프로그래밍 언어 Essay #1

-발전가능한 언어로써의 프로그래밍 언어-

2007-11727
전기공학부
신상민

1. 들어가며

언어(言語)라는 것은, 모든 인류에게 정의하기 힘든 개념일망정 친숙하지 못한 개념은 아니다. 우리는 태어나고 자라면서 자연적으로 모국어를 체득하고 필요에 따라서는 그 이상의 언어를 의식적으로 배우기도 한다. 심지어 일부의 동물에게까지 언어의 범주는 확장되어 있는데, 그것은 언어가 모든 사회성을 가진 객체에 있어 의사소통의 필수적인 수단으로써 기능하기 때문이다. 그것은 한 객체가 갖고 있는 생각과 의미를 다른 객체에게 전달하는 일련의 과정이며 체계이다. 그렇기 때문에 인간의 언어는 인간 객체 간의 의사 전달의 필요로 인하여 자연스럽게 생겨났다.

목적의 측면을 배제하고, 기능의 측면에서 바라본다면 프로그래밍 언어 또한 기존의 언어의 정의를 크게 벗어나지 않는다. 프로그래밍 언어는 인간 객체가 컴퓨터로 대표되는 계산가능한 기계에게 의사를 전달하고 원하는 작업을 수행하게끔 명령을 내릴 수 있도록 한다. 그러므로 프로그래밍 언어 또한 기존에 존재했던 언어의 범주에 속하며, 대부분의 언어가 갖는 성질을 그대로 상속받는다.

그럼에도 불구하고 프로그래밍 언어와 기존의 언어는 결정적인 면에서 중요한 차이를 갖는데, 이것은 자연 언어를 프로그래밍 언어를 보다 효율적으로 발전시켜 나갈 수 있도록 하는 데 중요한 뒷받침이 된다. 이 특징은 크게 두 가지로 나누어진다.

첫째는, 프로그래밍 언어가 갖는 가치 중립성이다. 명실상부한 세계 공용어가 영어임에도 편의를 위해 영어를 모국어로 바꾸는 국가는 없다. 그것은 자연 언어가 그 자체로 정치, 문화, 사회 전반의 많은 것을 담고 있는 유산으로써 기능하기 때문이다. 따라서 자연 언어는 가치 중립적일 수 없으며, 그러므로 사용자가 마음대로 하나의 언어를 다른 언어로 대체하기 힘들다. 반면 프로그래밍 언어는 그 자체로 의미를 갖지 않으며 따라서 효율성이 최우선이 되어 언어의 대체 사용이 가능하다.

둘째는, 프로그래밍 언어가 갖는 인위성이다. 자연 언어가 오랜 시간의 창조와 수정 및 보완의 시간을 가졌던 것은, 자연 언어는 특정한 집단에 의해 만들어져 배포된 언어가 아니며, 따라서 사용자 전원이 언어를 개선해 나갈 수 있었던 발전 체계를 갖고 있었기 때문인데, 이와 같은 상황에서 언어가 최적의 상태로 효율적인 구조를 띄고 있기를 바라는 것은 무리이다. 반면, 프로그래밍 언어는 특정 집단에 의해, 특정 목적과 환경에 최적화되어 배포된 언어이다. 따라서 '인위성'을 갖는다고

감히 말할 수 있는데, 이것은 곧 프로그래밍 언어는 기술이 뒷받침되기만 한다면 철저하고 안정성 있는 계획 하에 디자인된 채 배포되어 사용될 수 있다는 사실과 일맥상통한다.

즉, 프로그래밍 언어는 철저하게 계획되어 안정성 있는 논리를 이룰 수 있으며, 이와 같이 이상적인 언어가 등장했을 때 범용성을 갖기도 쉽다. 즉, 프로그래밍 언어는 그 어떤 자연어보다도 발전가능성이 큰 언어이다.

그렇다면 지금의 프로그래밍 언어는 어떤 점에서 '미개한가', 또한 어떤 측면에서 '발전 가능한가', 그리고 어떤 종류의 언어가 '발전에 적합한가'라는 화두는 매우 중요하고도 실용적인 화두이다. 이와 같은 물음들에 제대로 답할 수 있을 때 프로그래밍 언어는 그에 상응하는 발전을 이루어낼 수 있을 것이다. 이 레포트에서 다룰 내용은 그 중에서도 극히 일부분에 불과하지만, 과거와 현재의 프로그래밍 언어, 미래의 프로그래밍 언어에 대해 서술함으로써 위와 같은 질문에 대한 해답의 기저를 이룰 부분을 다룰 것이다.

2. 과거와 현재의 프로그래밍 언어

과거와 현재의 프로그래밍 언어는 많은 한계를 갖는다. '기존의 프로그래밍 언어는 미개하다' 라는 문장은 어쩌면 굉장한 설득력을 갖는 문장일지도 모른다. 컴퓨터는 모든 일을 할 수 있고, 현재 프로그램의 제한은 기계의 능력 부족보다는 프로그래머의 능력 부족에서 기인한다. 그러므로 미개한 기존의 프로그래밍 언어는 서론에서 제시한 바와 같이 발전이 매우 실용적으로 기능할 수 있기에 반드시 발전되어야만 한다.

그렇다면 기존의 프로그래밍 언어를 분석하는 것은 어떤 언어가 발전에 적합한가를 판별하는 데에 있어 매우 중요한 역할을 할 것이다. 구조적으로 발전 가능성이 없는 언어를 발전시키기 위해 들이는 노력은 자칫하면 무의미한 수고로 이어질 공산이 크다. 그러므로, 기존의 언어를 분석하는 것은 앞으로의 프로그래밍 언어의 발전 방향을 예측하는 것만큼이나 프로그래밍 언어의 발전에 있어서 중요하다.

1) C와 명령형 언어

현재 가장 잘 알려져 있고 가장 많은 사람들이 도구로써 사용하는 언어는 C언어이다. 실제로 이 수업을 제외한 절대 다수의 프로그래밍 수업에서 C를 공식적인 툴로 사용하고 있고, 많은 연구 및 산업의 현장에서 C를 이용한다. 그것은 C의 이식성과 강력한 하드웨어 제어 기능의 공존으로 인한 이점에서 기인한 것이기도 하나, 대체로 그 범용성으로 인해 관습적으로 사용해온 경우가 많다. 즉, 시장의 선점이 예상 외로 큰 힘을 발휘하는 것이다.

그러나 C언어를 좋은 언어라고 말할 수 있는가? 나아가, C언어를 지속적인 발전이 가능한 언어로 분류하고 C언어를 발전시키기 위한 노력을 지속하는 것이 의미가 있는가? 이에 대해서는 회의적인 입장을 견지하는 사람이 많다. '이식 가능한 고급 어셈블러'라는 비아냥거림은 비단 의미없는 시장 선두에 대한 시기에서 비롯된 것

만은 아닐 것이다. 즉 C언어가 친숙한 언어이고 따라서 C언어가 편리한 언어라는 사실은 C언어가 좋은 언어라는 사실과 C언어가 발전가능한 언어라는 사실과 구분해서 받아들여야 할 것이다.

C언어는 그 긴 역사만큼이나 드라마틱한 변화부터 세세한 변화에 이르기까지 상상을 초월하는 진화 과정을 갖는 언어이다. MIT와 General Electric, Bell Telephone Laboratories가 진행한 Multics Project가 그 효용성과 예산의 문제로 중단되고, Ken Thompson이 프로세스에 대한 개념, 트리 형식의 파일시스템 구성과 command line interpreter에 대한 개념 등 Multics Project로부터 파생된 개념들을 이용하여 유닉스를 개발하고 PDP-7 상에서 유닉스의 기초적인 운영체제의 '최소한'을 갖추었을 때, 독자적인 프로그래밍 언어를 필요로 하게 되었다. 따라서 그는 Martin Richard가 고안한 BCPL을 기반으로 새로운 언어를 탄생시키고 이를 B라고 불렀다. 이는 타입이 없는 언어였고, 따라서 character-handling mechanism, floating-point arithmetic, pointer 처리 등에서 문제점이 있었고, 또한 컴파일러의 스레드-코드 방식의 느린 속도를 개선해야 할 필요성이 있었다. 따라서 문자 타입의 추가와 컴파일러의 개발로 NB(as in New B)를 만들어 내었다. 이 언어로부터 현재에 쓰이는 C 언어의 여러 기본적인 사항,즉, 여러 타입 구조와, 언어의 이식성 개선, 타입 안전의 개선 등을 이루어내었다. 또한 표준 제정, 즉, 성문화 과정을 거침으로 언어의 약속을 만들어 내었다.

이와 같은 언어의 변화 과정은 일견 타당하고 그럴듯해 보이지만, 사실은 언어로써의 효용에 있어서 많은 취약점을 필연적으로 수반한다. 컴퓨터 구조 분야의 경우에서도, 기존의instruction set architecture에 그때그때 필요에 따라 instruction들을 추가해 가며 발전해 온 프로세서는 기존에 사용되어 왔던 구조와의 호환성을 필요로 하기 때문에 효율성의 손실을 발생시켰다. C 언어 또한 마찬가지이다. 그때그때의 필요에 맞추어 모래성처럼 쌓아 올린 언어는 처음부터 마지막까지 완전하고 일관성 있게 계획된 언어에 비해 많은 논리적 약점을 갖고 있을 것이다. 게다가 타입이 없는 언어로부터 발전했다는 사실은, 현재 C 언어의 고질적인 문제인 타입에 대한 취약성의 이유를 설명해 준다. C언어는 일일이 사용자가 타입을 지정해 주어야 하며, 그로 인해 인간의 생각과 프로그래밍 언어로 기술된 코드 사이의 괴리를 발생시킨다. 또한 C언어는 여러 가지 실행 에러들을 발생시키며, 메모리 관리를 사용자에게 맡긴다는 측면에서 그 안전성의 문제가 있다.

2)ML과 함수형 언어

ML은 논리식에 가까운 언어이다. ML을 사용하면서 느꼈던 것은 ML이 기본 논리학의 증명 체제와 비슷한 형태를 띄고 있고, 따라서 논리학에서 사용되는 증명 형식을 차용하여 프로그램을 수행해 나갈 수 있다는 점이었다. 증명 형식을 차용하여 프로그램을 수행한다는 말에는 언뜻 개념의 불합치가 존재하는 것처럼 보인다. 왜냐하면 '증명은 증명이고, 프로그램 실행은 프로그램 실행일 뿐이다' 라는 것이 이전에 만연했던 인식이었기 때문이다. 그러나 현재에 와서는 이 두 가지 개념이 궁

극적으로 동일한 과정이라는 것이 널리 알려져 있다.

증명과 프로그램이라는, 언뜻 보면 상이하게 들리는 이 두 가지 개념의 합치는 논리학의 발달과 기계 이론의 발달로부터 호워드가 도출해 낸 원리로부터 유래한다. (이 내용은 "Proofs are Programs: 19th Century Logic and 21st Century Computing"로부터 요약하고 발췌한 내용이다.) 현대 논리학의 Frege는 기존의 증명을 기호화하여 증명의 체계를 잡았고, Gentzen은 '가정'으로부터 '결론'을 이끌어내는 추론 과정을 통해 논리적인 증명체계를 설립할 수 있다는 것, 즉, 하위표현식 이론(subformula property)를 증명하여, 상향식 논리 증명을 가능하도록 했다. 또한 계산 이론의 분야에서는 lambda calculus가 각광받았는데, 이것은 기계적으로 계산가능한 모든 함수(프로세스)를 표현할 수 있는 내용으로써, 함수가 함수를 인자로 받을 수 있으며 또한 함수를 리턴할 수 있다는 점에서 획기적인 발상이었다. 여기에 term이 type를 갖도록 하는 부분을 추가한 것이 Type lambda calculus이다. 이 두 분야의 괄목할 만한 발전으로부터 호워드는, 증명 절차를 하위표현식 이론을 이용해 단순화 시켜나가는 과정이, 곧 람다 추상식으로 계산해 나가는 과정과 동일하다는 결론을 이끌어 냈다. 따라서 '증명은 프로그램이다'라는 간단한 문장이 설득력을 갖게 되었다.

그러므로, ML은 위와 같은 내용을 받아들여 인간의 직관적인 논리에 맞추어 작성된 코드가 그에 부합하는 계산식으로 전환되게끔 하는 깔끔하고도 효율적인 언어이다. 평상시에 사용하는 의미와 사고 과정을 논리 표현식에 맞추어 사용자가 기술하면, 논리적으로 증명하는 과정과 기계적으로 계산하는 과정은 동일하므로, 코드로부터 필연적으로 기계에 의해 수행될 수 있는 적합한 계산 과정이 만들어진다. 이는 ML이 C언어에서 발생했던 많은 Segmentation fault로부터 자유로워진 이유를 설명하는 하나의 이유가 되기도 한다. ML에서, 한 번 컴파일러를 통과한 코드는 왠만해서는 수행 중에 에러를 발생시키지 않는다. 이것은 프로그래밍 효율 측면에서 획기적인 일이다. 인력의 자원이 한정되어 있는 상태에서, 프로그래밍에 드는 노고가 줄어드는 것은 크나큰 수확이다. 또한, ML로 코드를 작성할 때, 사용자는 기계의 방식으로 사고하지 않아도 된다. 사용자는 직관적으로 의미를 기술하므로, 좀 더 인간의 사고 체계에 가깝고 자연스러운 프로그래밍을 가능하도록 한다. 일례로, 대부분의 프로그래머는 타인이 C로 작성한 코드를 알아보기 힘들어한다. 이것은 엄격한 Abstraction의 필요를 낳았고, 공동 작업의 한계를 설정했다. 그러나 ML의 경우, 인간의 언어에 가까운 자연스러운 언어로 프로그래밍한 코드는 사용자간의 이해도를 높이고 더불어 공동 작업의 범위를 확장한다. 패턴 매칭(pattern matching)같은 것이 직관적이고 자연스러운 프로그래밍의 예시이다. 사용자가 굳이 패턴을 나누어 타입을 지정해 줄 필요 없이, 등장할 수 있는 패턴의 예들을 '암시'해주는 것만으로 ML은 타입을 찾아내고 자동적으로 지정해 준다.

3. 미래의 프로그래밍 언어

이쯤에서 원래의 논의로 돌아가 보자. 어떤 언어가 '발전가능한 언어인가', 즉, 어떤 언어가 좋은 언어가 될 수 있고, 어떤 언어가 효율적인 언어가 될 수 있는가? 그것을 알기 위해서는 현재 프로그래밍 언어가 어떠한 측면에서 한계를 안고 있고, 또한 어떤 측면에서 개선가능한가를 살펴보아야 한다. 좋은 언어, 효율적인 언어의 기준은 무엇인가? 그 기준은 여러 가지가 될 수 있다. 강력한 제어 기능 역시 좋은 언어를 구별하는 중요한 기준이 될 수 있다. 그러나 예를 들어 프로세서를 직접 통제하는 어셈블리어의 경우 강력하면서도 고급 언어로 짜여지는 모든 프로그램을 구현할 수 있지만 좋은 언어라고 보기에는 무리가 있다. 어셈블리어를 이용해 우리가 일반적인 고급 언어로 구성하는 프로그램들을 짜기 위해서는 엄청난 시간이 걸릴 것이다. 또한 이식성 역시 제로이다. 그러나 자원에 있어서 굉장한 효율을 보인다는 장점 또한 있다.

그러므로 좋은 언어, 효율적인 언어라는 것은 절대적인 기준에 의지한다기보다는 상황과 목적에 맞추어 그때그때 달라지게 될 것이다. 그러므로 좋은 언어를 논하기 위해서는 특정 상황에 있어서 풍부한 자원과 상대적으로 부족한 자원에 대한 파악이 필수 불가결하다.

그러므로, 이 레포트에서는 '좋은 언어'를 구분짓는 기준의 하나로, 인간의 수고를 줄여 주는 언어를 이야기하고자 한다. 컴퓨터 산업의 병목 자원은 점차 기계 자원으로부터 인적 자원으로 이동하고 있다. 기계의 자원은 하드웨어의 발달로 인해 지수 함수로 증가하고 있다. 엄밀하고 낭비가 없었던 어셈블리어로부터, 인간 친화적인 고급 언어로 프로그래밍 언어의 메가트렌드가 이동한 것은 언어 자체의 개발 역사로부터도 기인하는 것이지만, 기계 자원이 무한해지고 인간의 노력과 시간의 자원이 소프트웨어 개발에서 주요한 Bottleneck이 되어가고 있다는 사실과 무관하지 않다. 그러므로, 프로그래밍 언어는 첫째로 인간이 작성하기 편하고, 둘째로 인간의 시행 착오를 줄여 주는 방향으로 발전해야 한다. 따라서 직관적인 언어가 조금 더 좋은 언어로써 기능하게 될 것이다. 또한, 정확한 프로그램의 수행을 가능하게 하는 언어, 즉, 기계적인 버그 처리의 범위가 넓은 언어가 '좋은 언어'의 이상에 가까워질 확률이 시간이 흐름과 동시에 점차 높아질 것이다.

따라서 위의 사항에 적합한 언어가 '발전가능한 언어'이고 동시에 '좋은 언어'가 될 가능성이 높은 언어인데, 작성이 쉬우며, 또한 프로그램의 정확한 수행을 위한 여러 가지 자동적인 버그 처리를 지원해 주는 쪽으로 발전할 수 있는 언어는 C와 같은 부류의 언어보다는 ML과 같은 언어이다. 그 이유는 다음과 같다.

첫째로, 앞서 언어를 분석하며 ML로 기술된 언어는 인간의 생각 체계를 그대로 반영하는 논리적 기술 방식을 차용하므로 직관적이고 작성하기 쉽다는 사실을 언급했었다.

둘째로, 일반적으로 인간이 프로그램을 완전하게 하는 과정은 크게 세 단계이다. 이는 우리가 프로그래밍을 할 때 겪는 시행 착오가 크게 세 가지임에서 착안한 것이다. 프로그램을 작성하고 나면 우선 syntax error를 수정해야 한다. 이는 C, ML에

서 컴파일러가 제공해 주는 기능이다. 그리고 나면 실행 에러를 수정해야 한다. 메모리의 잘못된 접근, 혹은 잘못된 타입의 연산과 같은 실행 에러를 수행하면 프로그램은 비로소 오류 없이 돌아간다. 마지막 과정은 원하는 결과로부터 다른 결과를 얻었을 때 프로그램을 수정하는 과정이다.

이중 첫번째 과정은 이미 기계에 의한 처리가 지원되고 있는 사항이고, 세번째 과정은 아예 새로운 개념이 필요한 사항이다. 왜냐하면 프로그램이 문제없이 수행되는 상황에서, 기계가 인간이 의도한 바를 파악하는 것은 불가능하기 때문이다. 따라서 버그 수정의 성패는 두 번째 과정에서 갈리게 된다. ML의 경우, 두 번째 실행 에러의 사항 중 타입 에러에 관한 수정은 이미 제공해 주고 있다(반면 C는 제공해 주고 있지 않다). 또한 실행 에러 중 타입 에러가 아닌 다른 에러들을 검증해 주는 기술 또한 C에서보다 ML에서 더욱 쉽게 제공해줄 수 있을 것으로 생각된다. 그러므로 안정성 측면에서는 ML이 C보다 더욱 밝은 미래를 가지고 있다고 볼 수 있을 것이다.

이외에도, 교수님의 번역본 문서인 "계산이란 무엇인가에 대한 아이디어들"에 수록된 문제들 또한 미래에 ML이 어째서 C에 비해 유리한 위치를 선점할 수 있는지를 설명해 준다. 논리가 결합된 언어로써의 ML은 "정의"와 "증명"에 있어서 C보다 훨씬 엄밀한 토대를 갖고 있다. 즉, 논리적, 수학적으로 중요한 화두가 될 여러 문제를 해결하는 데 있어서 C보다 강력한 도구를 갖고 있으므로, 후에 이 문제가 적용될 여러 가지 분야에서 ML은 더욱 효율적으로 기능할 것이다.

4. 나가면서

에세이를 쓰기 위해 참조한 수많은 개인적인 의견들(인터넷 토론 혹은 인터넷에 오픈된 에세이들)을 열람하며 가장 기억에 남았던 것은 '어떤 언어가 우수하다고 말할 수 없다'라는 것이었다. 고등학교 때 대기를 분석하는 연구를 하기 위해 서울대 자연대 교수님과 진행했던 연구에서는 데이터를 분석하는 데에 포트란을 이용했었다. 그때까지만 해도 C/C++등의 언어에 비해 포트란이 미개한 언어라고 생각했지만, 이번에 에세이를 쓰기 위해 언어에 대한 자료들을 검색하면서 단순한 수치 계산의 측면에서는 포트란이 독보적인 위치를 차지한다는 것을 알게 되었다.

이러한 특수성은 모든 프로그래밍 언어에 적용된다. 어떤 언어가 어떤 언어보다 절대적으로 '좋다'고 말할 수 없는 것이다. 단지 상황과 기준이 존재하고, 특정한 목적에 따라 적합한 언어가 있다는 것은 다시금 기술하는 것이 소모적으로 느껴질 만큼 자명한 사실이다. 그럼에도 불구하고 우리가 C를 현재 좋은 언어라고 흔히 말하는 이유는 오늘날의 '좋은 언어'를 판명하는 여러 기준에서 C가 확실히 타 언어의 평균적인 수준을 상회하기 때문일 것이다.

그럼에도 불구하고 이번 보고서에서 ML의 장점을 강조한 것은, 보고서의 초점이 '발전가능한 언어'에 맞추어져 있었기 때문이었다. 즉, ML은 앞으로 변화할 프로그래밍 환경을 기준으로 해서, 앞으로 등장할 기준들에 의해 '좋은 언어'로 판명될 기준

이 높기 때문이다. 그러므로 ML을 배우는 것은 어쩌면 시간 낭비가 아니었을 것이고, 어쩌면 몇십 년 뒤 나와 나의 세대가 소프트웨어 산업에 종사하게 될 경우, 기초적이고 중요한 경험으로 작용할 수 있을 것이다.