## 1-4 언어 디자인 입문(전편)

> 지금까지 언어의 구현에 대하여 배웠고 이제부터는 언어의 디자인에 대하여 생각해 보기로 하자. 이번에는 케이스 스터디로서 Ruby의 초기 디자인에 대해 회고해 본다. Ruby는 스크립팅을 지원하는 객체지향언으로서 개발을 시작했다.

당신이 새로운 프로그래밍 언어를 만들고 싶다고 가정하자. 단순히 개인 소장용 장난감이 아닌, 모두에 의해 사용될 ‘사랑받는 언어’를 목표로 한다고 하면 어떻게 언어를 만들어야 좋을 것인가?

#### 사랑받는 언어를 만드는 방법

새로운 프로그래밍언어가 태어난다. 이 언어가 사랑을 받아 자주 쓰이려면, 성능과 기능 보다 언어의 사양이 중요하다. 하지만 어떻게 언어의 사양을 디자인할 지에 대해서는 웹 페이지나 서적들은 거의 없다.

다시 생각해보면, 제대로 프로그래멍 언어를 디자인한 경험이 있는 사람이 없다. 프로그래밍 언어 관련 교과서들은 있지만, 여기에서 다뤄지고 있는 것은 ‘프로그래밍 언어의 만드는 법(구현법)’이다. 여기에서 다루는 언어도 샘플레벨, 다시말해 기존 언어의 서브셋 레벨이다. 언어의 디자인 부분은 교과서의 영역 밖이기도 하고 교과서의 저자들도 ‘사랑받는 언어’를 만들어본 경험이 없는 사람들이기 때문이리라.

그도 그럴것이, IT업계에서 사용될 정도의 프로그래밍 언어는 손에 꼽을 정도이다. IT역사를 되돌아봐도 과거의 언어까지 합하면 수백가지도 안 될 것이다. 여기서 ‘사랑’의 정의가 좀 모호한 하긴 하지만, 다시 말하면 언어의 설계자들도 세계에서 수백명 정도밖에 없다는 이야기도 된다. 이미 그 중 몇 명은 이 세상 사람이 아니다 .

그래서, 얼마 되지 않은 언어 설계자의 한명으로서, 언어 디자인의 비결에 대해 소개해 놓는 것이 나의 사명이 아닐까 생각하게 되엇다. 이 책의 목적은 여기에 있다.

#### 솟아오르는 의문

언어설계자에 뜻을 둔 사람이 새로운 언어를 작성하려 할 때, 항상 드는 의문이 아래와 같다.

- 정말 새로운 언어가 필요한가?

- 이 언어는 뭘 하는 언어인가?

- 예상 유저는 어떤 유저인가?

- 어떤 기능을 채용할까?

하지만 이 의문들의 대부분은 고민만으로 답이 나오지 않는다.

생각해 보자. 예를 들어 ‘정말 새로운 언어가 필요한가?’는 어떤가. 새롭다는 관점이 ‘튜닝완전[^1]’한 언어라면, 온갖 알고리즘의 기술이 가능하다. 하지만 기존 프로그래밍 언어가 튜닝 완전하다는 이미 증명되었다. 따라서 소프트웨어를 개발한다(=알고리즘을 기술한다)는 관점에서 보면 미래에 등장할 언어들을 포함해 새로운 언어는 전혀 필요가 없다.

언어를 만든다는 것은 이런 것은 아닐 것이다. 과거 50년 이상을 거쳐 언어가 만들어져 왔다는 것은, 지금까지 언어에서 알고리즘의 기술이 불가능했기 때문은 아닐 것이다. 새로운 언어가 더 ‘작성하기 쉽고’ ‘간결하다’ 는 느낌이 들기 때문이다. 어떻게 보면 ‘기능적으로 필요해서’ 라는 고민의 필요는 제로에 가깝다.

#### 자기만 쓸 거면 그것으로 충분

‘이 언어는 무엇을 하는 언어인가’ 와 ‘예상 유저는 어떤 유저인가’ 라는 의문에 대해서는 조금 내용 보충을 할 필요가 있다.

내가 오랜동안 ‘프로그래밍언어의 오타쿠’ 로서 수많은 프로그래밍 언어를 접하고 배워왔다. 또 Ruby가 널리 알려지게 된 이후로, 수 명의 언어 설계자와 교류를 해 왔다. 예를 들어 C++ 설계자 Bjarne Stroustrup라든 가 Perl의 Larry Wall, Python의 Guido van Rossum, PHP의 Rasmus Lerdorf 등등이다. 이 경험으로부터 아는 범위에서 이야기하면 ‘설계자 자신이 자기가 쓸 목적으로 설계한 언어가 사랑받는다’ 라는 것이다.

자기가 쓸 것이 아니라면, 세부까지 신경써서 설계할 일이 없겠지만, 인기을 얻을때까지 키워낼 모티베이션도 유지하기가 쉽지 않기 때문이다. 언어가 사랑을 받을 때 까지 10년 이상의 기간이 걸린다는 그렇게 드문일이 아니다. 세부까지 신경을 쓸 꼼꼼함 및 모티베이션의 유지, 이 두개가 사랑받는 언어의 불가결한 요소이다. 결국 사랑받는 언어의 ‘예상 유저’는 우선 설계자 자신이며 또 이와 비슷한 성격을 가진 유저들이다. 그리고 ‘뭘 할까?’ 는 설계자 자신이 뭘 하고 싶냐는 것에 의존한다고 할 수 있다.

예상 유저와 대상 장르가 정해지면, 마지막 의문인 ‘어떤 기능을 채용할 까?‘ 라는 것은 그렇게 고민할 것이 없다. 여기에도 비결과 팁이 있지만, 나중에 설명하도록 하겠다.

### ■ Ruby를 개발하게 된 계기

이렇게 얘기를 해도 설득력이 없을 것 같아, 케이스 스터디로서 Ruby의 예를 들어보겠다. Ruby는 20년간 내가 깊게 관여 해 왔기 때문에 여러가지 이야기거리가 있다. 여기에서는 특히 언어설계의 방향성을 정하게 된 개발 초기에 초점을 맞추어 회상을 해 보겠다.

우선 Ruby를 개발이 시작된 배경을 보자.

Ruby의 개발이 시작된 것은 1993년도이다. 내가 프로그래밍 언어에 관심을 가졌던 것은 1980년대초, 돗토리현에서 살았던 고등학생 때였다. 이때부터 Pascal, Lisp, Smalltalk를 시작으로 프로그래밍 언어에 깊게 빠져 있었다.

또한 컴퓨터를 개인이 소장하지 못해, 제대로 직접 프로그램을 작성할 수 없는 시대였기 때문에 프로그래밍 언어에 관심을 갖는다는 자체가 이상하게 생각될 때였다. 어떤 프로그램을 작성할 것인가보다 프로그램을 작성하기 위한 수단을 위한 언어에 매료되었었다.

이런 ‘프로그래밍 언어 오타쿠’ 였던 고등학생 이었지만, 어쨌든 살고 있던 곳이 지방이어서 자료, 문헌의 입수가 쉽지 않았다. 또, 인터넷이 당시에는 없었고, 학교 도서관에도 컴퓨터 관련 서적이 거의 없을 때여서 정말 활동을 하기는 쉽지 않았다.

프로그래밍 언어에 관련한 정보 수집은 컴퓨터 잡지에서 언어를 다룬 것을 찾아 본다던지 근처 서점에서 대학 교과서같은 책을 서서 읽어서(당시 고등학생들은 대학교재를 살 수 없었다) 하였었다.

그 후 대학교 진학후에는 도서관에 각종 서적, 잡지, 논문등이 잘 구비되어 있어 여기는 천국이 아닐까 싶을 정도였다. 그렇게 해서 배운 프로그래밍 언어 관련 지식이 나중의 언어 설계에 많은 도움이 되었다. 책을 읽지 않는 소설가나, 과거의 기보를 모르는 프로 기사가 거의 없는 것처럼, 새로운 언어 설계에도 기존의 프로그래밍 언어에 관한 폭넓은 지식이 중요하다.

#### 1993도에는 시간이 충분했다

1993년도였다. 그때에는 벌써 대학도 졸업하고, 직업 프로그래머로서 취직하고 있던 나는, 회사 업무 명령에 따라서 소프트웨어를 개발하고 있었다.당시 내가 개발하고 있던 것은, 사내에서 이용하기 위한 시스템이었고 주로 UNIX 워크스테이션상에 데스크탑이나 첨부 메일 시스템 등이었다. 지금은 Windows나 Mac을 많이 사용하지만, 당시의 UNIX 워크스테이션에는 그러한 시스템은 없었다.비슷한 게 있어도 일본어를 취급할 수 없어서 혼자 개발을 할 수밖에 없었다.

1992년경 거품경제가 붕괴되자 회사 전체의 경기가 나빠졌다. 비용이 발생하는 사내 시스템의 신규 개발은 모두 중지가 되어 버렸고, 개발이 끝난 부분만은 운용을 계속한다는 경영 방침이 내려왔다.

개발팀은 해체되어 몇 안 되는 인원만이 유지 요원으로 남았다.다행인지 불행인지, 나는 유지보수 요원에게 할당되어 남겨지게 되었다. 운영업무였기 떄문에 할 것은 별로 없고, 가끔 “컴퓨터의 동작이 이상해요” 라고 전화가 오면, “그럼, 재기동해 주세요”라고 대답하는 나날의 연속이었다. 그야말로 좌천된 인력이었다.

#### 서적의 기획이 기회가 되어..

하지만 나쁜 일만 있는 것은 아니었다.경기가 안 좋아 잔업은 사라지고 보너스가 없어졌다. 거품기에 비해 수입은 많이 줄었지만, (당시 신혼이었던 나에게는 경제적으로 어려웠지만) 해고되지 않았다. 때문에 일을 찾을 필요는 없고, 눈앞에는 컴퓨터가 있고, 일이 적고 중요도 낮아서 관리를 핑계로 간섭하는 이도 없었다.시간과 마음에 여유가 있어서, 한동안은, 수 개의 작은 유틸리티 프로그램을 개발하고 있었는데, 우연히 사소한 기회를 만나 오랜 꿈인 프로그래밍 언어를 만들어 보려고 결심을 하게 된다.

그 ‘사소한 기회’란,이렇다. 당시 나와 같은 부서에 있던 선배가 책을 출판하는 기획을 세워 놓았었다.

그리고 집필개시 단계에서 나에게 제안이 왔다. “이른바 「언어를 만들면서 배우는 오브젝트 지향」이라는 기획을 세웠는데, 언어 부분을 도와주면 안되겠니?”

언어의 오타쿠였던 나로서 상당히 끌리는 기획이어서 그 기획을 맡기로 했다. 하지만, 그 기획은 편집 회의를 거치지 않고, 곧바로 사장되어 버린다. 그러나, 그렇다고 해서 모처럼 의욕이 생긴 걸 그만 둘 생각은 없었다. 언어를 만드는 것이야말로 나의 오랜 꿈이었기 때문이다. 꿈이었지만, 언어가 완성되는 모습을 이미지화하지 못하고, 좀처럼 모티베이션이 올라가지 않고 있던 중, 겨우 불이 붙었는데 여기서 멈추는 것은 너무 아깝다고 생각했다.

이 ‘해보자’ 라는 마음이야말로, Ruby 20년의 역사의 시작이었다.이 때는, 온 세상에서 널리 사용되는 프로그래밍 언어로 성장할 것이라곤 꿈에도 생각하지 않았다.

그런데, 언어를 만든다고 할 때 고개를 드는 의문에 대해서는 이미 말했다.‘정말로 새로운 언어가 필요한가?‘ 등의 4가지 의문이다.

20년 후의 지금이라면, 그런 일은 신경쓸 필요가 없다고 단언할 수 있지만, 당시의 나는 아직 어렸기 때문에 그 시점에서 한참을 생각했다.결국 조금 더 생각한 후 자신의 언어는 자신을 위해 만들어야 한다는 생각을 했다.지금 생각하면 이때의 판단이 모든 것을 결정한 것 같다. 당시의 나는 C프로그래머였고 주로 사용하는 언어는 C나 쉘이었다. 중규모 이상의 시스템을 개발할 때에는 C, 일상적으로 이용하는 비교적 소규모의 프로그램을 개발할 때 쉘을 사용하고 있었다. 당시(실은 지금도), C에는 별로 불만을 느끼지 않았고, C가 대상으로 하는 시스템 언어를 신규로 만드는 것에는 별로 필요가 없다고 생각하였다.그 때 이미 C++가 존재했던 것도 있었고, 대학의 졸업 연구로 C베이스의 객체 지향 언어를 하나 설계한 것도 (너무 만족할 만한 완성도는 아니었지만) 관련되어 있었는지도 모르겠다.

#### 쉘에 불만이 있었다

오히려 불만이 있었던 건 쉘 쪽이었다.그때 나는 bash를 사용하고 있었고 명령어 들을 나열해 간단한 제어 구조를 추가할 정도라면 쉘 프로그래밍으로 충분했다.그러나, 프로그램이 점점 커지고 복잡해지면 무엇을 하고 있는지 알 수 없게 되기 십상인 것도, 제대로 된 데이터 구조를 가지지 않는 점도 불만이었다. 요컨대, 쉘에서는, 약간의 제어 기능이 있는 커맨드 라인 입력 밖에 없고, 너무 ‘간단한 언어’인 것이 문제였다. 당시 쉘에 가까운 영역(스크립팅이라고 부른다)으로, 좀 더 보통 언어에 가까운 언어로서는 Perl이 있었다.하지만 내 관점에서는 이 언어도 역시 ‘간이 언어’의 였다. 데이터 구조가 스칼라(문자열 및 수치)와 배열, 해시밖에 없다는 것도 불만이었다. 이것으로는 어느 정도 이상 복잡한 데이터 구조를 직접적으로는 표현할 수 없다.

당시만 해도 Perl4의 시대였고 Perl5의 객체 지향 기능은 소문 수준으로만 존재했다. 하지만 전해 듣기로 Perl5의 오브젝트 지향도 그다지 만족스럽지 않은 것 같았다.Perl보다 리치데이터 구조를 제대로 표현할 수 있는 언어가 좋다고 생각했고, 고교시절부터의 객체 지향 프로그래밍의 매니아로서, 구조체 뿐만 아니라, 제대로 된 객체 지향의 언어로 하고 싶다고도 생각했었다.

#### 파이썬은 너무 평범헀다

한편, python이라는 언어도 존재했다. 이 언어에 관한 정보가 별로 없었기에 힘들여서 여러가지를 조사했다. 그 결과 객체 지향기능이 뒷받침되지 않는 것이 불만이었고 하나 더, 이번에는 ‘너무 평범하게’ 느껴지는 게 마음에 들지 않았다.

내 자신이 제멋대로라고 생각은 되지만 ‘이상적인 언어’에 대한 나의 의지는 멈출수가 없다.

‘너무 평범하게’의 의미는 어떤 것인지 궁금해하는 분이 계실지도 모르겠다. 파이썬은 언어 레벨로의 정규 표현 서포트도 없고, 문자열 처리 기능도 빈약해서, 언어 사양 레벨로 스크립팅을 지원하고 있는 것 같지는 않았다.(어디까지나 20년 전의 파이썬의 이야기이다)파이썬의 특징 중 하나인 인덴테이션로 블록을 표현하는 시도는 재미있다고 생각했는데, 이것은 반대로 결점도 있었다. 예를 들면, 템플릿으로부터 프로그램을 자동생성하려고 했을 때에, 인덴테이션이 제대로 유지하지 않으면 프로그램이 정상적으로 동작하지 않는 점이나 혹은 인덴테이션에 의한 블록 구조를 위해, 언어 사양상, 식과 구문이 명확하게 구별되어 있는 점 등이다. 보통 Lisp의 파생언어를 사용했을 때와 비교해서, 문법이 조금 읽기 쉬운 것 이외에는 다르지 않다는 느낌이었다.지금 생각하면, 커뮤니티라든지 엄청난 자산등은 보지못했지만, 당시는 아직 그러한 부분들을 전혀 보지못했었다.

#### 스크립트 객체지향을..

하지만 다른 언어를 보면서 자신이 만들고 싶은 언어의 모습이 뚜렷하게 보이기 시작했다. 그것은 쉘에는 가까운 영역이며, Perl보다는 보통의 언어에 가까워 데이터 구조를 자유롭게 정의할 수 있고 또한 객체 지향 기능을 갖춘 언어였다. 당연히 파이썬보다 심리스(Seamless)하게 객체 지향 프로그래밍을 할 수 있고 문자열처리를 비롯한 스크립팅에 특징적인 기능을 지원하는 기능과 라이브러리를 갖추고 있어야 한다. 당시 점점 스크립팅의 중요성이 높아지고 있고 Perl이나 Python의 사용사례도 증가하고 있었다. 하지만 동시에 이러한 스크립팅 분야의 객체 지향 프로그래밍의 필요성은 아직 충분히 인식되지 않았다.

당시의 오브젝트 지향 프로그래밍은, Smalltalk(대학등에서는 Lisp도), 혹은 C++를 가리켰고 일반적으로는 대학 연구소에 연구용 프로그래밍이나, 대규모로 복잡한 시스템 개발에만 사용되는 테크닉라고 간주되고 있었다.스크립팅과 같은 소규모이면서 비교적 심플한 프로그래밍은 불필요하다고 인식되고 있었다. 이런 상황이 드디어 변화를 보이기 시작했다.

Perl은 이제 겨우 객체 지향 기능의 도입을 계획중이었고 파이썬은 이미 객체 지향 언어이긴 하지만 모든 데이터가 객체라고 하는 것은 아니고(당시에는), 오브젝트 지향 프로그래밍도 할 수 있다는 절차형 언어였다. 스크립팅을 주안점으로 한 절차형 프로그래밍도 가능한 객체 지향언어라는 것이 나오면 분명 사용하기 쉬울 것이다. 적어도 제 자신은 기꺼이 쓸 거다.

의욕충만이었다. 나는 프로그래머의 3대 미덕[^2]의 하나인 오만을 충분히 유지한 인물이기 때문에, 만들려면 Perl이나 파이썬에 지지 않는 것을 만들겠다고 결심했다. 또 그것이 가능할거라 믿었다. 근거 없는 자신감은 무서운 거지만, 이러한 근거 없는 자신감이야말로 종종 모티베이션의 원천이 된다.

### ■ Ruby 개발 개시

개발을 개시할 때 처음 결정한 것은 이름이었다. 보석 이름은 Diamond이나 Emerald 같은 것이 많아서 좀처럼 좋은 것을 찾지 못했는데 최종 후보에 남은 것이 Coral(산호)과 Ruby였다.결국, Ruby쪽이 짧고, 아름답다고 하여 이를 채용하기로 했다. 그때는 깊게 생각하지 않았지만, 프로그래밍 언어의 이름은 빈번히 불리는 것이니까, 부르기 쉽고 인상적인 것이 매우 중요하다. 만약 여러분이, 자신의 언어를 개발하게 된다면, 꼭 짧고 좋은 이름을 생각해보라.언어의 특징을 단적으로 \*\*나타내는 이름이 있다면 그 이상 좋은 게 없을 것이다.\*\* 하지만 Ruby처럼 언어의 성질과는 전혀 상관없는 이름도 문제는 없을 것이다. 최근의 문제는, 흔한 이름에서는 ‘구글러빌리티(Googlability)’가 낮다는 약점은 있다. Ruby를 개발하기 시작한 1993년에는 없었던 문제였었다.

#### 블록 구조의 표현방법 고안

이름 다음으로 결정한 것은 블록 구조의 표현에 end를 이용하는 것이었다. C나 C++, Java는 블록 구조로 복수의 구문을 정리하는데 중괄호({})를 이용한다.이 방식에는 한가지 문제가 있다. 단문을 복문으로 쓸 때, 괄호를 잊어버리기 쉽다는 것이다.(그림 1). 중괄호 대신에 begin과 end를 사용하는 Pascal에서도 결국은 단문과 복문의 구분이 있어 같은 문제가 있다.

```

//복문시에는 중괄호로 싼다

if (cond){

statement1();

statement2();

}

//단문의 경우 중괄호로가 없어도 된다

if (cond)

statement1();

//단문을 복문처럼 쓰는 경우 중괄호를 잊어버린다면...

if (cond)

statement1();

statement2(); //문법상 에러는 되지 않는다

```

<center>

(그림 1) 단문과 복문의 문제

</center>

나는 단문과 복문 문제를 그리 좋아하지 않았기 때문에 내가 만들 언어에서는 이런 문제들이 발생하지 않도록 하려고 했다. 이를 위한 대응 방법이 3가지가 있었다.

1. 단문에서의 중괄호 생략을 허용하지 않는 Perl의 방식

2. 블록 구조를 인덴테이션으로 표현하는 파이썬 방식

3. 단문과 복문의 구별 없이, 블록을 가지는 구문은 end로 끝내는 Eiffel방식(그림 2)

```

■ 단문의 경우

if cond

statement1();

end

■ 복문의 경우(단문과의 구별은 없음)

if cond

statement1();

statement2();

end

■ 복수의 블록이 있는 빗(Comb)과 같은 형태

if cond

statement1();

elsif cond2

statement2();

else

statement3();

end

```

<center>

(그림 2) Eiffel방식(빗과 같은 형태의 구조)

</center>

#### 자동 인덴트가 과제였다

그런데 나는 오랜동안 에디터로 Emacs를 써 와서 이 에디터의 언어 모드에 아주 익숙해져 있었다. 그리고, 이 언어 모드가 제공하는 자동 인덴트가 상당히 마음에 들었었다. 적당하게 프로그램을 입력하면 에디터가 자동적으로 인덴테이션을 해 주는걸 보면 ‘아, 에디터의 도움을 받으며 프로그램을 개발하고 있구나’ 라는 기분이 들었다.

2번 파이썬 방식으로는, 인덴테이션 그 자체가 블록 구조를 표현하므로 오토인덴트의 여지가 없다(파이썬의 경우, 행 끝의 콜론으로 인덴테이션의 단계 표현이 가능하지만). 또, 블록으로 인덴트을 표현하는 파이썬 에서는 문장과 식의 구별이 명확해져 식과 문장의 구별이 없는 Lisp의 강한 영향을 받고 있던 나는 별로 마음에 들지 않았다. 그래서, 파이썬 방식의 블록에 의한 인덴트는 채용하지 않기로 했다. 내가 학창시절에 큰 영향을 받은 언어는 Eiffel이었다.학창시절에 ‘Object-oriented Software Construction(객체 지향 입문)’이라는 책을 읽고 상당히 많은 영향을 받은 나는 졸업연구로서 세만틱(Semantic)한 Eiffel다운(하지만 문법은 C다운) 언어를 디자인한 적이 있다. 그 시도는 솔직히 실패했지만, 이번에는 세만틱한 것이 아니라 문법을 Eiffel로부터 이어 받는 게 어떨까 생각했다.

#### Emacs로 언어 모드를 만들다

계속 염려가 되는 것은 자동인덴트었다. 당시의 Emacs의 언어 모드로, 자동인덴트를 해 주는 것은 C 와 같은 블록을 기호로 표현하는 것이 주류였다. Pascal등의 예약어를 사용해 블록 구조를 표현하는 언어의 모드는 키 조작으로 인덴트를 깊게 하거나 얕게 하거나 하는 것 뿐이었다. 이래서는 자동 인덴트의 ‘기분 좋은 점’이 사라져 버린다.

며칠 동안 Emacs Lisp와 씨름했다. 정규 표현등을 사용해 간단하게 Ruby의 문법 해석을 실시해, end가 있는 문법에서도 자동 인덴트를 할 수 있는 Ruby 모드의 프로토 타입을 작성했다.이것에 의해 end가 있는 Eiffel스러운 문법의 언어로도 자동인덴트가 가능하다는 것이 증명되었다.이것에 의해, Ruby의 문법을 end를 사용하는 것이 가능했다. 반대로 말하면, 이 때 자동인덴트를 하는 Ruby 언어 모드의 개발에 성공하지 않았다면, 지금의 Ruby의 문법은 없었다고 해도 과언이 아니다.

이 end를 사용하는 블록 구조라는 디자인상의 선택은 예상치도 못한 점이 있었다. Ruby의 대부분은 C로 구현하였으므로, 필연적으로 C와 Ruby를 구사하게 되었다. 하지만 C와 Ruby는 외관이 전혀 다르기 때문에 현재 어느 언어로 작업하고 있는지는 한눈에 알 수 있었다. (그래서 뇌의 모드 전환 비용이 낮아졌다) 물론 그 비용차는 적은 것이겠지만 프로그래밍시 리듬을 유지하는데 매우 도움이 되었다. 또 나중에 Perl, Python, Ruby가 스크립트 언어의 라이벌로 간주되게 되었을 때, 각각의 언어가 완전히 다른 블록 구조를 가지고 있었다(Perl은 중괄호, Python은 인덴트, Ruby는 end)는 것도 생존에 효과적이었을 것이라고 생각된다.

\*\*else if 일까 elsif일까 elif일까\*\*

여담이지만, 상기와 같은 복문 시스템을 채택하면, 그 부작용으로 C와 같은 elseif을 쓸 수 없게 된다. C의 elseif는, else 뒤로 중괄호 없이 if가 단문으로 있다고 해석되기 때문이다(그림 3). Ruby와 같은 문법으로, 같은 일을 하면 (그림 4)와 같게 된다.

```

// (a) else if를 사용한 아래 구문은

if (cond){

....

}

else if (cond2){

....

}

// 중괄호를 생략하지 않은 구문이 된다

if (cond){

....

}

else {

if (cond2){

....

}

}

```

<center>

(그림 3) C의 else if

</center>

```

# 결국 Ruby에서 elsif가 없다면

# 언제나 아래와 같이 작성할 필요가 있다.

if cond

....

else

if cond2

...

end

end

#역시 elsif가 있는 쪽이 더 좋다

if cond

....

elsif cond2

....

end

```

<center>

(그림 4) Ruby의 else if

</center>

(그림 4)를 보면 역시 elsif가 있는 편이 더 명확하다. 덧붙여서 Perl과 Ruby는 elsif이고, 쉘과 Python(과 C프리프로세서)은 elif다.이 차이는 흥미롭다. 파이썬은 쉘이나 C프리 프로세서로부터 elif을 계승하였고, 쉘 등은 오래된 Algol계열로부터 계승했다고 한다(나중에 쉘의 fi나 esac과 같이, 시작 예약어를 거꾸로 표기해 끝남을 표기하는 것이 기원이라고 들었다).

Perl이 왜 elsif로 했는지는 안타깝게도 알 수 없으나 Ruby의 경우는 다음과 같은 이유가 있다.

- else if를 ‘발음나는대로’ 빠르게 읽으면 elsif(elseif는 좀 길기도 하고 elif는 발음이 좀 이상하지 않은가..)

- Ruby의 기본 문법은 대부분 Eiffel이며 여기서 elsif를 쓰고 있다.

예약어 하나에도 역사와 이유가 있는 것이다.

여담으로, Perl의 문법은 C와 거의 같지만, 중괄호를 생략할 수 없기 때문에, (그림 3)의 논리에 의해 elseif는 허용되지 않는다. 그러나 문법에 명시적으로 else와 if의 조합을 만들면 elseif을 허용할 수 있다. 벌써 꽤 오래됐지만 어느 날 문득 떠오른 Perl의 소스코드를 만져봤는데 yacc 기술을 약간 변경하는 단 몇 분간의 작업으로 elseif을 허용하는 Perl을 만들 수 있었다. 지금도 왜 Perl 커뮤니티 사람들이 이런 작업들을 안하는지 잘 모르겠다.

#### 구현 개시

기본적인 방침이 정해지고, 문법의 방향성도 정해지면 구현에 들어간다. 다행히 이전에 취미삼아 만든 장난감언어의 소스가 남아 있었기 때문에 그것을 바탕으로 개발하기로 했다.개발을 시작한 것이 1993년 2월, 이후 구문해석기와 런타임의 기초부분이 한 번에 완성되어, 최초의 Ruby프로그램(Hello World였다)이 동작한 것이 반년 후의 8월이었다. 솔직히 말하면, Ruby의 개발 중에 이 때가 가장 힘든 시기였다. 그래도 프로그래머라고 하는 인종(人種)은, 자신이 쓴 코드가 동작하는 모습을 보고, 프로그래밍의 기쁨을 느끼기 마련이다. 이 떄 Ruby로 동작하기 위해 필요한 것이 아무것도 갖추어지지 않았기 때문에 아무리 뭔가를 작성하더라도 동작할 수 있는 상태가 되지 않았다. 모티베이션의 유지가 힘들었다.

구문해석기를 작성해도 가능한 부분은 문법체크뿐이고, 정작 프로그램을 작동시키려 해도 ‘Hello World’는 문자열 객체이기 때문에 문자열 클래스가 필요하다. 문자열 클래스를 위해서는 객체 시스템이 필요하고, 출력을 위해서는 IO를 관리하는 객체가 필요하고... 필요한 것이 고구마 줄기처럼 늘어갔다. 프로그래머 3대 미덕 중 하나인 조급함을 충분히 갖춘 내가 이 반년을 견뎌낼 수 있었던 건 기적 같은 일이었다.

#### 인기는 디테일에 숨어 있다

하지만 Hello World의 출력 성공으로 이 언어를 쓸 수 있는 것은 아니다. 여기까지는 교과서의 샘플에 지나지 않는다. 사랑받는 언어가 지향하는 것은 이제부터다.

어떻게 언어에 특징을 부여할까? 사랑받기 위해서(널리 쓰이기 위해서)는 어떻게 해야할 까? Ruby의 초기 디자인에서 어떤 것을 고려해야 할 까.. 할 얘기는 끝이 없다.

하지만 더 얘기했다가는 꼰대가 옛날얘기하는 모양이 되어 버릴것 같아 여기서 접기로 하겠다. 1-5절에서는 이어서 Ruby 디자인의 케이스 스터디의 후반부를 설명하겠다.

<hr>

### 타임머신 칼럼

\*\*\*널리 사용되는 언어의 역사를 알자\*\*\*

> 이 글은 2014년 7월호 게재 분이다. 드디어 언어 디자인 얘기가 시작된다.이번에는, Ruby라고 하는 언어를 만들기 전의 배경이나 경위 말하고 있다. “왜 만들려고 했습니까”라든가 “어떤 부분에서 고생했는지”라든가 “왜 이런 문법을 채용했는가”라고 하는 질문에 답하는 내용위주이다.

> 꼰대의 옛날 이야기지만, 실제로 널리 사용된 언어의 배경이나, 다양한 설계의 배후에 있는 이유등을 이야기할 수 있는 사람은 별로 없기 때문에, 이번 회와 다음 회는, 본서의 하이라이트가 아닐까 생각한다.

> 그래도, 옛날 이야기만으로 치부하기에는 여러가지 도움이 되는 지식들이다. 후진을 위해서라도, 꼭 이러한 이야기로부터 교훈을 얻을수 있었으면 하는 게 바램이다. 우선, 이번의 범위부터 배울수 있는 교훈은

> - 디자인이라는 것은 결정이다

> - 구문이라는 기본적인 것이라도 여러가지 고려해야 할 것이 있다.

> - 생각을 잘 하지 않으면 잘못된 디자인이 나온다

> - 생각을 아무리 잘 해도 잘못은 하기 마련이다

정도가 아닐까.

<hr>

[^1]: 튜링 완전 (turing completeness)이란 어떤 프로그래밍 언어나 추상 기계가 튜링 기계와 동일한 계산 능력을 가진다는 의미이다. 이것은 튜링 기계로 풀 수 있는 문제, 즉 계산적인 문제를 그 프로그래밍 언어나 추상 기계로 풀 수 있다는 의미이다.제한 없는 크기의 기억 장치를 갖는 기계를 만드는 것이 불가능하므로, 진정한 의미의 튜링 완전 기계는 아마도 물리적으로 불가능할 것이다. 그러나, 제한 없이 기억 장치의 크기를 늘려갈 수 있다고 가정할 수 있는 물리적인 기계 혹은 프로그래밍 언어에 대해서는, 느슨하게 튜링 완전하다고 간주한다. 이런 맥락에서, 요즘 나온 컴퓨터들은 튜링 완전하다고 여겨지고 있다.(참조: https://ko.wikipedia.org/wiki/튜링\_완전)(옮긴이)

[^2]: Perl의 제작자 LarryWall에 의하면 프로그래머의 3대 미덕은 무정함, 조급함, 오만함이라고 한다. 보통 미덕이라 불리우는 성질들은 아니다.