Compiler 2024F Final Report

Team members:

- 110590002 王熯紘
- 110590004 林奕廷
- 110590017 陳姿安

Technical Choices

The **mini-Python compiler** was designed and implemented with an emphasis on correctness rather than optimization. Every statement and expression was directly compiled into assembly code from a Typed AST without intermediate optimization passes. Despite this simplistic approach, the compiler successfully passed all test cases, demonstrating functional correctness.

Reasoning Behind the Approach

The primary reason for adopting this direct compilation method was **time constraints and deadline pressure**. We prioritized an early start to avoid multiple overlapping deadlines, allowing for consistent progress even before covering the advanced compiler optimization techniques discussed in **Chapter 10** of our curriculum. As a result, the compiler was built incrementally, focusing on correctness rather than efficiency.

Current Compilation Strategy

1. **Direct Translation from Typed AST to Assembly**
    - Each expression and statement is compiled directly into x86-64 assembly instructions.
    - Complex expressions (e.g., binary operations, list handling, and conditionals) are broken down into function calls or inline assembly blocks.
    - Memory management relies on explicit calls to `my_malloc` and standard memory operations.
2. **Memory Management**
    - A custom memory allocation function (`my_malloc`) is used for dynamic data structures (e.g., lists, strings).
    - Type tags are explicitly assigned to ensure runtime type checking.
3. **Error Handling**
    - Runtime checks are embedded in the generated assembly code to validate types, index bounds, and operation compatibility.
    - Custom error labels handle invalid operations with descriptive error messages.
4. **Function Calls and Parameter Passing**
    - Function arguments are evaluated and pushed onto the stack.
    - Return values are handled consistently, with standard stack frame conventions maintained across calls.