<2010-fall/4190.310 Programming Language>
Essay 과제

Programming + Language
-언어는 의미를 담는 도구, 프로그래밍은 생각을 담는 과정

컴퓨터공학부
2008-11553
구원준

1. 서론

우리는 프로그래밍을 한다. 컴퓨터 공학부 학생으로서 많은 시간을 프로그래밍을 하면서 보낸다. 그런데 지금껏 과연 프로그래밍이 무엇을 하는 과정인지 진지하게 고민해 본적이 없는 것 같다. ‘ 프로그래밍’ = ‘ 프로그램을 만드는 일’ 이라는 동어 반복을 지금까지 그 정의로 생각하고 있었으니 말이다. 과연 프로그래밍은 무엇을 하는 것인가? 그리고 프로그래밍 할 때 사용하는 언어(Language)란 무엇이며 어떤 것을 골라야 하는가?
언어란 무엇인가? 일반적인 의미에서 언어는 의미 전달을 위해서 사용하는 매개체이다. 아마존 숲속에 사는 소규모 부족에서부터 현대식 아파트에 거주하는 사람에 이르기까지, 또 원시인에서부터 현대인에 이르기 까지 인간은 계속해서 언어를 사용해 왔고 인간의 진화와 더불어 사용하는 언어 역시 좀 더 복잡해지고 정교해졌다. 그 결과 나타난 인간 언어의 복잡성은 동물과 인간을 구분해주는 큰 기준이 된다. 동물은 울음소리를 이용하여 의사소통을 하지만 인간만큼 정교하게 하지 못하며 글자를 만들어 만나지 못하는 후손과 의미를 통하는 일은 거의 하지 못한다. 즉, 인간은 언어를 통해 나만 볼 수 있는 ‘ 나의 마음’ 을 외부에 노출시킨다.
프로그래밍이란 무엇인가? 프로그램을 만드는 것이다. 그렇다면 프로그램은 무엇인가? 프로그램은 컴퓨터라는 기계 위에서 동작하도록 만들어진 일련의 명령문들을 말한다. 결국 프로그래밍은 컴퓨터가 원하는 일을 하도록 컴퓨터에게 나의 의사를 전달하는 일이다.
그렇다면 프로그래밍 언어는 무엇인가? 위의 두 결론을 종합해보면, 기계(컴퓨터)에게 나의 의사를 전달하기 위해 사용하는 도구를 프로그래밍 언어라고 부른다. 생각해보면 맨 처음 기계어로 프로그래밍 할 때, 그것을 'Language'라고 이름붙이기가 쉽지 않았을 거 같다. 왜냐하면 그것이 실제 우리가 사용하는 언어와는 매우 다른 것이기 때문에 새로운 단어를 사용하거나 다른 단어를 붙였을 수도 있었기 때문이다. 그럼에도 불구하고 Language라는 말을 붙여 사용하고 있고, 그 선택은 꽤 합리적인 것 같다.
언어라고 부를 수 있다는 것의 놀라운 점은 우리가 사용하는 C, JAVA, OCaml은 그 자체로 언어이기 때문에 언어학에서 말하는 언어와 동일한 성질을 갖고 있다는 것이다. 예를 들어 나는 모국어가 한국어이기 때문에 생각할 때도 마음 속으로 한국어를 계속 말을 하고 있고, 꿈에서도 한국어로 말한다. 다시 말해 사고의 틀이 ‘ 한국어’ 로 제한되는 것이다. 내가 나의 두뇌를 이용해 어떤 놀라운 사실을 밝혀냈다면 그것은 분명 한국어를 통해서 만들어진 것임이 분명하다.
프로그래밍 언어도 마찬가지이다. 우리가 어떤 언어를 사용하느냐는 우리가 프로그램을 어떤 식으로 작성하느냐와 밀접한 관련이 있게 된다. 그렇다면 어떤 프로그래밍 언어가 우리의 생각을 담는데 더 적합한 성질을 갖고 있을까? 우선 만들어진지 40년이 넘도록 여전히 널리 사용되는 언어인 C의 성질을 그 발전과정을 통해 알아보도록 하자.

2. 본론

2-1. The Development of the C Language

이 논문은 BCPL, B 언어를 거쳐서 만들어진 C언어의 발전과정에 대한 것이다. 또한 C의 발전과정은 Unix의 발전과정과 그 궤를 같이하고 있다. 맨 처음 Library, Loader, Linker가 필요없이 output file만으로 바로 실행할 수 있게 하는 PDP-7 Assembler를 만들고, Unix가 PDP-7위에서 돌아가게 되자 Thomson은 System Programming을 위한 새로운 High-level Language를 필요로 하게 되었는데, 이것은 BCPL을 PDP-7에 맞도록(8K bytes Memory) 우겨넣은 B언어를 탄생시켰다.
그렇기 때문이 BCPL, B, C언어는 모두 닮아있다. 세 언어는 모두 System Programming을 위한 언어이며, 기계와 근접한 언어이다. 기계가 사용하는 data type, operation들, input-output 라이브러리 루틴, 운영체제와의 상호작용에 그 근간을 두고 있다. 다시 말해 기계를 어떻게 작동시킬 것인가에 초점이 맞춰져 있지 사람이 사용하는데 초점이 맞춰진 언어는 아닌 것이다.
그렇기 때문에 언어의 발전은 기계가 새롭게 도입되거나 사용자가 불편함을 느낄 때 마다 이루어지게 된다. BCPL은 서로 다른 파일에 존재하는 global 변수나 함수를 호출하기 위해서 global vector라는 것을 Programmer가 직접 세팅해주어야 하는 문제점이 있었다. 즉, 외부 변수를 얻기 위해서는 사용자가 이 파일과 저 파일 사이의 상대적 위치를 매번 계산하여 적어야 했었다. 이러한 문제점을 해결하기 위해 B언어에서는 컴파일에 필요한 파일들을 컴파일러가 한꺼번에 불러와 계산하는 방식으로 발전했고, 나중에는(후반기 B언어와 C언어에서는) Linker가 이 역할을 맡게 되었다.
B언어에서는 PDP-11이 도입되면서 문제가 불거지게 되었다. 첫 번째 문제는 Character Handling 방식이 그 전과는 달라져야 했다. B언어는 BCPL과 마찬가지로 워드 단위의 어드레싱을 하고 있고, type이 없이 메모리 안에 놓인 값 자체를 중시하였다. 하지만 PDP-11에서 character handling을 하기 위해서는 byte단위의 addressing이 필요했기 때문에 워드 하나를 읽어와서 한 byte를 수정하고 다시 저장하는 방식은 어울리지 않았다. 두 번째 문제는 B언어는 Floating-point 연산을 지원하고 있지 않았다. PDP-11에서 Floating-point 연산을 하기 위해서는 하나의 메모리 Cell이 아닌 여러 Memory Cell을 하나의 data로 바라보는 것이 필요했다. 하지만 type이 없고 Cell 단위의 addressing을 하는 B언어에서는 이와 같은 지원이 어려웠다. 세 번째는 Pointer의 문제였다. Word단위의 Addressing에서 배열을 참조할 때, 배열의 다음 위치를 계산하는 방법이 그 안에 담긴 data에 따라(int인지, char인지, float인지) 달라져야만 했지만 이것을 지원하는 것은 어려운 일이었다. 이와 같은 문제점을 해결하기 위해 Type이 등장하였고 이를 위해 Compiler를 새롭게 쓰면서 NB(New B)가 등장하였다. 그리고 NB언어에 몇 가지 문법적인 수정이 더해지자 C언어라는 새로운 이름을 붙이게 되었다. 그 이후 개발의 필요성에 의해, &(|) 연산자와 구분되는 &&(||) 연산자가 등장하고, preprocessor가 등장하는 등의 몇 가지 발전이 있게 된다.
이렇게 만들어진 C언어는 Unix System의 높은 Portability, 다양한 Type 지원 등의 장점을 통해 널리 퍼지게 되고 대부분의 System에서 C언어를 지원하는 Compiler가 우후죽순 등장하게 된다. 이 과정에서 자연스럽게 Standard에 대한 필요성이 생겨났고 C Standard와 reference compiler가 등장하게 된다. 이 표준화 작업은 매우 성공적이어서 C언어의 근간을 해치지 않았고 그 이후 등장한 C Standard Library는 C언어가 성공하는데 큰 도움을 주게 된다. 그 밖에도 심플하고 작은 규모의 언어라는 점, 기계와 가깝지만 인간이 사용하기에 충분히 Abstract된 언어라는 점, 다양한 Compiler가 존재함에도 대부분이 서로 호환 가능하다는 점 등이 성공의 열쇠가 될 수 있었다.
하지만 C언어는 위에서 언급했듯이 B언어의 관성에 따라 컴파일러가 type에 대해 관대하다는 점(pointer type이 int를 담을 수도 있고, 다른 structure type을 가리킬 수도 있는 등), pointer type이 가져오는 다양한 type 표현 방법 (int의 pointer의 pointer, int를 리턴하는 함수의 pointer, int를 리턴하는 함수의 pointer를 가리키는 pointer 등)의 애매모호함, 동적 메모리 할당과 함께 pointer를 사용할 때 나타날 수 있는 위험함, external / internal 밖에 지원하지 않아 대규모 프로젝트에 적합하지 않다는 문제점이

존재한다.

이 글을 읽으면서 든 생각은 C언어의 주먹구구식 발전과정에도 불구하고 매우 대중적인 언어로 자리매김하고 있다는 사실이다. 수학적인 배경에 의해 철저히 계획되고 수학적인 특성을 살린 ML과 같은 언어와 달리 기계의 발전과정을 따라 덧붙여진 언어이지만 그것을 사람들이 잘 사용하고 있다는 점은 C언어가 시대를 잘 타고났다는 느낌을 준다. 사실 프로그래밍을 처음 접한 사람이 "Hello World"를 출력하는데 사용하는 C언어의 문법은 매우 간결하고 직관적이다. 그것은 직관적이며 당연해 보인다. 내가 오직 C언어만을 사용할 때에도 정말 간결하고 편하다고 생각했다. 하지만 그것은 단지 '기계를 다룰 때'에만 좋은 접근 방법이다. 기계를 다루는 것이 아니라 프로그램을 통해 의미를 만들어 내고자 할 때 여러 난관들에 부딪히게 된다. Dynamic Memory Allocation과 Memory Free, Debugging, 읽는 것이 불가능한 Code, Code 수정의 어려움 등은 C언어를 중급 이상으로 다룬 사람들에게는 너무나도 당연한 문제들이다. 그런데도 C언어를 사용하는 이유는 무엇일까? 정말로 C언어와 같은 절차적 언어는 배우고 사용하는데 직관적이며 사용하기 쉬울까?
또한, 필요성에 의해서 자연적으로 생기는 언어는 언어가 지향하는 목표점이 달라지게 되고, 그 결과 언어는 점점 더 복잡해지고 제어가 불가능하게 되는 점을 볼 수 있었다. 예를 들면, BCPL은 처음에 Portability를 고려하지 않고 design된 언어이지만 나중에 C언어에서는 Portability가 중요한 목표로 부각 되었는 것과 C언어에서는 불필요한 메모리 주소 계산을 개선하기 위해 Type과 Pointer를 도입하여 메모리 안에 들어있는 정보의 의미(Seamantic,Type)를 따지게 되었지만 이 과정에서 대부분의 Programmer들이 BCPL이나 B언어에서 사용했던 typeless를 선호하는 관성을 보여주었기 때문에 강한 Type System으로 발전하지는 못하였다는 것이 있다. 이처럼 계획되지 않은 발전과정을 따라가게 되면 원래 근간이 되는 언어에 끌려가게 되는 문제점이 발생한다는 점이 눈에 띄었고, 그렇기 때문에 우리는 언어의 근간을 쉽게 변하지 않는 '수학'에서 찾아야 한다는 것을 알게 되었다. 그러면 다음 글에서 '수학'을 기반으로 ML이 어떻게 만들어 졌는지를 살펴보자.

2-2.An Interview with Robin Milner

이 글은 ML이라는 언어를 디자인한 Robin Milner와의 인터뷰를 담고 있다. 여러 가지 주제에 대해 얘기하면서 ML이 어떤 배경에서 만들어졌는지, Computer Science가 어떤 의미의 과학인지 등을 엿볼 수 있는 기회가 되었다. 자세한 내용을 살펴보면 다음과 같다. Robin Milner는 AI 분야와 Programming이란 무엇인가? 라는 두 가지 분야에 관심이 있었다. 특히, 1960년대 AI 분야의 화두는 과연 자동으로 theorem을 증명해주는 기계를 만드는 것이었고, Programming의 Semantic(의미)를 밝히는 것에서는 컴퓨터 프로그램이 동작하는 방식을 검증하는 것이었다. 그런데 사실 상 프로그래밍이 잘 동작하는 것을 증명하는 것이 theorem-proving program과 일치하는 것이었다. 그렇지만 theorem-proving program에서는 Formal하게 표현된 수학적 논리를 사용하지만 지금까지 사용되어 온 Program은 Formal하지 않았다. 그래서 theorem-proving program을 통해 program을 실행하기 전에 verify할 수 있는 언어, language를 기계가 검증하는 언어인 Meta-Language(ML)이 등장하게 되었다. 다시 말하면 수학적인 증명을 돕기 위한 도구로서 언어가 디자인되고 만들어진 것이다. 이렇게 ML을 뒷받침하고 있는 수학적인 배경 덕택에 다른 언어에서는 보기 힘든 완전히 형식적인 언어에 대한 정의 - 게다가, 매우 간결한 - 를 갖을 수 있게 되었다.
인터뷰에 후반부에는 병행성에 대해, 객체 지향 프로그래밍에 대해 이야기 하고 있는데, 그 핵심은 언어에 관한 것과 일치한다. Robin Milner는 컴퓨터 과학의 핵심을 세 가지로 요약하고 있는데, 첫 번째는 어떤 모델을 디자인하는 것이고, 두 번째는 그것을 실제 하드웨어에서 구현하는 것, 세 번째는 그 자체로 형식적인(Formal) 정의를 그 자체가

갖고 있어야 한다는 것이다. 이러한 성질 중 실제로 구현해보는 부분은 컴퓨터 과학이 왜 실험 과학으로 분류되어야 하는 지에 대한 근거를 마련해준다고 말한다. 어떤 이론을 수학적인 배경하에 세우고 이를 구현을 통해 검증하는 과정은 실험 과학과 동일하기 때문이다. 마지막으로 학생들에게 이론을 세우는 것과 그것을 구현을 통해 검증하는 2가지 측면 모두를 함께 병행할 것을 당부하고 있다.

이 인터뷰에는 흥미로운 부분과 공감할 수 있는 부분이 많았다. 특히, 컴퓨터 공학이 실험 과학이라는 부분에서는 저절로 고개가 끄덕여 졌다. 특히 실험 과학에서 중요한 부분은 '실험'이 아니라 무엇을 실험할 것인지 정하는 부분인 '이론'이 뒷받침 되어야 한다고 말한 부분에서는 내가 꽤 오랫동안 고민해 왔던 부분을 시원하게 긁어주는 느낌이었다. 그동안 나는 컴퓨터 공학을 전공하면서 다른 전공자들이 나보다 더 실제 현실에 프로그래밍을 잘 접목하는 것 같아 불안해 왔다. 기계공학을 전공하는 한 선배가 C++을 이용하여 이미지를 처리하고 그 결과를 바탕으로 기계를 만들고, 전기 공학을 전공하는 다른 선배 역시 C++을 이용하여 제품을 만드는 것을 보면서 컴퓨터 공학을 전공하는 나의 위치는 뭔가를 고민하고 있었다. Robert Milner는 이에 대해 'How could it be that this doesn't have a theory?'라고 물으면서 컴퓨터 공학 뒤에 숨은 수학적 배경이 실제 응용 프로그램을 만드는 것보다 중요함을 일깨워 주었다. 이론이 있고 실험이 뒤따르는 것이다. ML이 Meta Language의 약자라는 사실도 알게 되었다. 그리고 Meta라는 단어에는 theorem-proving이라는 수학적 배경이 숨어 있다는 사실을 알게 되었고, 지금까지 몰랐던 사실 - 왜 강한 type checking이 필요한가? 강한 type checking system이 의미하는 것은 무엇인가? -에 대해 알게 되었다. theorem-proving 방법에 대한 내용은 다음 글에서 더 자세히 살펴볼 수 있었다.

2-3. Proofs are Programs: 19th Century Logic and 21st Century Computing

이 글은 증명이라는 개념이 어떻게 프로그램과 같은 것인지를 설명하고 있는 글이다. 우선 Gentzen's natural deduction이라는 정규식 증명의 방법을 설명하고 Church's Lambda Calculus라는 정규식 프로그램을 설명하고 있다. 그 다음으로 여기서 파생된 Typed Lambda Calculs가 Gentzen's natural deduction과 본질적으로 같은 것임을 보여주고 이를 이용한 ML이라는 언어에 대해 설명하고 있다.
Gentzen's natural deduction은 번역하면 자연 추론이다. 다시 말해, 어떤 가정들이 존재할 때 자연스럽게 나오는 결론들을 추론해 낼 수 있는 방법을 말한다. Frege는 modus pones라 불리는 추론 규칙(A->B이고, A이면 B이다.)과 더불어 3가지 공리를 이용하면 모든 추론 가능한 정리들을 증명할 수 있게 됨을 알았다. 하지만 Gentzen은 더 나아가 3가지 공리보다 더 자연스러운 implication logic을 원했다. 이를 위해 Frege의 판단 규칙에 가정을 포함시켜 일반화 시켰다. 이를 통해 추론 규칙을 만들었는데, 그 추론 규칙에는 structure rule과 logical rule이 있고 logical rule은 다시 introduction rule - deduction 결과 connective가 생기는 -과 elimination rule - deduction 결과 connective가 사라지는 - 으로 나뉜다. 이러한 규칙들은 상당히 대칭적이어서 3가지 공리를 이용한 추론 규칙보다 더 적용하기 쉽고, 이러한 대칭성은 subformula property를 증명할 수 있게 해주었다. 즉, '감마'라는 가정들로부터 나온 A를 증명하는 과정은 감마와 A에 있는 부분들(subformula)로 만들 수 있다는 것이다.
Church's Lambda Calculus는 모든 계산은 치환 과정과 똑같다는 사실을 말해주고 있다. 다른 말로하면, 컴퓨터로 계산 가능하고 표현할 수 있는 모든 함수는 람다 기호를 이용하여 쓸 수 있다는 것이다. 함수는 람다식이며 계산은 람다를 값으로 치환하는 과정이다. Church는 이를 증명하였다. 그리고 이어서 Typed Lambda Calculus를 발표한다. Typed Labmda Calculus가 표현할 수 있는 모든 프로그램은 이전의 로직들이 갖고 있는 Halting Problem을 피할 수 있다는 큰 장점이 있었다. 다시 말해 프로그래밍 될 수 있는 모든 함수들 중에 Halting Problem을 해결할 수 있는 적절한 - 프로그램의

표현력을 제한하지 않는- Subset을 추려낸 것이다. 이전까지의 Lambda Calculus가 계속해서 치환만 했다면, Typed Lambda Calculus는 타입이 맞을 때에만 치환을 하여 계산하도록 한 것이다.
그런데 Typed Lambda Calculus를 Gentzen이 사용한 표시법을 사용하면 Gentzen이 제시한 natural deduction rule과 완전히 일치하는 것을 볼 수 있다. Curry, Howard는 이를 발견하였고 결국 증명하는 것과 프로그램이 계산되는 것은 동일한 것이라는 사실이 밝혀졌다.
함수는 다른 함수를 인자로 받아 함수를 반환할 수 있다는 사실이 알려지면서 이를 이용한 Programming Language들이 등장하게 되었다. 그 중의 하나가 ML이다. ML은 강한 타입 system을 제공하여 프로그램이 동작하기 전에 이 프로그램이 계산될 수 있는지를 미리 판단해 준다. Typed Lambda Calculus를 구현해 놓은 것임과 동시에 Natural Reduction을 통해 프로그램의 동작을 예측하는 것이기도 하다.

이 글을 통해 지금까지 수업시간에 배운 내용이 깔끔하게 정리되었다. 그동안 꽤 많이 들었었던 '증명과정' = '실행과정'이 어째서 그러한 것인지 수학적인 배경을 알 수 있었고 어떻게 강한 Type System을 사용하는 ML이 프로그램을 실행시키기도 전에 미리 프로그램의 동작을 추론해 낼 수 있는지를 알게 되었다.
글의 결론에는 이를 바탕으로 한 응용에 대해 소개하고 있는데, 'Trust is essential in computing.'이라는 말이 마음에 와 닿았다. 내가 C, JAVA를 이용하여 숙제를 하면서 얼마나 가슴 조렸던가? 테스트 셋을 입력해보며 안 돌아가면 좌절하고 밤새 디버깅을 하는 일이 다반사였다. 예측할 수 없는 input들이 난무하는 실제 세계에서 과연 내 프로그램이 정상적으로 동작한다고 확신할 수 있는 사람이 얼마나 될까? 특히 실제 판매되고 있는 프로그램을 만드는 사람의 경우 그러한 위험을 손으로 직접 잡아내기란 너무나도 어려운 일이다. 하지만 강한 Type System이 제공하는 Proving은 대다수의 프로그래머들에게 굉장히 소중한 것이 될 수 있다. 이러한 강점을 실제 예로 보여준 것이 다음에 나오는 강연이었다.

2-4. Yaron Minsky's talk at CMU, 2009 : Wall Street에서 전하는 프로그래밍 언어 이야기

카네기 멜론 대학에서 강의한 내용이다. Yaron Minsky는 Jane Street이라는 trading 회사에서 trading software개발자로 일하고 있는 사람인데, 보통 산업체에서 사용하는 언어인 C++, C#, JAVA 같은 언어를 사용하지 않고 Academic한 용도로 많이 사용되는 OCaml을 이용하여 상용 프로그램을 개발하였다. 그 이유를 강의에서 다음과 같이 밝히고 있다.
우선 Trading Firm에서 사용하는 Software는 다른 분야에서 사용하는 언어를 선택할 직접적인 이유가 전혀 없다. 이 분야는 컴퓨터 공학 분야와 연관이 있는 항목이 전혀 없으며 그래서 언어를 자유롭게 고를 수 있는 환경이다. 그러므로 회사에서 사용할 Software가 가져야할 특징들을 잘 분석하여 적합한 언어를 선정하는 것이 중요했다.
회사는 시장의 4가지 사람들(Investor, Speculator, Market Maker, Arbitrager) 중 Market Maker인 동시에 Arbitrager이다. Arbitrager는 차액거래를 하는 사람을 말하는데, 동일한 상품이 다른 값에 팔리고 있다면 이를 이용하여 돈을 버는 사람들을 말한다. 하지만 모든 거래과정이 전산화되어있는 현대 사회에서 동일한 상품이 다른 값에 팔리는 현상은 매우 짧은 시간 동안만 지속된다(Short-time Frame을 갖는 Business이다). 결국, 차익거래를 위해서는 자료를 분석하여 매우 빠른 속도로 처리해야만 한다(Performance). 동시에, 틀리면 안되며(Correctness) 주위 환경의 변화(법의 제정 등)에 빠르게 Software를 수정할 수 있어야 한다(Agility).
이러한 3가지 특징(Performance, Correctness, Agility)를 만족하는 언어가 OCaml이다. Jane Street에서 맨 처음 일을 시작할 때 C#과 같은 언어를 사용해 보려했지만 실패했다.

하지만 OCaml로는 성공할 수 있었다. Correctness 측면에서 보면, OCaml의 큰 특징은 읽기 쉽다는 것이다. 회사에선 Software를 검증할 때 코드의 모든 라인을 체크하고 체크하게 된다. 하지만 C#이나 SQL과 같은 언어는 읽기 어려울 뿐만 아니라 어떤 함수가 무엇을 Call하는지 따라가기가 매우 힘들다. 하지만 OCaml은 읽기 쉬우며 Common Pattern을 처리하기에 적합한 Functor와 같은 방법을 제공하여 준다. 이는 Common Pattern을 Copy & Paste를 통해 처리하는 것을 막아주고 유지 보수 과정을 쉽게 만들어 준다. 강한 Type System은 Code를 만들고 Type을 Tagging할 때 의도했던대로 작동하게 해주는 매우 좋은 도구이다. 특히 Match statement는 Missed Cases, Reduntant Cases, Impossible Cases를 컴파일러가 경고해 줘서 유지보수가 매우 쉽도록 도와준다. 이는 위에서 말한 Agility 특징을 얻을 수 있게 해주는 것이다. 즉, 어떤 Code를 우리가 수정했을때 그것이 미치는 영향을 컴파일러가 미리 알려준다는 것이다.
Performance는 어떤 trade-off관계에 있다. 표현력(자연스러운 코드)이 좋은 언어를 사용하면 성능이 나빠질 수 있고, 성능이 좋은 언어를 사용하면 표현력이 제한될 수 있다. OCaml은 그 중간에서 두 성질 모두를 어느 정도 만족시킬 수 있는 언어이다. Optimization Compiler가 아닌데도 불구하고 컴파일 된 결과물은 거의 Optimize된 코드가 된다.
하지만 OCaml을 사용할 때의 단점은 User Interface를 위한 라이브러리가 부족하고, 병행처리를 물리적으로 지원하지 않는다는 점, 사용자 그룹이 소규모라 사용할 수 있는 Library가 작다는 점, 대규모의 프로그래밍을 하기에는 부족한 점이 있다는 점이다. 하지만 이러한 단점에도 불구하고 OCaml은 Financial 분야에서 사용하기에 좋은 언어이다.

이 강의를 듣고 나서 든 생각은 위에서 말한 3가지 특징(Performance, Correctness, Agility)가 분명히 Finance 분야에만 적용되는 기준이 아니라는 것이다. 3가지 기준은 어디에나 따라온다. 자동차를 만들 때 들어가는 Software역시 위 3가지 특징을 갖고 있어야하며 우주선을 만들 때에도 아이폰의 App을 만들 때에도 게임을 만들 때에도 적용되는 기준이라는 것이다. OCaml이 위 3가지 특징을 만족한다면 게임을 만들 때도 사용할 수 있지 않을까?
나는 이번 여름방학 게임 회사에서 인턴을 하면서 OCaml을 이용하여 게임을 만들 수 있다면 너무나 편리할텐데 라는 생각이 간절했다. 왜냐하면 그 곳에서 게임 엔진이 제공하는 스크립트 언어를 이용하여 프로그래밍을 해야했는데, 스크립트에서 해야 할 일들이 어떤 맥락에서 이루어지는 것인지를 파악하기 위해 인턴기간의 대부분을 코드의 숲을 방황하며 보냈기 때문이다. 그리고 나서 프로그래밍을 시작했을 때엔, 코드에서 차지하는 대부분이 이 변수의 Class를 체크하는 것(type checking)과 변수가 가질 수 있는 boundary를 체크하는 것이었다. 그 일을 하면서 정말 type이 뭔지 바로바로 알아낼 수 있다면 프로그래밍 하는데 걸리는 시간을 많이 줄일 수도 있겠다 라는 생각이 너무 간절했었다. 그래서 이와 연관된 주제처럼 보이는 '게임 프로그래밍 정글의 원시인들'이란 글도 읽었는데, 2.5차원의 게임 엔진을 OCaml로 실제로 개발하고 성능이 C언어로 짠 엔진과 근접하다는 사실이 왠지 모르게 기쁘게 느껴졌다. 비록 게임 엔진에서의 C++의 위상을 넘보기에는 너무나도 작은 희망인 것처럼 보이지만 누가 알겠는가? Microsoft에서 .Net Framework를 제공하는, Windows에서 제공하는 강력한 Library를 사용할 수 있는, 또한 개발자에게는 너무나도 친숙한 Windows Visual Studio를 사용할 수 있는 F#이라는 언어로 3D엔진을 만들어 성공하는 기업이 생길지도 모르는 일이다. 그리하여 많은 사람들이 거대한 숲에서 방황하지 않게 되는 날을 기대해 본다.

3. 결론

지금까지 프로그래밍 언어 2가지가 등장하게 된 배경과 둘의 차이점 그리고 실제에서 사용되는 ML언어의 활용까지 살펴보았다. 한마디로 요약하자면 기계라는 땅에서부터

자라난 거대한 나무를 C언어라고 부르고 수학이라는 하늘에서부터 내려온 동아줄을 ML이라고 부른다고 요약할 수 있을 것 같다. 그리고 우리는 C언어의 숲에서 동아줄을 찾아 해매는 원시인이라고도 볼 수 있겠다. 과연 우리는 숲속에서 계속해서 살아가게 될 것인가 아니면 동아줄을 잡고 더 위로 올라갈 것인가?
서론에서 프로그램에 대한 간략한 정의를 내리고 프로그래밍 언어가 ‘ 언어’라는 바탕 아래서 얘기를 계속해 왔다. 하지만 프로그램이란 정의하기가 매우 까다로운 것이다. 나는 그 이유가 ‘ 프로그램’이라는 것 자체가 ‘ 사람의 생각’을 담고 있기 때문이라고 생각한다. 그래서 둘의 생각이 같은 것인지 알아내기가 매우 힘든 것과 마찬가지로 프로그램이 같은지 찾아내는 것은 쉽지 않은 것은 당연하다. 마찬가지로 사고의 흐름이 정확히 어떤 방식으로 흘러가는지 알기 어렵듯이 정보가 어떤 식으로 흘러가는지를 파악하기 힘든 것도 당연하다. 사람이 외부 환경과 계속해서 소통하기 때문에 지능을 갖는 것인지 아니면 인간이 혼자 있어도 지능을 갖는 것인지를 파악하기 힘든 것처럼 마찬가지로 프로그램이 그 자체로 존재하는 것인지 아니면 외부와 상호작용 대상 역시 프로그램의 일부로 봐야하는지도 파악하기 힘들다.
하지만 분명한 것은 프로그래밍 언어가 사람의 생각을 담기위해 존재한다는 것이고 점점 더 발전된 형태로 나아가고 있다는 것이다. C라는 언어가 인간이 기계에게 단순히 명령만 내릴 수 있는 일방 통행적 언어였다면 ML이라는 언어는 기계가 인간의 생각 중 미흡한 부분을 체크해 줄 수 있는 쌍방향적인 요소를 포함하고 있다. 그렇기 때문에 처음에는 기계가 말을 걸어오는 것이 낯설 수도 있다. 아마 C언어를 접하고 ML을 접하는 대부분의 사람들이 그랬을 것이고 나도 처음 사용하면서 어색한 점이 많았다. 하지만 곧 ML에 적응하게 되면 문제를 해결하는 방법이 달라지고 그것이 인간의 생각을 표현하기에 더 도움이 되고, 사고를 유사하게 표현할 수 있는 언어라는 생각을 갖게 된다. 아직까지 ML이 대중화 되지 못한 이유는 지금까지 사람들이 가지고 있는 사고의 관성 때문이라고 생각한다. 하지만 이내 사람들은 기계의 도움을 기꺼이 받아 생각을 표현하는 일에 익숙해질 것이다. 이것은 당연한 귀결이다.
하지만 ML과 같은 논리 중심의 언어가 대세가 된다고 하더라도 여전히 프로그래밍 언어는 미개하다. 인간의 생각을 표현하기에는 아직 너무도 많은 제약이 남아있다. 분명 언젠가는 구식의 키보드가 컴퓨터 앞에서 사라지고 나의 생각을 직접적으로 전달하고 그 자리에서 곧장 실행되는 수단이 마련될 것이다. 그러한 세계는 컴퓨터와 인간의 구분이 쉽지 않게 될 것이다. 컴퓨터가 나의 생각을 읽고 생각을 이어 나간다면 그것은 나의 생각인가? 아니면 기계가 생각하고 있는 것인가? 아직은 먼 미래의 이야기가 될 터이지만 그 앞을 가로막고 있는 수 많은 난제들이 프로그래밍 언어의 발전과 함께 해결되어 나갈 것이라는 것은 당연해 보인다.