프로그래밍 언어 과제 # 과제 6 - 논술 에세이

2006-11757 박원형

저번에는 수학적 기호들로 점철된 두꺼운 논문들이어서 읽기가 꽤나 까다로웠지만 이번의 읽기 자료들은 웹상에 게시된 좀 더 편한 글이어서인지 좀 더 어렵지 않게 읽을 수 있었습니다. 학기도 슬슬 마지막을 향해 가면서 몰려오는 많은 숙제들 틈에서 틈틈이 읽으려다 보니 가끔은 무얼 읽고 있는 지도 모르고 눈만 지나친 적도 많이 있었지만, 어찌어찌 시간을 쪼개다 보니 읽기 자료들을 대충 한 번쯤 훑어볼 수는 있었습니다.

첫 번째 나왔던 읽기자료들이 컴퓨터와 프로그램 쪽보다는 그에 기반하는 수학과 논리학 이론들에 좀 더 비중을 둔 쪽이었다면, 이번에 나온 자료들은 이제 슬슬 수학과 논리학에서 벗어나 하나의 분야를 구축해가고 있는 프로그래밍 언어 쪽에 좀 더 비중을 두고 있어서 인지, 쉽게 관심을 두고 읽을 수 있었던 것 같습니다. 수학의 엄밀한 증명이 프로그램과 불가분의 관계에 놓여 있다는 것은 여러 이론들을 직접 실습해보면서 체감할 수 있었지만, 아무래도 수학에 관련된 글들은 짬짬히 시간을 내어 조금씩 읽기 보다는 한 번 집중해서 저자가 어지럽게 - 나름대로는 합리적이라고 생각하겠지만 - notation들에 익숙해 진채 저자의 생각을 그대로 따라 가야하는 과정에 필요하기 때문에 이런 저런 프로젝트에 치어 지내는 요즘에 읽기에는 많이 힘든 감이 있습니다. 다행히 교수님께서 이런 점까지 배려해주셨는지(?) 그다지 깊은 생각을 하지 않고도 지나가듯이 읽을 수 있는 좀 더 실제적인 프로그래밍과 관련된 읽기 자료들을 내주셔서 참 다행이라고 생각했습니다.

Beating the average에서 비밀병기라고 칭해지고 있는 Lisp은 이런 프로그래밍 언어들을 사용해 보지 않은 사람이라면 절대 이해할 수 없을 만큼 뛰어난 언어라는 데에 ML을 사용해 본 사람으로서 쉽게 공감할 수 있었습니다. 저자는 실제적인 코드를 보여주기보다 이러이러해서 Lisp이 C 같은 High level Assembly에 비해 훨씬 뛰어나다고 하지만, 제가 보기에는 Lisp과 C를 함께 사용해 보지 않은 사람이라면 - 아마도 C만 사용해본 사람이라면 절대 이해할 수 없을 것 같았습니다. 마치 개미가 세계는 2차원이라고 느끼지만 날아다니는 벌은 3차원이라고 느끼는 것처럼, 좀 더 높은 부분에서 보는 사람만이 아래 차원에서 사용되는 수단들이 얼마나 부족한지 쉽게 알 수 있을 것입니다. 저 뿐만 아니라 많은 3, 4학년 학생들은 컴퓨터 구조나 운영체제 같은 과목을 들으며 하드웨어와 직접적으로 연관되는 C에 기반한 프로그램들을 많이 다루게 되는데, 그 때마다 이 코딩은 생각을 코드로 옮기는 과정이라기보다 인간을 기계의 생각에 억지로 끼워 맞추는 과정이라는 생각을 하곤 합니다. 특히나 저는 운영체제 프로젝트를 하면서 디버깅에 정말 미숙한 모습을 보이며 같은 조원들을 많이 힘들게 하곤 하는데, 그럴 때마다 이런 언어를 사용할 수밖에 없는지에 대한 고민을 하곤 합니다. 물론 UNIX가 만들어질 당시에는 기존의 어셈블리 기반에서 벗어난 획기적인 운영체제였다고 하지만, 그때보다 훨씬 더 하드웨어 속도가 빨라지고 사용할 자원이 풍부해진 지금이라면 정말 하드웨어에 밀접한 작업을 제외한다면 사람의 생각을 그대로 옮길 수 있는 Lisp이나 ML같은 언어를 사용하는 것이 훨씬 효율적이 아닐까 하는 생각이 듭니다. 애

초에 근본적으로 생각해보면 컴퓨터를 사용하기 위해 프로그래밍 언어를 만들어 낸 것이 아니라, 프로그래밍 언어를 좀 더 빠르게 연산하기 위해 컴퓨터가 만들어진 것이니 사람의 생각을 기계에 끼워 맞추는 그런 방식의 코딩은 이제 지양되어야 하지 않을 까합니다.

저만의 생각일지도 모르지만, 프로그래머들은 좀 더 일반인이 사용하기 힘들고 어려운 - 즉 진입장벽이 높아 보이기 위해 C를 사용하면서 코드의 가독성을 떨어뜨리는 데에 약간은 비상식적이라고도 할 수 있는 우월감을 느끼는 것이 아닐까하는 생각을 가끔 하기도 합니다. 이에 관련 되어서 저도 최근에 친구에게 들은 이야기가 있습니다. 한번은 컴퓨터 시스템 설계 실험 수업 때문에 하드웨어 실습실에서 한 쪽에는 프로그램을, 한 쪽에는 기판을 놓고 작업하고 있었는데, 여러 직원들이 컴퓨터 공학부 홍보영상을 찍기 위해 방문했었다고 합니다. 그 때 직원들은 그럴 듯한 프로그래밍하고 있는 모습을 한 번 연출해달라고 하는데, 그래서 그 친구는 한 쪽에 프로그래밍 언어 5번 숙제로 나왔던 Sm5 모듈을 틀어준 채, 옆에서 납땜하는 모습을 연출해 주었다고 했습니다. 아는 사람이 본다면 전혀 상관없는 그 모습에 폭소가 나오겠지만, 모르는 사람들이 보기에는 굉장히 그럴 듯한 모습으로 보일 것입니다. 납땜하는 모습은 컴퓨터 공학부의 이미지와는 좀 동떨어져 있지 않은 가 할 수도 있겠지만, 그냥 척 보기에도 암호처럼 보이는 Sm5 모듈은 모르는 사람들에게는 꽤나 그럴 듯하게 보이지 않았을 까 합니다. 그나마 생각하고 있는 그대로를 코드로 쉽게 옮길 수 있는 ML도 외부 사람들이 보기에는 암호문처럼 보이는 실정인데, 아마 C로 짜여진 운영체제의 시스템 관리 부분이나, 컴퓨터 구조의 파이프 라이닝 코드를 보여준 다면 모르는 사람들은 코드를 이해하기조차 포기해버릴 것입니다. 원래 공학이라는 분야가 아는 사람에겐 아무 것도 아니고, 모르는 사람에게는 이해도 가지 않고 하고 싶지도 않게 하는 특성이 있다고는 하지만, 굳이 이해하기 어려운 알고리즘이나 생각도 아닌 것을 단지 어려운 코드로 포장시킬 필요는 없지 않을 까요?

Beating the average 이외에 Revenge of the Nerds 역시 Lisp의 활용성을 찬양하는 글이었는데, 정말 좋다고 찬양하는 Lisp의 Macro를 비롯한 Lisp 만의 기능들을 구체적으로 설명해주지는 않고, 정말 좋다는 말만 반복하는 것은 조금 아쉬웠습니다. 끝부분에 달린 Appendix 만에서라도 짤막하게 C와 비교해보는 코드를 실어주었으면 어땠을 까하는 생각이 들기도 하지만, 아마도 저자는 Lisp라는 한 단계 높은 언어를 사용하고 있기 때문에 굳이 하위 레벨의 언어로 설명해 줄 필요를 느끼지 못한 것이 아닐까 하는 생각이 들었습니다. 수학과의 수업을 들을 때, 학생들의 이해도를 전혀 배려하지 않은 채 - 혹은 배려하더라도 - 정말 당연하다는 듯이 세세한 증명도 거치지 않고 대수학적 내용을 줄줄이 늘어놓는 교수님과 비슷한 마인드를 갖고 있지 않았나합니다. 마치 하루살이에게 내일이라는 개념을 설명해줄 수 없듯이 - 그렇다고 학생들이 그렇게 이해도가 떨어지는 것은 아니지만 - 하위 레벨의 언어를 쓰는 사람들은 그보다 상위 단계 언어의 영역을 전혀 이해할 수 없을 것입니다. 아마도 교수님께서 학기 초에 수학과의 수업을 한 번 쯤 들어보는 것도 좋을 것이라고 말씀하신 것도 이런 이유에서 였을 것이라고 추측하고 있습니다. 아무튼, 이 글을 읽고 있는 독자가 ML이나 Lisp 등을 사용해보지 않은 사람이었다면 저자를 C도 제대로 안 써본 미친 놈 취급이나 하지 않을 까 좀 걱정스럽기는 했습니다.(아마도 '고급 프로그래밍 언어는 모두 같은 일을 할 수 있다는 기본적인 정리도 모르고 있는 사람'이라고 비하했을 지도 모를 일입니다.) 저자가 정말 Lisp이 뛰어난 언어라는 걸 설명해 주고 싶었다면 저로서도 어떻게 할

지 구체적으로 말할 수 없지만 이렇게 말로만 대놓고 C를 비난할 것이 아니라, 좀 더 실제적인 방법을 사용하는 편이 나았을 것 같습니다.

Tastes for maker는 좋은 디자인에 대해 논하고 있었는데, 자연 과학 뿐만 아니라 인문과학에서도 일반적으로 이야기되는 좋은 디자인의 개념을 프로그래밍 언어에 적용해보니 현재에 정말 많이 쓰이는 언어들은 좋은 디자인에서 한참 벗어나 있다는 것을 손쉽게 깨달을 수 있었습니다.

앞에서도 말씀드렸듯이, C나 FORTRAN 등의 언어는 처음 컴퓨터가 나왔을 때 당장 컴퓨터의 성능을 계산에 활용하기 위해 사람을 기계에 끼워 맞추어 디자인된 언어들이기 때문에 여기서도 말하듯이 '**자연을 그대로 닮지 않은** 언어'입니다. 사람의 생각을 그대로 받아들이지 못하고, 기계가 원하는 대로 점점 맞추어 주다보니 어느새 프로그래밍 언어는 사람의 생각을 그대로 옮겨 놓은 수학적 기호에서 이만큼이나 멀리 떨어져 나오게 되었습니다. 그야말로 인공적으로 갈고 다듬어져 사람이 그대로 손을 댔다가는 베일 정도로 기계적으로 날카로운 모습을 갖추게 되었습니다. 자기 자신이 notation을 정의하고 자유롭게 사용하던 수학적 언어에서 벗어난 이 모습에, Tex을 만든 Donald Knuth 조차도 자기가 해본 작업 중에 코딩이 가장 어려운 것 같다고 하소연 했다고 한걸 보면, 원래 아무나 자유롭게 주무르고 다듬어 새로운 창조물로 만들 수 있었던 컴퓨터 이전의 프로그래밍 언어의 모습은 지금의 C에서는 거의 찾아볼 수 없습니다.(혹자는 프로그래밍 언어가 원래 수학에 사용되던 언어에 기반하고 있다는 사실조차 믿으려 하지 않을 것 같습니다.) 이 역시 '**쉽게 흉내 낼 수 있어야 한다**.'는 좋은 디자인의 관점에서 크게 어긋났다고 볼 수 있을 것입니다.

게다가 독창적인 아이디어를 프로그래밍 언어에 적용할 만큼 디자인 상 언어의 유연성이 뛰어나지도 않고, 기계처럼 딱딱 떨어지는 정형성을 갖고 있는 것처럼 보이지만 막상 뜯어보면 어떻게든 기계를 이해시키기 위해 덕지 덕지 붙여놓은 코드들에서는 쉽게 원리조차 파악해보기 힘듭니다. 이렇게 일반론적으로 알려진 좋은 디자인의 이런 저런 점들을 프로그래밍 언어에 빗대어 생각해보면 맞는 점이 거의 하나도 없다는 것을 쉽게 알 수 있습니다. 모르는 사람들이 쉽게 접근하기 힘드니, 이런 환경에서는 어지간하지 않고는 시대를 뒤엎을 만한 '괴짜' 한 명을 기대해 보기도 힘듭니다. 즉, 많은 사람과 잘 구축된 환경에서만 쥐어짜듯 노력해야 그럴 듯한 프로그램을 만들어 낼 수 있다는 것입니다.

지나치게 일반론으로 흐르는 것 같기도 하지만, 이런 디자인으로는 일반적으로 한 번 성공해서 된 길 이외에 다른 부분으로는 잘 시도해보려 하지 않기 때문에 계속 생산물이 새로운 전기를 맞지 못하고 고착화하게 될 것입니다 - 그러니까 만들어진 프로그램마다 별로 특이성을 갖지 못하고 이 프로그램이나 저 프로그램이나 다 비슷한 이런 현상을 맞게 될 것입니다. 이런 현상을 타파하기 위해서는 프로그래머들의 머릿속에 박혀 있는 어떤 한 방법으로 해야 만 쉽게 코딩 할 수 있다는 생각을 깨뜨려야할 필요가 있습니다. 그런 발상의 전환은 기계의 틀에 맞춰 디자인되었기에 지극히 제한적인 생각의 틀만을 제공하는 C나 Fortran 등 기계 중심의 언어에서 이루어 질 수 없고, 인간의 사고를 그대로 코드에 반영하는 Lisp이나 ML을 통해서 이루어질 수 있다고 생각합니다. 굳이 멀리서 찾을 것도 없이, C를 주로 사용하는 운영체제 프로젝트의 경우, 평균적으로 한 프로젝트를 마치는 데 소요되는 시간은

디버깅이 70% 이상을 차지합니다. 정말로 제가 만들어낸 코드는 100줄도 채 되지 않는 데, 단지 기계의 생각을 그대로 따르지 못했다는 이유 때문에 사흘 밤도 넘어가는 시간이 소요되는 것은, 형식의 제한을 넘어선 심각한 자원의 낭비라고 생각합니다. 아마도 저자는 이런 점에 주목하고 Beating the average에서 Lisp이 자신들의 비밀병기라고 한 것이라고 생각되지만 - 정말로 사람의 생각을 코드로 그대로 옮겨 놓을 수 있다면 디버깅하는 데에 그렇게 오랜 시간이 소요되지는 않을 것이라고 생각합니다. 일반 회사에서도 프로젝트를 수행하는 데 보다 유지 보수하는 데에 - 즉 디버깅과 업데이트에 가장 많은 비용이 소요되는 것을 생각해보면, 여기에 해당하는 자원의 낭비를 줄일 수만 있다면 더 많은 시간을 새로운 것을 만들고 새로운 것을 생각해 내는 데에 사용할 수 있을 것입니다. 더군다나, 기계의 틀에 맞출 필요없이, 즉 자신의 생각에 제한의 폭을 둘 필요 없이 상상의 나래를 멀리 펼칠 수 있다면, 교황청을 등에 업고 끊임없는 지원을 받았던 밀라노에서가 아니라 멀리 떨어진 변방의 도시국가 피렌체에서 르네상스의 발판이 마련되었던 것처럼, 독창적인 생각으로 세상을 뒤엎을 만한 '괴짜'를 컴퓨터 공학 분야에서도 그리 멀지 않은 미래에 만날 수 있지 않을 까 합니다.

Typing for perl manager에서는 그다지 길다고 볼 수 없는 컴퓨터 프로그래밍의 역사 가운데 이슈가 되었던 여러 주제들 중 아직도 논란이 지속되고 있는 String type checking을 다루고 있습니다. 이전에 많은 논란이 되었던 여러 이슈들은 GOTO 문과 For & While loop의 사용, Subroutine call과 Non-subroutine, Assembly와 High level Programming 등 하드웨어와 함께 발달하는 과도기에는 어느 쪽의 손을 들어주기 힘들었지만, 지금에 와서는 당연하게 생각하는 하나의 패러다임으로 자리 잡았습니다. 프로그래밍을 처음 배울 때 GOTO 문은 스파게티 코드라 불리며 어지간하면 사용하지 않도록 배우고, Assembly와 High level Programming 역시 특수한 경우가 아닌 이상에야 일반 프로그래머들은 Assembly를 볼 일조차 드문 것이 현실입니다. 또한, Subroutine 문제 역시 recursive call도 어지간하면 많이 쓰지 않도록 교육받기에 이런 저런 이전의 이슈들은 자그마한 논란은 있겠지만 하나의 패러다임으로 자리 잡았다고 보아도 무방합니다.

여기서 저자는 이제 떠오르고 있는 이슈 중 하나로 Strong Type Checking을 거론하며 Type Checking의 필요성과 발전해온 역사에 대해 설명하고 있습니다. 저는 써본 적은 없지만 굉장히 Type에 민감하다고 알려진 FORTRAN의 경우 compile time에 type을 체크하는 Static Type Checking을 사용하고 Lisp은 performance에 영향을 줄 수 있지만 language의 굉장한 flexibility를 제공하는 Run-time Type Checking을 사용합니다. 그리고 pointer 등을 제공하여 C나 Pascal 문법의 토대를 만든 Algol-based Type system도 소개하고 있습니다.

위에서 설명한 Type Checking의 문제점은 바로 False Positive, 즉 Type Checking이 틀릴 수 있다는 것입니다. 특히나 Casting을 제공하는 C의 경우 그런 문제는 더 심각해 지는데, 아무튼 저자는 C나 Pascal의 Static Type Checking은 실패라고 결론내리고 있습니다. 또한 일반적으로 알려진 Strong Type Checking Language가 Pascal이라는 것이 정말 크나큰 불행이라고 한탄하고 있습니다.

저자는 False Positive의 문제를 개선하기 위해 새로운 정책에 입각해 디자인된 ML과

Haskell 등 진정한 Strong Type Checking Language를 소개하고 있습니다. 자신은 좀 더 많은 연구가 이루어진 Haskell을 배웠지만 ML 역시 Strong Type Language를 소개하기에는 충분하다면서 일단 ML의 Type 들을 소개하고 간단한 예시 코드를 들어 어떤 방식으로 Type Checking이 이루어지는 지 설명하고 있습니다. 저 생각으로는 앞에서 말로만 Lisp이 뛰어나다고 찬양하는 글보다는 이렇게 직접 설명해 주는 편이 많은 사람들의 Strong Type Checking Language의 사용을 촉진시키는 데에 더 효과적이라고 생각합니다.

저자는 비록 C와 Pascal의 Type Checking은 실패한 것이지만("C and Pascal Type Checking Sucks!") 그렇다고 그 것이 Type Checking Idea를 포기할 이유는 되지 못한다고 하면서 Type Checking 기술을 더욱 발전되길 기대하며 발표를 마치고 있습니다.

Typing for Perl Managers를 읽으며 이번 숙제이기도 했던 Type Checking이 어떤 과정을 거쳐 발전해 왔는가를 볼 수 있었습니다. 기계의 틀에 사고를 맞추는 데에 염증이 난 프로그래머들의 오랜 기간의 연구 끝에 사람의 생각을 거의 그대로 코드로 옮길 수 있게 되었고, 이에 가장 큰 공헌을 한 것은 누가 뭐래도 코드 스스로 타입을 체크하는 기술이라고 생각합니다. Halting problem을 이용한 증명으로 완전한 타입 체커는 존재할 수 없다는 것은 쉽게 보일 수 있지만, 많은 사람들이 포기하지 않고 연구한 끝에 Type Checking이 불가능(할 정도로 믿을 만하지 못)한 C와 Fortran에 비해서 훨씬 손쉽게 프로그램의 문제성 여부를 판단할 수 있도록 해준 것은 좀 더 사고의 틀을 확장시켜 주었다는 데에서 큰 의미를 가질 수 있다고 봅니다. 아마도 이런 기계적인 사고의 틀이 사라지는 날에, 컴퓨터는 인공지능이라는 오명을 벗을 수 있을 것입니다.