A4 Handout

In this assignment, you will build an interpreter for an OCaml-like language called RML (Robot Meta-Language). (https://canvas.cornell.edu/courses/48987/pages/jocalf-semantics)

Overview

What you'll do: You will implement the evaluation function for the interpreter of RML and implement some simple programs using the asynchronous features of RML.

Objectives:

- Implement an interpreter for a non-trivial programming language.
- Gain experience with concurrent programming using promises.

Collaboration policy: This assignment is to be completed as an individual. Nonetheless, I encourage *limited* collaboration with other students as described in the syllabus.

Time: A good plan would be to allocate one hour on average per non-vacation weekday that the assignment is out. Please track the time that you spend. There will be a place to report it in your submission.

FAQs: There is a pinned FAQ in Ed for this assignment. Please consult it before posting a question, just in case your question has been frequently asked and answered.

Step 0: Learn About RML

RML is a simple programming language that combines functional, imperative, and concurrent features. The language is described in detail below. We have provided most of the front-end as well as a simple type-checker to help you develop RML code and debug your interpreter more effectively.

The Interpreter

The RML interpreter has the following phases:

1. Lexing and Parsing
   - `lexer.mll`: Translates an RML program into a stream of tokens
   - `parser.mly`: Translates a stream of tokens into an abstract syntax tree (AST).
2. Static Analysis
   - `types.ml`: Types and common error messages
   - `checker.ml`: Type checking function
   - `ast_factory.ml`: Functions for building AST values
3. Evaluation

- `ast.ml`: Types representing expressions (an AST), patterns, definitions, etc.
- `eval.ml`: Big-step evaluation function
- `promise.ml` : Helper functions for evaluating asynchronous expressions.

Preprocessing

In addition to the type-checker, we have implemented a basic pre-processor for RML files. Writing `include "<path>"` at the top of an RML file introduces the definitions from the file at `<path>` into the namespace of the current file. Includes are transitive, so if `a.rml` includes `b.rml`, and `b.rml` includes `c.rml`, then the definitions in `c.rml` will be in the scope of `a.rml`. Cyclic dependencies will cause the preprocessor to raise an exception. The path mentioned in an `include` must either be a filename in the current directory or a relative path to a filename. For example, a relative path `"../../folder1/folder2/file.rml"` indicates a file that can be found by navigating two folders up, and then down into two different sub-folders. Includes are only permitted at the top of the file, before any definitions.

As an example, given files,

```
// library.rml
let (x : int) = 0 in
```

and

```
// include.rml
include "library.rml"
let () = println x
```

the preprocessor will turn the above code into the program:

```
let (x : int) = 0 in
let () = println x
```

Note that the type annotation on `x` is required in RML: all variables in let-bindings need explicit type annotations.

The RML Language

A complete list of all RML expressions is given below. A formal definition of the operational semantics of RML can be found ***here*** **(https://canvas.cornell.edu/courses/48987/files/8056267/download?download_frd=1)** . However, we recommend reading the description here first to gain an intuitive understanding of the language.

Grouping and Comments

The concrete syntax of RML allows grouping expressions using parentheses, as well as `begin .. end`, similar to OCaml.

Comments are written similar to C and Java: use `//` for a single-line comments and `/* .. */` for multi-line comments.

Values

RML values include the following:

Value	Semantics
`()`	Unit value
`n`	Integer values
`s`	String values
`true` and `false`	Boolean values
clos	Function closures, which can be understood as a triple of the form `cl(p,e,env_cl)` that encodes a function that matches pattern `p` to its argument to obtain bindings, updates the defining environment `env_cl` with these bindings, and finally evaluates `e` in the resulting environment.
`(v1, v2)`	Pair values
`[]`	Empty list value
`v1 :: v2`	Non-empty list values
prom	Promises (result of async operations, as described below)
loc	Locations (result of `ref` operations, as described below)
hand	Robot handles (result of `spawn` operations, as described below)

Expressions (Functional)

RML expressions can be divided into several categories, starting with purely-functional expressions.

Expression	Semantics
`()`	Unit literal
`n`	Integer literal
`s`	String literals (note: **use double quotes only**; RML does not support escapes)
`true` and `false`	Boolean literals
`x`	Variables, which are evaluated by looking up their binding in an environment
`(e1, e2)`	Pairs, which evaluate to `(v1, v2)` if `e1` evaluates to `v1` and `e2` evaluates to `v2` (in that order)
`[]`	Empty list literal
`e1 :: e2`	Non-empty lists, which evaluates to `v1 :: v2` if `e1` evaluates to `v1` and `e2` evaluates to `v2` (in that order)

Expression	Semantics
`uop e`	Unary operations, which evaluates to `v` if `e1` evaluates to a value `v1` and `uop v1` is `v`
`e1 bop e2`	Binary operations, which evaluates to `v` if `e1` evaluates to `v1` and `e2` evaluates to `v2` (in that order) and `bop v1 v2` is `v`
`if e1 then e2 else e3`	Conditionals, which evaluate `v` if `e1` evaluates to `true` and `e2` evaluates to `v` or if `e1` evaluates to `false` and `e3` evaluates to `v`
`e1; e2`	Sequences, which evaluates to `v` if `e1` evaluates to `()` (possibly causing side effects) and `e2` evaluates to `v` (in that order)
`let (p : t) = e1 in e2`	Let expressions, which evaluate to `v` if `e1` evaluates to `v1` and evaluating `e2` in the environment extended with the bindings obtained by matching `p` with `v1`.
`let rec f (p : t) : t' = e1 in e2`	Recursive functions, which evaluates to `v` if `e2` evaluates to `v` in the environment extended so `f` maps to the closure `clos` = `cl(p, e1, env_cl)` Note that because `e1` may refer to `f`, the environment `env_cl` must also be updated to include the binding `f => clos`. Because `env_cl` needs to contain `f`, we will model the closure environment as a reference. When evaluating a recursive function, this allows us to first initialize the closure environment with the defining environment and then "back-patch" it with the binding `f => clos`.) Note that the parser desugars `let rec f p = e1 in e2` into `let rec f = fun p -> e1 in e2`.
`fun (p : t) -> e`	Anonymous functions, which evaluate to a closure `cl(p,e,env_cl)` containing the pattern, body, and current environment (i.e., RML uses lexical scope)
`e1 e2`	Function applications, which evaluate to `v` if `e1` evaluates to a closure `cl(p,e,env_cl)`, `e2` evaluates to a value `v2`, and evaluating `e` in `env_cl` extended with the bindings obtained by matching `p` with `v2` evaluates to `v`.
`match e0 with | p1 -> e1 | ... | pn -> en end`	Pattern matching, which evaluates to `v` if `e0` evaluates to `v0`, `pj` is the first pattern that matches `v0`, and evaluating the corresponding body `ej` in the environment extended with the bindings obtained by matching `pj` with `v0` evaluates to `v`. **(Note the keyword `end` to denote end of match statements).** If `v` does not match any pattern `pj`, the interpreter should raise `InexhaustivePatterns`.

Unary Operators

RML includes a few unary operators:

Operator	Semantics
`-e`	Integer negation, which evaluates to `-n` if `e` evaluates to `n`.

Operator	Semantics
`not e`	Boolean complement, which evaluates to `true` if `e` evaluates to `false`, and vice versa.

Binary Operators

RML also includes a number of standard binary operators:

Operator	Semantics
`e1 + e2`, `e1 - e2`, `e1 * e2`, `e1 / e2`, `e1 % e2`	Arithmetic operators (Note the `%` is the "mod" operator)
`e1 < e2`, `e1 > e2`, `e1 <= e2`, `e1 >= e2`	Comparisons on integers
`e1 = e2`, `e1 <> e2`	Equality and inequality on integers, strings, and booleans.
`e1 && e2`, `e1 || e2`	Boolean operators, which behave as in OCaml, including short-circuiting
`e1 ^ e2`	String concatenation operator
`e1 |> e2`	Function pipe-lining operator, which behaves the same as `e2 e1`.

Patterns

RML includes patterns, which can be used in `let` and `match` expressions as well as anonymous functions. Matching a value against a pattern can either succeed, in which case it produces a binding for each variable occurring in the pattern, or it can fail.

Pattern	Semantics
`_`	Wildcard pattern, which matches any value `v` and produces no bindings
`c`	Constant pattern, where `c` is a unit, integer, boolean, or string literal, which matches only itself and produces no bindings
`x`	Variable patterns, which matches any value `v` and produces the binding `x => v`
`(p1, p2)`	Pair patterns, which matches a pair `(v1,v2)` if `p1` matches `v1` producing bindings `b1` and `p2` matches `v2` producing bindings `b2`, and produces the bindings in both `b1` and `b2`. **(Note that if a variable is bound in both `b1` and `b2`, the second binding takes priority)**.
`[]`	Empty list pattern, which matches `[]` and produces no bindings
`p1 :: p2`	Non-empty list pattern, which matches `v1::v2` if `p1` matches `v1` producing bindings `b1` and `p2` matches `v2` producing bindings `b2`, and produces

Pattern	Semantics
	bindings `b1` and `b2`. **(Again, if a variable is bound in both `b1` and `b2`, the second binding takes priority.)**

Expressions (Imperative)

RML also includes imperative constructs analogous to OCaml's references:

Expression	Semantics
`ref e`	References, which evaluate to `loc` if `e` evaluates to `v` and `loc` is a newly created location that points to `v`
`!e`	Dereferences, which evaluates to `v` if `e` evaluates to a location `loc` that points to `v`
`e1 := e2`	Assignment, which evaluates to `()` and updates `loc` to point to `v` if `e1` evaluates to `loc` and `e2` evaluates to `v`

Note that the files `eval.ml` and `ast.ml` provided in the release code do not have any explicit references to state (unlike the formal semantics, which threads `sigma` through the evaluation). To simplify your solution, we recommend using OCaml's imperative features to implement the semantics of imperative expressions.

Expressions (Asynchronous)

Finally, RML also includes asynchronous expressions, including `spawn` which forks off a new robot executing asynchronously, `send` and `receive` primitives for communication between robots using *handles*. RML supports sending and receiving string messages. Most of these asynchronous constructs return *promises*, so RML also includes monadic constructs like `>>=` and `return` for computing with promises.

Expression	Semantics
`self`	The handle of the current robot. This allows robots to carry their own handles and pass their handles to robots they spawn.
`return e`	Returns, which evaluates to a promise that is resolved to `v` if `e` evaluates to `v`.
`await p = e1 in e2`	Awaits, which evaluates to a promise `prom` if `e1` evaluates to a promise `prom1` and when `prom1` resolves to `v1`, `v1` is bound to `p` and `e2` is evaluated under the new bindings to `prom2`, and finally when `prom2` eventually resolves to `v2` then `prom` also resolves to `v2`. **(Note the similarity between `await p = e1 in e2` and OCaml's `Lwt.bind e1 (fun p -> e2)`.)**

Expression	Semantics
`e1 >>= e2`	Binds, which evaluates to a promise `prom` if `e1` evaluates to a promise `prom1` and `e2` evaluates to a closure `f`, and when `prom1` resolves to `v1`, the closure for `f` is applied with argument `v1`, resulting in a promise `prom2`, and finally, when `prom2` eventually resolves to `v2`, then `prom` also resolves to `v2`. The type-checker will fail if `e1` or `e2` do not have the appropriate types. Note again the similarity between `e1 >>= e2` and `Lwt.bind`.
`send e1 to e2`	Sends, which evaluates to unit and sends message `v` to handle `hand` if `e1` evaluates to `v` and `e2` evaluates to `hand`.
`recv e1`	Receives, which evaluates to a promise `prom` that resolves to the message sent to the current robot from `hand` if `e1` evaluates to `hand`.
`spawn e1 with e2`	Spawns, which evaluates to a new handle `hand` if `e1` evaluates to function closure `f`, `e2` evaluates to a value `v`, and a new robot is created running `f v` at handle `hand`.

Async Implementation Hints. The heavy-lifting for evaluating asynchronous expressions has been encapsulated in a helper module `Promise.Mwt`, implemented in `promise.ml`. Think of `Mwt` as a run-time library that provides a datatype `Mwt.t` representing a promise and most of the necessary operations on promises. The implementation of `Mwt` follows **Chapter 8.6.3 of OP (https://cs3110.github.io/textbook/chapters/ds/promises.html#id1)** and **Chapter 8.6.4 of OP (https://cs3110.github.io/textbook/chapters/ds/promises.html#making-our-own-promises)**. **We highly recommend going through these subsections slowly to understand how promises are implemented.**

- For implementing `bind`, you'll want to take the RML promises and package them into `Mwt` promises. Then you can use the bind from `Mwt`, and then wrap the resulting `Mwt` promise back into a RML promise. The `await` expression is very similar.
- For implementing `send` and `recv`, you'll need to implement three channel functions in `Mwt`. The high level idea is that a channel holds two pieces of data: a queue of messages, and a queue of resolvers, i.e., tasks that are waiting for a message to arrive on this channel. When implementing send on a channel, you should check if anyone is waiting and directly resolve the promise if so. If no one is waiting, the message goes into the queue. The `recv` promise is similar: get a message if there is a message in the queue, otherwise make a promise and add a resolver to the channel.

Built-in Functions

You are responsible for implementing the following built-in functions, when you implement function application. This will involve extending the definition of `initial_env` in `eval.ml`. Thus, if the user explicitly rebinds the name of some built-in function to a different, that binding should take precedence.

Function name	Semantics
`print e`	`print e` evaluates `e` to `v`, `v` is converted to string `s` via `Eval.string_of_value`, then `s` is printed and a unit is returned.
`println e`	`println e` evaluates `e` to `v`, `v` is converted to string `s` via `Eval.string_of_value`, then `s` followed by a newline character is printed and a unit is returned.
`int_of_string e`	`int_of_string e` evaluates `e` to a string `s`, which it then converts to an integer `n`. You may follow the semantics of the OCaml built-in function with the same name.
`string_of_int e`	`string_of_int e` evaluates `e` to an int `n`, which it then converts to a string `s`. You may follow the semantics of the OCaml built-in function with the same name.

Syntactic Sugar

RML supports several forms of syntactic sugar. You do not need to do anything to implement these cases; they are desugared by the lexer and parser for you into one of the cases above. We mention this purely for your reference as you are coding in RML.

Syntactic Sugar	Desugars to:
`[v1;...;vn]`	`v1 :: ... :: vn :: []`
`[p1;...;pn]`	`p1 :: ... :: pn :: []`
`let x p1 ... pn = e1 in e2`	`let x = fun p1 -> ... -> fun pn -> e1 in e2`
`let rec f p p1 ... pn = e1 in e2`	`let rec f p = fun p1 -> ... -> fun pn -> e1 in e2`

Step 1: Get Started

Download the release code. There are many files, and you will need to read many of them before the assignment is over, but you don't have to do so yet. Nevertheless, here is an overview to the files:

- `promise.ml` and `promise.mli`: **(You will need to change this file.)** This module provides helper functions that will be useful in your implementation of asynchronous expressions. You should familiarize yourself with each of the functions in `promise.mli` before starting work on the concurrent features of RML.
- `eval.ml`: **(You will need to change this file.)** This module implements the RML interpreter. It currently has four main unimplemented functions along with several unimplemented helper

functions and stubs for the type `value`. You should fully implement these functions. You are also free to add any extra helper functions that you find useful.

- `lexer.mll`: This module implements a lexer that converts RML programs from a character stream into a token stream.
- `parser.mly`: This module implements a parser that converts the token stream for an RML program into an AST.
- `ast.ml`: **(You will need to change this file.)** This module defines the types for ASTs. Currently, most of these types are stubbed out as `unit`. You will decide how to design the AST and provide various operations on these types.
- `ast_factory.ml` and `ast_factory.mli`: **(You will need to change this file.)** This module provides "factory" methods that the parser and type checker can use to build ASTs. As you choose the types of the ASTs, you will need to fill in the functions in this module.
- `test/main.ml`: **(You may change this file.)** This module provides tests for `eval.ml`. You may find it useful to develop a test suite, however you do not need to include it in your submission.
- `Makefile`: The Makefile, which provides a number of targets as described below.
- `rml/examples`: This directory contains examples of RML code that you may look at in order to get a feel for how the language works. You do not need to understand these functions or modify them in any way, although your final implementation should be able to run them without crashing.
- `rml/async`: **(You may change these files.)** This directory contains the asynchronous functions you will implement in the optional exercises.
- `rml/test`: **(You may change these files.)** This empty directory is provided as a "scratch space" for writing your own RML test programs.

Create a new git repository for this assignment. Make sure the repo is private. Add the release code to your repo, referring back to the instructions in A1 if you need help with that. Make sure you unzip and copy the files correctly, including hidden dotfiles, as described in the A1 handout.

In the release code, there is a Makefile provided with the usual targets:

- `make build`: compile
- `make utop`: load project in utop
- `make test`: compile and run the test suite
- `make run`: compile and prompt for a file on which to run the interpreter
- `make repl`: compile and run an interactive interpreter shell for RML
- `make check`: check submission
- `make finalcheck`: final check for submission
- `make clean`: delete compiled files
- `make zip`: create the `.zip` file for final submission

As usual, you may not change the names and types of the files in the provided modules. The only files your should change are `ast_factory.ml`, `ast.ml`, `eval.ml`, `promise.ml`, and possibly `test/main.ml`. You should follow the instructions carefully to make sure you do not change any of our code.

This assignment will be autograded, so your submission must pass `make check`. Now that you're used to working with `.mli` files, we have omitted the comments about what you are allowed to change. Any names and types that we provide in an interface may not be changed, but you may of course add new declarations and definitions. If you're ever in doubt about whether a change is permitted or not, just run `make check`: it will tell you whether you have changed the interface in a prohibited way.

Step 2: Implement an Evaluator!

Implement an evaluator for RML by filling in the missing code in `ast.ml`, `ast_factory.ml`, `eval.ml`, and `promise.ml`. You are free to design your own types to represent ASTs in `ast.ml`. However, you must fill in the code in `ast_factory.ml` so the parser can construct them.

Implementation order.

- **Satisfactory scope:** This level of scope corresponds roughly to the language features we covered in the textbook, lecture, and recitation. You are of course free to re-use any of that code we gave you.
  - Constants: integers, strings, Booleans.
  - Three basic operators: the addition operators on integers (e.g., `1+1`), and the short-circuit Boolean operators (e.g., `true && false`, `false || false`). You can leave the rest of the binary operators, for Good scope.
  - Let expressions and definitions. This will require you to get environments working. Don't worry about recursion or back-patching closure environments yet; leave that for Excellent scope, and just do non-recursive `let` for now.
  - Anonymous functions and function application with variable patterns. This will require you to get closures working.
  - If expressions.
- **Good scope:** This level of scope involves adding mutability to the Satisfactory scope. It involves new features that we didn't cover in class. The challenge here, therefore, is to use the formal semantics to guide your implementation.
  - Constructing data types: pairs, lists, etc.
  - Sequences (i.e., semicolon).
  - References. This is the first feature that will cause you to modify state in a non-trivial way.

- The remaining basic operations.

- **Excellent scope:** As always, you should regard the Excellent scope as a chance to dig deeper, rather than something mandatory.
  - Asynchronous expressions, `return`, `bind`, `send`, `recv`, `self`, and `spawn`. To handle `send` and `recv`, you will need to complete the `Mwt` implementation in `promise.ml` to support asynchronous channels. (Note: implementing the `Fwt` module is **not required** for Excellent scope.)
  - All remaining features including support for pattern matching, recursive definitions, `await`, etc.

The autograder test suite is engineered to do a good job of getting you partial credit on any language features you implement, as long as you follow the above order. But it's not always possible to test each language feature completely in isolation, and of course we also need to test features in conjunction with one another to see whether they work together properly.

You should build this one language feature at a time. For language features that are unfamiliar or complex, consult the formal semantics carefully.

Implementation strategy. Here is a model work-flow you can follow to implement the interpreter.

- Pick a language feature to implement such as boolean literals.
- Implement a couple test cases for the feature in `test/main.ml`.
- Extend the AST type in `ast.ml` with a constructor for that feature–e.g. `Bool of bool` in `expr`.
- Implement the factory function in `ast_factory.ml`–e.g. `let make_bool b = Bool b`
- If necessary, extend the `value` type in `eval.ml` with a constructor. Also, make sure that your implementation of `string_of_value` handles the new value correctly.
- Implement the big-step rule for the new language feature from the formal semantics.
- Fully develop your test suite with more comprehensive test cases in `test/main.ml`. You can also try out the new feature in the REPL.

You will need to do this for patterns, expressions, and definitions. A few notes on each:

- **Patterns** will be a bit more challenging since it will be different from features we have done so far. Notice that patterns are not just used in pattern matching, but also in place of normal variables in `let` and `fun` expressions. This means that patterns need to be at least partially supported in order to get some of the basic language features to work. We recommend getting basic variable patterns to work first, and then adding more in later once you get to pattern matching. Also, the `bind_pattern` function returns an option. This is because a value may not match a pattern in a `match` expression. In theory, such a mismatch could occur in a `let` expression or a function application. However, our type-checker rejects such programs so you don't have to handle this case.

- **Expressions** are the main part of the interpreter. Recursion and concurrency are intended to be an extra challenge, so you should save those for last.

- **Definitions** will be the shortest part, and should be very similar to `let` and `let rec` expressions. However, you will not be able to run your interpreter on interesting examples in the REPL or using `make run` until definitions are at least partially done, so this should be filled in relatively early. We recommend doing so around the same time you implement `let` expressions.

Step 3: Optional Karma Problems

After your interpreter is complete, you may find it fun to use the language you have implemented to gain some experience implementing basic concurrency tasks and extending the promises library as **optional** karma problems. **Caution:** we recommend not attempting this section until you have completed the rest of your interpreter and are reasonably certain it is correct.

Exercise 1: The Promised Reference

In this exercise, you will use RML's promises to simulate the behavior of an RML reference. You implementation should not make use of the imperative feature of RML, only the concurrency.

We will implement this only for integer references. We have provided the implementation of the following function:

```
val promise_reference : int -> handle -> unit promise
```

Intuitively, `promise_reference v h` is a function that simulates the behavior of of a reference where `v` is the value currently stored in the reference and `h` is the handle on which the reference expects to receive dereference and assignment commands.

If you examine our implementation, you will see that it receives and responds to messages on the input handle. A `deref` message causes the promise reference to send its current value back to the handle, while an `assign` message causes the promise reference to change its current value by recursively calling itself with the new value.

Given this implementation, your job is to implement three functions that act as an interface for `promise_reference`. These functions correspond to the `ref` keyword, the `:=` binary operator, and the `!` unary operator, respectively:

- `val ref_async : int -> handle`
- `val assign_async : handle -> int -> unit`
- `val deref_async : handle -> int promise`

For `ref_async`, you will have to come up with the proper way to spawn an instance of `promise_reference` and return the resulting handle. For `assign_async`, you will simply need to make use of `send` to communicate to the instance that it should change its value. For `deref_async`, you will need to make use of `send` and `recv` to communicate with the instance and return a promise to the value the instance contains.

The release code for this exercise can be found in `rml/async/prom-ref`. This directory contains the file `prom-ref.rml` which implements `promise_reference` as above. It also contains `ref.rml`, `assign.rml`, and `deref.rml`. You will implement the three functions above in the respective files. Finally, `test.rml` includes the four functions and runs a basic program to test that your implementations are working. Full specifications of all these components can be found in the files.

Exercise 2: Human-In-The-Middle

Two students, Mike and John, are working together on A5 for their favorite CS class. Due to unforeseen circumstances, they have to work remotely. To make matters worse, there is a bug in the network they are using so that they can't communicate directly. Mike and John must send messages to each other with the help of their favorite TA, Timmy.

The files for this exercise are in `rml/async/routing`. To start, take a look at `main.rml`, which does some simple setup work before returning unit. It spawns robots for Mike, John, and Timmy. You will notice that Mike and John do not have each other's handles, so direct communication is impossible. However, they each have the handle for Timmy, and Timmy has both handles.

We have implemented simple functions for Mike and John so that they are able to have a brief conversation about their project as long as Timmy is helping them. Check out these implementations in `mike.rml` and `john.rml`. Right now, however, Timmy is busy and cannot help out yet. You can see in `timmy.rml` that he is not doing anything useful. Looking at these functions, try checking your understanding by guessing what will happen when you run `main.rml`.

Your task is to re-implement `timmy.ml` so that when `main.rml` is run, John and Mike have the following interaction:

```
"[John] Timmy, can you send this to Mike? - first message from john"
"[Mike] John says: first message from john"
"[Mike] Timmy, can you send this to John? - first message from mike"
"[John] Mike says: first message from mike"
"[John] Timmy, can you send this to Mike? - second message from john"
"[Mike] John says: second message from john"
"[Mike] Timmy, can you send this to John? - second message from mike"
"[John] Mike says: second message from mike"
```

Exercise 3: An Alternative Promises Library

The module `Mwt` in `promise.ml` implements a promises library based on the implementation in the 3110 textbook, by storing callbacks with each promise. An alternative way of implementing promises is by using a state machine, where the states of the promise are encoded by an OCaml type. This design can be used in languages with less support for closures.

We follow this approach in module `Fwt` in `promise.ml`. Your task is to complete this implementation. When you are done, you should be able to replace `Mwt` by `Fwt` everywhere in `eval.ml`, and everything should continue to work seamlessly.

Testing and Debugging

In order to make testing easier, we have provided you with an interactive shell that interprets RML expressions and definitions. Run `make repl` to start.

- The shell accepts input until it sees `;;` so you can enter multi-line programs.
- As in utop, you may input RML expressions or definitions.
- The command `#env` prints the current environment: the values and types of all variables defined so far. If you shadow a variable (define it twice), then it will be printed twice but only the most recent (top-most) binding is in effect.
- The command `#quit` exits the REPL.

We have also provided you with the `make run` directive, which compiles your code and then prompts you for a file `<filename>`. When you supply it with a directory path, it invokes the lexer, parser, type-checker, and interpreter on the text stored in the given file. Any `.rml` files you create for `make run` should be stored in `rml/test`. The files in this directory will be ignored by `make zip`.

Finally, we highly recommend developing a test suite for `eval.ml`, although you will not submit it as part of the final `.zip` file. We have provided you with some starter code and a few example test cases in `test/main.ml`.

Hints and Common Errors

Here are some tips for your implementations:

- Although the formal semantics mentions state (Delta and sigma), you are not responsible for keeping track of state. Your implementation should let OCaml update the state automatically using OCaml's references.
- RML does not allow n-place tuples. However, you should be able to encode them using nested pairs.
- Type annotations are required almost everywhere in RML. In particular, **all variables in let-bindings require type annotations**.
- The `self` value in RML must usually be stored in a variable with a type annotation in order for the type-checker to accept your program.

Here are some common bugs in RML to watch out for:

- RML's match statements require the `end` keyword after the last case. Don't forget it!
- While OCaml match statements allow you to omit the `|` in the first case, RML does not.
- In OCaml, if you put an expression on the left-hand side of a sequential composition `;` that does not have type `unit`, you get a warning. In RML, it is a type error. You can define

an `ignore` function and use it to explicitly discard the expression on the left-hand side of the composition.
- For `await p = e1 in e2`, remember that `e2` must be a promise.
- Unlike OCaml, your program cannot end with a semi-colon.

Rubric

- 25 points: submitted and compiles
- 50 points: satisfactory scope
- 20 points: good scope
- 5 points: excellent scope

The only functions that the autograder test suite will directly call are `Main.interp_file` and `Main.interp_expr`. So feel free to test other functions all you want, but to receive points you must ensure that `Main.interp_file` and `Main.interp_expr` are also working. The latter is the function that the provided (although minimal) test suite already tests. However, you will likely want to test it further.

We will not assess coding standards or testing on this assignment. We trust that you will use what you have learned in the rest of the semester to your advantage.

Each of the language features in Good scope will be worth about the same number of points, and the same is true for the features in Excellent scope. So please don't feel as though you have to implement all the features: it's perfectly fine to stop early. If all you do is the Satisfactory scope, you will still have learned a lot about interpreters.

Submission

Review the **A1 handout (https://canvas.cornell.edu/courses/48987/pages/a1-handout)** for details about submission. Here are a few reminders:

- Make sure to record your name and NetID in `author.mli`, complete the AI statement, and set the `hours_worked` variable at the end of `author.ml`.
- Run `make zip` to construct the ZIP file you need to submit on CMS, which on this assignment is named `rml.zip`. Remember: you must use `make zip` to create the file, not your OS's graphical file browser.
- Ensure that your solution passes `make finalcheck`.
- Submit on **CMS (https://cmsx.cs.cornell.edu/)**. If your submission is rejected for being too big, see **this page (https://canvas.cornell.edu/courses/48987/pages/big-repo-cleanup)**. Double-check that the MD5 sum is what you expected. Re-download your submission from CMS and double-check before the deadline that the contents of the ZIP are what you intended.

Congratulations! You've taken the first steps towards the robot revolution.

Acknowledgment: RML was inspired by an old CS 3110 assignment developed by Andrew Myers. Development of the current version was led by Timmy Zhu and Samwise Parkinson, with lots of help from the 2019sp, 2020sp, and 2022sp course staff.