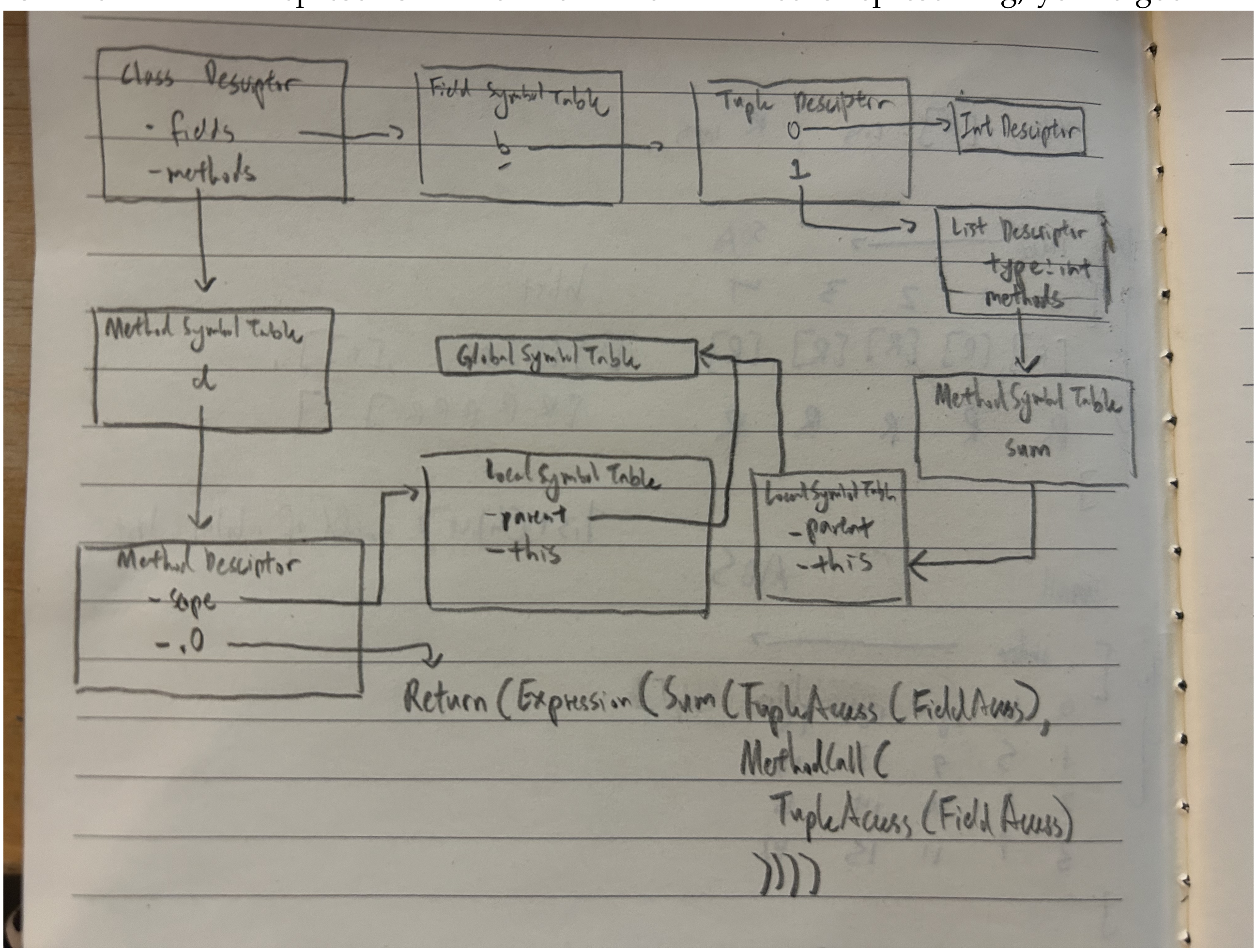
# 1 Intermediate Representation

Suppose we have the following code:

```
class A {
    b: (int, list<int>)
    int d(this) {
        return this.b.0 + this.b.1.sum();
    }
}
```

Write down a representation for this code in an IR of your choosing. Optionally, represent it using symbol tables and descriptors.

Something like this would suffice. On a quiz you don't have to match this exactly. As long as have an IR that represents all the information that needs representing, you're good.



## 2 Subclassing Semantics

Suppose we have the following class definitions:

```
class Rocket { .. }
class Spaceship extends Rocket { .. }
```

And the following code:

```
Rocket r;
Spaceship s;

fn launch(Rocket) -> Rocket { .. }
fn upgrade(Rocket) -> Spaceship { .. }
fn reboot_flight_software(Spaceship) -> Spaceship { .. }
fn retire(Spaceship) -> Rocket { .. }
```

Which of the following calls are valid?

- ☐ `Spaceship s' = launch(s);`
- ☐ `Rocket r' = upgrade(r);`
- ☐ `Spaceship s' = reboot_flight_software(r);`
- ☐ `Spaceship s' = retire(r);`

**Answer:**

`Spaceship s' = launch(s);` is fine as spaceships can be used anywhere rockets are needed.

`Rocket r' = upgrade(r);` is fine because the return type of `upgrade` is `spaceship`, which is a type of `rocket`

`Spaceship s' = reboot_flight_software(r);` is not fine as the argument to `reboot_flight_software` has to already be `Spaceship` (since apparently `Rockets` don't have flight software), and not all `Rockets` are `Spaceships`.

`Spaceship s' = retire(r);` is also not legal as only `Spaceships` can be retired, but not all `Rockets` are `Spaceships`. Also, all we know about a retired `Spaceship` is that it is a `Rocket`, so we cannot necessarily assign to returned value to a `Spaceship`.

# 3 Short Circuiting

Consider the following code, which is similar to Decaf and has similar precedence rules:

```
bool took_shower = false;
bool take_shower() {
    took_shower = true;
    return true;
}

bool touched_grass = false;
bool touch_grass() {
    touched_grass = true;
    return false;
}
```

---

Now, consider the following condition:

```
bool compiler_working = touch_grass() || take_shower();
```

**Once the student gets their compiler working, have they touched grass and showered? (what are the values of `took_shower` and `touched_grass`)? [yes/no]**

**Answer:** Yes, as the call to `touch_grass` returns false so the student evaluates the right side of the condition (showers).

---

Next year, the student's friend, who is slightly more hygienic, decides to shower first.

```
bool compiler_working = take_shower() || touch_grass();
```

**Does this student end up having to touch grass to get their compiler working? [yes/no]**

**Answer:** No, as the call `take_shower` returns true, so evaluation of the or stops (no touching grass).

---

In the third iteration of the class, the professor greatly boosts the difficulty, but also hosts office hours to compensate.

```
bool went_to_oh;
bool go_to_oh() {
    went_to_oh = true;
    return true;
}
bool compiler_working = take_shower() && go_to_oh() || touch_grass();
```

**Now, has the student touched grass by the time they get their compiler working? [yes/no]**

**Answer:** No, as logical and has higher precedence than logical or, so taking a shower and going to OH is enough.

Finally, consider a slightly different situation:

```
bool compiler_working = take_shower() && (go_to_oh() || touched_grass());
```

**Now, has the student touched grass? [yes/no]**

**Answer:** No, as the logical or stops evaluating once the student goes to OH.