차 례

Introduction

책 도입부로 삼기에는 다소 무례하게 들릴지 모르지만 약간 거짓말을 보태서 이야기를 시작해보겠다. 이 책의 전편인《밑바닥부터 만드는 인터프리터 in Go(Writing An Interpreter In Go)》(이하《인터프리터 in Go》또는 전편)는 내가 상상했던 것보다 훨씬 많은 인기를 누렸다. 뭐, 앞서 말한 대로 거짓말이다. 그냥《인터프리터 in Go》가 성공하는 '상상'을 잠시 해봤을 뿐이다. 내가 쓴 책이 베스트셀러 목록 가장 위에 놓여 있고, 각종 찬양과 존경의 미사여구에 둘러싸여 있으며, 훌륭한 이벤트의 연사로 초청되어 낯선 이들 앞에서 사인해주는 상상을 했다. 무려 Monkey 프로그래밍 언어로 책을 썼는데, 누구라도 이런 상상을 하지 않을까? 그런데 정말로 진지하게 말하자면, 나는 정말로《인터프리터 in Go》가 그렇게 성공할 줄은 몰랐다.

물론 나도, 최소한 몇몇 독자는《인터프리터 in Go》를 즐겁게 읽으리라 예상했다. 왜냐하면 나 역시도《인터프리터 in Go》와 같은 책을 바랐지만 찾을 수가 없었기 때문이다. 비록 그런 책을 찾는 데는 성과가 없었지만, 다른 이들 역시 그런 책을 찾고 있다는 사실을 알게 됐다. 다시 말해, 이해하기 쉽고, 다른 코드에 의존하는 바람에 너무 쉽게 쓰여 있지 않으며, 테스트가 잘된 구동할 수 있는 코드를 담고 있는 책이 없었다는 뜻이다. 그래서 내가 만약 이렇게 책을 쓴다면, 최소한 이런 책을 찾던 사람들은 재밌게 읽으리라 생각했다.

그러나 내가 어떤 생각을 했던 실제로 일어난 일을 말하자면, 정말 많은 독자가《인터프리터 in Go》를 즐겁게 읽었다. 그냥 책을 사서 읽은 수준이 아니라 내게 이메일을 보내서 이런 책을 써줘서 고맙다고 감사의 뜻을 표하기도 했다.《인터프리터 in Go》를 얼마나 즐겁게 읽었는지 블로그에 글을 쓰기도 했다. 그리고 그 글을 SNS에 공유하였고, 사람들

이 '좋아요'를 많이 눌렀다. 코드를 이리저리 실행해보면서 가지고 놀아보고, 그들이 확장한 코드를 GitHub에 올리기까지 했다. 심지어 책 내용 교정에도 많은 도움을 줬다. 내가 잘못 적은 내용을 고쳐 보내줘서 찾을 때마다 미안함을 표하기도 했다. 아마 이들은 자신들이 보내준 제안과 교정에 내가 얼마나 고마워하고 있을지 상상도 못할 것이다.

그리고 다른 책도 써달라는 메일을 한 통 받았는데, 이게 마음속에 잠들어 있던 무언가를 건드렸다. 잠자고 있던 어떤 생각이 의무로 변한 순간이었다. 다시 말해 후속 편을 써야겠다고 다짐했다. 내가 그냥 '두 번째 편'이 아닌 '후속 편'이라고 말한 데는 이유가 있다. 왜냐하면 전편인 《인터프리터 in Go》는 타협의 산물이었기 때문이다.

내가 《인터프리터 in Go》를 쓰기 시작했을 때는 후속 편을 염두에 두고 집필하지 않았다. 그냥 책 한 권만 쓰고 끝내려 했다. 물론 완성될 책이 너무 길어진다는 것을 깨달았을 때 생각이 바뀌긴 했지만 말이다. 나는 사람들이 책이 너무 두꺼워 꺼리게 되는 것을 원치 않았다. 그리고 설령 내가 그런 책을 쓰게 되더라도, 책을 쓰는 데 너무 오랜 시간이 걸려 진작에 포기했을 것이다.

그래서 타협하기로 했다. 먼저 트리 순회 인터프리터를 만들고 다음 단계로 가상 머신으로 바꾸는 게 원래의 계획이었지만, 트리 순회 인터프리터를 만드는 내용만 넣기로 했다. 그렇게 《인터프리터 in Go》라는 책이 탄생했다. 그리고 여러분이 지금 읽고 있는 이 책이 그 후속 편이다. 내가 항상 쓰고 싶었던 내용을 담고 있다.

여기서 후속 편이라는 게 정확히 무슨 뜻일까? 이 책이 전편과 매끄럽게 이어진다는 뜻이다. 같은 방식, 같은 프로그래밍 언어, 같은 도구 함수, 전편에서 끝마친 코드베이스까지 그대로 사용한다.

방법은 간단하다. 전편에서 끝마친 곳부터 다시 Monkey 언어를 만들면 된다. 이 책은 전편을 단순히 이어쓰기만 하는 책이 아니라, Monkey 언어를 계승하고 있다. Monkey 언어는 이제 진화해야 한다. Monkey 언어가 어떤 모습으로 변할지 보기 전에, Monkey 언어가 어땠는지 다시 한번 되돌아볼 필요가 있다.

진화하는 Monkey

과거와 현재

《인터프리터 in Go》에서는 Monkey 프로그래밍 언어용 인터프리터를 만들었다. Monkey는 단 한 가지 목적을 위해 만들어진 언어이다.

> "독자가 맨땅에서부터 직접 Go 언어로 인터프리터를 만드는 것"

Monkey 언어의 공식 구현체는 《인터프리터 in Go》에 포함되어 있다. 그러나 독자들이 직접 다양한 언어로 만들어낸 비공식적 구현체는 대단히 많고, 인터넷에서 쉽게 찾아볼 수 있다.

여러분이 Monkey가 어떻게 생겼는지 잊었을 것 같아, 내가 아래에 Monkey 언어가 가진 기능을 최대한 표현하도록 간추린 코드를 가져왔다.

```
let name = "Monkey";
let age = 1;
let inspirations = ["Scheme", "Lisp", "JavaScript", "Clojure"];
let book = {
    "title": "Writing A Compiler In Go",
    "author": "Thorsten Ball",
    "prequel": "Writing An Interpreter In Go"
};

let printBookName = fn(book) {
    let title = book["title"];
    let author = book["author"];
    puts(author + " - " + title);
};

printBookName(book);
// => prints: "쏠스텐 볼 - 밑바닥부터 만드는 컴파일러 in GO"

let fibonacci = fn(x) {
    if (x == 0) {
        0
    } else {
        if (x == 1) {
            return 1;
        } else {
```

```
            fibonacci(x - 1) + fibonacci(x - 2);
        }
    }
};

let map = fn(arr, f) {
    let iter = fn(arr, accumulated) {
        if (len(arr) == 0) {
            accumulated
        } else {
            iter(rest(arr), push(accumulated, f(first(arr))));
        }
    };

    iter(arr, []);
};

let numbers = [1, 1 + 1, 4 - 1, 2 * 2, 2 + 3, 12 / 2];
map(numbers, fibonacci);
// => returns: [1, 1, 2, 3, 5, 8]
```

기능 목록으로 뽑아보면, 아래와 같다. Monkey가 지원하는 기능이라고 보면 된다.

- 정수(integers)
- 불(booleans)
- 문자열(strings)
- 배열(arrays)
- 해시(hashes)
- 전위/중위/인덱스 연산자(prefix/infix/index operators)
- 조건식(conditionals)
- 전역/지역 바인딩(global/local bindings)
- 일급 함수(first-class functions)
- 반환문(return statements)
- 클로저(closures)

제법 길지 않은가! 우리가 Monkey 인터프리터에 이런 기능을 모두 구현했다니 정말 훌륭하지 않은가? 그리고 심지어 서드파티 툴이나 라이

브러리를 사용하지 않고 맨땅에서부터 직접 만들어냈다.

'렉서(lexer)'부터 만들기 시작해서 REPL에 입력된 문자열을 토큰으로 바꿔냈다. 렉서는 `lexer` 패키지에 정의했고, 렉서가 만들어낼 토큰은 `token` 패키지에 정의되어 있다.

렉서 다음으로는 파서(parser)를 구현했다. 토큰을 추상구문트리(AST, Abstract Syntax Tree)로 변환하는 재귀적 하향 파서(프랫 파서)를 만들었다. 추상구문트리에 있는 노드는 `ast` 패키지에 정의했다. 그리고 파서 구현체는 `parser` 패키지에 정의했다.

파서를 거치면, Monkey 프로그램은 메모리상에 트리 형태로 존재하게 된다. 따라서 다음 단계는 그 트리를 평가하는 작업이다. 트리를 평가하기 위해 우리는 '평가기(evaluator)'를 만들었다. 평가기는 `evaluator` 패키지에 정의된 `Eval`이라는 함수의 다른 이름이기도 하다. `Eval` 함수는 재귀적으로 트리를 순회하며 내려간다. `object` 패키지에 정의된 객체 시스템을 활용해서 노드를 평가하고 값을 만든다. 예를 들어 `1 + 2`를 표현하는 추상구문트리 노드는 `object.Integer{Value: 3}`로 변환된다. 이렇게 Monkey 코드가 갖는 생명주기는 완료되고 REPL에 출력된다.

문자열을 토큰으로, 토큰을 트리로, 트리를 `object.Object`로 바꾸는 변환 사슬(chain of transformations)은 우리가 만든 Monkey REPL의 메인 루프에 처음부터 끝까지 잘 표현되어 있다.

```
// repl/repl.go

package repl

func Start(in io.Reader, out io.Writer) {
    scanner := bufio.NewScanner(in)
    env := object.NewEnvironment()

    for {
        fmt.Fprintf(out, PROMPT)
        scanned := scanner.Scan()
        if !scanned {
            return
```

```
        }

        line := scanner.Text()
        l := lexer.New(line)
        p := parser.New(l)

        program := p.ParseProgram()
        if len(p.Errors()) != 0 {
            printParserErrors(out, p.Errors())
            continue
        }

        evaluated := evaluator.Eval(program, env)
        if evaluated != nil {
            io.WriteString(out, evaluated.Inspect())
            io.WriteString(out, "\n")
        }
    }
}
```

여기까지가 전편에서 마무리 지은 Monkey의 모습이다.

그러고 나서 반년 후, 〈잃어버린 챕터: Monkey 매크로 시스템〉[1]이 급부상했고 독자들에게 Monkey에서 Monkey 매크로로 프로그래밍하는 방법을 제시했다. 그러나 이 책에서는 매크로 시스템이 등장하지 않을 예정이다. 사실 매크로가 마치 없었던 것처럼 전편의 끝으로 되돌아갔다. 뭐 나쁘지 않다. 왜냐하면 인터프리터도 꽤 잘 만들었기 때문이다.

Monkey 언어는 우리가 원하는 대로 동작했다. 그리고 구현체는 이해하기도 쉬웠고 확장하기도 쉬웠다. 그러면 자연스레 "잘 동작하는데 왜 바꾸지?"라는 질문이 생길 수 있다. 그냥 Monkey를 있는 그대로 두면 안 되는 걸까?

이유를 말하자면, 우리의 목적은 학습이다. Monkey 언어는 여전히 우리에게 가르쳐줄 것이 많다. 《인터프리터 in Go》의 목표는 우리가 매일 작업하는 프로그래밍 언어 구현체가 무엇인지 더 학습하는 것

1 (옮긴이) 전편이 출간되고 반년 뒤에 추가된 장으로 매크로 시스템을 다룬다. 모든 내용을 무료로 다운로드할 수 있다(참고 〈The Lost Chapter: A Macro System For Monkey〉, *https://interpreterbook.com/lost*).

이었다. 그리고 실제로 학습을 했다. '현실 세계' 프로그래밍 언어들도 Monkey 언어와 비슷한 구현체로부터 시작했다. 우리가 Monkey를 만들면서 습득한 지식이 현실 세계 프로그래밍 언어 구현체와 그 기원을 이해하는 데 도움을 줬다는 뜻이다.

그러나 언어는 성장하고 성숙해진다. 상용 환경 그리고 성능과 언어 기능에 대한 요구 사항이 많아지면서, 그 언어가 가진 아키텍처와 구현이 바뀌기도 한다. 이런 변화를 겪음에 따라 언어 구현체는 Monkey 언어와의 동질성(성능과 상용을 전혀 고려하지 않은 초기 구현체의 모습)을 잃어버리게 된다.

이렇게 성장을 끝낸 언어와 Monkey 언어 간의 격차가 우리 Monkey 구현체가 갖는 가장 큰 단점이 된다. 다시 말해 Monkey 언어 아키텍처를 실제 프로그래밍 언어 아키텍처와 비교하면, 장난감 자동차와 스포츠카를 비교하는 정도의 차이가 난다. 물론, 바퀴가 네 개 달려 있고 좌석이 있어 바퀴를 조작하는 기초적인 방법을 학습하는 데는 도움이 되겠지만, 엔진이 없다는 점은 무시하기 어렵다.

이번 책에서 이런 격차를 좁혀보고자 한다. 우리 장난감 자동차에 엔진 비슷한 걸 달아볼 예정이다.

향후 계획

우리가 전편에서 작성한 트리 순회/즉시 평가 인터프리터를, 바이트코드 컴파일러와 바이트코드를 실행하는 가상 머신으로 바꿔보려 한다.

이런 아키텍처는 만들어볼 대상으로 아주 재밌기도 하지만, 가장 흔한 인터프리터 아키텍처이기도 하다. Ruby, Lua, Python, Perl, Guile, 그리고 JavaScript 구현체 중 몇몇과 그 밖에 여러 프로그래밍 언어가 바이트코드를 사용하는 아키텍처로 만들어져 있다. 심지어 그 유명한 자바 가상 머신(Java Virtual Machine)도 바이트코드를 평가하도록 만들어져 있다. 바이트코드 컴파일러와 바이트코드 가상 머신은 정말 쉽게 찾아볼 수 있으며, 이렇게 많은 데에는 그럴 만한 이유가 있다.

컴파일러에서 바이트코드를 만들어서 가상 머신에게 전달하는, 즉 새

로운 추상화 계층을 도입함으로써 시스템을 더욱 모듈화하는 장점과는 별개로, 바이트코드 아키텍처가 핵심적으로 지향하는 바는 성능이다. 다시 말해 바이트코드 인터프리터는 빠르다.

성능을 이야기하는데 수치를 빼놓을 수는 없다. 이 책을 마무리할 때쯤에는 전편에서 작성한 구현체보다 세 배나 빠른 구현체를 만들게 된다. 결과만 미리 확인해보자.

```
$ ./fibonacci -engine=eval
engine=eval, result=9227465, duration=27.204277379s

$ ./fibonacci -engine=vm
engine=vm, result=9227465, duration=8.876222455s
```

보다시피 세 배나 더 빠르다. 저수준에서 복잡한 최적화를 거치지 않았음에도 말이다. 훌륭하지 않은가? 지금쯤이면 우리 모두 코드를 써 내려갈 준비가 됐으리라. 그러면 본격적으로 시작하기에 앞서 책에 있는 코드를 실제로 따라 하기 위해 필요한 몇 가지 사전 지식을 설명하고자 한다.

책 활용법

전편과 마찬가지로 이 책에서도 지시사항은 그리 많지 않다. 처음부터 끝까지 주의 깊게 읽으며 제시된 코드를 읽고 가볍게 따라 한다면, 주어진 내용을 거의 다 활용할 수 있다.

이 책은 '실용적으로' 쓰여 있다. 코드를 작성하고 뭔가를 실제로 만들어낸다. 만약 여러분이 프로그래밍 언어 구성에 대한 이론을 탐구하길 원한다면 그냥 일반적인 대학 전공책을 읽는 게 더 나은 선택일지도 모른다. 그런데 그렇다고 해서 이 책에서 배울 수 있는 게 없다는 뜻은 아니다. 나는 여러분을 안내하면서 각 요소가 무엇이며, 요소끼리 어떻게 맞물려 나가는지 최대한 설명한다. 다만 컴파일러를 다루는 일반 대학 전공책과 같은 방식으로 다루지 않을 뿐이다. 그리고 그게 내가 의도한 바이기도 하다.

전편과 마찬가지로 이번 책에도 code라는 폴더를 제공한다. 아래의 링크에서 내려받으면 된다.

https://compilerbook.com/wacig_code_1.2.zip[2]

code 폴더의 하위 폴더에는 각 장에서 우리가 작성할 코드가 작성되어 있다. 각각의 코드는 각 장을 끝났을 때의 모습이라고 생각하면 된다. 따라서 만약 여러분이 중간에 막히더라도 코드를 보면서 도움을 받으면 된다.

코드 폴더의 하위 폴더에는 각 장별 폴더 이외에도 00이라는 폴더를 볼 수 있을 텐데, 이 폴더는 전편에서는 없던 폴더이다. 이 책은 백지에서 출발하지 않는다. 전편에서 작성한 코드를 바탕으로 시작한다. 즉, 00 폴더는 이번 편에 작성된 어떤 장과 대응되는 게 아니라, 전편에서 완성한 코드베이스를 담고 있다는 뜻이다. 그리고 전편 마지막 코드베이스를 기준으로 하므로, 나중에 추가된 장에서 다루는 매크로 시스템은 포함하지 않았다. 만약 여러분이 매크로를 정말 좋아한다면, 매크로를 확장해 넣는 게 그리 어렵지는 않을 것이다.

각각의 폴더에 포함된 코드가 책에서 초점을 맞추고 있는 내용이라고 보면 된다. 대부분의 코드는 직접 다루겠지만, 가끔 위치만 언급하고 실제로 지면에서 싣지 않을 수도 있다. 왜냐하면 대부분 앞서 본 내용을 그저 반복할 뿐인데 지면을 너무 많이 차지하기 때문이다.

code 폴더 얘기는 이쯤 하기로 하고, 우리가 사용하게 될 툴(tool)에 관해 이야기해보자. 좋은 소식이 있다면 우리는 그렇게 많은 도구를 쓰지 않는다. 정확히 말하자면, 코드 편집기와 Go 언어가 설치되어 있으면 충분하다. Go 버전은 최소한 1.10 이상은 사용해야 한다. 왜냐하면 1.10이 내가 코드를 작성할 때 사용한 Go 언어 버전이고, Go 1.8과 1.9에 새로 도입된 기능을 몇 개 써보려는 의도도 있기 때문이다.

2 (옮긴이) 저자가 제공하는 사이트로 url을 입력하면 바로 다운로드할 수 있다.

그리고 만약 여러분이 1.13 버전 이상을 사용하지 않는다면, direnv[3]를 사용해서 code 폴더를 여는 걸 권장한다. direnv는 .envrc 파일을 통해 셸 환경을 바꿔주기 때문이다. cd 명령어로 어떤 폴더에 들어갔을 때, direnv는 해당 폴더가 .envrc 파일을 포함하고 있다면 이 파일을 실행한다. 각각의 code 폴더 아래 포함된 각 하위 폴더에는 .envrc 파일이 포함되어 있어서 GOPATH를 해당 하위 폴더에 맞게 설정해준다. 따라서 여러분은 단순히 하위 폴더에 cd 명령으로 들어가서 코드를 실행하면 된다.

그러나 여러분이 1.13 버전 이상을 사용하고 있다면 GOPATH를 설정하지 않아도 된다. 왜냐하면 code 폴더가 go.mod 파일을 포함하고 있어서 go 명령어로 별도의 설정 없이 쉽게 실행할 수 있다.

이제 이 책에서 여러분이 직접 코드로 작성해야 할 내용에 대해서는 모두 다룬 듯하다. 그러니 이제부터 여러분은 책을 읽고 코드를 따라가면서, 무엇보다 즐기기를 바란다!

3 (옮긴이) direnv는 기존 셸에, 현재 디렉터리의 환경변수를 로드/언로드하기 위해 사용하는 프로그램이다. direnv는 .envrc 파일을 현재 디렉터리와 부모 디렉터리에서 찾아서 만약 파일이 존재한다면, .envrc의 내용을 셸에 로드하도록 만든 프로그램이다.

옮긴이의 덧붙임

이 책은 어느 정도 프로그래밍에 대한 배경지식을 전제하고 있습니다. 저자는 Go를 처음 접하는 사람일지라도 책의 코드를 이해하는 데 문제가 없을 것이라고 말하고 있습니다. 맞는 말이지만 다른 언어 사용자에게는 어느 정도 걸림돌이 있을 것이란 생각이 듭니다. 특히 최초 환경 설정과 Go로 코드를 작성하는 과정에서 최대한 어려움이 없기를 바라는 마음에서 내용을 조금 덧붙입니다.

Go 언어 설치

공식 설치 페이지(https://golang.org/doc/install)를 방문하여 운영체제에 맞게 Go 언어를 설치합니다.

개발 환경

저는 Visual Studio Code와 Google Go 팀에서 개발한 The VS Code Go Extension 플러그인을 설치하고 Go 코드를 작성했습니다. 이것 하나로 부족함 없이 시작할 수 있습니다.

단, 저장 시에 자동으로 `import`가 업데이트되고 규격화가 일어나는데, 이때 `monkey/token`이 아닌 `go/token`이라든가, `monkey/ast`가 아닌 `go/ast`로 의도치 않게 업데이트될 수 있으니 주의하기 바랍니다.

그 밖에 상세 내용은 Go 언어 공식 블로그의 글 Gopls on by default in the VS Code Go extension(https://blog.golang.org/gopls-vscode-go)을 참조하기 바랍니다.

코드 작성

처음에는 빈 프로젝트 디렉터리에서 시작할 텐데, 코드를 작성하기에 앞서 프로젝트 디렉터리 안에서 `go mod init monkey`를 셸에서 입력해 monkey 모듈을 선언하기 바랍니다. `go mod init`은 현재 디렉터리에 새로운 `go.mod` 파일을 생성하고, 현재 디렉터리에 루트를 둔 새 모듈을 만듭니다.

줄이며...

그 밖에 문법적 특징에 대해서는 굳이 설명하지 않겠습니다. 제가 여기서 짧게 다루는 것보다 직접 읽어서 익히는 편이 더 나을 것이라 판단됩니다.

그리고 당장 앞부분만 봐도 알겠지만 이 책은 테스트 주도 개발(Test Driven Development, TDD)에 입각해서 쓰인 책입니다. 때문에 TDD 맥락에서 설명하고 있는 내용이 자주 등장합니다. 그렇지만 어려운 내용은 없으며, 설령 여러분이 TDD를 잘 모르거나 익숙지 않더라도 아래 한 문장만 기억하면 됩니다.

> "실패하는 테스트를 먼저 작성하고, 테스트를 통과하게 만들며, 점진적으로 리팩터링한다."

제 생각에 준비는 끝난 것 같습니다. 저와 마찬가지로 여러분도 이 책을 읽고 따라 하면서 즐거운 시간을 갖기 바랍니다.

1장

Writing A Compiler In Go

컴파일러와 가상 머신

많은 프로그래머가 '컴파일러'라는 단어에 어느 정도 위압감을 느낀다. 설령 위압감은 아니더라도 컴파일러와 컴파일러가 하는 일에서 불가사의하고 신비로운 분위기를 느낀다는 사실은 부정하기 어렵다. 컴파일러는 살아있는 인간이라면 누구도 읽고 쓸 수 없을 것 같은 '머신 코드(machine code)'를 만들어낸다. 또한 컴파일러는 최적화라는 마법을 부릴 수 있어, 어떤 이유에선지 코드가 더 빠르게 실행되도록 만든다. 컴파일러는 실행에 오랜 시간이 소요되기도 하는데, 몇 분 또는 몇십 분이 걸릴 때도 있다. 소문에 따르면 몇 시간이 걸릴 때도 있다고 한다. 이렇게 오래 걸린다면, 컴파일러는 틀림없이 뭔가 특별한 일을 하고 있지 않을까?

소문에 따르면, 컴파일러는 엄청나게 크고, 엄청나게 복잡하다. 사실 컴파일러는 여태까지 만들어진 소프트웨어 중에서도 가장 복잡한 소프트웨어 프로젝트로 꼽힌다. 숫자로 증명할 수도 있다. 예를 들어, LLVM[1]과 Clang[2] 프로젝트는 코드가 약 300만 줄짜리 프로젝트이다.

1 (옮긴이) LLVM 프로젝트는 재사용이 가능한 모듈화된 컴파일러 툴 체인이다. 이름과는 다르게 가상 머신하고는 관계가 거의 없다. 원래는 Low Level Virtual Machine을 줄여서 LLVM이라고 썼으나, 프로젝트가 커지면서 혼선이 생기지 않게 공식적으로 약어가 아니라고 공표했다. 따라서 LLVM은 약자가 아니라 그 자체로 프로젝트 이름이다(참고 *https://llvm.org/*).

2 (옮긴이) Clang(클랭): LLVM의 서브 프로젝트이자 프런트엔드다.

GNU Compiler Collection(GCC)[3]는 훨씬 더 큰 프로젝트이고, 코드가 무려 1500만 줄로 짜여 있다.

이런 엄청난 숫자를 보고도 코드 편집기를 열어서 "하나 만들어보지, 뭘!"이라고 말할 수 있는 사람은 그리 많지 않다. 그들도 컴파일러가 얼마나 큰 프로젝트인지 본다면, 반나절 만에 컴파일러를 만들 것이란 생각이 자연스레 사라질 것이다.

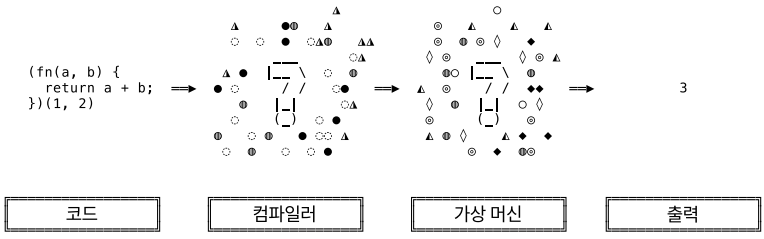

그림 1-1

프로그래머들이 가상 머신(Virtual Machines)에 대해 갖는 느낌 역시 비슷하다. 가상 머신은 소프트웨어 개발의 낮은 층인 어둡고 깊숙한 곳에서 배회하는, 빛이 있는 곳에서는 보기 힘들고 이해하기도 어려운 미지의 생물 같은 느낌을 준다. 가상 머신을 둘러싼 소문과 추측 역시 무성하긴 마찬가지다. 어떤 사람은 가상 머신이 컴파일러와 깊은 관계가 있다고 하고, 어떤 사람은 우리가 사용하는 프로그래밍 언어가 사실은 가상 머신이라고 주장한다. 또 어떤 사람은 가상 머신이 있어야 운영체제 위에서 다른 운영체제를 구동할 수 있다고 말한다.

모든 소문이 사실이라고 한들, 그다지 유용한 정보는 아닌 듯하다.

내가 말하고 싶은 것은 이렇다. 핵심을 말하자면, 컴파일러와 가상 머신 모두 추상 개념(패턴)일 뿐이다. 즉 '웹 서버(web server)', '인터프리터(interpreter)'와 같이 크기나 복잡도에 따라 다양한 구현체가 있는 개념일 뿐이다. 따라서. GCC같이 큰 프로젝트를 보고 기가 죽어 컴파일

3 (옮긴이) GCC: GNU Compiler Collection은 원래는 GNU 운영체제용으로 만들어진 컴파일러이다. C, C++, Objective-C, Fortran, Ada, Go, D가 사용할 컴파일러 프런트엔드를 담고 있으며, 앞서 언급한 언어 라이브러리도 포함한다(참고 *https://gcc.gnu.org/*).

러를 못 만들겠다고 말한다면, GitHub 같은 웹사이트를 보고 웹사이트 역시 만들 수 없다고 말해야 한다.

물론, 가상 머신을 사용할 컴파일러를 만드는 게 쉬운 일은 아니다. 그러나 악명이 높기는 해도, 만들지 못할 정도는 결코 아니다. 컴파일러와 가상 머신이 본질적으로 무엇인지 그 핵심 개념을 더 깊이 이해하게 되면, 여러분은 정말로 반나절 만에 컴파일러를 만들 수 있다.

더 깊이 이해하기 위한 첫 단계로, 가장 먼저 '컴파일한다(compiling)'는 게 무엇을 의미하는지 알아보자.

컴파일러

"컴파일러의 이름을 적어보시오."

위와 같은 문제가 나왔다면, 아마도 여러분은 망설이지 않고, GCC, Clang, Go 컴파일러 같은 이름을 적었을 것이다. 뭐가 됐든, 그 이름은 프로그래밍 언어를 위한 컴파일러일 것이다. 그리고 프로그래밍 언어를 위한 컴파일러는 보통 실행 프로그램(executables)을 만든다. 나라고 별다른 답을 갖고 있지 않다. 왜냐하면 우리 모두 '컴파일러'라는 단어를 프로그래밍 언어 컴파일러와 연관 지어 생각하기 때문이다.

한편, 컴파일러는 크기와 종류도 아주 다양할 뿐만 아니라, 컴파일할 대상이 프로그래밍 언어에 국한되지 않으며, 정규식(regular expressions), 데이터베이스 쿼리, 심지어 HTML 템플릿까지도 컴파일한다. 우리는 모두 일상에서 별생각 없이 이미 컴파일러를 한두 개쯤 사용하고 있다. 이렇게 컴파일러가 넓게 사용되는 이유는, 컴파일러라는 단어가 생각보다 훨씬 광의적이기 때문이다. 아래는 위키피디아가 정의하는 컴파일러이다.

> 컴파일러는 소스 프로그래밍 언어로 작성된 컴퓨터 코드를 목적 프로그래밍 언어로 변환하는 컴퓨터 소프트웨어이다. 컴파일러는 번역기(translator)의 한 유형으로 주로 컴퓨터 같은 디지털 장치를 지원하는 데

> 쓰인다. 컴파일러라는 이름은 고수준 프로그래밍 언어로 쓰인 소스코드를 저수준 프로그래밍 언어(어셈블리어, 목적 코드, 머신 코드)로 바꿔서 실행 프로그램을 만들어내는 프로그램을 가리키는 데 쓰인다.

컴파일러가 번역기(translator)라는 말은 너무 불명확하다. 고수준 언어를 번역해서 실행 프로그램으로 만드는 컴파일러가 특수한 유형의 컴파일러라는 말인가? 전혀 직관적이지 않다. 실행 프로그램을 만들어내는 작업이 원래 '컴파일러의 일'이지 않은가? GCC, Clang, Go 컴파일러가 하는 것처럼 말이다. 그렇다면, 위키피디아의 정의는 '실행 프로그램을 만들어낸다'는 정의가 첫 번째 줄에 와야 하지 않을까? 어떻게 이렇게까지 본질에서 벗어나 있을까?

이런 의문에 대한 해법은 다른 방향에서 풀어내야 한다. 컴퓨터가 직접 인식하는 언어로 된 소스코드가 아니라면 어떻게 실행할 수 있을까? 그래서, "네이티브 코드로 컴파일한다"라는 말은 "머신 코드(machine code)로 컴파일한다"라는 말과 같다. 실행 프로그램을 만든다는 것은 정말로 '소스코드를 번역'하는 다양한 방법 중 하나일 뿐이다.

알다시피, 컴파일러는 근본적으로 번역(translation)을 다룬다. 왜냐하면 컴파일러가 번역하는 방식이 곧 '프로그래밍 언어를 어떻게 구현하는지' 정의하기 때문이다.

위 문장이 말하는 바를 곱씹어보자. 프로그래밍은 컴퓨터에 명령을 내리는 행위이다. 프로그래머들은 컴퓨터가 이해할 수 있는 명령어(instructions)를 프로그래밍 언어로 작성한다.

다른 언어를 사용하면 의미가 없다. 프로그래밍 언어를 구현한다는 말은 컴퓨터가 이해할 수 있도록 만든다는 뜻이다. 컴퓨터가 이해하게 만드는 데는 두 가지 방법이 있다. 컴퓨터를 위한 언어로 즉시(on-the-fly) 해석(interpret)하거나, 컴퓨터가 이미 이해하고 있는 또 다른 언어로 번역(translate)하면 된다.

우리말을 할 줄 모르는 외국인 친구를 도와준다고 상상해보자. 우리는 우리말을 듣고 머릿속에서 외국어로 번역하고, 번역한 그대로를 친

구에게 전달한다. 번역한 내용을 종이에 적어서 줄 수도 있다. 외국인 친구는 번역한 내용을 읽고 자기식대로 이해하면 된다. 이 이야기에서는 우리가 인터프리터 혹은 컴파일러 같은 역할을 한 것이 된다.

위 이야기는 마치 인터프리터와 컴파일러가 대비되는 것처럼 들릴 수 있다. 그러나 인터프리터와 컴파일러는 접근하는 방식이 다를 뿐이지, 결과물을 만드는 방식에서는 공통점이 아주 많다. 둘 다 프런트엔드(frontend)를 가지는데, 프런트엔드에서는 소스 언어로 작성된 소스코드를 읽어 들여 특정 데이터 구조로 변환한다. 또한, 인터프리터와 컴파일러 프런트엔드 양쪽 모두 렉서(lexer)와 파서(parser)로 구성되고, 렉서, 파서는 함께 추상구문트리(Abstract Syntax Tree, 이하 AST)를 만든다. 따라서 앞 단계(front part)에서는 둘의 유사점이 아주 많다. 한편, 프런트엔드를 거치고 나면, 둘 다 AST를 순회하게 되는데, 여기서부터 컴파일러와 인터프리터가 하는 일이 달라진다.

우린 이미 인터프리터를 만들어봤기에[4], AST를 순회한다는 게 결국은 AST를 평가(evaluation)하는 일이라는 것을 잘 알고 있다. 그리고 나서 인터프리터는 AST에 부호화된 명령어를 실행한다. AST에서 어떤 노드가 소스 언어 명령문(statement)인 `puts("Hello World!")`라면, 인터프리터는 노드를 평가할 때, `"Hello World!"`라는 문장을 출력할 것이다.

한편, 컴파일러라면 아무것도 출력하지 않을 것이다. 컴파일러는 목적 언어(target language)로 된 소스코드를 만든다. 만들어진 소스코드는 소스 언어 `puts("Hello World!")`로 표현된 개체에 대응하는 목적 언어 개체를 갖고 있다. 그리고 나서 결과 코드를 컴퓨터가 실행하고, `"Hello World"`가 스크린에 나타난다.

이제부터 상황이 아주 재밌게 흘러간다. 컴파일러는 어떤 목적 언어로 소스코드를 만들까? 컴퓨터가 이해하는 언어가 무엇일까? 그럼, 컴파일러는 어떻게 컴퓨터가 이해하는 언어로 코드를 생성할까? 문자열

4 (옮긴이) 전편 《인터프리터 in Go》에서 우리는 인터프리터를 만들었다. 최종 완성된 인터프리터는 코드 폴더의 00 폴더에 있다. 이 책에서 컴파일러를 작성하는 데 큰 문제는 없지만, 인터프리터의 작동 원리를 이해하고자 한다면 전편을 꼭 참고하길 바란다.

로 만들어야 할까? 아니면 이진 형식(binary format)으로 만들어낼까? 파일로 만들어야 할까? 아니면 메모리에? 정작 제일 중요한 질문이 빠져있다. "목적 언어로 만들어내는 대상이 무엇일까?" 만약 목적 언어가 `puts`에 대응하는 개체를 갖고 있지 않다면, 컴파일러는 대신에 무엇을 만들어야 할까?

일반화해서 얘기하자면, 앞서 언급한 모든 질문에 단 한 문장으로 답할 수 있다. 일반화하기 어려운 소프트웨어 개발 분야에서 거의 모든 상황에 답할 수 있는 보편적 진리를 담은 문장이다. "상황에 따라 따르다."

실망스러운 답을 줘서 정말 미안하지만, 변수가 매우 많고 요구 사항도 다양하다. 예를 들면, 소스 언어, 목적 언어가 구동될 머신 아키텍처, 결과물을 바로 실행할지 아니면 컴파일할지, 번역(interpret)할지, 출력 결과가 얼마나 빠르게 구동되는지, 컴파일러가 얼마나 빨리 일을 끝내는지, 생성된 소스코드가 가질 크기는 얼마나 되어야 할지, 컴파일러에 허용되는 메모리는 얼마나 되는지, 결과 프로그램에 허용되는 메모리는 얼마나 될지 등등. 나열한 모든 것에 따라서 만들어야 할 대상이 달라진다.

컴파일러는 너무나도 다양하기 때문에, "컴파일러 아키텍처는 이렇다!"라고 보편적으로 정의하기 어렵다. 그러므로 상세한 내용은 잠시 미뤄두고 여러 컴파일러가 공유하는 아키텍처를 생각해보면서 윤곽을 잡아보자.

[그림 1-2]는 머신 코드로 번역된 소스코드가 갖는 생명 주기(life cycle)를 보여준다. 어떤 일이 일어나는지 설명해보겠다.

먼저 렉서가 소스코드를 토큰화(tokenize)하고 파서는 토큰을 파싱한다. 렉서와 파서는 인터프리터를 작성하면서 꽤 익숙해졌을 것이다. 렉서와 파서를 프런트엔드(frontend)라고 부른다. 프런트엔드를 거치고 나면, 텍스트였던 소스코드는 AST로 변환된다.

프런트엔드 다음에는 '옵티마이저(optimizer, 이것 역시 컴파일러라 부르기도 한다)'라는 컴포넌트가 AST를 또 다른 내부 표현(internal representation, 이하 IR)으로 변환한다. 옵티마이저가 만들어낸 IR은 다

양한 형식을 갖는다. 형태가 다른 구문 트리(syntax tree)일 수도 있고, 이진 형식이거나 단순히 텍스트 형식일 수도 있다. 옵티마이저가 별도

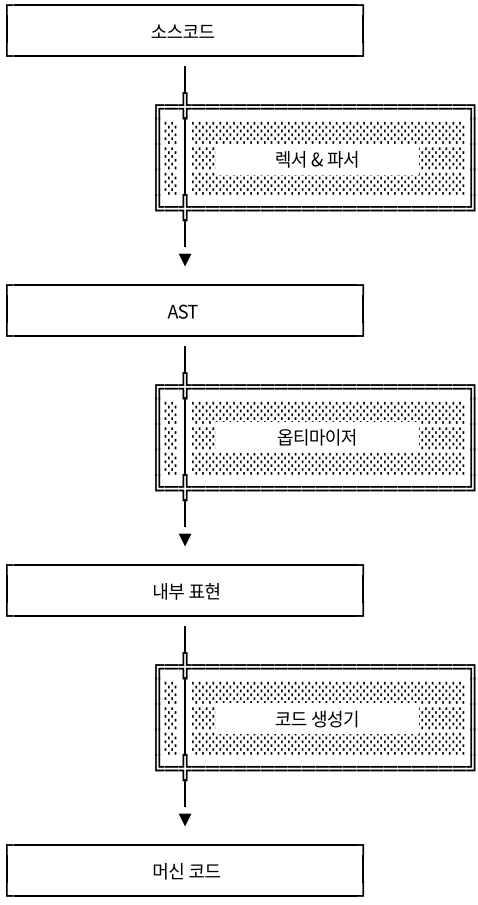

그림 1-2

로 IR을 만들어내는 이유는 다양하다. 그러나 가장 중요한 이유는 별도로 만들어낸 IR이 (기존 AST보다) 최적화 및 목적 언어로 변환하는 과정에 더 안성맞춤이기 때문이다.

옵티마이저가 만들어낸 새로운 IR은 예컨대 아래와 같은 최적화 과정을 거치게 된다.

- 불필요한 코드 제거[5]
- 단순한 산술 연산을 미리 계산
- 반복문 몸체에 있을 필요가 없는 코드를 밖으로 빼기[6]

위에서 열거한 내용 외에도 수많은 최적화 기법을 활용할 수 있다.

마지막으로 '코드 제너레이터(code generator)'가 목적 언어로 된 코드를 만든다. 코드 제너레이터는 백엔드(backend)라고 부르기도 한다. 코드 제너레이터에서 실제 컴파일이 일어난다. 마침내 코드가 파일 시스템에 접근한다. 파일로 결과물을 만들고, 그 결과물을 실행하면 컴퓨터가 원본 소스코드의 지시대로 명령을 수행하는 것을 볼 수 있게 된다.

컴파일러가 어떻게 동작하는지 최대한 간단하게 설명해보았다. 이렇게 간단하게 설명한 내용조차 수천 가지로 변형이 가능하다. 예를 들어, 옵티마이저가 IR을 여러 '패스(pass)'[7]에 걸쳐서 처리할 수 있다. 자세히 설명하자면, 옵티마이저가 IR을 여러 차례 순회하면서 순회마다 다른 형태로 최적화할 수 있다. 예를 들어 특정 패스(pass)에서는 데드 코드(dead code)를 제거하고, 다른 패스(pass)에서는 함수 호출을 인라이닝(inlining)[8]할 수 있다. 혹은 IR을 최적화하지 않고 목적 언어로 된 소스코드만 최적화할 수도 있다. AST만 최적화하기도 하고, AST, IR 모두를 최적화할 수도 있다. 때로는 AST 말고는 다른 IR이 애초부터 없을 수도 있다. 머신 코드를 출력하지 않고, 어셈블리어나 다른 고수준 언어를 결과물로 만들 수도 있다. 아니면 백엔드를 여러 개 구현해 아키텍처별로 머신 코드를 별도로 생성할 수도 있다. 언급한 모든 것을 어떻게 사용하느냐에 따라 달라진다.

또한, 컴파일러는 명령행 도구(command line tool)일 필요가 없다. gcc나 go 컴파일러같이, 소스코드를 읽어서 파일로 코드를 만들어내는

5 (옮긴이) dead-code elimination, DCE: 컴파일 결과에 영향을 주지 못하는 코드를 소거하는 최적화 기법

6 (옮긴이) code motion: 루프가 몇 번 반복되어도 루프 시작 전과 동일한 표현식이나 문장을 loop-invariant라고 하는데, loop-invariant를 루프 바깥으로 옮기는 최적화를 말한다.

7 (옮긴이) pass(패스): 컴파일러에서 여러 단계(phase)를 그룹화한 단위

8 (옮긴이) inlining(인라이닝): 함수 호출부를 함수 몸체로 변환하는 최적화 과정

형태일 수 있다. 컴파일러는 AST를 받아 단순히 문자열을 반환하는 함수일 수도 있다. 이것 역시 컴파일러라고 말할 수 있다. 컴파일러는 수백 줄로 작성할 수도, 수백만 줄로 작성할 수도 있다.

컴파일러가 이렇게 다양하지만, 모든 코드가 공통으로 공유하는 가장 기초적인 착안점은 변환(translation)이다. 컴파일러는 한 언어로 소스코드를 받아서 또 다른 언어로 소스코드를 만든다. 이후에 일어날 일은 상황에 따라 다르지만, 대부분 목적 언어에 따라 달라진다. 목적 언어가 갖춰야할 기능과 구동될 머신에 따라 컴파일러의 설계가 결정된다.

만약에 목적 언어를 골라야 할 필요가 없다면 어떨까? 그냥 목적 언어를 하나 새로 만든다면 어떨까? 새로 만들고 끝내는 게 아니라 새로 만들 언어를 실행할 머신도 만들어보는 것은 어떨까?

가상 머신과 실제 머신

'가상 머신(virtual machine)'이라는 단어를 들으면 VMWare나 VirtualBox 같은 소프트웨어를 떠올리게 마련이다. 이런 프로그램은 컴퓨터를 흉내 낸다. 디스크 드라이브, 하드 드라이브, 그래픽 카드 역시 모방해 만들어졌다. 모방해서 만들어낸 컴퓨터 안에서, (호스트 운영체제와) 다른 운영체제를 구동할 수 있게 해준다. 이런 소프트웨어를 가리켜 가상 머신이라고 한다. 그러나 우리가 이제부터 얘기할 가상 머신은 전혀 다른 유형의 가상 머신이다.

우리가 얘기할 (나중에는 만들어볼) 가상 머신은 프로그래밍 언어를 구현하기 위해 사용된다. 가상 머신은 함수 몇 개로 만들기도 하고, 때로는 가상 머신이 다른 모듈을 만드는 데 사용되기도 한다. 클래스와 객체 컬렉션 형태일 때도 있다. 가상 머신 역시 형태를 꼭 집어 말하기 어렵다. 그러나 형태는 중요치 않다. 우리가 만들 가상 머신이 기존 머신을 모방하지 않는다는 게 중요하다. 우리가 만들 가상 머신은, 어떤 것을 흉내 내는 머신이 아니라 자체가 (소프트웨어로 만든) 머신(machine)이다.

'가상(virtual)'이란 수식어가 붙는 이유는 머신이 하드웨어가 아닌 소프트웨어로 만들어져 있기 때문이다. 즉, 완전히 추상적인 개체이다. '머신(machine)'이란 단어가 가상 머신이 하는 일을 설명해준다. 가상 머신이라는 소프트웨어는 마치 기계처럼 동작한다. 그리고 여기서 기계(machine)가 지칭하는 대상은 일반적인 기계가 아니다. 하드웨어 관점에서 가상 머신과 대응되는 기계, 즉 '컴퓨터(computer)'가 하는 행위를 모방한다.

가상 머신이 컴퓨터를 모방하므로, 가상 머신을 이해하고 만들려면 실제 컴퓨터가 어떻게 동작하는지 알아야 한다.

실제 머신

"그래서 컴퓨터는 어떻게 동작하나요?"

어려운 질문이지만, A4 용지 한 장 분량으로 5분 안에 읽을 수 있게 정리할 수 있다. 여러분이 얼마나 책을 빨리 읽을지도 모르겠고, 그렇다고 종이에 직접 그림을 그려가며 설명할 수도 없는 노릇이니 그냥 설명해 보겠다.

우리 주변의 거의 모든 컴퓨터는 폰 노이만 아키텍처(Von Neumann architecture)[9]로 만들어졌다. 폰 노이만 아키텍처는 놀라울 만큼 적은 요소만으로 완전하게 동작하는 컴퓨터를 만드는 법을 기술한다.

폰 노이만 모델에서 컴퓨터는 핵심적인 장치 두 개로 나뉜다. 두 개의 핵심 장치는 산술 논리 장치(arithmetic logic unit, ALU)와 다중 프로세서 레지스터(multiple processor registers)로 이루어진 '처리 장치(processing unit)'와 명령어 레지스터(instruction register)와 프로그램 카운터(program counter)로 이루어진 '제어 장치(control unit)'로 구

9 (옮긴이) 폰 노이만 아키텍처(von Neuman architecture): 1945년 에드박 보고서 초안(First Draft of a Report on the EDVAC)에서 존 폰 노이만(John Von Neumann)을 포함한 여러 사람의 설명에 기반한 컴퓨터 아키텍처이다(참고 *https://en.wikipedia.org/wiki/Von_Neumann_architecture*).

성된다. 둘을 합쳐서 '중앙 처리 장치(central processing unit)', 줄여서 'CPU'라 부른다. 그리고 컴퓨터는 메모리(RAM), 대용량 저장 장치(하드 디스크 등), 입출력 장치(키보드와 모니터 같은 디스플레이 장치)도 포함하는 개념이다.

[그림 1-3]은 CPU, 메모리, 저장 장치, 입출력 장치로 컴퓨터가 무엇인지 간단하게 묘사해본 그림이다.

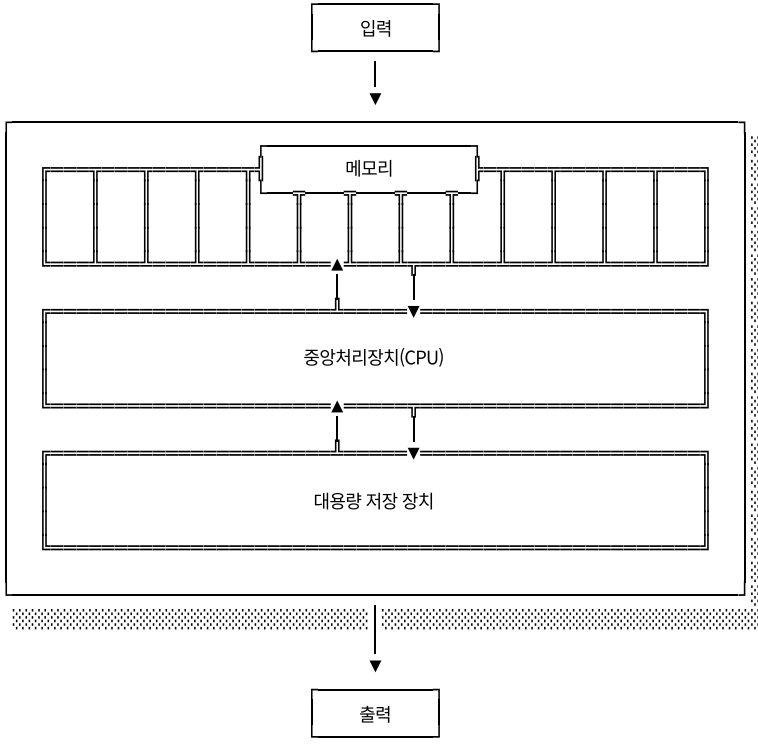

그림 1-3

컴퓨터를 켜면 CPU는 아래와 같이 동작한다.

1. 메모리에서 명령어를 인출(fetch)한다.
   프로그램 카운터가 CPU에게 다음 명령어가 메모리상에서 어디에 있는지 알려준다.

2. 명령어를 부호화(decode)한다.
   다음에 수행할 동작을 알아내기 위함이다.
3. 명령어를 실행한다.
   명령어를 실행한다는 말은, 레지스터가 기록하고 있는 내용을 변경한다는 뜻일 수도 있고, 레지스터에 있는 데이터를 메모리로 이전한다는 뜻일 수도 있다. 그 밖에도 메모리에서 메모리로 데이터를 이동시킬 수도, 출력물을 만들 수도, 입력을 읽어 들일 수도 있다.

…그리고 다시 1로 '되돌아(goto)'간다.

위 세 단계를 '인출-복호화-실행 주기(fetch-decode-execute cycle)'라고 말하며, 명령 주기(instruction cycle)라고도 한다. 아래와 같은 문장에서 사용되는 사이클(cycle)이 곧 명령어 주기를 말한다.

"컴퓨터 클록 속도는 초당 사이클(cycle) 반복 횟수로 표현한다. 예를 들어 500MHz 같으면 초당 500만 번 사이클이 돈다는 뜻이다."

"여기서 CPU 사이클이 낭비되고 있어."

컴퓨터가 어떻게 동작하는지 꽤 간략하면서도 쉽게 이해되도록 설명한 것 같다. 그러나 우리는 좀 더 단순하게 만들 필요가 있다. 이 책에서는 대용량 저장 장치 같은 구성 요소를 고려하지 않을 것이고, 입출력(I/O)이 어떻게 동작하는지 정도만 맛보기로 다루려 한다. 우리는 CPU와 메모리가 어떻게 상호작용하는지에 관심을 두어야 한다. 따라서 우리 관심사에 더 집중하기로 하고, 하드디스크나 디스플레이 장치 같은 요소들이 어떻게 동작하는지는 신경 쓰지 말자.

CPU는 메모리 주소를 어떻게 알아낼까? 달리 말하면, CPU는 메모리에 접근할 때, 어디에 저장하고 어디에서 꺼내올지 어떻게 알아낼까? 이 질문에 답하는 것부터 시작해서 CPU와 메모리가 어떻게 상호작용하는지 알아보자.

첫 번째 힌트는 CPU가 명령어(instruction)를 인출하는 방법에서 찾아

보자. 프로그램 카운터(program counter)는 다음으로 인출할 명령어를 파악하고 있는데, '카운터(counter)'는 단어의 의미 그대로 사용된다. 컴퓨터는 단순하게 카운터가 가진 숫자값으로 메모리 주소를 찾아간다. 정말 단순하지 않은가?

사실 나는 "메모리는 큰 배열 정도로 생각하면 된다"라고 쓰고 싶었다. 그러나 똑똑하신 분들께서 두꺼운 전공책으로 머리를 내려치며 "메모리가 어떻게 배열하고 같을 수가 있나, 이 덜떨어진 친구야"라고 말할까 봐 두려운 나머지 차마 그러진 못했다. 그러나 한편으로 그러고 싶은 마음이 드는 이유는, 우리 프로그래머들이 배열 요소에 접근할 수단으로 숫자를 인덱스로 사용하듯이, CPU도 메모리에 저장된 데이터에 접근하기 위해 숫자를 주소로 사용하기 때문이다.

컴퓨터 메모리는 배열과 달라서, '배열 요소(array elements)' 대신 '워드(words)'[10]라는 단위로 구분한다. 워드는 주소를 지정할 수 있는 가장 작은 영역이면서 접근 가능한 가장 작은 단위이다. 워드가 가질 수 있는 크기는 다양하지만, CPU 타입에 가장 많은 영향을 받는다. 모두가 표준으로 사용하는 워드 크기는 32비트와 64비트이다.

워드 크기가 8비트, 메모리 크기가 13바이트인 컴퓨터가 있다고 가정해보자. 메모리에서 워드 하나는 ASCII 문자 하나를 담을 수 있으므로, `Hello, World!`라는 문자열을 메모리에 저장한다면 [그림 1-4]와 같은 모습이 된다.

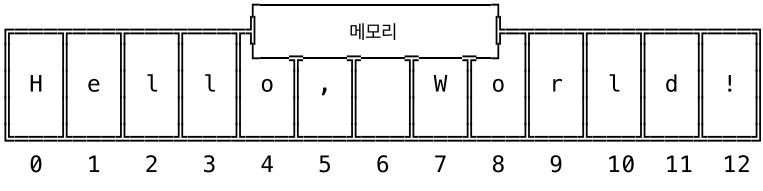

그림 1-4

10 (옮긴이) '단어(word)'라는 일반 명사와 구분하고, 메모리상에서 사용되는 단위라는 의미를 강조하기 위해 이후에는 워드로 표현하려 한다.

글자 H는 메모리 주소 0을 갖는다. e는 1을 갖는다. 첫 번째 l은 2를 갖고, W는 7을 가지며 나머지도 같은 방식으로 생각하면 된다. 그렇다면, Hello, World!에 포함된 각 글자를, 0에서 12라는 주소로 접근할 수 있다. "이봐 CPU, 주소 4에 있는 워드를 인출해 줘!"라고 명령을 내린다면, CPU는 글자 o를 인출한다. 꽤 직관적이다. 여러분이 지금 어떤 생각을 하고 있을지 짐작이 된다. 이렇게 숫자값(메모리 주소)을 가져와서 메모리 어딘가에 저장해두었다면 '포인터(pointer)'를 만들었다는 뜻이다.

지금까지 얘기한 내용이 메모리에 저장된 데이터를 찾는 방법이며, CPU가 데이터를 인출하고 저장하는 위치를 알아내는 방법이다. 한편 언제나 현실은 그리 녹록지 않다.

앞서 말한 대로, 워드의 크기는 컴퓨터에 따라 다르다. 8비트일 때도 있고 16비트, 24비트, 32비트, 64비트일 때도 있다. 때때로 CPU가 사용하는 워드 크기는 주소 크기와 무관할 때도 있다. 그리고 어떤 컴퓨터는 완전히 다른 방식으로 동작하는데, 여태껏 설명한 '워드 어드레싱(word-addressing)'이 아닌 '바이트 어드레싱(byte-addressing)'[11] 방식을 사용한다.

만약 여러분이 워드 어드레싱(word-addressing)을 사용하고, 바이트 하나에 주소를 부여하고 싶다면(드물지 않게 이런 방식을 채택한다), 단어 하나가 갖는 크기를 처리할 뿐만 아니라 오프셋(offset)까지 고려해야 한다. 따라서 이는 꽤 수고가 드는 일이며, 최적화를 수반하는 작업이다.

CPU가 메모리를 저장하고 데이터를 가져오는 방법을 간단하게 설명했는데, 이런 설명은 사실 아이들이 듣는 동화 수준의 설명이라고 생각하면 된다. 개념 수준에서는 꽤 맞는 얘기고 학습에는 도움이 될지 모른다. 그러나 최근 컴퓨터 기준에서 메모리 접근(memory access)이라

11 바이트 어드레싱(byte-addressing): 개별 바이트에 대한 접근을 지원하는 하드웨어 아키텍처를 말한다. 이런 컴퓨터를 바이트 머신(byte machine)이라고도 부른다. 이는 워드 어드레싱 아키텍처, 즉 워드 머신(word machine)이 워드(word)라 불리는 더 큰 단위의 데이터로 접근하는 방식과 대조된다(참고 *https://en.wikipedia.org/wiki/Byte_addressing*).

는 개념은 대단히 추상화되어 있다. 메모리에 접근하려면 보안과 최적화를 담당하는 수많은 추상화 계층을 거쳐야 한다. 메모리는 더 이상 옛날처럼 접근할 수 있는 대상이 아니며, 원하는 위치에 쉽게 접근할 수도 없다. 가상 메모리(virtual memory)와 보안 정책(security rules)이 메모리에 마구잡이로 접근하지 못하도록 만들었다.

메모리 이야기는 이쯤에서 그만하기로 하자. 더 옆길로 샐 수도 없는 노릇이고, 가상 메모리의 내부 동작이 어떤 것인지에 대한 얘기까지 할 수는 없다. 여러분은 메모리의 내부 동작을 알기 위해 이 책을 읽고 있는 게 아니다. 내가 이런 이야기를 꺼낸 이유는 여러분에게 요즘 CPU에서 일어나는 메모리 접근은 단순히 주소를 넘기는 행위 말고도 수많은 동작이 있다는 것을 알려주기 위해서였다. 한편 보안 정책처럼 엄격하게 제한을 두는 장치도 있지만, 최근 몇십 년간 메모리를 좀 더 자유롭게 사용하는 방법도 꽤 많이 등장했다.

폰 노이만 아키텍처는 컴퓨터 메모리에 데이터뿐만 아니라 '프로그램'도 담을 수 있다는 개념을 제시했다. 여기서 말하는 프로그램은 프로그램을 만들어내는 CPU 명령어 집합을 말한다. 코드와 데이터를 같이 쓴다고 하면 프로그래머들은 그다지 좋아하지 않을 것이다. 그리고 몇 세대 이전 프로그래머들 역시 비슷하게 느꼈으리라. 왜냐하면 코드와 데이터를 같이 사용하지 못하도록 하는 여러 관습을 정립해왔기 때문이다.

프로그램 역시 다른 데이터와 마찬가지로 메모리에 저장되지만, 대개 데이터와 같은 위치에 저장되지 않는다. 특정 메모리 영역은 데이터만 저장하게끔 만들어져 있다. 그것은 관습 때문만이 아니라 운영체제, CPU, 그 밖의 컴퓨터 아키텍처 모두 특정 메모리 영역만 사용하도록 규정하고 있기 때문이다.

'텍스트 파일 안에 담긴 내용'이나 'HTTP 응답처럼 순수한 데이터(dumb data)'가 저장되는 메모리 영역이 있고, 프로그램을 만드는 명령어를 저장하고 접근하기 위한 메모리 영역이 따로 있다. 후자에서 말하는 메모리 영역에 접근할 때, CPU는 더 쉽고 빠르게 명령어를 인출할 수 있다. 또 다른 메모리 영역에는 프로그램이 사용하는 정적 데이터(static