11장

프로그래밍 언어

프로그래밍을 배우다 보면 컴파일러 언어와 인터프리터 언어라는 말을 듣게 됩니다. 대표적인 컴파일러 언어에는 C나 자바가 있고 인터프리터 언어에는 파이썬이나 자바스크립트가 있습니다. 이번 장에서는 컴파일러 언어와 인터프리터 언어에 대해 살펴보고, 우리가 다루는 인터프리터 언어인 파이썬의 작동 방식을 자세히 알아보겠습니다.

1 컴파일러 언어와 인터프리터 언어

컴파일러 언어와 인터프리터 언어의 차이를 명확하게 이해하기란 생각보다 쉽지 않습니다. 다음 코드를 볼까요?

```
>>> s = '5 + 3'
>>> code = compile(s, '<string>', 'eval')
>>> eval(code)
8
```

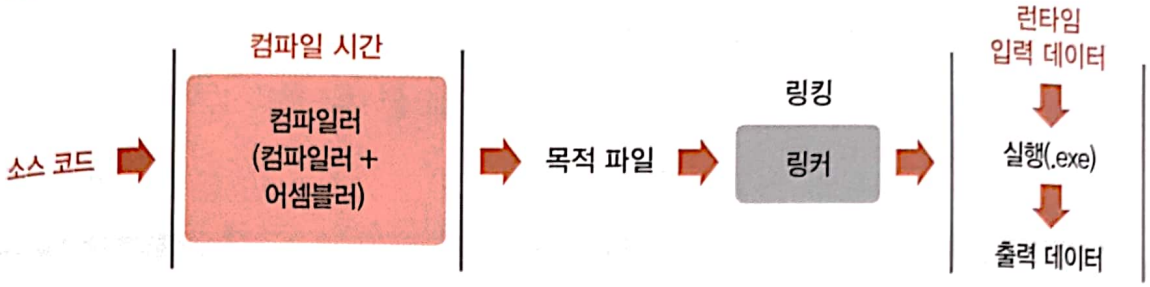
파이썬에는 compile()이라는 함수가 있어 문자열을 컴파일할 수 있습니다. 여기서 중요한 점은 파이썬에 소스 코드를 컴파일할 수 있는 컴파일러가 있다는 점입니다. 파이썬은 인터프리터 언어인데 컴파일러라니 어찌된 일일까요?

컴파일러 언어와 인터프리터 언어는 컴파일 타임이 있느냐 없느냐 즉, 소스 코드를 분석하는 시점과 입력 데이터를 받는 시점이 언제이냐에 따라 나뉩니다. 이 문장만으로는 아직 명확히 이해되지 않습니다. 컴파일러 언어인 C와 인터프리터 언어인 파이썬을 예로 들어 보겠습니다.

1.1 C 언어: 컴파일러 언어 분석

C 언어로 작성한 소스 코드가 실행 가능 파일이 되는 과정을 살펴봅시다. 그림 11-1은 컴파일러 언어 C로 소스 코드를 작성한 다음 실제 데이터를 입력받아 그 결과를 출력하는 실행 과정을 그림으로 나타낸 것입니다.

그림 11-1 소스 코드에서 실행까지(컴파일러 언어 C)



C 언어는 소스 코드를 컴파일하여 목적 코드(object code 또는 object file)인 기계어로 된 인스트럭션을 만들어 냅니다. 링커(linker)는 필요한 라이브러리를 가져오고 여러 개의 목적 파일을 함께 묶어 실행 파일(executable file)을 생성합니다. 이제 프로그램을 실행하고 데이터를 입력하면 결과 데이터가 출력됩니다. 중요한 점은 소스 코드를 분석하는 컴파일 타임(compile time)과 실제 데이터를 입력받아 결과를 출력하는 런타임(run time)이 분리되어 있다는 점입니다.

1.2 파이썬: 인터프리터 언어 분석

그림 11-2는 인터프리터 언어인 파이썬이 소스 코드와 데이터를 동시에 입력받아 결과를 출력하는 과정을 그림으로 나타낸 것입니다.

그림 11-2 파이썬 코드가 컴파일되어 실행되는 과정

파이썬도 소스 코드가 있으므로 이를 분석하는 컴파일러가 있습니다. 목적 코드로 기계어를 생성하는 C 언어와 달리 파이썬은 바이트 코드(byte code)를 생성합니다. 바이트 코드가 생성된 후에는 PVM(Python Virtual Machine, 파이썬 가상 머신)에서 바이트 코드를 하나씩 해석하여 프로그램을 실행합니다. 이러한 이유로 PVM을 파이썬 인터프리터라고 부르기도 합니다. 중요한 점은 소스 코드를 분석하는 컴파일 타임이 따로 없고 실행과 동시에 분석을

시작한다는 점입니다. 즉, 소스 코드와 입력 데이터가 같은 시점에 삽입됩니다. 컴파일러 언어와 비교했을 때 가장 큰 차이입니다.

2 파이썬: 소스 코드부터 실행까지

우리가 작성한 파이썬 소스 코드가 바이트 코드로 변환되어 실행되는 과정을 쭉 따라가 보겠습니다. 이 과정을 따라가다 보면 프로그래밍 언어에서 중요한 개념인 컴파일러, AST, 심벌 테이블, 바이트 코드, 가상 머신 등을 만나게 됩니다.

2.1 컴파일러

일반적인 컴파일러는 렉서(lexer)와 파서(parser)로 구성됩니다. 소스 코드가 렉서와 파서에 의해 어떻게 변하는지 간략히 알아봅시다.

그림 11-3 컴파일러의 구성

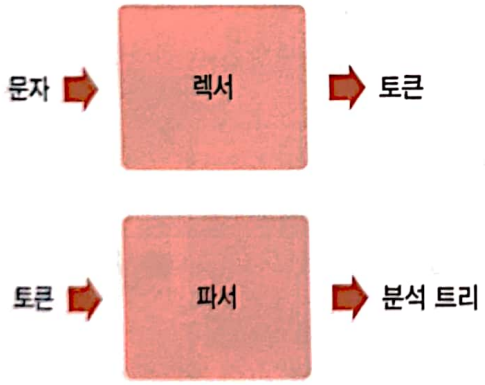


그림 11-3에서 렉서로 입력되는 것은 소스 코드입니다. 소스 코드도 결국에는 문자에 불과합니다. 이 문자들이 렉서를 거치면서 여러 개의 토큰(token)으로 변경됩니다.

토큰은 뭘까요? 예를 들어 "나는 사과를 먹었다"는 문장이 있습니다. 이 문장은 주어("나는"), 목적어("사과를"), 동사("먹었다")로 나눌 수 있습니다. 이렇게 문장을 종류별로 쪼갠 다음 종류와 문자를 함께 나타낸 것을 토큰이라고 합니다. 위 문장을 토큰으로 나타내면 다음과 같습니다.

〈주어, "나는"〉, 〈목적어, "사과를"〉, 〈동사, "먹었다"〉

총 세 개의 토큰으로 나타낼 수 있습니다. 프로그래밍 언어도 마찬가지입니다. 우리가 작성한 코드는 언어의 문법에 맞게 토큰으로 쪼갤 수 있습니다. 종류에는 변수나 함수 이름을 의미하는 식별자, `for` · `while` · `if` · `elif` 같은 키워드, `1` · `2` · `3` · `'a'` · `'b'` · `'c'` 같은 상수, `+` · `-` · `*` · `/` 같은 연산자 등이 있습니다.

파서는 토큰을 분석하여 분석 트리(parse tree)를 구성합니다. 컴파일러마다 분석 트리를 생성하기도 하고 생성하지 않기도 합니다. 분석 트리가 만들어지면 이를 이용해 목적 코드(C 언어는 기계어, 파이썬은 바이트 코드)를 생성합니다. 이를 코드 생성(code generation)이라고 합니다.

> **TIP** 분석 트리를 이해하려면 BNF 표기법 등 컴파일러 지식이 많이 필요하므로 이 책에서는 생략하겠습니다. 다만 분석 트리를 변형해 만들어지는 추상 구문 트리에 대해서는 잠시 후에 간략히 알아봅니다.

파이썬은 파이썬 컴파일러를 통해 다음과 같은 과정을 거쳐 바이트 코드를 생성합니다.

1| 소스 코드 → 분석 트리

2| 분석 트리 → 추상 구문 트리

3| 심벌 테이블 생성

4| 추상 구문 트리 → 바이트 코드

예제 코드를 작성하고 이 코드가 컴파일러를 거쳐 바이트 코드로 바뀐 후 PVM에서 작동하는 과정을 따라가 보겠습니다.

코드 11-1 programming_language/test.py

```
def func(a, b):
    return a + b

a = 10
b = 20

c = func(a, b)
print(c)
```

실행결과 30

코드 11-1은 매우 간단합니다. 두 인자를 더해 반환하는 함수 func()와 두 전역 변수 a와 b가 있고, a와 b를 인자로 받아 func() 함수를 호출한 다음 c에 그 값을 저장하고 이를 print() 함수로 출력합니다. 실행하면 가장 먼저 컴파일러의 렉서가 코드의 문자들을 토큰으로 바꿀 것입니다. test.py가 어떤 토큰들로 변경되는지 확인해 보겠습니다.

```
>>> from tokenize import tokenize
>>> from io import BytesIO
>>> s = open('test.py').read()
>>> g = tokenize(BytesIO(s.encode('utf-8')).readline)
>>> for token in g:
        print(token)
```

tokenize 모듈을 이용해 소스 코드를 여러 개의 토큰으로 쪼갠 다음 토큰을 출력하였습니다. 토큰을 출력해 보면 타입(type)과 실제 문자열(string), 시작 위치(start)와 끝(end) 등의 정보를 알 수 있습니다.

실행결과

```
TokenInfo(type=59 (ENCODING), string='utf-8', start=(0, 0), end=(0, 0), line='')
TokenInfo(type=1 (NAME), string='def', start=(1, 0), end=(1, 3), line='def func(a, b):\n')
TokenInfo(type=1 (NAME), string='func', start=(1, 4), end=(1, 8), line='def func(a, b):\n')
(중략)
```

이렇게 얻어진 토큰으로 분석 트리를 만든 다음 추상 구문 트리로 변형합니다.

잠깐만요

파이썬에는 얼마나 많은 토큰 종류가 있을까요?

token 모듈을 사용하면 다음과 같이 파이썬이 사용하는 토큰 종류를 알 수 있습니다.

```
>>> import token
>>> token.tok_name
{0: 'ENDMARKER', 1: 'NAME', 2: 'NUMBER', 3: 'STRING', 4: 'NEWLINE', 5: 'INDENT', 6: 'DEDENT', 7: 'LPAR', 8: 'RPAR', 9: 'LSQB', 10: 'RSQB', 11: 'COLON', 12: 'COMMA', 13: 'SEMI', 14: 'PLUS', 15: 'MINUS', 16: 'STAR', 17: 'SLASH', 18: 'VBAR', 19: 'AMPER', 20: 'LESS', 21: 'GREATER', 22: 'EQUAL', 23: 'DOT', 24: 'PERCENT', 25: 'LBRACE', 26: 'RBRACE', 27: 'EQEQUAL', 28: 'NOTEQUAL', 29: 'LESSEQUAL', 30: 'GREATEREQUAL', 31: 'TILDE', 32: 'CIRCUMFLEX', 33: 'LEFTSHIFT', 34: 'RIGHTSHIFT', 35: 'DOUBLESTAR', 36: 'PLUSEQUAL', 37: 'MINEQUAL', 38: 'STAREQUAL', 39: 'SLASHEQUAL', 40: 'PERCENTEQUAL', 41: 'AMPEREQUAL', 42: 'VBAREQUAL', 43: 'CIRCUMFLEXEQUAL', 44: 'LEFTSHIFTEQUAL', 45: 'RIGHTSHIFTEQUAL', 46: 'DOUBLESTAREQUAL', 47: 'DOUBLESLASH', 48: 'DOUBLESLASHEQUAL', 49: 'AT', 50: 'ATEQUAL', 51: 'RARROW', 52: 'ELLIPSIS', 53: 'OP', 54: 'AWAIT', 55: 'ASYNC', 56: 'ERRORTOKEN', 57: 'COMMENT', 256: 'NT_OFFSET', 58: 'NL', 59: 'ENCODING'}
```

2.2 추상 구문 트리

추상 구문 트리(Abstract Syntax Tree, AST)란 소스 코드의 구조를 나타내는 자료 구조입니다. 추상 구문 트리를 바탕으로 심벌 테이블을 만들고 바이트 코드를 생성할 수 있습니다. ast 모듈을 이용하여 test.py의 추상 구문 트리를 볼까요?

```
>>> import ast
>>> node = ast.parse(s, 'test.py', 'exec')     #1
>>> g = ast.walk(node)                          #2
>>> next(g)
<_ast.Module object at 0x03A89490>
>>> next(g)
<_ast.FunctionDef object at 0x03A893F0>
>>> next(g)
<_ast.Assign object at 0x03A895B0>
```

TIP 트리는 자료 구조의 일종인데 우리는 아직 자료 구조를 배우지 않았습니다. 그러므로 다 이해되지 않을 수 있습니다. 그럴 때는 그냥 실행하고 결과만 확인해 보세요.

노드를 생성하고(#1) walk() 함수를 사용하면(#2) 트리의 모든 노드를 순회할 수 있는 발생자 객체를 얻을 수 있습니다.

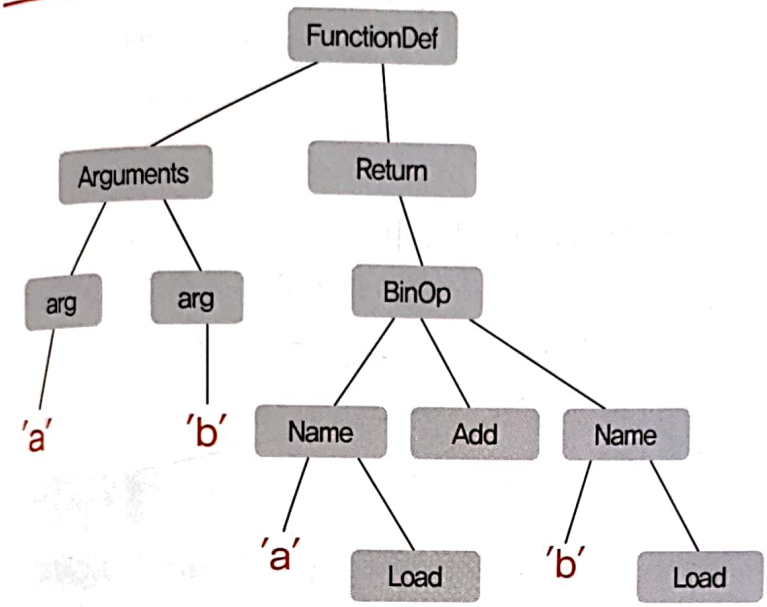
발생자(generator)는 함수를 실행 도중에 멈췄다가 원하는 시점에 다시 시작할 수 있도록 하는 함수입니다. 여기서는 walk() 함수를 통해 만들어진 발생자 객체가 next() 함수를 호출할 때마다 노드를 하나씩 넘겨준다는 점만 기억하면 됩니다.

발생자를 만든 다음 next() 함수를 통해 노드를 하나씩 획득하고 있습니다. 이렇게 봐서는 트리가 어떻게 구성되어 있는지 알기 어렵습니다. 함수 func()의 정의 부분만 따로 떼어낸 트리 구성을 살펴보겠습니다. 다음은 함수 func()의 정의 부분입니다.

```
def func(a, b):
    return a + b
```

이 코드의 추상 구문 트리를 그림으로 나타내면 그림 11-4와 같습니다.

그림 11-4 함수 func의 추상 구문 트리



함수 func()의 정의 부분을 그림 11-4와 비교해 보면 AST 트리가 어떻게 구성되는지 알 수 있습니다. 인자(arguments)는 a와 b고, 변수 이름 a와 b 값을 불러와(load) 이진 연산자(BinOp, binary operator)인 +(Add)로 연산한 다음 반환(return)합니다.

이번에는 함수 func()의 반환 값을 c에 저장하는 코드의 추상 구문 트리를 살펴봅시다.

```
c = func(a, b)
```

함수 func()을 호출하면서 인자 a와 b를 전달하고 그 반환값을 c에 대입하였습니다. 그림 11-5는 이 코드의 추상 구문 트리 구성입니다.

그림 11-5 함수 호출 부분의 추상 구문 트리

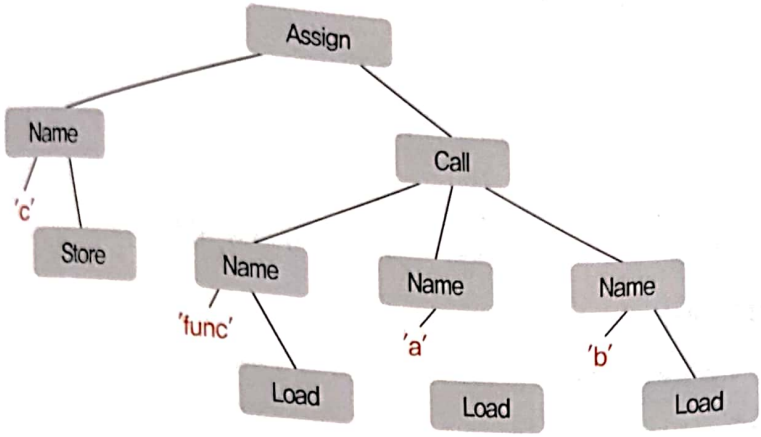
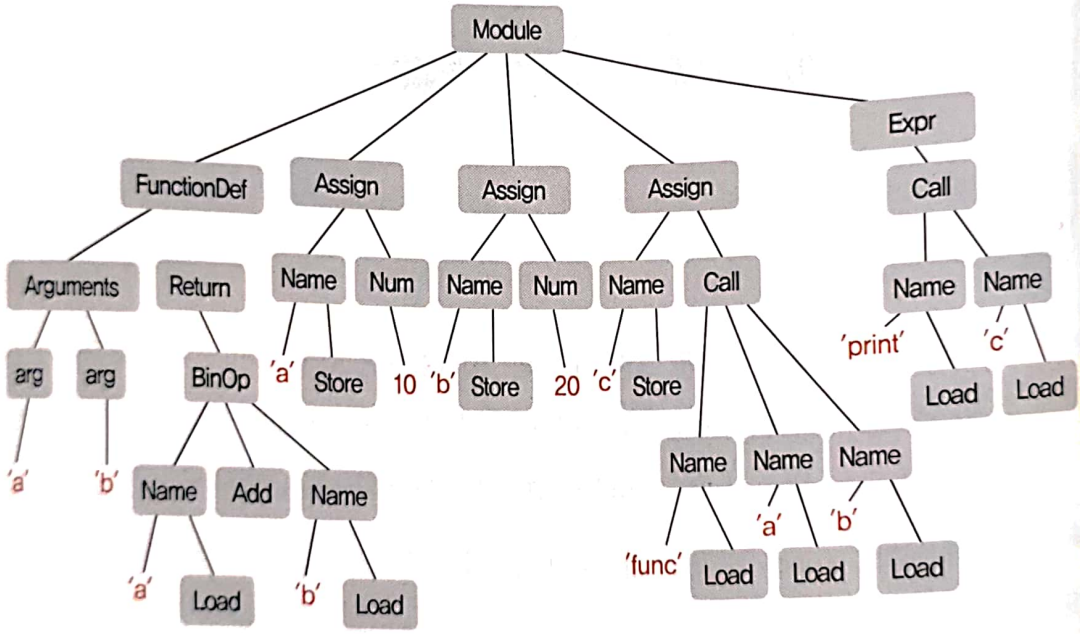


그림 11-6은 test.py 전체 코드의 AST를 나타낸 것입니다.

그림 11-6 test.py 전체 코드의 추상 구문 트리



2.3 심벌 테이블

AST도 살펴보았으니 이제 심벌 테이블에 대해 이야기해 보겠습니다. 심벌 테이블(symbol table)은 변수나 함수의 이름과 그 속성에 대해 기술해 놓은 테이블입니다.

test.py를 보면 전역 영역(global)에 변수 a, b, c와 함수 func, print가 있는 것을 알 수 있습니다. 심벌 테이블을 확인해 보겠습니다.

```
>>> import symtable
>>> sym = symtable.symtable(s, 'test.py', 'exec')  #1
>>> sym.get_name()          #2
'top'
>>> sym.get_symbols()       #3
[<symbol 'func'>, <symbol 'a'>, <symbol 'b'>, <symbol 'c'>, <symbol 'print'>]
```

symtable() 함수를 이용해 테이블을 받아오고(#1) 이름을 확인합니다(#2). 이름이 'top'이라고 나오는데 이는 이 테이블이 글로벌 테이블이라는 의미입니다. 클래스 테이블이면 클래스 이름이 나오고, 함수 테이블이면 함수 이름이 나옵니다. 다음으로 get_symbols() 함수로 현재 영역에 있는 심벌을 확인합니다(#3).

이번에는 함수 func의 심벌 테이블을 보겠습니다.

```
>>> sym.get_children()                  #1
[<Function SymbolTable for func in test.py>]
>>> func_sym = sym.get_children()[0]    #2
>>> func_sym.get_name()                 #3
'func'
>>> func_sym.get_symbols()              #4
[<symbol 'a'>, <symbol 'b'>]
```

글로벌 심벌 테이블 안에 다른 심벌 테이블이 있는지 알아보고(#1) 테이블을 받아옵니다(#2). 심벌 테이블의 이름을 확인하면(#3) 'func'입니다. 함수 func의 심벌 테이블입니다. 심벌을 얻어오면(#4) 인자 a와 b를 볼 수 있습니다.

2.4 바이트 코드와 PVM

마지막 단계로 바이트 코드를 생성할 차례입니다. 다음 코드를 입력해 test.py의 바이트 코드를 확인해 봅시다.

```
>>> import dis
>>> g = dis.get_instructions(s)                    #1
>>> for inst in g:
        print(inst.opname.ljust(20), end = ' ')    #2
        print(inst.argval)                         #3
LOAD_CONST          <code object func at 0x00D0B6A8, file "<disassembly>",
line 1>
LOAD_CONST          func
MAKE_FUNCTION       0
STORE_NAME          func
LOAD_CONST          10
STORE_NAME          a
LOAD_CONST          20
STORE_NAME          b
LOAD_NAME           func
LOAD_NAME           a
LOAD_NAME           b
CALL_FUNCTION       2
STORE_NAME          c
LOAD_NAME           print
LOAD_NAME           c
CALL_FUNCTION       1
POP_TOP             None
LOAD_CONST          None
RETURN_VALUE        None
```

모듈 dis를 불러온 다음 get_instructions() 함수를 통해 바이트 코드를 제공하는 발생자를 만듭니다(#1). 바이트 코드 이름(#2)과 인자 값(#3)을 출력해 보면 어셈블리어와는 다른 파이썬의 바이트 코드 인스트럭션을 확인할 수 있습니다.

바이트 코드는 파이썬의 가상 머신 위에서 실행됩니다. 가상 머신이라고 하니 뭔가 난해할 것 같지만 그렇지 않습니다. 파이썬의 가상 머신인 PVM은 그저 굉장히 큰 무한 루프일 뿐입니다. 다음 코드는 CPython 소스 코드 중 ceval.c 파일에 있는 PVM 코드를 알아보기 쉽게 재구성한 것입니다.

```
PyObject *
_PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
{
    int opcode;         /* Current opcode */
    int oparg;          /* Current opcode argument, if any */
    int word;

    opcode = _Py_OPCODE(word); \
    oparg = _Py_OPARG(word); \

    for (;;) {
        switch (opcode) {
        case NOP:
            // Things
            break;
        case LOAD_FAST:
            // Things
            break;
        case STORE_FAST:
            // Things
            break;
        /*
        The rest of bytecode cases
```

```
            */
            }
        }
    }
```

C 언어 문법이 나왔다고 당황하지 마세요. 완벽하게 이해하지 않아도 괜찮습니다. 그저 PVM이 어떻게 구성되었는지 흐름을 파악하는 용도입니다. _PyEval_EvalFrameDefault() 함수는 바이트 코드의 실제 실행을 담당하는 함수로, 이 함수 안에서 opcode라는 변수가 바이트 코드를 받아옵니다.

초기식이나 증감식이 없는 for(;;) 문은 무한 루프를 의미합니다. 파이썬의 while True: 와 같습니다. 이 무한 루프가 바로 PVM입니다. 무한 루프 안에는 실제 바이트 코드를 분석해서 실행하는 switch 문이 있습니다(파이썬에는 switch 문이 없는 대신 if~elif 문이 그 역할을 합니다).

switch 문 안에 있는 case 문을 살펴보면 NOP, LOAD_FAST, STORE_FAST 같은 바이트 코드가 눈에 띕니다. Opcode가 NOP라면 break 문을 만날 때까지 case NOP: 부분이 실행됩니다. 다른 opcode라면 그에 해당하는 case 문이 실행됩니다. PVM이 어떻게 작동하는지 어렵지 않게 유추할 수 있습니다.

3 마무리

이번 장에서는 컴파일러 언어와 인터프리터 언어에 대해 자세히 알아보았습니다. 또한 파이썬의 소스 코드가 바이트 코드로 변환된 다음 실행되는 과정도 살펴보았습니다. 파이썬의 내부 구조를 몰라도 코드를 작성하는 데 지장은 없지만, 알고 있으면 프로그래밍을 하는 도중에 발생하는 여러 이슈에 대해 좀 더 직접적으로 접근할 수 있습니다.

다음 장부터는 자료 구조를 알아봅니다.