<프로그래밍 언어 과제 #3 - essay>

진화하는 언어

컴퓨터공학부
2005-11866
정병호

서론

언어는 진화한다. 프로그래밍 언어뿐만 아니라 모든 언어가 그렇다. 새로운 개념이 생기면 그것을 표현할 수 있는 언어가 필요하고, 잘 쓰지 않는 말은 고어가 되기도 한다. 이렇게 언어가 진화하는 과정에서 고려되는 것들은 무엇일까. 일단은 의사 전달의 간결성과 명확성에 있는 것 같다. 자신이 생각하는 바를 간단하게, 그리고 다른 뜻과 혼동되지 않도록 정확하게 전달할 수 있다면 그것이 언어를 가장 바람직하게 사용하고 있는 것이라고 생각한다.

그런데 일반적인 진화에서는 작은 문법 하나가 바뀌거나 새로운 단어가 추가되겠지만, 새로운 패러다임이 등장했을 때는 언어 체계 자체가 송두리째 바뀔 수도 있을 것이다. 극단적인 예를 들어 사람이 입으로 말하는 대신 전자기파 신호로 멀리 있는 사람과도 혼선 없이 1Mbps 속도의 대화가 가능하다면, 지금부터라도 입으로 말하는 것을 당장 관둘 것이다. 입으로 말하는 것 이외에 다른 수단을 못 찾아서 그렇지, 사실 우리는 입으로 말하는 것이 상당히 불편한 것이라는 사실을 모르고 있는 것일 수도 있다.

그렇다면 프로그래밍 언어에서는 어떨까? 우리가 모르는 기상천외한 수단들이 존재하는 것은 아닐까.

C의 소심한 진화

1960년대 후반에 MIT와 General Electric사, Bell lab에서 공동으로 추진하는 Multics Project가 있었다. 이 프로젝트는 비용과 시간 측면에서 전망이 좋지 않다고 판단되어, Ken Thompson의 주도 하에 다른 대안을 찾게 되었다.

Thompson은 Multics의 혁신적인 점들을 살려서 컴퓨팅에 적합한 환경을 만들고 싶어 했다. 이에 유닉스를 구상하였는데, 이것을 적용할 PDP-7은 종이 테이프를 읽는 방식을 쓰고 있어서, 프로그램의 개발과 실행에 제한이 있었다. 여기에 원시적인 유닉스 커널과 에디터, 어셈블러, 셸, 명령어 등이 추가되고 나서야 독자적인 프로그램 개발 능력이 생겼다.

이 시기에 McIlroy는 TMG라는 언어를 개발하여 Thompson이 각성하는 계기가 되었다. Thompson은 유닉스에 시스템 프로그래밍 언어가 필요하다고 생각하게 됐다. 그래서 기존에 있던 언어인 BCPL을 바탕으로 B 언어를 개발했다. B 언어는 BCPL을 8KB 메모리에서 구현한 것이며, C와의 차이점은 타입의 유무에서 가장 두드러진다.

B 언어에는 몇 가지 문제점이 있었는데, character handling mechanism의 문제, floating-point arithmetic의 불완전함과 포인터 연산이 느렸던 점 등이었다. 그래서 B의 기능을 확장하여 문자 타입을 추가하고 작고 빠른 컴파일러를 개발하여 보완한 언어가 NB이다. 후에 declaration 방식 등을 더욱 보완하여 이것이 C가 되었다.

이 언어의 역사를 보면, BCPL, B에서 C에 이르기까지 근본적인 체계의 변화는 느껴지지 않는다. 마치 새 옷을 사지 않고 헌 옷을 계속 수선해서 입는 느낌이다. 문제점이 발견되면 고치고, 쓰다 보니 문제점이 또 생기고. 오랜 세월이 흐르면서 다듬어지고 표준화된 C 언어지만, 지금까지 써 왔기 때문에, 과감한 변화가 두려워서, 사회에 만연한 나쁜 풍조처럼 방치하고 있지는 않았을까 생각해봄직하다.

신선한 생각들, 그리고 제대로 된 진화

강의 홈페이지의 읽을거리들에는 읽을 만한 거리들이 많았던 것 같다.

Church's lambda calculus - 람다 계산법은 Church라는 사람이 1930년대에 제안하였고, 이 계산법으로 모든 함수를 일정 규칙에 따라 lambda term으로 표현 가능하다. 이는 곧 컴퓨터 소프트웨어가 계산하는 과정을 수학적으로 모두 표현할 수 있음을 암시한다. 이 중에서도 주목할 만한 것은 바로 함수의 argument로 함수가 들어갈 수 있고, 함수 자체를 결과 값으로 나타낼 수 있다는 것이다. 실제로 Lisp 등의 값 중심 프로그래밍 언어들이 lambda calculus 방식을 쓰고 있다. 함수의 argument로 함수가 들어갈 수 있음은 기존의 틀을 깨는 파격적인 발전으로서 생각의 확장을 일으키게 해 주고, 복잡한 문제를 더 간단하게 풀 수 있을 것만 같은 느낌이 든다. 이런 가능성이 값 중심 프로그래밍의 발전에 크게 이바지했다고 한다.
Typed lambda calculus는 계산하는 과정에서 쓰이는 자료의 타입이 올바른 것인지 판단할 수 있도록 기존의 람다 계산법에 타입이라는 개념을 도입한 계산법이다.

"A basis for a mathematical theory of computation" 에서는 컴퓨터에서 쓰는 mathematical theory에 대해 이야기하고 있는데, 그 중에서도 필자는 recursive function에 관심이 많이 갔다. 이 recursion이라는 개념이 상당히 추상적이고 처음엔 직관적으로 받아들이기 힘들지만, 복잡한 문제를 단번에 풀어버리는 능력이 있다. 특히 ML 숙제를 할 때 거의 모든 함수에 recursion이 들어갔었는데, 몇 가지 케이스에 대한 처리만 해 주면 어떤 크기의 문제든지 결국

엔 풀어내고 마는 것이 경이롭기까지 했다.

그리하여 위의 색다른 발상들을 가지고 프로그래밍 언어의 놀라운 진화가 일어났다. 기계 중심의 언어에서 값 중심의 언어로. 발상 자체를 전환함으로써 그동안의 사고 방식으로는 생각지도 못했던 생각을 생각하게 되었다. 그리고 기계 중심 언어를 쓰면서 고질적으로 앓고 있던 문제들이 자연스럽게 사라지는 효과까지 나타났다.

컴퓨터공학도의 머릿속

위의 참신한 발상들이 이미 옛날 사람들의 머릿속에 들어 있었다는 사실이 필자를 절망 속에 빠뜨리는 와중에, "계산이란 무엇인가에 대한 아이디어들"을 읽고 컴퓨터공학도의 뇌 구조는 보통 사람들하고 달라야 하는구나 하고 깨달음이 있었다. 그리고 아직도 갈 길이 멀다는 것도.

먼저, 같은 프로그램이란 무엇인가(Semantic equivalence). 프로그램은 생각대로 실행되어야 한다. 프로그래머의 생각이 곧 프로그램의 생각이다. 그렇다면 같은 프로그램이란, 두 프로그램이 같은 생각을 가지고 있다는 뜻인가 보다. 프로그램의 마음속을 어떻게 알 수 있을까. 단지 '같은 입력을 주었을 때 같은 출력이 나온다'라고 생각해도 되는 것일까. 좀 더 수학적인 접근이 필요할 것 같다.

순서대로 계산한다는 것은 무엇인가(Sequentiality). 근데 먼저 순서라는 것은 정확히 무엇인지 생각해 봐야겠다. 개별적으로 보면 순서가 있고, 개별적으로 보지 않으면 순서가 없다. 세상은 점점 개별적이지 않은 세상이 되어가고 있다. 병렬적인 세상을 구현하기 위해서는 조물주의 지혜가 필요할 것만 같다.

그밖에도 프로그램의 안과 밖은 무엇인가(Interaction), 옮겨다니는 계산이란 무엇인가(Mobility) 등등, 다른 학문에서는 다루지 않을만한 문제들이 필자의 머릿속을 자극하는 순간이다.

결론

사실 예전에는 앞으로 컴퓨터공학의 미래가 고리타분해질 것이라고 생각을 했다. 이미 고착화된 시스템을 기반으로 물량 싸움의 규모만 점점 커질 것이라고 생각하고 있었는데, 우물 안 개구리 격이 되어 버린 것 같다. 미래를 주도하는 선구자들은 항상 보통 사람들과는 다른 혁신적인 생각을 하고 있음을 망각해서는 안 되겠다.

프로그래밍 언어 강좌를 수강하기 전에는 프로그래밍 언어라는 것이 하나의 틀에서 벗어나지 않는다고 생각했으며, C와 Java의 차이만 해도 엄청나다고 생각을 하고 있었다. 지금이라도 필자의 생각이 틀렸음을 짐작했다니 그나마 다행인

셈이다.

값 중심 언어를 써 보면서, 기계 중심의 언어보다 더 직관적으로 해석이 가능하고, 작성자의 의도를 한 눈에 척 볼 수 있다는 느낌이 들었다. 서론에서 얘기한, 간단함과 명확함을 두루 갖추지 않았나 싶다. 그래도 아직 전반적으로 기계 중심의 언어가 주도권을 잡고 있는 상황에서, 현실에 안주할 것인지 아니면 과감한 변화를 시도해 볼 것인지 흥미롭다.