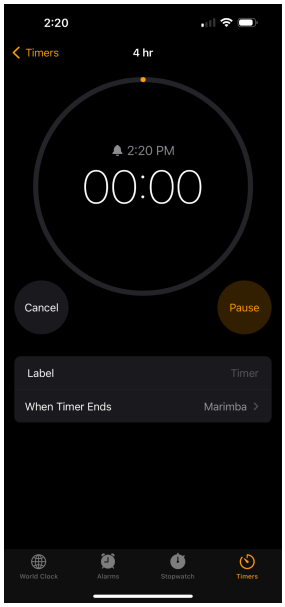
Group Members: Ella, Ben, Grace
Timer: Saturday, December 13, 10:15 am to 2:20 pm.



1. The core interpreter for behami was implemented in Elixir. We focused on evaluating the core language, and we ran out of time to fully complete the parser or desugaring phase, although we started it. Our goal was to stay as close as possible to the Project B behami interpreter, and that is what mainly shaped both the structure and design choices. It was easier to find similarities in the languages this way.
   The interpreter supports numbers, booleans, variables, binary operations, conditionals, let bindings, lambdas, closures, and function applications. Short circuit behavior for && and || was implemented directly in the interpreter.
   We weren't able to fully complete the parser, the desugaring phase, and some tests because we completed the 4 hours.
   Behami components completed:
   - Core expression AST representation
   - Value representation with closures
   - Environments
   - Interpreter for the core expressions
   - Binary ops with runtime error handling
   - Short circuiting for && and ||
   - Unit tests for core structure, values, operations, and interp
   - Partially the parser, named ElixhamiSyntax
   - Elixhami

   Code Highlight we're proud of (Biop in interp, courtesy of Grace):

```
{:biopCE, op, l, r} ->
    cond do
        op == :"&&" ->
            lv = interp(l, env)
            case lv do
                {:boolV, ln} ->
                    if ln do
                        rv = interp(r, env)
                        case rv do
                            {:boolV, rn} -> {:boolV, ln and rn}
                            _ -> raise "expected operand to be a boolean"
                        end
                    else
                        {:boolV, false}
                    end
                _ -> raise "expected operand to be a boolean"
            end
        op == :"||" ->
            lv = interp(l, env)
            case lv do
                {:boolV, ln} ->
                    if ln do
                        {:boolV, true}
                    else
                        rv = interp(r, env)
                        case rv do
                            {:boolV, rn} -> {:boolV, ln or rn}
                            _ -> raise "expected operand to be a boolean"
                        end
                    end
                _ -> raise "expected operand to be a boolean"
            end
        true ->
            BiopModule.biop(op, interp(l, env), interp(r, env))
    end
```

2. Elixir was interesting because of its immutable data structure and pattern matching. However, it was kind of similar to Plait in the sense of nodes and values. It was rather simple to look at Project B, the documentation for Elixir, and find the pieces of the puzzle that would do the same thing. Maps in Elixir were super helpful and were a perfect fit for environments, basically replacing the Plait hash tables with ease. Case in Elixir was also very similar to Plait's type-case, and that was easy to translate over to Elixir. Very thankful Elixir had good documentation!

Easier than Plait:

- Pattern matching was easy
- Maps for environments straightforward
- Translating core ideas from Project B was good

Harder than plait

- No static type checker, dynamic only = runtime errors
    - Meaning that type errors were found later
- Manual error checking in tests
- Does not allow mutable variables

Elixir doesn't enforce static type checking the same way Plait does, so we had to have type correctness at runtime instead of earlier, meaning all of our checks had to be interp. However, Elixir's library made operations like mapping and building environments easier to build.

Differences in implementation/syntax (Elixir vs. Racket)

- Symbols were represented using :
- Tests existed in their own separate "environment" (e.g defined an empty env for an individual closure test case)
- Each test had its own name
- Everything was implemented using block expressions (do..end)

3. This project showed how much the language you use affects how the interpreter turns out, even when the goal is the same. It also made the ideas more concrete, and the design choices easier to understand. This language especially made us realize just how important strong type checks are!

   Around the idea of type checking, writing behami in Elixir made Plait's type system so much more helpful and valuable, especially with a limited amount of time.

   All that said, I don't think this language was that bad and I wouldn't mind using this language again, mainly because of its similarity to Plait because they are both functional, rely on pattern matching, and representing AST nodes feels natural in both.