 해당하는 유한 제어기, 입력장치인 입력 테입으로 구성된다. 입력 테입에는 Σ상의 스트링이 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.

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

스트링 $w \in \Sigma^*$에 대해 $\delta^*(q_0, w) \in F$면, DFA M이 w를 받아들인다고 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 스트링의 읽지 않은 부분으로 결정된다. 한 번의 전이에 의해 DFA M의 상황이 변화하는 과정을 $\vdash$로 나타낸다. 중간 전이를 생략하고 싶으면 $\vdash^*$로 나타낸다.

$$(q_0, 0011) \vdash (q_1, 011)$$

$$(q_0, 0011) \vdash^* (q_F, \epsilon)$$

유한 오토마타를 이해하기 쉽도록 방향 그래프로 나타낸다. 그림 2.1는 1이 홀수개 있는 DFA를 나타낸다.

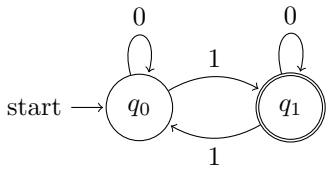


Figure 2.1

원이 두 개 그려진 상태는 최종 상태를 의미한다. 나머지는 잘 이해할 것이라 믿는다.

예제 2.5. 010을 부분스트링으로 갖는 스트링을 받아들이는 DFA를 구하라.

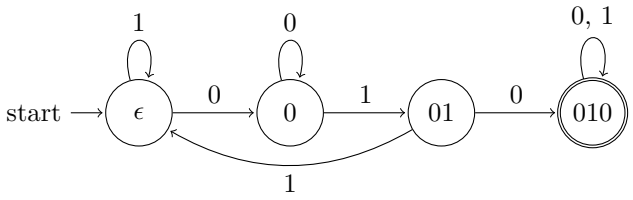


Figure 2.2

예제 2.6. 010을 부분스트링으로 갖지 않는 스트링을 받아들이는 DFA를 구하라.

예제 2.4에서 F와 $Q - F$를 서로 바꿔주면 된다. 이와 같이 DFA에서는 어떤 언어 집합의 여집합을 표현하는 것이 매우 편리하다. (한 번 예제 2.4, 예제 2.5를 정규식으로 표현해보자!)

예제 2.7. 000 또는 111을 갖지 않는 스트링을 받아들이는 DFA를 구하라.

예제 2.8. 000을 부분스트링으로 가지고 111을 부분스트링으로 갖지 않는 스트링을 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.)

2.3 비결정론적 유한 오토마타

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

정의 2.2. 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. 최종상태들의 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.
$\Delta^* : Q \times \Sigma^* \to 2^Q$ 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 스트링을 쭉 따라 갔을 때 갈 수 있는 모든 상태들의 집합으로 전이하는 함수라고 생각하면 된다.
스트링 w에 대해 $\Delta^*(q_0, w)$의 원소들 중에 F의 원소가 하나라도 있으면 M이 w을 받아들인다고 한다. 즉, $L(M)$은 M이 받아들이는 모든 스트링의 집합이다.

예제 2.9. $(010 + 01)^*$ 를 받아들이는 NFA를 만들어라.

그림 2.3을 보면 알겠지만, DFA와는 다르게 어떤 입력은 전이하는 edge가 없으며, 입력을 받지 않고서도 전이가 가능하다.
NFA의 최종상태는 하나로 만들 수 있다. $\forall q \in F$에 대해 (q, ϵ, q_F)를 넣어주면 된다.

예제 2.10. 끝에서 세 번째 글자가 1인 스트링을 받아들이는 NFA를 구하라.

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

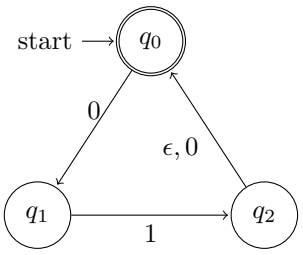


Figure 2.3

- 비결정성을 이용하면 쉽게 문제를 해결할 수 있는 경우가 많다. 예를 들어, 그래프 내의 최장 경로(longest path)를 찾는 문제를 생각해보자. 어떤 node에서 어떤 edge로 가야 최장 경로가 될 수 있는지는 deterministic하게 풀 경우 모든 edge에 대해 확인해봐야 하지만, nondeterministic하게 접근할 경우 모든 edge를 '동시에' 접근하는 것이 가능하므로 쉽게 문제를 해결할 수 있다. (멀리 안 가고 예제 2.9를 DFA로 만들려고 하면 좀 까다로울 것이다.)
- 우리가 풀어야 하는 문제의 도구는 결정론적이지만 문제의 세계에선 수많은 비결정성이 존재한다. 따라서 우리가 문제의 세계를 이해하기 위해서는 비결정성이라는 개념이 무조건적으로 필요하다.

2.4 DFA와 NFA는 동등하다!

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

정리 2.3. 임의의 NFA에 대해 이와 동등한 DFA가 존재한다.

증명. 컴퓨터과학에서 '존재'함을 보이려면 어떤 NFA에서 동등한 DFA를 만드는 과정을 보이면 된다.

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, 이 때 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 pesudo-code로 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. (더이상 증명이 '감동'적이지 않기 때문.) □

예제 2.11. 예제 2.8의 NFA를 DFA로 만들어라.

먼저 원래의 NFA를 생각하자. 시작 상태는 q_0이다. q_0에서 0을 입력받으면 q_1로 가고, 1을 입력받으면 dead state로 가게 된다. q_1에서 0을 입력받으면 dead state로 가게 되고, 1을 입력받으면 q_2, q_0으로 동시에 가게 된다.(빈 스트링에 의한 전이도 가능하므로) 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 dead state로 간다. 이를 계속해서 반복하고, F_N의 원소가 하나라도 있는 상태를 최종 상태로 만들어주면 그림 2.4을 얻을 수 있다.

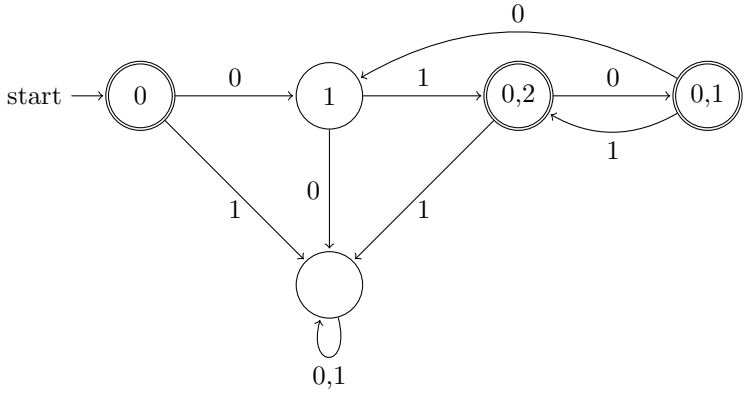


Figure 2.4

예제 2.12. 예제 2.9의 NFA를 동등한 DFA로 바꾸어라.

2.5 정규 언어 = DFA = NFA

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

정의 2.3. 정규식 r에 대해 $L(r)$을 받아들이는 NFA가 존재한다.

증명. 정의 2.1의 각 요소들을 만족시키는 NFA를 만들면 된다.

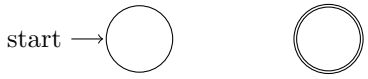


Figure 2.5

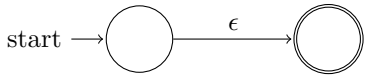


Figure 2.6

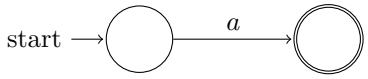


Figure 2.7

그림 2.5, 2.6, 2.7이 나타내는 NFA는 각각 $\emptyset, \epsilon, \{a\}$를 의미한다. 또한, 그림 2.8,

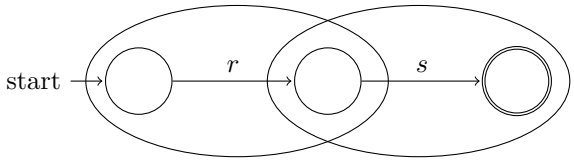


Figure 2.8

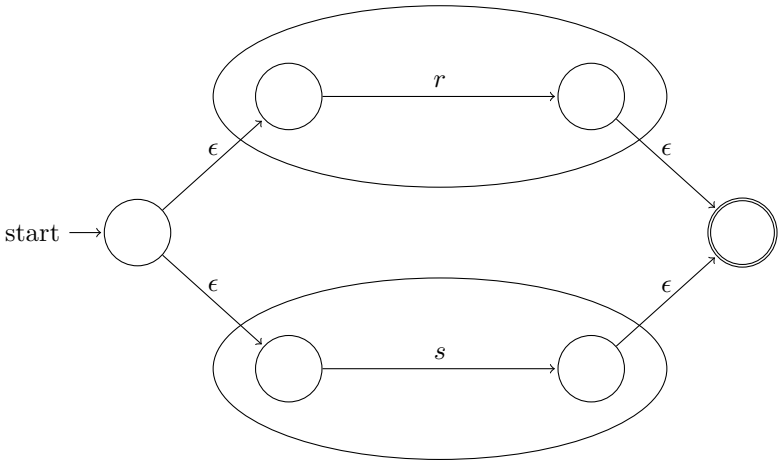


Figure 2.9

2.9, 2.10는 각각 $rs, r + s, r^*$를 의미한다.

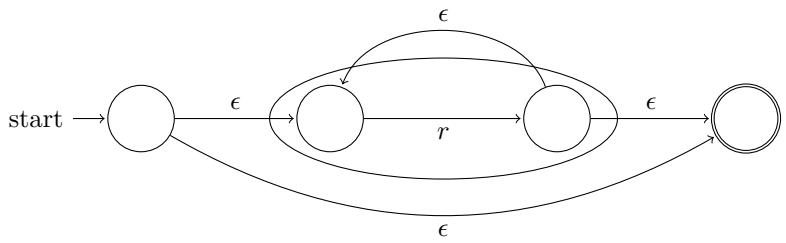


Figure 2.10

예제 2.13. 정규식 $(0+11)^*$를 받아들이는 NFA를 구하라.

정리 2.4. DFA M에 대해 $L(M)$을 표시하는 정규식이 존재한다.

증명. '감동'적이지 않으므로 자세한 내용은 생략한다. 그냥 믿자...

DFA의 상태에 대해 각각 번호를 1부터 k까지 부여하고(시작 상태는 무조건 1번), 경로 내에서 k 번호 이하의 상태만을 지나오면서 i번에서 j번으로 가는 모든 스트링의 집합을 $R_{i,j}^k$라고 하자. 이는 다음과 같이 재귀적으로 구할 수 있다.

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

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

2.6 정규언어의 특징

정리 2.5. 정규언어는 (1) 합집합, (2) 접합, (3) Kleene 곱$(*)$, (4) 여집합, (5) 교집합에 대해 닫혀있다.

증명. (1) $\sim$ (3)은 당연하다.

(4) 정규언어 L을 받아들이는 DFA는 존재하므로, DFA의 최종상태 F를 $Q-F$로 대체하면 된다.

(5) $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ □

예제 2.14. (소속문제) 정규언어 L에 스트링 w이 속하는지 결정하라.

L을 표시하는 DFA를 만들고 이를 읽어서 결정하면 된다. 아직은 매우 간단하지만 다른 언어 집합을 배우면 소속 문제는 꽤나 어렵거나 심지어는 풀 수 없는 (!) 문제라는 것을 알게 될 것이다.

예제 2.15. $L_1 \cup L_2$가 정규언어이고 L_1이 유한 언어집합이라고 하자. L_2가 정규언어인지 아닌지 증명하라.

2.7 정규언어가 아닌 것들

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

정리 2.6. (펌프 정리) L을 무한 정규언어라고 하자. 이때 다음을 만족하는 어떤 양의 정수 t가 존재한다. 길이가 t 이상인 임의의 스트링 $w \in L$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. 정규언어 L을 받아들이는 DFA M에 대해 M의 상태의 개수를 t라 하자. $w = a_1 \cdots a_n$ $(a_i \in \Sigma, n \geq t)$에 대해 $\delta^*(q_0, a_1 \cdots a_i) = q_i$ 라고 하자. 이때 M은 t개의 상태만을 가지므로, $q_j = q_k$ $(0 \leq j < k \leq t)$가 존재한다. 즉, $y = a_{j+1} \cdots a_k$라고 하면 $xy^iz \in L$이므로 펌프 정리를 증명할 수 있다. □

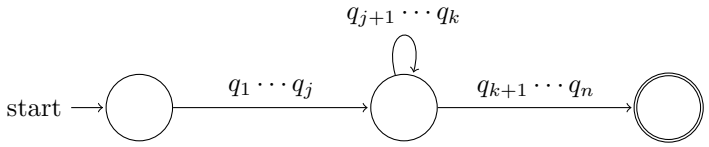


Figure 2.11

펌프 정리의 역은 성립하지 않는다.

예제 2.16. $L = \{0^n1^n | n \geq 0\}$은 정규언어가 아님을 증명하라.

먼저 L을 정규언어라고 하자. 그럼 임의의 t에 대해, $w = 0^t1^t$라고 하자. 그럼 $w = xyz$에 대해 $y = 0^k (k \leq t)$이므로, $i = 0$일 때, $xy^iz = 0^{t-k}1^t \notin L$ 이므로 펌프 정리에 의해 L은 정규언어가 아니다.

예제 2.17. $L = \{a^{n^2} | n \geq 0\}$은 정규언어가 아님을 증명하라.

예제 2.18. $L = \{0^n1^m | n \neq m\}$은 정규언어가 아님을 증명하라.

예제 2.19. $L = \{ab^nc^n | n \geq 0\} \cup \{a^kw | k \neq 1, w$는 a로 시작하지 않는 스트링$\}$이 펌프 정리가 성립하지만 정규언어가 아님을 증명하라.

정규언어인지 아닌지 알아내는 직관적인 방법은 '이 언어를 DFA로 표현하는 것이 가능한가?'를 생각해보면 된다. 예를 들어, 예제 2.16의 경우 0의 개수를 저장해 둔 뒤 1의 개수와 비교해야 하는데, 0의 개수로 가능한 가짓수는 무한하다. 따라서 유한 개의 상태만을 가진 DFA로는 구현해내는 것이 불가능하므로 정규언어가 아님을 알 수 있다.

2.8 연습 문제

** Exercise 1

Hello

CHAPTER 3

문법과 오토마타

3.1 문법

정의 3.1. 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 단원에서는 약간 문법을 제한한 형태만을 다룰 것이다.

정의 3.2. 문법 중 문맥 무관 문법(context-free grammar) G는 $A \in V, x \in (V \cup \Sigma)^*$에 대해 다음과 같은 생성규칙만을 가진다.

$$A \to x$$

이때 $A \to x, B \to y$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.

문맥 무관 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 프로그래밍 언어(Fortran, C 등)가 있다.

예제 3.1. $L = \{0^n 1^n \mid n \geq 0\}$을 만드는 문맥 무관 문법을 구하라.

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

예를 들어 0011을 만드는 유도 과정은 다음과 같다.

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

예제 3.2. '()(())'와 같이 적절히 닫혀있는 괄호를 만드는 문맥 무관 문법을 구하라.

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

예제 3.3. $L = \{0^n 1^m \mid n \leq m \leq 2n\}$를 만드는 문맥 무관 문법을 구하라.

예제 3.4. $L = \{0^n 1^m \mid n \neq m\}$를 만드는 문맥 무관 문법을 구하라.

$n > m$일 때와 $n < m$일 때를 구분하여 생각하면 된다.

예제 3.5. $L = \{a^m b^n c^u d^v | m + n = u + v\}$를 만드는 문맥 무관 문법을 구하라.

예제 3.6. 3+5*7와 같이 숫자, $+, *$로 구성된 수식을 만드는 문맥 무관 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.

3.2 정규문법

정의 3.3. 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 정규언어 종류이기 때문이다.

정리 3.1. 정규문법이 생성하는 언어는 정규언어이다.

증명. 증명이 '감동'적이지 않으므로 자세한 내용은 생략한다.

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

정리 3.2. 정규언어 L에 대해 $L = L(G)$인 정규문법 G가 존재한다.

증명. 이것도 딱히 '감동'적이지 않으므로 자세한 증명은 생략한다.

정규언어 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 3.7. 예제 2.9의 NFA와 동등한 정규문법을 구하라.

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

예제 3.8. 다음과 같은 정규문법과 동등한 NFA를 구하라.

$$\begin{aligned} S &\to aA \\ A &\to abS \mid a \end{aligned}$$

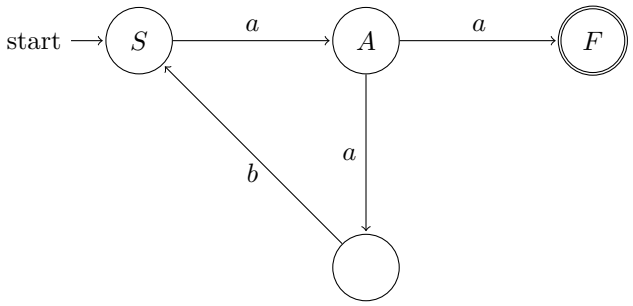


Figure 3.1

문맥무관 문법은 정규문법을 통해 정규언어를 표현할 수 있지만 정규언어의

경우 $\{0^n1^n\}$와 같은 언어집합을 표현하지 못한다. 따라서 정규언어는 문맥무관 언어의 진부분집합임을 알 수 있다.

3.3 파스 트리

어떤 문장(스트링)의 구조를 잘 파악하기 위해 파스트리를 정의하자.

정의 3.4. 파스 트리(parse tree)는 문맥무관 문법 G에 대해 다음과 같은 성질을 만족시키는 트리이다.

1. 루트 노드는 S이다.
2. 내부 노드는 V의 원소이고, A의 자식 노드가 $X_1, \cdots, X_k$ 이면 $A \to X_1 \cdots X_k$와 같은 생성규칙은 P에 있다.
3. 단말 노드는 $\Sigma \cup \{\epsilon\}$의 원소이고, 단말 노드가 ϵ이면 이 노드는 부모 노드의 유일한 자식 노드여야 한다.

예제 3.9. 예제 3.1와 같은 문맥 무관 문법에 대해 다음과 같은 유도과정을 나타내는 파스트리를 그려라.

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

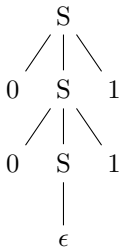


Figure 3.2

예제 3.10. 예제 3.2와 같은 문맥 무관 문법에 대해 다음과 같은 유도과정을 나타내는 파스트리를 그려라.

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

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.

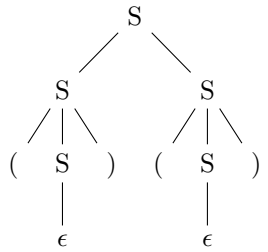


Figure 3.3

예제 3.11. 예제 3.10에서의 유도과정을 좌측유도로 표현하고 이 파스트리가 그림 3.3와 같다는 것을 보여라.

정의 3.5. 문맥 무관 문법 G에 대해 어떤 $w \in L(G)$에 대해 2 개 이상의 파스트리를 가지면 우리는 문법 G가 **애매하다**(ambiguous)라 한다.

예제 3.12. 다음 문법 G를 고려하자.

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

이는 예제 3.6와 같은 문법인데, 3+5*7에 대한 파스트리를 2개 이상 그려서 이 문법이 애매하다는 것을 보이고, 이를 애매하지 않은 문법으로 고쳐라.

예제 3.13. 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 파스트리를 2개 이상 그려서 이 문법이 애매함을 보여라.

예제 3.13의 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 문맥 무관 언어는 아니지만 문맥 무관 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 Tom

에게 많이 이야기한다는 것을 말했다.'라는 뜻일 수도 있다. 이에 대한 트리를 그려보면 다음과 같다. (다음 트리는 많이 간략화 되어 그려졌다.)

그림 3.4의 경우 1)을 의미하고, 그림 3.5의 경우 2)를 의미한다. 이렇게 단어

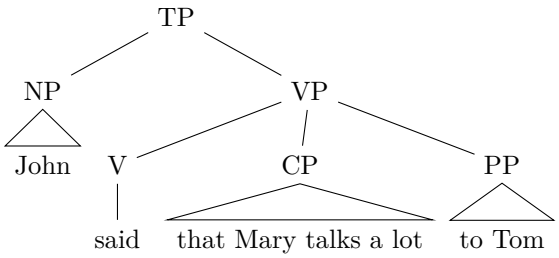


Figure 3.4

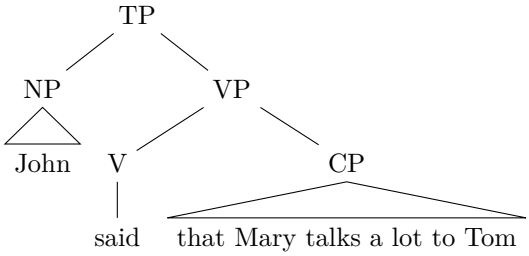


Figure 3.5

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 3.6. 문맥무관 언어 L을 생성하는 애매하지 않은 문법 G가 존재하면, L은 **애매하지 않다**라고 한다. L을 생성하는 모든 문법 G가 애매하면, L은 **본질적으로 애매하다**(inherently ambiguous)고 한다.

예제 3.14. 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.

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 L은 본질적으로 애매하다.

3.4 표준형

여기서부터 다루는 문맥무관 언어 L은 ϵ을 포함하지 않는다고 가정하자.

정의 3.7. (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 3.8. (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 3.3. 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 A가 nullable하다고 하자. $B \to C_1 \cdots C_k$에 대해 nullable 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 unit pair라 하자. P_1에서 $A \to B$와 같은 단위 생성규칙을 제외한 모든 규칙을 P_2에 넣는다. 그 후 unit pair인 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 $A \to x_1 \cdots x_k$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.

$A \to B_1 \cdots B_k$에 대해 새로운 변수 $D_1 \cdots D_{k-2}$를 V'에 도입하고

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

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

□

예제 3.15. 다음 문맥무관 언어에 대해 동등한 촘스키 표준형을 구하라.

$$\begin{aligned} S &\to ASA \mid aB \\ A &\to B \mid S \\ B &\to b \mid \epsilon \end{aligned}$$

$$\begin{aligned} S &\to AA_1 \mid UB \mid a \mid SA \mid AS \\ A &\to b \mid AA_1 \mid UB \mid a \mid SA \mid AS \\ A_1 &\to SA \\ U &\to a \\ B &\to b \end{aligned}$$

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 스트링 w가 문맥무관 언어 $L(G)$에 속하는지 결정하는 문제를 CYK 알고리즘을 통해 $O(n^3)$에 쉽게 해결하는 것이 가능하다. (본래라면 지수시간이 걸린다.)
두 번째로, 통사론에서 문장의 트리 구조를 그릴 때 그림 3.6과 같이 주로 이분지 구조(자식 노드가 두개 뿐인 트리)를 그리게 되는데 이러한 생성 규칙은 촘스키 표준형의 일종이라고 할 수 있다.

정리 3.4. 임의의 문맥무관 언어를 생성하는 그레이바흐 표준형이 항상 존재한다.

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

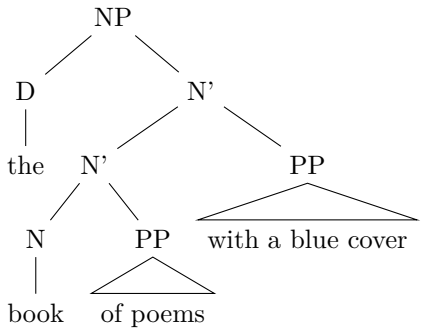


Figure 3.6

3.5 내리누름 오토마타

정의 3.9. 내리누름 오토마타(Pushdown Automata, PA) M은 $(Q, \Sigma, \Gamma, \Delta, q_0, F)$로 구성된다.

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

Figure 3.7: 그림그리기귀찮아.....

PA는 Δ가 전이 '관계'이므로 기본적으로 비결정론적이다. 비결정론적이므로 NFA와 마찬가지로 최종상태는 하나라고 가정할 수 있다.
$((q, a, X), (p, Y))$에 대해, q는 현재 상태, a는 입력 알파벳, X는 현재 스택의 top에 있는 글자이다. X가 ϵ일 때는 스택의 top을 읽지 않는다는 뜻이다. 그리고 p는 그 다음 상태, Y는 X를 대체하는 글자이다. $((q, a, \epsilon), (p, A))$의 경우, A를 스택에 넣는 push 연산이고, $((q, a, A), (p, \epsilon))$은 스택에서 A를 빼는 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 스트링, 현재 스택의 내용에 의해 결정된다. 어떤 한 번의 전이에 의해 PA의 상황이 변화하는 과정을 $\vdash$로 표현할 수 있다. 중간 과정을 생략하고 싶으면 $\vdash^*$와 같이 쓸 수도 있다.
어떤 입력 스트링 w를 PA가 읽었을 때의 상태가 최종상태이면 우리는 PA가 w를 받아들인다라고 한다. PA M이 받아들이는 스트링의 집합을 $L(G)$라고 표시하며 이는 다음과 같다.

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

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

예제 3.16. $\{0^n 1 0^n \mid n \geq 0\}$을 받아들이는 PA를 구하라. 그림 3.8을 보라.

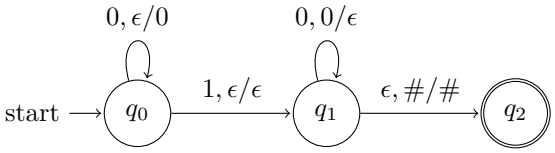


Figure 3.8

예제 3.17. $\{0^n 1^n \mid n \geq 0\}$을 받아들이는 PA를 구하라.

예제 3.18. $\{0^n 1^m \mid n \leq m \leq 2n\}$을 받아들이는 PA를 구하라.

예제 3.19. 0과 1의 개수가 같은 스트링을 받아들이는 PA를 구하라.

예제 3.20. $\{vw \mid v, w \in \{0,1\}^*, w \neq v^R, |v| = |w|\}$를 받아들이는 PA를 구하라.

3.6 PA = CFG

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

정리 3.5. 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 3.21. 다음 문맥무관 문법과 동등한 PA를 구하라.

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

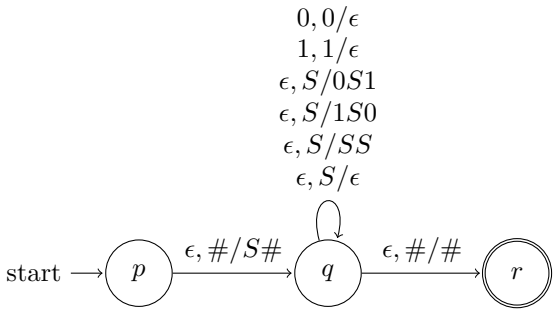


Figure 3.9

정리 3.6. 임의의 PA M에 대하여, $L(M)$을 표현하는 문맥무관 문법 G가 존재한다.

증명. M과 동등한 문맥무관 무법 G는 다음과 같이 만들 수 있다. 아 몰라 귀찮아 별로 안 감동적이라 나중에 씀.

1. 모든 $p, q \in Q$에 대해 $\langle pq \rangle \in V$
2. $S = \langle q_0 q_f \rangle$
3. (())

□

예제 3.22. 예제 3.16의 PA와 동등한 문맥무관 문법을 구하라.

3.7 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 3.7. (펌프 정리) 문맥무관 언어 L에 대해 다음을 만족하는 양의 정수 t가 존재한다. 길이가 t 이상인 임의의 스트링 $w = uvxyz \in L$는

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

을 만족한다.

증명. 문맥무관 언어 L을 생성하는 촘스키 표준형을 G라고 하자. 스트링 $w \in L$의 파스트리를 만들 때 G가 촘스키 표준형이므로 파스트리의 높이가 i이면 $|w| \leq 2^{i-1}$이다. (촘스키 표준형의 경우 파스트리에서 자식 노드를 두 개 이하로만 가질 수 있기 때문)
$|V| = k$라고 하자. 그럼 $t = 2^k$로 잡고, $|w| \geq t$이면, 파스트리의 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.
$A_1 \overset{*}{\Rightarrow} uxy$에 대해 A_1까지의 높이는 $k+1$ 이하이므로, $|vxy| \leq t$이다. 또한 $A_1 \to BC$이므로 $|vy| \geq 1$이어야 한다. 또한 $A_1 \overset{*}{\Rightarrow} vA_2y$이므로 이를 바꿔 말하면 $A \overset{*}{\Rightarrow} vAy$임을 알 수 있다. 따라서 모든 i에 대해 $uv^ixy^iz \in L$임을 알 수 있다.

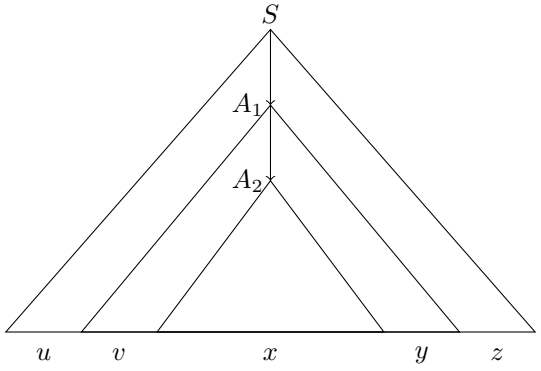


Figure 3.10

예제 3.23. $L = \{a^nb^nc^n \mid n \geq 0\}$이 문맥무관 언어가 아님을 보여라.
$w = a^tb^tc^t$라고 하자. $|vxy| \leq t$이므로 vy는 a, b, c모두를 포함할 수 없다. 따라서 $uv^2xy^2z \notin L$이므로 문맥무관 언어가 아니다.

예제 3.24. $L = \{ww \mid w \in \{0,1\}^*\}$이 문맥무관 언어가 아님을 보여라.

예제 3.25. $L = \{a^n \mid n \text{ is prime}\}$ 이 문맥무관 언어가 아님을 보여라.

정리 3.8. 문맥무관 언어는 (1) 합집합, (2) 접합, (3)Kleene 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 3.9. 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 3.23에 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 3.26. (소속 문제) 스트링 w 가 문맥무관 언어 L 에 속하는지 결정하라.
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. CYK 알고리즘은 '감동'적이진 않으므로 이 책에서 다루진 않는다. (궁금하면 구글에 검색해보기 바란다.)

3.8 결정 내리누름 오토마타

앞에서 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.
결정 내리누름 오토마타(deterministic pushdown automata, DPA)의 전이함수 δ는 $Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$에서 $Q \times \Gamma^*$로의 부분함수(partial function)로 정의한다. 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 3.27. $0^n1^n \mid n \geq 0$을 받아들이는 DPA를 구하라.

정리 3.10. 결정 문맥무관 언어는 여집합에 대해 닫혀있다.

정리 3.11. 결정 문맥무관 언어가 아니면서 문맥무관 언어인 언어집합이 존재한다.

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

정리 3.12. 문맥무관 언어 L에 대해, L을 받아들이는 DPA P가 존재하는 것과 L이 애매하지 않은 문법 G를 가지는 것은 동치이다.

CHAPTER 4

튜링 기계와 재귀 언어

4.1 튜링 기계

지금까지 우리는 정규식, 유한 오토마타를 이용해 정규언어를 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 튜링기계에 대해 알아보자.

정의 4.1. 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

튜링기계는 한쪽으로 무한한 길이의 테입과 하나의 제어기, 그리고 테입을 가르키는 헤드로 구성된다. 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 명시해주어야 한다. 따라서 정지 상태 $h \in H$에 진입하면 튜링기계는 정지한다.

튜링 기계의 상황(configuration)은 현재 제어기의 상태 q, 테입의 내용 w와 헤드가 가르키고 있는 위치로 구성된다. 헤드 왼쪽에 있는 스트링이 u, 오른쪽에 있는 스트링이 v, 가르키고 있는 스트링을 a라 할때 현재 상황은 $(q, u\underline{a}v)$로 나타낸다.

정의 4.2. TM $M = (Q, \Sigma, \Gamma, \delta, q_0, H)$이 정의하는 언어 $L(M)$은 다음과 같다.

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

이때 $h \in H$이고, w'는 임의의 스트링이다.

정의 4.3. 어떤 언어 L에 대하여, $L = L(M)$인 TM M이 존재할 때 언어 L을 **재귀열거 언어**(recursive enumerable language) 또는 **튜링 기계가 인식 가능한 언어**(Turing-recognizable language)라고 한다. (이 책에서는 주로 재귀열거 언어라는 표현을 사용할 것이다.)

예제 4.1. $L = \{0^n \mid n \geq 1\}$을 정의하는 TM M을 구하라.

예제 4.2. $L = \{0^n 1^n \mid n \geq 1\}$을 정의하는 TM M을 구하라.

예제 4.3. $L = \{ww^R \mid w \in \{0,1\}^*\}$을 정의하는 TM M을 구하라.

예제 4.4. $L = \{ww \mid w \in \{0,1\}^*\}$을 정의하는 TM M을 구하라.

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

정의 4.4. 언어 L에 대해 TM M에 대해 정지상태가 $\{y, n\}$이라 하자. $w \in L$이면 $(q_0, \underline{\#}w) \vdash^* (y, w')$이고 $w \notin L$이면 $(q_0, \underline{\#}w) \vdash^* (n, w')$를 만족하면 M이 L을 결정한다고 하고, L을 결정하는 M이 존재하면 L을 재귀 언어라고 부른다.

정리 4.1. 재귀 언어 종류는 재귀열거 언어 종류의 부분집합이다.

증명. 너무 당연해서 할 말이 없다... 그냥 n 상태를 무한하게 돌아가게 만들면 된다. □

예제 4.5. 앞에 나온 튜링 기계가 정의하는 언어가 재귀 언어임을 보여라. (매우 간단하다.)

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

정의 4.5. $f : \Sigma^* \to \Sigma^*$ 인 함수에 대해 TM M 이

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

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

여기서는

예제 4.6. 입력 w

4.2 튜링 기계의 확장

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

정의 4.6. k**-테입 튜링기계**는 일반적인 튜링 기계에서 테입이 k 개로 늘어난 형태다. 즉, 튜링 기계의 여섯 가지 요소가 다음과 같이 정의된다.

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

정리 4.2. k-테입 튜링 기계의 계산 능력은 튜링 기계와 동일하다.

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, k 개의 테입의 첫 번째 칸에 각각 a, b, c, d 라고 적혀있으면, 튜링 기계에서는 이를 $abcd$ 라는 하나의 심볼로 생각하면 된다. 이때 k 테입의 전이 함수는 이를 한 번에 수정할 수 있지만 일반적인 튜링기계는 이것이 불가능하므로 k 번 전이함수를 쪼개서 전이를 하면 된다. □

4.3 비결정론적 튜링기계

4.4 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 4.7. (naive한 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 $R_0, R_1, \cdots$, 테이프의 각 조각을 $T[1], T[2], \cdots$, 프로그램 카운터를 κ라고 하자. 각 레지스터와 테이프 한 조각은 임의의 자연수를 저장할 수 있다.
랜덤 접근 기계는 여러개의 **인스트럭션**(instruction)으로 이루어진 고정된 **프로그램**(program)을 기반으로 작동한다. (CPU에 있는 ISA와 근본적으로 다른게 없다는 걸 느끼면 된다.) 가능한 인스트럭션 종류로는 그림 4.1이 있다. (하나의

인스트럭션	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		$k \leftarrow 0$

Figure 4.1

예일 뿐이며 다른 인스트럭션들로도 충분히 다양한 연산이 가능하다.) 레지스터는 0, 프로그램 카운터는 1로 초기화 된다. κ의 값은 몇 번째 인스트럭션을 수행해야 하는지를 가르킨다. 한 인스트럭션이 수행될 때마다 특별하게 κ의 값을 지정해주는 인스트럭션이 아닌 이상 1씩 증가한다. κ가 0이 되면, 즉 halt 인스트럭션을 수행하면 랜덤 접근 기계는 정지한다.

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

4.5 무제한 문법

4.6 μ-재귀 함수

튜링기계가 받아들일 수 있는 언어는 재귀열거 언어, 그 중에서도 튜링기계가 항상 정지하면 계산 가능한 문제, 즉 재귀 언어라 한다. 그런데 왜 하필 '재귀' 일까? 이는 재귀적인 방식으로 정의되는 함수를 통해 계산 가능한 함수를 모두 표현할 수 있기 때문이다.

정의 4.8. $\mathbb{N}^k$에서 $\mathbb{N}$으로 가는 **기본 함수**(basic function)는 다음 세 함수를 의미한다.

1. 모든 $x_1, \cdots, x_n \in \mathbb{N}$에 대해 $zero(x_1, \cdots, x_n) = 0$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. $x_1, \cdots, x_n \in \mathbb{N}, k \geq j > 0$에 대해 $id_k(x_1, \cdots, x_n) = x_k$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, $h_1, \cdots, h_k$는 인자가 l개인 함수라 하자. g와 $h_1, \cdots, h_k$의 **합성**(composition)은 다음과 같은 인자가 l개인 함수이다.

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

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.

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

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

예제 4.7. 함수 $plus(m,n) = m+n$은 원시 재귀 함수임을 보여라.

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

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

예제 4.8. 함수 $mult(m,n) = m \cdot n$은 원시 재귀 함수임을 보여라.

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

예제 4.9. 함수 $exp(m,n) = m^n$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 4.9. 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 4.10. 음이 되지 않는 뺄셈 함수 $m \sim n = \max\{m-n, 0\}$가 원시 재귀 함수임을 보여라.

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

예제 4.11. $m > n$이면 1이고 이외에는 0인 원시 재귀 술어 $greater(m,n)$을 구하라.

예제 4.12. $m \geq n$ 이면 1 이고 이외에는 0 인 원시 재귀 술어 $greater\text{-}or\text{-}equal(m, n)$ 을 구하라.

정의 4.10. 인자가 k 개인 함수 f, g, h 와 원시 재귀 술어 p 에 대해 **조건으로 정의된 함수**(function defined bt cases) 는 다음과 같다.

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

예제 4.13. 조건으로 정의된 함수가 원시 재귀 함수임을 보여라.

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

예제 4.14. $m \div n$ 의 나머지와 몫을 각각 구하는 함수 $rem(m, n)$ 과 $div(m, n)$ 이 원시 재귀 함수임을 보여라.

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

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

예제 4.15. n 을 p 진법으로 나타냈을 때 m 번째 자릿수의 숫자를 표현하는 함수 $digit(m, n, p)$ 이 원시 재귀 함수임을 보여라.

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

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

정리 4.3. 원시 재귀 함수가 아닌 함수 중 계산 가능한 함수가 존재한다.

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

$$f_0, f_1, f_2, f_3, \cdots$$

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.

정의 4.11. 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 4.12. 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. $a_1, \cdots, a_k \in \mathbb{N}$에 대해

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

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

정리 4.4. A가 Ack에 의해 주요화되는 함수들의 집합이라 하자. 원시 재귀 함수들의 집합은 A의 부분집합이다.

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

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

다음으로 합성 연산에 대해 닫혀있음을 보이자. 함수 $g_1, \cdots, g_m$와 h는 각각 인자가 k개, m개이고, A의 원소라 하자. 이는

$$\begin{aligned} g_i(x_1, \cdots, x_k) &< Ack(r_i, x) \\ h(y_1, \cdots, y_m) &< Ack(s, y) \end{aligned}$$

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

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

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

$$\begin{aligned} g(x_1, \cdots, x_k) &< Ack(r, x) \\ h(y_1, \cdots, y_{k+2}) &< Ack(s, y) \end{aligned}$$

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

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

$q = 1 + \max\{r, s\}$ 라 하자. 먼저

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

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

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

이때 수학적 귀납법에 의해 다음이 성립한다.

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

$w = \max\{x, y\}$ 라 하자.

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

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

정리 4.5. 아커만 함수는 계산 가능하지만 원시 재귀 함수가 아니다.

증명. Ack가 원시 재귀 함수면 정리 4.4에 의해 $Ack \in A$인데, 이는 불가능하므로 A는 원시 재귀 함수가 아니다. □

따라서 우리는 모든 계산 가능한 함수를 표현하기 위해서 다음과 같은 방법을 정의한다.

정의 4.13. 인자가 $k+1$개인 함수 g에 대해 g의 **최소화**(minimalization)은 인자가 k개인 함수 f로 다음과 같이 정의된다.

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

이때 $g(n_1, \cdots, n_k, m) = 1$이 되게 하는 $m \geq 0$이 존재하면 g가 **최소화 가능하다**(minimalizable)고 하고, 만약 g가 최소화 가능하지 않으면 $f(n_1, \cdots, n_k) = 0$으로 정의한다. 이러한 g의 최소화 과정은 $\mu\ m[g(n_1, \cdots, n_k, m) = 1]$와 같이 표기한다.
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 $g(n_1, \cdots, n_k, m) \neq 1$ **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 앞에서 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 $f_0, f_1, \cdots$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 앞에서 사용한 논리를 사용할 수 없다.

예제 4.16. 로가리슴 함수 $log(m, n) = \lceil \log_{m+2}(n+1) \rceil$가 μ-재귀 함수임을 보여

라.(log가 범위를 벗어나지 않도록 정의했다.)

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

정리 4.6. 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 $h_1, \cdots, h_l : \mathbb{N}^k \to \mathbb{N}$의 합성으로 정의되었다고 하자. 이는 튜링 기계로 다음과 같이 계산할 수 있다.

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

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

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

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

$m \leftarrow 0$
while $g(n_1, \cdots, n_k, m) \neq 1$ **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. (인자가 여러개인 함수도 유사하게 확장하여 증명할 수 있다.) 이제 함수 f가 μ-재귀 함수임을 보이면 된다.

일반성을 잃지 않고 Q, Σ, Γ가 서로소라고 하자. $b = |Q| + |\Sigma| + |\Gamma|$라고 하고 함수

$E : Q \cup \Sigma \cup \Gamma \to \{0, 1 \cdots, b-1\}$를 일대일 함수라고 하자. 우리는 M의 상황을 b를 통해 표현할 것이다. 상황 $(q, a_1 a_2 \cdots \underline{a_k} \cdots a_n)$은 $a_1 a_2 \cdots a_k q a_{k+1} \cdots a_n$과 같은 형태로 b를 이용해 다음과 같이 정수로 표현한다. 이를 b진법 표기법이라 하자.

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

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 다음과 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 튜링기계 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. □

CHAPTER 5

계산 가능성

우리는 지금까지 튜링기계의 뛰어난 계산능력에 대해 알아보았다. 튜링 기계는 정말 다양한 일을 할 수 있으며, 이러한 기초적인 개념에서 확장되어 현재 우리가 사용하는 컴퓨터가 되었다. 그런데 튜링기계는 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.

5.1 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 튜링기계는 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 때문이다. (다른 알파벳이 있다고 하더라도 ASCII 코드처럼 적당히 부호화를 하면 된다.)
튜링기계 $M = (Q, \{0, 1\}, \{0, 1, \#\}, \delta, q_1, H)$ 를 다음과 같은 방식으로 $\{0, 1\}$ 상의 스트링으로 만들자.

1. $Q = \{q_1, \cdots, q_n\}$ 에 대해 q_i 는 각각 0^i 로 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, S 는 각각 0, 00, 000으로 표시한다.
5. $\delta(q, a) = (p, b, d)$ 는 각 q, a, p, b, d 의 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 스트링으로 대응시킨 것을 M의 **부호화**라고 하고, $\langle M \rangle$으로 표시한다.

예제 5.1. 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.

$$111010001001000100110010100100 10011$$
$$001001001010011001000100010001000111$$

튜링 기계 M에 들어가는 입력 스트링 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.

5.2 보편 만능 기계

컴퓨터가 지금까지 인류가 만들어온 기계와 다른 점은 무엇일까? 형광등은 불을 밝힐 수만 있다. 세탁기는 세탁만 할 수 있다. 하지만 컴퓨터는 이렇게 책도 쓸 수 있고, 게임도 할 수 있고, 계산도 할 수 있다. 즉, 어디에서나 사용가능한 **보편 만능 기계**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 5.1. 보편 만능 튜링 기계 M_u는 3-테입 튜링 기계로 다음과 같이 정의된다.

1. 첫 번째 테입은 $\langle M \rangle$을 저장한다. 먼저, 첫째

2. 두 번째 테입은

3. 세 번째 테입은

5.3 정지 문제

5.4 정지 문제의 응용

정지 문제는 우리가 해결하고 싶어하는 컴퓨터 과학의 문제들에 대해서 '그건 불가능하다.'라고 말할 수 있게 해주는 아주 좋은 증명 도구이다. 다음 예제를 보자.

예제 5.2. 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 '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 type checker는 항상 맞을까? 언제나 옳은 답(타입 오류가 없으면 없다고 해주고, 있으면 있다고 해주는)을 내놓는 type checker는 존재할까? 이와 같은 type checker가 존재하지 않는다는 사실을 증명해라.

증명. 다음 코드를 생각하자.

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

만약 M이 정지한다면 이 코드는 type error가 발생한다. 만약 정지하지 않는다면, 이 코드는 type error가 발생하지 않을 것이다. 만약 완전한 type checker가 존재한다면 type error를 검증할 수 있고, 즉 이를 통해 M이 정지하지 않는지를 알 수 있다. 하지만 이는 불가능하므로 완전한 type checker는 존재하지 않는다. □

그래서 요즘 사용하는 type checker는 '안 되는 것 같아 보이지만 사실은 맞는 프로그램'을 검증하는 것을 포기한다. 위 증명에서 사용된 코드의 경우, 아마 python IDE에서 사용하는 type checker의 경우 빨간 줄을 표시할 것이다. 즉, 최소한 type checker가 된다고 했으면 무조건 type에 있어서는 오류를 내뿜지 않는 프로그램이다. (그렇다면 syntax error는 어떨까? 궁금하면 3장으로 돌아가자.)

5.5 람다 대수

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 정리(Gödel's incompleteness theorems)를 자신만의 방식으로 증명하고 싶었던 것이다.

정리 5.1. (제 1 괴델의 불완전성 정리) 모든 공리계는 무모순인 동시에 완전할 수 없다.

정리 5.2. (제 2 괴델의 불완전성 정리) 어떤 공리계가 무모순일 경우, 그 공리계로부터 그 공리계 자신의 무모순성을 도출할 수 없다.

매우 어렵게 들리지만, 간단하게 이야기하면 참인지 거짓인지 판별할 수 없는 명제는 항상 존재한다는 것이다. 그 예시로는 **연속성 가설**(continuum hypothesis) 등이 있다. 튜링은 수학의 추론 규칙들을 튜링 기계라는 기계적인 방식으로 환원하였고, 이를 통해 정지 문제와 같은 증명할 수 없는 문제들을 제시해냈다.
그런데 앨런 튜링보다 먼저 알론조 처치(Alonzo Church)가 비슷한 방식으로 이러한 증명을 제안해냈는데, 이를 **람다 대수**(lambda calculus)라고 한다.

정의 5.2. 람다식(lambda expression)은 다음과 같이 귀납적으로 정의된다.

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

CHAPTER 6

시간 복잡도

6.1 점근적 표기법

지금까지는 어떤 문제가 계산할 수 있는 문제인지 알아봤다. 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 스트링의 길이 n에 대해 M의 최대 스텝(step)을 의미한다.

우리가 어떤 알고리즘(튜링 머신)의 시간 복잡도를 계산할 때 입력(입력 스트링)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 상수항을 무시하면서 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.

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.

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

예제 6.1. 다음 점근적 표기법이 올바르다는 것을 증명하라. (사실 증명할 필요 없고 그냥 맞다는 것을 느끼면 된다.)

1. $\sqrt{n} = o(n)$
2. $n \log n = o(n^2)$
3. $3n \log n + 2n \log \log n = O(n \log n)$
4. $2^n = o(n!)$

6.2 복잡도 분석

정의 6.4. 함수 $t : \mathbb{N} \to \mathbb{R}^t$ 에 대해 **시간 복잡도 클래스**(time complexity class) $\mathrm{TIME}(t(n))$ 은 시간 복잡도가 $O(t(n))$ 인 튜링 기계로 결정할 수 있는 언어들의 집합이다.

예제 6.2. 언어 $L = \{0^n 1^n \mid n \geq 0\}$ 을 받아들이는 튜링기계 M_1 에 대한 점근적 상한을 구하라.

1. 입력 스트링 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 스트링을 받아들이지 않고, 만약 다 지워지면 이를 받아들인다.

먼저 1의 경우 $O(n)$ 번의 스텝이 필요하다. 3의 경우, 각 스트링을 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 튜링기계 M_2 를 생각해보자.

예제 6.3. 언어 $L = \{0^n 1^n \mid n \geq 0\}$을 받아들이는 튜링기계 M_2에 대한 점근적 상한을 구하라.

1. 입력 스트링을 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.

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.
그럼 이보다 더 빠른 수행시간을 가지는 알고리즘은 존재할까? 아쉽게도 불가능하다. 이는 정리 6.1에서 알 수 있다.

정리 6.1. 어떤 언어가 정규언어임은 그 언어가 $\mathrm{TIME}(o(n \log n))$의 원소인 것과 동치이다.

증명. 증명 아무리 찾아도 제대로 된 증명이 없어서 일단 생략. 믿으셈... □

대신에 2-테입 튜링 기계를 사용하면 더 적은 스텝으로 판별할 수 있다.

예제 6.4. 언어 $L = \{0^n 1^n \mid n \geq 0\}$을 받아들이는 2-테입 튜링기계 M_3에 대한 점근적 상한을 구하라.

1. 입력 스트링을 전체를 스캔하면서 1 오른쪽에 0이 있으면 받아 들이지 않는다.
2. 처음부터 읽으면서 0이 나올 때마다 두 번째 테입에 하나씩 0을 복사해둔다. 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.

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 동일

했으나, 계산 복잡도 이론에서는 그렇지 않다. 어떤 튜링기계를 사용하는지에 따라 우리가 알고자 하는 복잡도가 차이나게 된다. 그럼에도 불구하고 결정론적인 튜링기계들 사이에서 계산 복잡도는 '큰 차이'가 없다는 것을 보일 것이다.

정리 6.2. 함수 $t(n) \geq n$에 대해, 수행 시간이 $t(n)$인 k-테입 튜링기계 M과 동등하고 수행시간이 $O\left(t^2(n)\right)$인 1-테입 튜링기계 M'이 존재한다.

증명. 이는 우리가 앞 장에서 동등한 튜링기계를 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 스트링의 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(입력 테입을 다 읽지 않는 튜링기계는 의미가 없으므로 $t \geq n$이라 가정해도 무방하다.) M'의 수행시간은 $O(t^2(n))$이다. □

정리 6.3. 수행시간이 $t(n)$이고 테입 알파벳이 Γ인 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 6.5. 수행시간이 $t(n)$인 양방향 튜링기계 M에 대해 이와 동등하고 수행시간이 $O(t(n))$인 1-테입 튜링기계가 존재한다.

위 정리들에 의해 우리가 알고 있는 '상식적인' 결정론적 튜링기계의 수행시간이 다항시간이라면 가장 기본적인 형태인 동등한 1-테입 튜링기계가 수행시간이 다항시간이라는 사실을 알 수 있다.
그럼 실제 알고리즘을 분석하는 데에 있어서는 어떤 모델을 사용할까? 현실의 컴퓨터는 **랜덤 접근 머신**(Random Access Machine, RAM)의 추상화이므로 이 모델을 사용한다. 1-테입 튜링기계와 가장 큰 차이라면 테입에 해당하는 레지스터(register)를 바로 접근할 수 있다는 것이다. 이 또한 결정론적 튜링기계의 한 종류이므로 RAM으로 다항시간내에 해결할 수 있는 문제는 1-테입 튜링기계로도 다항시간 내에 해결이 가능하다.

정의 6.5. 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.

정리 6.4. 함수 $t(n) \geq n$에 대해 수행시간이 $t(n)$인 비결정론적 1-테입 튜링기계와 동등하고 수행시간이 $2^{O(t(n))}$인 1-테입 튜링기계가 존재한다.

증명. 이것도 앞 장에서 비결정론적 튜링기계를 흉내내는 결정론적 튜링기계를 만드는 과정을 생각하면 된다. 비결정론적 튜링기계 N의 수행시간이 $t(n)$이라 하자. 그럼 우리는 앞 장에서 했듯이 N을 흉내내는 결정론적 튜링기계 M을 만들 수 있다.
입력 스트링의 길이 n에 대하여 N이 가질 수 있는 비결정론적 계산 트리는 최대 $t(n)$이다. 트리의 각 노드에 대해서 가질 수 있는 최대 자식의 수가 b라고 하자.(N의 전이 관계에서 전이하는 최대 상태의 개수를 잡으면 된다.) 따라서 트리의 잎 노드의 수는 최대 $b^{t(n)}$이다. M이 N을 흉내낼 때 너비 우선 탐색을 하게 된다. 이때 트리의 총 노드 수는 $2b^{t(n)}$ 이하 이므로 $O(b^{t(n)})$이라 할 수 있다. 모든 노드를 탐색하는 시간은 $O(t(n))$이므로 M의 수행시간은 $O(t(n)b^{t(n)}) = 2^{O(t(n))}$이다. (여기서 이 수식이 성립하는 이유는 과제!) □

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 6.6. **P**는 1-테입 튜링기계로 다항시간 내에 풀 수 있는 언어들의 class이다. 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 6.6. 문제(언어) **RELPRIME**은 다음과 같이 정의된다.

$$\mathrm{RELPRIME} = \{\langle x, y\rangle \mid x\text{와 } y\text{는 서로소이다.}\}$$

이때 $\mathrm{RELPRIME} \in \mathrm{P}$임을 보여라.

증명. 유클리디안 알고리즘을 사용하면 당연하게 다항시간 내($O(n)$)에 해결이 가능하다. □

예제 6.7. 문제 **PATH**는 다음과 같이 정의된다.

$$\text{PATH} = \{\langle G, s, t\rangle \mid \text{방향 그래프 } G\text{에 대해 } s\text{에서 } t\text{로 가는 경로가 존재한다.}\}$$

이때 PATH $\in$ P임을 보여라.

증명. DFS 또는 BFS를 이용해 탐색하면 다항시간 내($O(|V| + |E|)$)에 해결이 가능하다. □

예제 6.8. 문제 **SHORTPATH**는 다음과 같이 정의된다.

$$\text{SHORTPATH} = \{\langle G, s, t, k\rangle \mid \text{가중치가 있는 그래프 } G\text{에 대해 } s\text{에서 } t\text{까지 가중치의 합이 } k\text{이하인 경로가 존재한다.}\}$$

SHORTPATH $\in$ P임을 보여라.

증명. 벨만-포드 알고리즘을 사용하면 $O(n^3)$에 해결이 가능하다. □

6.4 NP

정의 6.7. 언어 A의 **검증자**(verifier)는 다음을 만족시키는 알고리즘 V이다.

$$A = \{w \mid V\text{가 } \langle w, c\rangle\text{를 어떤 스트링 } c\text{에 대해 받아들인다.}\}$$

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 6.9. 헤밀토니안 경로(hamiltonian path)는 방향 그래프 G에서 모든 노드를 한 번씩만 방문하는 경로이다. 이때 문제 **HAMPATH**는 다음과 같이 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 가능한가?

정의 6.8. **NP**는 다항시간 내에 검증 가능한 언어들의 집합이다.

예제 6.10. 문제 **COMPOSITE**은 다음과 같이 정의된다.

$$\text{COMPOSITE} = \{n \mid \exists p, q > 1, n = pq\}$$

COMPOSITE $\in$ NP임을 보여라.

예제 6.11. 문제 **LONGPATH**는 다음과 같이 정의된다.

$$\text{LONGPATH} = \{\langle G, s, t, k\rangle \mid \text{가중치가 있는 그래프 } G\text{에 대해 } s\text{에서 } t\text{까지 가중치의 합이 } k\text{이상인 경로가 존재한다.}\}$$

LONGPATH $\in$ NP임을 보여라.

정의 6.9. 비결정론적 시간복잡도 클래스 **NTIME**은 다음과 같이 정의된다.

$$\text{NTIME}(t(n)) = \{L \mid L\text{은 비결정론적 튜링 기계로 } O(t(n)) \text{ 내에 결정되는 언어이다.}\}$$

정리 6.5. 어떤 언어가 NP라는 것은 그 언어가 어떤 비결정론적 튜링기계로 다항시간 내에 결정 가능하다는 것과 동치이다. 즉, $\text{NP} = \bigcup_k \text{NTIME}(n^k)$ 이다.

증명. $A \in$ NP라 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 스트링 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 튜링기계에서 다양한 상황 중 하나를 선택해 가면서 흉내낸다. (앞에서 비결정론적 튜링기계를 흉내내는 결정론적 튜링기계를 생각하면 된다.)
2. 만약 이 N이 받아들이면 받아들이고, 아니면 받아들이지 않는다.

□

예제 6.12. 문제 **SUBSET-SUM**은 다음과 같이 정의된다.

$$\text{SUBSET-SUM} = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\} \text{ 에 대해 } \sum y_i = t\text{를 만족하는 } \{y_1, \cdots, y_l\} \subseteq S\text{가 존재한다.}\}$$

이때 SUBSET-SUM $\in$ NP임을 보여라.

증명. 다항시간 검증자 V는 다음과 같이 만들 수 있다.

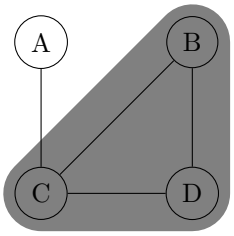
1. c의 원소의 합이 t인지 확인한다.
2. c가 S의 부분집합인지 확인한다.
3. 만약 1, 2 둘 다 만족하면 받아들이고, 그렇지 않으면 받아들이지 않는다.

다른 증명으로는 다항시간 내에 작동하는 비결정론적 튜링기계를 다음과 같이 만들면 된다.

1. $\langle S, t\rangle$에 대해 S의 부분집합 c를 비결정론적으로 선택한다.
2. c의 원소들의 합이 t인지 검사한다.
3. 만약 통과하면 받아들이고, 그렇지 않다면 받아들이지 않는다.

□

예제 6.13. **클릭**(clique)은 무방향 그래프의 부분 그래프 중 모든 노드가 서로 연결되어 있는 그래프를 의미한다. k-클릭(k-clique)은 노드가 k개인 클릭을 의미한다. 문제 **CLIQUE**는 다음과 같이 정의된다.



3-clique

Figure 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 컴퓨터과학자들의 노력이 있었으나 아쉽게도 이는 밝혀지지 않았다.

6.5 NP-완비

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-완비라는 것을 정의하기로 한다.[1]

정의 6.10. 함수 $f : \Sigma^* \to \Sigma^*$에 대해 다항시간 내에 작동하고, $f(w)$를 테입에 적고 정지하는 튜링기계 M이 존재하면 f를 **다항시간 내 계산 가능한 함수**(polynomial time computable function)라고 한다.

정의 6.11. 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 6.6. $A \leq_p B$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.

□

[1] M. Garey와 D.Johnson의 책인 Computers and Intractability: A Guide to the Thoery of NP-Completeness에 있는 유명한 예제이다.

예제 6.14. 리터럴(literal)이란 불 변수나 불 변수의 역을 의미한다. 이때 리터럴들이 $\vee$로 연결 된 것을 **절**(clause)라고 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 6.12. 모든 언어 $L \in$ NP에 대해 언어 A가 $L \leq_p A$이면 A를 **NP-하드**(NP-Hard)라 한다.

정의 6.13. 언어 A가 다음 두 가지 조건을 만족시키면 A를 **NP-완비**(NP-Complete)라 한다.

1. $A \in$ NP
2. $A \in$ NP-하드

정리 6.7. $A \in$ NP-완료이고 $A \in$ P면, P = NP이다.

즉, 어떤 NP-완료인 언어 A를 해결하는 다항시간 알고리즘을 발견한다면 무려 P=NP임을 증명할 수 있게 된다. (참고로 어떤 NP 문제의 다항시간 알고리즘이 존재하지 않는다면 NP-완료가 아니어도 알아서 P $\neq$ NP임을 알 수 있다.)
그런데 어떤 언어가 NP-완료임을 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 6.8. (쿡-레빈 정리) 언어 **SAT**는 다음과 같이 정의된다.

$$\text{SAT} = \{\langle\phi\rangle \mid \text{부울식 } \phi\text{가 만족 가능하다.}\}$$

이때 SAT는 NP-완료이다.

증명. 먼저 SAT가 NP임은 자명하다. 모든 변수들의 T/F를 선택하여 만족하는지 확인하면 되기 때문이다.
이제 모든 언어 $A \in$ NP에 대해 A에서 SAT로 다항시간 내에 변환 가능함을 보이자. A를 결정하는 비결정론적 튜링기계 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 N의

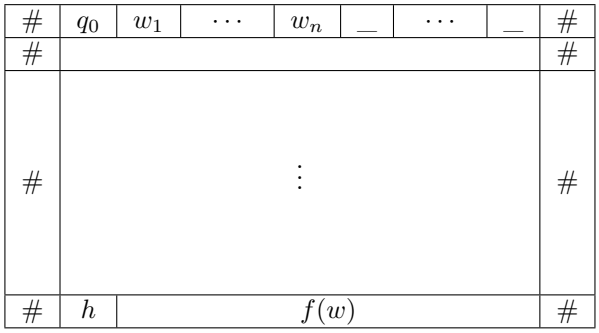


Figure 6.2

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 **셀**(cell)에 대해 i행 j열에 있는 셀을 $cell[i,j]$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 셀이 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 셀이 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 셀이 C의 원소를 하나만 담고 있는지 확인한다.

ϕ_{start}는 첫번째 행이 제대로 시작 상황인지 확인한다. 이는 다음과 같다.

$$\begin{aligned} \phi_{start} = & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\_} \wedge \cdots \wedge x_{1,n^k-1,\_} \wedge x_{1,n^k,\#} \end{aligned}$$

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)

Table 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 셀 중 상태(헤드)와 붙어있지 않은

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)

Table 6.2

셀은 값이 변하지 않아야 한다. 그리고 상태(헤드)와 붙어있는 셀은 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.
이제 ϕ_{move}를 만들어보자. ϕ_{move}는 창문이 적법한지 검사해서 도표가 적절히

전이 관계에 따라 구성되는지 확인한다. 각 창문은 총 6개의 셀로 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 자세히 쓰면 다음과 같다.

$$\bigvee_{(a_1,\cdots,a_6)\text{는 적법}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

이제 이러한 변환이 다항 시간 내에 이루어짐을 보이자. 그러기 위해서 ϕ의 크기를 구해야 한다. 표는 n^{2k}개의 셀들로 이루어지고, 각 셀에서 가능한 문자의 개수는 튜링기계 N에 의해 결정되므로(즉 상수이므로 무시할 수 있음) 변수의 개수는 $O(n^{2k})$이다.
이제 각 4가지 부분에 대해 크기를 구해보자. ϕ_{cell}는 각 셀에 대해 검사하므로 크기가 셀의 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.
즉, 모든 NP 문제는 다항 시간 내에 SAT 문제로 환원가능하므로 SAT는 NP-완료이다. □

예제 6.15. 앞의 증명에서 우리는 2×3 크기의 창문을 사용했다. 왜 2×2 크기의 창문을 사용하면 안 되는지 설명하라.

예제 6.16. 3-SAT는 NP-완료임을 보여라.
우리가 이미 알고 있는 NP-완료 문제인 SAT 문제를 적절히 변환하면 된다. 예를 들어, $(a_1 \vee a_2 \vee \cdots \vee a_n)$와 같은 절은 다음과 같이 새로운 변수 $z_1, \cdots, z_{n-3}$를 도입해 변형하면 된다.

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge \cdots \wedge (\overline{z_{n-3}} \vee a_{n-1} \vee a_n)$$

이는 당연히 다항시간 변환이다.

6.6 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 6.14. (다항 시간 변환의 확장) 문제 A의 사례 α를 문제 B의 사례 β

CHAPTER 7

공간 복잡도

Bibliography

[1] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[2] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[3] Harry R Lewis and Christos H Papadimitriou. Elements of the theory of computation. *ACM SIGACT News*, 29(3):62–78, 1998.

[4] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

[5] 문병로. 쉽게 배우는 알고리즘. 한빛미디어, 2018.

[6] 박근수. 오토마타 이론 강의록, 2021.