프로그래밍의 수련
(A Discipline of Programming)

엣져 위베 다익스트라
(Edsger Wybe Dijkstra)

해역(解譯): 김도형

해역자(解譯者) 서문

다익스트라의 '프로그래밍의 수련'의 해역 서문을 쓰게 될 기회를 '결국' 가지게 된 것은 저의 크나큰 기쁨입니다. 왜냐하면 이 책은 제가 전공 분야에서 만난 거의 유일의—해역자가 워낙 고루과문(固陋寡聞)하기 때문에 별로 많이 읽고 많이 보지 못한 탓이겠지만—'고전(古典)'이라는 이름에 값할 만한 것이기 때문입니다. 너무나 당연한 얘기지만, 많은 사람들이 읽는다는 이유만으로 어떤 책을 고전이라고 부를 수는 없을 것입니다. 전산학 분야에서 베스트 셀러로 꼽히는 책들, 예를 들어—국내의 것은 말썽의 소지가 있으니까 생략하고—에이호(Alfred Aho)와 얼맨(Jefferey Ullman)의 컴파일러 책—우리가 흔히 '공룡책(dragon book)'이라고 부르는 것—같은 것이 그 전형적인 것이 되겠는데, 그 책이 잘 쓰여진 훌륭한 책이고 따라서—컴파일러라는 분야의 특성과도 결합하여—앞으로도 상당히 오랜 기간 동안 널리 읽혀질 것이라는 것은 명백한 사실이지만 고전이라고 부를 수는 없다고 생각합니다. 고전이라고 불릴려면 단순히 해당 분야의 연구를 잘 정리했다든가, 입문자에게 쉽게 잘 설명했다든가 하는 점으로는 부족하다고 저는 봅니다. 호어(C. A. R. Hoare)가 쓴 권두언에 나와 있듯이 해당 분야에 관심을 가지는 다른 사람들에게 자신의 통찰력으로 '영감을 줄' 수 있어야 한다고 생각합니다. 해역자의 '고전'에 대한 정의는 상당히 좁게 잡혀 있는 셈이죠. 이 책은 그러한 기준의 고전으로 분류될 수 있는 책입니다.

본 역자가 이 책을 처음 만난 것은 석사 과정 때였습니다. 지금으로부터 10여년 전이었죠. 논문 지도 교수님—최광무 교수님이십니다—의 추천 때문이었습니다. "수필(隨筆) 읽는 기분으로 읽어 봐라"고 하시면서 당신이 가지고 계신 책을 빌려 주셨습니다.

처음 이 책을 읽을 때는 정말 괴로웠습니다. 거의 전혀 이해가 되지 않았기 때문이었죠. 책은 주로 설명하는 문장들로 이루어진 장(章)—진짜 수필같은 장—과 수식들이 대부분인 장이 섞여 있는데, 구별없이 이해가 곤란했습니다. 말로 설명하는 문장의 경우, 뻔히 알고 있다고 생각한 단어들로 이루어진 문장인데도 번역을 해 보면 사전적인 단어들의 잔해(殘骸)만 서로 유기적인 관계가 없이 나열되어 서걱거리고 있을 뿐 내재(內在)한 의미는 전달이 되지 않아서, 거듭하여 사전을 뒤적거리는 일이 되풀이되었습니다. 한 문장 내에서 뿐만이 아니라 문장과 문장, 단락과 단락 사이에서도 마찬가지 상황이었습니다. 한편 수식들을 많이 사용한 장의 경우, 수식들의

기계적인 유도 과정의 각 단계는 이해할 수 있었으나 그 직관적인 의미는 전혀 감이 잡히지 않는 경우가 태반이었습니다. 당연히 억지로 몇 장을 읽은 뒤 포기하고 말았습니다.

이후 몇 달의 시간이 흐른 후, 다시 도전을 해 봤습니다. 이번에는 훨씬 나았습니다. 상당 부분을 이해할 수 있었습니다. 어설프게나마 이해하면서 예닐곱 장을 읽었을 때, 저자가 말하고자 하는 바가 무엇인가 하는 점을 어렴풋이 깨닫게 되었습니다. 그런데 저자의 논지(論旨)는 제가 지금까지 들어 보지 못했던—역시 고루과문해서—주장이고 관점이었으나, 한편으로 생각해보면 너무나 맞는 말들이었습니다. 상당한 충격이었습니다. 처음부터 다시 읽기 시작했죠. 그러자 지난 번까지 이해가 되지 않았던 여러 부분들이 자신의 의미를 드러냈습니다. 사소하다고 할 수도 있지만, 기존의 여러 관념들을 깨뜨리는 언급들이 계속 나타났습니다. 사람의 취향에 따라 또 생각에 따라 조금씩 다르겠지만, 저는 저자의 주장에 전폭적인 공감을 느꼈고 감복(感服)하지 않을 수 없었습니다. 물론 책의 모든 것을 이해할 수는 없었으나, 번역을 생각한 것은 그때였습니다. 제가 느낀 이 감동을 다른 사람에게도 전해 주고 싶었죠. 말할 것도 없이 원서를 읽고 이해하는 것이 최상일 것입니다. 그러나 언어적인 측면의 장벽을 느끼는 사람도 상당수 있을 수 있다는 생각이 들었습니다. 특히 전산학을 학문으로 전공할 것을 결심한 학부(學部) 저학년생들이 다익스트라의 주장을 듣고 생각을 해 보도록 하고 싶었습니다. 그런데 건방진 생각인진 모르겠으나 단순한 번역으로는 많은 사람들에게 저자의 주장을 제대로 전달하기가 어렵다는 생각이었습니다. 많은 보충 설명을 삽입하여, 가능하면 처음 읽는 독자들도 곧장 이해가 가능하도록 하여 시간의 낭비—진정한 낭비라고는 별로 생각되지 않지만—를 줄이고 싶다고 생각했습니다. 그때 이후, 이 생각은 계속 머리에서 떠나지 않았고 이 책도 몇 번 더 읽을 기회가 있었습니다. 아직도 완전한 이해를 했다고는 생각되지 않지만—따라서 번역을 한다는 것이 조금은 무책임한 일이라는 생각도 들지만—잠재적으로 이 책으로부터 많은 감동과 영향을 받을 수 있는 사람들이 기회를 상실하고 기존의 관념 체계(?)에 물들어 갈 것이라는 점이 저를 계속 재촉했습니다. 그래서 눈 딱 감고 부분적이나마 번역과 해제(解題)를 시작했습니다. 이 해역은 그 결과입니다.

막상 우리말로 옮길려고 하니까 너무 어려웠고 그 결과는 창피할 지경입니다. 1976년에 처음 나온 책을 90년대에 와서, 그것도 이 정도 수준으로 밖에 번역을 내지 못한다는 것이 우리나라 전산학의 전반적으로 낮은 수준과 해역자의 능력 부족을 노정(露呈)하는 것 같아서 자괴감(自愧感)이 앞섭니다. 그러나 이 책은 컴퓨터 개론, 프로그래밍, 프로그래밍 언어, 소프트웨어 공학, 운영 체제, 병렬 처리 등의 여러 분야에서 아직도 많은 참조가 되고 있으며, 해역자의 개인적인 의견으로는 이 책의 논지를 따르는 후속 저작이 있었지만 그 출발점으로서의 다익스트라의 저술은 그 가치를 상실하지 않으리라고 봅니다. 원저(原著)의 높은 가치가 현재의 졸역을 어느 정도

보충하기를 바랍니다. 앞으로 잘못된 부분이 발견될 때마다 계속 현재의 번역을 수정할 것을 약속드립니다. 많은 독자들의 지도와 편달이 있기를 빕니다. 번역할 때 사용한 텍스트는 1976년 프렌티스-홀 사(Prentice-Hall, Inc.)에서 나온 엣져 위베 다익스트라(Edsger Wybe Dijkstra)의 'A Discipline of Programming'입니다.

끝으로 책의 구성과 해역의 방침에 대해서 간단하게 설명하겠습니다. 이 책은 0장부터 27장까지 총 28장으로 구성되어 있고, '전혀' 형식적이지 않은 서문이 붙어 있습니다. (이 서문은 상당한 내용을 담고 있습니다. 해역자는 독자들이 이 서문을 주의깊게 읽을 것을 강력하게 권합니다.) 제일 앞에는 호어가 쓴 추천사라고 할 만한 권두언이 있습니다. 본문은 크게 세 부분으로 나뉩니다. 0장부터 11장까지가 1부, 12장부터 25장까지가 2부, 그리고 26장과 27장이 3부를 이룹니다. 1부에서는 여러 가지 문제를 위한 알고리즘 개발에 필요한 기초 내용으로서 저자의 주장의 핵심을 다루고 있습니다. 2부는 1부의 내용을 발판으로 삼아 여러 가지 다양한 문제들을 위한 알고리즘을 실제로 설계해 봅니다. 3부는 부록의 성격을 가지고 있는데, 사용 지침서(manual)의 작성, 일반적으로 구현시 고려할 사항, 컴퓨터의 등장 이후를 회고하는 여러 가지 생각을 서술하고 있습니다. 해역자의 생각으로는 1부가 가장 중요하다고 봅니다. (그래서 본 해역의 일 단계 목표를 1부의 완역(完譯)으로 두고 있습니다.) 번역은 절대적으로 직역(直譯)을 원칙으로 하여 가능한 한 원문을 건드리지 않으려고 하였습니다. 해역자는 저자의 글을 변형시킬 능력도, 자격도 없다고 생각하기에 그러합니다. 정 번역이 매끄럽지 못하거나 내용 전달이 분명치 않은 경우에는, 역시 가능한 한 해역을 통해서 뜻을 전달하고 본문은 그대로 두는 방향을 택했습니다(가끔은 본문에 괄호로 둘러싼 어구나 절을 추가하여 이해를 도울려고 한 부분도 있습니다). 해역자의 경험으로는 이 책의 문장들을 여러 번 읽고 되씹으면서 우러나는 맛을 느꼈기 때문에, 원문을 지나치게 풀어헤친 의역(意譯)을 통하여 그러한 뒷맛을 빼버리고 싶지 않아서입니다. 편의상 주석문에서는 경어를 사용하지 않았습니다. 양해를 바랍니다. 원서에는 각주(脚註)가 전혀 없습니다. 이 해역의 모든 주석은 해역자가 덧붙인 것이므로, 주석 내에 있을지도 모르는 오해나 잘못된 지식의 전적인 책임은 역시 해역자에게 있습니다.

김도형

권두언(卷頭言)

시, 음악, 미술, 그리고 과학과 같은 보다 오래된 지적 수련 분야의 역사를 연구하는 사람들의 경우, 그러한 분야의 몇몇 특출한 종사자들에게 찬사를 바치곤 한다. 그들의 성취는 그들을 존경하는 사람들의 경험과 이해의 폭을 넓히고, 그들을 모방하는 사람들의 재능에 영감을 주고 신장시키는 역할을 해 왔다. 그들의 혁신적인 창조력은, 자기 분야의 기술이나 기법에 대한 높은 숙련도에 더해서 그 기술이나 기법의 근저(根低)에 있는 원리들을 꿰뚫어보는 날카로운 통찰력(洞察力)이 결합된 것이 바탕을 이루고 있다. 많은 경우에 있어서, 그들이 끼친 영향력은 그들의 교양의 폭과 강력하면서도 명징(明澄)한 표현력에 의해서 더욱 증폭되었다고 할 수 있다.

이 책은 컴퓨터 프로그래밍의 본질에 대한 저자의 급진적(急進的)이고도 새로운 통찰을, 저자 특유의 교양 강좌 형식으로[1] 설명하고 있다. 이러한 통찰로부터 저자는 새로운 프로그래밍 방법과 그 표현의 도구를 발전시킨다. 이 방법과 도구는 많은 수의 멋지고 적절한 예들을 통해서 예시(例示)되고 검증된다. 이 책은 컴퓨터 프로그래밍이라는 지적 수련의 발전에 있어서 크나큰 업적 중의 하나로 기록될 것임에 틀림없다.

찰즈 앤터니 리챠드 호어
(Charles Antony Richard Hoare)

[1] 원문에는 'in it's author's usual cultured style'이라고 되어 있다. 독자가 본문을 읽어 보면 느끼겠지만, 마치 얼굴을 마주하고 얘기하는 식의 서술이나, 혼잣말을 계속 섞어 가면서 강의하는 듯도 하고, 또는 어떻게 보면 관련이 없는 사족(蛇足)같아 보이는 언급을 뜬금없이 하는 것처럼 보이는 부분이 상당히 포함되어 있다. 그러나 사실은 그러한 것들이 이 책의 깊은 맛을 이루고 있는 부분이라고 해역자는 생각한다.

서문

오랫 동안 나는 다음과 같은 생각의 연장선 상에 있다고 할 수 있는 책을 쓰고 싶었다. "나는 프로그램이라는 것이 강렬하면서도 심오한 논리적 아름다움을 가질 수 있다는 것을 알고 있다. 그러나 다른 한편으로는 대부분의 프로그램은 기계적 수행에 적합한 방식으로 표현되고 있어, 설령 약간의 아름다움이나마 가지고 있다 하더라도 인간이 그것을 인식하기에는 전혀 적합하지가 않은 것이 현실이라는 것을 싫더라도 인정하지 않을 수 없다. 또 한 가지 못마땅한 것은 알고리즘이 종종 완성된 제품의 형태로 출판되며, 그 설계 과정에서 중요한 역할을 했고 따라서 완성된 프로그램의 최종적인 형태의 이유를 설명하는 고려 사항들의 대부분은 대개 거의 언급되지 않는다는 점이다." 나의 원래 생각은 많은 멋진 알고리즘들을 독자가 그 아름다움을 인식할 수 있는 방식으로 서술하는 것이고, 그 목표를 달성하기 위해서 각 프로그램이 유도(誘導)되는 설계 과정(그것이 실제로 존재하는 것이든 아니면 상상의 것이든 간에)을 묘사하는 것이었다. 이러한 처음의 의도는 이 책이 하나의 문제를 잡고 씨름하여 해결하는 각 장들로 이루어져 있다는 점에서는 지켜졌다고 할 수 있으나, 한편으로는 최종적으로 완성된 책은 처음에 내가 예상했던 것과는 상당히 달라졌다. 허나 이러한 차이가 문제들에 대한 해(解)를 자연스럽고 확신을 줄 수 있는 방식으로 서술하려고 하는 목적 때문에 생겼으므로, 나는 이렇게 작업이 이루어진 것을 후회하지는 않는다.

이런 부류의 책을[2] 접하면 대부분의 사람들은 "어떤 프로그래밍 언어를 사용할 작정이지?"라는 자문(自問)부터 하게 된다. 그런데 사실 이 질문은 표현 양식에만 관련된 단순한 것이 아니다! 쉽게 간과되나 매우 중요한 도구의 측면은 그것을 사용하도록 훈련받은 사람들의 습관에 미치는 도구의 영향력이다. 만약 그 도구가 프로그래밍 언어라면 이 영향력은—우리가 좋든 싫든 간에—우리들의 사고 습관에 미치는 것이 된다.[3] 그러한 영향력을 가능한 한 세밀히 분석하고난 뒤, 나는 기존의 언어나 그 일부분으로

[2]문제에 대한 해, 즉 알고리즘을 기술하는 것을 주된 내용으로 가지는 종류의 책을 말한다.

[3]언어학 분야의 논란이 많은 가설 중의 하나로 '사피어-워프 가설(Sapir-Whorf Hypothesis)'이라는 것이 있는데, 이 가설은 언어의 범위가 사고의 범위를 한정짓는다는 것이다. 그 극단적인 형식은 언어는 철저하고 완전하게 지각을 형성한다는 것이다. 이 가설의 주창자인 언어학자 에드워드 사피어(Edward Sapir)와 벤자민 리 워프(Benjamin

만들 수 있는 어떤 언어도 나의 목적에 부합되지 않는다는 결론을 내리게 되었다. 그렇다고 하여 내가 새로운 프로그래밍 언어를 설계할 수도 없다. 나는 과거 그럴 준비가 전혀 안되어 있었기에 이후 5년 간은 그런 시도를 하지 않겠다고 맹세한 바가 있고, 그 약속한 시간이 아직 지나지 않았음을 분명히 알고 있기에 말이다! (그러나 그 시간이 지나기 전에, 다른 무엇보다도, 이 책을 써야만 했다.) 이 갈등을 해소하기 위해 나는 내 목적에 적합한 '미니 언어(mini-language)'를 설계하기로 했다. 이 언어는 충분한 정당성을 얻을 수 있을 정도로 불가피한 기능만 갖추도록 하였다.

이 망설임과 스스로 부과한 제약(制約)은,[4] 만약 잘못 이해된다고 한다면, 이 책의 많은 잠재적 독자들에게 이 책을 실망스러운 것으로 만들 수도 있을 것이다. 특히 '고수준 프로그래밍 언어'[5] 혹은—한술 더 떠서!—'프로그래밍 시스템'으로 알려진 복잡하고 거창한 도구를 재간있게 사용해야 할 때 만난 어려움을 프로그래밍의 어려움으로 동일시하는 많은 사람들은 더욱 불만족스럽게 느낄 것이다.[6] 내가 그 모든 여러 가지 기능들을[7] 그냥 무시

Lee Whorf) 자신들도 이런 생각을 단언적(斷言的)인 형식으로 진술한 적은 없었지만(가설이니까), 지각을 형성하는데 있어서 언어의 중요성은 누구나 동의하는 바일 것이다. 모든 문화에는 그 고유의 언어가 있고, 또 그 언어는 그 문화의 구성원이 세계를 인식하는 방법에 영향을 끼친다. 언어는 좀 더 거창하게 표현하자면 한 문화가 그 구성원에게 우주(宇宙)의 제요소(諸要素)를 조직화 하도록 가르치는 가장 주요한 수단이다. 분명히 다른 언어 전통에서 자란 사람들은 사물을 보는 관점이 다르고, 다른 것을 본다. 에스키모인은 우리보다 훨씬 많은 눈이 내리기 때문에 사실 우리보다 많은 종류의 눈을 식별하고 그것을 호칭하는 용어도 다양하다고 한다(물론 이것이 '사기'라는 주장도 있지만). 이렇게 되고 나면, 용어의 결여가 다른 지역의 사람들이 많은 종류의 눈을 인식하는 것을 가로막게 된다. 이는 다른 형식적인 측면의 경우도 마찬가지다. 20세기의 천재라고 하는 앨버트 아인슈타인(Albert Einstein)의 독창적 사고의 뿌리를 심리학적으로 연구한 의견 중의 하나는 그가 어릴 때 언어 습득이 상당히 늦은 아이였음을 주목한다고 한다. 결국 인간의 사고를 위한 도구는 언어가 되는데, 아인슈타인은 언어 습득의 부진으로 인해 언어가 아닌 심상(心像)으로 사고하는 것을 배웠다는 것이다. 수년 전 우리나라에서 유행했던 터니 부전(Tony Buzan)의 마인드 맵(mind map) 이론이라는 것도 아마도 비슷한 원리일 것이다.

[4]알고리즘 표현을 위한 언어의 선택과 관련한 망설임과, 그래서 만들어진 미니 언어에 부과한 제약을 말한다.

[5]원문에는 'higher level programming language'로 되어 있다. 일반적인 프로그래밍 언어와 관련한 책에는 'higher level programming language'와 'high level programming language'를 같은 뜻으로 취급하고 우리 말로는 다 같이 '고급 프로그래밍 언어'라고 번역한다. 그러나 여기서는 문맥과 인용 부호의 사용으로 보아 다른 뉘앙스를 가지고 사용되고 있다. 1950년대에 최초의 성공적인 고급 프로그래밍 언어인 포트란(FORTRAN)이 등장한 이후, 1960년대와 1970년대 초반까지 무수한 프로그래밍 언어가 등장했었다. 앞서 등장한 언어에 비해 '보다 고급(higher level)'—이것이 얼마나 모호한 개념인가!—인 것은 미덕이었다. 저자는 이러한 상황을 못마땅한 시선으로 보고 있는 듯하다. 이런 점은 이어서 나오는 글에서도 느낄 수 있다.

[6]그러한 언어나 시스템을 사용할 때도 프로그래밍이 어려웠는데 하물며 그것에 비하면 많은 기능이 결여된 것으로 보이는 미니 언어로 프로그래밍을 하는 것은 얼마나 어렵겠느냐고 생각하기 쉽다는 뜻이다.

[7]원문에는 'all those bells and whistles'로 되어 있다. 이것은 앞서 말한 '고수준 프로그래밍 언어'나 '프로그래밍 시스템'이 포함하고 있는 여러 가지 자칭 '보다 고급'의 기능들을

해 버린다는 이유로 속았다는 느낌이 드는 사람에게는 다음과 같은 질문을 대답으로 대신 하고 싶다: "당신은 정말로 그 모든 기능들, 소위 말하는 '강력한' 프로그래밍 언어의 그 모든 멋진 기능들이 문제를 새로 만들기 보다는 해결하는 데 필요한 것이라고 확신하는가?" 그래도 나는 미니 언어의 사용에도 불구하고 그들이 이 책을 읽어 주기를 바랄 뿐이다. 그러고나면 그들도 앞서 말한 여러 가지 기능들을 제외하고도 다루어야 할 주제가 워낙 많기에, 그러한 기능들의 대다수가 도대체 그렇게 우선적(于先的)으로 도입되었어야 하는 것인지에 대해 의문을 가지게 될 것이다.[8] 프로그래밍 언어의 설계에 특별한 관심을 가지고 있는 독자들에게는, 유감스럽게도 내가 아직 그 문제에 뚜렷한 의견을 표할 준비가 되어 있지 않다고 말할 수 밖에 없다. 그러면서도 한편으로는 당분간은 이 책이 그들에게[9] 영감을 주거나, 범할 지도 모를 실수를 피하는데 도움이 되기를 희망한다.

책을 계속 써—그 과정은 끊임없는 놀람과 흥분의 연속이었다—나가다 보니까, 처음에 염두에 두고 있던 것과는 제법 다르게 되어 버렸다. 처음 시작할 때의 (명백한) 의도는, 내가 프로그래밍 (입문) 강의 시간에 사용한 것보다는 좀 더 형식화된[10] 도구를 사용하여 프로그램을 개발하는 것을 보여주는 것이었다. 강의 시간에는 프로그램의 의미론은 직관적으로 언급되고, 프로그램의 정합성(整合性; correctness)은 대개 엄밀한 논증 부분과 적당히 넘어감,[11] 그리고 변설(辯舌)이 뒤섞인 형태로 다뤄지곤 했기 때문이다. 그러한 보다 형식화된 접근을 위한 기초를 다지는 과정에서 나는 두 가지 주목할 점을 발견했다. 첫째는 내가 (형식화의) 도구로 선택한 '술어 변환자(述語變換子; predicate transformer)'라는 것이, 프로그램 수행 도중에 거칠 수 있는 중간 상태를 전혀 사용하지 않고도 (수행의) 초기 상태와 최종 상태 사이의 관계를 직접 정의하는 수단을 제공하더라는 것이다. 이것은 매우 기쁜 일이다. 왜냐하면 그럼으로 인해서 프로그래머의 두 가지 주요 관심사, 즉 수학적인 면의 정합성(해당 프로그램이 우리가 바라는 초기와 최종 상태 간의 적절한 관계를 정의하는지의 여부—술어 변환자는 실제 계산 과정에

비아냥거리며 지칭하고 있다. 겉으로만 '땡땡거리고 빽빽거릴' 뿐 본질적인 기능과는 아무 관련이 없다는 뜻이다.

[8] 이것은 프로그래밍이라는 것의 본질에 대한 우리의 충분한 이해가 부족함에도 불구하고, '고수준 프로그래밍 언어'의 '보다 고급'의 기능들은 저자가 보기에는 그것과는 거의 관계가 없는 표피적인 부분의 개선이나 확장에만 몰두하고 있다는 것이다. 이것은 본말(本末)이 전도(顚倒)된 것일지도 모른다. 본질을 알고난 후에야 우리는 그러한 기능들이 진실로 문제 해결에 도움이 되는 것인지 아닌지를 판단할 수 있을 테니까.

[9] 프로그래밍 언어의 설계에 관심을 가지거나 실제로 하는 사람.

[10] 원문에는 'formal'이라고 되어 있다. 이 책에서는 이것을 일관되게 '형식화된' 또는 '형식적인'으로 번역한다. 그 뜻은 주지하다시피, '수학적으로 혹은 기호적으로 엄밀히 정의된 논리 체계나 표기 체계를 사용하여 나타낸, 따라서 사용된 기호나 명제의 현실적인 해석은 큰 의미를 가지고 있지 않은'이다.

[11] 원문에서는 'handwaving'으로 되어 있다.

대해서는 언급하지 않고도 이것의 여부를 조사할 수 있는 형식화된 도구를 제공한다)과 공학적인 면의 효율성(이것은 당연히 실제 구현과 연관지어서만 정의될 수 있을 것이다)을 분명하게 구분지을 수 있기 때문이다. 동일한 프로그램이 두 개의 어느 정도 상호 보완적인 해석을 허용한다는 사실은 대단히 유용한 발견이라고 할 수 있었다. 그 두 가지 해석의 첫째는 프로그램을 술어 변환자를 위한 코드로 보는 것인데, 이 책이 다루는 주제를 고려할 때 이 해석이 우리에게는 보다 적당하다. 두번째 해석은 프로그램을 수행 가능한 코드로 보는 것인데, 나는 이러한 관점은 기계에게 미루고 싶다! 두 번째로 주목할 점은 내가 생각할 수 있는 가장 자연스럽고 체계적인 '술어 변환자를 위한 코드'를 설계한 뒤 그것을 '수행 가능한 코드'로 간주했더니 비결정적인(nondeterministic) 구현을 요구하더라는 것이다.[12] 잠시 동안 나는 이제 단일 프로그래밍(uniprogramming)에서도 비결정성이 나타날 것이라는 생각에 몸을 떨었으나(다중 프로그래밍(multiprogramming)에서의 비결정성으로 인한 복잡함을 나는 지나칠 정도로 잘 알고 있었다!), 프로그램을 술어 변환자를 위한 코드로서 해석하는 것은 독자적인 존재 이유가 있는 것임을 깨닫게 되었다. (그리고 돌이켜 보면 과거에 다중 프로그래밍에서 발생했던 많은 문제점들은 우리가 결정성(determinacy)을 지나치게 중시하는 경향 때문에 빚어진 결과라는 것을 관찰할 수 있을 것이다.) 마침내 나는 비결정성을 정상적인 상황으로 간주하게 되었고, 따라서 결정성은 그것의—딱히 흥미롭지도 않은—특별한 한 경우로 귀착되었다.

이렇게 기초를[13] 다진 뒤,[14] 나는 할려고 늘상 생각해 왔던 일, 즉 많은 일련의 문제를 푸는 일을 시작했다. 그 일은 예기치 못했던 즐거움이었다. 그 형식화된 장치[15] 덕분에 나는 내가 하는 것을 과거보다 더욱 완전하게 파악할 수 있음을 경험했다. 또한 알고리즘의 종료(termination)에 대한 명시적인 고려가 매우 큰 휴리스틱(heuristic) 값을[16] 가질 수 있음을—어

[12]이야기의 중복의 위험을 무릅쓰고 조금 더 설명을 붙이면 이렇다. 즉 전술했다시피 프로그램은 두 가지 관점에서 볼 수가 있는데, '술어 변환자를 위한 코드'로 보는 것이 수학적인 정합성을 고려할 때는 보다 적합하다. '수행 가능한 코드'로 보는 것은 구현의 효율성을 고려할 때 보다 맞는 것이다. 그래서 일단 프로그램을 '술어 변환자를 위한 코드'로 보아 (옳고, 자연스럽고, 체계적으로) 설계를 했다. 그런데 그 결과의 프로그램을 실제 구현을 고려하기 위해 '수행 가능한 코드'로 보았더니 그 구현을 위해서는 비결정성(nondeterminacy)이 포함되더라는 것이다. 익히 알다시피 대부분의(사실은 거의 전부) 프로그래밍 언어의 경우 비결정성은 피해야만 하는 성질이다. 다익스트라의 여기 설명을 그대로 따르자면 비결정성이 오히려 자연스러운 것이다. 이것에 대해서는 3장 이후에 또 언급이 있을 것이다.

[13]형식화된 도구를 위한 기초를 말한다.

[14]여기까지가 1부의 내용이 될 것이다.

[15]이 책에서 알고리즘 설계의 도구로 사용하기로 한 술어 변환자를 말한다.

[16]'휴리스틱'의 원래 사전적인 뜻은 잘 알려져 있다시피 '스스로 발견하게 하는'이다. 인공 지능 분야에서 특히 많이 사용되는 개념으로서, '반드시 항상 성립하는 것은 아니지만, 대개의 경우에 문제 해결에 유용한 작용을 하는 경험적 지식을 이용하는 방법의'란 뜻으로 주로 사용된다. 여기서는 알고리즘의 종료를 어떻게 하면 보장할 수 있을까 하는 고려로

느 정도인가 하면, 아직까지도 흔히 발견되는 부분적 정합성을 중시하는 경향이[17] 애석하게 생각될 정도로—발견하는 기쁨도 가졌다. 그러나 다른 무엇보다도 큰 기쁨은, 내가 전에 풀어봤던 문제들 대부분에 대해서 이번에는 과거보다 좀 더 멋진 해를 구할 수 있었다는 것이었다! 이것은 매우 힘이 되는 일이었는데, 그 사실이 이 책에서 개발한 방식이 정말로 내 프로그래밍 능력을 향상시켰음을 나타낸다고 받아들였기 때문이다.

이 책은 어떻게 공부해야 할까? 내가 할 수 있는 최상의 조언은, 하나의 문제가 제시되면 일단 읽는 일을 중단하고 다음 내용을 보기 전에 그 문제를 독자 스스로가 풀려고 시도해 보라는 것이다. 문제를 직접 풀어 볼려고 해야만 그 문제의 난이도를 판단할 수 있는 것이고, 자신의 해를 나의 것과 비교할 기회를 가지게 될 것이며, 그러다 보면 저자의 해보다 나은 것을 찾아 냈다는 만족을 가질 수도 있을 것이다. 그리고 미리 안심시킬 목적으로 하는 말인데, 책이 전혀 쉽게 읽히지 않는다고 하여 너무 기가 죽지 말라는 것이다! 원고를 미리 살펴본 사람들도 상당히 어려운 (그러나 그만큼 수확이 있는!) 부분과 종종 마주쳤다고 했다. 그럴 때마다 나는 그들과 함께 그들이 어렵다고 하는 점을 분석해 보았는데, (어렵다고) '비난받아야 할 것'은 책(즉, 서술의 방식)이 아니라 (책이 다루고 있는) 주제 자체라는 결론에 항상 도달했다. 이 얘기로부터 우리가 배울 수 있는 교훈이 있다고 한다면 그것은 만만찮은[18] 알고리즘은 만만찮을 뿐이라는 것이다.[19] 그리고 프로그래밍 언어로 쓰여진 그 (알고리즘의) 최종적인 서술은 (알고리즘이) 그런 모습이 되는 이유를 설명해 주는 (설계 과정에서 한) 여러 가지 고려 사항에 비하면 매우 압축된 것이다. 따라서 최종 알고리즘의 짧은 길이만 보고 오도(誤導)되어선 안된다! 내 조교 중의 한명은 제안하기를—가치있는 제안일 수도 있을 것 같아서, 그가 한 얘기를 그대로 전달하는 것이다—이 책은 학생들이 작은 그룹을 지어서 함께 공부해야만 한다고 했다. (여기서 말이 나온 김에 책의 '난해(難解)함'과 관련하여 한마디 해야겠다. 나의 과학자로서의 경력의 상당 기간을 프로그래밍이란 작업을 지적으로 보다 잘 다룰 수 있게 할 목적으로, 프로그래머가 할 일이 무엇인가를 명백하게 만드는 데 바쳤음에도 불구하고, 이 노력에 대한 되풀이되는 보답은 놀랍게도(또한 괴롭게도) "다익스트라가 프로그래밍을 어렵게 만들어 버렸다"는 비난이었다. 그러나 그 어려움은 (원래부터) 항

부터 정합성을 갖춘 알고리즘을 설계하는 것이 크게 도움을 받을 수 있다는 뜻이다. 물론 뒤에 본문에서 자세히 다루어질 것이다.

[17] 이러한 경향은 현재에도 여전히 볼 수 있다. 알고리즘의 필수 조건 중의 하나인 종료성을 먼저 중시하는 것이 전체적인 정합성을 가진 알고리즘을 설계하는 데 매우 유익하다는 것이 저자의 얘기이다. 전체적인 알고리즘에서 볼 때 특정한 기능을 수행하는 국부적(局部的)인 부분의 정합성이나 효율성에 지나치게 집착하는 경향을 비판하고 있다.

[18] 원문에는 'nontrivial'로 되어 있다.

[19] 본질적으로 어려운 것은 우리가 어떻게 그 겉모습을 바꾸어 놓던 간에 어려운 것이다.

상 거기 그렇게 존재해 왔고, 그것을 드러내어 밝힌 후에야 우리들은 높은 신뢰성을 가진 프로그램을 설계할 수 있게 되리라는 희망을 품을 수 있다. 그렇지 않다면 우리는 '구질구질한 코드'[20] 밖에 만들 수가 없다. 그러한 코드는 첫번째 반례(反例)를 만나면 폐기돼 버릴 준비가 되어 있는 빈약한 가설과 동일한 상태에 있다고나 해야 할 것이다. 말할 필요도 없이, 이 책에 있는 어떤 프로그램도 실제 기계에서 검사되지 않은 것이다.[21])

내가 왜 미니 언어를 프러시져(procedure)나 순환(recursion) 조차도 포함되지 않을[22] 정도로 그렇게 작게 만들었는가 하는 점에 대해서는 독자들에게 설명이 필요할 것 같다. 하나의 기능이 언어에 더해질 때마다 책에 몇 개의 장이 추가될 것이고 또 그 만큼 (책의) 가격도 비싸질 터이니, 대부분의 생각할 수 있는 기능 확장(예를 들어, 다중 프로그래밍 같은 것)이 행해지지 않은 점에 대해서는 더 이상의 설명이 필요하다고 생각되지 않는다. 그러나 프러시져는 항상 가장 핵심적인 위치를 차지해 왔고,[23] 순환은 전산학 분야에서 학구적인 경모(敬慕)의 대상이[24] 되어 왔으므로, 약간의 설명은 필요하다고 본다.

무엇보다도 먼저, 이 책은 (프로그래밍 분야의) 문외한을 위해 쓴 것이 아니다. 따라서 나는 독자들이 이러한 개념들에[25] 익숙하리라고 기대한다.[26] 두번째로는, 이 책은 특정한 프로그래밍 언어에 대한 입문용 교과서가 아니므로 위의 구조(construct)들이[27] 책에서 빠져 있다거나 또는 그것들의 사용에 대한 예가 없다고 하여, 내가 그것들을 사용하지 못한다거나 사용을 꺼려한다고 생각해서는 안된다. 또한 그것들을 사용할 수 있는 다른

[20] 원문에는 'smearing code'라고 되어 있다. 약간은 사소하지만, 'Smearing program'이 아닌 'smearing code'임을 주의하라. 'Object program' 또는 'object code'처럼 '프로그램'과 '코드'를 혼용하는 경우도 많지만, 구별하는 경우에는 (많이 알려져 있다시피) '프로그램'은 상대적으로 높은 수준의, '코드'는 상대적으로 낮은 수준의 명령어들의 조합을 지칭한다. 정합성에 확신을 가질 수 없는, 신뢰성이 없는 프로그램은 코드라고나 불러야 한다는 뜻을 내포하고 있을 듯 하다.

[21] 모든 프로그램의 정합성에 확신을 가질 수 있도록 애시당초 설계되었으므로 뒤에 따로 검사할 필요가 없었다는 말이다.

[22] 뒤에 보면 알겠지만, 이 책에서 사용되고 있는 미니 언어에는 이러한 기능들이 현재 포함되어 있지 않다.

[23] 이것은 포트란 시절(서브루틴(subroutine)의 사용)부터, 즉 처음부터 그랬다. 하향식(top-down) 접근 방법에 의한 '구조적 프로그래밍(structured programming)'이 제창(提唱)된 이후는 더더욱 그러하다.

[24] 원문에서는 'academic respectability'라고 되어 있다. 이렇게 강한 어구의 선택으로 볼 때, 순환에 대한 학계의 경도(傾倒) 경향에 대해서 저자는 약간은 마뜩치 않아 하는 듯 싶다. 그 이유는 계속되는 서술로부터 유추할 수 있다.

[25] 프러시져와 순환을 지칭한다.

[26] 그래서 다른 특별한 이유가 있지 않다면, 단지 개념을 설명하는 효과 밖에 없을 경우에는 (프러시져와 순환을 포함하지 않고도 하고 싶은 얘기를 충분히 할 수 있으므로) 그 개념들을 포함시키지 않았다는 뜻이다.

[27] 프러시져와 순환을 말한다.

사람들이 그러지 말아야 한다고 제안하는 것도 아니다. 이 얘기의 요점은, 내가 말하고자 하는 바를 전달하는 데 있어서 그것들의 필요를 내가 느끼지 않았다는 것이다. 그 말하고자 하는 바란, 고려 사항의 분별있는 구분이[28] 고품질의 프로그램 설계의 모든 면에 있어서 얼마나 필수적인가 하는 것이다. 미니 언어가 가지고 있는 그 수수한 도구들만으로도[29] 결코 만만치 않은 알고리즘의 매우 만족스런 설계를 위한, 충분하고도 남음이 있는 여지는 이미 제공되었던 것이다.

위의 설명은, 비록 충분하다고 느끼지만, 모든 얘기를 다 한 것은 아니다.[30] 여하튼 간에 나는 반복(repetition)은 태생적으로 구조로서의 존재 이유를 가진다는 것을 말하고 싶다. 사실 그 점에 대해서는 언급이 너무 늦었다는 생각조차 든다.[31] 처음 프로그래밍 언어라는 것이 등장했을 때, 치환문(assignment statement)의[32] '동적(動的)인' 본질은 전통적 수학의 '정적(靜的)인' 본질과 별로 잘 맞지 않는 것 같았다. 적합한 이론의 결여로 인하여,[33] 수학자들은 치환문에 대하여 마음이 편치 못했고, 변수의 값을 치환할 필요성을 만드는 것이 반복 구조이므로 수학자들은 반복에 대해서도 마음이 불편하기는 마찬가지였다.[34] 그래서 치환과 반복이 없는 프로그래밍 언어—초기 리스프(pure LISP)같은 것—가 개발되었을 때, 많은 사람들은 크게 안도감을 느꼈었다. 그들은 다시 익숙한 대지(大地)

[28]수학적인 정합성 측면의 고려 사항들과 공학적인 효율성 측면의 고려 사항들로의 구분 등을 말한다. 기타의 고려 사항들의 구분은 뒤에 다시 설명될 것이다.

[29]프러시저나 순환 같은 '거창한' 도구를 포함하고 있지 않은 미니 언어의 도구(구조)들.

[30]미니 언어에 프러시저와 순환을 포함시키지 않은 것과 관련하여, 프로그래밍 언어에 어떤 구조가 포함되는 것이 필수적인가 또는 바람직한가 등의 논의를 말한다.

[31]이 책에서 프로그래밍 언어는 대부분의 경우 묵시적으로 '절차적(procedural)' 또는 '명령적(imperative)' 프로그래밍 언어를 말한다. 이런 경우 다른 무엇보다도 반복 구조는 프로그래밍 언어에 꼭 포함될 이유가 있다는 뜻이다. 그 이유가 지금까지는 별로 언급이 되지 않았었는데, 이 자리에서 저자는 얘기를 하려고 한다. 물론 지금까지도 반복 구조의 필요성에 대해서는 그것을 포함하고 있는 모든 프로그래밍 언어의 입문서에서 상식적인 설명을 하고 있음은 우리 모두 알고 있다. 하나의 자료 개체에 대해서 동일한 처리를 반복할 필요가 있음을 예를 들어서 설명하는 식이다. 그러나 그것보다 더 근본적인 이유가 있다고 저자는 설명한다. 그 설명은 뒤에 계속하여 나올 것이고, 먼저 반복이라는 구조의 태생적인 존재 이유에 대해 의문을 가진 사람들이 있었는데 그 이유를 언급하고 있다.

[32]'Assignment statement'를 '할당문'이나 '배정문'으로 번역하느냐 아니면 '치환문'으로 번역하느냐 하는 것이 좀 생각을 하게 했는데, 여기서는 치환문으로 번역하는 것이 원문에서 사용되는 맥락과 일치한다고 보았다. 예를 들어서 변수의 초기값으로 값을 '할당'하는 일은 치환문이 하는 일과는 성격이 다르다고 저자는 보고 있다. 이 점은 뒤에 10장에서 다시 토의될 것이다.

[33]치환문의 본질을 규명하는 데 적절한 전통적 수학 이론을 말한다.

[34]치환문이 필요한 것은 반복 구조 때문이라는 말인데, 독자들 중에는 이 얘기가 너무나 당연한 것이라고 생각되는 사람도 있을 것이고 잘 이해가 가지 않는다는 사람도 있을 것이다. 사실 단순히 생각하면 자명하게 이해될 수 있다. 반복 구조란 일군(一群)의 동일한 명령이 집합을 여러 번 수행하는 것이다. 절차적 프로그래밍 언어의 핵심 개념은 기억 장치의 반영인 변수와 그 내용을 수정하는 치환문이다. 반복 구조 내의 치환문이 말 그대로 '치환'을 하지 못한다면, 반복문은 무용(無用)하다.

위로 돌아왔으며, 프로그래밍을 확고하고 훌륭한 수학적 기반을 가진 작업으로 만들 수 있다는 희망의 불빛을 보았다고 생각한 것이다. (지금 이 순간까지도 이론적인 경향의 전산학자들 중에는 순환적 프로그램이 반복적 프로그램보다 '더 자연스럽게 도출(導出)되는' 것이라는 느낌이[35] 광범위하게 퍼져 있다.)

여기서[36] 빠져 나오는 다른 대안—즉, '반복'과 '변수 값의 치환'이라는 짝에 건실하고 현실적인 수학적 기반을 제공한다는 것—이 나오기 위해서는 10년의 세월이 추가로 필요했다. 이 책에서 보듯이, 그 결과는 반복 구조의 의미는 숫어듞 간의 순환 관계(recurrence relation)로 정의될 수 있다는 것이었다.[37] 이에 반해 일반적인 순환은 그 의미 정의를 위해서는 숫어 변환자듞 간의 순환 관계가 필요하다. 이 사실이 내가 일반적인 순환을 단순한 반복보다 아예 규모도(order of magnitude)에서 보아 더 복잡한 것이라고 간주하는 명백한 이유이다. 따라서 다음과 같은 반복 구조

"**while** B **do** S"

의 의미를 다음과 같은 순환적 프러시져(알골(ALGOL) 60의 구문을 빌려 서술되어 있다)

procedure *whiledo* (*condition*, *statement*);
begin if *condition* **then begin** *statement*;
whiledo (*condition*, *statement*) **end**
end

의 아래와 같은 호출

"*whiledo* (B, S)"

로 정의하는 것에 대해 (나는) 정말 마음이 편하지 않다.

비록 (그것이) 틀린 것이 아니라 할지라도, 나는 대장간의 망치로 계란을 깨고 싶지 않은 것과 마찬가지 이유에서 (위의 해석이) 아주 편치가 않다. 대장간의 망치가 그 일을 하는데 아무리 효율적이라 하더라도 말이다.[38] 60년대에 이 주제와 연관을 맺었던 이론 전산학자 세대에게는, 위의 순환적 정의가 '자연스런 것'이었을 뿐만 아니라 심지어 '진실(眞實)인 것'이라고 종종 생각되곤 했다. 그러나 반복 개념에 의지하지 않고는 튜링 기계(Turing

[35] '생각'이 아닌 '느낌'이다. 어떤 논리적 근거에 의한 것이 아니다.

[36] 전통적인 수학에 기댈 수 없다고 보여지는, 이 골치 아픈 치환과 반복을 말한다.

[37] 4장에서 설명될 것이다.

[38] 닭을 잡는 데 소 잡는 칼을 쓰고 싶지는 않다는 뜻이다. 왜냐하면 앞서 설명했듯이 순환은 반복보다 그 복잡도가 훨씬 높은 구조이기 때문이다.

machine)가 실행하는 일을 정의할 수 조차 없다는 사실을 고려할 때, 좀 균형을 회복할 필요가 있어 보인다.[39]

참고 문헌의 부재(不在)에 대해서는 설명이나 사과 또는 변명을 표할 필요를 느끼지 않는다.[40]

감사의 말. 책이 의도하는 내용을 기꺼이 토론해 주거나 최종 원고(의 부분들)에 대한 조언을 통해서, 이 책에 직접적인 영향을 끼친 분들은 다음과 같다: 브론(C. Bron), 버스톨(R. M. Burstall), 페이젠(W. H. J. Feijen), 호어(C. A. R. Hoare), 크누스(D. E. Knuth), 렘(M. Rem), 레이놀즈(J. C. Reynolds), 로스(D. T. Ross), 숄턴(C. S. Scholten), 제그뮬러(G. Seegmüller), 워스(N. Wirth), 그리고 웃저(M. Woodger). 그분들의 협력에 대한 나의 감사를 표할 지면을 가질 수 있음은 특혜라고 할 수 있다. 또한 나는 이 기회와 필요한 시설을 제공해 준 버로우즈사(Burroughs Corporation)에 심심한 감사를 드린다. 그리고 아내의 충실한 내조와 격려에 깊은 고마움을 전한다.

엣저 위베 다익스트라
(Edsger Wybe Dijkstra)

뉴넨(Nuenen),
네덜런드

[39] 반복의 가치가 재평가되어야 한다는 뜻이다. 물론 저자도 문제의 성격에 따라 순환이 반복보다 훨씬 자연스러운 알고리즘 설계를 가능하게 해 주는 경우를 부정하는 것은 결코 아니나, 앞에서 설명한 이유로 순환은 그 의미가 수학적으로 건실하게 정의되나 반복은 그렇지 못한 미운 오리 새끼 취급을 당하는 경향은 개선되어야 한다는 뜻이다.

[40] 대단한 다익스트라!