프로그래밍 언어 두 번째 에세이

- 첨단 언어의 상용화를 기대하며

서울대학교 공과대학 컴퓨터공학부
2008-11640
신 기 정

1. 들어가며

“저 글 쓰는 것 좋아하잖아요, 컴퓨터 프로그래밍이란 것도 뭐 비슷하지 않겠어요? 잘할 수 있을 것 같아요.”

컴퓨터 공학부에 진학하기로 결정한 뒤, 학교 선생님께 이런 말을 했었다. 지금 돌아보면 프로그래밍을 시나 소설을 쓰는 것과 같은 예술 활동 정도로 생각했던 것 같다. 하지만 실제로 프로그래밍을 배우기 시작하자 생각이 바뀌었다.

C라는 기계중심적인 언어를 다루는 것은 시나 소설을 쓰는 것과는 달랐다. 사람을 대상으로 하는 예술은 같은 내용도 어떻게 표현하는지가 중요하지만, 컴퓨터를 대상으로 하는 프로그래밍은 내용이 옳은지 틀린지 만이 중요할 뿐 그 밖의 것은 중요하지 않았다. 실상은 그보다 심각해서, 작성자와 컴퓨터만 겨우 이해할 수 있을 정도의 엉망인 코드가 탄생하기 일쑤였다. 물론 좋은 코드를 작성하기 위해 노력해보지 않은 것은 아니었다. 나름 코드 전체의 구성을 설계하고 구현을 시작하지만, 여기저기서 발생하는 예외적인 상황을 처리하기 위해서는 실행 흐름이 여기저기를 왔다갔다해야했고, 코드는 복잡해질 수밖에 없었다. 친절한 주석으로 이를 만회하려고도 해봤지만 사람의 사고 과정과 프로그램의 흐름 간에는 차이가 커서 대응관계를 찾기 어려웠다. 사고 과정을 담으려던 주석은 단순히 기계적인 과정을 번역하는 것에 그치고는 말았다. 게다가 표현력이 작아서, 간단한 생각도 여러 줄에 걸쳐 작성해야했고, 코드가 길어지니 전체 흐름을 파악하기도 어려웠다. 누더기 같은 소스 코드도 프로그램이라는 포장 안에 감추면 아무도 못 본다는 사실이 다행스럽게 여겨질 정도였다.

하지만 ML을 사용하면서 프로그래밍에 대한 생각이 바뀌기 시작하였다. 마치 수학 문제를 증명하듯, 사고의 흐름을 그대로 코드로 옮길 수 있다. 코드의 흐름이 사고의 흐름과 다르지 않다보니 주석도 많이 필요하지 않았다. 또한 표현력도 커서, 전체 흐름을 파악하면서

작성하는 것이 가능해졌다. 아름답게 프로그래밍 하는 것도 불가능한 것만은 아니었다.

컴퓨터 과학의 다른 분야들처럼, 프로그래밍 언어도 급격한 발전을 이루었고 그 과정에서 여러 유용한 개념들이 등장하였다. 이런 개념들은, 실제 프로그래머에 의해 활용될 수 있도록 ML, Haskell 등 첨단 언어에 충실하게 구현되었다. 위 경험에서 제시한 것들은, 첨단 언어가 갖는 많은 장점 중 일부분에 불과할 정도이다.

하지만 이러한 첨단 언어들이 실제 산업 현장에서 활용되는 경우는 드물다. 현장에서 주로 사용되는 C나 C++, Java는 새로운 기술들이 적용되지 않은 언어들이다. 사실 ML도 횟수로는 30년이 넘은 언어이지만, 그 당시에 등장한 개념들조차 아직 활용되지 못하고 있다. 한 친구가 입사 면접에서 어떤 언어를 다룰 수 있냐는 질문을 받았고, ML이라고 대답했다가 그게 뭐냐는 재 질문을 받았다고 한다. 본인도 자기만 이상한 사람이 될 것 같다는 생각에 자세한 설명은 하지 않았다고 한다. 이것이야말로 산업 현장의 상황을 잘 보여주는 사례라고 생각한다.

“왜 기업은 첨단 언어를 사용하지 않으며, 프로그래머들도 그것을 당연하게 여기고 있는 걸까?”

2. 왜 첨단 언어를 사용하지 않을까?

좋은 언어를 사용해야하는 정당성은 충분하다. 언어는 도구이기 때문이다. 즉 언어는 그 자체가 목적이 아니라, 결과를 위한 수단으로서 가치를 갖는다. 좋은 프로그램을 작성하기 위해서 좋은 프로그래밍 언어를 사용해야 한다는 점에는 반론의 여지가 없다. 하지만 이런 당연한 이치가 현장에서는 적용되지 않고 있다. 심지어 좋은 도구가 저렴하기까지 한데도 말이다.

『Beating for average』[1]와 『Revenge of the nerds』[2]라는 글에서 Paul Graham은 여러 장점에도 불구하고 첨단 언어가 산업 현장에서 사용되지 않는 현상에 대해 두 가지를 이유를 제시하였다.

첫 번째는 프로그래머들의 현재 언어에 만족하려는 성향을 갖고 있다는 것이다. 프로그래머들은 자신이 사용하는 언어가 충분히 괜찮다고 생각한다. 그 이유는 언어는 직접 써보기 전까지는 장점을 제대로 파악할 수 없다는 것에 있다. 프로그래머들은 부족한 언어에서 불편함을 찾는 것에는 능숙하지만, 훌륭한 언어의 장점을 뚫어 보는 데에는 능숙하지 못하다. 이와 관련해서는 과거에 이루어졌던 논쟁들을 참고할 수 있다. 서브루틴을 사용할 것인가 사용하지 않을 것인가, 추상화된 언어를 사용할 것인가(여기서 추상화된 언어란 지금은 비교적 기계적인 언어로 분류되는 C같은 언어를 의미한다.) 어셈블리 언어를 사용할 것인가 While문과 Goto문 중 어떤 것을 사용할 것인가. 지금은 이런 것들을 논쟁거리로 생각한다는 것 자체가 우습다. 당연히 전자가 더 우수하기 때문이고, 대부분 사람들이 이에 동의한다. 후자를 주장했던 사람들은 단순히 후자가 주는 익숙함에 빠져서 올바른 판단을 하지 못했던 것이다. 몇 년 뒤에 현재를 돌아보면, 첨단 언어를 사용할지 안 할지의 논쟁도 이와 다르지 않을 것이다. 프로그래머들의 언어에 대한 이런 맹목적인 사랑에 대해 혹자는 ‘프로그래밍 언어는 반은 기술이고 반은 종교’라고 표현하기도 하였다.

Paul Graham이 제시한 두 번째 이유는, 산업 내에서 결정권을 가진 사람들의 위험을 기피하려는 태도이다. 사용 언어에 대한 결정권을 가진 책임자들은 주로 프로그래밍 언어에 대해 잘 알지 못한다. 그들은 표준, 엄밀히는 다수가 선택하는 것을 쫓아간다. 특수한 선택

을 해서 성공할 경우의 보수는 미미하다. 그 것을 자기 덕분이라고 인정받기도 어렵거니와, 그렇다고 해도 월급이 약간 올라가는 정도에 불과할 것이다. 하지만 특수한 선택을 했다가 실패할 경우에는 모든 책임을 져야하고 이는 큰 대가로 돌아올 것이다. 즉 그들은 실패할 경우에 대비한 변명거리를 준비하기 위해서 무난한 선택을 하는 것이다. Paul Graham에 의하면 책임자들은 흔히 '모든 언어의 본질은 동일하다'고 이야기 한다고 한다. 이는 모든 언어가 튜링-완전하다는 의미일 텐데, 이것만으로는 그들의 선택을 정당화하지 못한다. 언어는 그 자체가 목적이 아닌 도구이다. 빠른 시간 내에 오류 없이 프로그램을 만들 수 있는지가 언어의 가치를 결정하고, 그렇기 때문에 첨단 언어는 가치가 있는 것이다. 이는 그들이 튜링-완전한 기계어나 어셈블리 대신, C나 Java를 선택하는 것과도 동일한 이유이다.

위에서 제시한 두 가지 이유에는 공통적인 원인이 있다. 그 것은 새로운 언어들이 왜 좋은지에 대해서 납득할만한 설명을 하지 못했다는 점이다. 이와 관련해서는 Java의 사례를 들 수 있다. Java는 비교적 최근에 나온 언어이다. 1995년에 등장하자마자 인기를 얻어 C와 C++을 대체하며 주요한 언어로 자리 잡았다. 현재의 방대한 기능을 갖춘 Java를 보면 그럴 만도 하지만, 당시 처음 등장했을 때에는 다른 신생 언어들과 크게 다르지 않았을 것이다. 그럼에도 불구하고 성공할 수 있던 이유는 무엇일까? 자바 개발자 Gosling의 전략이 큰 역할을 했다고 생각한다. Gosling은 논문을 통해, Java는 C++의 주요 기능을 포괄하며, 추가로 C++이 갖는 이러저러한 문제점들을 보완하였다는 점을 분명하게 명시하였다. 기존에 C++을 사용하던 프로그래머들은 논문에서 제시한 문제점들을 납득하고, 자바의 필요성에 공감할 수 있었을 것이다. 이는 또한 책임자들에게도 충분히 변명거리를 제공해준 셈이다. 누군가가 왜 그 언어를 선택했냐고 실패의 책임을 물으려할 때, Gosling의 설명을 이유로 제시한다면 누가 그를 탓할 수 있겠는가.

『Toward Programmer Interactivity: Writing Games In Modern Programing language』[3]에 제시된 사례는 설명의 부족으로 인해, 첨단 언어의 장점이 얼마나 과소평가되고 있는지를 보여준다. 일반적으로 첨단 언어들은 높은 추상화 수준을 제공하기 때문에, 느리다고 생각한다. 필자도 ML을 어느 정도 사용했지만, 당연히 그런 줄로만 믿고 있었다. 하지만 실험결과, 꼭 그렇지만은 않다. 동일한 게임을 하나는 C언어를 사용해, 다른 하나는 ML의 방언인 Ocaml을 사용해 구현하고 실행했을 때, 속도가 각각 125fps, 100fps로 측정되었다. 게임 프로그래밍, 특히 그래픽을 다루는 것이 기계에 가까운 C언어에 유리한 분야라는 점과, 당시 Ocaml이 거의 최초 버전이었다는 점을 고려할 때, 이는 두 언어 간에 일반적으로 유의미한 속도 차이가 있는지 여부도 판단하기 어려운 정도의 수치이다. 하지만 아무도 알지 못했다. 설명을 해주지 않았기 때문이다.

3. 첨단 언어의 우수성

위에서 살펴본 것처럼, 첨단 언어의 가치를 인정받기 위해서는 이것이 기존의 언어들보다 왜 우수한지를 설명할 수 있어야 한다. 다른 언어의 익숙함에 빠진 프로그래머들이 납득할 수 있게, 책임자들까지도 위험을 감수할만하다고 느끼도록, 그리고 이들을 질책할 더 높은 사람들이 보기에도 충분한 변명거리가 될 수 있도록 말이다.

이를 위해서는 누구나 이해하고 인정할 수 있는 기준이 필요하다. 단순히 전문 용어나, 기술 이름을 나열하는 것으로는 불가능할 것이다. 그렇다고 이해할 수도 없는 코드를 무더기로 제시하는 것도 좋은 방법이 아니다. 그리고 그 기준을 『Taste for maker』[4]에서 찾

을 수 있었다. 이 글에서 Paul Graham은 분야를 막론하고 좋은 설계가 갖는 열네 가지 성질을 소개하고 있다. 이 중 일부를 ML에 적용해 보았다.

첫째, 간단하다. ML의 한 줄은 C언어의 열 줄에 대응되는 표현력을 갖는다고 한다. 코드의 길이와 프로그램 작성에 걸리는 시간이 비례한다고 할 때, ML은 프로그램 개발 시간을 1/10 수준으로 줄여준다. 하지만 사실은 그 이상이다. 사람의 머리에는 한꺼번에 고려할 수 있는 용량의 한계가 있기 때문이다. 코드가 길어지면 전체 흐름을 머릿속에 담기가 어려워 실수가 잦아지고 이는 오랜 디버깅으로 이어진다. ML의 간략한 코드는 디버깅 시간을 줄여줌으로써 추가적인 효율성을 낳는다.

둘째, 쉬워 보인다. 대부분 프로그래밍을 처음 배울 때, “Hello world!”를 출력하는 코드를 접했을 것이다. 이것이 C언어로 작성된 코드였다면, include, main, void, %d 등 그 의미를 파악하기 어려운 단어들로 인해 혼란을 겪었을 것이다. 간단하다고 소개된 예시 코드임에도 불구하고 완벽히 이해하기 위해서는 많은 개념을 익혀야 한다. 하지만 ML의 경우 print_string ("Hello World")라는 간단한 문장으로 동일한 기능을 표현할 수 있고, 이는 프로그래밍을 처음 접하는 사람도 의미를 파악하는 데에 어려움이 없다. 이처럼 ML은 함수나 변수 정의 시에 타입을 명시해줄 필요가 없다는 점, 복잡한 포인터를 다룰 필요가 없다는 점 등 덕분에 초보자가 쉽게 익힐 수 있다. 이런 장점은 비단 초보자들에게만 적용되는 것이 아니다. ML에서는 함수를 값으로 취급하는 것이 가능한데, 이는 함수까지도 실행시간에 동적으로 결정할 수 있도록 함으로써 프로그램에 추가적인 유연성을 제공한다. 그리고 실제 ML 프로그램에서 유용하게 활용된다. 물론 C에서도 같은 기능이 제공된다. 함수형포인터가 것이 그것인데, 이는 사용법이 꽤나 복잡하다. 혹자는 C코드에서 함수형포인터가 거의 사용되지 않는다는 점을 들어서 해당 기능의 유용성을 부인할 수 있다. 하지만 이는 선후관계가 뒤바뀐 생각이다. 실제로는 C언어에서 함수형포인터를 다루는 것이 복잡했기 때문에 사용을 피하게 되었고, 자연스레 이를 사용하지 않고 문제를 해결하는 것에 사고과정이 길들여진 것이다. 이 역시 기존 언어가 주는 익숙함의 함정인 것이다.

셋째, 자연(혹은 사람)을 닮는다. C언어의 경우 컴퓨터라는 기계로부터, 이를 효율적으로 다루기 위한 목적으로 등장하였다. 반면 ML은 프로그램이란 무엇일까라는 질문에서 시작해, 수학과 논리학을 바탕으로 설계되었다. 그 결과 전자가 컴퓨터의 연산 과정과 유사한 흐름을 가졌다면 후자는 보다 사람의 사고 과정과 유사한 흐름을 가지게 되었다. 서론에서도 언급했듯이 ML로 프로그램을 작성하는 것은 귀납적으로 수학을 증명하는 과정과 유사하다. 프로그램의 흐름이 사고의 흐름과 유사한 덕분에 주석도 많이 필요하지 않다.

넷째, 문제를 제대로 해결한다. 『Strong Typing』[5]에서 M-J. Dominus는 C와 ML의 타입 시스템을 비교하고 있다. C와 ML 모두 프로그램을 실행하기 이전에, 주어진 프로그램의 타입이 아귀가 맞는지 여부를 확인하는 정적 타입 검사를 택하고 있다. C언어의 타입 검사는 안전하지 않다. 타입 검사를 통과한 프로그램도, 실행 중에 타입과 관련해 오류를 발생시킬 수 있다. 반면 ML의 타입 검사는 안전하다. ML의 타입 검사를 통과한 프로그램은 적어도 타입과 관련된 오류를 내지는 않는다. 물론 ML의 타입시스템은 완전하지는 않다. 올바르게 돌아갈 수 있는 프로그램인데도 불구하고 타입 검사를 통과하지 못할 수 있는 것이다. 이 경우는 코드를 다시 작성함으로써 문제를 해결할 수 있다. 하지만 C의 타입 검사처럼 안전성에 문제가 있는 경우는 비용이 크다. 에러라는 것이 소프트웨어를 배포한 후에 혹은, 나로호에 실려 머나먼 우주에 도착했을 때 발생할 수도 있기 때문이다. 이 경우 수정을 위해서는 막대한 비용을 감수해야만 한다.

다섯째, 연상 능력을 활용한다. 언어가 사용자에게 필요한 기능을 제공하는 방법에는 두 가지가 있다. 첫째는 언어에 해당 기능을 포함하는 것이다. 두 번째는 사용자가 기능을 구현할 수 있도록 일종의 레고 블록들을 제공하는 것이다. 첫 번째 방법은 효율적이지 못하다. 언어가 복잡해져야하고 시간도 오래 걸리며 사용자의 다양한 요구를 모두 만족시키는 것이란 사실상 불가능하다. 사용자에게 약간의 불편을 초래하기는 하지만, 두 번째 방법이 실질적으로 사용자의 요구를 충족시킬 수 있는 유일한 방법일 것이다. 이런 불편을 최소화하기 위해 ML은 사용자 정의 타입과, 순서쌍, 리스트 등 훌륭한 도구를 제공한다. 이를 이용하면 다양한 타입의 자료구조들을 손쉽게 디자인할 수 있다. 예를 들어 C에서는 구현이 까다로운 트리를 ML에서는 (Left, Node_Value, Right)이라는 순서쌍만으로 쉽게 디자인할 수 있다. 이것이야 말로 물고기를 주는 대신, 물고기를 쉽게 잡는 법을 알려주는 셈이다.

이 밖에도 그 탄생 과정을 고려할 때 ML은 어려움, 대담함 등의 성질을 갖고 있다. 비록 [4]에는 포함되어있지는 않지만, 실행기 형식을 취해 상호작용성을 증대, 디버깅을 용이하게 한 점, 값 중심의 언어로서 포인터 사용을 제한, 안전성을 향상시킨 점 등도 ML의 장점이 될 것이다.

4. 첨단 언어가 보완해야 할 점

첨단 언어가 우수하긴 하지만, 기존 언어들이 상대적으로 우월한 측면도 있으며, 오랫동안 많은 사람들에 의해 사용되면서 이러한 장점은 더욱 뚜렷해졌다. 이는 첨단 언어의 진출을 가로막는 장벽이 되고 있다. 기존 언어와 비교해 첨단 언어가 보완해야할 점은 크게 세 가지를 들 수 있다. 첫째, 다른 언어로 작성된 기존 프로그램과의 이식성이 문제가 될 수 있다. 둘째, 하드웨어 혹은 시스템 단계에서 조작하기 위한 지원이 부족하다. 셋째, 사용할 수 있는 라이브러리가 부족하다.

『Growing a Language』[6] 을 참고하면, 첨단 언어의 부족한 점들은 실제로는, 오랜 기간 사용되면서 다양한 기능을 갖추고 덩치가 커진 기존 언어들에 비해, 작은 언어인 신생 언어들이 갖는 문제점임을 알 수 있다. 하지만 위 글의 저자에 따르면 작은 언어라는 점을 오히려 기회로 활용할 수도 있다.

언어를 설계하는 방식에는 두 가지가 있다. 성당 방식과 시장 방식이 그 것이다. 성당은 많은 사람이 함께 짓지만, 한 명의 설계자가 전체적인 방향과 모든 세부 사항을 결정한다. 반면 시장은 미리 정해진 계획 없이 시장에 참여하는 사람들의 행동에 의해 전체 모습이 결정된다. 이를 프로그래밍 언어에 적용할 경우, 소수의 개발자들에 의해 개발, 관리되는 언어는 성당 방식, 해당 언어를 사용하는 프로그래머들의 필요와 노력에 의해 언어가 발전해나가는 방식은 시장 방식에 해당할 것이다.

작은 언어가 시장 방식을 택해 발전할 경우 여러 이득을 얻을 수 있다. 다양한 라이브러리를 빠르게 확보할 수 있다는 것, 전체 설계의 방향이 자연스럽게 사용자의 필요성을 충족시키는 쪽으로 나아간다는 것이 그것이다. 또한 언어와 프로그래머가 함께 발전해 나갈 경우, 그 언어를 충실히 이해하고 있는 사용자층이 두텁게 형성된다는 장점도 있다. 현재의 C나 Java 같은 커다란 언어가 갑자기 시장에 등장한다고 해보자. 그 많은 기능에 질려서 아무도 익히려고 하지 않을 것이다. 프로그래머들과 함께 발전해나갔기 때문에 지금과 같은 복잡함에도 많은 사람들이 사용할 수 있는 것이다. 즉 첨단 언어들은 작은 언어이기 때문에 오히

려 발전할 수 있는 가능성이 있는 것이다.

작은 언어인 첨단 언어가 프로그래머들의 참여를 통해 효율적으로 발전하기 위해서는 언어를 성장가능하게 만들어야 한다. [6]에서 일반형 타입과 연산자 오버로드라는 적절한 사례를 제시하고 있다. 더하기라는 연산자는 여러 의미를 갖는다. 정수, 실수는 물론이고 벡터, 행렬 등 대수적 구조에서 공통적으로 사용하기 때문이다. 하지만 피연산자가 무엇이냐에 따라서 그 의미는 조금씩 다르다. 각 피연산자에 대해서 다른 버전의 더하기 연산자를 제공하는 것은 비효율적인 성장이다. 겉으로 보이는 부피만 커져서 프로그래머들을 복잡하게 하지만, 실질적으로 얻는 것은 적고 모든 경우를 다 포괄할 수도 없다. 하지만 일반형 타입과 연산자 오버로드를 적용하여, 임의의 타입을 매개 변수로 전달할 수 있고, 각 타입에 맞게 더하기 연산자의 의미를 사용자가 직접 바꾸어서 사용할 수 있도록 한다면 다르다. 한 개의 라이브러리만으로도 모든 경우를 포괄할 수 있다. 부피는 얼마 커지지 않지만 기능은 더욱 강력해지는 말 그대로 '효율적인 성장'인 것이다.

마지막으로 시장 방식의 성장을 택하기 위해서는 기반이 탄탄해한다. C언어의 경우 양적으로는 방대한 발전을 경험했지만, 첨단 기술을 추가하는 질적인 성장을 이루기는 어렵다. 질적인 성장을 위해서는 기존의 엉성한 기반 자체를 바꾸어야 하는데 이 경우 기존에 누적된 라이브러리나 상용 프로그램이 올바르게 작동하지 않는 이식성 문제가 발생할 수 있기 때문이다. 첨단 언어들도 기반이 탄탄하지 않다면 상용화되더라도 몇 년 후에는 같은 상황에 처하게 될 수 있음을 경계하여야 할 것이다.

5. 나가며

첨단 언어가 산업에 활용되지 않는 현실은 미래의 프로그래머들에게는 불행이다. 이미 첨단 언어의 맛을 들인 사람이, 다시 낮은 수준의 언어를 사용하는 데에서 오는 불편함과 오류, 디버깅을 감수해야 한다는 사실은 생각만으로도 괴롭다. 하지만 이런 현실을 역으로 기회로서 이용하는 것도 가능하다. 그 것이 [1],[2]의 저자인 Paul Graham의 조언이다.

창업을 하는 경우, 첨단 언어를 사용함으로써 경쟁력을 확보할 수 있다. 즉 제작 기간을 단축하고 사용자의 요구에 빠르게 대처함으로써 표준 언어를 사용하는 기존 기업을 압도할 수 있다. 이식성 문제가 적은 서버 기반 사업일수록, 설계에 오랜 시간을 들여야 하는 어려운 문제일수록 첨단 언어의 효과는 커질 것이다. 사실 이런 노력은 단순히 한 기업의 이익에 그치는 것이 아니다. 이런 기업들이 성공하고 일자리를 늘려간다면 혹은 기존의 기업들이 경쟁력을 확보하기 위해 첨단 언어로 전환한다면, 산업 전체적으로 첨단 언어의 보급을 앞당기는 결과로 이어지게 될 것이다.

물론 창업을 계획하지 않는 친구들에게도 기회는 있다. 교수님이 수업시간에 말씀하셨듯이, 첨단 언어 혹은 필요에 의해 직접 설계한 언어를 C나 Java 코드로 변환해주는 컴파일러를 작성하는 방식으로 말이다. 편안한 환경에서 작업하고, 여유를 되찾을 수 있도록 도와줄 자신만의 비밀무기를 만드는 것이다. 물론 이런 능력을 갖추는 것이 아직은 머나먼 일처럼 생각되지만 말이다.

---

[1] Paul Graham, 『Beating for average』, 2003년 4월, (http://www.paulgraham.com/avg.html, 2010. 11. 17)

[2] Paul Graham,『Revenge of the nerds』, 2002년 5월,
(http://www.paulgraham.com/icad.html, 2010. 11. 17)

[3] James Hague,『Toward Programmer Interactivity: Writing Games In Modern Programing language』, 1999년 8월

[4] Paul Graham,『Taste for maker』, 2002년 2월
(http://www.paulgraham.com/taste.html, 2010. 11. 17)

[5] M-J. Dominus,『Strong Typing』, 1999년 9월

[6] Guy L. Steele Jr.『Growing a Language』, 1998년 10월