A Mercury Tutorial

Zetian Lin (Sebastian Zack Tin Lahm-Lee)

May 30, 2025

Contents

1 Intro: what you're about to endure

I believe Mercury is within the list of the programming languages that people with at least a little bit of ambition would like to learn but give up midway because it's simply too unconventional and requires a more than average amount of knowledge to even get started. I can say this because I myself was in the same position eight years ago; and after writing this tutorial, I've found out that despite it being more than average, it is still a manageable amount, and a part of me thus started hating myself for not doing this sooner; one shall always keep in mind that two of the most important virtue

of a learner is resolution and resilience, especially when it comes to computer science.

Alan Perlis, a computer scientist, wrote a list of quotes under the name "Epigrams on Programming" in 1982. The No. 19 of such quote says: "A language that doesn't affect the way you think about programming is not worth knowing". Ever since the birth of this quote, it itself and the idea behind it have gathered a cult-like following of a certain size, which I was once a part of alongside with my programming language theory hobbyist peers from China eight years ago. If anyone were to ask me if Mercury fits this description, I would recommend him to ignore the opinions of anyone who says no.

If you still don't know what Mercury is (which I suppose is unlikely since you've consciously chosen to start reading this) - Mercury is a logic-functional programming language... or functional-logic programming language? which means that it will have features from both of those worlds. Among the logic-functional hybrid, Mercury is visibly more on the "logic" side, closer to Prolog than to Haskell (unlike Curry, which is the opposite). From a cynical point of view, I don't know why anyone would choose to learn this unless they're a very specific kind of masochist (just like me), but I do wish whatever experience you'll have while learning it could become an inspiration of making something that would not be boring in the future.

2 Prerequisite: logic programming

> You can skip this part if you have learned Prolog before - in fact, I might even go as far as to tell you to learn Prolog instead of reading this chapter, which is my poor attempt at stuffing as much essential knowledge within as short a span as possible. You don't have to learn a lot - basic Prolog, like the amount taught by Learn Prolog Now!, is more than sufficient.

While it might not seem obvious from an outsider's view, logic programming has its roots in first-order predicate logic; more specificly, it comes from a specific interpretation of a specific class of first-order predicat logic formula. This class of formula, called **Horn Clause** after the logician Alfred Horn, can be one of the following forms:

- "Rule", which takes the form of $Q \leftarrow P_0 \wedge P_1 \wedge \cdots \wedge P_n$.
- "Fact", which takes the form of $P \leftarrow \top$.

- "Goal", which takes the form of $\bot \leftarrow P_0 \wedge P_1 \wedge \cdots \wedge P_n$.

within which Q and all P_n are first-order predicate logic terms. If how this has anything to do with writing programs isn't clear to you, it probably will after reading the definition of the factorial function, written in (somewhat) Horn clauses:

- $fac(0, 1) \leftarrow \top$
- $fac(1, 1) \leftarrow \top$
- $fac(N, M) \leftarrow minus(N, 1, N_0), fac(N_0, M_0), mult(M_0, N, M)$

Compare this with the following:

- $fac(0) = 1$
- $fac(1) = 1$
- $fac(N) = M$ `where` $\{N_0 = N - 1, M_0 = fac(N_0), M = N \times M_0\}$

The similarity should be obvious at this point; in a very inaccurate sense, you can think of it as:

- Rules as function definition;
- Predicates as function calls;
- Conjunction as sequencing;
- Facts as base cases;
- Goals as entry points;

This, of course, is not all there is to it. We've covered the representation, we shall now cover the reduction.

2.1 Unification

One thing to keep in mind is that theoretically there is no "computation" (as in calculation) involved; what we have (mostly) is unification. To some people this (and resolution, which we will discuss later) looks like a rewriting system, and some people do claim that Prolog is just a rewriting system with backtracking. As someone who've worked with rewriting systems before, I'm not going to comment on their opinions, but they do be looking similar.

Anyway, unification is a thing that happens between terms. All terms are of one of the following forms:

- Variables, which we will denote with words that starts with an uppercase letter;
- Atoms, which we will denote with words that starts with a lowercase letter;
  - (This name, despite being the commonly used name for languages like LISP or Erlang, might come as obscure for other people; a more descriptive name would probably be "scalar values".)
- Functors, which is of the form $f(a_0, \cdots, a_n)$, in which f is an atom and all the a are terms (which can themselves be variables or atoms or functors, etc.)

The unification of two terms follows the following rules:

- A variable matches with anything (unless it forms recursive reference; see below);
- Atoms match with themselves.
- Functors match if and only if all the following conditions are met:
  - The "head" (or "the functor's name") matches, e.g. $f(A, B)$ matches with $f(a, b)$ but not $g(a, b)$ because the latter is g instead of f.
  - The arity (i.e. the number of arguments) matches, e.g. $f(A, B)$ matches $f(a, b)$ but not $f(c, d, e)$ despite both being f; the former has 2 arguments while the latter has 3. The convention is to denote the arity after the functor's name, e.g. $f/2$ and $f/3$.
  - All of the arguments match with each other, e.g. $f(a, B)$ matches with $f(a, b)$ and $f(a, c)$ but not with $f(b, a)$ despite both being $f/2$; the first arguments, namely a and b, does not match.

The action of unification results in a mapping from variables to terms; (one could say that this is how "value assigning" happens, despite that really isn't the accurate wording.) A common name for this mapping would be "subst", short for substitution. The observant might want to ask if one can match a variable with a term that contains the same variable either directly or indirectly (e.g. A with $f(A)$). Theoretically this is not valid because one can never have a fully grounded solution for such a subst, but whether an occurence of such case will trigger some kind of runtime panic depends on

the language and its implementation. Some languages (like Prolog) permits it (even if this kind of situations almost certainly end in a failure state), some languages (like miniKanren) performs strict occur check and does not allow any such occurences.

Some of you may ask if pattern matching can also be seen as a form of unification. I've met people online who were very pedantic about the name and fervently insisted that pattern matching is not unification; almost all of those kind of people are doing this because they want to be perceived as more sophisticated than they truly are so they can look down upon other people more safely. I would say one can think of it as somewhat similar; one must understand that pattern matching (and the subsequent binding of variables) is a "one way thing", e.g. unifying $f(a, J)$ with $f(K, b)$ would result in a subst of $J = b, K = a$, but with pattern matching one side is always a fully grounded term, if we can even consider values as terms in this sense.

2.2 Resolution

In this section I would only introduce one way of resolution. It is - I believe - conceptually the simplest kind of resolution. There probably are other kind of resolution technique out there, but they are out of scope for this tutorial.

Remember that a goal is of the form $FALSE \leftarrow P_0 \wedge \cdots \wedge P_n$ (we'll omit the $FALSE$ part for brevity from now on) and act as entry points in our interpretation; naturally, if there's no more P, the goal is empty, and the computation halts.

The simplest method of resolution is thus as follows. For the goal $\leftarrow P_0 \wedge \cdots \wedge P_n$, we take the left-most subterm, in this case P_0; we attempt to find among all the previously defined rules and facts the one that can successfully unify with P_0 (instantiated according to the current subst). If we couldn't find one, then the current attempt is considered a failure (what happends after the failure depends; see later section on cut); If we were successful in finding one, then two things happen:

- Because we performed a unification, we now have a subst;
- And because we picked a rule/fact, we now have a "right-hand side".

This subst resulted from the unification is combined with all the subst we've had so far, and the "right-hand side" of the rule/fact we found would be appended to the left-hand side of the current goal.

Let's look at an example. Assume we have the following definitions:

- $parent(tom, john) \leftarrow \top$
- $parent(tom, evans) \leftarrow \top$
- $parent(tom, sarah) \leftarrow \top$
- $sibling(A, B) \leftarrow parent(C, A) \wedge parent(C, B) \wedge not(equal(A, B))$

and the goal we're asking is $\leftarrow sibling(john, A)$; the following things happen:

- $sibling(john, A)$ matches with $sibling(A, B)$, (but we cannot directly say that $john = A, A = B$, because the two A should clearly mean different things! with a bit of renaming, we shall say the goal is $sibling(john, A_0)$).
- The unification of the step above results in the subst $john = A, A_0 = B$.
- It contains a right-hand side of positive length; we shall add them to the goal with respect of the subst. The goal is now $\leftarrow parent(C, john) \wedge parent(C, A_0) \wedge not(equal(john, A_0))$.
- We now try to find a rule/fact that can unify $parent(C, john)$, and we found $parent(tom, john)$. This gives us $C = tom$. Since it's a fact, there is no right-hand side to be added. The goal is now $\leftarrow parent(tom, A_0) \wedge not(equal(john, A_0))$.
- We now try to find a rule/fact that can unify $parent(tom, A_0)$, and we again found $parent(tom, john)$. This gives us $A_0 = john$.
- The goal, which is now $\leftarrow not(equal(john, john))$ with respect to the current subst, fails. If we have committed to this choice (by cut or by other means), the entire thing fails; but we didn't, thus we backtrack. The goal is now once again $\leftarrow parent(tom, A_0) \wedge not(equal(john, A_0))$, and we attempt to find the next thing that can unify with $parent(tom, A_0)$.
- We found $parent(tom, evans)$, which results in the subst $A_0 = evans$.
- The goal is now $\leftarrow not(equal(john, evans))$, which would succeed without a subst (or with an empty subst, depending on how you see it), leaving the goal empty. Since the goal is now empty, we consider whatever we have now is a valid solution. The subst for this solution would be $A = john, C = tom, A_0 = evans$, and the solution itself would be $sibling(john, evans)$.

- If more result is requested, the latest unification and its subst would be discarded and a new instance of $parent(tom, A_0)$ would be looked for, which would result in the subst $A_0 = sarah$ and the solution $sibling(john, sarah)$.
- It's easy to see that we can no longer find any new solutions after this point, so the resolution for the goal ends here.

2.3 Cut

Cut is mainly a Prolog thing; when triggered, it forces the resolution to commit to all the choices it has made up till that point, artificially cutting the backtracking tree. This is one of the major reasons why Prolog is being considered as "not so pure". Similar constructs exists in other logic programming languages (e.g. Mercury's `cc_multi` and `cc_nondet`), albeit they may work in a fundamentally different manner.

Using our program above as an example; if we write the first fact as $parent(tom, john) \leftarrow !$ instead of $parent(tom, john) \leftarrow \top$ (the exclamation mark is the "cut operator" in Prolog), we wouldn't have any solutions, since by the time we reached the goal $\leftarrow not(equal(john, john))$, we have already committed to the choices of $john$ and discarded other possibilities.

2.4 Disjunction

I thought I'd quickly mention this here - you can use disjunction in Horn clauses. The resolution of a disjunct goal naturally depends on the resolution of its subgoals, and any solution is a solution of the overal disjunct goal if it is also a solution of any of its subgoals. The resolution of a disjunct goal starts from one of its subgoals. (The order theoretically doesn't matter, but you have cut in Prolog which would affect the backtracking tree, which means the order does kinda matter in case of cut.) If that subgoal fails, it moves onto the next subgoal, until any one of these subgoals succeed (which results in an overall success) or all of them failed (which results in an overall failure).

2.5 Unification, part 2

In short, you can do (certain) thing "in reverse". For example, assume that we have a predicate `append/3` which, at surface, is something you append two lists together:

- $append(nil, nil, nil) \leftarrow \top$
- $append(nil, A, A) \leftarrow \top$
- $append(cons(H, T), A, cons(H, TR)) \leftarrow append(T, A, TR)$

If you query goals where the first two arguments are "solid" terms, you will get the result bound to the variable at the third argument; but at the same time, you can:

- Call the predicate with three "solid" lists, with which you can check if appending the first two lists does end up with the same value as the third one. If that *is* the case, then you'll eventually hit one of the first two facts and have an overall success; if that *isn't* the case, you'll have a failure.
- Call the predicate with two variables and one "solid" list, with which you have a goal that asks "which two lists can be appended and end up with the same value as this third list".

For the second one, we'll discuss the case $append(A, B, cons(c, nil))$ here; you can try to resolve a few examples on your own as an exercise. In the case of $append(A, B, cons(c, nil))$, the resolution would first match with the second fact (which will give $A = nil, B = cons(c, nil)$) and then it would match with the third rule (which, if you work it out, will give $A = cons(c, nil), B = nil$.)

3 First example: `rall`

In this part of the tutorial, we will start learning Mercury by trying to write a program which I called `rall`.

`rall`, in its most basic form, is a very small utility that converts Unix-style newlines (LF) to Windows-style newlines (CRLF). It covers some of the most basic things when it comes to software development, and I reckon it would serve as a nice beginner (in terms of language, not programming in general) project.

3.1 File name / module name

Each Mercury source file would be a module, whose name should be the same as the file name (without the extension name part). Each module would need to have at least one `interface` section and one `implementation` section.

We would save the file under the name `rall.m`, thus the module name would be `rall`.

```
:- module rall.

:- interface.

:- implementation.
```

3.2 Entry point

With Mercury, the entry point (e.g. `main` in C) must be defined as a `main/2`, i.e. `main` with 2 parameters. If you attempt to define `main` as anything else, you would get a compile error:

```
% mmc ./rall.m
/usr/bin/ld: rall_init.o: in function ‘mercury_init’:
rall_init.c:(.text+0x4a0): undefined reference to ‘<predicate ’main’/2 mode 0>’
collect2: error: ld returned 1 exit status
```

The two parameters of this `main/2` must be of type `io.state`. If you don't define it as `io.state`, you would get a compile error:

```
./rall.m:005: Error: ‘main’/2 must have mode ‘(di, uo)’.
./rall.m:005: Error: both arguments of ‘main/2’ must have type ‘io.state’.
```

So we now have the first few lines of Mercury:

```
:- interface.
% NOTE: to use io.state we must import the io module.
:- import_module io.
:- pred main(io.state, io.state).
```

You might wonder what exactly is `io.state` and why `main` needs two of them. We will explain this later. At the very least, these are **not** the `argc` `argv` thing you would see in C. There are dedicated predicates for retrieving command line arguments; we will get to that later.

The `(di, uo)` part needs explanation. The message says it's a **mode**. What is a mode? Modes are things that describe the change of instantiatedness before and after the execution of a goal. Instantiatedness - that's a

long word. What is instantiatedness? Roughly speaking, it's a concept for describing whether a slot is free or bound. (In Mercury, instantiatedness is actually a tree, because obviously you'd have - or at the very least you could manually construct - terms that are only bounded at certain parts.) Two kinds of basic instantiatedness exists in Mercury: `free`, which refers to variables that are not bound by any values; `bound`, which means that the term is not a variable but rather some concrete term at least at that level. Consider the term `blah(A, b)`; it's `bound` at the top (with `blah`) and at one of the children node (at `b`) but `free` at another children node (at `A`, provided that `A` does not have a value already). A mode, `ground`, also exists, and refers to the terms that are *completely* bound (e.g. `blah(a, b)` is `ground` but `blah(A, b)` isn't, provided that `A` is still `free`.)

With this knowledge, we should first understand what `in` and `out` modes actually are. If you have worked with some other languages you might have seen thing similar to this before: C libraries often ask you to pass in a reference for retrieving the actual result because the return value is used for returning error status; you can think of it as "in" (inputting) parameters and "out" (returning) parameters; some languages (e.g. Ada) even explicitly label them as such. I do not wish to introduce you to wrong analogies that will become detrimental for your future learning, but I have to say they do be somewhat similar.

That said, `in` and `out` is properly defined in Mercury, as follows:

```
:- mode in  == ground >> ground.
:- mode out == free >> ground.
```

This largely fits with our intuition about in and out parameters: we expect `in` arguments to be fully grounded and they stay grounded, and we expect `out` arguments are variables which we will provide a solid term to (thus making it grounded).

There are also "polymorphic" version of `in` and `out`, defined as follows:

```
:- mode in(Inst)  == Inst >> Inst.
:- mode out(Inst) == free >> Inst.
```

You can indeed define your own instantiatedness, with which then you can use these definitions to get the `in` and `out` version of it; but we will not need this for our example this time.

Now we can finally come back to `di` and `uo`. `di` stands for "destructive input", and `uo` stands for "unique output". If you have used Rust (or, in

the case where you are really adventurous, Clean) before, you might have a vague idea of what this is. In Mercury, there are two special instantiatedness named `unique` and `dead`, the former conceptually refers to values that can only have one reference at all time, and the latter refers to reference that are once "unique" but is now "dead". (Mercury also has `muo`, `mdi` and `mui` which stands for "mostly unique output", "mostly destructive input" and "mostly unique input". They are meant to support backtracking and when the program backtracks

```
% unique output - used to create a "unique" value
:- mode uo == free >> unique.

% unique input - used to inspect a unique value without causing
% reference to become dead
:- mode ui == unique >> unique.

% destructive input - used to deallocate or reuse the memory
% occupied by a value that will not be used.
:- mode di == unique >> dead.
```

Up to now, our code would be something that's similar to this:

```
:- module mercury_rall.

:- interface.

:- import_module io.
:- pred main(io.state, io.state).

:- implementation.

main(_, _) :-
    % some dummy body for our main predicate.
    1 = 1.
```

If you compile this, the Mercury compiler would complain about not having a mode declaration. For this reason, we will add the following line and compile:

```
:- mode main(di, uo).
```

But this time we would see the compiler complain about different things:

```
./rall.m:007: Error: no determinism declaration for exported predicate
./rall.m:007:   ‘main’/2.
./rall.m:012: In clause for ‘main(di, uo)’:
./rall.m:012:   mode error: argument 2 did not get sufficiently
./rall.m:012:   instantiated.
./rall.m:012:   Final instantiatedness of ‘HeadVar__2’ was ‘free’,
./rall.m:012:   expected final instantiatedness was ‘unique’.
```

The compiler expects the second argument would be supplied with a `free` argument and that argument should become a `unique` value at the end of `main`! How do we do such a thing? Fortunately the `io` module has what we need to stuff the body of `main`, and we will use one of them:

```
main(In, Out) :-
    io.write_string("blah", In, Out).
```

But this time the compiler started complaining about other things:

```
./rall.m:007: Error: no determinism declaration for exported predicate
./rall.m:007:   ‘main’/2.
```

Determinism in this case, informally speaking, refers to "how a certain thing would succeed/fail". It's not something we'd care about in other languages, at least not in an explicit, supported-by-the-language-itself manner; (we normally only talk about when programs terminate at a certain state or not.) In Mercury, we have the following determinism categories:

- Deterministic (`det`): guaranteed to have one and exactly one solution.
- Semideterministic (`semidet`): have exactly one solution, but does not guarantee to produce it.
- Multisolution (`multi`): guaranteed to have a solution among possibly many solutions.
- Nondeterministic (`nondet`): have possibly many solutions, does not guarantee to produce one.
- Failure (`failure`): cases where there's zero solutions. They are not actual errors but a part of the logic flow (e.g. arity mismatch during unification, which will never produce a solution because the arity is different).

- Errorneous (`errorneous`): also have zero solutions, but they **do** represent actual errors which in other languages would be represented in the form of runtime exception throw or panic.

(Some people might not understand why `det` and `semidet` are separate things. Imagine a function that takes the "head" of a linked list; this function is obviously only defined on non-empty list and not defined on empty lists, so for any given list there's either only one solution or no solution, thus `semidet` instead of `det`.)

(NOTE the difference between `errorneous` and `failure` might be clearer if I explain their behaviours when it comes to negation. Basically in Mercury (and other logic programming languages), you can take a predicate, put a negation on it, and then ask for a case where it **does not succeed**. Naturally, the negation of determinisms that are guaranteed to produce a result like `det` and `multi` would be `failure` (i.e. negating a definite success would be a failure), the negation of `failure` would be `det` (i.e. negating a definite failure would be a definite success), and the negation of `semidet` and `nondet` would be `semidet` (i.e. negating these cases turns it into a case that asks if the match would be successful or not); but negating an `errorneous` would only produce an `errorneous`.)

(If you're going to ask this question - yes, solving the problem of perfect determinism inference does mean solving the halting problem, and is thus undecidable.)

Other than these six categories there are also this thing called the "committed choice nondeterminism", which adds two more determinism class: `cc_multi` and `cc_nondet`. The difference between committed choice nondeterminism and "normal" nondeterminism is that normal nondeterminism supports backtracking while CC nondeterminism, despite potentially having more than one solutions, commits to only one of them and thus does not backtrack. The `main` entry point can be defined to be `det` or `cc_multi`, since both of them guarantee one and only one solution (`det`, of course, is more strict than `cc_multi`, and if the Mercury compiler can determine something that should be able to be a `det` got labelled as a `cc_multi`, it would spit out a warning saying you could've gone with the stricter option.)

In this case, we should add the string `is det` at the end of our mode declaration, so that the whole line would be `:- mode main(di, uo) is det.`. After this modification, the compiler should have finally stopped complaining and gives you an executable; when you run it, it would display a string `blah`, which should be obvious. At this point, the code shall look like something like this:

```
:- module rall.

:- interface.

:- import_module io.
:- pred main(io.state, io.state).
:- mode main(di, uo) is det.

% also: you can combine the ‘pred‘ and the ‘mode‘ line into one like this:
%
%     :- pred main(io.state::di, io.state::uo) is det.

:- implementation.

main(In, Out) :-
    io.write_string("blah", In, Out).
```

Now, if you attempt to write multiple strings like this:

```
main(In, Out) :-
    io.write_string("blah", In, Out),
    io.write_string("blah", In, Out).
```

The compiler would produce this error:

```
./rall.m:013: In clause for ‘main(di, uo)’:
./rall.m:013:   in argument 2 of call to predicate ‘io.write_string’/3:
./rall.m:013:   unique-mode error: the called procedure would clobber its
./rall.m:013:   argument, but variable ‘In’ is still live.
For more information, recompile with ‘-E’.
```

If we look up the declaration of `io.write_string`:

```
:- pred write_string(string::in, io::di, io::uo) is det.
```

You should already know that `di` specifies a turn of a `unique` value into a `dead` value, so `In` after the first `write_string` would be considered `dead` and not `unique`, which does not fit the requirement of the `di` of the second `write_string`. If you try to experiment and do this:

```
main(In, Out) :-
    io.write_string("blah", In, Out),
    io.write_string("blah", Out, Out2).
```

This would have the following violation: since we declared `Out` to be `uo` which is `free >> unique`, `Out` must be `unique` at the end of `main`, but the second `write_string` turns `Out` into a `dead` because that slot is `di` which is `unique >> dead`. A solution to this would be to do this:

```
main(In, Out) :-
    io.write_string("blah", In, Out1),
    io.write_string("blah", Out1, Out).
```

This would compile, and the generated executable would write "blah" twice, as expected.

To keep coming up with new variable name is tedious if this gets long. To solve this problem, Mercury have something called the "state variable". With state variables, the example above can be written like this:

```
main(!.IO, !:IO) :-
    io.write_string("blah", !.IO, !:IO),
    io.write_string("blah", !.IO, !:IO).
```

`!.IO` refers to the value bound to `IO` at the current moment, and `!:IO` makes the value at that slot bound to `IO`; so in the first `write_string`, the *current* value `IO` was destroyed, and the value bound to `Out1` in our non-state-variable version was bound to `IO` as its next value; and in the second `write_string`, the value of `Out1`, which itself has become the current value of `IO` and being referred to with `!.IO`, was destroyed, and the value bound to `Out` in our non-state-variable version was bound to `IO` as its next value, and subsequently became the output of `main`.

You can write `!.IO, !:IO` as `!IO`, because the former is also quite tedious as well:

```
main(!IO) :-
    io.write_string("blah", !IO),
    io.write_string("blah", !IO).
```

It would still feel a bit tedious for people who are used to other programming languages, but I suppose this is what you give up for being as explicit as possible for the sake of improved correctness and things...

3.3 Command line arguments

The predicate we need for retrieving the command line arguments is also in the `io` module:

```
main(!IO) :-
    io.command_line_arguments(Argv, !IO),
    % ...
```

This would bound the actual value to `Argv`. From the definition of `command_line_arguments` we can know that the type of Argv would be a `list(string)` - a list of strings. It would be necessary to import the `list` module from now on. This value does not contain the name of the program (for which `io` module has other predicates to retrieve). It would also be useful to check if the length of `Argv` is exactly 2:

```
:- import_module list.
% ...
main(!IO) :-
    io.command_line_arguments(Argv, !IO),
    ( if length(Argv) \= 2
      then (
          % ...
      )
      else (
          % ...
      )
    ).
```

(Note that if you have previous experience with Prolog, the `if length(Argv) \= 2` part might be surprising; normally this wouldn't work, and you would need to do something like `length(Argv, NArgv), N \= 2` or maybe even use an `is` in between since `=` and `\=` is technically about unification instead of evaluation (e.g. `3 + 4 = 7` would fail in prolog because `3 + 4` is not "structurally equivalent" to `7`); the fact is, `length` in Mercury is defined both as a predicate and as a **function**, and in this case the function got evaluated as it's defined in the standard library resulting an actual number. Also, while we mostly expect functions to have either one or no result i.e. `det` or `semidet`, in Mercury we can define functions of determinism like `multi` and `nondet`, although I struggle to think of an example where this would work or be useful.)

The `then` clause is simple - we write the usage string and call it done:

```
:- import_module list.
% ...
```

```
main(!IO) :-
    io.command_line_arguments(Argv, !IO),
    ( if length(Argv) \= 2
      then (
          % you can specify which stream you are writing to btw
          io.write_string(
              io.stderr_stream,
              "usage: rall [inputfilename] [outputfilename]\n",
              !IO)
      )
      else (
          % ...
      )
    ).
```

Now we extract the command line arguments. `list` module contains `index0` and `index1` - yes, you can choose either to use 0-based index or 1-based index, but these aren't the ones we use, because they are `semidet` (exercise: can you figure out the reason why they're `semidet` instead of `det` on your own?); we use their `det` counterparts: `det_index0` and `det_index1`, which will throw runtime exceptions when failing:

```
:- import_module list.
% ...
main(!IO) :-
    io.command_line_arguments(Argv, !IO),
    ( if length(Argv) \= 2
      then (
          io.write_string(
              io.stderr_stream,
              "usage: rall [inputfilename] [outputfilename]\n",
              !IO)
      )
      else (
          InputFilePath = det_index0(Argv, 0),
          OutputFilePath = det_index0(Argv, 1),
          rall(InputFilePath, OutputFilePath, !IO)
          % ...
      )
    ).
```

It would be also be a good time to write our first predicate that is not the entry point...

3.4 Opening & closing files for input/output

In the last code snippet we've write `rall(InputFilePath, OutputFilePath, !IO)`. Despite this looks like `rall/3`, but it's secretly `rall/4`, and we should write our declaration as such:

```
:- pred rall(string, string, io.state, io.state).
:- mode rall(in, in, di, uo) is det.
```

`io` module has `open_input/4` and `open_output/4`, both of which are `det`:

```
rall(InputFilePath, OutputFilePath, !IO) :-
    io.open_input(InputFilePath, RIn, !IO),
    % ...
```

The declaration of `open_input` is as follows:

```
:- pred open_input(
       string::in, io.res(io.text_input_stream)::out,
       io::di, io::uo) is det.
```

We know that whatever variable we put at the second slot would be the input stream we want (or in the case of error, whatever error we could catch). Its value is of type `io.res`, whose definition is as follows:

```
:- type res(T)
   --->    ok(T)
   ;       error(io.error).
```

(Yes, this is how Mercury defines algebraic datatypes. If you have previous experience with Haskell, OCaml, Rust or similar languages, this should be natural for you.)

Pattern matching is quite simple - you just do `Var = ok(OkValue)` and `Var = error(ErrorValue)` like this:

```
% ...
( Var = ok(OkValue),
  % `ok` clause
```

```
);
( Var = error(ErrorValue),
  % ‘error‘ clause
)
```

Since we combined them with a disjunction, when the unification in the first branch (in this case, `Var = ok(OkValue)`) fails, the program flow would go to the second branch, and if the unification there fails (and there's a third branch), it would go to the third clause, etc.; and if all branches fail, the whole thing fails (and of course we'd expect the Mercury compiler gives us a determinism verdict of something like `nondet`, but I digress). We'll handle all the cases a value of type `res(T)` could have, and if we do that, the compiler should recognize that we'll not fail as long as both of our `ok` clause and `error` clause don't have a determinism verdict that implies a possibility of failing.

The definition of `io.error` is as follows:

```
:- type io.error.   % Use error_message to decode it.
```

This type is opaque; we need `error_message` to handle it. Its definition is as follows:

```
    % Look up the error message corresponding to a particular error code.
    %
:- func error_message(io.error) = string.
:- pred error_message(io.error::in, string::out) is det.
```

Now we know we can obtain a `string` from an `io.error`; we can decide either to `throw` (which would cause the program to exit prematurely) or to write it to standard error (and try to exit afterwards). In this example we use `throw` because it's simpler:

```
% throw/1 requires the exception module.
:- import_module exception.

% ...
( RIn = error(ErrorValue), throw(error_message(ErrorValue)) );
```

Before we move on with it, I must also explain something else. This isn't the case in Prolog, but in Mercury calls are "curried". Using the declaration of `error_message` above as an example, the term `error_message(Var)`

matches both `func error_message/1` *and* `pred error_message/2`, and would thus be judged as having the type `string` from `func error_message/1` and the type `pred(string)` from `pred error_message/2` (which is a pred that takes a `string` as its argument) *at the same time*, i.e. a type ambiguity. In Prolog it will only match `error_message/2`, but in Mercury this is how it is...

To resolve this you need to explicitly state the type you want by writing `:{type}`; in this case, we'll choose the `func` one and specify it as `string`, just like this:

```
%                                 here, the ":string" part. ---v
( RIn = error(ErrorValue), throw(error_message(ErrorValue):string) );
```

If you don't do this, you'll get a compile error like this:

```
./rall.m:025: In clause for predicate main'/2:
./rall.m:025:   warning: variable OutputFilePath' occurs only once in
./rall.m:025:   this scope.
./rall.m:031: In clause for predicate rall'/3:
./rall.m:031:   error: ambiguous overloading causes type ambiguity.
./rall.m:031:   Possible type assignments include:
./rall.m:031:   V_20:
./rall.m:031:     pred(
./rall.m:031:       string
./rall.m:031:     )'
./rall.m:031:   or
./rall.m:031:     string'
./rall.m:031:   You will need to add an explicit type qualification to
./rall.m:031:   resolve the type ambiguity. The way to add an explicit
./rall.m:031:   type qualification is to use "with_type". For details see
./rall.m:031:   the "Explicit type qualification" sub-section of the
./rall.m:031:   "Data-terms" section of the "Syntax" chapter of the
./rall.m:031:   Mercury language reference manual.
```

Now we can have the following code. Opening an output stream is a near-identical process:

```
% throw/1 requires module `exception`.
:- import_module exception.
% ...
```

```
:- pred rall(string, string, io.state, io.state).
:- mode rall(in, in, di, uo) is det.
rall(InputFilePath, OutputFilePath, !IO) :-
    open_input(InputFilePath, RIn, !IO),
    (   ( RIn = error(ErrorValue), throw(error_message(ErrorValue):string) );
        ( RIn = ok(InputStream),
          open_output(OutputFilePath, ROut, !IO),
          (   ( ROut = error(ErrorValue), throw(error_message(ErrorValue):string) );
              ( ROut = ok(OutputStream),
                process_stream(InputStream, OutputStream, !IO)
              )
          ),
          close_output(OutputStream, !IO),
          close_input(InputStream, !IO)
        )
    ).
```

3.5 Actually processing the data

We'll do it as follows: each time we read a single character from the input stream, we check if it's a line feed; if it is, we write a carriage return *and* a line feed to the output stream; or else, we simply write that character to the output stream; we repeat this until we've hit the EOF of the input stream. `read_char` produces a different kind of result named `result`, which has a separate constructor `eof`. You should know how to handle this by now...

Character literals are enclosed with single quotes, by the way:

```
:- pred process_stream(
      io.text_input_stream,
      io.text_output_stream,
      io.state, io.state).
:- mode process_stream(in, in, di, uo) is det.
process_stream(InStream, OutStream, !IO) :-
    read_char(InStream, InRes, !IO),
    (  ( InRes = eof );
       ( InRes = error(Error), throw(error_message(Error):string) );
       ( InRes = ok(Char),
         ( if Char = '\n'
           then (
               write_char(OutStream, '\r', !IO),
```

```
            write_char(OutStream, '\n', !IO)
          )
          else (
            write_char(OutStream, Char, !IO)
          )
        ),
        process_stream(InStream, OutStream, !IO)
      )
    ).
```

3.6 Full program listing

The following code is the listing of the completed `rall` program:

```
:- module rall.

:- interface.
:- import_module io.

:- pred main(io.state, io.state).
:- mode main(di, uo) is det.

:- implementation.

:- import_module list, exception.

main(!IO) :-
    io.command_line_arguments(Argv, !IO),
    ( if length(Argv) \= 2
      then (
          io.write_string(
              io.stderr_stream,
              "usage: rall [inputfilename] [outputfilename]\n",
              !IO)
      )
      else (
          InputFilePath = det_index0(Argv, 0),
          OutputFilePath = det_index0(Argv, 1),
          rall(InputFilePath, OutputFilePath, !IO)
      )
```

```
    ).

:- pred rall(string, string, io.state, io.state).
:- mode rall(in, in, di, uo) is det.
rall(InputFilePath, OutputFilePath, !IO) :-
    open_input(InputFilePath, RIn, !IO),
    (   ( RIn = error(ErrorValue), throw(error_message(ErrorValue):string) );
        ( RIn = ok(InputStream),
          open_output(OutputFilePath, ROut, !IO),
          (   ( ROut = error(ErrorValue), throw(error_message(ErrorValue):string) );
              ( ROut = ok(OutputStream),
                process_stream(InputStream, OutputStream, !IO)
              )
          ),
          close_output(OutputStream, !IO),
          close_input(InputStream, !IO)
        )
    ).

:- pred process_stream(
      io.text_input_stream,
      io.text_output_stream,
      io.state, io.state).
:- mode process_stream(in, in, di, uo) is det.
process_stream(InStream, OutStream, !IO) :-
    read_char(InStream, InRes, !IO),
    (  ( InRes = eof );
       ( InRes = error(Error), throw(error_message(Error):string) );
       ( InRes = ok(Char),
         ( if Char = '\n'
           then (
               write_char(OutStream, '\r', !IO),
               write_char(OutStream, '\n', !IO)
           )
           else (
               write_char(OutStream, Char, !IO)
           )
         ),
         process_stream(InStream, OutStream, !IO)
       )
```

```
).
```

It might not be idiomatic Mercury code, but at least it compiles and runs.