계산 이론과 프로그래밍 언어의 철학적 여정

2019-12551 박재언

[알게된 점]

컴퓨터 과학의 출발점은 기술이 아니라 철학과 수학이었다. 읽은 자료와 강연을 통해 컴퓨터 과학의 핵심 개념들이 철학적 질문과 수학적 형식성에서 비롯되었음을 알게 되었다. 자비에 르루아는 계산 가능성의 기원을 라이프니츠에서 찾고, 괴델과 튜링의 논리 체계로부터 컴퓨터 개념이 출현한 배경을 설명하였다. 오늘날 우리가 사용하는 시스템과 언어는 오랜 이론의 축적과 형식 논리 위에 세워진 것이라는 점이 인상 깊었다.

프로그래밍 언어는 사고를 표현하는 형식 체계로 진화해왔다. ML 언어는 논리 증명 시스템을 위한 메타언어로 설계되었으며, 함수형 구조와 정적 타입 시스템은 사고의 명확한 표현과 오류 방지에 기여한다. 밀너의 인터뷰를 통해 언어의 실용성과 수학적 엄밀함을 어떻게 조화시킬 수 있는지를 이해하게 되었다. 언어는 단순한 도구가 아니라, 인간의 논리적 사고를 드러내고 구조화하는 수단이다.

소프트웨어는 물리와 논리 사이의 독특한 존재다. 르루아는 소프트웨어를 물질처럼 복제 가능하면서도 논리처럼 추상적 구조를 지닌 존재로 설명했다. 수많은 코드, 모듈, 언어 체계로 구성된 현대 소프트웨어는 프로그래밍 언어의 정형 정의와 의미론에 크게 의존하고 있다. 예를 들어 'x = x + 1' 같은 문장은 상태 변화를 전제로 한 의미론 덕분에 타당하게 해석될 수 있다.

앨런 튜링의 업적은 '영감'보다는 '맥락과 재해석'의 결과였다. 강연 요약문은 튜링의 1936 년 논문이 괴델의 정리를 자기 방식으로 재구성한 결과였으며, 천재성보다는 학문적 맥락 속에서 형성된 통찰임을 강조한다. 이를 통해 창의성은 영감이 아니라 깊은 학습과 이론 해석에서 비롯된다는 점을 배웠다.

결과적으로, 컴퓨터 과학은 논리, 언어, 의미론, 철학이 만나는 복합적 학문이다. 프로그래밍은 단지 기능 구현이 아니라 사고를 구조화하고 표현하는 과정이며, 나는 이 과정을 더 깊이 탐구하고 싶다는 생각을 하게 되었다.

[느낀 점]

컴퓨터 과학의 철학적 기초는 그 자체로 매혹적이다. 컴퓨터 과학은 흔히 실용적인 기술로만 인식되지만, 읽은 자료들은 이 학문이 수천 년 전부터 내려온 철학적 질문과 맞닿아 있다는 사실을 알려주었다. 특히 자비에 르루아의 강연에서는 라이프니츠가 언어와 논리를 통해 기계를 상상하고, 계산이라는 개념을 철학의 영역에서 정의하려 했던 역사적 시도가 소개되었다. 나는 컴퓨터 과학이 단지 명령을 입력해서 결과를 얻는 기술이 아니라, 인간의 사고를 형식화하고 그것을 수학적으로 정의하려는 노력 속에서 발전해왔음을 처음으로 실감했다. 이 과정을 따라가다 보니 컴퓨터 과학이라는 학문이 '논리의 구현'이라는 철학적 깊이를 품고 있다는 점에서 큰 인상을 받았다.

튜링을 천재로만 보는 시각은 편견일 수 있다. 강연 요약문에서는 튜링의 1936 년 논문을 단순히 '천재의 영감'으로 포장하는 사회적 통념에 문제 제기를 한다. 그보다는 튜링이 당시의 수학 이론, 특히 괴델의 증명을 배우고 자기 식으로 재구성한 과정이 논문의 핵심이라는 점을 강조한다. 나는 이 이야기를 통해 지금까지의 내 사고방식에도 반성이 생겼다. 우리는 너무 자주 '천재'라는 단어를 통해 위인들을 신격화하고, 그들과 나의 거리를 멀게 만든다. 하지만 튜링조차도 맥스 뉴먼 교수의 강의를 듣고, 기존의 논문을 꼼꼼히 따라가며 개념을 재해석하는 과정을 통해 새로운 모델을 만들어낸 것이다. 그가 '처음'이 아니라 '다르게 본 사람'이었음을 깨닫고 나니, 나도 이론을 반복해서 공부하고 문제를 다시 보는 습관을 들인다면 충분히 의미 있는 통찰에 도달할 수 있겠다는 자신감이 생겼다.

프로그래밍 언어는 단지 명령어의 조합이 아니라 사고를 담는 구조다. 밀너의 인터뷰에서는 ML 언어가 LCF 증명 시스템을 위한 메타언어로 설계되었음을 소개하면서, 단순히 기능을 수행하는 코드가 아니라 논리적 증명을 표현하는 수단으로 언어가 작동하는 과정을 설명한다. 나는 이 내용을 통해 처음으로 '프로그래밍 언어가 사고의 모양을 결정한다'는 말을 실감했다. 예를 들어, 함수형 언어는 상태 변화 없이 입력과 출력의 관계를 정의함으로써 사고방식 자체를 선언적으로 만든다. 반면 절차형 언어는 순서대로 명령을 실행하는 구조에서 사고의 흐름도 그에 맞춰진다. 그렇다면 어떤 언어를 배우느냐에 따라 사고의 프레임이 달라진다는 점에서, 언어 선택은 단순한 도구 선택이 아니라 철학의 선택이라는 사실을 인지하게 되었다.

디자인과 프로그래밍 사이에도 미학이 존재한다. 첫 번째 문서인 '디자인은 왜 중요한가'에서 강조된 '시간을 초월하는 단순함'이라는 개념은 언뜻 보면 시각 디자인이나 제품 설계에 해당되는 이야기 같지만, 코드 작성에도 그대로 적용된다. 예를 들어 내가 작성한 어떤 코드를 시간이 지나 다시 봤을 때 여전히 명확하고 간결하게 읽히는 경우가 있다면, 그것은 그 코드가 '좋은 디자인'을 담고 있다는 의미다. 쓸데없이 복잡하지 않고, 구조적으로 자연스럽고, 재사용 가능한 패턴을 포함하며, 읽는 사람에게 예측 가능한 흐름을 제공하는 코드가 바로 '디자인이 좋은 코드'라는 점은 이 글을 통해 새롭게 인식했다. 앞으로는 코딩을 단순한 기능 구현이 아니라 설계와 표현, 독자의 감정까지 고려하는 행위로 확장해 사고하려고 한다.

함수형 언어는 수학적 사고의 훈련장이 될 수 있다. ML 과 OCaml 같은 언어들은 단지 프로그래밍 언어가 아니라, 수학적으로 증명 가능한 프로그램을 작성할 수 있도록 설계된 구조를 갖고 있다. 나는 평소 함수형 언어가 너무 복잡하다고 생각했지만, 자료들을 읽고 나니 그 복잡함이 오히려 논리적 사고의 정밀함을 만들어주는 틀이라는 점을 깨달았다. 고차 함수나 재귀 함수는 단순한 기능 구현 이상의 사고력을 요구하며, 프로그램의 각 부분을 수학적으로 분해하고 증명하는 태도를 기르게 한다. 이는 마치 논문을 쓰는 과정처럼, 각 문장을 논리적으로 설명하고 정당화하는 작업과 유사하다는 점에서 매우 흥미로웠다. 앞으로는 이 언어들을 단지 '코딩 언어'로만 보지 않고, 내 사고를 단련하는 훈련 도구로 활용할 수 있을 것 같다는 가능성을 보았다.

현대 소프트웨어는 철학, 수학, 사회의 융합 결과물이다. 르루아의 강연에서는 현대의 소프트웨어가 단지 코드의 집합이 아니라, 수많은 과학적 사유와 사회적 조건들이 얽혀 있는 산물이라는 점을 강조한다. 특히 수억 줄에 달하는 거대 코드베이스, 다수의 개발자 간 협업, 다양한 하드웨어와 운영체제 환경 속에서 오류 없이 작동하는 시스템은 단지 기술적인 성취를 넘어선 설계 철학과 조직 운영 철학의 결합이라는 설명에 깊이 공감했다. 내가 생각하던 '개발자'의 이미지는 혼자 코딩하는 엔지니어였지만, 실제 세계의 개발자는 논리, 협업, 조직, 책임, 설계 미학을 동시에 다루는 종합적인 사유를 실천하는 사람이라는 인식으로 전환되었다.

기계가 이해하는 언어가 아니라 사람이 이해하는 언어가 중요하다는 관점에 감동했다. 르루아는 프로그래밍 언어는 기계를 위한 것이 아니라 사람을 위한 것이라고 말한다. 코드는 결국 사람이 읽고 고치며 확장하는 것이기 때문에, 언어는 인간의 사고와 감정을 담을 수 있어야 한다는 철학이 녹아 있어야 한다. 이 말을 듣고 나니 지금까지 내가 써온 코드들이 얼마나 나만 이해할 수 있는 '나만의 언어'였는지를 반성하게 되었다. 내가 지금부터라도 '다른 사람이 함께 읽을 수 있는 언어로 생각을 표현하는 연습'을 한다면, 그것이 진정한 프로그래밍 능력의 시작이 될 수 있다는 점에서 방향 전환의 계기가 되었다.

병렬성과 통신을 수학으로 다룬다는 발상은 전혀 새로운 사고방식을 제시한다. 밀너의 CCS 이론은 단지 시스템의 동시 실행을 다룬 것이 아니라, 두 개체가 어떻게 대화하고 상태를 바꾸며 상호작용하는지를 형식 언어로 정의하려는 시도였다. 이 개념은 단순히 컴퓨터 간 통신을 넘어서, 사회적 상호작용, 조직 내 의사소통, 나아가 인간 관계를 분석하는 틀로까지 확장 가능하다는 점에서 놀라웠다. 프로그래밍 언어가 기계의 언어를 넘어서 사고와 관계의 패턴을 분석하는 수단이 될 수 있다는 시각은, 언어와 수학의 경계가 어디까지 열릴 수 있는지를 상상하게 만들었다.

튜링이 괴델의 영향을 받아 자기만의 해석을 만들어냈다는 점에서 새로운 학습의 모범을 발견했다. 튜링이 한 것은 완전히 새로운 개념을 만들어낸 '영감'이 아니라, 기존의 이론을 자기만의 방식으로 재해석하고, 언어적 장치로 구성하여 명확하게 설명한 것이었다. 그가 1935 년에 괴델의 정리를 접하고, 1936 년에 컴퓨터의 원천 개념을 담은 논문을 발표한 사실은, 창의성이란 공백에서 태어난 것이 아니라 깊이 있는 맥락 학습과 문제의식에서 비롯된다는 것을 보여준다. 이것은 나에게 "공부하는 방식"에 대한 새로운 기준을 제시했다. 기존의 이론을 단순히 이해하는 데 그치지 않고, 그 구조를

내 것으로 재구성하여 새로운 방식으로 표현해보는 것이 진정한 학습이자 창의성의 시작이라는 메시지를 받았다.

결론적으로, 이 모든 내용을 통해 나는 개발자라는 직업이 단지 기술을 다루는 것이 아니라 철학적 사고, 미적 감각, 사회적 책임, 그리고 깊이 있는 학문적 탐구가 함께하는 통합적 존재라는 사실을 실감했다. 코드를 작성한다는 것은 그저 컴퓨터에게 명령을 내리는 것이 아니라, 내 사고를 조직하고 구조화하며, 그 사고를 다른 사람과 공유 가능한 형태로 번역하는 작업이다. 앞으로 나는 프로그래밍을 단지 실습 과제가 아니라, 나 자신을 훈련하고 표현하는 철학적 훈련장으로 삼고 싶다. 이러한 시각의 전환이 나의 앞으로의 학습 방식과 진로 선택에도 큰 영향을 줄 것으로 생각한다.

[궁금한 점]

프로그래밍 언어는 '도구'인가, 아니면 '사고의 체계'인가? 이번 자료들을 통해 언어가 단지 컴퓨터를 제어하기 위한 수단이 아니라, 인간의 사고 구조를 반영하고 표현하는 방식이라는 인식을 갖게 되었다. 그렇다면 프로그래밍 언어는 문법을 가진 수단이 아니라, 인간의 논리와 철학, 심지어 감정까지 담을 수 있는 구조일 수 있는가? ML 언어나 Coq 와 같은 언어들은 논리적 증명과 사고의 전개를 위한 틀로 사용되며, 수학적인 정합성을 확보하기 위한 언어로 기능한다. 이러한 언어를 사용하는 프로그래머는 단순한 명령 작성자가 아니라, 사고와 논리를 조작하는 사상가라고 할 수 있다. 이런 시각에서 본다면, 앞으로 언어 설계는 논리학, 심리학, 언어학과 어떤 방식으로 교차하며 발전할 수 있을까? 프로그래밍 언어는 결국 인간의 지적 구조를 반영하는 거울일까, 아니면 인간 사고를 재편성하는 도구일까?

계산 불가능성은 현실에서 어떻게 작동하는가? 튜링이 증명한 정지 문제처럼 이론적으로 풀 수 없는 문제들이 존재한다는 사실은 잘 알려져 있지만, 실무에서는 이런 한계를 어떻게 넘는가에 대한 구체적인 사례가 궁금하다. 예를 들어, 정적 분석 도구는 어떤 방식으로 정지 문제를 피하거나 제한하여 검사를 수행하는가? 또, 우리가 작성하는 프로그램이 이론상으로는 분석 불가능한 구조를 포함할 수도 있는데, 그럴 때는 안전성을 어떻게 보장하는가? 인공지능처럼 예측 불가능한 조건에서 행동하는 시스템의 경우, 우리는 어떤 수준의 안정성을 '충분하다'고 판단하는 기준을 어떻게 세우는가? 현실적인 시스템 설계에서 계산 불가능성과 안정성, 효율성은 어떤 관계를 맺고 있는지 깊이 알고 싶다.

'천재 신화'는 창의성을 억압하는가? 강연 요약문에서 보듯 튜링은 '천재'라는 수식어로만 소비되곤 한다. 하지만 그의 1936 년 논문은 괴델의 논문을 바탕으로 재해석하고 구조화한 결과였고, 주변의 교육 환경과 학습 맥락이 함께 만든 산물이었다. 이런 관점은 내게 중요한 질문을 던진다. 우리가 지금 배우는 이론들을 단순히 외우거나 이해하는 데 그치지 않고, 어떻게 '내 언어로', '내 구조로' 재구성할 수 있을까? 튜링이 괴델의 기법을 거울처럼 비추어 자신만의 방식으로 새로운 모델을 제시했듯, 나도 기존의 지식을 어떻게 재해석하고 전개해나갈 수 있을까? 이처럼 학문적 창의성은 영감보다 해석력과 구조화 능력에서 오는 것은 아닐까?

좋은 프로그래밍 언어의 기준은 무엇인가? 문법이 간결하고, 실행 속도가 빠르며, 라이브러리가 풍부한 언어가 좋은 언어인가? 아니면 사람의 사고를 잘 표현하고, 프로그램의 의미를 정확하게 담을 수 있는 언어가 좋은 언어인가? 예를 들어, 파이썬은 입문자에게 매우 친숙하고 간결하지만, 대규모 시스템 설계에는 한계가 있다는 지적이 있다. 반면 OCaml 은 형식적으로 매우 정교하지만 학습 진입 장벽이 크다. 그렇다면 좋은 언어란 기능의 효율성과 사고 구조의 표현력 중 무엇을 우선해야 하는가? 또, 언어의 '좋음'이 개인의 경험과 문화에 따라 상대적으로 달라진다면, 보편적인 언어 설계 원칙이라는 것이 존재할 수 있는가?

의미론은 프로그래밍 언어 설계에서 얼마나 중요한가? 르루아는 프로그래밍 언어의 의미론을 수학적으로 정의하는 것이 중요하다고 강조했다. 예를 들어 'x = x + 1'이라는 문장이 수학에서는 무의미하지만, 상태 변화를 포함한 의미론에서는 정당하다. 이는 언어가 단순히 문법적으로 올바른 것이 아니라, 실행 환경과 함께 해석될 때 그 의미가 결정된다는 점을 보여준다. 그렇다면 실제 프로그래밍 언어 설계자들은 의미론을 어떻게 정의하고, 프로그래머들이 그것을 이해하도록 돕기 위해 어떤 메커니즘을 사용하는가? 의미론적 모호성이 줄어든 언어는 오류를 줄일 수 있을까? 그리고 이런 구조가 실제로 컴파일러 최적화나 정적 분석, 자동 증명 도구의 설계에 어떤 영향을 주는지도 궁금하다.

자동 검증 도구의 가능성과 한계는 어디까지인가? Coq, Isabelle 같은 자동 증명 시스템은 수학적 정당성을 코드 수준에서 검증할 수 있게 해준다. 하지만 실제로 이런 도구들이 적용된 사례는 일부 시스템에 한정되어 있고, 보편적인 프로그래밍 환경에서는 아직까지 널리 사용되지 않고 있다. 그렇다면 이 도구들이 실무에 적용되기 위해서는 어떤 조건들이 충족되어야 하는가? 프로그래머가 직접 증명 과정을 설계해야 하는가, 아니면 시스템이 논리 구조를 추론해주는가? 앞으로 인공지능과 증명 시스템이 결합된다면, 완전한 자동화가 가능해질까? 이러한 기술이 보편화될 경우, 개발자는 코드 작성자가 아니라 논리 설계자 혹은 증명 조율자로 변화하게 될지도 궁금하다.

소프트웨어의 복잡성은 인간의 어떤 능력을 요구하는가? 현대 소프트웨어는 수천만 줄의 코드, 수백 명의 협업자, 복잡한 하드웨어 환경 위에 구축된다. 이런 복잡한 시스템을 설계하고 유지하는 데 필요한 능력은 무엇인가? 단지 코딩 능력과 알고리즘 지식만으로 가능한가? 아니면 커뮤니케이션 능력, 구조화된 사고력, 윤리적 판단력, 미적 감각까지도 필요한가? 르루아가 말했듯, 소프트웨어는 물리와 논리 사이의 중간 존재이다. 이런 성격 때문에 개발자라는 직업은 단순한 기술자의 영역을 넘어서 사회적 존재로서 어떤 역할을 감당해야 하는가에 대한 궁금증이 생긴다.

프로그래밍 언어의 진화는 어디로 향하고 있는가? 최근에는 자연어 처리 기반 코딩 도구, 블록 기반 프로그래밍 환경, 노코드/로우코드 플랫폼이 점점 확산되고 있다. 이러한 흐름은 전통적인 텍스트 기반 언어의 시대가 저물고 있음을 의미하는가? 그렇다면 미래에는 언어 없이 개발하는 시대가 올 수도 있는가? 또는 시각적 언어, 음성 기반 언어, 의미 기반 추상 언어가 새로운 표준이 될 수 있을까? 이처럼 언어의 진화는 인간과 기계의 소통 방식, 그리고 창작의 방식까지 바꾸게 될 것이다. 앞으로의 언어는 누구를 위한 언어가 되어야 하는가? 기계가 이해하기 쉬운 언어? 사람이 표현하기 쉬운 언어? 아니면 둘 사이의 번역을 중재하는 제 3 의 언어?

학제 간 융합이 언어 설계에 미치는 영향은 무엇일까? 자비에 르루아는 언어 설계가 수학, 논리, 언어학뿐 아니라 심리학, 사회학, 예술, 디자인과도 연관되어야 한다고 말한다. 그렇다면 실제로 어떤 방식으로 이러한 분야들이 언어 설계에 기여할 수 있는가? 예를 들어 심리학의 인지 이론이 코딩 패턴의 이해도와 피로도에 영향을 줄 수 있을까? 혹은 디자인 이론이 코드 레이아웃, 색 구성, 가독성 향상에 응용될 수 있을까? 다양한 분야와 교차하는 언어 설계는 그만큼 다층적인 사고를 필요로 하며, 이로 인해 프로그래머가 요구받는 사고 능력도 더욱 넓어지게 되는 것은 아닐까?