컴퓨터에 양보하세요.

컴퓨터 공학부
2008-11747 최홍림

1. 인간의 능력을 믿는(?) 언어

C언어는 사람을 부려먹는다. C언어는 나에게 이런 걸 시킨다. “Memory management, type checking, pattern matching…… 알아서 하세요.” 숙제를 해야 하니, 별 수 없이 시키는 대로 한다. 그리고 C 컴파일러에게 넘긴다. 컴파일러는 대충 쓱 보고 “되네?”, executable을 뱉는다. 나는 “될까?” 반신반의하는 심정으로 실행한다. 돈다. 좋다. 그러다 뻗는다. 보다 못해 C에 항의한다. “된다며?” C는 대답하지 않는다. 오늘도 자장면을 먹는다.

C는 인간의 능력을 믿는지 이것저것 시키는 게 많다. 그러다 안돼서 물어보면 모르쇠로 일관한다. 이건 숫제 상관이나 마찬가지다. 화가 좀 나지만 숙제할 때는 참고 쓸 만하다. 하지만 수백 만 라인의 프로그램이 뻗으면? 오류를 찾기란, 모래사장에서 바늘 찾는 것만큼 힘들다.

C로 프로그래밍하는 것은 힘들다. C는 놀고 있고, 사람만 일하기 때문이다. Segmentation fault, memory leak, type 조차 다른 결과. 언어 차원에서 방지할 수 있는 오류들이다. 그런데 C는 안 한다. 왜?

2. Close to machine

왜 C는 인간에게 불친절한가? 애초에 기계를 돌리는데 중심을 두고 만들어졌기 때문이다. 그럼 C가 인간에게 불친절하게 된 역사적 배경을 알아보자.

2.1 C의 역사

(아래 내용은 『The Development of the C Language』 (Dennis M. Ritchie.)의 내용을 토대로 작성한 것이다.)

1960년대 Bell Lab에서 Multics라는 machine을 개발하는 project를 진행 중에 있었

다. 하지만 이 project는 "could be fulfilled only too late and too expensively". 이에, Multics project의 리더인, Ken Thompson은 Multics에서 취할 것은 취하고 버릴 것은 버려 DEC PDP-7 용 Unix를 개발한다. 그리고 Unix의 시스템 언어로 사용하기 위해 Thompson이 개발한 것이 B이다. B는 이전에 개발된 언어인 BCPL을 8K byte memory로 제한하고, Thompson의 입맛으로 바꾼 언어이다. BCPL과 B는 시스템 언어로 개발되었기 때문에 close to machine하였다. B는 BCPL의 단점들을 극복했는데 예를 하나 들면 external vector의 offset을 일일이 적어 줘야하는 global vector mechanism을, 컴파일 시 필요한 모든 파일을 binding하여 컴파일러가 알아서 linking 해주는 방식으로 바꾼 것이다. 하지만 이 B도 PDP-11이 개발되면서 문제점이 부각되었다. 비효율적인 character handling mechanism과 machine에 floating point instruction이 있어야만 floating point 연산이 가능한 점, pointer 연산의 overhead가 대표적인 세 가지 문제이다. 이를 해결한 것이 New B인데 여전히 record 할당의 문제, type이 거의 없는 점 등의 문제점이 있었다. C는 참조한 논문의 저자인 Dennis M. Ritchie가 New B를 계승하며 New B의 대안으로 개발한 언어이다. C는, array pointer의 생성은 expression 내부에 array name이 언급되는 때에 생성되는 방식, 다양한 type을 제공하는 등의 혁신을 가져왔다. 기존 언어의 많은 문제를 해결한 C는 표준화를 거쳐 Unix의 확산과 더불어 널리 사용하게 되었다.

C의 조부모, 부모 격인 BCPL과 B가 개발될 당시는 지금에 비해, 제한된 메모리, 제한된 저장공간, 제한된 machine performance, 제한된 programming tools, ……, 한 마디로 열악한 컴퓨팅 환경이었다. 따라서 제한된 자원을 최대한 효율적으로 쓰는 것이 관건이었다. 때문에 최대한 기계 동작을 이해하기 쉽게, 기계에 가깝게 만들어졌다. 한마디로 기계가 돌리기 편하게 프로그래밍을 했다.(인간이 쓰기 편하게 만드는 기술이 부족했기도 했지만.) C가 개발될 당시도 크게 다르지 않은 상황이었고, BCPL과 B를 계승한 C가 시스템 언어의 위상을 갖는 것은 당연한 일이다.
즉, C가 노는 이유는, 안 놀고 일하면 기계가 힘들어지기 때문이다.

2.2 C의 문제점

기계에 가까운 언어이기 때문에 정말 제대로 설계한다면 C로 짜는 것만큼 컴퓨팅 자원을 효율적으로 쓰는 언어는 드물 것이다. 하지만 기계에 가깝다는 성질은 자원의 효율적 사용이란 측면에서는 장점이지만 프로그래밍이란 측면에서는 악몽이다. 그 악몽의 예는 다음과 같다.

2.2.1 Memory management

C는 user가 pointer를 통해 직접 virtual memory에 접근할 수 있다. 하지만 이 때문에 프로그래밍은 복잡해진다. pointer를 이용한 프로그래밍은 변수의 값 변화가 pointer로 엮인 부분의 원치 않는 변화를 불러일으킬 수 있지만, 이 변화의 원인을 찾아내기란 쉽지 않다.

C는 heap 영역의 memory를 할당받았으면 user가 free해 주어야한다. 간단한 룰이지만 프로그램이 커지고, 복잡해지면 free해줘야할 곳에서 free하지 못해 memory leak이 발생한다.

2.2.2 Type checking

C는 int와 pointer, float 등 type간 이동이 자유롭다. 아니, Type checking을 하지 않는다는 말이 옳다. int 값을 pointer type 변수에 넣어도 경고 메시지만 뜨고 만다. 하지만 이는 자칫 중대한 오류로 파생될 수 있다. 하지만 오류가 나더라도 어디서 type이 잘 못 됐는지 찾는 것은 그리 쉽지 않다.

이 밖에도 자잘한 C의 문제는 많다. 이런 비인간적인 C의 문제를 해결하기 위해 나온 언어가 close to human의 성질을 갖는 ML이다.

3. Close to human

ML은 C와는 다르다. 철저히 lambda calculus에 기반을 두고 작성된 언어이고, garbage collector가 있어 memory management를 할 필요가 없으며, 철저히 type checking한다. 강력한 pattern matching tool이 있다. 이러한 기능은 기계를 피곤하게 하지만 사람은 편해진다.

3.1 ML의 탄생

ML은 철저한 수학적 배경을 두고 탄생했다. 『Proofs are Programs』에는 Gentzen's natural deduction과 Church's lamda calculus에 관한 설명이 있다.

Gentzen's deduction은 Frege's notion of judgement에 assumption을 include한 것으로, premise와 conclude로 이루어진 logic과, roundabout proof를 사용해 주어진 logic을 증명하는 방법이다. Church's lamda calculus는 기계적으로 계산 가능한 모든 함수를 lambda term으로 표현 가능하다는 것을 말한다. Church는 Typed lambda calculus도 고안하였다. 1960년, Howard에 Gentzens' natural deduction과

Church's lambda calculus가 일치함을 증명되었다.

기계적으로 계산 가능한 함수는 논리적으로 계산 가능한 것과 동일하다는 것이 밝혀짐으로써, 프로그래밍 언어의 역사는 일대 전환기를 맞이한다. 기존 프로그래밍 언어는 인간이 쓰기 편하게 작성하는 기술이 부족했기에, 기계를 중심으로 작성될 수밖에 없었다. 하지만 인간의 논리적 사고를 프로그래밍 언어로 표현 할 수 있다는 것이 밝혀짐으로써, 좀 더 인간적인, ML과 같은 functional language의 수학적 토대가 생겼다.

ML의 개발자 Robin Milner는 자신의 저서 Computing Tomorrow에서(『계산이란 무엇인가에 대한 아이디어들』 이광근 역.) 컴퓨터 분야의 독립적인 아이디어들을 제시했다.

같은 프로그램이란 무엇인가, 순서대로 계산한다는 것은 무엇인가, 프로그램의 안과 밖은 무엇인가, 옳겨다니는 계산이란 무엇인가.

『An Interview with Robin Milner』 에서 ML이 이러한 고찰과 lambda calculus를 토대로 1974년 Edinburgh에서 ML이 탄생했다.

3.2 close to human 언어의 필요성

Yoran Minsky는 그의 강의 『Caml trading talk on CMU』 에서 Wall Street에서의 경험을 이야기해준다. 그에 의하면 financial market의 관리 프로그램은 다음 3가지 조건을 만족해야 한다. Correctness, agility, performance. 그는 이 조건을 만족시키는 관리 프로그램은 functional programming을 이용하는 것이 좋다고 한다. 그는 세 가지 조건에서 ML의 code(MC)가 다른 Object Oriented Code(OOC)보다 비교우위에 있고, ML의 문제점에 대해서 이야기했다.

3.3 장점

3.3.1 Correctness

MC는 읽기 쉽고, Pattern에 대한 high quality tools를 제공하며, 강력한 Type System을 가지고 있다. 때문에 정확하며, 틀린 부분을 찾기 쉽다. 하지만 OOC의 경우 function이 어디서 불렸는지도 알기 힘들며, code도 길고 읽기 힘들다.

3.3.2 Agility

Financial market에서는 spec의 변화가 빈번하다. 변화에 맞춰 code를 수정하는 것도 중요한 능력이라고 한다. MC의 경우 강력한 pattern matching tool에 힘입어 pattern의 변환이 쉽지만, OOC의 경우 pattern이 여기저기 흩어져있어 변환하기 어렵다.

3.3.3 performance

OOC의 경우 performance가 unstable한다.

3.4 단점

물론 ML도 poor UIs, hard to handling concurrency, few external libraries, hard to programming in the large 의 문제가 있다고 한다. 하지만 financial market에서 ML의 장점은 단점을 충분히 덮을 수 있다고 한다.

Minsky의 주장에 덧붙이자면, 장점으로 강력한 garbage collector와 recursive handler를 들 수 있고, 단점으로 compile 혹은 interpreter overhead를 들 수 있다.

Ocaml의 가장 큰 장점은 correctness라 생각한다. 프원, PL 숙제할 때, Ocaml을 사용하면 Type checking을 통과하는 것이 문제이지, 통과만 하면 대부분 잘 돌아간다. Type checking 때 걸리면 어디가 문제인지 어떤 type을 써야하는지 친절히 알려준다. 오작동의 경우에도 강력한 Pattern matching tool로 Pattern에 따라 분기가 되어있어 내가 잘못한 곳을 쉽게 찾을 수 있다. C나 Java는 그렇지 않다. 컴파일은 쉽게 하지만 정상 동작 하지 않는 경우가 대부분이다. 어디가 문제인지 코드를 읽어가며 확인해야한다. 그런데 code 읽는 게 만만치 않다.

처음에 ML로 프로그래밍하는 것은 쉽지 않았다. for, while loop도 없지, 기본적인 자료구조는 array가 아니라 list이지, 변수를 쓰려면 ref 라고 따로 선언해주어야 하지, 함수가 값처럼 쓰이지, type checking에 걸려 실행도 잘 안되지, …….. 열거하면 끝이 없을지도 모른다. 하지만 이젠, ML에 어느정도 적응되니 C나 Java로 프로그래밍 하는 것이 쉽지 않다. Type checking도 안 해주지, list는 따로 만들어 써야지, 함수는 값처럼 쓸 수 없지, pattern matching은 왜 안 되지, …….

4. 결론

C는 기계가 편한 언어이다. ML은 인간이 편한 언어이다.

Operating System, Compiler, Driver 등 system 프로그래밍 영역에서는 최대한 optimal solution을 찾아야하며, 최대한 기계를 잘 돌리게 짜야하므로 C와 같은 시스템 언어의 필요성은 없어지지 않을 것이다. 하지만 심지어 손에 들고 다니는 핸드폰의 프로세서도 40년 전, PDP-7보다 빠른 지금, 컴퓨팅 자원을 조금 아낀다고 C로 application을 짜는 것은 인력 낭비이다.

물론 3.4에서 언급한 바와 같이 ML도 많은 단점이 있다. 하지만 그 중 poor UIs, few external libraries등은 더 많은 사람들이 ML의 발전에 힘쓰면 메울 수 있는 부분이고, compile 혹은 interpreter overhead는 interpreter 기술이 발전해 나감으로써 줄어들고 있다.

프로그래밍 언어와 에디터는 종교라는 말도 있다. 한 번 정하면 그만큼 바꾸기 어렵다는 뜻인데, 익힐 때 드는 비용을 포기하기란 쉽지 않은 법이다. 특히나 close to machine 언어에서 close to human 언어로 갈아타는 것은 체계의 틀을 바꿔야하기 때문에 더욱 큰 비용이 들 것이다. 그러나 프로그래밍 효율은 ML은 사람이 할 일을 기계가 도와주기 때문에 C에 비해 ML이 월등하다. 이 효율의 차이는 언어를 갈아타는 비용을 충분히 감내하고도 남는다.

마지막으로 Skin food의 광고 카피가 생각난다. “먹지마세요. 피부에 양보하세요.” 나는 이렇게 말하고 싶다. “하지마세요. 컴퓨터에 양보하세요.”