4190.310 Programming Language Assignment #3: Essay

프로그래밍 언어 육아 개론

서울대학교 컴퓨터공학부
2003-11914
곽원영

1. 서론: 미개하디 미개한 프로그래밍 언어

"*프로그래밍 언어는 미개하다.*" Programming Language 수업 첫 Chapter 첫 단원 제목에 적혀있는 한마디이다. "이게 무슨 소리야!?" 나는 이 한마디에 속으로 반발을 하지 않을 수가 없었다. 나뿐만이 아니었다. 주위 모든 이들이 이 한마디를 곧이 곧대로 받아들이리라고 생각하기엔 힘들었다. 나는 대학교에 들어와서 프로그래밍을 처음 해본, 어찌 보면 늦게 배운 케이스이다. 컴퓨터의 기초 라는 과목을 통해 첫 프로그래밍 언어, 모국어로 Java를 배우고 단순 계산을 하는 함수를 작성하는 과제에도 벅차했던 나였다. 그런 내가 컴퓨터공학부에 재학을 하면서 서로 문법적으로 비슷한 점이 많은 C와 Java를 중심으로 과제를 했다. 모르는 내용이 너무 많아 과제가 벅찼음을 부인할 수가 없다. 그리고 군문제로 휴학을 하고 회사를 다니게 되었다. 주로 C++을 사용한 프로그래밍을 하였는데 학교에서 했던 과제보다도 언어 사용의 난이도가 높았고 그 결과물의 Fancy함을 보았다. 그리고 회사동료들과의 대화나 경험을 통해 현재 구현이 가능한 프로그래밍의 세계에 대해 그 범위와 사례들을 직간접적으로 두루 경험하였다. 내가 경험하고 느낀 바는 이 세계는 너무나도 넓다는 것이었다. "내가 학교에서 처음 프로그래밍을 해보았고 학교에서 공부를 계속하고 회사에서 실용적인 부분을 많이 배워 계속 성장해 나아가면서 현재 어느 정도 수준의 실력을 갖고 있다. 그러나 내가 앞으로 경험해야 할, 혹은 경험할 수 있는 세계는 너무나 방대하구나." 나의 프로그래밍 세계에 대한 impression이었다. 비록 그 끝을 경험하기에 아직 많은 것을 알지 못해 부족한 점이 많지만 적어도 프로그래밍의 시작점을 알고 현재 기술의 한계점을 어렴풋이나마 느낄 수 있었기에 학교 과제를 통해서 느낀 나의 부족한 점, 회사 프로젝트를 통해 느낀 나의 부족한 점이 크게 느껴졌고 이는 현대 프로그래밍 세계가 크고 정립이 되어 있어 내가 앞으로 배워야 할 것이 많다는 느낌으로 다가왔다. 아니 가볍게 생

각만 해 보아도 프로그래밍 언어로 로켓을 발사하고 미사일을 조준하고 가깝게는 우리가 쓰는 운영체제 및 응용 프로그램도, 그리고 지금 내가 즐기는 게임조차도 프로그래밍 언어가 관여 되어 있지 않은가? 이 것이 미개하다니 말이나 되는 소리인가?

프로그래밍 언어는 미개했다. 그렇다. 내가 봐오고 느꼈던 것을 부정한다는 것은 아니다. 하지만 수업이 진행되면서 생각이 바뀌기 시작했다. 프로그래밍 분야는 기술적으로 많은 것을 이루었고 많은 것을 갖고 있지만 그 것을 표현하는 데는 서툰 점이 많다는 것을 깨닫기 시작했다. 이번 과제에 Reading에 포함되어있는 *계산이란 무엇인가에 대한 아이디어들*은 이러한 나의 생각을 뒷받침 해주었다. 이 글이 강조한 4가지 부분에 대해 의문을 가져보자.

"같은 프로그램이란 무엇인가 (semantic equivalence)"
현재 우리가 갖고 있는 프로그래밍 언어는 같다라는 정의를 표현하기에 편리한 언어인가. 내가 의도한 바를 컴퓨터에게 전달하는 Language의 역할을 제대로 수행하고 있는가. 언어의 종류 별로 전달이 가능한 범위가 좁지는 않은가?

"순서대로 계산한다는 것은 무엇인가 (sequentiality)"
정보의 흐름을 사람과 컴퓨터가 이해하기 쉽게 정의할 수 있는가? 프로그램이 내부에서 순차적으로 움직이는 모습을 사람이 쉽게 이해할 수 있는가?

"프로그램의 안과 밖은 무엇인가 (interaction)"
프로그램의 안과 밖의 통신은 잘 되는가? 프로그램과 프로그램 사이의 외부에서의 소통은 원활한가? 언어가 달라도 호환이 잘 되는가? 언어의 차이까지 가지 않아도 현재 내 프로그램은 저 프로그램에 호환이 되는가?

"옮겨다니는 계산이란 무엇인가 (mobility)"
프로그램의 메모리 이동을 프로그램 자신은 이해하고 있는가? 사람은 그 이동의 인지를 잘 할수 있는가?"

앞서 가진 의문에 대해 나의 생각은 부정적이다. 고급언어가 되었든 Assembly 같은 중간단계언어가 되었든 기계언어가 되었든 간에 사람이 컴퓨

터가 speak 하는 언어를 알아듣기에는 힘든 점이 많고 역으로 프로그램 또한 사람의 command에 대해서 헷갈려 한다. "*프로그래밍 언어는 미개하다* 가이 소리구나. 그렇네. 정말로 미개하네." 내가 가진 생각이다. 동시에 나를 스쳐간 생각은 현재가 미개하다면 발전가능성이 많다는 점이 아닌가 하는 당연한 생각이다. 사람과 컴퓨터와의 의사소통을, 또 컴퓨터와 컴퓨터 간의, 아니 프로세스와 프로세스간의 의사소통을 원활히 하려면 우리가 세계를 이해하려 역사를 배우듯이 현재 프로그래밍 언어의 역사와 그 형성 과정을 이해하는 것이 필수라는 점을 깨달았다. 앞으로 추릴 나머지 세 가지 글의 존재가치와 교수님의 의도를 비로서 이해한 것 같은 느낌이다.

2. 본론 1: "유명한 나의 친구 C언어"

그래서 나는 내가 가장 편해하는 C언어를 The Development of the C Language라는 글을 통해 알아보기로 하였다. C언어는 과연 최고의 언어인가? C언어는 과연 의사소통이 편한 언어인가?

1960년대 후반은 Bell Telephone Laboratories의 컴퓨터 시스템 리서치에 있어 불안정한 시기였다. MIT와 General Electric와 Bell Telephone Laboratories는 Multics project 라는 것을 진행하였다. 그러나 누가 보기에도 이 project는 많은 시간과 방대한 cost를 동반하지 않고는 성공하지 못한다는 것을 알 수 있었기에 Ken Thompson은 그 대안으로 편안한 computing environment를 창조하기를 원했다. 그러나 이 계획은 쉽지 않았다. Thompson은 하드웨어적인 문제등 여러가지 문제를 겪었다.

Unix 가 PDP-7 에서 작동하기 시작한지 얼마 되지 않아 Thompson은 이 Unix 시스템이 시스템 프로그래밍 언어가 필요함을 느꼈고 Martin Richard가 고안한 BCPL 을 바탕으로 새로운 언어를 탄생시키고 이를 B라고 불렀다. 그러나 B는 BCPL로부터 온character-handling mechanism에 대해 문제가 많았고 floating-point arithmetic에 대한 준비가 전혀 되어있지 않았으며 pointer 를 사용하는데 있어 문제가 많았다. 그리하여 Thompson은 이 문제점을 직시하고 B를 확장하였다. 문자 타입을 추가했고 부피가 작고 속도가 빠른 compiler의 개발도 같이 병행하였다. 이리해서 만들어진 언어를 NB (as in New B) 라 불렀다. 그리고 이 NB를 토대로 declaration 표현등의 여러가지 표현을 수정하여 추가한 것이 오늘 날 쓰이는 C의 모태라 할 수 있다.

C는 탄생 뒤 빠른 성장을 거듭했다. && 와 || operator의 발견을 효시

로 발전하기 시작한 C는 그 시대에 초점이 된 portability를 해결한다. Portability가 해결되지 않은 그 시대의 컴퓨터 환경은 사용자를 한 시스템에 묶어 종속시켜 많은 많은 이사 비용으로 사용자를 협박하여 이득을 보고 있었다. 그 상황에서 portability가 해결된 C는 가뭄의 단비와 같은 존재였다. 많은 이들은 C를 이용하여 Unix를 이용하기 시작했고 80년대에 널리 퍼지게 되었다.

C는 언어를 연구한 끝에 계획적으로 차근차근 탄생한 언어가 아니라 필요에 의해 BCPL을 토대로 탄생한 B에 살을 덧붙인 꼴이라고 할 수 있다. 그러한 연유로 C는 문법적으로도 여러 형태가 섞여있음을 알 수 있다. 그 비획일성이 여러가지 혼잡 및 에러를 동반한 것도 부인할 수 없을 것이다.

나는 C언어와 길지 않은 나의 프로그래밍 역사의 대부분의 시간을 함께 하였다. 첫만남은 상당히 껄끄럽고 의사소통이 쉽지 않았지만 시간이 지날수록 C언어와 대화하는 법을 익혀나갈 수 있었다. 그래서 생각했다. nML로 수업을 한다니. 편하고 친숙한 언어를 내버려두고 C언어와의 첫만남 때와 같은 불편함을 다시 감수해야 한다는 점에 약간의 두려움을 가졌다.

그러나 내가 생각한 만큼 C언어가 좋은 언어인가? nML로 과제를 하면서 문득 생각에 잠겼다. 나는 nML에 대해서 많이 몰랐다. 아니 백지였다. nML을 아는 친구에게 처음으로 배운 것은 nML의 중요한 요소 중 하나인 type checking이었다. 이것 하나만으로도 언어의 반을 깨달은 느낌이 들었다. 이를 이용한 나의 어눌한 nML 어를 가지고 컴퓨터를 이해시키려고, 과제를 해결하려고 많은 노력을 하였다. 수많은 에러를 만나면서 나의 어눌한 언어 실력은 점점 나아지고 있었다. 그러면서 나의 C 입문 시절이 떠올랐다. C를 이용할 때 많이 본 segmentation fault와 C++ 를 이용하면서 본 링킹 에러를 필두로 한 수많은 에러들. 요즘은 보는 횟수가 부쩍 줄고 있었다. 거기까지 생각이 미치니 깨달은 바가 있었다. C언어가 편하고 좋은 언어가 아니고 내가 C언어에 젖어서 편하게 느낄 뿐이라는 것을. 다시 말해 문제점과 에러가 적은 C언어가 아니라 많은 경험을 토대로 나는 본능적으로 그리고 또 반사적으로 에러를 피하는 언어를 구사하고 있었던 것이다. 컴퓨터가 불편해 하지 않을 자신만의 jargon을 습득하여 유용하게 쓰고 있던 것이다. 그래서 원하는 결과를 비교적 쉽게 얻을 수 있었고 이는 익숙함으로 이어졌다. 반면 nML이 익숙하지 않은 탓에 많은 에러를 만나면서 불편함을 겪었다. 그런데 편하고 불편함은 좋은 언어에 대한 기준이 되지 못한다. 태생부터 C가 갖고 있는 문제점에도 불구하고 Unix 의 배급을 통한 portability를 기반으로 wide-

spread 할 수 있었던 덕에 사용자들은 C를 많이 접했고 이는 곧 익숙함으로 이어졌다. 하지만 익숙한 것과 언어의 표현력은 다른 것이다. 언어의 표현력에 있는 한계는 익숙함으로 극복할 수 없는 문제이고 이 언어의 표현력이야 말로 바로 언어의 좋고 나쁨의 기준이 될 수 있음을 깨달았다.

3. 본론 2: 프로그래밍 언어의 근간인 수학

그러면 좋은 언어를 만들기 위해서 우리는 언어가 만들어진 역사를 알아봐야 한다고 생각한다. 컴퓨터의 언어는 수학으로부터 이어져왔다고 해도 과언이 아니고 수식적인 개념과 함수의 정의가 모두 수학으로부터 나왔다. *A Basis for a Mathematical Theory of Computation* 이라는 글은 computation이 어떻게 수학으로부터 이루어져왔는지를 설명하였다.

Computation이라는 것은 프로세스를 기계에게 전달하여 수행시키는 것을 뜻한다. 이 Computation을 수행하기 위해서 Mathematical science 분야는 Numerical analysis, the theory of computability as a branch of recursive function theory, 그리고 the theory of finite automata 를 연구하여 computation이 갖는 한계점을 극복하는데 힘을 보태고 있다.

Computation에 중점을 두고 있는 만큼 논리학적인 부분에 대한 연구가 활발히 이루어지고 있었고 Computable / Non-computable function 에 대한 정의와 그 properties에 대한 내용을 다루고 있었다. 특히 recursive function에 대해 많이 다뤄지고 있었다. 사실 recursion에 관한 것이 수학에서 다뤄지는 것은 그다지 놀랄 일이 아니다. 흔히 과목에서도 다루는 Fibonacci series나 factorial calculation 등이 recursion을 사용한 수학의 쉬운 예이고 수학에서 하나의 function이 자기 자신을 자신이 사용하여 compute 하는 일은 비일비재 하다.

언어가 컴퓨터에게 명령을 전달하여 컴퓨터가 수행하게 될 Computation을 수학을 통해 알아보는 과정을 거쳤다. 이는 논리학으로 이어져 언어 발달의 근간이 되었다.

4. 본론 3: 논리학에서 언어까지

논리학은 컴퓨터와 인간이 의사소통하는 언어의 바로 전단계라고 할 수 있다. 컴퓨터가 이해할 수 있는 기본은 True / False 이기 때문에 논리학의 대한 이

해가 반드시 필요했다. *19th Century Logic and 21st Century Computing*은 이러한 논리학에 대한 여러가지 정리를 잘 설명해주었다.

Gentzen's natural deduction 은 A->B 이다라는 논리를 3가지 공리로 나눠 설명을 한다. 우선 여기서 A는 premises 라 하고 B는 conclusion이라 한다. Premises와 conclusion을 유념하면 공리를 쉽게 이해할 수 있다.

A면 A다 (A->A) 라는 id 와 새로운 정보를 introduce 할때 사용되는 공리A->B->A, 그리고 이를 거꾸로 이용하는 Elimination 공리가 Gentzen's natural deduction에 해당된다.

Church's lambda calculus 는 lambda term 을 이용하여 기계로 compute할 수 있는 모든 프로세스를 표현할 수 있다는 내용이다. 개인적으로 이 학문이 컴퓨터에 기여한 바는 크다고 생각한다. 왜냐하면 Church's lambda calculus의 특징은 function이 function을 argument로 취할 수 있고 또한 function 자체를 결과값으로 리턴 할 수 있다는 점이 있기 때문이다. 이는 value-based 프로그래밍을 뒤집은 획기적인 발상이라고 생각한다.

또 Typed lambda calculus가 소개 되었다. Typed lambda calculus은 앞선 lambda calculus가 약간 확장된 내용으로 term이 type를 갖기 시작한 점이 새롭다. 그리고 type를 assume 하고 확인하는 과정은 type checking 을 떠올리게 하였다.

마지막으로 The Curry-Howard correspondence 에 대한 내용이 소개되었다. Church's typed lambda calculus는 Gentzen's natural deduction은 상호관계가 있다는 내용이다. Type derivation이 주어졌을 때 쉽게 그에 해당하는 증명을 찾을 수 있고 또 역으로 증명을 가지고 해당하는 type derivation에 다다를수 있다는 내용이다.

Lambda calculus가 프로그래밍에 끼친 영향은 지대하다. Lambda notation 자체가 프로그래밍의 기초가 되었다는 점을 제외하고도 lambda term을 가지고 기계가 표현 가능한 모든 프로세스를 나타낼 수 있다는 점은 상당히 impressive 하다. 역으로 말하면 lambda term으로 표현이 된 프로세스는 프로그래밍을 통해 구현이 가능하다는 결론에 도출할 수 있다. 그리고 lambda term으로 표현이 불가능한 halting problem도 machine으로는 표현될 수 없음이 증명될 수 있는 것 같다.

5. 본문 4: 논리학과 프로그래밍 언어?

앞선 두 글에는 굉장히 많은 수식과 논리식이 있었다. 문뜩 드는 생각이 있었다. 내가 방금 전에 사용한 nML 과 형태가 비슷하다!?

논리학에서는 자연 현상의 진행 과정을 사람이 알아 볼 수 있게 기호와 문자로 나타내었다. 이는 무언가와 비슷하다. 바로 사람이 컴퓨터가 사람의 의도를 이해할 수 있게 일정한 규칙을 가지고 설명을 하는 것과 너무나도 흡사하다고 할 수 있다. 이런 논리적 expression 은 특징이 있다. 사람이 알아보기 쉽다는 점이다. 사람이 알아보기 쉽다는 점은 굉장히 큰 장점이다. 내가 아닌 다른 사람과의 의사 소통이 쉬워진다는데 의의가 있는데 만약 대화 소통 상대가 사람이 아닌 컴퓨터이면 어떠할까?

앞서 nML이 논리학식 과 비슷하다고 언급을 하였다. 실로 그러하다 평소 수학적 식에 익숙해져 있는 사람에게 있어 논리식은 기호의 정의만 알게 되면 쉽게 이해가 가능한 표현이다. 이러한 표현이 그대로 컴퓨터의 언어로 사용되게 되면 상당한 파급효과를 가져온다. 무엇보다도 내가 작성하기가 쉽다. 컴퓨터가 알아들을 수 있게 남의 나라 언어로 돌려 말하지 않더라도 우리가 사용하는 사람의 언어(와 비슷한 형태로) 컴퓨터에게 원하는 바를 손쉽게 전달할 수 있는 장점이 있다. 또한 사람끼리의 co-work가 쉬워진다. 프로그래밍에 있어 co-work가 힘든 점은 서로 간의 언어 사용 convention이 조금씩 다르기도 할뿐더러 사용 되는 언어를 마치 외국어 쓰듯이 작성하는 사람은 자신의 생각을 컴퓨터 언어로 번역을 해서 작성을 하고 읽는 사람은 컴퓨터 언어를 다시 번역하여 이해를 해야 한다. 하지만 직관적으로 이해가 가능한 언어는 그러할 필요가 전혀 없다. nML은 내가 컴퓨터에 이야기를 하고 있는 듯이 사용이 가능하지만 C같은 경우 내가 이야기를 하기 전에 알아야하는 규칙 몇가지가 있었다. 내가 사용할 변수나 함수는 '선언'이라는 것을 해야하며 먼저 선언 되지 않은 함수나 변수는 실제 사용하지 못한다는 규칙이 있었다. 이러한 실제 우리가 쓰는 언어와 다른 규칙은 아무래도 컴퓨터에게 얘기하는데 초반에는 장벽이 될 수 밖에 없다. 실제로 C 스타일 언어에 젖어 있던 나는 nML로의 과제 도중에 느낀 점이 있었다. 내가 만약 이 것을 C라는 외국어로 컴퓨터에게 설명을 하려면 컴퓨터에게 부연설명을 해야했다. "이것은 이거고 저것은 저거고 이 것이 이러니까 얘는 이렇게 되는거야" 하지만 nML이라는 언어로 과제할 때는 그럴 필요가 없었다. nML 컴파일러가 직관적으로 표현한 나의 언어를 이해하여 바로바로 수행해주었다. 이러한 경우의 예로 내가 가장 놀랐던 것은 pattern matching 이다. 내가 어느 정도의 힌트를 주면 컴

파일러는 바로 내 의도대로 case를 나누어 주었다. 만약 내가 이 것을 C로 구현했다 하면 나는 파싱을 해서 상황별로 나누거나 if-else 절로 모든 것을 대신 해야 했을 지도 모른다. 그렇다고 C언어가 나쁘다는 것은 아니다. Unix의 Portability를 이루어주었고 무엇보다 현대 세계에서 가장 많이 쓰이고 있고 또 가장 많이 쓰이고 있기 때문에 support되는 라이브러리가 많다. 이는 nML 과는 비교가 되지 않는 장점이라 할 수 있다. 하지만 nML과 같은 표현하기 쉬운 언어가 저러한 상황의 혜택을 받았었으면 나의 과제, 나의 프로젝트는 조금도 쉬운 방법으로 표현이 가능했고 표현이 쉬워져서 더 advanced한 내용을 쉽게 접할 수 있지 않았을까 라는 아쉬움도 조금 든다.

6. 결론: 프로그래밍 언어의 2차 성징

이렇듯 프로그래밍 언어는 아직 사람과 소통하기에는 모자란 점이 많다. C와 같은 언어는 라이브러리 기반과 support가 장점이지만 사용자가 컴퓨터에게 원하는 것을 직관적으로 전달할 수 있는 좋은 언어는 아니라고 할 수 있다. 반면 nML 과 같은 언어는 직관적으로 쓰이기에 너무나도 편한 언어지만 널리 쓰이지 않는 탓에 같은 내용의 프로그래밍을 하려면 C가 어느 정도 구현이 된 라이브러리 선에서부터 시작을 할 때 바닥부터 프로그래밍을 시작 해야 하는 단점이 있다. 나는 Programming Language를 듣고 있는 학생이다. 단순히 오늘 나오는 과제 내일 나오는 과제를 숙제하는 기계마냥 생각 없이 할 것이 아니라 그 과제에서 교수님이 우리에게 전달하고 싶었던 표현은 무엇이었나를 곰곰히 생각해 보고 싶다. 내가 군휴학을 하기 전에 진행된 컴씨에서 교수-학생 면담도중 질문시간이 있었다. 어느 학생이 질문했다. "교수님, 하나의 언어를 깊게 파는 것이 좋나요? 아니면 여러가지 언어에 익숙해지는 것이 좋나요?" 그 때 마이크를 잡으신 것은 이광근 교수님이 셨다. "여러가지 언어를 해보시는게 좋습니다." 장난 스럽게 하신 한마디로 좌중은 웃음바다가 되었다. 나도 여러가지 언어를 해본다는게 말만큼 쉬운 것이 아니라는 것을 알기 때문에 그 자리에서 웃고 지나갔다. 하지만 몇 년이 지난 지금 생각해보건데 그 말 뜻은 한 가지 언어를 파서 그 언어로 모든 것을 구현할 수 있는 기계가 되지 말고 여러 가지 언어를 써보아서 언어의 장단점을 파악하고 여러 언어에 대한 사용 경험을 쌓아 앞으로 사람들이 그 장점에 매혹되어 "한 가지 언어" 로 선택해서 팔 수 있을 만큼 좋은 언어를 만들어라 혹은 그 만큼의 연구를 진행할 수 있게 되어라. 라는 말씀이셨던 것 같다.

이렇게 많은 논란 거리를 제공할 만큼 우리가 현재 들고 있는 프로그래밍 언

어는 미숙한 점이 많다. 그리고 이 미숙한 점을 해결할 사람들은 지금 Programming Language를 듣고 있는, 나를 비롯한 수강생들이 되어야 할 것 같다. 미개한 프로그래밍 언어를 사용하는 사람이 아니라 미개한 프로그래밍 언어를 키울 수 있는 사람이 되고 싶다는 생각을 하게 되었다.

7. 끝으로..

사실 나는 비슷한 과제를 4년전에 한 경험이 있다. 공학수학2 에서 들었던 같은 과제를 한 경험이 있는데 그 때는 읽어야 할 문서도 양이 적었고 써야 할 분량도 많지 않았다. 그래서 그런지 4년이 지난 지금 그 문서를 나 자신이 읽어보았을 때 내가 그 과제에 대해 깊히 생각해보지 않았었다라는 것이 느껴져 왔다. 그러나 지금은 다르다. 고되다고 소문난 Programming Language course를 단지 학점만을 위해서 혹은 졸업만을 위해서 수강한다는 것보다는 더 한 무언가가 목표로 자리를 잡은 느낌이다. 그만큼 이번 과제는 프로그래밍 스킬이나 컴퓨터 지식 하나를 더 배우는 것보다 더 의미가 있었던 것 같다.