언어의 이동: 컴퓨터에서 사람으로

전기공학부
2009-20769
김성준

1. 서론

인간은 옛날부터 자신의 일을 대신해줄 기계를 만들기 위해 노력했다. 그러한 노력의 결정체 중에 하나가 컴퓨터이다. 컴퓨터는 연산을 처리하는 CPU와 데이터를 저장하는 메모리로 이루어져 있다. 인간은 이러한 컴퓨터에 일정한 형식을 가진 명령의 집합을 입력함으로써 인간의 일을 컴퓨터로 하여금 대신 처리하게 하였다. 이 명령의 집합을 소프트웨어라고 한다.

하지만 컴퓨터는 인간의 언어로 표현된 내용을 그대로 이해할 수 없다. 따라서 컴퓨터가 이해할 수 있게 변환을 해야 한다. 이러한 변환은 편리하게 하기 위해 프로그래밍 언어가 개발되었다. 인간과 컴퓨터 모두가 이해할 수 있는 언어인 것이다.

"The Development of the C language"와 "19세기 논리학, 21세기 논리학" 두 편의 글은 프로그래밍 언어의 발달 과정에 대해 논하고 있다. 이 두 글을 읽으면서 프로그래밍 언어가 일정한 방향으로 변해간다고 생각했다. 초기의 프로그래밍 언어는 지극히 컴퓨터 중심이었다. 많은 개념들이 컴퓨터를 좀 더 쉽게 쓰는 것에 초점이 맞춰 개발되었었고 변화해 왔다. 그러나 현재에 와서는 인간 중심으로 변해왔다. 어떻게 하면 프로그래머가 좀 더 쉽게 프로그래밍 할 수 있게 할 것이라는 물음에 답하기 위해 노력했다.

초기의 언어(BCPL, B, C)는 컴퓨터 중심의 언어였다. 기계어를 인간이 좀 더 이해하기 쉽고 구성하기 쉽도록 바꾸어준 정도였다. C 이전의 BCPL, B의 경우 type이 없는 언어로서 memory를 word의 모임으로 보는 컴퓨터의 시각이 반영되어 있었다. 그리고 이를 기반으로 하여 개발된 C 또한 포인터, casting 등 컴퓨터 중심의 언어로서의 특징을 가지고 있다. 이러한 언어를 통해 프로그래밍 하기 위해서는 인간이 생각한 바를 다시금 인간이 컴퓨터가 해석하기 쉽도록 풀이하여(일정한 흐름을 가진 명령어의 흐름으로 구분하여) 코딩을 해야 한다.

그 후의 언어는 인간 중심의 언어로 변해가고 있다. 언어가 인간의 논리 자체를 표현할 수 있는 언어로 변해왔다. 이러한 언어의 예가 OCaml과 Haskell이다. 다음으로 인간이 논리의 표현에만 집중할 수 있도록 편해왔다. 편리한 컴파일과 디버깅. 컴파일이 정상적으로 되고 논리적 오류만 없다면 문제없이 수행될 수 있도록 한다. 이를 위해서 type checking, type matching 등의 기법이 사용되었다. 그리고 이 외에도 가비지 컬렉션을 통해 번거로운 메모리 관리를 하지 않아도 된다.

Functional language가 아니지만 C++, java 또한 인간 중심의 언어로 변해가는 과정으로 볼 수 있다. 인간에게 있어서 object 중심의 언어는 매우 자연스런 개념이다. 이러한 개념을 표현하기 위해서 C++, java에서는 class, 상속, overloading 등의 개념이 소개 되고 발전되어 가는 것이다.

이러한 변화의 원인은 무엇일까? 그리고 우리는 이러한 변화에 어떻게 대응해야 할까? 앞으로 두 언어 그룹의 특징을 읽을 거리와 연계하여 살펴보고 변화의 원인, 우리의 대응에 대해서 생각해보자.

2. 본론: 컴퓨터 중심의 언어

대표적으로 잘 알려진 프로그래밍 언어인 C 언어는 1970년대 초기 Unix 운영체제의 시스템 프로그래밍 언어로서 개발되었다. 그리고 현재는 주류 프로그래밍 언어 중 하나이다. "The Development of the C Language"에서는 BCPL부터 B를 거쳐 C에 이르기까지, C 언어의 발달 과정에 대해 설명하고 있다.

1960년대 후반에 MIT와 General Electric, Bell Labs가 모여 Multics(Multiplexed Information and Computing Service) 프로젝트를 시작하게 되었다. 이 프로젝트는 다중 사용자를 잘 뒷받침하는 운영체제를 구현하는 것으로 필연적으로 많은 시간과 인력, 막대한 비용을 필요로 했다. 이 계획을 효율적으로 운영하여 성공시키기 위해 Ken Thompson은 편리한 컴퓨팅 환경을 만들기 위해 노력했다. 그래서 Thompson은 이 프로젝트에 시스템 프로그래밍 언어가 필요함을 알고 BCPL을 바탕으로 B라는 새로운 언어를 만들었다.

BCPL은 1960년대 중반에 Martin Richards에 의해 디자인 되었고 그 중 많은 부분을 B가 계승하였다. 두 언어 모두(그리고 C 언어 또한) 특정한 기계(컴퓨터)를 대상으로 하여 그 기계를 효율적으로 사용하기 위해서 만들어진 언어이다. 이 언어들의 프로그램은 전역 선언(global declaration)과 함수 선언(function declaration)들의 연속으로 구성된다. 함수 선언을 일반적인 선언과 다르게 보는 것이다.

BCPL을 발전시킨 B에는 세 가지 문제가 있었다. 첫 번째로는 문자 혹은 문자열을 다루는 방식이 정교하지 못 했다. BCPL과 마찬가지로 단일 타입(cell)만을 지원하였기 때문에 B에서 문자열을 다루기 위해서는 따로 문자열 라이브러리를 활용해야 했다. 두 번째로 실수 연산을 지원하지 않았다. 당시에는 아직 컴퓨터에서 실수 연산을 하지 못 했지만, 조만간 가능해질 것이기 때문에 이를 지원해야 했다. 세 번째로 포인터 연산의 오버헤드가 너무 컸다. 매번 포인터를 참조할 때마다 포인터에서 바이트 주소로의 런타임 변환이 필요했다.

이러한 언어 자체의 문제뿐 아니라 B 컴파일러의 threaded-code 테크닉도 문제가 되었다. B 컴파일러는 어셈블리 코드를 생성하는 것이 아니라 threaded-code를 생성하였기 때문에 느린 속도가 문제가 되었다.

그래서 Thompson은 B언어 타입 시스템을 추가하고, 컴파일러를 다시 쓰는 작업을 통해 B의 문제점을 해결하려 했다. 문자열과 실수 연산, 바이트 주소 문제를 해결하기 위해서는 타입이 필요했다. 1971년 Dennis Ritchie는 B에 문자열 타입을 추가하고 컴파일러를 고쳐 NB라는 언어를 만들었다. 그리고 이 NB에 많은 타입을 추가하고 다듬은 결과 C가 만들어지게 되었다. 이 후 표준화(standardization)와 이식성(portability) 문제를 해결하면서 C가 주류 언어로 떠오르게 된다.

이러한 발전 과정에서 초기의 프로그래밍 언어가 매우 컴퓨터를 중심에 주고 설계되었다는 것을 알 수 있다. 이 시기의 프로그래밍 언어는 인간과 컴퓨터 중에 컴퓨터 좀 더 가까이 있다는 느낌이다. 언어에서 나타나는 다양한 기계적 특성이 이 같은 느낌을 주는 것 같다.

단일 타입 시스템이 이 같은 대표적 기계적 특성이라 할 수 있다. 컴퓨터는 오직 0과 1로 이루어진 데이터만을 받아들이고 처리할 수 있다. 프로그램도 다룰 데이터도 오직 0과 1로만 이루어져 있다. 컴퓨터가 수행할 수 있는 동작들도 이러한 이진 데이터만을 대상으로 한다.

이와 달리 인간은 정보를 다양한 형태로 표현한다. 수의 경우만 하더라도 자연수, 정수, 무리수, 허수, 실수 등 다양한 분류를 가진다. 그리고 숫자 이외에도 문자, 소리, 영상 등 다양한 형태로 정보를 표현한다.

앞서 "The Development of the C Language"에서 살펴본 바와 같이 C 언어 이전의 BCPL, B에서는 단일 타입, 셀(cell)만을 가지고 있었다. 이러한 셀 타입은 메모리나 레지스터의 한 word 길이에 해당하는 비트들을 표현하고 있다. 11bit PDP-11에서 셀은 11개의 bit를 표현하고 있다.

이렇듯 BCPL과 B는 타입에 있어서 컴퓨터와 매우 비슷하다고 할 수 있다. 단지 (이진) 정수만을 지원하였으며 문자열을 처리하기 위해서는 별도의 라이브러리가 필요했다. 그리고 실수 연산도 컴퓨터가 실수 연산을 지원하기 시작하면서 프로그래밍 언어도 이 같은 특성을 가지게 된 것이다.

다양한 타입을 가지는 C에서도 이 같은 흔적을 찾을 수 있다. C에서는 비슷한 타입은 별도의 명령 없이 자동으로 서로 호환이 된다. 부호를 가지는 정수 타입 변수에 부호가 없는 정수 값을 할당할 수 있다. 문자열에 정수 1을 더하는 명령문이 유효한 의미를 가진다. 이 같은 암시적 변환(implicit type casting)은 부호를 가지는 정수, 부호가 없는 정수, 문자열을 모두 0과 1로 이루어진 bit들로 보기 때문이다. 이러한 전통이 BCPL과 B로부터 C까지 이어진 것이다.

포인터 또한 초기 프로그래밍 언어의 기계적 특성 중 하나라고 할 수 있다. 컴퓨터는 물리적 메모리를 가지고 있다. 그리고 이 메모리는 word단위로 데이터를 쓰거나 읽을 수 있다. 물리적 메모리는 그 크기가 한정되어 있으므로 메모리의 word 단위 공간마다 정수로 순서를 메길 수 있다. 이 같은 정수로 된 메모리 공간 상의 순서를 메모리 주소라 한다. C 언어에서는 이러한 메모리 주소를 포인터 저장할 수 있다. 그리고 이 포인터를 통해 메모리의 데이터를 직접 다룰 수 있다.

이 같은 포인터는 C 언어에 풍부한 표현력을 가져다 주었다. 동일한 타입의 변수들의 묶음인 배열도 포인터(ex. int arr[10])로 표현할 수 있었고 각 변수들은 포인터 연산(ex. arr + 2 or arr[2]) 등으로 접근할 수 있다. 또한 이를 통해 이진 트리, 연결 리스트 등과 같은 다양한 자료 구조를 구현할 수 있게 되었다.

3. 본론: 인간 중심의 언어

계산이론의 발전과 함께 논리학에서도 "프로그램이란 무엇인가"에 대한 증명이 이루어지고 있었다. 세 번째 글에서 그것에 관한 논증을 하고 있는데 저자가 말하는 요지는 그 제목에서 보듯 "증명과 프로그램은 같다"라는 것이고, 본문은 이 명제가 증명되기까지의 역사를 말하고 있다. Frege부터 시작하는 현대 논리학은 그림으로 나타내던 많은 증명들을 간단한 기호를 이용해서 나타내기 시작하였다. Gentzen은 Frege의 판단방법에 "가정"이라는 것을 도입해서 직관적인 추론규칙들을 통해 논리적 증명체계를 구성하는 것이 가능함을 보여주었다. 동시에 전체의 논리적 식들이 언제나 결론이 되는 논리식의 부분으로 단순화 될 수 있다는 하위 표현식 이론 (subformula property)을 증명하였고 이것을 통해서 상향식(bottom-up) 논리 증명이 가능하게 되었다.

한편 Church는 기계적으로 계산 가능한 모든 함수를 표현할 수 있는 람다 계산법(lambda calculus)을 제시하였다. 수학에서 함수를 등식으로 표현하는 것과 달리, 함수를 하나의 추상적인 개체로 나타내는 표기법을 통해 모든 계산을 대입으로 환원시킬 수 있다. 람다 계산법에서 재미있는 개념 중 하나로는 함수가 함수를 인자로 받는 다는 발상이다. 이러한 발상은 훗날 여러 언어들에서 유용하게 쓰이게 된다. 이후 처치는 타입 람다 계산법(typed lambda calculus)을 고안 하

였는데 람다 항의 타입을 결정하는 타입 도출절차를 포함한 것이다.

위의 두 논문이 동전의 양면과 같이 동일한 내용을 나타낸다는 사실이 1969년에 와서야 Howard에 의해서 증명되었다. 단순하게 말해서 증명 절차를 하위 표현식에 의해 단순화 시켜나가는 과정과 람다 추상식(lambda abstraction)에 인자를 대입해서 연속적으로 계산해 나가는 절차와 본질적으로 완전히 동일하다는 것이다. 람다 계산법은 튜링 기계에서 계산 가능한 함수들을 계산하는 체계이기 때문에 람다계산법과 자연 연역(natural deduction) 체계가 동일하다는 것은 기계적으로 계산 가능한 것들의 집합과 논리적으로 계산 가능한 것들의 집합이 동일하다는 것을 나타낸다.

논리학과 계산학의 이러한 관계는 프로그래밍 언어에도 많은 영향을 미쳤다. 기존의 프로그래밍 언어는 인간의 생각을 표현하기에 부적절 했다. C의 예에서 알 수 있듯이 초기 프로그래밍 언어의 많은 부분은 컴퓨터의 동작을 표현하기에 더 적합했다. 그래서 프로그래머는 프로그래밍을 하기 위해서 자신의 생각을 프로그래밍 언어의 틀에 맞춰 표현해야 했다. 당연히 직관적이지 않고 추가적인 노력이 필요하였다.

그러나 기계적으로 계산 가능한 것들의 집합과 논리적으로 계산 가능한 것들의 집합이 동일하다는 것은 인간의 논리적 구조를 컴퓨터가 이해 가능한 형태로 표현할 수 있다는 것이다. 이와 같은 생각이 적용되어 ML, Haskell과 같은 functional language가 개발되었다. 이 언어들에서의 프로그래밍은 원하는 동작을 논리적으로 정의한 뒤에 이를 기술하는 것으로 한결 직관적이고 단순해졌다. 이러한 언어를 이용해 프로그래밍 하기 위해서는 만들고자 하는 프로그램의 동작을 논리적으로 정리만 하면 된다. 다시금 컴퓨터가 이해 가능하도록 풀어 쓰는 과정이 없어지는 것이다.

이러한 변화를 퀵 소트를 이용해 살펴볼 수 있다. 퀵 소트의 논리적 증명 방식은 간단하다. 리스트의 어떤 원소에 x 대해서, x보다 앞에 위치하는 모든 원소들은 x보다 작고, x보다 뒤에 위치하는 모든 원소들은 x보다 크거나 같다면 이 리스트는 부분적으로 정렬되어 있다. 만약 리스트의 모든 원소가 부분적으로 정렬되어 있다면 이 리스트는 정렬되어 있다.

이러한 논리를 바탕으로 퀵 소트를 C 언어로 나타내면 다음과 같다.

```
void qsort (itemType a[], int l, int r) {
    int i, j;
    itemType v;
    if (r > l) {
      v= a[r]; i=l-1; j=r;
        for(;;) {
           while (a[++i] < v);
           while (a[--j] > v);
           if (i>=j) break;
           swap(a, i, j);
        }
        swap(a, i, r);
        quicksort(a, l, I-1);
        quicksort(a, i+1, r);
```

```
    }
}
```

위의 코드에서 알 수 있듯이 코드가 위의 논리와 일대일로 매칭된다고 보기는 힘들다. 위의 논리를 다시금 프로그래밍 언어로 풀어 썼다고 보는 것이 옳다. 위의 논리를 순차적인 명령으로 풀어 쓴 것이다.

퀵 소트를 OCaml로 나타내면 다음과 같다.

```
let rec qsort list =
  match list with
       [] -> []
     | h::t -> qsort (List.filter (fun x -> x < h) t)
               @ [h]
               @ qsort (List.filter (fun x -> x >= h) t);;
```

한눈에 봐도 이 함수가 어떤 동작을 하고자 하는지 직관적으로 알 수 있다. 여기서 List.filter 함수는 리스트 중에서 특정 조건을 만족하는 원소만을 걸러낸다. 그리고 @ 연산자는 두 개의 리스트를 하나의 리스트로 합쳐준다. 코드를 보면 빈 리스트가 들어오면 빈 리스트를 반환한다. 그리고 원소가 있는 리스트가 들어오면 맨 앞의 원소 h를 기준으로 하여 이 보다 작은 원소들은 h의 왼쪽에, 이 보다 크거나 같은 원소들은 h의 오른쪽에 배치한다. h를 기준으로 부분적으로 정렬된 것이다. 그리고 나서 왼쪽, 오른쪽 리스트를 재귀적으로 qsort 함수를 호출함으로써 전체 리스트를 정렬하고 있다. OCaml 문법과 퀵 소트의 논리를 알고 있는 사람이라면 한 눈에 이 코드가 무엇을 하고자 하는지 알 수 있는 것이다.

인간이 쓰기 쉽고 이해하기 쉬운 프로그래밍 언어는 컴퓨터 보다 인간에 더 가까이 있다고 말할 수 있다. 종전의 프로그래밍 언어가 기계적인 특성을 지니고 있었다면 이 언어들은 인간적인 특성을 가지고 있다. 위에서 말한 것처럼 논리적인 구조를 기술할 수 있고 이를 계산(증명)할 수 있다. 이러한 특성은 인간이 훨씬 편리하게 프로그래밍을 할 수 있도록 도와준다. 그리고 빠르게 이해할 수 있기 때문에 유지, 보수에도 편리한 면을 보인다. 마지막으로 코드가 논리 구조만을 담고 있기 때문에 훨씬 짧고 간결하다.

프로그래밍 언어의 인간적 특성은 잘 짜인 타입 시스템에서도 찾아볼 수 있다. 단일 타입 시스템을 가지는 BCPL이나 B와는 달리 OCaml, Haskell 등에서는 다양한 타입 시스템을 가지고 있다. 기본적인 int, float, bool, char, string 타입 이외에도 튜플, 레코드 등 기본 타입을 조합하여 새로운 타입을 만들 수 있는 방법을 제공함으로써 인간이 생각하는 바를 좀 더 쉽게 표현할 수 있다. 이러한 면은 Object-oriented programming을 지원하는 C++, java의 클래스에서도 찾아볼 수 있다. 그리고 C에 남아있던 암시적 변환의 흔적도 인간적 프로그래밍 언어에서는 사용을 제한하거나 아예 존재하지 않는다. 기계적인 특성으로 생겨난 개념들이 사라지고 더 인간 친화적으로 변해가는 것이다.

OCaml, Haskell 등에서는 포인터의 존재도 사라졌다. 포인터는 C에서 주로 사용되었으며 유한한 크기를 가지는 메모리를 가리켰다. 그러나 개념적인 메모리는 변수에서 값을 매칭시키는 일대일

함수로 볼 수 있다. 즉, 유한한 크기, word 단위로 구분된 bit 등은 이러한 언어에서 드러나지 않는다. 언어 차원에서 추상화하여 일대일 함수로서의 유한하지 않은 공간을 나타내게 되었다. 그러한 이유로 이들 언어에서는 포인터를 통해 물리적 메모리를 직접 다루지 않는다. 그리고 가비지 컬렉션을 통해 물리적으로 유한한 메모리를 효율적으로 사용함으로써 논리적으로 무한한 메모리를 제공하고 있다.

4.4. 본론: 변화의 동기

인간의 특성과 기계적 특성의 차이에서 오는 불편함이 이런 변화를 만들어냈다. 앞서 두 읽을거리를 통해 알아봤던 것처럼 인간과 기계(컴퓨터)는 그 특성이 판이하게 다르다. 그리고 기계적인 특성인 단일 타입 시스템, 암시적 변환, 포인터, 메모리 관리 등은 이전까지의 프로그래밍에서 많은 문제를 야기시켜 왔다. 단일 타입 시스템, 포인터 때문에 복잡도가 증가하여 프로그래밍 하는 것부터가 만만치 않았다. 프로그램 작성 후 컴파일이 되었다 하더라도 잘못된 암시적 변환, 직접적인 메모리 접근, 비효율적인 메모리 관리 등의 문제로 런타임에 프로그램이 비정상적으로 종료되는 일이 부지기수였다. 이러한 오류는 프로그래머가 예측하기 힘들며 찾아내는데 많은 시간이 든다. 더군다나 반복적으로 재현되지 않는다면 디버깅 난이도는 기하급수적으로 올라간다. 이러한 문제는 그 동안 프로그래머의 생산성 향상에 큰 장애물이 되었다. 프로그램을 디자인하고 코드를 작성하는 시간보다 오류를 찾고 고치는 데 쏟는 시간이 훨씬 커져버린 것이다.

또한 코드를 인간이 이해하기 어려웠다. 프로그래밍은 인간의 생각을 컴퓨터가 쉽게 이해 가능한 형태로 기술하는 과정이었다. 따라서 후에 인간이 프로그램 코드를 봤을 때 반대의 과정을 거쳐야 되고 당연히 코드를 이해하는 것이 어렵고 번거로운 일이 되었다. 단체로 진행되는 프로젝트에서 다른 사람의 코드를 잘 못 이해하거나, 심지어 자신이 이전에 작성한 코드를 잘 못 이해하여 잘 못된 코드를 작성하는 일이 왕왕 일어나고 있다. 이 같은 문제도 프로그래머의 생산성에 큰 문제를 일으킨다.

초기의 컴퓨터는 수가 적고 단가가 비쌌기 때문에 그 가치가 매우 높았다. 이러한 프로그래밍의 불편함을 감수하고 많은 노력을 들일만큼 가치가 높았다. 프로그래머의 편의를 위해 컴퓨터의 성능을 희생해가면서 편리한 프로그래밍 환경을 만들 수가 없었다. 다양한 편법을 통해 코드의 길이를 줄이고 조금의 성능 향상을 도모해야 했다.

그러나 최근 컴퓨터의 급속한 발달은 가치의 역전을 가져왔다. 반도체 제조 기술의 발달로 컴퓨터의 성능은 빠른 속도로 높아졌고 그 가격은 낮아지고 있다. 이제는 누구나 컴퓨터를 가지고 있으며 이를 사용할 수 있다. 수십 MHz의 CPU와 수 Kbyte 메모리를 가지던 컴퓨터가 이제는 다수의 수 GHz CPU를 탑재하고 수 Gbyte 메모리를 탑재하고 있다. 이전에 비해 컴퓨터의 성능은 그 가치가 낮아졌다. 프로그램이 조금 느리고 비 효율적이라 하더라도 높은 성능이 체감상의 차이를 숨기고 있다.

이러한 변화는 숙련된 프로그래머의 가치를 높였다. 이전과 달리 프로그래머가 프로그램을 빠르게 디자인하는 것이 성능이 좋은 프로그램을 만드는 것보다 더 중요해졌다. 이러한 흐름에 맞춰 프로그래밍 언어도 편리하고 빠른 프로그래밍을 지원하도록 변화한 것이다.

이 같은 변화는 Functional Language에만 국한된 것이 아니다. JAVA의 경우 포인터를 없애고 메모리로의 직접적인 접근을 허용하지 않는다. 그리고 언어 자체에서 가비지 컬렉션을 제공한다. 이

를 통해 C에서 문제가 되었던 Segmentation fault와 같은 메모리 관련 문제가 발생하지 않게 되었다. 그리고 Ruby, Python 등 컴파일 없이 인터프리터를 통해 바로 실행되는 Script Language도 많이 사용되고 있다. 간략하고 편리한 문법, 다양한 library 등은 다소 느린 성능을 충분히 보상하고 남는 것이다. 그리고 최근 컴파일러 및 인터프리터의 발전으로 이 같은 성능 문제도 대부분 해결된 상태이다.

이런 경제적인 이유가 아니라 프로그래머의 개인적 취향도 발전을 가속화시키는 계기가 되었다. 프로그래밍은 자신이 생각한 바른 프로그램 언어로 기술하여 컴퓨터가 그 동작을 하도록 만드는 과정이다. 여기서 프로그래머는 생각한 바를 기술하는데 더 많은 노력을 들이고자 한다. 지저분한 파일 입출력 함수나 기본적인 자료구조를 디자인하는데 시간을 들이고 싶어하지 않는 것이다. 프로그래밍의 핵심만을 원하는 것이다. 이러한 요구가 현재의 프로그래밍 언어를 만들어 가는 것 같다.

4. 결론

두 읽을거리에서 나타난 내용을 통해서 프로그래밍 언어가 어떠한 방향으로 변해왔는지를 살펴보았다. 이전의 프로그래밍 언어가 컴퓨터라는 굴레에 잡혀 있었다. 최근에는 하드웨어의 급속한 발달, 소프트웨어의 이론적인 연구를 바탕으로 프로그래밍 언어가 컴퓨터에서 좀 더 자유로워지고 있다. 이러한 변화는 프로그래밍의 경제성이라는 측면과 프로그래머 개인의 취향이라는 동기를 가지고 있다. 이러한 동기는 당연히 앞으로도 계속 될 것이고 이러한 변화 또한 점점 더 가속화 될 것이라 예상된다.

다만 인간보다는 컴퓨터에 다소 가까운 언어(C, C++, JAVA)들이 현재의 주류 언어로 취급받는 것에 의문을 가질지 모른다. 이런 언어들은 오랜 시간 동안 많은 사람들이 사용했기 때문에 많은 연구가 진행되었다. 그리고 결과 프로그래머가 생산성을 높일 수 있도록 다양한 기법이 개발되었고 현재의 소프트웨어 기반을 만들고 있다.

하지만 이 같은 소프트웨어의 발전이 언어의 효율성을 의미하는 것은 아니다. 언어 자체의 비효율성을 시간과 노력으로 무마시키고 있는 것이다.

그러나 인간 중심의 언어는 언어 자체의 효율성을 가지고 있다. 이제까지의 예를 보더라도 그 자체의 편리함에 빠른 속도로 확산되어 가고 있다. 그 만큼 더 많은 사람들이 사용하고 연구하고 있으며 많은 경험들이 축적되고 있다. 이러한 경험들은 언어의 확산에 더 힘을 실어주는 순환이 계속될 것이다.

언어 자체의 효율성에 앞으로의 많은 노력이 더해진다면 이제와는 다른 좀 더 즐겁고 생산성이 높은 프로그래밍이 될 것이다. 지금 소프트웨어 엔지니어링을 배우고 있는 우리도 현재의 주류 언어뿐 아니라 현재 개발되고 있는 새로운 언어들을 많이 활용하는 것이 중요하다. 새로운 언어의 특성을 주류 언어에 접목하는 것에서부터 시작하여 새로운 언어를 또 하나의 주류 언어로 만들기 위해 연구하는 것이 우리의 의무일 것이다.