Programming Language Essay1

전기공학부 2007-11662

김우열

1. 서론 : 프로그래밍 언어란

우리가 살고 있는 이 지구상에는 대략 7천개나 되는 언어들이 존재한다고 한다. 그중에서 그나마 많은 사람들이 쓴다고 하는 언어만 해도 열손가락으로 꼽을 수 없을 정도로 다양하다. 그런데 신기한 것은, 이렇게 다양한 언어들이 존재하고 있음에도 불구하고 서로 다른 나라(혹은 다른 언어를 사용하는 사람)간에 의사소통의 부재는 거의 존재하지 않는다는 사실이다. 이러한 것이 가능한 것은 바로 '번역'이라는 것이 있기 때문이다.

사람과 컴퓨터간의 관계도 서로 의사소통을 하려는 두 사람의 관계와 똑같다. 컴퓨터에게 언어는 0,1로만 이루어진 기계어일 것이다. 그러나 사람이 직접 기계어로 프로그램을 만드는 일은 없다. 이것이 가능한 것은 바로 '프로그래밍 언어(이하 PL)'라는 일종의 '번역자'가 존재하기 때문이다. PL은 컴퓨터를 '이용'하여 구동되는 프로그램을 만드는데 사용되는 언어이다. 즉, 우리는 PL을 이용하여 원하는 일을 하도록 코딩을 하게 되면, 해당 PL의 컴파일러가 이를 기계어로 번역을 시켜주고 컴퓨터는 그 내용을 받아서 처리하게 되는 것이다. 이러한 목적을 가지고 있는 언어는 대단히 많다. 흔히 많은 사람들이 알고 있는 C, C++, Java부터 시작해서 Perl, Python, Lisp, O`Caml뿐만 아니라 XML, HTML같은 것들도 일종의 PL로 볼 수 있다.

그런데 여기서 문제가 되는 것은 바로 '번역자가 얼마나 내가 원하는 내용을 정확히 전달하는가?'이다. 그리고 그 답은 현재까지 널리 알려진 PL이 얼마나 많은가를 보면 알 수 있다. 만약 우리의 뜻을 정확히 컴퓨터에게 전달할 수 있는 PL이 있다면 더 이상 새로운 PL이 나올 이유가 없을 것이다. 즉, 인간들은 아직 완벽한 PL을 만들지 못하였다고 보는 것이 옳다고 볼 수 있다. 그리고 이것이 우리가 '프로그래밍 언어' 첫 수업시간에 교수님에게 '프로그래밍 언어는 미개하다'라는 듣게 되는 원인이 되었다.

나는 전기공학부 학생으로서 처음에 이 말을 들었을 때, 상당한 충격을 받을 수밖에 없었다. 전기공학부에서는 (안타깝게도) 거의 모든 프로그래밍 관련 수업이 C++로만 진행된다. 즉, C++는 우리에게는 PL의 전부인데 이것을 교수님께서는 '1세대 언어 중에서도 미개하고 복잡한' 언어라고 하셨기 때문이다.

그러나 점차 수업이 진행되고 Essay과제를 받아서 읽어본 6개의 Article들에 의하면, 내가 알고 있던 대다수의 PL들은 그야말로 미개했다.

2. 전기공학부의 PL : C/C++

전기공학부에서의 컴퓨터와 컴퓨터공학부는 상당히 비슷하긴 하지만, 전기공학부에서는 프로그램의 원리를 이해하고 이를 더 발전시키기 보다는 단순히 프로그램 개발용으로 PL을 사용하는 입장이라는 것이 큰 차이라고 볼 수 있다. 앞서 서론에서 말한 것처럼, 전기공학부의 거의 대부분의 (컴퓨터 관련)수업은 C++를 이용한다. 그중 매우 중요한 이유는 세계에서 가장 많이 사용되고 있는 언어가 여전히 C/C++라는 사실이다. 이것은 C의 역사를 보면 쉽게 이해할 수 있다.

C라는 언어는 원래 BCPL이라는 언어에서 시작되었다고 볼 수 있다. C가 나타나기 이전에, 1960년대에 MIT, Bell Lab, GE에서는 GE-645컴퓨터에서 돌리기 위한 'Multics' 프로젝트를 진행하였다. 그러나 그 결과물은 매우 느리며 매우 값비쌌기 때문에 중단되었다. 그러자 Ken Thompson은 그 대안을 찾기 위해 DEC PDP-7을 이용한 새로운 운영체제를 개발하게 되었으며, 그것이 바로 'Unix'이다. 그리고 Unix의 시스템 프로그래밍 언어로 BCPL을 기반으로 하여 새로운 언어인 'B'를 만들었고 이는 어느 정도 성공적으로 보였다.

그러나 PDP-11이 등장하면서 B의 여러 문제점들이 드러나게 되었다. Character-handling, floating-point 연산의 불가, pointer 사용에 있어서의 문제점들로 인하여 이를 수정한 'NB' 가 나타났고, 여기에 '&&', '||'등의 연산자의 도입과 표기법등의 수정을 통하여 우리가 사용하고 있는 'C'가 만들어지게 되었다.

이후, 1973년 Unix는 C로 다시 작성됨으로써 여러 플랫폼에서 작동할 수 있도록 됨으로써 Portability 문제를 해결하였다.

위 과정에서 가장 중요한 부분은 Unix와 C간의 관계이다. Unix는 C를 기반으로 작성되어 있기 때문에, 여러 플랫폼에서도 동작할 수 있는 OS로서 Unix가 성공하게 되자 C 역시 함께 발전하게 되었다. 1980년대 들어서는 C가 널리 퍼져 어느 OS, Machine Architecture에서도 C 컴파일러를 볼 수 있었으며 상업용 소프트웨어 개발용 언어 및 프로그래밍에 관심있는 일반인들도 사용할 수 있게 되었다.

이러한 연유로 '실용성을 (대단히)중요시 하는' 전기공학부에서는 모든 학부수업 뿐만 아니라 대학원 프로젝트에서도 대부분 C/C++를 사용한다. 물론 C/C++가 가지고 있는 많은 장점 때문에 C/C++를 사용하는 것이다. 우선 C/C++는 다른 PL들에 비해서 효율성이 매우 좋다. 아니, 좋을 수밖에 없다. 왜냐면 B가 OS를 개발하기 위한 목적으로 탄생했고, C는 B의

발전형이기 때문이다. 또한 C는 저수준의 프로그래밍이 가능하다. 그렇기 때문에 OS의 크기와 효율이 매우 중요한 임베디드 시스템의 경우는 C/C++를 사용할 수밖에 없다. 마지막으로, C/C++를 이용한 무수히 많은 라이브러리들이 존재하기 때문에 굳이 다른 언어를 이용하여 힘들게 코딩 할 필요가 없다.

그렇지만 이러한 장점들에도 불구하고 C++를 사용하면서 상당히 많은 불편함을 겪을 수밖에 없었다. 대표적인 것이 Type Checking이다. C++와 같은 1세대 언어인 Java만 해도 Type Checking이 대단히 엄격하다. (사실, Java와의 비교는 약간 무리가 있다. Java는 PL이라기 보다는 JVM을 이용하여 구동되는 Interpreter에 가깝기 때문이다. 그렇지만 JIT를 이용하면 PL로써 사용할 수도 있다.) 물론 Type Checking이 없기 때문에 Pointer를 이용하면 C++의 다양한 Type들을 다 똑같이 취급할 수 있다는 장점(?)도 있다. 그렇지만 분명 type checking의 부재는 여러 종류의 data를 모두 똑같이 취급하게 된다는 문제가 있다. 예를 들면, C/C++에서는 조건문을 사용할 때 true는 0을 제외한 모든 숫자, false는 0으로 사용할 수 있다. 이를 이용하면 코드의 길이는 짧아질지언정, 사람이 볼 때 무엇을 의도한 것인지 쉽게 알 수 없다.

뿐만 아니라, C의 최대 장점이지만 단점이 되기도 하는 Pointer의 경우는 상당히 골치 아프다. 오죽하면 Pointer만 완벽히 알면 C를 완벽히 할 수 있다는 말이 나왔을까. Pointer도 한 가지 종류가 아니라, type별로 pointer가 따로 있으며 pointer의 pointer라는 것도 존재한다. 전문가가 사용하면 pointer만큼 강력한 기능도 없지만, 당사자가 아니고서는 그만큼 복잡하고 헷갈리는 것도 없을 것이다.

3. 계산이란 무엇인가

잠시 복잡한 C/C++는 잊고, 프로그램에 대한 본질적인 질문을 해보자. 과연 우리가 어떤 프로그램을 만들었을 때, 그것이 올바르게, 우리가 원하는 대로 작동한다는 보장이 있는가? 또한 같은 프로그램이란 무엇일까?

이는 항상 코딩을 하면서 느끼는 점이기도 하다. 과연 이 코드대로 프로그램을 만들면, 내가 가지고 있는 생각대로 동작하는 것일까? a+b같이 간단한 프로그램의 경우에는 의심의 여지없이 '당연하다'라고 생각해 왔었지만, 프로그램이 조금만 복잡해져도 의심을 하지 않을 수 없었다. 그리고 그때마다 나의 해결책은 항상 '임의의 입력에 대해서 올바른가?'를 확인하는 작업이었다. 물론 쉽지 않았다.

이러한 고민들은 나뿐만 아니라 모든 사람이 가지고 있던 고민이었을 것이다. 그리고 이를 해결하기 위하여 연구하는 분야가 있었으니, 그것은 (적어도 나에게는)상당히 의외의

학문이었다. 바로 '논리학'이었던 것이다.

보통 사람들은 논리학이라 하는 학문은 당연히 인문계 과목이며 그중에서도 상당히 철학적인 내용을 다루는 것으로 알고 있다. 나도 논리학 수업을 들은 적이 있는데, 이 논리학이 컴퓨터가 계산하는 과정에 밀접히 관련이 있었다는 사실을 꿈에도 생각하지 못하였다. 그렇지만 '19세기 논리학, 21세기 계산학'이라는 글을 읽고 나자 논리적인 추론을 이용하면 우리는 두 프로그램이 같다는 것을 아직은 보일 수 없을지 몰라도 결국에는 알아낼 수 있을 것이라는 생각이 들었다.

4. 새로운 언어 : 논리학

예전에 '프로그래밍 방법론'이라는 수업을 들을 때의 일이다. 어느 날은 교수님이 미국 대학의 한 교수 분을 모셔서 특강을 하신 적이 있었다. 그 교수님 역시 프로그래밍 언어 관련 전공을 하시는 분이었다. 어쩌면 '전기공학부'인 내가 '프로그래밍 언어'라는 수업을 듣는 것은 그분의 강의가 워낙 감명 깊었기 때문일 것이다.

특강교수님은 오시자마자 우리들에게 프로그래밍 언어를 아는 대로 말해보라고 하셨다. 그러나 우리는 고작 C/C++, Java, Python정도(그나마 Python은 스타리그 맵 중에 Python이라는 맵이 있어서 알게 된)정도밖에 몰랐다. 그러자, 교수님은 프로그래밍 언어 교재에 첫 페이지에 있는 말보다 더 과격하게, '이 언어들은 쓰레기'라고 하셨다. 그리고 교수님께서 적으신 언어는 Lisp, ML, Smalltalk, Scheme등이 있었다.

그리고 위와 같은 언어가 안 좋은 이유는, '수학적으로 표현이 불가능하다'라는 것이었다. 사실 이때 나는 그 말뜻을 잘 이해하지 못하였다. 분명히 C/C++에서도 프로그램의 구동 과정은 항상 연산을 통해서 일어나고 있는데, 어째서 수학적 표현이 불가능 하다는 것일까. 그리고 그 답을 1년이 지나서야, '19th Century Logic and 21st Century Computing' 이라는 Article을 읽고 얻을 수 있었다.

논리학이란 학문의 탄생은 우리가 생각하는 것보다 훨씬 오래되어서, 고대 그리스의 아리스토텔레스 때로 거슬러 올라간다. 그리고 그때의 논리학이 거의 19세기 후반, Frege가 1879년 'Begrischrift'를 출판하여 현대 논리학이 시작되기 전까지 계속되어 왔다. Frege는 새로운 Logic기호와 공식들을 만드는 것으로 시작했다. 우리가 논리학에서 쉽게 볼 수 있는 '⊦'같은 기호를 만들었다. 그러나 그가 만든 논리학에서는 modus ponens(긍정논법)를 위해서는 몇 가지 axiom들이 필요했다.

그로부터 대략 반세기가 지난 1934년, Gentzen은 복잡한 axiom들이 필요 없는, 보다

자연스러운 논리학을 위해서 'natural deduction'을 만들어 내었다. 그리고 그의 System에서는 '가정'과 여러 추론규칙들을 이용한 덕분에 그 어떠한 axiom도 필요 없었다. 또한 '전체의 논리식은 결론논리식의 부분들로 간소화 될 수 있다'라는 Subformula property를 보였다.

또한 그와 거의 비슷한 시기인 1932년, Church는 새로운 논리식을 세우는 데 'Lambda Calculus'라는 것을 도입했다. 이 Lambda Calculus는 함수를 표현하는데 있어서 등식을 사용하는 대신, 'λx.t'와 같이 함수 역시 하나의 개체로 생각하는 표기법을 만들었다. 이 Lambda Calculus의 재밌는 점은 함수가 함수를 인자로 받을 수 있다는 점이다. 예를 들면 인자를 2개 이상 받는 경우, 함수가 또 다른 함수를 인자로 받음으로써 여러 개의 인자를 처리할 수 있었다. Church는 이후 각 인자들의 type을 결정하는 절차를 도입함으로써 'Typed Lambda Calculus'를 만들어 내었다. Lambda Calculus는 이후에 '함수는 1등 시민이다'라는 모토 아래 만들어진 Lisp, ML, Scheme, Haskell등의 PL들의 이론적 근거가 되었으며 심지어는 Java의 강력한 type checking에까지 영향을 미쳤다.

사실 Gentzen의 Natural deduction과 Church의 Lambda calculus는 본질적으로는 같다. Natural deduction에서의 추론과정은 Lambda calculus에서의 대입과 같다는 것이 1969년 Howard에 의해서 밝혀졌다. 그런데 우리가 흔히 알고 있듯이 Lambda calculus는 Turing machine과 동일하다. 즉, 우리가 사용하고 있는 프로그램들은 모두 증명과정과 같다는 것이다.

앞서 특강교수님이 말씀하신 언어들은 모두 Lambda calculus의 영향을 받아 만들어진 언어들이기 때문에 수학적일 수밖에 없다. 그렇다면 도대체 프로그램이 증명과정과 똑같다는 것일까? 그것은 아마도 이렇지 않을까 싶다.

프로그램은 항상 입력을 받는다. 그리고 그 입력을 적당한(우리가 원하는 대로) 처리를 하여 결과를 내 놓는다. 증명과정에서는 어떠한 전제를 바탕으로 이루어진다. 전제를 얻게 되면 이를 추론규칙들을 이용하여 새로운 정리를 만들어 낸다. 즉, 우리는 입력을 '전제'라고 볼 수 있다. 그렇지만 컴퓨터에서는 무조건 받아들이는 전제이므로, Axiom으로 봐도 무방할 것이다. 프로그램의 실행 결과는 출력이고, 이것은 증명과정에서의 결과인 '정리'와 같을 것이다. 마지막으로, 그 중간단계인 프로그램은 곧 추론규칙과 마찬가지이고 이것은 곧 증명과 같다. 그렇기 때문에 PL이 수학적으로 증명 가능하다고 볼 수 있는 것이라 생각한다.

Typed Lambda calculus는 앞서 말한 것처럼 Natural deduction과 같기 때문에 type checking을 하는 과정이 곧 프로그램의 구조를 Proof tree처럼 분석하는 과정과 같다. 그렇기 때문에 Lambda calculus를 기반으로 한 PL들은 Syntax error 뿐만 아니라 Semantic error역시 파악할 수 있다. 이것이 Lambda calculus의 강력한 장점이라 할 수 있겠다.

Lambda calculus는 앞서 말한 대로 Turing Machine과도 동일하기 때문에, Lambda calculus를 이용한 표현식은 모두 다 Turing Machine에서 실행이 가능하다. 이를 반대로 말하자면, Turing Machine에서 실행 가능한 모든 프로그램은 Lambda calculus를 이용하여 표현이 가능하다. 이것은 정말 멋진 성질이 아닐 수 없다. 우리는 Lambda calculus를 사용하면 이 세상에서 실행 가능한 프로그램을 한 개도 빼놓지 않고 만들 수 있다는 사실이기 때문이다.

5. 과거, 현재 그리고 미래의 PL

현재 사용되고 있는 C와 C++는 앞서 논의된 것처럼 본래 기계를 구동시키기 위해서 필요한 OS를 만든 언어이다. 즉, 철저히 기계 중심의 언어인 것이다. 그렇지만 Unix와의 연계성으로 인하여 대중적인 인기를 얻어서 현재까지 가장 많이 쓰이고 있지만 그 문제점은 적지 않다.

물론 C/C++도 자체적인 노력을 거듭하여 새로운 표준이 매번 나오고 있고, 그때마다 이전 표준의 단점들을 커버할 수 있는 새로운 feature들을 도입하고 있다. 올해 3월에도 C99를 대체할 새로운 C표준의 도안이 공개되었다.

하지만 근본적으로 기계를 위한 언어를 사람이 보다 쉽게 사용할 수 있도록 하는 것은 쉽지 않았다. 포인터를 도입하고, 본래 untyped language였던 B에서 사용자를 위하여 type을 도입하였지만 이는 무용지물이나 다름없다. BCPL의 string처리의 부재가 여전히 계속되어 C 특유의 string을 Character배열로 처리하는 방식이 생겨났다. 그리고 이것은 string처리를 위해서 대단히 복잡한 과정을 거치게 만들었다.

처음 C가 나왔을 때에는 여러 단점은 그다지 큰 문제가 되지 않았지만 현재 컴퓨터 분야는 엄청난 발전을 거듭하여, 더 이상 C의 문제점을 가만히 보고 있을 수만은 없게 되었다. 현재 인터넷을 1분만 찾아봐도 각종 프로그램의 버그들을 수백 가지는 찾을 수 있으며 모두 '프로그래머가 원하지 않던 결과'인 것이다.

현재는 많은 컴퓨터 공학자들의 노력으로 기존 1세대 언어가 알아낼 수 있는 문법적 에러뿐만 아니라 type checking을 통한 논리적 오류를 어느 정도 알아내는데 성공하였다. 바로 이것이 2세대 언어의 탄생이다. 2세대 언어는 이러한 특징 이외에도 논리적이기 때문에 사람의 입장에서 쉽게 이해할 수 있다는 장점도 가지고 있다. 즉, 이전에는 PL은 기계를 동작시키기 위해서 존재했던 언어였지만 이제는 입장이 바뀌어서 기계가 PL을 동작시키기 위해서 존재하게 된 것이다.

그러나 사람의 욕심은 끝이 없기 때문에 컴퓨터 공학자들은 앞으로도 더욱 발전한 PL

을 개발할 것이다. 대표적인 예가 실행하기 전에 미리 에러를 알아내는 기술이다. 문법적, 논리적 오류는 없을지라도 실행해보면 원치 않던 결과를 얻게 되는 경우가 있다. 대표적인 것이 무한 Loop이다. 아마 그 누구도 무한 Loop에 빠져서 아무것도 할 수 없는 프로그램을 원하진 않을 것이다. 그렇지만 무한 Loop에는 단지 조건이 잘못 되었을 뿐 어떠한 문법적, 논리적 오류도 없다.

이러한 오류를 잡아낼 수 있는 기술이 탄생할 수 있을까? 프로그래밍 언어 교수님께서는 첫 시간에 아마 2~30년 내로 구현이 될 것이라고 하셨던 기억이 난다. 1세대에서 2세대 언어로 갈 때는 새로운 논리체계의 힘을 빌어서 발전하였지만, 과연 2세대에서 차세대 언어로 갈 때에는 어떠한 힘을 빌어서 발전을 하게 될까? 그것은 아마 2~30년 뒤의 우리가 해결할 문제가 아닐까 싶다.

6. 마치며

에세이를 쓰면서 머리에 남아 있던 글들은 대부분 'C-history'와 '19세기의 논리학, 21세기의 계산학'이었다. 물론 나머지들을 읽지 않은 게 아니라, 그만큼 앞의 두 글이 감명 깊었다는 뜻이다. 특히 가장 머리에 남는 것은 Lambda calculus였다.

나는 전기공학부 학생이기 때문에 프로그래밍에 관해서 아는 것은 기껏해야 자료구조, 알고리즘 정도였다. 그렇지만 Lambda calculus를 보는 순간 나는 '프로그래밍도 결국은 수학적 과정이다'라는 것을 깨달을 수 있었다. 그리고 더 이상 코딩을 할 때 아무 생각 없이 '이렇게 저렇게 하면 요렇게 된다.'라는 기존의 지식만으로는 결국 동작은 할지 몰라도 치명적인 에러를 발생시킬 소지가 있는, 어설픈 프로그래머에 불과할 것이라는 생각을 하게 되었다.

마지막으로 위와 같은 5개의 좋은 글들과 한편의 동영상을 볼 기회가 주어진데 대하여 교수님께 감사의 말씀을 올리며 글을 마친다.