Overview

The goal of this project is to write a Python-to-C transpiler. We aim to limit some of the functionality in order to keep the project within the scope of time given.

Limitations:

- Only type-annotated functions e.g. `def sampleFunction(input: string) -> int`
- No closures (functions within functions)
- We won't be focusing on object-oriented constructs
- Primitive types + strings {Int, Bool, String, Float}
- Operations {Add, Sub, Mult, Div, Pow, Mod, And, Or, Not}
- Basic control flow {If/else, For (range-based), While}, no for each loops
- Only basic core functions supported {print() -> write(), input -> read()}
- Cannot import 3rd-party libraries
- Cannot use None keyword, must declare its empty/default type e.g. int's default type is 0, strings are "", etc.
- No function pointers.

Features:

- Dependencies/modules e.g. header files and source files compiling
- Depending on time, we may support data structures like lists

Further Development (Time Permitting):

- REPL (live convert, show AST)

Libraries

Core!

Module type declarations

ast.mli

- Functionality:
  - Defines AST and helper functions
- Declarations (see definitions in ast.mli):
  - type ast = statement list
  - type statement = …
  - type expression = …
  - type binaryOp = …
  - type unaryOp = …
  - type primitive = …
  - val showAst: ast:ast -> string

io.mli

- Functionality:
  - Reads in the Python File(s) and outputs C File(s)
- Declarations:
  - val readFile: fileName:string -> string
  - val writeFile: fileName:string -> buffer:string -> string

lex.mli

- Functionality:
  - Lex python code to generate a list of tokens
- Declarations:
  - type token = …
  - val lex: source:string -> token list

parse.mli

- Functionality:
  - Parse tokens and generate AST
- Declarations:
  - val parse: token list -> AST result

codegen.mli

- Functionality:
  - Traverses through the AST to generate the code.
- Declarations:
  - val traverseAst: ast:ast -> string

Use cases

Example 1: Simple hello world script with IO.

```
print('hello world')
```

```
int main() {
      printf("hello world");
}
```

Example 2: Use of range-based for loops and control flow.

```
for i in range(10):
      if i % 2 == 0:
            print(i)
```

```
int main() {
      for (int i = 0; i < 10; i++) {
            if (i % 2 == 0) {
                  printf("%d", i);
            }
      }
}
```

Example 3: Keywords, function calls.

```
def foo(a: int, b: int) -> int:
      answer = 0
      if a == 0 and b == 0:
            answer = a*5
      else:
            answer = b*2
      return answer

print(foo(2, 3))
```

```
int foo(int a, int b) {
      int answer = 0;
      if (a == 0 && b == 0) { answer = a*5; }
      else { answer = b*2;}
      return answer;
}

int main(){
```

```
    printf("%d", main_sample(2, 3));
}
```

Implementation order

1. We will have separate test_XX.ml files for each module, as well as end-to-end tests for the entire Python-to-C pipeline
2. `io.mli` — implement file I/O first so we can test using sample Python scripts. We can also check the outputs from the C file generated.
3. `ast.mli` — define a common representation for the AST that will be shared between the parse and codegen modules.
4. We will be separating the work so that one person focuses on Python-to-AST and the other person focuses on AST-to-C
    a. `lex.mli, parse.mli` – functionality and tests
    b. `codegen.mli` – functionality and tests
5. `convert.ml` — create a top-level file that will compile into an executable.