Problems

1. ........................................ Subtyping enumerations

In this problem, we will explore the subtyping rules for enumerations.

\[ e ::= ... \]
\[ \mid \#l \]
\[ \mid \text{case } e \text{ of } \#l_1 \rightarrow e_1 \mid \ldots \mid \#l_i \rightarrow e_i \]

Enumeration values are identified via labels \( l \) drawn from some predetermined set (analogously to how fields in records are identified with labels). To create an enumeration value, prefix a label with a \( \# \), e.g. \#true or \#false (the \( \# \) disambiguates enumeration labels from regular variables); and you can case-split on an enumeration value using case. For example, case \#true of \#true \rightarrow e_1 \mid \#false \rightarrow e_2 evaluates to \( e_1 \).

Labels have their own type:

\[ \tau ::= \ldots \]
\[ \mid \langle l_1, \ldots, l_i \rangle \]

Intuitively, \( e :: \langle \text{true}, \text{false} \rangle \) means that \( e \) is an expression which evaluates to either \#true or \#false; i.e., it represents the type of booleans. Here are the typing rules:

\[
\begin{align*}
\text{#l} &::= \{ l \} \\
\text{e} &::= \langle l_1, \ldots, l_i \rangle \quad \text{e}_1 :: \tau \ldots \text{e}_i :: \tau \\
\text{case } e \text{ of } \#l_1 \rightarrow e_1 \mid \ldots \mid \#l_i \rightarrow e_i &:: \tau
\end{align*}
\]

(a) Is \( \langle \text{yes, no} \rangle \) a subtype of \( \langle \text{yes, no, maybe} \rangle \)? Justify your reasoning.

(b) Using plain English or an inference rule, describe the subtyping rule for enumerations.

(c) Briefly describe the relationship between this subtyping rule and Java’s rule that an overriding method cannot throw more checked exceptions than those declared in the parent class.

(d) Give the most general type for the expression:

\[
\text{case } \#\text{cucumber of } \#\text{cucumber} \rightarrow \#\text{cucumber} \mid \#\text{almond} \rightarrow \#\text{almond}
\]

i.e., the type \( \tau \) such that for all valid types \( \tau' \), \( \tau \) is a subtype of \( \tau' \).

(e) Records whose fields all have the same type \( \tau \) can be defined as a function from an enumeration to \( \tau \), e.g., \( \{ l_1 = \tau, \ldots, l_i = \tau \} \equiv \langle l_1, \ldots, l_i \rangle \rightarrow \tau \). Using the subtyping rule for enumerations, show that the width subtyping rule (with each type specialized to \( \tau \)) holds for these records.

(f) A way to further extend enumerations is to let them carry data; rather than \( \langle l_1, l_2 \rangle \), we could specify \( \langle l_1 :: \tau_1, l_2 :: \tau_2 \rangle \), so that \#l_1 is a constructor which takes a single argument of type \( \tau_1 \). For example, to represent success and failure of a computation, you could use the type \( \langle \text{success :: int}, \text{failure :: string} \rangle \), specifying that you can an int on success, and a string error message on failure. Are the types \( \tau_1 \) and \( \tau_2 \) in a covariant or contravariant position (i.e., what must be true for \( \langle l_1 :: \tau_1, l_2 :: \tau_2 \rangle <\langle l_1 :: \tau_1', l_2 :: \tau_2' \rangle \) to hold)? In one or two sentences, justify your answer.
In this lab, you will implement C++-style virtual dispatch in JavaScript. (A more “normal” language to do an exercise like this in would be C; however, we’re trying to avoid assigning this exercise to you in a memory-unsafe language.) The primary components of this lab will be the generation of virtual tables from class descriptions, as well as small (but critical) components of a mini-C++ calling convention.

**Overall structure**  The scaffolding code consists of the following files:

- **lib/class.js** The definition of the data type describing C++ classes.
- **lib/method.js** The definition of the data type describing C++ methods. Together with *lib/class.js* these files describe the input format of your program.
- **lib/layout.js** The definition of the data type describing virtual table entries and object layout.
- **lib/metadata.js** An API for describing the layout of virtual tables and other data about the compiled form of classes. Together with *lib/layout.js* these files describe the output format of your program.
- **lib/pointer.js** A low-level API for manipulating *typed array pointers* in JavaScript.
- **lib/abi.js** An API built on top of *lib/pointer.js* which implements the calling convention for your implementation of C++.
- **lib/compiler.js** The file where your compiler will go, which will create the virtual tables and layout information that *lib/abi.js* will use to run your programs.

In this lab, you will be editing *lib/abi.js* and *lib/compiler.js*, as well as examples/null_manual.js and examples/diamond_manual.js.

**Class syntax**  In this lab, you will be compiling C++ class descriptions into vtable descriptions, which, combined with the runtime we have provided, will allow you to run various programs interacting with C++ style classes. The subset of C++ you will be working with supports data members, virtual and nonvirtual methods and multiple inheritance. Your compiler will not need to support visibility modifiers, method overloading or virtual inheritance. Class declarations are encoded using the *Class* data type defined in *lib/class.js*. For example, the following C++ class:

```cpp
class Point {
public:
    int x, y;
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
    virtual void move(int dx, int dy) {
        this->x += dx;
        this->y += dy;
    }
}
```

would be encoded as an object with the following properties:

```json
{
    name: "Point",
    data: ["x", "y"],
```
virtual: [ Method("move", impl_Point_move) ],
nonvirtual: [ Method("Point", impl_Point_Point) ],
inherits: []
}

where impl_Point_move and impl_Point_Point are appropriate functions. (The Method structure is defined in lib/method.js and has two fields: name and impl). You can see more examples of encoded objects in examples/point.js and examples/multiple.js.

The Metadata API  The way you will compile these class descriptions is by making calls to the metadata API, which will keep track of the global state that constitutes the object layout you will be constructing. You will need to tell this API various things about object layout, including where to keep data fields, how the virtual table is setup, what the size of the object is, etc. Here is an example of all the calls to the metadata API you would need to make to fully lay out the Point object:

```javascript
Meta.setSizeof("Point", 3);
Meta.setParents("Point", []);
Meta.setGlobal("Point", "Point", impl_Point_Point);
Meta.setVirtualOffsets("Point", { "move": 0 });
var vtable = [ Layout.VEntry(impl_Point_move, "Point", 0) ];
Meta.setTemplate("Point", [Layout.Layout(0, "Point", vtable)]);
Meta.setCasts("Point", {});
```

Hopefully, these calls do not seem too obscure. Here are each of them in more detail:

**setSizeof** Takes the size of the object. In our ABI, the Point object has size three: one entry is used for the vtable pointer, while the other two entries are used for the x and y fields. In our implementation, we’ll assume each field uses exactly one unit of space.

**setParents** Takes the class names which the class inherited from. Since Point is a base class, the list is empty; however, the ColoredPoint example from lecture would have ["Point"].

**setDataOffset** Takes where the data for a given (fully-qualified) field identifier is stored. In our ABI, we will store the vtable pointer in the first slot (index 0); in which case the first data member is stored in index 1, the second in 2, and so on.

**setGlobal** Takes the implementation for any non-virtual methods, such as constructors, qualified by class name.

**setVirtualOffsets** Takes a mapping from virtual method names to an offsets, which are the position of the virtual method in the virtual table.

**setTemplate** Takes a list of Layout entries which describe how an object of this class is laid out in memory; specifically, each layout entry specifies where the virtual table pointer lives in an object. Each entry is a tuples consisting of (1) the offset in the object of the vtable pointer, (2) the most specific type which the vtable supports (e.g., for ColoredPoint this entry would be ColoredPoint) and (3) the actual vtable in question. Virtual tables are an array indexed by the virtual offsets, containing VEntries which record (1) the implementation of the method, (2) the type of this that the method is expecting, and (3) any delta offset necessary to adjust the this pointer before passing it to the function. In the previous example, Point has a single vtable pointer at position zero, and the table has a single virtual method for move that was defined in the Point class.

**setCasts** Takes a mapping from parent types to offsets; this offset is applied to the pointer if you cast to one of the parent types.

There are a few other variants of these functions, as well as equivalent functions to retrieve this information which you may find useful; check lib/metadata.js for full information.
The ABI  We will assume the following conventions for how we lay out objects, which will closely resemble the layouts you saw in lecture:

- Every object always has a vtable pointer (even if the vtable is empty).
- The vtable pointer is stored in the initial position of the object.
- Data fields are laid out in the order they are declared, after the vtable pointer.
- When a class inherits from a single class, any new data members and virtual functions are placed after the existing members and functions of the parent class. This means that a cast up a single inheritance chain is a no-op.
- When a class inherits from multiple classes, each parent class is laid out in the order they are listed in the inheritance list (and then any new data members are laid out), with a vtable pointer per object. Any new virtual functions are placed in the vtable of the first parent class, so casting up to the first parent is a no-op.

Typed array pointers  Since this lab is not written in C, we need to simulate pointer arithmetic in order to accurately implement code which interacts with C++-style object layout. Our implementation of typed array pointers (Ptr objects) can be found in lib/pointer.js. Here is a summary of the API (methods on Ptrs):

```javascript
// Get the class name associated with the pointer
ptr.type

// Write val to the location pointed to by ptr + offset, e.g. *(ptr + offset) = val;
ptr.write(offset, val)

// Dereference the location pointed to by ptr + offset, e.g. *(ptr + offset)
ptr.deref(offset)

// Add some offset to a pointer, returning a new pointer which is backed by the same array and with new type ty, e.g. (ty*)(ptr + offset)
ptr.add(offset, ty)

// Creates a null pointer of some type, e.g. (ty*)null
ptr.null(ty)
```

Task 1: Finish the ABI  In lib/abi.js, there are methods which implement various operations on C++ object pointers which are common. However, the implementations for setMember (data member write), getMember (data member read), call (non-virtual method call) and vcall (virtual method call) are incomplete. Complete these definitions, using the information available from Meta.

- You may find Meta.getDataOffset(type, member), Meta.getVirtualOffset(type, method) and Meta.getGlobal(type, method) useful.
- In most of the functions, we will have provided in the local scope a ptr representing the object in question. You may assume that this pointer has a type which the member or virtual method was originally defined in (ensuring this is the case was done by the invocation of autocast, which implements C++ implicit casting rules, although for this lab you don’t need to know how this is done.)
• When implement `vcall`, remember that a virtual call may need to adjust the `this` pointer before passing it along to the function body (`shifted_ptr`). The necessary offset and new type can be found in the `VEntry` in the vtable as `ventry.type` and `ventry.delta`, and you can add an offset to a pointer using the `add` method. The test script will let you know if you’ve done this incorrectly.

The test script `task1.js` uses the hand-coded layout in `examples` to run some test scripts. When you get the code working, you should get the following output from the script:

```
-- point -------------------------------
{ array: [ [ [Object] ], 4, 6 ], offset: 0, type: 'Point' }
{ array: [ [ [Object], [Object] ], 6, 9, 2 ],
  offset: 0,
  type: 'ScaledPoint' }
-- multiple ---------------------------
C::f()
0
1
2
B::g()
1
C::f()
0
1
2
C::f()
0
1
2
B::g()
1
```

**Task 2: Null pointers**  In this task, we will explore what happens when virtual and nonvirtual method calls are made on NULL pointers. In `examples/null.js`, there is a class-definition and function implementations corresponding to the following C++ code:

```cpp
#include <iostream>
using namespace std;

class Widget {
public:
  int b;
  Widget(int b) { this->b = b; }
  void output1() { cout << "Non-virtual fn" << endl; }
  void output2() { cout << "Non-virtual fn: b=" << b << endl; }
  virtual void outputV() { cout << "Virtual fn: b=" << b << endl; }
};

int main(void) {
  Widget* a = new Widget(2);
  a->outputV();
  return 0;
}

int main_outputV(void) {
  Widget* a = NULL;
  a->outputV();
}  ```
Fill out the class layout in examples/null_manual.js by making calls to Meta so that task2.js can run on all of the functions. You may find it helpful to consult examples/point_manual.js and examples/multiple_manual.js for more examples of how to use this API.

Task 2: Null pointers explanation  In the locations indicated in examples/null_manual.js, say what the runtime behavior of each of the four functions is, and if there is an error, explain why. Be specific. You may find it useful to cross-check the output of these functions against a real C++ compiler.

Task 3: Diamond inheritance  In this task, you will hand-compile the following set of class definitions in examples/diamond.js:

```
#include <iostream>
using namespace std;

class A {
public:
  int x;
  A(int x) { this->x = x; }
};
class B : public A {
public:
  int y;
  B(int x, int y) : A(x) { this->y = y; }
};
class C : public A {
public:
  int z;
  C(int x, int z) : A(x) { this->z = z; }
};
class D : public B, public C {
public:
  D(int x, int y, int z) : B(x,y), C(x,z) {}
};

text main(void) {
  D* d = new D(1, 2, 3);
  B* b = d;
  C* c = d;
  b->x = 9;
  cout << c->x << endl;
```
return 0;
}

Note: virtual inheritance was not used. In examples/diamond_manual.js, fill in the class layout so that task3.js can run. (We've provided calls to do function layout, so you don't have to worry about those.) You may find it useful to cross-check the output of these functions against a real C++ compiler.

Task 4: Implement the compiler In lib/compiler.js, there is scaffolding code for the compiler proper, which will convert class descriptions into object layouts (stored by Meta). Implement compileClass. For this task, you can assume that the number of parent classes in inherits is either zero (base class) or one (single inheritance): you do not need to implement multiple inheritance. You may find the copy method on the Layout object useful when updating virtual tables with new virtual methods.

When you are done, you should be able to run ./cli.js point and ./cli.js null. In cli.js, there is also a commented out line Meta.dump(). If you uncomment it, the program will dump all of the internal state recorded by the metadata API. You can use this to find out what your code is doing, and cross-check it with the manually written examples in examples. There are also some tests in tests. You can run them using the tester.sh script.

Bonus: Implement multiple inheritance Extend your compiler to support multiple inheritance. When you are done, you should be able to run ./cli.js multiple and ./cli.js diamond. You can also enable the multiple inheritance tests by editing tester.sh.

Our model solution is 70 lines long and strictly generalizes the compiler from Task 4. The trickiest part of the code is virtual pointer layout (setTemplate); you will need to go through all of the virtual pointers for the parent classes and update their virtual tables with any new virtual functions defined by the current class; one strategy which works is to copy the Layout and offsets (we've provided a helper function copyOffsets which you might find useful) and then update them with the new information.