< 2008Fall. Programming Languages - 논술 에세이 >

컴퓨터와 인간, 그 사이 프로그래밍 언어의 위치
- 절대적으로 좋은 것이 있을까?

이 현 민
서울대학교 컴퓨터공학부

초 록

프로그래밍 언어는 사람의 생각을 컴퓨터에 전달해주기 위해 사용된다. 언어는 사람의 생각, 창조적인 폭 등 여러 가지를 제한할 수 있기 때문에, 좋고 수준 높은 생각을 위해서는 좋은 언어가 필요한 것은 당연하다. 그렇다면 좋은 언어란 무엇일까? 주어진 4개의 논문을 읽고 프로그래밍 언어에 대한 내 견해를 펼쳐보았다.

목 차

1. 서론

내가 프로그래밍 언어에 대해 아는 것은 학부 PL 수업을 1달 들은 어린아이 수준에 불과하다. 하지만, 프로그래밍 언어에 관심을 가지게 된 것은 2학년 말부터였다. 수학적이고 논리적이라는 것이 마음에 들었다. 또한 알고리즘 분야처럼 너무 이론적으로만 치우치지 않고, 컴퓨터 그 실제 상황을 같이 고려해주어야 한다는 점이 매력적이였기 때문이다. 그 후로 ROPAS 홈페이지나 외국의 유명한 교수님들 홈페이지를 들여다보며 설레는 마음을 갖기도 했다.

†

좋은(?)[1] 프로그래밍 언어를 개발하고, 좋은 프로그래밍 언어를 사용하는 것은 말 못할 정도로 중요하다고 생각한다. 우리는 한국어를 쓰기 때문에, 한국어처럼 생각한다. 미국은 영어를 쓰기 때문에, 영어처럼 생각한다. 어차피 그 언어로 표현하고 의사소통해야하기 때문에, 그 언어 틀 안에서 생각하게 되는 것이다. 예를 들어, 언어에 "blue"와 "yellow"밖에 없다면 그 언어를 사용하는 사람들은 평생 색깔을 2가지로만 분류하며 살아갈 것이다.

컴퓨터에서는 어떤가. 같은 문제를 놓고도 ML로 짜야한다면 C로 짜야할 때와는 다른 각도로 문제를 바라봐야 한다. 평소에 C를 많이 접하고 C처럼 생각하는 사람에게는 어려워 보이는 문제가 ML을 많이 접하고 ML처럼 생각하는 사람에게는 쉬워 보일 수 있다. 또, 그 머리 속에 잡히는 사고체계들은 서로 다른 방향의 창조력을 발휘해낼 것이다.

문법은 많은 것이 좋을까, 적은 것이 좋을까. 장단점이 있겠지만, 생각하는 측면에 있어서는 문법이 많을수록 편하다(문법을 다 알고 있다는 가정 하에). 함수라는 것도 프로그램에서 일종의 내가 원하는 문법을 만드는 것이라 할 수 있다. "나는 이러이러한 기능을 하는 문법이 필요하기 때문에 옆에 미리 만들어 놔야지."

While()문의 이름이 While인 것처럼, 쓰는 법과 단어 하나하나도 중요하다. 세종대왕에 의해 한글이 개발되기 전까지 우리 한국 민족은 말은 국어로 하면서 쓰기는 한문으로 하는, 구어와 문어의 불일치에서 오는 2중 체제의 기형적 상태를 오랫동안 경험하고, 그로 말미암아 한문적 요소가 대량으로 국어 속에 침투하는 결과를 낳았다고 한다.[2] 언어로 의사 소통이 되더라도, 그 언어가 표현하는 바를 그대로 적을 수 있는 수단이 없자, 생긴 일이다.

만약 우리나라 말이 국제적인 언어였다면, "하는동안()"과 같은 함수를 사용하게 됐을 것이고, 그러면 프로그래밍 언어가 더욱 친숙하게 다가왔을 것이다. 미국 사람들은 마치 "~하는 동안 'ㄱ'에 3을 더해서, 마지막에 출력('ㄱ')을 해야지"처럼 정말 컴퓨터와 대화하는 느낌으로 프로그래밍을 하고 있을런지 모른다.

그럼 어떤 것이 좋은 프로그래밍 언어일까.

기계 중심의 언어 vs 값 중심의 언어

기계 중심의 언어는 값이 변한다. 말 그대로 기계에 명령하면서, 값을 조작해나가는 것이다. 함수는 값이 아니며, 값들을 입력받아 이런저런 일을 수행하는 로직일 뿐이다. 함수를 인자로 넘겨주지 못하는 것이 아쉽다. 하지만, 값을 바꿔나가며 쓰기 때문에 효율이 좋다. 값 중심의 프로그램은 수학적으로 탄탄한 모델이다. 하지만 항상 새로운 값이 만들어져야 하므로, 현대의 디지털 컴퓨터에서는 최상의 선택이 될 수 없다. 사용 의도와 상황에 따라 어느 쪽도 더 좋은 프로그래밍 언어가 될 수 있다.

문법이 많은 언어 vs 문법이 적은 언어

문법이 많은 언어는 생각하기가 쉽다. 작성하고 기술하기도 쉽고, 다른 사람이 알아보기도 쉽다. 좀 더 인간에 가까운 언어이다. 하지만, 이렇게만 생각하다 보면 본질을 놓치게 될 수 있다. 문법이 적은 언어를 사용하게 되면, 본질적인 바닥에서 생각하게 되면서, 사고의 폭이 넓어질 수 있다. 내 생각에 훈련은 문법이 적은 언어로, 실전에서는 문법이 많은 언어를 사용하는 것이 옳은 것 같

1) 좋은....
2) 두산백과사전 Encyber.com 인용.

다.

타입이 까다로운 언어 vs 타입이 유연한 언어

타입이 까다로우면, 프로그램을 작성할 때 하나하나가 신중해야한다. 값이 어디서 어디로 흘러드는지 다 파악하고 타입을 정확히 맞추어 주어야 한다. 그만큼 안전해지고 에러나는 일이 적을 것이다. 타입이 유연한 언어는 프로그램을 빠르게 작성할 수 있고, 쉽게 생각할 수 있다. 마치 우리가 일상생활에서 사용하는 언어와 같다. 하지만, 복잡해지다보면 어디서 에러가 터질지는 알 수 없는 일이다.

- 절대적으로 좋은 언어는 존재할 수 있을까?

언어는 환경에 영향을 많이 받는다. 각 나라가 각 언어를 쓰는 데에는 그런 문화적 바탕과 역사가 깔려 있다. 우리나라 안에서도 전라도 말은 농사가 잘 되서인지 구수하고, 경사도 말은 사납다. 프로그래밍 언어는 컴퓨터와 의사소통하기 위해 만들어졌다. 그런데 현재 우리의 컴퓨터는 전기로 메로리를 조작하는 방식이다 보니, 이런 환경에 맞추어 현재와 같은 C, JAVA같은 것이 주도하고 있다. 그러나 만약 양자컴퓨터, 바이오컴퓨터와 같은 것이 제대로 등장하게 된다면, 그 때도 C와 JAVA가 좋은 언어일지는 장담할 수 없는 일이다.

그럼, 양자컴퓨터, 바이오컴퓨터와 같은 것의 개발은 어떻게 이루어져야할까. 이광근 교수님께서 말씀하셨듯이 기계가 언어를 돌리기 위한 수단이지, 언어가 기계를 돌리기 위한 수단은 아니다. 그렇다고 해서, 먼저 언어를 고안해놓고 그에 맞는 컴퓨팅 방법을 찾는 것도 쉽지 않은 일이다. 협력이 필요한 때인 것 같다. 바이오컴퓨팅을 연구하는 사람들이 만약 이것이 성공하게 된다면, 이러이러한 방식으로 값이나 함수를 관리하고 저장하게 될 것이라고 말해주면, 언어를 연구하는 사람들은 그에 걸맞는 언어를 개발하면 된다.

2. "The Development of the C Language"

이 논문은 현재 세계에서 가장 많이 쓰이고 있는 언어인 C언어의 개발 역사를 말해주는 논문이다. 프로그래밍을 배우게 되면 대부분이 처음 접하게 되는 언어는 C이다. 컴퓨터를 전공하지 않는 사람들도 프로그래밍에 관심이 있으면 모두 C언어를 배운다. 과연, 이러한 언어가 어떻게 탄생한 것일까?

C는 BCPL, B를 따라 발전된 언어이며, Unix 운영체제의 영향을 많이 받았다. 1960년대 후반에 MIT와 General Electric와 Bell Labs가 모여 Multics project를 시작하게 되었다. BCPL은 1960대 중반에 Martin Richards에 의해 디자인되었다.

BCPL, B, C의 공통점은 시스템 프로그래밍에 중점을 맞춘 언어란 것이다. 기계를 앞에 놔두고 기계를 잘 사용하기 위한 언어를 만든 것이다. 전통적인 컴퓨터의 복잡한 데이터 타입들과 연산을 바탕으로, OS와의 입출력 교류를 위한 라이브러리에 의지한다. 또한 세 언어는 모두 프로그램이 global declaration과 function declaration의 sequence로 구성된다. 함수를 일반적인 declaration과 다르게 보는 것이다. BCPL은 nested procedure를 허용하지만, B와 C는 이를 허용하지 않는다.

BCPL을 발전시킨 B에는 3가지 문제가 있었다. 첫 번째로는 캐릭터 핸들링 메카니즘, 두 번째로는 약한 Floating point 연산, 세 번째로는 포인터 연산의 오버헤드 문제였다. 그래서 저자는 B언어에 character type을 추가하고, 컴파일러를 다시 쓰는 작업을 통해 확장하는 작업을 시작하게 되는데, 그것이 바로 NB이다. 그리고 이를 더 다듬어서 C가 만들어지게 된다.

1973년에 현대 C언어의 필수적인 부분이 완성되게 된다. 그리고 1980년대까지는 조금씩 성장하게 된다. 그 중 특히 1977년대에 portability와 type safety를 큰 관심을 가지고 고려하게 된다.

그리고 C언어는 Standardization을 위해 다음과 같은 목표를 세우게 된다.

"to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environment."

C는 본질적으로 typeless language에서 출발하였다는 것이 가장 큰 약점이다. 나는 사실 예전부터 C, C++을 사용할 때, typeless의 성질이 너무 좋았었다. char로 작업하다가 'A'에 1을 더해서 'B'을 만든다던지 하는 것들이 너무 편하고 그럴 듯 했기 때문이다. 하지만, PL수업을 듣고 이것저것 알게 되면서 이것이 꼭 장점만은 아니라는 것을 느끼게 되었다.

그래도, 'A'에 1을 더해서 'B'을 만드는 기능은 쓰고 싶다. 모든 타입을 유지하면서 직관적으로 여러 가지를 편하게 할 수 있는 언어는 어디 없을까? 나중에 기회가 된다면 꼭 관심있게 들여다보고 싶은 분야이다.

3. "A Basis For a Mathematical Theory of Computation"

Computation 관련 수학적 연구에는 3가지 관점이 있다고 한다. 첫째로 수치해석, 둘째로 재귀함수이론, 셋째로 유한 오토마타 이론이다. 그래서 이 논문에서는 computation의 수학적 이론 토대 만들기를 시도한다. 그래서 기대되는 실용적인 결과는 다음과 같다고 한다.

1. 다용도 프로그래밍 언어를 개발한다.
2. computation 과정에서의 equvalence를 정의한다.
3. 알고리즘을 symbolic expression을 이용하여 표현한다.
4. 컴퓨터를 computation처럼 표현한다.
5. computation의 많은 이론을 제공한다.

또 논문은 크게 descriptitive formalism과 Mathematical result to prove the equivalenc로 나누어 진다.

먼저 Class of functions computable in terms of F라는 것을 정의한다.

(p1→e1, p2→e2, ... , pn→en)

과 같은 표현을 사용하는데, 이는 condition 구문이다. 이를 이용해 삼각형 모양의 함수, factorial 등을 표현할 수 있다. 이 정의를 조금씩 늘려서, set집합 등 여러 가지를 만들 수 있다.

다음은 정수의 재귀함수이다. 재귀함수가 도입되면, 표현할 수 있는 집합이 훨씬 많아진다. 재귀

는 전산학에서 참으로 중요한 주제인 것 같다. 같은 것을 다른 변수로 계속 반복하는 것, 그리고 그 리턴 값을 이용하고 연결해서 큰 모델을 만든다는 것.

Computable Functionals는 함수를 인자로 받는 함수이다. 함수를 특별하게 취급하지 않음으로써 여러 가지를 자연스럽게 정의하는 것이 가능해진다. 이광근 교수님 수업에서도 강조된 것처럼 실제 수학에서는 함수를 함수의 인자로 사용하는 일이 잦다. 그런데, 현재 C, C++같은 언어는 함수를 인자로 사용하지 못한다.

C, C++만 사용할 때는 이것이 불편한 것인지 전혀 눈치채지 못했었는데, 막상 nML같은 언어를 사용하고 보니, 이게 참 편하구나 하는 생각이 들었다. 역시 이것도 서론에서 말했던, 어떤 언어를 사용하느냐에 따라 생각하는 법이 달라진다, 와 일맥상통한다.

함수 중에는 Non-computable Function과 Non-computable Functional이 있다. 이는 계산 불가능한 함수들의 집합을 말한다. 이는 튜링머신 개념으로 생각해볼 수 있다. 또한 Ambiguous Function도 존재한다.

재귀의 성질을 이용하면, Recursion Induction을 사용할 수 있다. 두 함수가 같은 기능을 한다는 것을 증명하는 한 가지 방법이다. 이를 통해 증명되는 모습을 실제로 보니, 증명이 곧 프로그래밍이라는 다음 논문의 제목이 떠올랐다.

<Relation to Other Formalisms>는 단원에서는 3". Proof procedures and proof checking procedures"를 가장 재미있게 읽었다. Proof를 생성한 뒤에 이를 다시 컴퓨터에게 체크하라고 하는 것은 중요하다고 한다.

check[statement;proof]

이가 어느 정도까지 연구되었는지 모르겠지만, 만약 이것이 어느 정도 좋은 정확도로 작동한다면, 굉장히 강력한 툴이 될 것임을 느꼈다.

계산 이론의 기초에는 여러 가지의 탄탄한 수학모델들이 자리 잡고 있었다. 프로그래밍 언어는 우리 현실의 언어와 달리 정확성과 엄밀성이 중요하다. 가장 낮은 기초적인 단계부터 논리의 흐뜨러짐 없이 쌓아 올려서 언어를 만든다면, 그 언어만큼 완벽한 언어는 없을 것이다. 하지만, 그런 단계에서 ambiguous 등 다른 변수들이 발생하기 때문에 이에 맞는 처리가 필요할 것이다.

4. Proofs are Programs: 19th Century Logic and 21st Century Computing

Proofs are Programs. 증명이 곧 프로그램이다.
수업 시간에 계속 들은 말이기도 했지만, 이 논문을 읽고 나니 더 와 닿는 느낌이 들었다.

(1) Gratzen's natural deduction

Gratzen이 1934년에 발표한 논문이라고 한다. 이는 한마디로 연역법에 의한 추론이다. modus ponens라고 하는 룰을 바탕으로 해서, 여러 가지 기본 연역 법칙들을 만들고, 이를 사용해서 증명하는 것이다.

(2) Church's lambda calculus

Church는 교환에 의한 계산을 모두 제거했다. 보통, 수학자들은 함수를 등식으로 정의했다. 그러나 Church가 새로운 방식을 제시한 것이다. "f(x)=t"는 (lambda)x.t 와 같은 형태로 쓰여진다. lambda calculus를 사용하면, 함수를 들고 다닐 수 있다. 함수를 들고 다니며, 다른 함수의 인자로도 사용하고 함수를 리턴할 수도 있다.

이 방식을 사용하면 compactable한 모든 것을 표현할 수 있다. 그리고 이런 말이 나온다.

> "Just as the physical universe appears to be built from a small number of fundamental particles and forces, the computing universe can be built from just three term forms and one reduction rule."

3개의 term form과 reduction rule만 있으면, 컴퓨팅의 모든 것을 할 수 있다는 말이다.

(3) Typed lambda calculus

Church는 1940년에 lambda calculus의 typed version을 발표한다. 대부분의 컴퓨터과학자들은 halting problem이 풀 수 없는 문제라고 말하지만, typed lambda calculus가 halting problem을 풀 수 있는 충분한 언어를 제공한다고 한다. 또 타입은 프로그래밍 언어를 구성하는 가장 강력한 원리 중에 하나라고 증명되었다고 한다.

Typed lambda calculus를 이용하면, 타입을 유추할 수 있고, 타입을 안전하게 고정할 수 있다. 이는 C와 같은 typeless language에 필요한 것이며, 뭔가 응용력 있는 활용이 필요할 것이다.

natural deduction과 lambda calculus에는 1:1 대응이 존재한다. 비슷한 시기에 발표되었던 이 두 논문이 같은 개념을 다루었다는 것이 주목할 만하다. Christopher Strachey는 다음과 같은 motto를 제시했다. 'function as first-class citizens'. 함수를 1등 고객으로 모시자는 말이다.

믿음은 컴퓨팅에서 필수적인 요소이다. 안전하고 믿음직한 컴퓨팅을 위한 언어를 개발하고 연구하는 것이 필요한 때이다.

5. 계산이란 무엇인가에 대한 아이디어들

프로그램의 실행을 이해하는 기본이 되고 있는 개념들은 크게 두 줄기라고 한다. 하나는 수리논리의 이론적인 내용들, 하나는 디지털 컴퓨터. 수리 논리의 이론적인 내용들로부터 온 내용들은 프로그램의 뿌리이자 기초이다. 즉, 약간 개조될 수는 있어도 변하지 않는다는 것이다. 하지만, 디지털 컴퓨터는 수단으로 이해할 수 있다. 다음 세대에 이 튼튼한 수리 논리 모델을 어떻게 돌릴 수 있을지는 계속 연구되어지고 있다.

- **같은 프로그램이란 무엇인가 (semantic equivalence)**

Robin Milner에 따르면, 두 프로그램이 같은 것에 대한 정의는 아직 나오고 있지 않다고 한다. 정의 내리기는 어려운 일이다. 하지만, 가능한 모든 입력에 대해 모두 같은 출력을 낸다면 같은 프로그램이라고 할 수 있지 않을까? 물론 가능한 모든 입력이 무한인 경우가 많을테니, 이는 귀납법등을 이용해서 증명되어져야 할 것이다. 그냥 같다고 판단하는 것과 엄밀한 정의는 다르니 쉽게 논할 문제가 아니긴 하다.

- **순서대로 계산한다는 것은 무엇인가 (sequentiality)**

그렇다, 정보의 흐름. 실제로 세상은 병렬적으로 돌아가고 있다. 하지만, 현재 프로그래밍 기술은 순차적이다. 병렬 프로그래밍이 연구되어지고 있긴 하지만 성과물은 많이 아쉬운 상황이다. 병렬 프로그래밍은 속도 효율, 자료 활용 등등 측면에서 정말 많은 변화들을 이끌고 올 것이다. 그에 맞는 자료구조, 알고리즘 등의 연구가 아직은 많이 필요할 것이다.

- **프로그램의 안과 밖은 무엇인가 (interaction)**

현재의 프로그래밍 시스템은 내부에 고정되어 입출력에 맞게만 작용한다고 볼 수 있다. 하지만, 실제 상호 작용을 고려하며, 그에 맞게 프로그램이 반응하며 환경을 바꾸어 나갈 수 있다면, 훨씬 고차원의 프로그래밍이 가능해질 것이다.

- **옮겨다니는 계산이란 무엇인가 (mobility)**

파이 계산법, 울타리 계산법 등 새로운 계산법들이 개발되어 간다고 한다. 자세히는 모르지만, 연결과 관계 중심의 계산법이라면, 수준 높은 프로그래밍 언어가 탄생할 것임을 기대해본다.

프로그래밍은 증명이며, 계산이다. 계산을 어떻게 할 것인가. 현 시대에 있어 계산이란 무엇인가? 이 글을 읽고 나서, 현실에 안주하지 않고 새로운 영역에 창조성을 더해 생각해 볼 필요가 있다는 것을 느꼈다.

6. 결론

프로그래밍 언어는 튜링이 고안한 수리 모델의 언어 구현체이다. 현재의 컴퓨터는 이 계산을 빠른 속도로 수행할 수 있게 하기 위해 전기적으로 구현된 것이다. 이런 컴퓨팅의 초기 목적은 빠르게.. 보다는 '자동으로'가 목적이었을 것이다. 하지만, 점점 발전되면서 여러 가지 부가적인 요소들의 향상이 필요해진 것 같다.

현재 나와 있는 프로그래밍 언어는 많다. 그리고 상황과 목적에 맞게 이용되어지고 있다. 하지만, 위의 논문들에서 살펴보고 지적했듯이 아직 연구되어질 것이 많다. 그럼 과연 우리는 어느 방향으로 나아가야 하는 것일까.

단순히 수학적인 모델에 치중하거나, 기계적인 모델에 치중해서는 안 될 것이다. 프로그래밍 언어는 중간에서 매개체 역할을 한다. 사람의 생각을 표현해주고, 이를 컴퓨터에게 전달하는 역할을 하는 것이다. 수학자, 논리학자들이 만들어낸 모델을 구현하되, 아키텍쳐나 컴파일러를 하는 사람

들과도 많은 대화가 필요하다.

서론에서 비교한 모델들에서 어느 한쪽으로 치우치는 것은 좋지 않다. 여러 가지 특색있는 프로그래밍 언어 모델들을 구현하고 상황에 따라 맞추어 사용하는 것이 바람직할 것이다. 또, 그러기 위해서는 프로그래밍 언어 사이의 원활한 소통 관계도 필요하다. A 언어와 B 언어가 쉽게 결합해서 돌아갈 수 있는 시스템 말이다.

위의 4개의 논문들을 읽으면서 프로그래밍 언어의 역사와 배경과 앞으로 나아갈 점들에 대해서 조금이나마 더 알게 된 것 같다. 아직은 확신할 수 없는 미래이지만, 이런 것들을 연구할 기회가 주어진다면 이 분야의 수준 높은 향상을 위해 힘써보고 싶다.