‘아름다운’ 프로그래밍 언어-언어 설계의 철학

2008-11710 정우근

서론

프로그래밍과의 첫 만남은 매우 고통스러웠던 것으로 기억이 난다. 대학교 1학년, 컴퓨터공학에 처음 입문하는 수업에서 어셈블리어와 C언어를 배웠던 것이 프로그래밍과의 첫 만남이었는데, 그 때 꽤 힘들게 프로그래밍을 익혔었다. 특히 어셈블리보다는 C와의 만남이 더 힘들었던 기억이 난다. 과목을 수강하면서 어찌어찌, 마치 사람의 행동을 흉내내는 원숭이처럼, 프로그래밍을 하긴 했지만 많은 부분들이 직관적으로 와 닿지 않았기 때문이다.

C의 이해를 힘들게 하던 요소는 그 잡다한 문법들이었다. static, external같은 specifier들이 그 대표적인 예이다. 분명 나의 숙제는 local변수들과 함수들만으로 이루어져 있는데. 저런 specifier는 필요 없을 것 같은데. 포인터와 배열의 문제도 그랬다. 왜 같은걸 두 개나 만들어서. 포인터가 있으면 배열은 필요 없는 게 아닐까. 좀더 깔끔하게 언어를 정제할 순 없는 걸까.

함수형 언어와의 만남은 그런 면에서 매우 즐거웠던 것으로 기억된다. 특히 scheme과의 만남이 매우 즐거웠었는데, scheme은 문법이라고 할만한 것이 거의 없는데도 만들고자 하는 프로그램을 ‘아름답게’ 만들 수 있었기 때문이다. ML또한 매우 매력적인 언어였다. 지저분하게 손을 대지 않아도 ‘아름답게’ 프로그램이 완성되는 묘미 때문일까. 프로그래밍 원리 시간에 이러한 ‘아름다운’ 코드들을 작성하면서, 동시에 다른 과목의 숙제를 C로 ‘아름답지 못하게’작성하면서, C에 대해 많은 불평 불만을 늘어놓았던 기억이 있다.

재미있는 사실은, 당연히 널리 공감되리라 생각했던 이러한 불평 불만은 미리 C를 접하고 C를 이용하여 프로그래밍을 하던 학우들(이하 C-oriented)과의 대화에서는 항상 논쟁거리가 되곤 한다는 것이다. C-oriented들이 가장 많이 내놓는 반박은 주로 다음과 같다. “C는 실용적이잖아.” “ML이나 scheme은 표현하기가 까다로워서 프로그램을 마음대로 할 수 없다고.” “아무리 불평해도 C가 대세란다.”

이러한 논쟁 과정에서 많은 의문들이 생겼었다. 왜 사람들은 이해하기 힘든 C를 사용하는 것일까. 그보다 앞서 왜 C는 저렇게 이해하기 힘들고, 반대로 이해하기 쉬운 언어는 어떤 조건이 필요할 것인가. 나아가 아름다운 언어가 되기 위한 조건은 무엇일까.

이번 에세이는 이러한 의문들이 해결되는 과정을 그리고자 한다. 우선 이해하기 쉬운 언어의 조건을 설정하고, 구현법에 대해 생각해보며, 더 나아가 아름다운 프로그래밍 언어를 만들기 위해서는 어떠한 과정을 거쳐야 하는지에 대해 고찰하고자 한다..

C는 왜 이해하기 힘든가

우선 C에 대해 살펴보자. 왜 C는 지금과 같은 모습을 하게 되었는가? 이 의문에 대한 해답은 Dennis M.Ritchie의 paper, “The Development of the C language”에서 얻을 수 있다. 이 paper는 C의 탄생과 성장에 대해서 자세하게 소개한다.

우선 C의 아버지격인 B, 할아버지격인 BCPL을 소개한다. BCPL은 본래 MULTICS라는 기계와 밀접하게 관계된 언어인데, 하드웨어의 발전과 함께 새 언어의 필요성이 대두되어, 정확히는 유닉스 기반에서의 새로운 시스템 프로그래밍 언어가 필요하게 되어 B라는 언어로 발전하게 된다. 두 언

어는 많은 특징을 공유하고 있으나, 몇몇 중요한 부분이 변화하게 되었다. 예컨대 프로시져 내에서 또다시 프로시져를 정의하는 것이 금지되었으며, 글로벌벡터의 처리가 더 발달하게 된다. 또한 주석 처리법 등의 사소한 변화도 존재한다. 이렇게 해서 만들어진 B는 원시적인 C의 모습과 유사한 형태를 가지게 된다.

그러나 BCPL과 B의 치명적인 단점-타입 시스템의 부재로 인해 B는 다시 수정되고, C가 탄생하게 되었다. B는 word라는 유일한 형태의 타입만을 가졌으나, C에서는 int와 char, 그리고 그로 이루어진 배열과 포인터를 제공하여 B보다 풍부한 타입 시스템을 지닌다. 이러한 타입 시스템의 도입 후, logical operater의 변화와 preprocessor의 도입 등의 발전을 거쳐 현대적인 의미의 C를 완성하게 된다.

이후의 C는 standard I/O등의 도입으로 인해 portability가(정확히는 unix system의 portability가) 증가하고, 널리 쓰이게 되어 표준 규격이 생기며 가장 활발하게 사용되는 언어가 된다. Paper는 이러한 C언어의 단점과 성공 이유를 분석하며 내용을 마무리한다.

이 paper에서 얻을 수 있는 C의 가장 근본적은 특성은, C가 철저한 기계 중심 언어라는 점이다. C는 그 조상 격 언어들이 전부 기계를 구동하는 데 그 목적을 둔 언어들이었으며, C 자체도 기계의 발달과 함께 발달해 왔다.

이러한 측면에서 살펴볼 때 C의 문법을 이해하기 힘든 것은 자명하다. C는 기계 중심 언어이기 때문이다. 기계 중심 언어를 인간이 이해하기 힘든 이유는 두 가지로 요약할 수 있다. 우선 이러한 언어의 흐름과 달리 인간의 사고는 기계 중심적이지 않다. 프로그램은 본디 프로시져의 집합이고, 프로시져는 인간의 생각을 표현하는 도구인데, 이러한 도구를 인간의 사고와 친숙하지 않은 방식으로 서술한다는 것은 자연스럽지 못하다. 또 다른 이유로는 기계 중심 언어가 가지는 조잡함을 들 수 있다. 인간의 사고와 달리 기계는 기술적 발전에 영향을 받아 끊임없이 변화하고, 기계 중심 언어는 그 변화에 맞추어 조금씩 문법을 변화시키기 때문에 문법에서 조잡함을 느낄 수 밖에 없다. 예컨대 long, unsigned등의 자료형은 기계가 지원하는 자료형에 맞추어 추가된 것으로, 기계 중심적인 언어가 조잡함을 보이는 단적인 예이다.

위의 논의로부터 생각해 보았을 때, 탈 기계적인 사고는 인간이 이해하기 쉬운 언어를 만드는 데에 필수적이라는 결론을 얻을 수 있다. 즉, 프로그래밍 언어를 디자인 할 때의 기준은 우선적으로 기계가 아닌 다른 어떠한 추상적인 개념, 즉 철학에 기반해야 한다는 것이다. 그렇다면 이 새로운 디자인 철학은 어디에 근거하여 만들어야 할 것인가.

논리학과 lambda calculus, 그리고 ML

이는 기계의 발달과 별개로 발전한 컴퓨터과학의 또다른 한 축, 수학과 논리학의 고찰에서 하나의 해답을 찾을 수 있다. 이러한 고찰이 어떤 철학을 제시할 수 있는지를 알아보기 위하여, Philip Wadler의 paper, Proofs are Programs: 19th Century Logic and 21st Century Computing를 살펴보는 것이 도움이 될 것이다.

이 paper는 논리학의 이론과 프로그래밍 문제를 연결하여 설명함으로써 논리학과 컴퓨터과학이 연계될 수 있음을 보인다. 우선 논리학의 증명 체계인 Gentzen's natural deduction를 언급하고, 이를 통하여 모든 논리 증명을 체계적으로 할수 있음을 보인다. 그 후 논리학의 또다른 체계인 lambda calculus를 설명하며, 이 둘이 실제로는 같은 체계임을 보인다. 그리고 lambda caculus의 reduction rule을 Gentzen's natural deduction에 적용시켜, 복잡하게 전개된 논리증명을 가장 단순한 형태로 바꿀 수 있음을 보인다. 또한 이러한 아이디어들이 프로그래밍 영역에 미치는 영향들

을 언급한다

프로그래밍 언어를 결정하는 철학을 얻기 위해서는 lambda calculus에 주목할 필요가 있다. Lambda calculus는 Church가 고안한 체계로서, 본래 논리 체계를 쉽게 표현하기 위해 고안되었다. 그런데 이 체계는 기계적으로 연산할 수 있는 모든 함수들을 표현할 수 있음이 밝혀진다. 좀더 자세히 말하자면, lambda calculus로 표현 가능한 함수들의 위상은 turing-computable한 함수들의 위상과 일치한다. 아이디어는 여기서 시작한다. 즉, 위에서 언급했듯이 프로그램은 프로세스, 즉 함수들의 집합이고, 따라서 lambda calculus로 함수를 모두 표현할 수 있다는 것은, 프로그램을 lambda calculus로 표현할 수 있을 것임을 시사한다.

그렇다면 lambda calculus에서 얻는 프로그래밍 철학, 즉 lambda calculus로 표현하는 언어가 기계 중심의 언어보다 나은 점은 무엇인가. 정답은 인간의 사고흐름에 부합하는 프로그램을 구현할 수 있다는 점이다. Lambda calculus는 논리의 전개를 인간이 이해하기 쉬운 형태로 표현하기 위해 등장한 체계이고, 따라서 인간의 사고과정과 유사하도록 구성되어 있다. 즉, lambda calculus의 표현식을 차용하여 프로그램을 표현한다면 이는 곧 인간의 사고과정을 따르는 프로그램이 될 것이다.

위와 같은 아이디어가 과연 실제로 활용될 수 있을 것인가. 물론 그렇다. 친숙한 언어, ML이 그 살아있는 예시이다. ML이 실제로 어떤 철학을 가지고 제작되었는지는 ML의 창시자 Robin Miller와의 인터뷰 자료에서 그 언급을 찾아볼 수 있다. 그는 ML을 tool of reasoning이라고 부르며, 실제로 ML을 LCF, 즉 logic for computable functions를 프로그래밍 언어로 구현한 metalanguage라고 소개하고 있다. 이 logic for computable function은 곧 lambda calculus와 동치이며, 따라서 ML이 우리가 위에서 구현하고자 했던 그 언어에 해당함을 알 수 있다.

이상에서 우리의 첫 번째 목적, 이해하기 쉬운 언어를 구현하는 법에 대해서 논의해보았다. 정리하자면 다음과 같다. 기계 중심의 언어는 그 사고방식과 발달 과정의 한계로 인해 인간이 이해하기 어려운 형태를 가진다. 따라서 인간의 사고 형태와 비슷한 언어를 구상해야 하는데, 이를 위한 아이디어를 인간의 사고 그 자체인 논리학의 분야에서 찾아올 수 있다. Lambda calculus는 그 표현 범위가 컴퓨터 프로그램이 표현할 수 있는 범위와 같기 때문에 이러한 아이디어에 적합하고, 그렇게 해서 만들어진 언어의 예시로는 ML을 들 수 있다.

아름다운 언어란?

이제 한 단계 더 나아가서 생각해볼 차례가 되었다. 이해하기 쉬운 언어는 아름다운 언어인가? 프로그래밍 언어가 가져야 하는 다른 조건으로는 어떤 것을 살펴볼 수 있을까? C-oriented들은 이러한 측면에서 ML을 공격할 수 있다. "ML은 실용적이지 못하다. 교육용 외에 용도가 없지 않느냐. 비 실용적인 언어는 아름답지 못하다."라는 식으로 말이다. 물론 Jane street의 상용 금융 프로그램 등의 예시는 충분한 반례가 되지만, 이러한 비판이 시사하는 점은 언어의 가치는 단지 이해하기 쉬움의 여부로만은 결정할 수 없다는 점이다.

그렇다면 아름다운 프로그래밍 언어가 가져야 할 중요한 덕목들은 어떻게 설정할 것인가. 이를 위해서는 컴퓨터 분야의 특성을 생각해볼 필요가 있다. 프로그래밍 언어는 결국 컴퓨터의 특성과 어떤 식으로든 연계되기 때문이다.

컴퓨터 분야의 특성에 대한 논의와 정리는 Robin Miller의 또 다른 글, Semantic ideas of Computing에서 얻을 수 있다. 그는 중요한 4가지 성질(같은 프로그램의 문제, 안과 밖의 문제, 차례대로의 프로그램 문제, 이동성의 문제)를 중심으로 프로그래밍에 관한 연구가 지니는 특성을 서

술한다.
따라서 아름다운 프로그래밍 언어를 다음과 같이 정의해 볼 수 있다. 이러한 여러 프로그램의 특성들이 효과적으로 서술되고, 또 그와 관련된 문제들을 해결하는 데에 도움을 줄 수 있는 언어는 프로그램의 본질을 서술하는 데에 가까우므로, 아름답다고 할 수 있다.
추상적인 논의만 지속되는 것을 피하기 위해 적절한 예시를 들자면, 프로그래밍 언어에서 자연스럽게 interaction시의 동시성 문제를 해결할 수 있는 언어나, 강력한 타입 체킹을 통해 프로그램이 하는 일을 예측하게 (즉, 동일한 다른 프로그램을 제시하는 역할과 유사하다) 해주는 프로그래밍 언어는 위에서 논의하고 있는 아름다운 프로그래밍 언어의 조건을 부분적으로 만족시시킨다. 그렇다면 모든 조건을 만족시키는 언어의 개발은 어떤 식으로 이루어져야 할 것인가.
해결책을 얻기 위한 실마리는 다시 논리학, 그리고 수학으로 돌아오게 된다. 다시 Robin Miller의 인터뷰를 참조한다면, "OOP, Imperative program, 심지어는 functional program까지 통틀어 설명할 수 있는 논리체계"를 구축 중이며 이는 가능할 것이라 한다. 이러한 논리 체계의 구축은 현재 존재하는 모든 프로그래밍 언어들을 포괄하여 설명할 수 있는 하나의 수학체계로 이어질 것이며, lambda calculus에서 ML을 얻어낼 수 있었듯, 이러한 발전된 수학체계로부터도 새로운 프로그래밍 언어를 만들어 낼 수 있다. 이러한 언어는 진정한 의미로 프로그램의 특성을 잘 표현하는 언어일 것이고, 따라서 진정 아름다운 프로그래밍 언어라고 할 수 있을 것이다.

결론

이상에서 '아름다운 프로그래밍 언어란 무엇인가'라는 다소 추상적이고 주관적인 주제에 대해 논해보았다. 에세이를 위해 읽기자료들을 읽으면서, 특히 Proofs are Programs: 19th Century Logic and 21st Century Computing를 읽으면서, 처음부터 프로그래밍 언어를 이러한 배경과 함께 제시하여 배웠으면 더 좋지 않았을까 하는 아쉬움이 많이 들었다. 또, 논리와 수학이 컴퓨터 과학에서 생각보다 더 큰 중요도를 가짐을 깨닫게 되었다. 물론 프로그래밍 언어를 설계함에 있어 하드웨어를 무시할 수는 없겠지만, 언어의 설계 과정에서는 보다 이론적이고 논리/수학에 기반한 설계 철학이 있어야 한다고 생각한다. 또한 이러한 논리와 수학에 대한 더 깊은 고찰과 포괄적은 이론 체계를 구축함으로써, 더 아름다운 프로그래밍 언어를 설계할 수 있을 것이라 기대해 본다.