프로그래밍의 수련(修練)
(A Discipline of Programming)

엣져 위베 다익스트라
(Edsger Wybe Dijkstra)

두번째: 프로그래밍 언어의 역할/상태와 그 특성의 규정

해역(解譯): 김도형(성신여자대학교 전산학과)

1장. 프로그래밍 언어(言語)의 역할(役割)

0장 '수행의 추상화'에서 나는 두 양(陽)의 (그리고 너무 크지 않은) 정수의 최대 공약수를 계산하는 여러 '기계'를 비형식적(非形式的)으로 서술했었다. 하나는 판지(板紙) 위를 움직이는 한 개의 조약돌로 나타내었고, 두번째 것은 각각 (별개의) 축 위를 움직이는 두 개의 반쪽 조약돌로 묘사되었다. 마지막 것은 각각 (정해진 범위 내의) 하나의 정수값을 저장할 수 있는 두 개의 레지스터로 표현되었다. 물리적으로 보면 이 세 개의 '기계'는 무척 다르나, 수학적으로 보면 매우 비슷하다. 즉, 그것들이 최대 공약수를 구할 수 있다는 논증의 주요 부분은 세 기계 모두 동일하다. 이 이유는 그것들이 동일한 '게임의 규칙들' 집합에 대해서 구체화(具體化)시키는 부분만 상이할 뿐이고, '유클리드 알고리즘'으로 알려져 있는 이 고안의 핵심을 이루고 있는 것은 그 규칙들의 집합이기 때문이다.

앞 장에서는 유클리드 알고리즘을 다소 비형식적인 방식으로 말로 묘사했다. 그러나 그 알고리즘에 대응하여 가능한 계산들의 수가 너무 많으므로, 그 정합성(整合性)에 대한 증명이 필요함도 주장했었다.[1] 어떤 알고리즘이 단지 비형식적으로만 주어져 있다면 그것은 형식화된 처리를 위해서는 매우 부적합한 대상이다.[2] 형식화된 처리를 하려면 뭔가 적당한 형식적인 표기법으로 알고리즘을 기술(記述)하는 것이 필요하다.

그러한 형식화된 표기 기법이 가질 법한 장점들은 많다. 어떤 표기 방법을 사용한다는 것은 곧 그것을 사용해서 서술된 것은 무엇이든 간에, 그 방법에 의해 기술 가능한 대상들의 (종종 무한한) 집합 내에 포함되는 특정 원소가 됨을 의미한다. 우리가 사용할 표기 방법은 물론 유클리드 알고리즘을 우아하고 간결하게 기술할 수 있어야 한다. 그러나 일단 그 작업이 달성되고 나면, (우리의 표기 방법을 사용해 기술된) 그 알고리즘은 (우리의 표기 방법을 사용해 기술 가능한) 모든 종류의 알고리즘들로 이루어진 거대한 집단의 한 구성원이 되는 것이다. 그래서 이들 다른 알고리즘을 서술하는 과정에서 우리가 사용한 표기 기법의 보다 흥미있는 응용을 발견하길 기대할 수 있다. 유클리드 알고리즘의 경우는 너무 간단하기 때문에 비형식적으로 서술할 수도 있

[1] 만약 어떤 알고리즘에 속하는 가능한 계산의 수가 아주 작다면 형식적이고 엄격한 정합성 증명은 필수적이지 않고, 오히려 비효율적일지도 모른다. 가능한 몇 가지 경우의 계산에 대해 그 결과가 옳음을 비형식적으로 보이거나 검사하는 방식으로 옳음을 보이는 것이 훨씬 빠르고 비용이 절약될 수도 있기 때문이다.

[2] 정합성 증명 같은 것이 형식화된 처리의 대표적인 예이다.

다고 주장할 수 있다. (그러나) 형식화된 표기법의 진가(眞價)는 그것 없이는 할 수 없는 일에서 스스로 드러날 것이다!

형식적인 표기 방법의 두번째 장점은 그것으로 인해서 우리가 알고리즘을 수학적인 대상(對象)으로 삼아 연구할 수 있다는 것이다. 알고리즘의 형식화된 기술은 이 때 우리들의 지적 이해를 위한 관건(關鍵)이 된다.[3] 예를 들면, 형식화된 서술 덕분에 알고리즘 집합에 대한 여러 정리(定理; theorem)를 증명할 수도 있을 것이다. 왜냐하면 각 알고리즘에 대한 형식화된 기술들은 어떤 구조적 성질을 공유할 것이기 때문이다.[4]

마지막으로 그러한 표기 방법 때문에 우리는 알고리즘을 전혀 모호(模糊)하지 않게 정의할 수 있으므로, 그렇게 서술된 알고리즘과 그 매개 변수(입력)의 값이 주어진다면 그것에 대한 답(출력)이 어떤 것인가에 관해서는 의심이나 불확실성이 있을 수가 없다. 그렇다면 그 계산을, 알고리즘(의 형식화된 기술)과 매개 변수가 주어지면 더 이상 사람이 개입할 필요없이 답을 생산해 내는 자동장치(自動裝置; automaton)에 의해서 수행하는 것도 생각할 수 있다. 알고리즘과 매개 변수를 서로 결합시키는 그러한 자동 장치는 실제로 만들어졌다. 그것을 '자동 컴퓨터'라고[5] 부른다. 컴퓨터에 의해 자동적으로 수행할 목적의 알고리즘을 '프로그램'이라고 부르고, 1950년대 후반 이래로 프로그램을 표시하기 위한 형식화된 방법을 '프로그래밍 언어'라고 부른다. (프로그램을 위한 표기 기법과 관련해 '언어'라는 용어를 도입한 것은 장점과 단점을 다 가지고[6] 있었다. 한편으로는 기존의 언어학(言語學) 이론이 토론을 위한 자연스런 (이론의) 체제와 확립된 용어('문법', '구문(句文; syntax)', '어의(語義; semantics)' 등)를 제공했다는 점에서는 매우 도움이 되어 왔다. 다

---

[3] 원문은 "the formal description of the algorithm then provides the handle for our intellectual grip."으로 되어 있다.

[4] 이 두번째 장점에 대한 것은 이 책의 주된 내용과도 밀접한 관련이 있기 때문에 뒤에 나오는 많은 예들을 이 장점과 연관하여 생각해 볼 수 있을 것이다.

[5] 원문에도 'automatic computers'라고 되어 있다. 물론 저자가 'automatic'을 덧붙인 것은 인간의 개입없이 알고리즘의 수행이 가능함을 강조하는 것이 목적이다. 그러나 한편으로는, 여기서 '자동'이 붙어 있는 것을 보면서 다른 몇 가지 생각이 떠오르기도 한다(원문의 인용 부호는 'automatic'을 강조하기 위한 것이다). 첫째는 이 책이 출판된 연도를 돌아보고 꽤 오래 전이라는 것을 다시 상기하게 된다—우리나라에서 본격적으로 자동차를 자체 생산하기 시작한 초기에는, 자동 변속장치(變速裝置)를 채택한 차의 경우 차체 옆에 큼지막하게 'automatic'이라고 붙이고 다니지 않았던가! (요즘도 그러던가...?) 두번째는 '프로그램-내장(內藏) 개념(stored-program concept)'이 등장하기 전이라면 그 때 만들어진 계산 기계는 엄격한 의미에서 '자동'이란 말을 붙일 수 없을 법도 하기 때문에 이해가 되는 면도 있다는 것이다. 세번째는 모름지기 인간의 편리를 위한 장치라면 인간에게 편리를 가져다 주는 그 측면에서는 어느 정도의 자동성(automaticity)은 당연히 있을 것 같은데 굳이 '자동'이라는 말을 붙이는 것을 비꼬는 것은 아닌가 하는 생각이다. 잘 알려져 있다시피, 초창기 컴퓨터 중 프로그램 내장 개념을 처음으로 채택한 EDSAC은 'Electronic Delayed Storage Automatic Computer'의 약어(略語)이고, 최초의 상업용 컴퓨터인 UNIVAC은 'Universal Automatic Computer'의 약어이다.

[6] 원문에는 'a mixed blessing'이라고 되어 있다.

른 한편으로는 (이제는 소위!) '자연 언어'와의[7] 그 유사성은[8] 또한 심각한 오해를 불러 일으켜 왔다. 왜냐하면 비형식적인 자연 언어는 그 약점과 강점 모두가 그 언어의 막연함이나 모호함, 부정확함 등에서 나오기 때문이다.[9])

역사적으로 볼 때, 프로그래밍 언어가 현재의 자동 컴퓨터에게 명령을 지시하는 수단으로 쓰일 수 있다는 앞 단락의 마지막 관점이야말로 오랫동안 그 언어의 가장 중요한 특성으로 간주되어 왔다.[10] (따라서) 현재 가지고 있는 컴퓨터에서 어떤 프로그래밍 언어로 쓰여진 프로그램을 수행할 때의 효율(效率)이 그 언어의 질(質)을 판단하는 가장 큰 기준이 되었다! 이러한 상황의 유감스런 결과로서, 어떤 프로그래밍 언어로 쓰여진 프로그램을 이해하여 다루는 데 있어서의 용이성을 희생시켜 가면서까지, 현존하는 기계에 포함되어 있는 변칙(變則; anomaly)들이[11] 그 언어에 그대로 반영되어 있는 것을 드물지 않게 발견할 수 있다(마치 그러한 변칙이 없으면 프로그래밍하는 것이 어렵지나 않은 것처럼!).[12] 우리들의 접근 방식은 균형(均衡)을 회복하려는 것이

---

[7] 원래는 '언어'라고 하면 당연히 한국어나 영어 등과 같은, 사람 사이의 의사 소통을 위한 자연 발생적인 언어를 지칭했었다. 그러나 프로그래밍 언어처럼 인간과 컴퓨터 사이의 의사 교환을 위한 '인공 언어(artificial language)'가 등장하고 나서는, 한국어나 영어 등은 '자연 언어'로 이제는 불리게 되었다는 뜻이다.

[8] (자연) 언어학의 기본적인 이론이나 용어를 프로그래밍 언어의 연구나 토의에 차용할 수 있는 그 유사성을 말한다.

[9] 만약 자연 언어에 이러한 측면이 없다고 한다면 문학적인 표현은 설 자리를 잃을 것이다. 수사학(修辭學)이라는 학문 분야는 없어질 것이고, 많은 정치인들은 크나큰 곤란을 겪을 것이다. 그 반면에 프로그래밍 언어와 같은 인공 언어는 의사 소통의 한 대상이 인간이 아닌 컴퓨터인 경우가 많기 때문에, 기본적으로 언어에 모호한 부분이 포함되어 있어서는 절대 안 된다. 따라서 자연 언어와 프로그래밍 언어는 비슷한 부분도 많이 있지만 상이한 부분도 많은 것이다. 그러나 이름만 보면 같은 '언어'이기 때문에 오해를 생기게 할 수 있다는 뜻이다.

[10] 앞의 세 단락에 걸쳐서 논의한 형식화된 표기 방법—곧 프로그래밍 언어—의 장점 중 제일 마지막 것이 현실적으로는 가장 중시(重視)되어 왔다는 것이다.

[11] 잘 알다시피 많은 하드웨어(전체로 본 컴퓨터 시스템에서부터 마이크로프로세서(microprocessor)에 이르기까지)가 현재의 시각에서 볼 때, 이론적인 면이나 실질적인 면에서 이해할 수 없는 설계나 구조를 포함하고 있다. 특히 컴퓨터 관련 기술이 몇십 년 동안 급속한 발전을 해 왔기 때문에, 비교적 초기(初期)에 완성된 하드웨어에서 이런 것을 많이 볼 수 있다. 그 이유는 하드웨어가 설계될 당시의 기술 수준이 이론적으로 바람직하다고 생각되는 것을 현실화(現實化)시킬 수준이 되지 못했거나, 기술 수준은 되었으나 미래의 기술이나 시장 동향을 잘못 예측한 것 등이다.

[12] 어떤 언어로 쓰여진 프로그램이 기계에서 수행될 때의 효율성이 언어의 질을 판단할 때 가장 중요시되었기 때문에, 비록 인간이 프로그램을 작성하거나 작성된 프로그램을 이해하는 데는 도움이 되는 기능이라 하더라도, 수행시 효율이 나쁘면 결국 좋지 않은 것이 되었다. 마찬가지 이유로, 기계 설계시 어떤 이유로 포함된 변칙적인 부분도 그것을 효율적으로 사용할 수 있게 하기 위해서 이후 그 기계에서 사용될 언어의 기능에 거꾸로 영향을 미쳤다. (하드웨어의 구조가 심지어 프로그래밍 언어의 설계에 영향을 미친 예도 많이 있다. 예컨대 초기 포트란의 인식자(認識子; identifier) 길이 제한은 당시 구현용 기계인 IBM 704의 한 단어(單語; word)에 인식자 이름을 집어 넣기 위한 것이었다.) 그렇게 포함된 변칙적인 부분이 언어에서 발견할 수 있었다는 것이다. 마지막 괄호 안에 든 것은 원문에 "as if programming without such anomalies was not already difficult enough!"으로 되어 있다. 우리 말과 영어의 어순 차이 때문에 이 문장의 번역문이 들어갈 적당한 장소를 만들기가 어려웠다. 문장의

고,[13] 그러기 위하여 우리 알고리즘이 실제로 컴퓨터에 의해 수행될 수도 있다는 사실은 우리들의 원래 고려에서[14] 중심적인 위치를 차지하고 있었던 것이 아닌, 그저 우연히 일어난 기쁜 상황으로 간주하기로 한다.[15] (PL/I 프로그래머를 위한 최근의 한 교본(敎本)에는 '프러시져의 호출은 프로그램(의 수행)을 매우 비효율적으로 만들므로' 가능한 한 그것을 피하라고 강하게 권고하고 있다. PL/I에 있어서 프러시져가 구조를 표현하는 주된 도구 중 하나임을 감안하면, 이것은 얼토당토 않은 충고이다. 너무 당치 않아 나는 (그런 권고를 담고 있는) 그 책을 '교본'이라고는 거의 부를 수 없다.[16] 만약 당신이 프러시져라는 개념의 유용성을 확신하고 있는데 주변의 구현에서는 온통 프러시져의 메커니즘으로 인한 비용이 너무 과다(過多)하다고 한다면, 그러한 구현을 표준으로 받아들일 것이 아니라 그 부적절한 구현을 비난해야 한다! 참으로, 균형을 회복할 필요가 있다!)

나는 프로그래밍 언어란 것을 (매우 복잡해질 개연성이 있는) 추상적 메커니즘을 서술하기 위한 수단이라고 주로 생각한다. 0장 '수행의 추상화'에서 보았듯이, 알고리즘의 특출(特出)한 점은 그 메커니즘(의 옳음)에 대한 우리의 믿음이 압축이 가능한[17] 논증에 근거할 수 있다는 것이다. 이 간결성(簡潔性)을 상실했을 때 알고리즘은 그 '존재 이유'의 대부분을 잃어 버리는 것이므로, 우리는 의식적으로 그 간결성을 유지하려고 할 것이다. 마찬가지로 우

앞 부분에 '프로그램의 용이성' 얘기가 나오는 뒤에 들어가는 것이 적당하나, 전체 문장이 원문과는 절의 순서가 바뀌어서 그것이 곤란했다. 그 뜻은, "프로그래밍이라는 것은 매우 어려운 작업이기 때문에—이 말에 동의하지 않을 사람은 없을 것이다!—설령 이론적으로 보아 아무런 변칙적인 요소도 포함하고 있지 않은, 인간 본위로 훌륭하게 설계된 프로그래밍 언어라 할지라도, 그 자체만으로 프로그래밍이라는 작업을 무조건 쉽게 만들 수는 없는 것이다. 하물며 인간의 지적 이해에 방해가 되는 여러 가지 (기계에서 영향받은) 변칙적인 요소를 포함하고 있음에야!"라고 강조하고 있는 것이다.

[13] 여태(이 책이 쓰여질 때)까지의 상황은 기계 본위라고나 할까, 효율적인 기계의 사용이 중시되어 왔지만, 저자는 인간의 입장을 중시하여 프로그램의 설계나 그 정합성의 확신—이런 것들이야말로 인간의 진정한 지력(知力)을 필요로 하는 부분들일 것이다!—과 같은 작업을 효율적이고 용이하게 하느냐 하는 점을 어떤 가치의 판단 기준으로 삼겠다는 것이다. 사실 이러한 관점은 오늘날에 와서는 당연한 것으로 받아들여지기도 하지만 다익스트라가 이 책을 저술하는 시점까지도 본문의 뒤에 있는 설명에서 나오겠지만 그렇게 보편화된 것은 아니었다. 물론 저자도 기계의 효율적인 사용이란 측면을 완전히 도외시하는 것은 아니고, 당시의 시각이 (저자가 보기에는—오늘날의 우리가 보기에도) 너무나 기계 쪽으로 기울어진 것이었기에 그 반대 입장을 균형을 회복할 목적으로 강조하는 것이다.

[14] 알고리즘을 설계할 때의 고려 사항을 말한다.

[15] 극단적으로 말해서, 어떤 표기 방법을 사용해 서술된 알고리즘이 컴퓨터에 의해 ('자동적으로') 수행될 수 있으면 좋은 일이지만, 현존하는 컴퓨터에서는 그 알고리즘을 수행할 수 없다고 하여도 상관없다는 뜻이다—만약 그 표기 방법이 알고리즘을 자연스럽고 이해하기 용이하게 표현할 수 있으면. (혹은 앞의 문장에서 '수행된다'는 것을 '효율적으로 수행된다'는 말로 바꾸어 보다 현실성을 띠게 할 수도 있을 것이다.)

[16] 원문에는 '교본'이 'educational text'라고 되어 있다. 그런 말도 안되는 말을 충고라고 적어 놓은 책은 전혀 '교육적(educational)'이 아니라는 뜻이다.

[17] 원문에는 'potential compactness'로 되어 있다.

리가 선택하는 프로그래밍 언어도 그 목표를[18] 지향할 것이다.

[18] 역시 '간결성'을 지칭한다.

2장. 상태(狀態)와 그 특성(特性)의 규정(規定)

오랜 동안 인간은 자연수(自然數)의 특징을 규정해 왔다. '숫자'라는 개념이 처음으로 선사 시대의 우리 선조들에게 떠올랐을 때는, 지칭하고 싶은 각 숫자마다 개별적(個別的)인 이름을 만들었으리라고 나는 상상해 본다. 그것은 마치 현재 우리가 '하나, 둘, 셋, 넷 등등'의 이름을 사용하는 것과 마찬가지로, 다른 많은 숫자들에 대해서도 이름을 가지고 있었음에 틀림없다.

'하나, 둘, 셋'이라는 순서를 살펴 본다고 해서 다음에 오는 것이 '넷'이 된다는 것을 유도(誘導)해 낼 규칙은 없다는 뜻에서, 그것들은 정말로 (말 그대로) '이름'이다.[19] 당신은 그것을[20] 진짜 **알아야만**[21] 한다. (내가 수를 세는—네덜런드어로—법을 완벽하게 잘 알게 된 나이에 이르렀을 때, 영어로 수를 세는 방법은 **배워야만** 했다. 또한 학교에서 시험을 볼 때, 'seven'과 'nine'이란 단어를 아무리 꼼꼼히 들여다 본다고 해서 'eight'이라는 단어의 발음은 물론이고 철자가 어떻게 되는지도 유도해 낼 수가 없었다!)

그러한 비체계적인 명명법(命名法)을 가지고서는 매우 제한된 수의 상이한 숫자들만 구별할 수 있을 뿐이라는 것은 명백하다. 이 한계를 극복하기 위해서 문명화(文明化)된 세계의 모든 언어는 자연수를 위한 (다소) 체계적인 명명법을 도입했고, 곧 수 세는 방법을 배운다는 것은 주로 그 명명법의 바탕을 이루고 있는 체계를 알아내는 것이 된다. 아이가 천까지 세는 법을 알게 되었을 때, 그는 그 천 개의 이름을 (게다가 그 순서대로!) 외운 것이 아니라, (그 명명의) 규칙을 알고 있는 것이다. 임의의 숫자로부터 다음으로 넘어가는 방법을 알게 되어서, (예컨대) '사백 삼십 구'로부터 '사백 사십'으로 넘어가는 것을 아는 순간(瞬間)이 온 셈이다.

숫자를 다루는 용이함의 정도(程度)는 우리가 그것을 위해 선택한 명명법

---

[19] 어떤 사물의 '이름'은 알다시피 고유 명사이다. 어떤 이름이 왜 그것이어야 하는지에 대해서는 (대부분의 경우) 논리적인 설명을 부여할 수 없다. '하나'와 '둘'이라는 이름들을 현재 우리가 사용하듯이 '1'과 '2'를 지칭하는 것이 아니라, 각각 '2'와 '1'을 지칭한다고 옛날부터 정해져서 내려 왔다고 한다 해도 아무런 문제가 없다. 단지 '1'과 '2'를 지칭하는 것이 서로 다르고, 다른 어떤 것(특히 숫자)과 중복이 되지 않는다고 하면 우리가 그것들을 일관되게 사용하는 한, 문제가 될 것은 전혀 없으니까. '셋' 다음에 '넷'이 온다고 하는 것은 인간이 일상 생활에서 동의한 약속(約束)일 따름이다.

[20] '셋' 다음에 오는 '넷'을 말한다.

[21] '유도하는(derive)' 것이 아니라 '알아야(know)' 한다는 것이다. 우리가 외국어를 배울 때 숫자 세는 법을 습득하는 것을 상상해 보면 된다. 우리는 일단 '0'에서 '9'까지를 나타내는 이름은 '외워야만' 한다. 그리고 나서는 많은 자리의 숫자들을 말하는 '규칙'을 익히게 된다.

에 크게 의존한다. 다음

열 둘 곱하기 한 다스(dozen) = 한 그로스(gross)
열 하나 더하기 열 둘 = 스물 셋
XLVII + IV = LI[22]

과 같은 등식들은 다음

$$12 * 12 = 144$$
$$11 + 12 = 23$$
$$47 + 4 = 51$$

등식들보다 그 성립함을 파악하는 것이 훨씬 더 어렵다. 왜냐하면 뒤에 나온 세 개의 등식은 여덟 살짜리 아이면 누구나 습득할 수 있는 간단한 규칙들로부터 만들어 낼 수 있기 때문이다.

숫자들을 기계적으로 다루는 경우에는, 십진수 체계의 장점들이 더욱 두드러진다. 이미 수 세기 동안, 우리는 답을 표시창(window)에 나타내는 기계적 가산기(加算器)를 사용하고 있다. 그 표시창 뒤에는 열 개의 상이한 위치에 있을 수 있는 여러 개의 바퀴가 있고, 각 바퀴는 열 개의 위치마다 하나의 십진 숫자를 보이게 되어 있다.[23] ('00000019'를 표시하고, 거기에 4를 더해서 '00000023'을 나타내는 일은 이제 더 이상 문제도 아니다. 그러나 대신 '열 아홉'과 '스물 셋'을 표시—적어도 순전히 기계적인[24] 도구만으로—할려고 하면 그것은 문제일 것이다!)

그러한 바퀴 장치의 핵심적(核心的)인 요소는 그것이 열 개의 상이하면서도 안정된 위치를 가질 수 있다는 것이다. 전문 용어로는 이 점은 다양한 방식으로 표현된다. 예를 들어 바퀴는 '10-가 변수(十價變數; ten-valued variable)'라고 부르며, 분명히 하고 싶을 때는 그 (변수가 가질 수 있는) 값들을

[22] 로마 숫자(Roman numerals)에서 L은 50을 나타낸다. 잘 알다시피 V는 5, X는 10을 또한 표시한다. 그리고 어떤 기호의 좌측에 놓이면 그 좌측에 놓인 기호가 나타내는 값을 우측의 기호가 표시하는 값에서 빼는 것을 나타내고, 오른쪽에 놓이면 더하는 것을 나타낸다. 그래서 XL은 40을 나타낸다.

[23] 요즘은 많이 사용하지 않지만, 우리나라에 부가가치세가 도입된 당시에 상점들에서 흔히 볼 수 있었던, 물건의 값을 합산하고 영수증을 발급하던 기계식 금전 등록기를 연상하면 될 것이다. 충분히 잘 알고 있으리라고 생각하지만 노파심으로 약간 더 부가 설명을 하겠다. 예를 들어 계산해야 될 최대값이 1,000 미만인 기계식 가산기라고 한다면, 열 개의 톱니를 가지고 있는 세 개의 톱니 바퀴가 같은 축에 끼워져 있고, 오른쪽에 있는 톱니 바퀴가 한 바퀴 돌면 그 바로 좌측에 있는 톱니 바퀴가 한 톱니만큼, 즉 36도만큼 회전하도록 서로 연결되어 있다. 각 톱니 바퀴의 열 개의 톱니에는 0부터 9까지의 십진 숫자들이 각각 새겨져 있어서, 하나의 톱니만 보일 정도의 틈이 가로로 나 있는 표시창을 이 톱니 바퀴들에 덮어 씌우면 그 표시창에는 세 자리 십진 숫자가 나타나게 된다.

[24] 여기서 '기계적인(mechanical)' 것은 기계 역학에 원리를 두고 있다는 뜻으로, '전기적인' 혹은 '전자적인' 것이 아니라는 말이다. 우리 말에서는 맥락에 따라서 전자적인 장치가 포함되어 있어도 '기계'라는 말을 사용하기도 하기에 이렇게 사족을 붙이는 것이다.

0부터 9까지 나열하기도 한다. 여기서 바퀴의 각 '위치'는 변수의 '값'과 동등하다. 바퀴는 비록 (10개의) 각 위치는 안정되어 있지만 다른 위치가 되도록 회전할 수 있으므로 '변수'라고 부르는 것이다. 즉 '값'이 변할 수 있는 것이다. (이 용어는 유감스럽게도 여러 면에서 오해의 소지가 있다. 첫째, 항상 10개의 위치 중 하나에 있고 따라서 (거의) 항상 하나의 '값'을 나타내는 그러한 바퀴는 수학자들이 '변수'라고 부르는 개념과는 상당히 다르다. 대개 수학에서 말하는 변수란 특정한 값을 결코 나타내지 않기 때문이다. 예를 들어 임의의 정수 n에 대해서 $n^2 \geq 0$은 참이라고 말할 때의 이 n은 판이한 성격을 갖고 있는 변수이다.[25] 둘째로, 우리가 (이 책과 전산학 분야에서) '변수'라는 용어를 사용할 때는 건드리지 않는 한 그 값이 상수로 남아 있는, 지속적으로 존재하는 어떤 것을 나타낸다![26] '가변 상수(可變常數)'[27]라는 용어가 보다 좋았을 수도 있었지만, 이미 확고하게 성립된 관례(慣例)를 따르고 그 용어는 도입하지 않겠다.)

(거의) 항상 10개의 상이한 위치 혹은 '상태' 중 어느 하나에 있는, 그러한 바퀴 장치의 본질을 전문 용어를 빌려 나타내는 또 다른 방식은 그 바퀴와 '10개의 점으로 이루어진 상태 공간(狀態空間)'을 연계하는 것이다. 여기서 각 상태(위치)는 '하나의 점'과 대응하고, 이 '점들'의 집단을 '공간'—이것은 수학적 관례와도 일치한다—이라고 부르거나 좀 더 구체적이고 싶으면 '상태 공간'이라고 부른다. 변수가 가능한 값들 중의 하나를 (거의) 항상 가진다고 표현하는 대신, 이제는 이 변수로 구성된 체계가 그 상태 공간의 점들 중 하나에 (거의) 항상 머물러 있다고 표현할 수 있다. 상태 공간은 어떤 체계의 자유로운 정도를 표시한다. 그 상태 공간 외에는 그 체계가 머무를 곳은 아무데도 없다.

하나의 바퀴에 대한 얘기는 그쯤 해 두고, 이제 그런 바퀴 8개가 한 줄로 꿰어 만들어진 금전 등록기로 주의를 돌리기로 하자. 이 8개의 바퀴는 각각 10개의 다른 상태 중 하나에 있을 수 있으므로, 이 금전 등록기는 전체로 볼 때 100,000,000개의 상이한 상태 중 하나에 있을 수 있으며, 그 각 상태는 표시창을 통해 보이는 수(혹은 여덟개의 숫자들의 열)에 의해 적절하게 인식된다.

각 바퀴의 상태가 주어지면, 전체로서의 그 금전 등록기 상태도 유일(唯一)하게 결정된다. 역(逆)으로, 전체로서의 금전 등록기의 한 상태에 대해서 각 바퀴의 상태는 유일하게 정해진다. 이러한 경우, 8개의 개별 바퀴의 상태 공

---

[25] 말할 필요도 없이 이 수학에서의 변수 n은 특정한 값을 '어느 순간이라도' 결코 나타내는 것이 아니다. 저자 서문에서 그런 얘기가 나왔었지만, 이러한 괴리(乖離)는 근본적으로 수학과 컴퓨터의 성격이 차이가 나는 데서 유래한다. 수학의 정적(靜的)인 면과 컴퓨터의 동적(動的)인 면이 맞지 않는 것이다. 이 예에서도 나왔지만, 컴퓨터의 동적인 면이 가장 두드러지게 나타나는 것이 '변수—치환문—반복 구조'의 연장선이다.

[26] 물론 이러한 뜻으로 '변수'라는 용어를 사용하는 것은 수학에서의 사용과는 판이한 것이다.

[27] 원문에는 'changeable constant'라고 되어 있다. 상당히 묘한, 자가당착적(自家撞着的)인 이름이란 생각이 들기도 할 것이다.

간들의 '데카르트 곱'(이 용어는 이미 앞의 장에서 쓴 바 있다)을 형성하여 전체로서의 그 금전 등록기의 상태 공간을 얻는다(혹은 구성한다)고 말한다. 그러한 상태 공간 내의 점들의 총수(總數)는, 그 공간을 구성하는 상태 공간들 내의 점들의 수를 곱한 것이 된다(이것이 그 상태 공간을 데카르트 곱이라고 부르는 이유이다).

그러한 금전 등록기를 10^8개의 상이한 값을 가질 수 있는 하나의 단순(單純) 변수로 생각할지, '바퀴들'이라고 부르는 8개의 10-가 변수들로 구성한 하나의 복합(複合) 변수로 생각할지의 여부는 그것에 대한 우리의 관심(이 어떠한 것인가)에 달려 있다. 만약 우리가 (그 금전 등록기에) 표시되는 값에만 흥미가 있다면, 그것을 나눠지지 않은 하나의 개체로 간주할 것이다. 반면에 톱니가 닳은 바퀴를 교체해야 하는 유지·보수공(維持補修工)은 그 금전 등록기를 (8개의 바퀴로 이루어진) 복합체로 볼 것이다. 이미 우리는 훨씬 작은 상태 공간들의 데카르트 곱으로 하나의 (큰) 상태 공간을 구성하는 다른 예를 본 적이 있는데, 유클리드 알고리즘에 관해 토의하며 판지 위 한 조약돌의 위치는 각각 축 위에 놓이는 두 반쪽 조약돌 즉 두 변수 'x'와 'y'의 조합(혹은 좀 더 정확하게 하자면 순서쌍)에 의해서도 마찬가지로 잘 나타날 수 있음을 보면서였다. (평면 위의 한 점의 위치를 그것의 x-좌표와 y-좌표로 표시한다는 생각은 해석 기하(解析幾何; analytical geometry)를 전개한 데카르트(René Descartes)로부터 나온 것이고, 데카르트 곱이라는 용어는 그를 기념하여 붙여진 이름이다.) 판지 위의 조약돌이라는 것은, 진행되는 계산 과정—유클리드 알고리즘의 수행 같은 것—을 그 체계가[28] 자신의 상태 공간 속에서 움직이는 것으로 볼 수 있다는 사실을 시각화(視覺化)하기 위해 도입되었다. 이 비유를 따라서 초기 상태는 '출발점'이라고도 부른다.

이 책에서는 데카르트 곱으로 구성되었다고 결국 간주될 상태 공간을 가진 체계만을 주로 아니 아마도 전적(全的)이랄 정도로 취급할 것이다. 그렇다고 이것이 상태 공간이 데카르트 곱으로 구성되는 것만이 문제에 대한 유일하고도 최종적인 답이라고 내가 제안하고 있는 것으로 해석되어서는 물론 안 된다. 왜냐하면 그렇지 않음은 내가 너무나 잘 알고 있기 때문이다.[29] 책이 진행됨에 따라 왜 그것들이[30] 그토록 우리의 관심을 끌 만한 가치가 있는지, 왜 그 개념이[31] 많은 프로그래밍 언어에서 그토록 중심적인 역할을 하는지가 명백해질 것이다.

앞으로 나아가기 전에 우리가 차후 맞닥뜨려야만 할 한 가지 문제를 언급

[28] 여러 직선이 그어진 판지와 그 위에 놓인 조약돌로 이루어진 체계를 말한다.

[29] 모든 상태 공간이 데카르트 곱으로만 구성되지는 않는다는 이 진술에 꼭 들어맞는 예는 아니지만, 동일한 상태 공간이라고 하여 그 구성하는 방식이 꼭 유일하지는 않음을 보여주는 간단한 예가 있다. 고등학교 수준의 수학에도 나오리라고 생각되지만, 복소수(複素數) 공간을 나타낼 때 직교 좌표(直交座標)로 나타낼 수도 있고 또한 극 좌표(極座標; polar coordinate system)로 나타낼 수도 있다. 어느 좌표계이든지 복소수 공간을 제대로 구성한다. 둘 중 어느 좌표를 사용할 것인가는 응용 분야의 성격에 의해 결정된다.

[30] 데카르트 곱으로 구성되는 상태 공간을 가지는 체계를 말한다.

[31] 데카르트 곱에 의한 상태 공간 구성을 말한다.

한다. 우리가 데카르트 곱의 형성에 의한 상태 공간을 구성할 때, 우리가 그 안의 모든 점들을 유용하게 사용할 것인지는 전혀 확실치가 않다. 이 점을 보여주는 흔한 예는 어떤 해(年)의 날들을 (month, day) 쌍으로 표시하는 것이다. 여기서 'month'는 12-가 변수('Jan'부터 'Dec'까지)이고 'day'는 31-가 변수('1'부터 '31'까지)이다. 그러면 366일이 넘는 해는 없는 반면에 372개의 점을 가진 상태 공간을 만든 셈이다. 예를 들어 (Jun, 31)을 가지고는 무엇을 할까? 그 점은 허용하지 않는다고 하여 '불가능한 상태'를[32] 마련할 수 있다. 그러면 어떤 뜻에서는 그 체계가 자기모순에 스스로 빠지는 것을 용인하는 것이 된다. 혹은 그러한 점은 '맞는' 날들 중의 하나에 대한 다른 이름이라고 허용할 수 있다. 앞서의 예인 경우는 (Jul, 1)과 같다고 보는 것이다. 이러한 '상태 공간의 미사용 점들'이라는 현상은 우리가 구별하고자 하는 상이한 가능 상태들의 갯수가 공교롭게 솟수(素數; prime number)일 때는 일어나게 되어 있다.[33]

데카르트 곱으로 상태 공간을 형성할 때 자동적으로 도입된 명명법은 우리가 하나의 점을 적시(適示)하는 것을 가능하게 한다. 예컨대 이제 나는 내 생일이 (May, 11)이라고 서술할 수 있다. 그러나 데카르트 덕분에 이 사실을 서술하는 다음과 같은 다른 방식도 이제 나는 알고 있다: 나의 생일은 다음 식

$$(\text{month} = \text{May})\ \textbf{and}^{34}\ (\text{day} = 11)$$

의 해인 month와 day의 쌍 (month, day)이다. 위의 식은 오직 하나의 해만 가지고 있기에 일년 중의 어떤 한 날을 지시하는 방법으로는 다소 번거로운 것이다. 그러나 식을 사용하는 이점은 그 해가 되는 모든 점들의 집합을 (한 번에) 규정할 수 있다는 것이며, 그런 집합은 단순히 하나의 점보다는 훨씬 클

[32]이 이름은 자기모순적(自己矛盾的)인 이름이기에 인용 부호로 강조하였다. 어떤 체계의 상태 공간 내의 모든 점들은 그 체계가 머무를 수 있다고 앞서 나왔었다. 이제는 그 상태 공간의 어떤 상태로는 그 체계가 절대 전이(轉移)해서는 안된다고 말하고 있는 것이다. 원래의 정의에 충실하려면, 애시당초 그러한 상태들을 제외하고 상태 공간을 만들었으면 아무 문제가 없었을 것이다. 그런데 그러한 상태 공간은 데카르트 곱으로 구성할 수 없는 경우가 있을 수 있음을 말하려고 한다.

[33]물론 이 얘기는 데카르트 곱으로 상태 공간을 구성할 때 적용되는 말이다. 너무나 당연한 언급이지만 하나의 예를 들어보면, 우리가 구별하고 싶은 상태의 갯수가 솟수인 7이고 곱해져서 전체 상태 공간을 구성하는 성분 상태 공간이 2개라고 할 때, 우리가 각 성분 상태 공간 내에 포함된 점들의 갯수를 몇 개로 하든지간에 '상태 공간의 미사용 점들'은 생기게 마련이다. 7이 솟수이기 때문에 그러하다. 솟수의 인수는 2개 밖에 없으므로(1과 자기 자신; 7의 경우는 1과 7) 2개의 성분 상태 공간 내의 점들 갯수가 각각 1개, 7개이면 '미사용 점들'은 안 생기겠지만 그런 경우는 굳이 2개의 성분 상태 공간의 데카르트 곱으로 전체 상태 공간을 구성할 필요가 없어진다. 7개의 점을 포함하고 있는 하나의 성분 상태 공간으로 충분할 것이니까, 굳이 데카르트 곱으로 구성해도 장점은 전혀 없고 불편함만 있다. 성분 상태 공간이 3개 이상인 경우도 마찬가지이다.

[34]이 책에서 '**and**'는 논리학에서의 논리곱 또는 공접(共接; conjunction) 연산자를 나타낸다. 그 뜻은 (다 아는 얘기겠지만) 논리곱으로 연결된 두 피연산자가 공히 참일 때만 전체가 참이고 다른 경우는 거짓이 된다.

수가 있다.[35] 시시한 하나의 예로서는 크리스마스(Christmas)의 정의를

$$(\text{month} = \text{Dec})\ \textbf{and}\ ((\text{day} = 25)\ \textbf{or}^{36}\ (\text{day} = 26))$$

으로 하는 것이 될 수 있다.[37] 보다 인상적(印象的)인 예는 내가 월급 받는 날들의 집합의 정의를

$$(\text{day} = 23)^{38}$$

으로 하는 것이고, 사실 이것은 '(Jan, 23), (Feb, 23), (Mar, 23)' 등과 같이 나열하는 것보다는 월씬 간략한 명세(明細)이다.

우리가 상태들의 집합을 규정하기 위해 식을 사용할 때 그 용이함의 여부는, 규정하고자 하는 집합이 상태 공간의 구조와 얼마나 잘 '부합(符合)하는가'[39] 즉 도입된 좌표계와 얼마나 잘 '부합하는가'에 달려 있다는 것은 위의 서술로부터 명백하다. 상기의 좌표계에서는 예를 들어 (Jan, 1)과 같은 요일인 날들의 집합을 규정하는 일은 다소 거북스럽다. 프로그래머의 많은 결정들은 자신의 목표에 적합한 좌표계를 가진 상태 공간을 도입하는 것과 관계가 있으며, 그것을 위하여 자기가 구별해야만 하는 상이한 상태들의 갯수보다 여러 갑절 많은 점들을 가진 상태 공간을 종종 도입하게도 된다.[40]

상태들의 집합을 규정하기 위해 식을 사용하는 또 다른 예를 $\mathrm{GCD}(X, Y)$를 계산하는 판지 기계를 묘사할 때 우리는 이미 보았었다. 그것은 우리가 '해답선(解答線)'이라고 부르는 것 위의 모든 점들을 규정하기 위한

$$x = y$$

였다. 그 점들의 집합은 최종 상태들, 곧 식 $x = y$를 만족하는 상태들의 집합으로서, 그 중 임의의 상태에 도달하면 그리고 그 경우에만[41] 계산 과정은 종료된다는 것이다.

상태 공간의 좌표들, 즉 계산 과정의 진행이 그 값의 변화에 의해 표현되게 되는 변수들 외에도 식에서는 상수('May'나 '23'과 같은)가 있음을 보았

---

[35] 집합을 서술할 때 '원소 나열법'으로 하느냐 '조건 제시법'으로 하느냐의 문제와 같다.

[36] 이 책에서 'or'는 논리학에서의 논리합 또는 이접(離接; disjunction) 연산자를 나타낸다. 그 뜻은 (역시 다 아는 얘기겠지만) 논리합으로 연결된 두 피연산자가 공히 거짓일 때만 전체가 거짓이고 다른 경우는 참이 된다.

[37] 왜 크리스마스가 12월 25일이나 26일인지는 여러 기독교인에게 물어 봤으나 본질적인 이유가 있는지의 여부를 알아낼 수가 없었다. 다만 유럽 여러 지역에서 크리스마스 휴일(休日)을 26일로 하는 곳이 있다는 것만 들었다.

[38] 다익스트라의 월급날은 매달 23일이었는가 보다.

[39] 이 표현은 잘 형식화된 의미를 가지고 있다기 보다는 직관적(直觀的)인 의미를 전달하고 있기 때문에 인용 부호를 붙였다.

[40] 무릇 프로그래밍을 해본 사람이라면 크게 공감(共感)이 가는 서술이리라.

[41] 영어의 'if and only if···'를 '···이면 그리고 그 경우에만'으로 번역하였다.

다. 그것들 말고 '규정되지 않은 상수'로 생각할 수 있는 소위 '자유 변수(free variable)'를 가질 수도 있다. 특히 그것들은 상이한 상태들을 연계시키기 위해 사용되는데, **동일한** 계산 과정의 연속되는 단계들에서 나타나기 때문이다.[42] 예를 들어 유클리드 알고리즘에서 출발점이 (X, Y)인 하나의 **특정한** 수행 기간 동안에는, 모든 상태 (x, y)는 다음 식

$$\mathrm{GCD}(x, y) = \mathrm{GCD}(X, Y) \textbf{ and } 0 < x \leq X \textbf{ and } 0 < y \leq Y$$

를 만족할 것이다. 여기서 X와 Y는 x나 y와 같은 변수가 아니다. X와 Y는 '그것들의 초기값'을 나타내고, 특정한 하나의 계산에 있어서는 상수와 마찬가지이다. 그러나 우리가 조약돌을 격자(格子) 위의 아무 위치에나 두고 유클리드 알고리즘을 시작할 수 있다는 뜻에서는 규정되어 있지 않은 값이다.[43]

몇 가지 용어를 마지막으로 정리한다. 앞서 나온 것과 같은 식을 '조건(條件)' 혹은 '술어(述語)'라고 부를 것이다. (나는 그 두 가지 용어를 구별할 수, 아니 어쩌면 구별해야만 했을 수도 있다. 그 경우 '조건'을 나타내는 형식화된 표현에 대해 '술어'라는 용어를 사용하는 것이다. 그러면 예컨대 두 개의 상이한 술어 '$x = y$'와 '$y = x$'는 같은 조건을 나타낸다고 말할 수도 있을 것이다. 나 자신을 알기에, 내가 이런 타성(惰性)에 흠뻑 빠지리라고는 기대되지가 않는다.[44]) 나는 '어떤 술어가 참인 상태'나 '어떤 조건을 만족하는 상태', 혹은 '어떤 조건이 성립하는 상태' 등의 표현을 같은 의미로 사용할 것이다. 어떤 체계가 조건 P를 만족하는 상태에 도달할 것이 확실한 경우, 우리는 그 체계가 'P의 참을 성립시키는 것이' 확실하다고 말할 것이다.

각 술어는 고려하고 있는 상태 공간의 각 점에서 정의된다고 가정한다: 각 점에서의 술어의 값은 '참'이든지 아니면 '거짓'이며, 그 술어가 참인 모든 점들의 집합을 규정하는 데 그것이 사용되는 것이다.

두 개의 술어 P와 Q가 같은 조건을 나타낼 때, 즉 같은 상태들의 집합을 규정할 때 우리는 P와 Q가 같다(식으로는 '$P = Q$')고 말한다.

특별한 역할을 담당할 두 개의 술어가 있는데 그것들의 이름으로 'T'와

---

[42] 뒤따르는 예를 보면 알겠지만, 어떤 계산을 나타내는 체계가 자신의 상태 공간 내에서 이동할 때, 그 체계가 머무르는 상이한 상태들 간의 관계를 나타내기 위해 '자유 변수'가 사용된다는 것이다. 왜냐하면(또 당연히 그렇게 되어야만 그러한 일을 달성할 수 있겠지만) 이 '자유 변수'들이 각 상태마다 나타나기 때문이라는 것이다. 여기서 '동일한'이라는 단어가 강조되었음을 주의해야 한다. 역시 다음 예를 보면 명확해지겠지만, 하나의 '특정한' 계산의 상이한 여러 상태에서 하는 얘기이지 만약 다른 계산들 간에서 본다면 '자유 변수'가 일반 변수와 구분되는 특성 중의 일부가 상실되므로 안된다. 얘기가 좀 추상적인데 이어서 나오는 예를 보면 이 설명은 명확해질 것이다.

[43] 현대의 프로그래밍 언어에서 보면 부프로그램의 입력 매개 변수가 여기서의 '자유 변수'에 해당한다.

[44] 원문에는 "Knowing myself I do not expect to indulge very much in such a mannerism."이라고 되어 있다. 여기서 'mannerism'은 두 용어—조건과 술어—를 굳이 구별할 필요가 이 책에서는 없음에도 불구하고, 습관에 젖어 이러한 구분을 따르려는 것을 말한다.

'F'를 사용한다.[45]

T는 상태 공간이 관련된 모든 점에서 참인 술어로서, 상응(相應)하는 집합은 전체 집합(universe)이다.[46]

F는 상태 공간 내의 모든 점에서 거짓인 술어로서, 공집합(空集合; empty set)에 대응된다.

[45] 이름에서 짐작하겠지만, 각각 '참'과 '거짓'을 나타내는 술어이다.

[46] 조건이나 술어는 상태 공간 내에서 그 진위를 판별할 수 있다는 점에서 '명제(命題; proposition)'와 같다고 보아도 된다. 고등학교 정도의 수학에서 나오는 것으로서 명제는 그것에 대응하는 '진리 집합(眞理集合)'을 가지고 있다. T라는 명제의 진리 집합은 전체 집합이다. 여기서 전체 집합은 논의의 대상이 되고 있는 상태 공간(혹은 달리 표현하면, 상태 공간 내의 모든 점들로 이루어진 집합)이다.