

Python 실행과정 훑아보기

Table of Contents

1. 왜 느림?

2. 기본적인 컴파일 과정

3. Python의 컴파일 과정

Problem

Python language의 문제를 해결해보자.

- **Slow**
- Type Error

Why slow?

1. Dynamic Typing Overhead
2. Interpreted Execution (CPython)

Compile Process

- Frontend: Lexing, Parsing, Semantic Analysis
- Middleend: Code Optimization
- Backend: Code Generation

Lexing

- code string to **token sequences**
- using **Thompson's construction** and **Subset construction**

Parsing

- Generate Syntax Tree from Token Sequences
- Top-down parsing
- Bottom-up parsing

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow id$

Python Compile

- Code to Bytecode
- Byte code is executed by Interpreter(CPython)

Python Lexing

- by `tokenize`
- backtracking regex engine

Python Parsing

1. Token Sequence to CST
2. CST to AST
3. AST to Bytecode

Token Sequence to CST

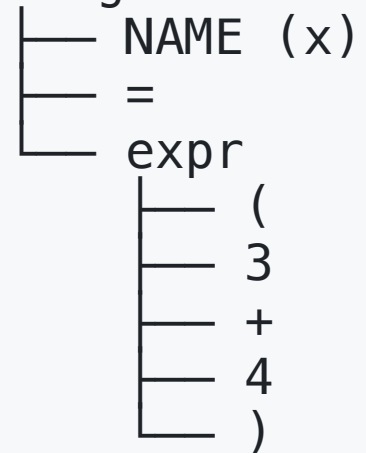
- Concrete Syntax Tree
- using `pegen` (PEG Generator)
- PEP617
- [python.gram](#)

CST to AST

- CST

```
x = (3 + 4)
```

Assign



CST to AST

- AST

```
Assign(  
    targets=[Name(id='x')],  
    value=BinOp(left=Constant(3), op=Add(), right=Constant(4))  
)
```

CST to AST

- [Python/ast.c](#)
- [Parser/Python.ast](#)
- [Python ast](#)

AST to Bytecode

- low-level, stack-based instruction set

1. Symbol Table Construction (symtable.c)

```
x = 1  
def f():  
    y = x + 2
```

The symbol table detects:

- x: global in f
- y: local in f

2. Compiler State Setup (compile.c)

```
struct compiler c;  
compiler_init(&c, ...); // Setup for AST → bytecode
```

3. AST Tree Walk (compile.c)

```
static int compiler_visit_expr(struct compiler *c, expr_ty e);  
static int compiler_visit_stmt(struct compiler *c, stmt_ty s);
```

Node Type	Handler Function	Emits
Assign	compiler_visit_assign()	STORE_NAME , LOAD_CONST , etc.
BinOp	compiler_visit_binop()	BINARY_ADD , BINARY_MULTIPLY , etc.
If	compiler_visit_if()	Labels, conditional jumps
Call	compiler_visit_call()	CALL_FUNCTION

4. Emit Bytecode

- EMIT(c, opcode) → emit an opcode
- ADDOP(c, opcode) → same, with stack effect tracking
- ADDOP_O(c, opcode, oparg, location) → opcode with operand (like STORE_NAME)

```
ADDOP_0(c, LOAD_CONST, index_3, loc);  
ADDOP_0(c, LOAD_CONST, index_4, loc);  
ADDOP(c, BINARY_ADD);  
ADDOP_0(c, STORE_NAME, index_x, loc);
```

E.O.D