Programming Languages의 중요성에 대한 깨달음

2008-11699
이중호

1. 'Hard Programming, Hard Programming Languages'

C를 사용해 프로그램을 만드는 것은 결코 쉬운일은 아니다. 특히나 시도 때도 없이 일어나는 segmentation fault 때문에 C로 프로그램을 만드는 일이 짜증이 날 정도로 힘들다. 하지만 대부분의 C 사용자들은 이것을 프로그램을 만드는 일이 힘든 것이지 'C로 프로그램을 만드는 일'이 힘든 것이라고 알아채지 못하고, 나 또한 OCaml을 접하기 전까지는 그래왔다.

2. '기계를 위한 언어 C'

'The Development of the C Language, Dennis M. Ritchie, (HTML) HOPL-II, 1993' 에는 C가 어떻게 개발되었는지에 대해 자세하게 서술되어 있다. 1960년대 후반 MIT, General Electric와 Bell Telephone Laboratories 에서 Multic project를 진행중에 있었다. 하지만 이 프로젝트를 Low-Level language를 사용해서 진행하는데에는 많은 시간과 비용이 들고, 또 그 복잡도가 매우 크기 때문에 불가능하다는 의견이 많았다. 이에 따라 이 프로젝트의 Leader인 Ken Thompson 편리한 computing environment를 만들기를 원했다. 이후 Ken Thompson은 Martin Richards가 만든 BCPL이라는 언어를 발전시켜 B라는 언어를 만들었다. 하지만 B에도 여러 문제점이 발견되자 그는 이를 확장시켜 NB라는 언어를 만들었다. 이후 C가 등장했고 이는 B에서 좀 더 발전된 형태로 타입시스템과 가벼운 컴파일러를 만들 수 있었다. C는 Portability 문제를 해결하여 널리 사용되었으며 특히 Unix의

System Programming에 많이 사용되었다.

하지만 C에도 여러 가지 문제점이 존재한다. C사용자들은 pointer가 매우 강력한 기능이라 말한다. 하지만 이러한 pointer 연산 때문에 Parsing도 어려워지고 메모리관리면에서도 효율성이 떨어져 Segmentation fault 에러가 자주 일어난다. 이러한 문제점을 접하고 논리학의 발달에 의해 2세대 언어들이 탄생하기 시작했다.

사람들은 C의 이러한 문제점이 어디서 나타나는지 파악하기 위해 먼저 프로그램의 실행에 대해 이해하려고 노력했다. '계산이란 무엇인가에 대한 아이디어들'에는 프로그램의 실행을 이해하는 데에 필요한 다음의 네가지 요소를 제시한다.
같은 프로그램이란 무엇인가
순서대로 계산한다는 것은 무엇인가
프로그램의 안과 밖은 무엇인가
옮겨다니는 계산이란 무엇인가
이러한 요소들을 제대로 파악하기에는 아직 부족한 점이 있기는 하지만 오래 전부터 수학이 이를 이해하는데 큰 기여를 했다.

프로그램이라는 것은 논리적이다. 'A Basis for a Mathematical Theory of Computation'에는 프로그램이 실행된다는 것을 수학적으로 해석하는 방법을 보여준다. 수학적으로 해석한다는 것은 그것이 수학적 논리를 어느정도 따르고 있다는 것을 단적으로 보여준다. 따라서 프로그램이라는 것은 논리를 따르고 있다는 것을 의미한다.

C는 논리적이기는 하지만 그 기본적인 틀은 기계중심적이다. Low-Level에서 이용되는 Instruction을 만들어주는 Assembly를 사용하다가 그를 좀더 쉽게 사용하게 만들기 위해 고안된 언어이기에 어찌보면 당연한 결과이다. 이러한 기계중심적 성향에 의해 C로 짠 프로그램이 어떠한 일을 수행할지에 대해 인간이 쉽게 파악 할 수 없다. 또 인간의 생각은 논리적인데 반해 C는 기계중심적이다보니 생각대로 프로그램을 만들기가 쉽지 않다.

3. '인간을 위한 언어 ML'

‘An Interview with Robin Milner(ML designer, Turing Award winner)’를 읽어보면 어떻게 ML이라는 언어가 만들어 졌는지 알 수 있다. ML은 2세대 언어로 기계중심적인 C와는 달리 논리중심적이다. 이는 Lambda Calculus에서 아이디어를 착안하여 만들어졌기에 강력한 리컬시브 기능을 갖고 있다. 특히나 이것으로 만든 프로그램은 논리적으로 증명이 가능하다. 즉 논리중심적으로 만들어진 언어인 것이다.

'Programming Languages‘수업을 처음 들었을 때 당황을 감추지 못했다. 모든 숙제를 ML의 일종인 Ocaml을 사용하여 프로그래밍한다는 것이 나를 당황케 한 것이다. 이전까지 기계중심적인 C와 C++, Java등의 언어를 사용해 왔고, 또 그들에 익숙해져 있는 나로서는 또 새로운 스타일의 언어를 배워야 한다는데 압박감을 가지고 있었다.

하지만 머지않아 Ocaml은 1세대 언어와 달리 사용하기가 정말 편리하다는 것을 깨달았다. 포인터란 기능은 존재하지 않지만 그 기능이 딱히 필요하진 않았다. 이 덕분에 C에서 매번 나를 괴롭혔던 Segmentation fault를 본적이 없고, 또한 메모리도 Garbage Collector가 알아서 관리해주니 메모리에 신경쓰지않고 편하게 프로그램을 만들 수 있었다. OCaml이 1세대 언어와 큰 차이를 보이는 부분이 타입에 있어서 철저하다는 것이다. 이로서 타입 매칭 기능은 실로 엄청난 기능이라 말할 수 있다. 어찌하여 이런 타입매칭이 이루어 질 수 있었을까.

‘19세기 논리학, 21세기 계산학’에서는 Church의 Lambda Calculus와 Gentzen의 Natural Deduction 을 비교하면서 타입체킹에 대한 내용을 중점적으로 다룬다. Gentzen의 Natural Deduction은 premise와 conclude 로 이루어진 Locgic을 사용해 Roundabout Proof를 사용해 주어진 Logic을 증명하는 방법이다. Church의 Lambda Calculus는 모든 함수를 표현할 수 있게 함수의 매개변수로 함수가 올 수도 있고, 함수의 리턴형이 어떤 함수일 수도 있다. 복잡한 함수식을 간단한 형태로 표현 할 수 있다는게 Lambda Calculus의 강점이다. 이러한 Lambda Calculus가 발전하여 Typed

Lambda Calculus가 나오게 되는데, 이는 Lambda Calculus로 표현된 함수의 정의역과 공역의 집합을 표기하는 표기방식이다. 이러한 논리적인 기반으로 ML이라는 언어가 탄생하였고, 강력한 기능인 타입매칭이 이루어질 수 있었다.

ML로 만든 프로그램은 인간이 파악하기에 쉽다. 위에서도 말한 바와 같이 C는 기계중심적이어서 C로 만든 프로그램을 인간이 파악하기 쉽지 않다. 인간은 논리적이기 때문이다. 하지만 ML은 논리학에서부터 나온 언어이기 때문에 프로그래머는 생각대로 프로그램을 짜면 되고, 그러한 프로그램을 ML을 사용하는 사람이라면 쉽게 파악이 가능하다. 따라서 C보다는 ML이 인간을 위해 만들어진 언어라고 할 수 있다. 하지만 전적으로 인간을 위한 언어라고는 할 수 없다.

4. 'Programming Languages의 현주소'

2세대 언어 ML이 탄생했다 하더라도 아직까지 ML에도 많은 단점이 존재한다. Lambda Calculus로 표현되는 것들은 모두 ML로 표현이 가능하다고는 하나, 이 세상의 모든 현상을 Lambda Calculus로 표현 할 수는 없기 때문에 ML은 완벽한 언어라고는 보기 어렵다. 또한 인간이 사용하는 언어와는 아직도 큰 차이가 있다. 특히 이것은 Artificial Intelligent 분야에 있어서 걸림돌이 된다. 예를 들어 자연언어 처리는 현재까지 나와 있는 Programming Language로는 구현이 힘들다. 인간은 이미지를 보고도 한번에 그것이 무엇인지를 파악한다. 하지만 AI를 사용하는 컴퓨터도 이미지를 보고 그것이 무엇인지 파악하지는 못한다. 간단한 미로정도는 인간은 직관적으로 길을 찾을 수 있지만, 컴퓨터는 그것조차 꽤나 많은 계산을 하여야 한다. 분명 미로의 길을 찾는데 걸리는 시간은 컴퓨터가 더 적게 걸릴지 몰라도, 그러한 알고리즘을 인간이 구현하는데 현재의 Programming Language로는 상당한 시간과 노력이 들어간다.

ML도 2세대 언어라는 타이틀을 갖고 있기는 하지만, Programming Language의 비젼을 봤을때 아직도 걸음마를 하는 수준이다. ML이 인간의 논리학을 따라 만들어 졌다해도, 인간은 논릭적인 사고만을 하지 않기 때문에 인간의 언어와 큰 차이를 보인다. 아직도 많은 부분이 수정되어야 하기

때문에 Programming Language의 발전은 끝이 보이지 않는다.

5. ‘Computer Science의 발전을 위해’

Programming Language가 발전한다면 이는 Computer Science분야에 지대한 영향을 줄 것이다. 물론 Computer Science분야에 Programming Language도 포함되기에 이는 당연한 결론이다. 그만큼 Programming Language가 큰 중요성을 갖고 있다는 의미이다. 현재는 Programming 교육을 받은 사람만이 프로그래밍을 할 수 있지만, Programming Language가 인간이 쓰는 언어와 점점 더 비슷하게 만들어질수록, 프로그래밍이 가능한 사람들이 늘어날 것이다. 이렇게 된다면 Computer Science 뿐만이 아닌 다른 분야의 사람들도 Computer Science 분야와 연관된 연구를 할 수 있다. 이렇게 되면 시너지 효과에 의해 Computer Science 분야까지도 점점 더 발전 할 수 있게 된다.

이 글의 서론부분에서 말했듯이 대부분의 프로그래머는 프로그래밍을 하던 중 마주하는 문제에 대해 프로그래밍이 어려운 것이라고 결론 내린다. 대부분의 프로그래머가 1세대 언어만을 접하고 그것에 익숙해져 있기 때문에 언어의 문제점을 제대로 파악하지 못한다. 하지만 이러한 사람들이 Ocaml을 접해보게 되면 프로그래밍이 문제가 아니라 언어가 문제라는 사실을 깨닫게 고 Programming Languages에 관심을 갖게된다.

결론적으로 Computer Science를 발전시키기 위해서는 많은 프로그래머들이 다양한 Programming Language를 접해보고 컴퓨터 언어에 관심을 가지고 그것들을 발전시켜 나아가야 한다.