컴파일러와 인터프리터

HandyPost는 한 도영(HDNua)이 작성하는 포스트 문서입니다.

1. 개요

컴파일러와 인터프리터의 개념, 기계어와 어셈블리에 대해 간단히 공부하고, 컴파일러와 인터프리터의 차이를 올바르게 이해하여 후에 우리가 작성할 프로그램이 어떠한 것인지를 분명하게 정의한다.

2. 상식

컴퓨터가 처음에 등장했을 때는 모든 것을 오로지 0과 1로만 표현했다. 처음에 필자가 이 말을 들었을 때 필자는 0과 1로만 프로그래밍을 해본 경험이 없었기 때문에, 사실 이 말은 그렇게 와 닿지 않았다. 0과 1로 표현했다고 하는데 도대체 뭘 어떻게 했다는 건지 적절한 예시가 없기 때문이었다. 사실 초보자나 비전공자에게 0과 1로 컴퓨터를 만들었다는 것 이상을 설명하면 벙찐 표정으로 고개만 끄덕이고 있을 게 분명하므로 설명해봐야 소용이 없는 일이긴 하다. 하지만 우리는 이제 나름 컴파일러를 만든다는 사람, 즉 작성한 소스 코드를 기계어로 번역하는 프로그램을 개발하는 사람이다. 따라서 우리가 이것을 모른다는 건 말이 안 되는 것 같다. 이 문서에서는 컴퓨터의 개론적인 부분을 다루어 실제로 기계어가 무엇이며, 이것으로 어떻게 컴퓨터를 만들 수 있다는 건지에 대한 단서를 제공할 것이다. (컴퓨터는 알다시피 아주 복잡한 기계이므로, 필자 수준의 지식에서 온전하게 설명하기 힘들다) 이야기를 시작하기 전에 먼저, 프로그래머라면 상식적으로 알고 있어야 하는 내용을 정리하는 시간을 가지는 것이 좋겠다.

2.1) 컴파일 & 링크 & 빌드

초창기의 컴퓨터는 기계어로 프로그래밍을 했다. 그러나 기계어는 사람이 이해하기 아주 어려워서, 이를 보다 편하게 사용하기 위해 이런 방법을 생각했다.

- 기계어의 집합을 더 간단하게 표현하는 텍스트 문서를 만든다. 예를 들어 C는 긴 코드를 간단하게 표현하기 위해 함수나 매크로(macro)를 사용할 수 있는데, 기계어로 10줄짜리의 코드를 매크로 A로 정의하고 문서에는 A만 써넣는 경우를 생각하자.
- 이 텍스트 문서를 기계어로 자동 번역하는 프로그램 A를 만든다.
- 텍스트 문서를 프로그램 A를 이용하여 기계어로 자동 번역한다. 이 프로그램을 실행하면 위에서 예시로 작성한 문서의 A가 기계어 10줄로 번역된다.

이렇게 하면 필요할 때마다 텍스트 문서만 수정하여 프로그램을 간단하게 만들 수 있다. 위 내용은 사실 순서가 이상한데, 다시 한 번 순서를 맞춰보자.

- 일정한 형식으로 작성된 문서를 기계어로 자동 번역하는 프로그램 A를 먼저 만든다.
- 이후에 프로그램을 만들 때마다 A가 번역할 수 있도록 일정한 형식으로 문서를 작성한다.
- 문서 작성이 완료되면 프로그램 A를 실행하고 파일을 넘겨서, A가 자동으로 번역해준 기계어 파일을 얻게 된다.

그리고 바로 여기서 사용되는 프로그램 A를 **컴파일러(compiler)**라고 하고, 이때 작성한 일정한 형식의 컴퓨터 명령을 **소스 코드(source code)**, 소스 코드가 저장된 텍스트 파일을 **소스 코드 파일(source code file)** 또는 간단히 **소스 파일(source file)**, 그리고 이를 번역하는 행위를 **컴파일(compile)**이라고 한다.

컴퓨터가 발전하고 작성하는 소스 코드의 양이 늘어남에 따라, 한 파일에 모든 소스 코드를 작성하는 방식이 불편하다는 것을 깨닫게 되었다. 사람들은 소스 코드를 다른 파일에 분리하는 방법을 생각해냈다. 원래 하나였던 파일을 분리했으므로, 프로그램을 완성하려면 분리했던 파일은 모두 연결해야 한다. 이렇듯 분리된 파일을 모아 하나의 실행 가능한 파일을 만들면 이를 두고 파일들을 **링크(link)**했다고 하고, 이때 사용되는 프로그램을 **링커(linker)**라고 한다.

종합하면, 우리는 기계어를 이용하지 않고 실행 파일을 생성하기 위해 다음의 순서를 거친다.

- 소스 코드를 작성하고 파일로 저장한다.

- 저장한 소스 파일을 컴파일러를 이용하여 컴파일 한다. 목적 파일이 생성된다.
- 컴파일러가 생성한 목적 파일들을 링커를 이용하여 링크 한다. 실행 가능한 목적 파일이 생성된다.

링커는 실행 가능한 '목적' 파일을 생성한다. 컴파일러가 생성하는 파일과 링커가 생성하는 파일의 차이는 생성한 목적 파일이 실행 가능 하느냐에 있다.

컴파일과 링크 과정을 합쳐 **빌드(build)**라고 하고, 이때 사용되는 프로그램을 **빌더(builder)**라고 한다.

2.2) 메모리

우리가 C 이상의 고급 언어를 사용할 때는 변수가 정의되는 위치를 별도로 고려하지 않는다. 당연한 말이지만 이는 원래부터 그렇게 프로그래밍이 가능했던 것이 아니다. 메모리는 바이트의 집합이고 변수를 사용하려면 빈 바이트를 찾아서 정의하려는 변수의 크기만큼을 메모리에서 확보하는 작업을 거쳐야 한다. 이에 대해 자세히 알아보자.

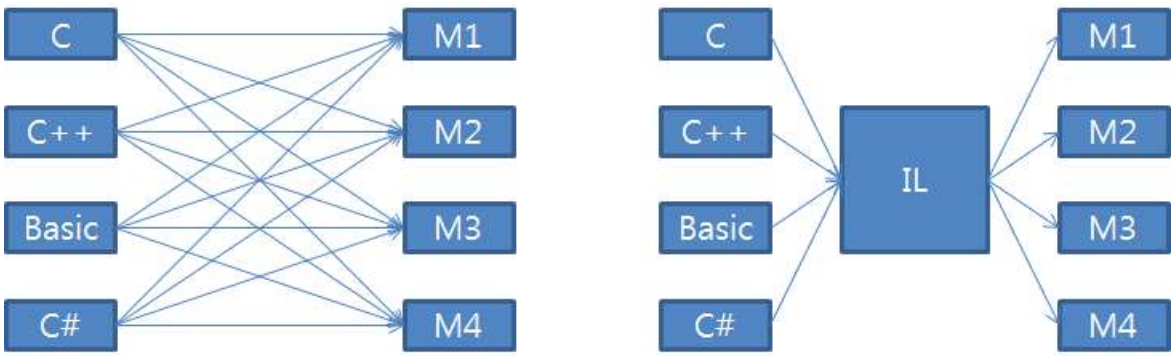
프로그램을 실행하면 해당 프로그램을 위한 메모리 공간을 운영체제가 마련해준다. 프로그램 내부의 지역 변수, 전역 변수, 문자열 상수와 같은 데이터는 모두 이 메모리 공간 내에서 정의되고 사용된다. 이 메모리 공간은 크게 네 가지의 영역으로 구분할 수 있는데 각각 다음과 같다.

- 코드 영역(code area): 실행할 프로그램의 기계어 명령이 올라오는 메모리 영역이다.
- 데이터 영역(data area): 정적 변수 및 전역 변수, 문자열 상수 등 프로그램 실행 시에 정의되고 프로그램이 종료될 때 해제되는 데이터를 저장하는 영역이다.
- 힙 영역(heap area): 동적 할당한 데이터를 저장하는 영역이다.
- 스택 영역(stack area): 지역 변수 등 임시적으로 사용되는 데이터를 저장하는 영역이다.

이 정도로 다음 내용을 진행하기 위한 상식들을 정리할 수 있었다.

2.3) 중간 단계 언어

먼저 생각해보자. C는 익히 알고 있듯이 기계어보다 사람이 이해하기 쉬운 고급 언어다. 또한 우리는 C를 이용하여 작성한 소스 코드를 실행 가능한 파일로 만들기 위해 컴파일 과정을 거쳐야 한다. 그렇다면 우리가 작성한 소스 코드와 기계어 사이에 중간 단계의 언어가 있어서, 우리의 소스 코드가 이 중간 단계의 언어로 먼저 번역된 다음, 번역된 중간 단계의 언어가 기계어로 번역되는 상황도 생각할 수 있다. 번역할 거면 바로 번역하지 왜 가운데 단계가 하나 더 들어가느냐고 묻는다면, 사실 생각대로 없어도 된다(없는 것이 당연히 번역이 더 빠르게 된다). 그러나 이것이 문제되는 이유 첫 번째는 기계어에 대해 우리가 아는 것이 하나도 없다는 점이고, 기계어를 안다손 쳐도 소스 코드를 하나하나 기계어로 번역하는 과정은 무지하게 헷갈린다는 점이다. 둘째는 C 이외의 다른 언어에 대한 컴파일러 B를 새롭게 개발하는 상황이 왔을 때 이 과정에서 사용한 중간 단계 언어 번역기를 그대로 사용할 수 있다는 이점이 있기 때문이고, 셋째는 기계어는 기계마다 그 내용이 다르기 때문에 기계가 바뀔 때마다 해당 기계에 맞는 기계어로 번역해야 하는데, 중간 단계 언어 번역기가 도입되면 이 과정이 아주 단순해지기 때문이다. 다음 이미지를 보면 이에 대해 이해할 수 있다.



이는 여기서만 사용되는 기술이 아니다. Java 프로그래밍 언어는 JVM이라는 가상 머신에서 실행되기 전에 중간 단계 언어인 자바 바이트 코드로 변환된다. C# 프로그래밍 언어는 IL이라는 중간 단계 언어로 컴파일 된다. 이제 여러분도 중간 단계 언어가 필요하다는 사실을 받아들일 수 있을 것이다.

3. 인터프리터(interpreter)

인터프리터(interpreter)란 컴파일 과정을 거치지 않고 소스 코드를 바로 해석하여 결과를 출력하는, 소스 코드 실행기 프로그램을 말한다. 컴파일러와 많은 부분이 공통적이지만 차이도 많은데 이에 대해 간단하게 얘기해보자. 먼저 컴파일러는 소스 코드를 기계어로 번역하는 행위를 한다. 이는 인터프리터도 수행하는 작업이다. 컴퓨터에 명령을 내리려면 기계어로 작성해야 하니까. 그런데 컴파일러는 기계어를 번역한 후에 그 즉시 프로그램을 실행하지는 않는다. 반면 인터프리터는 실행기이므로 번역과 분석이 끝나면 프로그램을 실행하여 결과를 바로 내놓는다. 이것이 컴파일러와의 첫 번째 차이다.

다른 점 또 하나는, 컴파일러는 기계어로 번역을 한 후 목적 파일을 생성하는 반면 인터프리터는 그렇지 않다는 점이다. 컴파일 하는 데 시간이 걸리지만, 컴파일이 끝난 프로그램은 완전히 분석된 상태이므로 다시 코드를 분석할 필요 없이 바로 실행할 수 있다는 점 및 최적화가 용이하다는 점, 실행 속도가 빠르다는 점이 장점으로 꼽히며, 이는 곧 인터프리터의 단점도 된다. 인터프리터는 실행 중에 동적으로 소스 코드를 분석하고, 최적화가 어려우며, 실행 중에 매번 분석을 진행하므로 컴파일러로 번역한 프로그램보다 느릴 수밖에 없다. 하지만 인터프리터는 결과를 바로 확인할 수 있다는 점, 컴파일 하는 데 시간이 걸리지 않는다는 점을 장점으로 꼽을 수 있다.

이와 같이 인터프리터에 대해 간단히 알아볼 수 있었다.

4. JSCC의 개발 방향

JSCC는 컴파일러와 인터프리터가 결합된 형태로 작성할 것이다. JSCC를 실행하고 작성한 소스 코드 파일을 입력으로 넣으면, JSCC 내부의 컴파일러 모듈(Compiler)이 소스 코드는 “필자가 독자적으로 만든 중간 언어”로 번역하여 텍스트 형식의 목적 파일에 기록한다. 그렇게 생성한 목적 파일은 다시 JSCC 내부의 링커 모듈(Linker)이, 생성된 목적 파일을 모아 실행 가능한 목적 파일을 새롭게 생성한다. 그리고 최종적으로 JSCC 내부의 실행기 모듈(Runner)이 링커가 생성한 실행 가능한 목적 파일을 가상머신처럼 실행하여 결과를 화면에 출력한다.

이와 같이 JSCC의 개발 방향에 대해 살펴볼 수 있었다.

5. 단원 마무리

이전에 공부한 내용에 비해 아주 짧은 내용을 공부했다(1장의 소개 글보다도 짧다!). 이는 2주 만에 60쪽 가까이를 공부한 여러분에게 내용을 다시 한 번 되짚어볼 기회를 주는 것이 좋겠다고 생각한 필자의 판단이다(사실 이게 원피스 같은 만화도 아니고 올라오면 바로바로 챙겨봐야 할 이유는 전혀 없지만). 다음에 배울 내용은 CIL이라고 하여 필자가 기계어에 더 적응하기 쉽도록 C를 이용하여 고안한 기계어와 C의 중간 언어인데, 이를 공부하면 어셈블리 언어는 차이점을 배우는 정도로 단순하게 공부할 수 있으리라 생각한다.