순간의 선택이 결과를 좌우한다.

2008-11707
컴퓨터공학부
전범렬

언어의 선택은 중요하다.

새로운 프로그램을 작성하고자 한다. 먼저 무엇을 해야 하는가? 가장 근본적으로는 그 프로그램이 무엇을 하는 프로그램인지부터 결정되어야 할 것이다. 이것이 결정되고 나면 이제 프로그램을 어떻게 구현할 것인가에 대해 생각한다. 이 과정에서 프로그래머는 프로그램을 작성할 언어를 선택하게 된다. 대부분의 사람들은 이 때 자신이 익숙한 언어를 선택하거나 많은 사람들이 사용하는 언어를 선택한다. 하지만 이 한 번의 선택은 프로그래머가 프로그램 작성하는 난이도를 결정할 뿐만이 아니라 후에 그 프로그램의 운명을 좌지우지 할 수도 있다.

<Beating the Average>에서는 지금의 'Yahoo Store'가 된 'Viaweb'의 성공요인이 개발 언어로 Lisp을 선택하였기 때문이라 말한다. 다른 사람들이 C++나 Perl을 이용하여 프로그램을 작성하는 동안 'Viaweb'의 개발자들은 Lisp을 이용하여 프로그램을 작성하기로 결정하였다. Lisp이 프로그램을 빨리 작성하도록 할 것이라는 확신을 가졌기 때문이다. 과거에 아무도 시도하지 않았기 때문에 그들의 선택은 위험성이 컸지만 결과는 매우 성공적이었다.

C++와 달리 Lisp은 구현이 쉽고 빨랐기 때문에 개발을 위해 많은 프로그래머가 필요하지 않았으며 짧은 시간 내에 새로운 요소들을 추가할 수 있었다. 심지어 다른 경쟁자들이 새로운 요소들을 만들어 냈다 하더라도 그들은 같은 것을 단지 2~3일 만에 구현하여 추가해 버렸다. 소프트웨어 시장의 특성상 짧은 주기로 새로운 기술이 생겨나고 이를 따라가지 못하면 시장에서 사라져 버린다. 하지만 그들의 프로그램은 모든 기술들을 따라갈 수 있었으며 심지어 경쟁자보다 더 많은 요소들을 지니고 있었다. 이러한 경쟁력을 바탕으로 그들의 프로그램은 대성공을 거두게 된다. 그러면 Lisp은 어떻게 프로그래머가 적은 비용으로 빠른 시간 내에 개발할 수 있는 환경을 제공해 주었을까?

<Revenge of the nerds>에서는 Fortran과의 비교를 통해 Lisp이 가지는 장점과 특징들에 대해서 소개하고 있다. Lisp은 1950년대 후반에 Turing Machine의 쉬운 대안으로서 정의된 언어이다. 애초에 프로그래밍 언어로써 사용하기 위해 제작된 언어가 아니다. 하지만 추후에 컴퓨터에서 Lisp을 실행할 수 있는 실행기가 만들어 졌고, 프로그래밍 언어로서 사용될 수 있게 되었다. 이 때문에 Lisp은 그 언어의 기반을 수학에 두

고 있다. 하지만 이와 달리 Fortran은 처음부터 프로그래밍 언어로서 사용하기 위해 제작된 언어이다. 그래서 그 기반을 기계적인 구조에 두고 있다는 특징을 가지고 있다. 저자는 Lisp이 수학에 기반하고 있다는 특징 때문에 간단하지만 더 강력한 요소를 가지고 있다고 주장한다.

이와 더불어 저자는 Lisp을 특별하게 해주는 요인 9가지를 소개하고 있다. Conditional과 Function Type, Recursion, Dynamic typing, Garbage-collection, Expression으로 구성된 프로그램, Symbol Type, 심볼과 상수의 트리로서 코드를 표현하는 것, 읽고 컴파일하고 실행하는 시간이 따로 구분되지 않는 것이라는 점이다.

하지만 이 중 대부분은 다른 언어에서도 찾아 볼 수 있으며 마지막 두 요인만이 Lisp을 특별하게 만들어 준다. 이와 더불어 Lisp만이 특별한 메크로 또한 그만의 장점을 가지며 Lisp은 짧은 코드만으로도 프로그램을 작성할 수 있게 해준다. 프로그램 코드의 길이는 대게 프로그램 작성시간에 영향을 미치기 때문에 Lisp이 개발시간을 줄여줄 수 있게 되는 것이다. Lisp이 가지는 이러한 특징 때문에 'Viaweb'이 성공할 수 있는 중요한 요소가 된 것이다.

지금 쓰고 계시는 언어. 왜 택하셨나요?

<Beating the Average>에서는 사람들이 관습적으로 프로그래밍 언어를 선택하는 것에 대해 비판한다. 사람들이 Lisp이 좋은 언어라는 것을 알면서도 '이미 익숙하니까', '다른 사람이 사용하니까' 기존에 사용하던 언어만을 고수하는 것이다. 저자는 프로그래밍 언어는 절반은 기술이지만 절반은 종교라는 말을 하면서 이 상황에 문제가 있음을 지적한다.

<Revenge of the nerds>에서도 비슷한 이야기가 이어진다. 하지만 이번에는 프로그래머가 아닌 프로그램에 대해서 무지한 상사들이 개발자들에게 표준적인 언어를 사용하기를 강요하는 현실을 비판한다. 상사들은 프로그래밍 언어가 모두 동일하다고 믿고 있다. 그래서 경쟁자들과 다른 언어를 사용함으로써 발생할 수 있는 위험성을 피하려 하고 표준적인 언어 사용을 강요하는 것이다.

하지만 애석하게도 프로그래밍 언어는 완벽히 동일하지 않다. 물론 모든 프로그래밍 언어는 Turing Complete하기 때문에 그들이 표현할 수 있는 능력이 다르다는 이야기는 아니다. 다만 언어마다 가지는 특징이 다르기 때문에 얼마나 잘 표현할 수 있는가라는 점에서는 분명한 차이가 있다. 예를 들면 C언어는 기계적인 동작들을 표현하는데 탁월하다. 메모리와 같은 하드웨어들을 관리하기에 C만큼 좋은 언어는 없다. 하지만 ML과 같은 언어는 프로그래머로 하여금 안전한 프로그램을 빠른 시간에 구현하도록 도와준다. 이처럼 구현하고자 하는 문제에 따라 적합한 프로그래밍 언어가 서로 다르다.

따라서 이 특징들을 잘 고려하여 그에 적합한 언어를 선택해야하지만 사람들은 익숙한 언어만을 고집하기 때문에 문제가 있는 것이다.

문제마다 적합한 언어가 있다는 점은 여러 문제들이 혼합된 프로그램을 효율적으로 구현하는 방법에 대해서도 생각해 볼 수 있게 해준다. 프로그램이 작다면 그 속에 한, 두 개 정도의 문제만이 있겠지만 프로그램의 규모가 크다면 그 속에 다양한 문제들이 존재한다. 앞에서 말하였듯이 문제에 따라 그를 해결하기에 적합한 언어는 다르다. 따라서 하나의 프로그램 내에서도 각 문제에 적합한 프로그래밍 언어는 서로 다를 수 있다. 나도 예전에 다른 언어로 프로그램을 작성하면서 특정 부분을 구현할 때 '아 이 부분은 OCaml로 짜면 더 쉬울 텐데'라는 생각을 해본 적이 있다.

이런 이유에서 하나의 프로그램이더라도 여러 언어를 혼합하여 작성하는 것이 더 좋은 결과를 만들어낼 수 있다. 인라인 어셈블리가 한 예이다. C언어 보다 어셈블리 언어가 더 적합한 경우 C프로그램 내에 어셈블리 코드를 추가하는 것이다. 다른 예로 어떤 프로그램을 작성할 때 하드웨어와 직접 인터페이스하는 부분은 C로 작성하고, 나머지 부분은 구현이 간단한 ML로 작성하는 경우도 생각해볼 수 있다.

하지만 대부분의 사람들은 프로그램을 하나의 언어로만 작성하고자 경향이 있는 것 같다. 그들은 어떤 문제에 대해 특정 언어로 작성하는 것이 더 좋다고 인지하고는 있다. 하지만 그들은 여러 언어를 사용하여 프로그램을 작성하지는 않는다. 나 또한 'OCaml'로 구현하는 것이 더 좋다고 생각한 부분을 결국에는 그냥 사용하던 기존의 언어로 작성하였다. 왜 더 좋다는 사실을 알면서도 사용하던 언어만으로 작성할까?

첫 번째 이유로는 여러 가지 언어를 사용하여 프로그래밍을 하더라도 그 연결과정이 까다롭기 때문이다. 이전에는 알지 못했지만 이번 에세이를 작성하면서 연결방법에 대해 찾아보니 OCaml코드를 C에서 사용할 수 있게 하는 방법이 있었다. 하지만 이를 위해 몇 가지 과정을 거쳐야했고 C가 아닌 다른 언어와는 연결하는 방법은 존재하지 않았다. 이런 과정을 거치지 않더라도 어려움은 있겠지만 그 부분을 이미 사용하고 있는 언어로 작성하는 일이 전혀 불가능하지는 않기 때문에 대부분의 사람들이 그냥 사용하던 언어로만 작업하는 것이다.

두 번째로는 사람들이 여러 가지 언어로 프로그램을 구현하는 작업하는 일을 잘 하지 않기 때문이다. 위에서 밝혔듯이 OCaml프로그램과 C프로그램을 연결하는 방법이 존재하지만 이를 알고 있는 사람은 많지 않다. 많은 사람들이 한 가지 언어만으로 구현하는 것을 너무 당연하게 여겨 왔기 때문이다.

아직 많은 프로그래밍 언어들은 쉽게 다른 프로그래밍 언어와 연결할 수 있는 환경을 제공하지 않고 있다. 하지만 세부 문제마다 적합한 언어로 작성하여 이를 하나로 합치는 방법은 충분히 장점이 있어 보인다. 따라서 추후에 언어 간의 연결을 쉽게 할 수 있도록 지원하는 방향으로 프로그래밍 언어가 발전한다면 프로그래머들이 더 좋은 프로그램을 쉽게 작성할 수 있는 환경을 제공하여 'Viaweb'의 성공과 같은 효과를 낼 수 있지 않을까 생각해본다.

언어는 좋은 프로그래머를 만든다.

지난 에세이에서는 C언어와 ML과 같은 함수형 언어의 차이에 대하여 이야기하면서 언어가 프로그래머가 표현하는 법을 한정하고 있다고 하였다. 그러면 프로그래머가 두 언어를 모두 사용하면 어떨까? 이 질문의 답을 위해 이번에는 Lisp이나 ML 언어를 배움으로서 더 좋은 프로그래머가 되는 법에 대해 말해보고자 한다.

<Beating the Average>에서 저자는 Lisp을 배우는 것이 더 좋은 프로그래머가 될 수 있게 도와준다고 주장한다. 라틴어를 배움으로써 영어에 대해 더 깊게 이해할 수 있듯이 Lisp을 배우는 것이 다른 프로그래밍 언어를 작성하고 이해하는데 도움을 준다는 것이다.

이는 이번에 읽을거리 중 가장 가슴에 와 닿았던 대목 중 하나이다. 실제로 내가 OCaml을 배운 것이 다른 프로그래밍 언어로 프로그래밍을 하는데 많은 영향을 주었기 때문이다. 작년에 프로그래밍 언어과목에서 Scheme과 OCaml을 접하기 전에는 오직 C나 Java만 프로그램을 작성했다. 작은 프로그램을 하나를 작성하는데도 많은 오류들을 봐야만 했다. 하지만 이들 언어는 생성되는 프로그램의 타입을 따로 신경 쓰지 않아도 동작하였기에 나의 프로그래밍 스타일은 이들 언어에만 맞추어져 있었다. 그래서 처음 OCaml을 접하였을 때에는 강력한 Type Checking System때문에 상당히 고생했었다. 하지만 OCaml로 숙제들을 하면서 항상 타입을 고려하면서 프로그래밍을 하는 습관이 들어서 인지 다른 과목의 숙제들을 진행하기 위해 C나 Java를 사용할 때에도 나도 모르게 타입을 고려하면서 프로그램을 작성하고 있었다. C를 사용하더라도 더 이상 예전에 나를 괴롭히던 오류들을 보지 않아도 됐다. OCaml을 배운 것이 내가 좀 더 안전한 프로그램을 작성하는 좋은 프로그래머가 될 수 있게 도와준 것이다.

더불어 OCaml의 습득은 문제를 재귀적으로 보는 능력을 함양하고 프로그램의 구현 방법의 선택에 대한 안목을 넓혀 주었다. 재귀적인 발상은 프로그램을 구현하는데 있어 문제를 간단히 만들어 주고 이를 쉽게 해결할 수 있게 하는 틀을 제공한다. 하지만 C, Java를 배우면서 재귀가 쉽게 프로그램을 구현할 수 있는 좋은 방법임에도 불구하고 '자원을 많이 요구하기 때문에 나쁘다'라는 관념을 가지고 있었다. OCaml을 배우면서 많은 프로그램을 재귀로 작성하였고 그 과정에서 재귀에 대해 더 잘 이해할 수 있었다. 그로 인해 어떤 프로그램에서 재귀를 사용하고 사용하지 않아야 하는지에 대한 선택을 하는 안목이 넓어졌다. 이처럼 언어의 습득만으로도 좋은 프로그래머가 될 수 있게 하는 것이다.

좋은 언어는 무엇인가?

앞에서 프로그램을 개발하는데 언어의 선택이 어떠한 의미가 있으며 그 선택이 미치는 영향에 대하여 이야기 해보았다. 그러면 우리는 프로그램을 작성하기 위해 어떤 언어를 선택해야 하는가? 즉, 좋은 언어는 무엇인가에 대해 질문을 할 수 있다.

좋은 언어에 대해 말하기에 앞서 그보다 더 일반적인 좋은 디자인에 대해서 먼저 이야기해 보자. <Taste for marker>에서는 공학, 예술, 수학, 과학 등에 걸쳐 좋은 디자인이 가지는 요소들에 대해 소개하고 있다. 몇 가지 요소들을 소개해 보면 먼저 좋은 디자인은 단순하고, 암시적이며, 자연을 닮았다는 점이다.

<Growing a Language>에서는 좋은 언어가 가져야하는 특징들에 대해서 이야기한다. 언어는 다양한 방식으로 만들어질 수 있다. 언어를 하나의 기계라 생각해보자. 완전히 새로운 부품을 만들어 내어 새로운 기계를 만들 수도 있고 아니면 이미 가지고 있던 부품만을 조합하여 새로운 기계를 만들어 낼 수도 있다.

그러면 부품의 종류가 많은 것이 더 좋은 것인가? 아니면 부품이 종류가 적은 것이 더 좋은 것인가? 얼핏 보면 위의 질문에 대한 답은 간단해 보인다. 부품의 종류가 많으면 더 다양한 종류의 기계를 만들 수 있기 때문에 더 좋을 것이라고 판단할 수 있다. 하지만 기계를 조립하기 위해서는 부품의 종류가 많을수록 그 부품들이 어떻게 사용되는지를 모두 알아야 하기 때문에 사용하기 까다롭다는 단점을 지닌다. 그렇다 하더라도 아직까지는 부품이 많은 것이 더 좋아 보인다. 하지만 부품들을 이용하여 새로운 '부품'을 만들어 낼 수 있다면 어떨까? 그러면 더 이상 부품의 종류가 많은 것이 좋다고 말할 수 없을 것이다.

이는 프로그래밍 언어에서 '작은 언어와 부유한 언어 중 어떤 것이 더 좋은가?'라는 질문의 답과 일맥상통한다. 프로그래밍 언어의 역사를 살펴보면 위의 질문에 답을 알아낼 수 있다.

Fortran과 PL/I는 3,40년 전에 만들어진 프로그래밍 언어이다. Fortran은 부품의 종류가 작은 언어이고 PL/I는 부품의 종류가 다양한 언어이다. 이와 함께 또 다른 둘의 큰 차이는 Fortran은 자랄 수 있게 디자인된 언어이고 반면 PL/I는 자라기 힘들게 디자인 된 언어이다. 그러면 두 언어 중 무엇이 더 좋은 언어인가? 답은 의외로 간단하다. 지금까지 사용되는 언어가 더 좋은 언어이다. 진화론에서 적자생존의 법칙을 언어에 적용해 보는 것이다. PL/I는 더 이상 사용되지 않지만 Fortran은 새로운 부품들을 만들면서 자라나 현재까지 계속 사용되고 있다. 즉, 좋은 언어는 작으면서도 자랄 수 있는 언어여야 한다.

작고 자랄 수 있는 언어는 <Taste for maker>에서 이야기한 좋은 디자인이 가지는 요소들을 가지고 있다. 작기 때문에 단순하고 배우기 쉽다. 또한, 이미 있던 것을 조합함으로써 새로운 부품을 만들어내는 암시성을 지니고 있고, 자연에서처럼 생물이 자라

듯이 언어가 자라난다. 단순하기 때문에 배우기 쉽고, 사람들이 사용하면서 언어가 점점 성장해 사람들의 요구사항들을 잘 반영할 수 있어야 좋은 언어로써 오랫동안 살아남을 수 있는 것이다.

글을 마치며

세상에는 많은 도구가 있지만 상황에 적합한 도구는 따로 있다. 과일을 깎는 데는 과도를 사용해야 하고, 고기를 자르는 데는 식칼을 사용해야 하며, 나무를 자르는 데는 톱이 적합하다. 프로그래밍 언어 또한 프로그램 작성을 위한 도구이다. 하지만 많은 사람들은 언어의 선택에 있어서 익숙한 언어를 택한다. 이것만으로도 구현하고자 하는 것을 모두 구현할 수 있기 때문이라 말한다. 하지만 이는 '과도를 가지고도 모두 자를 수 있고 나는 과도를 다루는데 익숙하기 때문에 나무를 자르는데도 과도를 사용할 것이다' 라는 한심한 말과 별반 다르지 않다. 상황에 맞는 프로그래밍 언어의 선택이 필요함을 알 수 있는 대목이다.

옛말에 '좋은 목수는 연장을 탓하지 않는다.'라는 말이 있다. 얼핏 들으면 도구는 어느 것이던 간에 모두 동일하니 실력을 키우라는 말을 내포하는 것처럼 보인다. 하지만 좋은 목수가 연장을 탓하지 않는 이유는 모든 도구가 동일하기 때문이 아니다. 그들은 상황에 따라 적합한 연장에 대해 잘 알고 있으며, 항상 그에 맞는 연장을 사용하기에 연장을 탓할 이유가 없는 것이다. 프로그래밍에서도 마찬가지다. 좋은 프로그래머는 언어를 탓해선 안 된다. 대신 다양한 언어를 익힘으로써 어떤 상황이 오더라도 그에 맞는 언어를 선택하는 안목을 길러야 할 것이다. 많은 사람들이 이 글을 읽고 언어 선택의 중요성을 깨달아 '과도를 가지고 나무를 자르는' 우를 범하지 않았으면 한다.

주어진 글들을 읽으면서 '나는 어떤 기준으로 프로그래밍 언어를 선택하였는가?'에 대해 되돌아보게 되었다. 예전에 베이직을 잠깐 배운 것 외에는 대학에 진학한 이후에야 제대로 프로그래밍 언어들을 배우게 되었기 때문에 딱히 익숙한 언어는 없었다. 하지만 항상 프로그래밍 언어를 선택할 때에는 마치 <Revenge of the nerds>의 Pointy-hair header처럼 많은 사람들이 사용하니까 그 언어를 선택해 온 것 같다. 이번 기회를 통해 프로그래밍 언어의 선택에서 가진 자세를 반성하고 그 중요성을 깨닫게 되었다.