언어에 대한 두 가지 생각

2006-11866

조강원

이 글은 서울대학교 4190.310 프로그래밍 언어 과목의 2008년 가을학기 3번째 과제를 위해, 프로그래밍 언어와 관련된 4개의 글을 읽고 느낀 점을 쓴 것이다.

1 들어가며

현재 대중적으로 널리 사용되고 있는 프로그래밍 언어는 대부분 기계 중심의 언어이다. 즉, 기계가 어떤 작업을 수행할지를 차례로 나열함으로써 프로그램을 만들게 된다. 대표적인 기계 중심의 언어로 C가 있다. 물론 C에서도 단순한 명령의 나열 외에 여러 문법적 요소들을 지원하지만, 이는 단순히 '명령을 쉽게 내리기 위해' 첨가되어 있는 것이며, 그 기본 원리가 기계 중심이라는 것은 변하지 않는다. 심지어 C++이나 Java와 같은 일부 물건 중심의 언어들도, 명령을 쉽게 내릴 수 있도록 물건이라는 개념을 가미해 놓았을 뿐, 그 속을 파헤쳐 보면 기계 중심의 언어와 다를 바가 없다는 것을 알게 된다.

많은 사람들이 기계 중심의 언어에 너무나 익숙한 나머지, 모든 프로그래밍 언어는 기계 중심이어야 한다는 착각을 하게 된다. 기계 위에서 실행되는 프로그램이라면 당연히 기계 중심적이어야 하지 않은가? 그 외에 다른 방식이 가능하기는 한가? 프로그래밍 언어의 역할은 단지 기계에 명령을 쉽게 내려주는 것이

아닌가? 부끄럽지만 나 역시 몇 달 전까지만 해도 마찬가지 생각을 가지고 있었다.

하지만, 다른 시각에서 만들어진 프로그래밍 언어들도 존재한다. 값 중심의 언어가 대표적이다. 값 중심의 언어는 프로그램이 다루는 여러 가지 값들이 어떻게 만들어지고 사용되는지를 서술함으로써 프로그램을 만든다. ML 등의 언어가 그 예이다. 그 외에 함수 중심의 언어나, 위에서 언급했던 물건 중심의 언어 등 프로그램을 바라보는 여러 가지 시각이 존재한다. 게다가 컴퓨터공학은 이제 겨우 걸음마를 뗀 단계에 불과하며, 앞으로 계속 학문적 발전이 이루어짐에 따라 새로운 시각을 가진 프로그래밍 언어가 수도 없이 만들어질 것이다.

이렇듯 여러 종류의 언어가 존재하지만, 크게 분류하자면 두 가지로 나눌 수 있을 것 같다.[1] 첫째는 기계를 우선적으로 바라보며, 기계를 쉽게 작동시키기 위하여 만들어진 언어이다. 이는 '기계를 바라보고 만든 언어'라 할 수 있겠다. 위에서 언급했던 기계 중심의 언어와 일부 물건 중심의 언어가 그 예이다. 둘째는 논리를 우선적으로 바라보며, 프로그램이 어떤 논리에 따라 움직이는지를 고민함으로써 만들어진 언어이다. 이는 '논리를 바라보고 만든 언어'라 할 수 있겠다. 예컨대 값 중심의 언어의 경우, '프로그램은 여러 개의 값들이 만들어지고 사용되는 것이다.'라는 논리를 바라보며 만들어진 것이다.

읽을거리로 주어진 4가지 글을 통하여 두 종류의 프로그래밍 언어에 대해 좀 더 자세히 알아볼 수 있다. 『The Development of C Language』는 기계를 바라보고 만든 언어의 대표 격인 C가 어떤 식으로 만들어졌는지를 설명하고 있다. 『계산이란 무엇인가에 대한 아이디어들』은 논리를 바라보고 언어를 만들었을 때 어떤 것들이 가능한지를 간단히 설명하고 있다. 『A Basis for a Mathematical Theory of Computation』에서는 계산한다는 것을 어떻게 수학적으로 해석할 수 있는지를 설명하고, 『19th Century Logic and 21st Century Computing』에서는 실제로 Lambda Calculus가 프로그래밍 언어의 논리로써 적용된 사례를 소개한다.

1 여기서 제시된 구분 방법이 널리 통용된다거나 하는 것은 아니며, 어디까지나 필자의 개인적인 의견이다.

여기서는 위의 4가지 글에서 설명한 내용을 바탕으로, 기계를 바라보고 만든 언어와 논리를 바라보고 만든 언어의 사례를 살펴보고, 어느 쪽이 더 옳은, 혹은 더 그럴싸한 방법인지를 논할 것이다.

2 기계를 바라보고 만든 언어

『The Development of C Language』에서는 C의 원조 격인 BCPL이 어떻게 만들어졌고, B, C를 거치면서 어떻게 언어가 발전해 왔는지를 설명한다. 특히 BCPL과 B에 어떠한 문제점이 있었고, C에서는 이를 어떻게 해결했는지 설명한다.

여기서 우선 주목해야 할 점은, 애초에 BCPL이 만들어진 이유가 Multics라는 기계를 사용하기 위해서였다는 것이다. BCPL을 만드는 과정에서 프로그래밍 언어가 어떤 논리를 가져야 하는지는 전혀 고려 대상이 아니었다. 어떻게 하면 Bell에서 새롭게 만들어 낸 Multics라는 기계를 잘 작동시킬 수 있을 것인지, 어떻게 하면 이 기계의 여러 가지 장점들을 잘 살려낼 수 있을 것인지가 유일한 문제였다. 즉, BCPL은 철저하게 기계만을 바라보고 만들어진 언어인 것이다.

이렇듯 논리에 대한 별다른 고려 없이 만들어진 언어인 BCPL과 B는, 곧 여러 가지 문제점에 봉착하게 되었다. 특히 타입의 개념이 없는 것이 가장 큰 문제가 되었다. B에서는 ‘cell’이라는 한 가지 타입만이 존재하는데, 이는 메모리상의 한 Word에 해당했다. 즉, 프로그램에서 다루는 모든 값은 메모리상의 한 Word에 저장되어야 했는데, 그러다 보니 문자나 실수 등 다양한 형태의 값을 다루기가 힘들었다. 값을 다루는 방법에 대한 논리가 언어에 전혀 포함되어 있지 않다 보니 일어난 일이었다.

이를 해결하기 위해 C에서는 타입 시스템을 도입하였다. 처음에는 int와 char, 이들에 대한 배열과 포인터를 지원하는 것이 고작이었지만, 얼마 지나지 않아 여러 개의 값을 하나의 레코드로 묶어 사용할 수 있게 되었다. 함수나 배열을 포인터로 가리키는 것도 가능해졌다. 산술 연산과 논리 연산이 분리됨으로써 미

약하나마 참/거짓을 나타낼 수 있게 되었다. 이는 분명 BCPL이나 B에 비하면 괄목할 만한 변화가 아닐 수 없다.

하지만 기계를 바라보고 만든 언어에 뒤늦게 논리를 첨가하는 것에는 한계가 있었다. 억지로 끼워 넣은 타입 시스템은 여전히 불완전해서, 어떤 변수에 다른 타입의 값을 대입하는 일도 얼마든지 가능하다. 기존 언어의 틀을 유지하면서 문자열을 다루기 위해 어쩔 수 없이 배열을 문자열처럼 취급하는 지금의 방식이 생겨났고, 그 결과 C에서 문자열을 다루기 위해서는 갖가지 일들을 고려해 주어야 한다. BCPL에서부터 관습적으로 전해져 내려온 여러 가지 표기법 역시 코드를 쉽게 이해하는 데 있어 방해가 된다. 포인터가 가리키는 값을 나타내는 연산자인 *가 그 대표적인 예인데, 많은 사람들이 C를 배우면서 포인터가 어려운 개념이라고 생각하는 것은 상당 부분이 그 난해한 표기법 때문이다.

이처럼 기계를 바라보고 만든 언어에 뒤늦게 논리를 첨가하는 것은 매우 어려운 일이다. 『The Development of C Language』는 C가 괄목할 만한 성과를 거두었다는 결론으로 끝이 나지만, 사실 그 내용은 C에 새롭게 타입 시스템이라는 논리를 추가하는 것이 얼마나 제한적이었는지를 보여 줄 뿐이다. 간단히 말하자면, 첫 단추를 잘못 끼운 셈이다.

그런데, 왜 우리는 언어를 만드는 데 있어 논리를 고려해야 하는가? 이러한 의문에 대한 답이 없다면, 기계를 바라보고 만든 언어에 굳이 논리라는 요소를 갖출 필요가 없을 것이며, 따라서 기계를 바라보면서 언어를 만드는 것도 나쁜 선택이 아닐 수 있다. 뒤에서 살펴 볼 나머지 3개의 글은 언어가 논리를 갖춤으로 인해 어떤 장점을 가지는지를 보여준다.

3 논리를 바라보고 만든 언어

『계산이란 무엇인가에 대한 아이디어들』은 프로그램의 실행이라는 것을 이해하기 위해 필요한 4가지 요소들을 언급하고 있다: 같은 프로그램이란 무엇인가,

순서대로 계산한다는 것은 무엇인가, 프로그램의 안과 밖은 무엇인가, 옮겨 다니는 계산이란 무엇인가.

특히 첫 번째의 '같은 프로그램이란 무엇인가'라는 질문은 컴퓨터공학에 있어 중요한 화두로 떠오르고 있다. 나머지 3개의 요소가 추상적인 문제를 해결하려 하고 있다면, 두 프로그램이 같은지 판별하는 문제는 보다 현실적이다. 점점 우리 주위에서 프로그램들의 쓰임새가 늘어남에 따라, 프로그램이 잘못 동작했을 때의 파급력도 커지게 되었다. 이러한 사태를 방지하려면, 우리가 처음에 원했던 프로그램과 실제로 만들어진 프로그램이 동등한지, 같은 기능을 하는지 알아야 할 필요가 있다. 하지만 컴퓨터공학이 생겨난 지 이제 고작 수십 년이 지났으며, 저러한 문제를 해결하기에 우리가 프로그램에 대해 알고 있는 지식은 너무나도 부족하다.

다행히도, 이러한 문제에 대해 오래 전부터 다뤄 온 학문 분야가 있는데, 그것이 바로 수학이다. 지난 수천 년 동안 많은 수학자들이 주어진 명제가 참이라는 것은 무엇인지, 두 가지가 서로 동등하다는 것은 무엇인지 연구해 왔으며, 그 결과 이제는 많은 성과가 누적되어 있는 상태이다. 우리는 두 수가, 두 집합이, 두 함수가 서로 같다는 것을 논리적 추론을 통해 증명해낼 수 있다. 논리적 추론의 과정은 엄밀하며, 정형화되어 있다. 그렇다면 논리적 추론을 통해, 혹은 다른 수학적 방법을 통해 두 프로그램이 같다는 것을 알아내는 것도 가능하지 않을까?

수학을 이용해 이 문제를 해결하기 위해서는, 좀 더 프로그램이 수학적인 논리 하에서 만들어질 필요가 있다. 우리는 두 함수가 같다는 것은 증명할 수 있지만, 두 사람이 쌍둥이라는 것을 수학적으로 증명하지는 못하는데, 이것은 사람이라는 것이 수학적인 정의를 기반으로 만들어진 것이 아니기 때문이다. 마찬가지로 프로그램도 수학적인 논리를 기반으로 만들어지지 않는다면, 수학의 힘을 빌려 두 프로그램이 같은지 증명하는 것도 불가능할 것이다.

『A Basis for a Mathematical Theory of Computation』는 프로그램이 실행된다는 것을 어떻게 수학적으로 해석할 수 있을지에 대한 간단한 예를 보여 준다.

저 글이 설명하고 있는 계산 가능성에 대한 논의는 이 글의 전개에 있어서는 별로 중요한 요소가 아니지만, 계산한다는 것에 대해 수학적인 분석이 가능하다는 점에 주목해야 한다. 컴퓨터공학이 독자적인 분야로 발전해 나가고 있기는 하지만 아직도 그 기저에서 엄밀함을 제공하는 것은 수학의 역할이다. 프로그래밍 언어가 논리를 갖춤으로써, 비로소 우리는 프로그램을 분석하고 이해할 수 있는, 수천 년 동안 검증된 도구인 수학을 사용할 수 있게 된다.

논리를 바라보고 만든 언어는 사람이 이해하기에도 훨씬 직관적이며 자연스럽다. 『19th Century Logic and 21st Century Computing』에서는 Typed Lambda Calculus가 실제로 프로그래밍 언어의 논리로 채택된 사례가 제시되어 있다. 프로그래밍 언어 수업에서 사용하는 nML도 Lambda Calculus의 영향을 받은 언어 중 하나인데, 처음 nML로 프로그램을 만들어 보았을 때, 기존에 사용했던 기계를 바라보고 만든 언어들보다 훨씬 자연스럽다는 느낌을 받았다. 이는 어떻게 생각해 보면 당연한 일인데, 기계를 바라보고 만든 언어가 기계의 작동 원리에 따라 설계된다면, 논리를 바라보고 만든 언어는 사람의 사고 원리에 따라 설계되기 때문이다.

4 마치며

지금까지 기계를 바라보고 만든 언어와 논리를 바라보고 만든 언어에 대한 4개의 글을 통하여, 간단하게 나의 생각을 적어 보았다. 논리를 갖춘 언어는 그렇지 않은 언어에 대해 많은 장점을 가지고 있을 뿐만 아니라, 프로그램을 만드는 사람에게도 더 자연스러운 형태이다. 기계를 바라보고 만든 언어에 논리를 갖추기는 어려운 일일 뿐만 아니라, 그 한계를 극복하기도 힘들다. 아직까지는 기계를 바라보고 만든 언어가 대세를 이루고 있지만, 컴퓨터공학이 수학과의 시너지 효과를 통하여 더욱 발전하기 위해서라도, 이제부터는 논리를 바라보고 만든 언어를 지향할 필요가 있다.