ACC&PA Spring 2022
Final Project Requirements

The Final Project in ACC&PA course amounts to a design and implementation of a programming language by students, in teams of up to 4 people. A (non-exhaustive) list of features that students can implement in their projects, together with approximate difficulty of implementation, is available in this spreadsheet.

This document gives a more detailed description for some of these features.
We will refer to the programming language implemented by students as **object language**.

Interpreter, Compiler or Transpiler

Obviously, implementation of a programming language implies creating a tool that is able to process the language. In this course, it is sufficient to produce one of the following:

1. **Interpreter** is an executable program that is able to
    a. Run files (scripts), evaluating/executing commands in those files.
    b. Run an interactive session, a REPL (read-eval-print loop).
2. **Compiler** is an executable program that is able to
    a. Generate object code for a low-level machine (e.g. C--, LLVM, WASM).
    b. Link and build an executable program (binary) from source code in the object language.
3. **Transpiler** is an executable program that is able to
    a. Generate object code in another high-level language (e.g. C, JavaScript, Prolog, LISP).
    b. Transpile, compile, link, and build executable program (binary) from the source code in the object language.

Important: type checking should be performed explicitly by your implementation of interpreter/compiler/transpiler. You cannot delegate type checking to the object code!

Type Checker

Whatever interpreter/compiler/transpiler you implement, it should be possible to easily perform the following actions:

1. **Check** the type of any expression against an expected type (i.e. user should be able to provide expression, its expected type, and ask the tool to check whether the typing is correct).
2. **Infer** the type of expressions (i.e. user should be able to provide an expression, perhaps with some type annotations inside, and ask the tool to compute the type for the whole expression).

Type checking should result in one of the following outcomes:

1. Type checking/inference is successful.
2. Type checking fails and a proper human-readable type error is presented to the user.
3. Type checking fails because of lack of type annotations (not enough information for the type checker), and a proper human-readable type error is presented to the user, suggesting to add type annotations somewhere.
4. Parsing fails and a proper human-readable error message is presented to the user.

Important: type checker should perform statically, i.e. without executing the expressions.

Base Types

Base types are elementary types that are built into the language. Generally, base types can be split into general purpose and domain-specific:

1. General-purpose base types:
    a. Numeric types (Integer, Double)
    b. String, Text types
    c. Unit type
    d. Empty type
2. Domain-specific base types:
    a. Excel Cell/Formula Types
    b. DNA/RNA nucleotide base type (A, C, G, T), amino acid type
    c. Musical Notation Types (e.g. Staff, Note, Pitch)

User-defined Terms and Types

It should be possible for the user of the object language to define or, at least, alias terms and types. For example, your language may support top-level syntax like this:

```
1. val myDefinition = <expression> ;
2. type MyType = <type> ;
```

Note: you are **not required** to support recursive definitions, or forward references.

Standard Library

You must implement a (small) **standard library** that is either available by default, or is easily imported in by the user. Note, that standard library is not built-in, it must be written entirely in the object language.

First-class functions

The object language must support (even if indirectly), **first-class functions**. This can be implemented via anonymous functions (lambda abstractions) or via anonymous classes/objects. In particular, the user should be able to perform the following in the object language:

1. **Define (anonymous) functions** in the place of their use

Example (C++):

```
void func() {
  auto f = [](int i) { return i;};
}
```

2. **Pass functions as arguments**

Example (C++):

```
int func(std::function<int(int)> f) {
  return f(5);
}
…
auto f = [](int i) {return i + 1;};
func(f);
```

3. **Return functions** as result values of other functions

Example (C++):

```
std::function<int(int)> foo() {
  auto f = [](int i) {return i;};
  return f;
}
```

or

```
auto foo() {
  auto f = [](int i) {return i;};
  return f;
}
```

Nested definitions

The object language must support **nested definitions** (e.g. nested functions, classes). It should be possible to define a local variables/function/classes (e.g. helpers) in any scope:

1. Inside of another function definition
2. Inside of another class
3. Inside of any block of code (statement block)

Examples:

Nested classes in C++:

```
class A {
public:
  class B {
  };
  class C {
  };
};
…
auto b = A::B();
```

Nested functions in Python:

```
def func1():
  def func2():
    return 1
  return func2()
…
print(func1())
```

Simple Constraint-Based Type Inference

The object language must support **Simple Constraint-Based Type Inference**. In particular:

1. In the syntax, any type annotation in the source code should be optional.
2. A type reconstruction algorithm should be implemented, so that the type checker can infer omitted types, at least in many practical situations (see Chapter 22 of TaPL).

Important: it should be possible to omit types **anywhere**, including function/method arguments and results.

Example in Language O with explicit types:

```
method MaxInt(a: Array[Integer]) : Integer is
    var max = Integer.Min
    var i = Integer(1)
    while i.Less(a.Length) loop
        if a.get(i).Greater(max) then max := get(i) end
        i := i.Plus(1)
end
    return max
end
```

Example in Language O with **auto** types (syntactic equivalent of omitted types):

```
method MaxInt(a : auto) : auto is
    var max = Integer.Min
    var i = 1
    while i.Less(a.Length) loop
        if a.get(i).Greater(max) then max := get(i) end
        i := i.Plus(1)
end
    return max
end
```

Example with **auto** types in C++20 (with a function parameter declaration as auto):

```
auto func(auto i) {
  return i;
}
…
func(1);
```