A Gentle Introduction To Mercury

The Functional-Logic Programming Language

Zetian Lin

A Gentle Introduction to Mercury

The Functional-Logic Programming Language

Zetian Lin

(Very early draft; October 2025)

Contents

Contents

CHAPTER 1
Introduction

1.1 What is Mercury?

Mercury[1] is a *functional logic* programming language, which is a paradigm that combines features common in certain functional languages (e.g. higher-order constructs, strong advanced typing) and logic programming semantics (e.g. non-determinism, built-in search). It attempts to combine the clarity and expressiveness of declarative programming with advanced static analysis and error detection features. According to its creators, it's a "general purpose language intended to support the creation of large, reliable and efficient applications." Although a short-lived renewed interest in this paradigm has been generated by Epic Games's programming language Verse[2] a few years earlier, it has now seemingly died down unfortunately; the paradigm remains interesting regardless, for the obvious reason that it provides extra edge against both purely functional and purely logic languages.

As are most of the languages that are of similar categories, Mercury has been mostly used for academic/research purposes, but a few commercial products have chosen (or once chosen) Mercury as their main language as well, like PrinceXML[3], which is an HTML rendering tool used in publishing.

[1]`https://mercurylang.org`

[2]`https://dev.epicgames.com/documentation/en-us/fortnite/verse-language-reference`

[3]`https://www.princexml.com`

Other similar languages

A few languages have been said to be similar to Mercury on its different aspects:

- Curry[4]: When discussing functional logic programming, people naturally lean towards a conceptual Haskell-Prolog continuum; if one were to say Mercury is closer to the Prolog side, Curry is clearly closer to the Haskell side of things.
- Clean[5]: Clean is the lesser-known sibling of Haskell famed for its uniqueness typing. It was said that the ability to explicitly specify uniqueness is not unlike that of Mercury.

1.2 The state of things as of now

Mercury has been considered a difficult language to learn, even for people who has previous functional language experiences. While Mercury do have unfamiliar aspects from the viewpoint of programmers who have only written in "conventional" languages like Scala or Haskell, I personally find those aspects aren't as difficult as one may think; I believe, that to a certain point this "difficult" feel comes from other factors, like the lack of a visible software ecosystem:

- Lack of example, tutorials and other introductory text: To be honest, it's not that we don't *have* those, it's simply that most of them tend to be slightly too confusing for common programmers who are used to conventional imperative/functional languages. As of the time of writing (October 2025), we have the following attempts:
  - The tutorial written by Ralph Becket which is listed on Mercury's official website[6]. Unfortunately this tutorial was left unfinished years ago and its author was said to have since stopped doing Mercury-related things.
  - The tutorials on Mercury's GitHub repository wiki[7], which seemingly has ceased major activities since 2018.

[4] `https://curry-lang.org`
[5] `https://clean-lang.org`
[6] `https://mercurylang.org/documentation/learning.html`
[7] `https://github.com/Mercury-Language/mercury/wiki/Tutorial`

  - "Learn Mercury By Y Minute"[8], which didn't do a very good job at explaining things to a depth a learner would need their materials to have.
  - "Mercury Crash Course" written by J Fondren, on the now-defunct mercury-in.space website[9], which honestly was too confusing for newcomers to provide any help.
  - A book titled "Declarative Programming in Mercury" is also available by compiling from the TeX source yourself using the source code[10], but this one really isn't an introductory text...

- Lack of package manager: Same as above, it's not like we *don't* have one - in fact we have two, one called Merchant by Stewart Slocum[11], and one called `mmc-get`[12] by J Fondren, which used to have a handful of packages[13]. They are not compatible with each other (of course), which I doubt have ever created any kind of problem since nearly no one used them.

- Lack of libraries: This is the real pain point. If a language did not have enough libraries, people are much less likely to use the language to make things; and if not enough people are using the language, there wouldn't be enough demand and thus there wouldn't be enough libraries. It's a viscious cycle that requires a *lot* of effort to break.[14]

But all of these situations won't change if we simply do nothing and hope for that one dumbass to put their love and effort for little to none rewards, which is why I have decided to become that dumbass myself and write this book to bridge the gap somewhat.

[8]`https://learnxinyminutes.com/mercury/`

[9]You can still read it on Wayback Machine: `https://web.archive.org/web/20220815044519/https://mercury-in.space/crash.html`

[10]Available on GitHub: `https://github.com/Mercury-Language/books/tree/master/decl_prog`

[11]`https://github.com/stewy33/merchant`

[12]`https://github.com/jrfondren/mmc-get`

[13]`https://web.archive.org/web/20221130042103/https://mercury-in.space/packages/packages.list`

[14]However, a list of available resources (managed by Volker Wysk) can be found at `https://volker-wysk.de/mercury/resources.html`; although they might not necessary be the ones you want.

1.3 Who's this book for

This book is for programmers who have previous experiences with imperative/functional languages who knows how programming works in principle and are used to filling in the details which could've been thought as incomplete or even missing. While learning Mercury as your first programming language might be a fun experiment to run on yourself, as the state of things being they are right now, it's just a little too much to be truly beginner-friendly, so I can't say I recommend it. Maybe years later when we have as much things as Haskell, this option would be viable again.

1.4 The tool we will be using

As of now (October 2025) there is only one implementation of Mercury: the Melbourne Mercury Compiler. All code in this book has been tested against version 22.01.8. Installing the compiler is probably one of the most painful part: since there is no official prebuilt binaries, you will have to download the source code, compile it and install it yourself. If you're using an operating system with a package manager (like most Linuxes), their package repositories might have packages that ease the complexity of managing a little bit, but the package might simply be a automated build instructions without any pre-built binaries and your machine still needs to build the compiler, which takes a very long time.

Strangely, for the Melbourne Mercury Compiler, I actually recommend you to build from the source code instead of relying on the package repository available: MMC contains different backends and configurations called "grades" and is actually capable of generating code for Java and C Sharp:

> `-s` *`grade`*
> `--grade` *`grade`*
>
> Select the compilation model. This model, which Mercury calls a grade, specifies what properties the resulting executable or library should have. Properties such as '`generates profiling data for mprof when executed`' and '`can be debugged with the Mercury debugger`'. As such, it controls decisions that must be made the same way in all the modules of a program.
>
> — *From the man page of* `mmc`

Certain things also requires a separate "grade" being available (e.g. weirdly, single-precision floating point numbers; see the section about basic types in Chapter 3). The package in the package repository of your operating system might not be configured with the grades you're interested in enabled.

1.5 More importantly... *why?*

A standard answer to this question would be "why not?"[15], but I believe that people who would ask would like a different answer. The biggest appeal of Mercury, I think, is probably its ability to provide the power of both functional and logic programming in a rigorous way that can support developments of large-scale projects, much like having a static type system has been proven to be helpful (at least to certain degree) in this regard – before this, most logic programming languages either don't have the static checks Mercury do or were only really suitable for academic purposes. You may argue that you don't need all those and can still do everything with conventional languages like C; I will give you the win since I have no arguments against that, but can you truly convince yourself that that's a good argument?

[15]But seriously, if you've got the time, *why not?*

CHAPTER 2

A First Taste of Mercury

Unfortunately, due to Mercury currently not having a read-eval-print-loop, it's nigh impossible to properly introduce the basic part of Mercury in details without introducing some kind of scaffolding first. For this reason, in this chapter we'll be forcing our way into writing and completing a tiny utility program; after this, we will have a basic idea of how to write a bare-bone command line program in Mercury, which we can then use as a starting point for exploring the language.

2.1 Prerequisite: logic programming

While it might not seem obvious from an outsider's view, logic programming did in fact come from 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 $Q \leftarrow P_0 \wedge P_1 \wedge \cdots \wedge P_n$. We will read this as "$Q$ holds when all of P_0 and P_1 and the rest – all the way to $_n$ – holds"
- "Fact", which takes the form of $P \leftarrow \top$. We will read this as "P holds (regardless)". Sometimes it's also shortened further into $P.$ in some literature.
- "Goal", which takes the form of $\perp \leftarrow P_0 \wedge P_1 \wedge \cdots \wedge P_n$. We will read this as "Show all of P_0 and P_1 and the rest holds".

Within these formula, all the Q and P_n have the form of $f(arg_0, arg_1, ...arg_n)$ where f is simply a name and all the arg_n are themselves 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 Horn clauses:

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

Compare this with the following:

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

...and the following code in Python:

```python
def fac(n):
    if n == 0: return 0
    elif n == 1: return 1
    else:
        n0 = n - 1
        m0 = fac(n0)
        return n * m0
```

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 function calls & entry points;

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

Unification

One thing to keep in mind is that theoretically there is no "computation" (as in calculation) involved; what we have (mostly) is a thing we call *unification*.[1] Unification roughly means to check if two terms "match".

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[2], 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 thus can be said as follows:

- A variable matches with anything (unless it forms recursive reference; see below);
- Atoms only match with themselves, e.g. the atom *apple* would only match with *apple* and will not match with *pear*. (But obviously it would still match with variables, since variables match with anything)
- 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.

[1]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.

[2]The name "functor" has been used to refer to a handful of different concepts; in the context of logic programming, especially languages that's derived from Prolog, you can simply think of it as "some data that's shaped like a function call".

- 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, although 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.

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 $\bot \leftarrow P_0 \land \cdots \land P_n$ (we'll omit the $\bot$ 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", which could be empty or containing other goals.

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 program in Horn clauses:

$$
\begin{aligned}
&parent(tom, john) \leftarrow \top \\
&parent(tom, evans) \leftarrow \top \\
&parent(tom, sarah) \leftarrow \top \\
&sibling(A, B) \leftarrow parent(C, A) \wedge parent(C, B) \wedge not(equal(A, B))
\end{aligned}
$$

...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 $\leftarrow parent(tom, A_0) \land not(equal(john, A_0))$.
- 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 $\leftarrow parent(tom, A_0) \land not(equal(john, A_0))$, 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.

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.

Disjunction

Other than conjunction which can be interpreted as sequencing, one can also use disjunction in Horn clauses, which can be interpreted as branching. 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; 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). The order in which each branch gets resolved theoretically shouldn't matter, but in reality different order would lead to different backtracking tree, which in languages with cut (e.g. Prolog) and similar constructs could lead to order-dependent results.

Unification, part 2

In short, due to how unification works, 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:

$$
\begin{aligned}
&append(nil, nil, nil) \leftarrow \top \\
&append(nil, A, A) \leftarrow \top \\
&append(cons(H, T), A, cons(H, R)) \leftarrow append(T, A, R)
\end{aligned}
$$

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 $A = cons(c, nil), B = nil$.)

2.2 Writing the program

In this part of the chapter, we will trying to write a program which I called `rall`, which stands for "adding carraige **r**eturn to **all** lines". From this "full name" you can tell it 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.

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).[3] Each module would need to have at least one `interface` section and one `implementation` section.

For our program, we would save the file under the name `rall.m`, thus the module name would be `rall`.

```
:- module rall.
:- interface.
:- implementation.
```

Entry point

With Mercury, the entry point of the program (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:[4]

[3]It's possible to have the module name *not* to be the same as the file name, but it's too early to talk about it here. See Chapter 3.

[4]As you may have noticed, this is technically a *linking error* instead of a *compile error*, which means that the compiling *itself* did not generate any error. `main` with other arities does not get special treatment like `main/2` do, and some people do adopt the pattern of having both a `main/2` and a different `main` with a different arity.

```
% 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'.
```

To be able to use `io.state`, we should first import the `io` module, which is done with the `import_module` declaration.

So we now have the following:

```
:- 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 same as `argc` and `argv` which 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”.

```
% 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 (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`.

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...

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 (
          % you can specify which stream you are writing to btw
          io.write_string(
              io.stderr_stream,
              "usage: rall [inputfilename] [outputfilename]\n",
              !IO)
      )
      else (
          % ...
      )
    ).
```

(Note that if you have previous experience with Prolog, the condition for the `if` construct might be surprising, since in Prolog, `=` unify things *structurally*; for example, the goal `3 + 4 = 7` would fail in Prolog despite 3 plus 4 do equal to 7, since `3 + 4` is a functor (interpreted as `` `+`(3,4) ``) and `7` is an atom; this is different in Mercury, where things *would* be evaluated. Also, `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.

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`, which are predicates/functions that uses 0-based index and 1-based index respectively; but these aren't the ones we would choose to use here 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)
          % ...
      )
    ).
```

By the way, if we import module simply by doing `:- import_module` *name*, We don't need to fully specify the module name, as long as there's no ambiguity. Naturally, the good practice for a lot of common cases is to fully specify; but some editors (Emacs, for example) might implement some kind of "smart indent" algorithm which could put the cursor at undesirable positions and be a nuisance. In cases like this, you might find writing code this way to be more comfortable.

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

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).
```

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)
        )
    ).

```

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)
        )
    ).

```

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)
        )
    ).
```

Listing 2.1: Full code for `rall.m`

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

We can compile it and produce an executable by calling the command `mmc rall.m`.

CHAPTER 3

Programming in Mercury

3.1 Trace Goals

Before we continue we must talk about how to explore things on one's own. A common strategy of debugging in conventional languages would be to insert printing statements at places where one would expect the program to reach with its state containing certain values; this strategy doesn't work well with Mercury, since to perform IO in Mercury you would need to do it through `io.state`, which naturally has the implication of spreading out and forcing everything that uses the predicate to adopt two extra `io.state` parameters.[1]

For example, assume we have the following program that prints out the factorial of 5:

```
:- module ex01.
:- interface.
:- import_module io.
:- pred main(io.state::di, io.state::uo) is det.
:- implementation.

:- import_module int.

:- pred fac(int, int).
:- mode fac(in, out) is det.
fac(N, M) :-
    ( if ( N = 0 ) then ( M = 1 )
      else if ( N = 1 ) then ( M = 1 )
      else if ( N < 0 ) then ( M = 1 )
      else ( fac(N-1, M1), M = N * M1 )
    )
```

[1]Which is not unlike the famous "color of function" problem in languages with `async` (e.g. JavaScript)

```
.

main(!IO) :-
    io.write_string("Hello, world!\n", !IO),
    fac(5, M),
    io.write_int(M, !IO).
```

Let's say we want to add a printing goal in `fac`. Printing, naturally, requires two `io.state` just like we've been mentioning. If we do it by adding `io.state`, we need to perform the following changes:

- Change the `pred` line into `:- pred fac(int, int, io.state, io.state)`;
- Change the `mode` line in a similar fashion;
- Change every occurence of `fac` accordingly;
- Since for every call site of `fac` we now need to pass two extra `io.state` arguments, the environment surrounding the call site should have those extra arguments ready;
  - and for some predicates this could mean adding extra parameters just like what we've done to `fac`;
  - and now the call sites of *those* predicates also need to have their signature changed, and this change would also affect call sites that calls *them*, and the need of change would thus propagate upwards...
- ...and after the debugging is done, we need to change them *back*.

This kind of coding process not only goes against the spirit of Mercury but is also just extremely tedious. Luckily, for this Mercury prepared a feature called **trace goals**.

Trace goals in Mercury are of the format `trace [`*`param1, ...`*`] Goal`. In such a goal, all variables bound in the parameter list part is available within the *`Goal`* part. There are many different parameters that one can use, but the most common is probably going to be `io`; by having `io(!IO)` which bound a pair of `io.state` to the state variable of your choice, you can use it to display debug messages in *`Goal`* freely without having to do the changes listed above. For example, assume we want our program to display the value of each call to `fac` before they return, we can add a trace goal that introduces a pair of `io.state` arguments like this:

```
:- module ex01.
:- interface.
:- import_module io.
:- pred main(io.state::di, io.state::uo) is det.
:- implementation.

:- import_module int.

:- pred fac(int, int).
:- mode fac(in, out) is det.
fac(N, M) :-
    ( if ( N = 0 ) then ( M = 1 )
      else if ( N = 1 ) then ( M = 1 )
      else if ( N < 0 ) then ( M = 1 )
      else ( fac(N-1, M1), M = N * M1 )
    ),
    trace [ io(!IO) ] (
        io.write_string("fac(", !IO),
        io.write_int(N, !IO),
        io.write_string(") = ", !IO),
        io.write_int(M, !IO),
        io.write_string("\n", !IO)
    )
.

main(!IO) :-
    io.write_string("Hello, world!\n", !IO),
    fac(5, M),
    io.write_int(M, !IO).
```

Notice that without changing the signatures of `fac` we can still obtain `io.state` and display things.

3.2 Using the standard library

As you read through the documentation of Mercury's standard libraries[2], you'll see the notice saying that "we recommend that if possible you use only those modules whose stability is described as either 'medium to high' or 'high'", you'll see a `stability` declaration in each of the documentation page, and you'll see that certain modules that *should* be stable, like `int` and `uint`, has a stability of `low`; should you rely on these modules that are of low stability? Sometimes the answer is yes – these modules have reached `high` stability in the latest nightly, which means that the interface of these modules, as it's described in the documentation of latest nightly, shouldn't change so much that it breaks basic programs in the future, and thus should be safe to use. In general, my recommendation is to read the

[2]Available at `https://mercurylang.org/information/doc-release/mercury_library/index.html` and `https://mercurylang.org/information/doc-latest/mercury_library_manual/index.html`, for latest stable and latest nightly, respectively.

documentation for the latest nightly when you're not sure if you can still use the same library predicate when the next major release come out.

3.3 Basic Datatypes

Finally, after all that build-up, we can look at the basic datatypes.

Signed and unsigned integers

In Mercury, the types of signed integers are: `int`, `int8`, `int16`, `int32` and `int64`, and the types of unsigned integers are: `uint`, `uint8`, `uint16`, `uint32` and `uint64`. The types with a number suffix are exactly what you think they are, e.g. `int8` means an 8-bit signed integer, etc.. The length of `int` is said to be "implementation-defined" but at least 32 bits. If you want to know how long an `int` actually is, you can have the following program:

```
:- module ex01.

:- interface.

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

:- implementation.
:- import_module int, string.


main(!IO) :-
    int.bits_per_int(BPI),
    string.int_to_string(BPI, BPIStr),
    io.write_string(BPIStr, !IO),
    io.write_string("\n", !IO).
```

As of now (October 2025), to enable the arithmetic and comparing operations of these types you must import the corresponding module, e.g. to be able to compare `uint` you must import the `uint` module. Actually, as a rule of thumb, if you need to do stuff with any of these types (and the basic types described in sections below), it's good measure to import the corresponding module even if you ended up not needing them.

Syntax of integers

A decimal integer is any sequence of decimal digits. Unlike in some languages, having the prefix `0` doesn't make it octal. In Mercury, the corresponding prefixes for non-decimal integers are: `0b` for binary, `0o` for octal, and `0x` for hexadecimal. Mercury also supports a prefix of `0'` followed by any single character; this would result in the character code that character has. (Note that the character immediately after the single quote is considered part of the integer literal regardless of its common usage in other parts of Mercury code; e.g. to get the character code of the character `,` you write `0',` with no extra measure of escaping. This works for Unicode characters as well; see the example below.)

A suffix may also be present, with different suffixes and their meanings listed as follows:

Suffix	Type
i or no suffix	Signed (`int`)
i8	Signed 8-bit (`int8`)
i16	Signed 16-bit (`int16`)
i32	Signed 32-bit (`int32`)
i64	Signed 64-bit (`int64`)
u	Unsigned (`uint`)
u8	Unsigned 8-bit (`uint8`)
u16	Unsigned 16-bit (`uint16`)
u32	Unsigned 32-bit (`uint32`)
u64	Unsigned 64-bit (`uint64`)

An arbitrary number of underscore `_` can be inserted between the radix prefix and the first digit (e.g. `0x_123`), between the digits (e.g. `0b_0001_0011`, and between the last digit and the type suffix (e.g. `0b10010011_u8`. Inserting underscores do not affect the actual value.

An example program showcasing the syntax for integers is listed as follows.[3]

```
:- module ex01.

:- interface.

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

:- implementation.
:- import_module string.
:- import_module int, int8, int16, int32, int64.
:- import_module uint, uint8, uint16, uint32, uint64.

main(!IO) :-
    io.write_string(string.int_to_string(32i) ++ "\n", !IO),
    io.write_string(string.int8_to_string(32i8) ++ "\n", !IO),
    io.write_string(string.int16_to_string(32i16) ++ "\n", !IO),
    io.write_string(string.int32_to_string(32i32) ++ "\n", !IO),
    io.write_string(string.int64_to_string(32i64) ++ "\n", !IO),
```

[3]As you can see, sometimes some lexers cannot properly handle valid Mercury code...

```
    io.write_string(string.uint_to_string(32u) ++ "\n", !IO),
    io.write_string(string.uint8_to_string(32u8) ++ "\n", !IO),
    io.write_string(string.uint16_to_string(32u16) ++ "\n", !IO),
    io.write_string(string.uint32_to_string(32u32) ++ "\n", !IO),
    io.write_string(string.uint64_to_string(32u64) ++ "\n", !IO),
    io.write_string("\n", !IO),
    io.write_string(string.uint_to_string(0x_123u) ++ "\n", !IO),
    io.write_string(string.uint_to_string(0b_0001_0011u) ++ "\n", !IO),
    io.write_string(string.uint8_to_string(0b0001_0011_u8) ++ "\n", !IO),
    io.write_string(string.int_to_string(0'K) ++ "\n", !IO),
    io.write_string(string.int_to_string(0' ) ++ "\n", !IO),
    io.write_string(string.int_to_string(0',) ++ "\n", !IO),
    io.write_string(string.int_to_string(0':) ++ "\n", !IO).
```

Floating-point numbers

In Mercury, there is only one type of floating-point number: `float`. As of now (October 2025; MMC 22.01.8), the default size of this type is double-precision (i.e. 64 bits). Single-precision can be enabled by using the `.spf` grade suffix, which means to use the command line argument `--grade asm_fast.gc.spf` or `--grade hlc.gc.spf` or others while compiling, depending on which backend you want to use. `.spf` isn't always available; if your call to `mmc` results in an error like this one:

```
mercury_compile: error: the Mercury standard library cannot be found in grade
 `asm_fast.gc.spf'.
```

It means that this particular configuration of backend isn't installed and is thus unavailable. This is likely to be the case, since the default configuration does not include `.spf` grades.

Character and string

In Mercury, the character type is `char` and the string type is `string`.

Syntax of string

Strings in Mercury are enclosed in double-quotes `"`. A few escape sequences are supported:

Escape sequence	Meaning	Example
`\\`	Backslash \	`\\`
`\'`	Single quote	`\'`
`\"`	Double quote	`\"`
`\a`	Beep	`\a`
`\b`	Backspace	`\b`
`\r`	Carriage return	`\r`
`\f`	Form feed	`\f`
`\t`	Tab	`\t`
`\n`	Newline	`\n`
`\v`	Vertical tab	`\v`
`\xXX\`	Hexadecimal escape	`\x37\`
`\XXX\`	Octal escape	`\037\`
`\uXXXX`	Unicode	`\u3707`
`\UXXXXXXXX`	Unicode	`\U00003707`

The length of a hexadecimal or an octal escape sequence is not predetermined (viz. there can be 2, 3, 4 or potentially more hexadecimal/octal digits that comes after a `\x`/`\` prefix); but if the escaped value turned out to be invalid (e.g. a very big number that goes over the Unicode range), a compile error would occur.

The ending backslash for a hexadecimal/octal escape sequence is necessary. For example, this small program:

```
:- module ex01.

:- interface.

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

:- implementation.

main(!IO) :-
    io__write_string("\x4b\\\\n", !IO),
    io__write_string("\x4b\\\n", !IO).
```

...prints two lines of text: `K\` and `K\n`, with the second line *not* ending in a newline character. This is because the first string is being read as the combination of `\x4b\`, `\\` and `\n`, but the second string is being read as the combination of `\x4b\`, `\\` and `n`.

Other than `\"`, you can also use `""` to represent one single double-quote character within a string.

Multi-line string literals are also possible by having a single backslash at the end of the line; the backslash character and the newline character after it would be ignored. Unlike some languages, you need to insert newline manually; for example, the first call of the following would only produce *one* single line of output:

```
    % ...
    io__write_string("a\
    multi\
        line\
            string\
                literal\
", !IO),
    io__write_string("a\n\
    multi\n\
        line\n\
            string\n\
                literal\n\
", !IO),
    % ...
```

Syntax of character

The syntax of character is slightly trickier to describe. In many languages, characters are quoted with single-quote `'`; but per Prolog tradition, single-quotes are also for atoms (or *names*, as defined in Mercury's language manual), which may contain more than one character. Currently, it seems like that Mercury treats quoted names containing one single character (or escape sequence) as character literals under the right context.

One needs to be careful when using quoted atoms when they intend to write character literals: if a character is an operator, simply quoting it isn't enough, since Mercury considers single-quoted operators are still operators themselves. To prevent such cases from being interpreted as an operation of values, one should surround the quoted name with parentheses. For example, `Char = '+'` is syntatically invalid, since it's being interpreted as `Char = +`; one should write `Char = ('+')` in this case.

Tuples

Mercury supports using N-tuple for any N larger than or equal to zero. In Mercury, tuples are enclosed with braces `{}` both for types and for values. See the example below:

```
:- module ex01.

:- interface.

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

:- implementation.
:- import_module int, string.

:- pred f1({int, string}::in, {string, int}::out) is det.
f1({X, Y}, {Y, X}).

main(!IO) :-
    % bind X to a tuple value.
    X = {3, "abc"},

    % get a reversed tuple w/ f1 (defined above).
    f1(X, Reversed_X),

    % break the reversed tuple down into different values.
    {MyString, MyInteger} = Reversed_X,
    io.write_string(string.int_to_string(MyInteger) ++ ", " ++
    MyString ++ "\n", !IO).
```

Maybe

`maybe` is a very common construct and occurs in many ML[4]-inspired languages under names like “Maybe”, “Option” or “Optional”. In Mercury, it’s defined as follows:

```
:- type maybe(T)
    --->    no
    ;       yes(T).
```

Essentially, it’s a type whose values can be either a wrapper that contains something (i.e. `yes(SomeValue)`) or an empty value that doesn’t contain anything (i.e. `no`). It’s often seen as a stricter (in terms of type-checking) version of having the null pointer, and some OOP languages have adopted this for this reason.

Many variants of Maybe exists for slightly different purposes. The most common variant is probably where one combines one “succeed” type and one “failure” type of value (which is often a string type representing the error message), like the Haskell `Either` and the Rust `Result`. Mercury provide three different variants of this usage:

```
:- type maybe_error
    --->    ok
    ;       error(string).

:- type maybe_error(T) == maybe_error(T, string).

    % Either a T, or an error E.
:- type maybe_error(T, E)
    --->    ok(T)
    ;       error(E).
```

[4]“Metalanguage”, i.e. the “ML” in Standard ML and OCaml, not “machine learning”.

Universal type

In Mercury, this type (named `univ`) is simply a type that can be used to contain any value, similar to `Object` in Java for objects. It is sometimes used to produce values that need to be able to have different types; for example, the `list` type in Mercury is homogeneous (i.e. its elements must be of the same type), so one way to achieve heterogeneous lists is to convert all the values of different types into `univ` type and have a value of `list(univ)`.

Two main predicates have been provided in the `univ` module: `type_to_univ/2` and `univ_to_type/2`, which converts value into `univ` and converts `univ` into other types of values respectively. A function `univ/1` is also defined to have the same effect of predicate `type_to_univ/2`, which also probably get used more often.

Converting from `univ` to other types needs extra caution; since you don't know what the underlying type of an `univ` actually is during compile time, any such conversion could have a chance of failure. The predicate `univ_to_type/2` is thus naturally of determinism `semidet`. You can also use `det_univ_to_type/2`, which would throw a runtime error when the type mismatches.

Here is an example program showing how one can implement a form of heterogeneous list with `univ`:

```
:- module ex01.

:- interface.

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

:- implementation.

:- import_module string, list, exception, univ.

:- func elem_to_string(univ::in) = (string::out) is det.
elem_to_string(X) = Y :-
    % the `:int`, `:string` and `:float` in the end is necessary.
    % this is used to specify which type you wish to convert the univ
    % value to.
    (univ_to_type(X, XVal:int) -> string.int_to_string(XVal, Y)
    ; univ_to_type(X, XVal:string) -> Y = XVal
    ; univ_to_type(X, XVal:float) -> string.float_to_string(XVal, Y)
    ; throw("Invalid type")
    ).

:- pred write_elem_list(list(univ)::in, io.state::di, io.state::uo) is
     det.
write_elem_list([], !IO).
write_elem_list([X|Xs], !IO) :-
    io.write_string(elem_to_string(X), !IO),
    io.write_string("\n", !IO),
    write_elem_list(Xs, !IO).
```

```

main(!IO) :-
    X = [univ("x"), univ(123), univ("y"), univ(456.78)],
    write_elem_list(X, !IO).
```

3.4 If-Then-Else

There are two syntaxes for if-then-else:

```
% syntax 1:
if [condition]
then [then-clause]
else [else-clause]

% syntax 2:
[condition]
  -> [then-clause]
  ; [else-clause]
```

The second one is a carry-over from Prolog.

These are not just used for goals; you can use them as expressions as well. See the example below:

```
main(!IO) :-
    io__write_int((3 > 5 -> 999; 1000), !IO).

% prints "1000"
```

3.5 Fundamental concepts

We must go through some of the concepts that occurs frequently in Mercury before we move on; since understanding them is vital for understanding some of the more advanced things about this language...

Instantiatedness and mode

In Chapter 2 we discussed about some of the most fundamental instantiatedness[5] (`free`, `bound`, `ground`) and modes (`in`, `out`). Insts in Mercury are not necessarily about being bound or not – it's something that describes a term's properties beyond its type (and mode describes the *change* of a term's insts instead of the insts themselves; so what we've been calling as the "mode signature" should really be called something like "inst declaration", but I digress). We will discuss this more in Chapter 4.

[5]The word "instantiatedness" is too long. From now on, we will be calling it "inst"s instead.

Determinism

As we have seen in Chapter 2, in Mercury, the following determinism categories exist:

- 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 (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.

Commited-Choice Determinism

Other than these six categories there are also two "committed-choice" classes: `cc_nondet` and `cc_multi`, which are committed-choice variants of `nondet` and `multi` respectively. `cc_nondet` and `cc_multi` goals backtrack, but when declared to be of `cc_nondet` or `cc_multi`, they *commit* to the first choice they would make during the resolution process and will not backtrack. This kind of determinism is thus used in cases where there could be multiple solutions but you don't care which one you end up with.

In Mercury, the `main` entry point can be defined to be either `det` or `cc_multi`, since both of them guarantee one and would only produce 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.)

3.6 Defining functions

By now we have all seen how to define *predicates*: you use the `:- pred` declaration like this:

```
:- pred pred_name(arg1_type, arg2_type, ..., argn_type).
:- mode pred_name(arg1_mode, arg2_mode, ..., argn_mode) is pred_det.
```

For example, we have defined a few predicates in Chapter 2 like this:

```
% "string" and "io.state" are types.
% "in", "di" and "uo" are modes.
% "det" is determinism.
:- pred rall(string, string, io.state, io.state).
:- mode rall(in, in, di, uo) is det.
```

Sometimes people declare the type signature and the mode signature separately, because in logic programming predicates can work "in reverse" and thus would require multiple mode signature. Take the signatures of the predicate `list.append` for example:

```
:- pred append(list(T), list(T), list(T)).
:- mode append(di, di, uo) is det.
:- mode append(in, in, out) is det.
:- mode append(in, in, in) is semidet.    % implied
:- mode append(in, out, in) is semidet.
:- mode append(out, out, in) is multi.
```

To explain this line by line:

- `list(T)` is a type, where `T` is a type variable, declaring that it works for any `list`.
- The first mode is simply a version of the second one that works on unique inputs.
- The second mode is for appending two lists together into a third list.
- The third mode is to check if combining the first two arguments result in the same value as the third argument. The check could succeed or fail, thus having the overall determinism of `semidet`.
- The fourth mode is to remove the "prefix" (described by the first argument) from the third argument. The third argument might not contain the first one as a prefix, thus having the overall determinism of `semidet`.
- The fifth mode asks the question of "which two lists can be combined and create the third one". There obviously must be at least two solutions (exercise: can you figure out why this is the case?) and

there could be even more for some input, thus having the overall determinism of `multi`.

```
:- func f_name(arg1type, arg2type, ...) = result_type.
:- mode f_name(arg1mode, arg2mode, ...) = result_mode is f_det.
```

For example, a factorial function might have the following declaration:

```
:- func factorial(int) = int.
:- mode factorial(in) = out is det.
```

You also need to write `f_name(args) = result` instead of `f_name(args, result)` when providing the definition; other than that, the rest is mostly the same. Compare the following two definitions of the factorial function, one defined as a predicate with only one mode, one defined as a function:

```
:- pred factorial1(int, int).
:- mode factorial1(in, out) is det.
factorial1(X, Y) :-
    ( X =< 1 -> Y = 1
    ; ( factorial1(X - 1, Y0),
        Y = X * Y0
      )
    ).

:- func factorial2(int) = int.
:- mode factorial2(in) = out is det.
factorial2(X) = Y :-
  ( X =< 1 -> Y = 1
  ; Y = X * factorial2(X - 1)
  ).
```

Just like predicate declaration, you can combine the type signature and the mode signature in one; and if you do, you need to wrap the return type in parentheses, like this:

```
:- func factorial2(int::in) = (int::out) is det.
```

You can technically also declare functions (and predicates) like this, but this is technically not the “proper” way. The semantics of this, is to declare the function as a whole is of some higher-order function/predicate type.

```
:- func factorial2 : (func(int) = int).
```

A function, just like a predicate, can have multiple mode declaration. While most (if not all) fucntions we have come across would have their arguments having the mode `in` and their return value the mode `out`, but in Mercury, the reverse (i.e. arguments having `out` and return value having `in`) is technically possible. See the following example, in which the definition of function `f1` can be seen as asking the question “which tuple value can be seen as a result of reversing which tuple’s components”:

```
:- module ex01.

:- interface.

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

:- implementation.
:- import_module int, string.

:- func f1({int, string}) = {string, int}.
:- mode f1(out) = in is det.
f1({X, Y}) = {Y, X}.

:- func first({T1, T2}) = T1.
:- mode first(in) = out is det.
first({X, _}) = X.

main(!IO) :-
    {"abc", 3} = f1(P),
    io.write_string(string.int_to_string(first(P)) ++ "\n", !IO).
```

3.7 User-defined types

Equivalence types

The phrase “equivalence types” in Mercury refers to what’s more commonly known as *type aliases*, i.e. they are types that are defined to be fully equivalent in every way to some other types. This is often done for the sake of code readability, for example:

```
:- type money == int.
:- type assoc_list(Key, Value)
        == list(pair(Key, Value)).
```

While `money` and `assoc_list(Key, Value)` are equivalent to `int` and `list(pair(Key, Value))`, one reading the code would have a basic idea that certain `int`s are intended for values representing an amount of money and certain `list`s are intended to be used as an association list.

(Honestly, the name “equivalence types” might bring confusion for people who have a Standard ML background, in which the idea of *equality type* refers to types that one can compare values with `=` in instead of type aliases, similar to types that have an `Eq` typeclass constraint in Haskell.)

Discriminated unions

Discriminated union, or “tagged union” is a very useful (and thus commonly supported among functional languages) construct. First of all, it can be used to implement what's often called the "enumerated types" in other languages. For example:

```
:- type days_of_week
        ---> monday
        ;    tuesday
        ;    wednesday
        ;    thursday
        ;    friday
        ;    saturday
        ;    sunday.
```

This is not unlike the following in C:

```
typedef enum DaysOfWeek {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSAY,
    FRIDAY,
    SATURDAY,
    SUNDAY,
} DaysOfWeek;
```

The general syntax is as follows. (Note that the arrow here consists of three dashes. Mercury uses 2-dash arrow and 1-dash arrow for other things, so one should be mindful not to mix them up.)

```
:- type {type_name} --->
        {variant_1} ; {variant_2} ; ... ; {variant_n} .
```

The core idea of a discriminated union type is exactly how it sounds: it's a union type with a special value (often called a "tag"; this is also why discriminated unions are also called "tagged union") that one uses to discern the different possible forms of the union. For example, you might often see patterns like this in languages like C:

```
typedef enum ExprType {
    LIT_INTEGER,
    BIN_OP,
    UNARY_OP,
    FUNC_CALL,
} ExprType;

typedef struct Expr {
    ExprType typ;
    // the ~typ~ field is checked before using the ~value~ field.
    union {
        int lit_int;
        struct { struct Expr *l, *r; char* bop; } bin_op;
        struct { struct Expr *b; char* uop; } un_op;
        struct { char* fun_name; struct Expr* args } func_call;
    } value;
}
```

It can be simply defined in Mercury as follows:

```
:- type expr
        ---> lit_integer(int)
        ;    bin_op(expr, expr, string)
        ;    unary_op(expr, string)
        ;    func_call(string, list(expr)).
```

We don't need to have a separate `:- type exprtype`; the tags are "built into" the type `expr`. When a type is defined this way, these tags are also defined as constructors, i.e. we can obtain a value of `expr` by simply using the terms `lit_integer/1`, `bin_op/3`, `unary_op/2` or `func_call/2`; for example, the term `lit_integer(3)` and the term `bin_op( lit_integer(3), lit_integer(4), "+")` would've automatically determined to have the type of `expr`.

To retrieve the different values stored within, we can use unification:

```
% ...
SomeVar = some_funcall(some_args),
( ( SomeVar = lit_integer(IntVal),  % ...do something w/ IntVal
  );
  ( SomeVar = bin_op(Left, Right, Op),  % ...do something
  );
  ( SomeVar = unary_op(Body, Op),  % ...do something
  );
  ( SomeVar = func_call(Head, Args),  % ...do something
  )
)
```

This can be tedious at times, especially in cases where you need to have a new value where only some of the fields have changed, in which you have no choice other than reconstructing the term from the ground up once you've defined your data type this way. But - if we choose to use the record syntax,

```
:- type expr
        ---> lit_integer(lit_int :: int)
        ;    bin_op(l :: expr, r :: expr, bop :: string)
        ;    unary_op(b :: expr, uop :: string)
        ;    func_call(fun_name :: string, args :: list(expr)).
```

Certain commonly-used types are actually defined this way. See the definition of lists and key-value lists:

```
% in module `list`
:- type list(T)
    --->    []
    ;       [T | list(T)].

% in module `kv_list`
:- type kv_list(K, V)
    --->    kv_nil
    ;       kv_cons(K, V, kv_list(K, V)).
```

Booleans are also defined this way:

```
% in module `bool`
:- type bool
    --->    no
    ;       yes.
```

Abstract types

Simply put, they are the declarations we've discussed above but without the definition part. These are mainly used in the `interface` section of a module for the cases where you don't wish to expose the internals of a type to other module:

```
:- interface.
:- type t1.
:- type t2(T1, T2).
% ...

:- implementation.
:- type t1 --> % ...
:- type t2(T1, T2) == % ...
% ...
```

Of course, you must properly define them in the `implementation` section of the same module.

3.8 Example 1: Interpreter of a small language

With what we have learned up till now we can now try to write some small programs.

The language

The language we'll be implementing would be a small, imperative one with very basic features like assignment, branching and loops. Its grammar can be described with the following Backus-Naur Form:

$\langle stmt \rangle$ ::= IDENT '`:=`' $\langle expr \rangle$
| '`IF`' $\langle expr \rangle$ '`THEN`' $\langle stmt \rangle$ '`ELSE`' $\langle stmt \rangle$
| '`BEGIN`' $\langle stmt \rangle$* '`END`'
| '`WHILE`' $\langle rel \rangle$ '`DO`' $\langle stmt \rangle$
| '`PRINT`' $\langle expr \rangle$

⟨*expr*⟩ ::= INTEGER
  | VAR
  | ⟨*expr*⟩ (‘+’|‘-’|‘*’|‘/’) ⟨*expr*⟩
  | (‘-’) ⟨*expr*⟩

⟨*rel*⟩ ::= TRUE
  | FALSE
  | ⟨*expr*⟩ (‘>’|‘<’|‘>=’|‘<=’|‘!=’|‘==’) ⟨*expr*⟩
  | ⟨*rel*⟩ (‘&&’|‘||’) ⟨*rel*⟩
  | ‘!’ ⟨*rel*⟩

From this BNF we can easily obtain the type definition:

```
:- import_module list.
:- type imp_expr
   ---> e_val(int)
   ; e_var(string)
   ; e_binop(string, imp_expr, imp_expr)
   ; e_unaryop(string, imp_expr).
:- type imp_rel
   ---> r_rel(string, imp_expr, imp_expr)
   ; r_binop(string, imp_rel, imp_rel)
   ; r_unaryop(string, imp_rel)
   ; r_true
   ; r_false.
:- type imp_stmt
   ---> s_assign(string, imp_expr)
   ; s_begin(list(imp_stmt))
   ; s_if(imp_rel, imp_stmt, imp_stmt)
   ; s_while(imp_rel, imp_stmt)
   ; s_print(imp_expr).
```

We will need a place to store the mapping between the variable names and their corresponding values:

```
:- import_module map.
:- type imp_env == map(string, int).
```

The evaluation of expression is easy. Since we don't need to update the environment while evaluating expressions for this language, we don't need an extra `imp_env::out` parameter:

```
% needed by `throw`
:- import_module exception.

:- pred imp_eval_expr(imp_expr::in, imp_env::in, int::out) is det.
imp_eval_expr(X, Env, Result) :-
    ( X = e_val(XVal), Result = XVal );
    ( X = e_var(XVar),
      ( map__contains(Env, XVar) -> Result = lookup(Env, XVar)
      ; throw("Cannot found var: " ++ XVar)
      )
    );
    ( X = e_binop(Op, L, R),
      imp_eval_expr(L, Env, LRes),
      imp_eval_expr(R, Env, RRes),
      ( Op = "+" -> Result = LRes + RRes
      ; Op = "-" -> Result = LRes - RRes
      ; Op = "*" -> Result = LRes * RRes
      ; Op = "/" -> Result = LRes div RRes
      ; throw("Unsupported bin op: " ++ Op)
      )
    );
    ( X = e_unaryop(Op, Body),
      imp_eval_expr(Body, Env, BRes),
      ( Op = "-" -> Result = 0 - BRes
      ; throw("Unsupported unary op: " ++ Op)
      )
    ).
```

Evaluating relations, however, requires some not-that-intuitive steps if you choose to first evaluate things into boolean values and dispatch on it. The comparison of numerical values are technically predicate calls, which only succeed or fail instead of producing a value like function calls; if we need a boolean value, we must convert this “succeed or fail” state into a boolean value. This is actually very simple, you just do this:

```
% the instantiatedness here restricts its argument to those that
% conforms to certain constraints. we will talk about this more
% in the section about instantiatedness.
:- func pred_to_bool( (pred)::in((pred) is semidet) )
        = (bool::out) is det.
pred_to_bool(P) = (if P then yes else no).
```

This is available in the standard library by importing the `bool` module.

Equality check is even more different - when written directly, they’re considered as a goal of unification instead a predicate call, which means that we can’t even use `pred_to_bool`. One way to go around this is define a dedicated predicate:

```
:- func int_eq(int::in, int::in) = (bool::out) is det.
int_eq(X, Y) = (if X = Y then yes else no).
```

With this being available we can define the predicate for evaluating relations like this:

```
:- import_module bool.
:- pred imp_eval_rel(imp_rel::in, imp_env::in, bool::out) is det.
imp_eval_rel(X, Env, Result) :-
    ( X = r_rel(Op, L, R),
      imp_eval_expr(L, Env, LRes),
      imp_eval_expr(R, Env, RRes),
      ( Op = ">" -> Result = pred_to_bool((RRes < LRes))
      ; Op = "<" -> Result = pred_to_bool((LRes < RRes))
      ; Op = "==" -> Result = int_eq(LRes, RRes)
      ; Op = ">=" -> Result = pred_to_bool((LRes >= RRes))
      ; Op = "<=" -> Result = pred_to_bool((LRes =< RRes))
      ; Op = "!=" -> Result = not(int_eq(LRes, RRes))
      ; throw("Unsupported rel op: " ++ Op)
      )
    );
    ( X = r_binop(Op, L, R),
      imp_eval_rel(L, Env, LRes),
      imp_eval_rel(R, Env, RRes),
      ( Op = "&&" -> Result = and(LRes, RRes)
      ; Op = "||" -> Result = or(LRes, RRes)
      ; throw("Unsupported rel op: " ++ Op)
      )
    );
    ( X = r_unaryop(Op, B),
      imp_eval_rel(B, Env, BRes),
      ( Op = "!" -> Result = not(BRes)
      ; throw("Unsupported rel op: " ++ Op)
      )
    );
    ( X = r_true, Result = yes );
    ( X = r_false, Result = no ).
```

The predicate for executing statements can thus be defined as follows. Evaluating a statement could update the environment (by assignment statements), so we need an `imp_env::out` to receive the newest version of it:

```

:- pred imp_exec_stmt(imp_stmt::in, imp_env::in, imp_env::out) is det.
:- pred imp_exec_stmt_list(list(imp_stmt)::in, imp_env::in, imp_env::
    out) is det.
:- pred imp_exec_stmt_while(imp_rel::in, imp_stmt::in, imp_env::in,
    imp_env::out) is det.

imp_exec_stmt(X, Env, NewEnv) :-
    ( X = s_assign(Var, Expr),
      imp_eval_expr(Expr, Env, Val),
      NewEnv = map__set(Env, Var, Val)
    );
    ( X = s_begin(Body),
      imp_exec_stmt_list(Body, Env, NewEnv)
    );
    ( X = s_if(Cond, Then, Else),
      imp_eval_rel(Cond, Env, CondRes),
      ( CondRes = yes -> imp_exec_stmt(Then, Env, NewEnv)
      ; imp_exec_stmt(Else, Env, NewEnv)
      )
    );
    ( X = s_while(Cond, Body),
      imp_exec_stmt_while(Cond, Body, Env, NewEnv)
    );
    ( X = s_print(Expr),
      imp_eval_expr(Expr, Env, Val),
      trace [ io(!IO) ] (
          io__write_int(Val, !IO),
          io__write_string("\n", !IO)
      ),
      NewEnv = Env
    ).

```

```
imp_exec_stmt_list([], X, X).
imp_exec_stmt_list([X|Xs], Env, NewEnv) :-
    imp_exec_stmt(X, Env, NewEnv1),
    imp_exec_stmt_list(Xs, NewEnv1, NewEnv).

imp_exec_stmt_while(Cond, Body, Env, NewEnv) :-
    imp_eval_rel(Cond, Env, CondRes),
    ( CondRes = yes -> (
          imp_exec_stmt(Body, Env, NewEnv1),
          imp_exec_stmt_while(Cond, Body, NewEnv1, NewEnv)
      )
    ; NewEnv = Env
    ).
```

You can tell the Mercury compiler didn't pick up the fact that `imp_exec_stmt_while` being potentially diverge since its determinism `det` somehow checks out.

A program in our language is conceptually just a list of statements. We will execute it by calling `imp_exec_stmt_list` with an empty environment:

```
:- pred imp_run(list(imp_stmt)::in) is det.
imp_run(X) :-
    Env = map__init,
    imp_exec_stmt_list(X, Env, _).
```

Then we can call it in our `main`:

```
main(!IO) :-
    % X := 100
    % SUM := 0
    % WHILE X > 0 DO
    %     BEGIN
    %         SUM := SUM + X
    %         X := X - 1
    %     END
    % PRINT SUM
    imp_run(
        [
            s_assign("X", e_val(100)),
            s_assign("SUM", e_val(0)),
            s_while(
                r_rel(">", e_var("X"), e_val(0)),
                [
                    s_assign("SUM", e_binop("+", e_var("SUM"), e_var("
    X"))),
                    s_assign("X", e_binop("-", e_var("X"), e_val(1)))
                ]
            ),
            s_print(e_var("SUM"))
        ]
    ).
```

Compile and run the program, and you'll see the number 5050 is printed.

Full listing

```
:- module ex01.

:- interface.

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

:- implementation.

:- import_module int, string, list, exception, bool, map.

:- type imp_expr
   ---> e_val(int)
   ; e_var(string)
   ; e_binop(string, imp_expr, imp_expr)
   ; e_unaryop(string, imp_expr).
:- type imp_rel
   ---> r_rel(string, imp_expr, imp_expr)
   ; r_binop(string, imp_rel, imp_rel)
   ; r_unaryop(string, imp_rel)
   ; r_true
   ; r_false.
:- type imp_stmt
   ---> s_assign(string, imp_expr)
   ; s_begin(list(imp_stmt))
   ; s_if(imp_rel, imp_stmt, imp_stmt)
   ; s_while(imp_rel, imp_stmt)
   ; s_print(imp_expr).

:- type imp_env == map(string, int).

:- pred imp_eval_expr(imp_expr::in, imp_env::in, int::out) is det.
imp_eval_expr(X, Env, Result) :-
    ( X = e_val(XVal), Result = XVal );
    ( X = e_var(XVar),
      ( map__contains(Env, XVar) -> Result = lookup(Env, XVar)
      ; throw("Cannot found var: " ++ XVar)
      )
    );
    ( X = e_binop(Op, L, R),
      imp_eval_expr(L, Env, LRes),
      imp_eval_expr(R, Env, RRes),
      ( Op = "+" -> Result = LRes + RRes
      ; Op = "-" -> Result = LRes - RRes
      ; Op = "*" -> Result = LRes * RRes
      ; Op = "/" -> Result = LRes div RRes
      ; throw("Unsupported bin op: " ++ Op)
      )
    );
    ( X = e_unaryop(Op, Body),
      imp_eval_expr(Body, Env, BRes),
      ( Op = "-" -> Result = 0 - BRes
      ; throw("Unsupported unary op: " ++ Op)
      )
    ).

:- func int_eq(int::in, int::in) = (bool::out) is det.
int_eq(X, Y) = (if X = Y then yes else no).
```

```

:- pred imp_eval_rel(imp_rel::in, imp_env::in, bool::out) is det.
imp_eval_rel(X, Env, Result) :-
    ( X = r_rel(Op, L, R),
      imp_eval_expr(L, Env, LRes),
      imp_eval_expr(R, Env, RRes),
      ( Op = ">" -> Result = pred_to_bool((RRes < LRes))
      ; Op = "<" -> Result = pred_to_bool((LRes < RRes))
      % ; Op = "==" -> Result = (LRes, RRes)
      ; Op = "==" -> Result = int_eq(LRes, RRes)
      ; Op = ">=" -> Result = pred_to_bool((LRes >= RRes))
      ; Op = "<=" -> Result = pred_to_bool((LRes =< RRes))
      ; Op = "!=" -> Result = not(int_eq(LRes, RRes))
      ; throw("Unsupported rel op: " ++ Op)
      )
    );
    ( X = r_binop(Op, L, R),
      imp_eval_rel(L, Env, LRes),
      imp_eval_rel(R, Env, RRes),
      ( Op = "&&" -> Result = and(LRes, RRes)
      ; Op = "||" -> Result = or(LRes, RRes)
      ; throw("Unsupported rel op: " ++ Op)
      )
    );
    ( X = r_unaryop(Op, B),
      imp_eval_rel(B, Env, BRes),
      ( Op = "!" -> Result = not(BRes)
      ; throw("Unsupported rel op: " ++ Op)
      )
    );
    ( X = r_true, Result = yes );
    ( X = r_false, Result = no ).

:- pred imp_exec_stmt(imp_stmt::in, imp_env::in, imp_env::out) is det.
:- pred imp_exec_stmt_list(list(imp_stmt)::in, imp_env::in, imp_env::
    out) is det.
:- pred imp_exec_stmt_while(imp_rel::in, imp_stmt::in, imp_env::in,
    imp_env::out) is det.

imp_exec_stmt(X, Env, NewEnv) :-
    ( X = s_assign(Var, Expr),
      imp_eval_expr(Expr, Env, Val),
      NewEnv = map__set(Env, Var, Val)
    );
    ( X = s_begin(Body),
      imp_exec_stmt_list(Body, Env, NewEnv)
    );
    ( X = s_if(Cond, Then, Else),
      imp_eval_rel(Cond, Env, CondRes),
      ( CondRes = yes -> imp_exec_stmt(Then, Env, NewEnv)
      ; imp_exec_stmt(Else, Env, NewEnv)
      )
    );
    ( X = s_while(Cond, Body),
      imp_exec_stmt_while(Cond, Body, Env, NewEnv)
    );
    ( X = s_print(Expr),
      imp_eval_expr(Expr, Env, Val),
      trace [ io(!IO) ] (
          io__write_int(Val, !IO),
          io__write_string("\n", !IO)
      ),
      NewEnv = Env
    ).

imp_exec_stmt_list([], X, X).
imp_exec_stmt_list([X|Xs], Env, NewEnv) :-
    imp_exec_stmt(X, Env, NewEnv1),
    imp_exec_stmt_list(Xs, NewEnv1, NewEnv).

imp_exec_stmt_while(Cond, Body, Env, NewEnv) :-
    imp_eval_rel(Cond, Env, CondRes),
    ( CondRes = yes -> (
          imp_exec_stmt(Body, Env, NewEnv1),
          imp_exec_stmt_while(Cond, Body, NewEnv1, NewEnv)
      )
    ; NewEnv = Env
    ).

:- pred imp_run(list(imp_stmt)::in) is det.
imp_run(X) :-
    Env = map__init,
    imp_exec_stmt_list(X, Env, _).

main(!IO) :-
    imp_run(
        [
            s_assign("X", e_val(100)),
            s_assign("SUM", e_val(0)),
            s_while(
                r_rel(">", e_var("X"), e_val(0)),
                s_begin(
                    [
                        s_assign("SUM", e_binop("+", e_var("SUM"), e_var("X"))),
                        s_assign("X", e_binop("-", e_var("X"), e_val(1)))
                    ]
                )
            ),
            s_print(e_var("SUM"))
        ]
    ).
```

3.9 Using multiple modules

There will be times where you would want to put things into different modules; it's better to talk about how to do this early than late.

Basic module usage

There are two ways to import a module:

- `import_module`, which we have been using up till now. Names imported this way do not need to be fully qualified; the module name, when referring to the names defined within, can be omitted until there's ambiguity.
- `use_module`, whose syntax is the same as `import_module`, but the names imported this way needs to be fully qualified all the time.

Most of the time people would use a period `.` to separate the module name and the referred name, but Mercury supports using two underscores `__` as well; for example, instead of calling `io.write_string`, you can call `io__write_string`.

There is also this thing called "module-local mutable variable", which does exactly what it sounds like. However, this feature involves Mercury's purity system, and for that reason we'll talk about it when we get there.

Building with multiple modules

The Melbourne Mercury Compiler comes with two build tools named `mmake` and `mmc --make`. The reason why the latter is named `mmc --make` is because that to use this build tool you do need use the same executable but with a special command line argument. Both covers roughly the same set of features (with an amount of difference in details), both are terribly confusing and under-documented, and both are very confusing. The official documentation recommends `mmc --make` because "this is the build tool that is likely to receive more development in the future". I recommend using `mmc --make` instead of directly calling `mmc` because compiling multi-module program with the Melbourne Mercury Compiler is quite complex and not a task that can be achieved by only calling `mmc`; it's not unlike building a multi-file C project.[6] For single-module programs (the ones we have seen till now), you can directly replace `mmc [main_module]` with `mmc --bare [main_module]` (e.g. `mmc --make rall.m` instead of `mmc rall.m`). For multi-module programs, you can also use `mmc --make [entry]` and expect it to handle module dependencies automatically and give you an executable without much errors.

Submodules

There are times when you'd like to group certain things within a module into its own subgroup; in Mercury this can be achieved by defining a submodule.

[6]I'd say Mercury might have more hassle than C at this point.

There are two kinds of submodule in Mercury: ones that are defined and embedded within the "main" module, and ones that are in a separate file. For the former ones, you do it like this:

```
:- module main_module.
:- interface.

:- module submod.
:- interface.
    % submod interface goes here.
:- end_module submod.

:- implementation.

:- module submod.
:- implementation.
    % submod implementation goes here.
:- end_module submod.
% ...
```

Naturally, if you don't want the definition of `submod` being visible from the outside, you put its interface section in the declaration of the parent module's implementation section; but to put the implementation section of a nested submodule in the interface section of the parent module is forbidden.

If you want to put the definition of the submodule in a separate file, you must do the following:

- The content of that submodule must be in a file with the file name being the fully qualified name of the submodule (e.g. the source for the submodule `submod` of the parent module `mainmod1` must be put in the file with the name `mainmod1.submod.m`). (Using a different module name is possible; see the next section.)
- Add an `include_module` declaration in the parent module, like this:

```
:- module parent_module.
:- interface.
:- include_module submod1, submod2, ..., submod_n.
```

Either way, having the definition of the submodule doesn't mean the names within the submodule is immediately available within the parent module; you still need to import the submodule explicitly if you want to use it.

Using a different module name

Normally module names should match with the name of the file; but if you insist on using a different name, the Mercury compiler would need to know which name corresponds to which file. The compiler would expect this information in a file named `Mercury.modules` residing in the same directory as the module that uses the name-mismatch module. The `mmc` command provides a flag to generate this file: to check all the modules within a directory and generate the `Mercury.modules`, run `mmc -f`. This file can technically be modified so that the build tool would use modules located at other places; one may manage different libraries a project might use this way, but I'm not sure this is part of the intended usage.

3.10 Higher-order programming

In Mercury, predicates and functions are "first-class citizen" in commonly-used functional language lingo; this means that we can produce them as output values, take them as arguments, and create them without binding them to any name. The form of these predicates/functions without name is aptly called "anonymous predicate/function"; they are also (and more often) called "*lambda* (expression)", which is a name from the lambda-calculus.

Currently you must explicitly state the mode and determinism in your anonymous predicate/function, but type is not required as long as Mercury could infer them. It's still recommended to be explicit about types anyway. The way you state both the type and the mode of a parameter is to write `arg_name:type::mode`, like this:

```
main(!IO) :-
    P = (pred(In:int::in, Out:int::out) is det :-
            trace [ io(!IO) ] (
                io__write_int(In, !IO),
                io__write_string("\n", !IO)
            ),
            Out = In + 1
        ),
    P(3, _).
```

State variables cannot be arguments. For example, this is not allowed:

```
main(!IO) :-
    P = (pred(In:int::in, !IO) is det :-
            io__write_int(In, !IO),
            io__write_string("\n", !IO)
        ),
    P(3, !IO).
```

You must state them separately with `!.` and `!:`, after which you can use it like normal:

```
main(!IO) :-
```

```
    P = (pred(In:int::in, !.IO:io.state::di, !:IO:io.state::uo) is det
     :-
            io__write_int(In, !IO),
            io__write_string("\n", !IO)
        ),
    P(3, !IO).
```

Anonymous functions can be written in a similar fashion, but you can omit the mode declaration, since functions are considered to have the default inst of `func(in, in, ...) = out is det`.

```
main(!IO) :-
    P = (func(In:int) = (Out:int) :-
            trace [ io(!DebugIO) ] (
                io__write_int(In, !DebugIO),
                io__write_string("\n", !DebugIO)
            ),
            Out = In + 1
        ),
    io__write_int(P(3), !IO),
    io__write_string("\n", !IO).

% outputs two line: "3" and "4".
```

Functions with no arguments need special care when calling. If you define a predicate with no arguments, you can still call it directly. For example:

```
main(!IO) :-
    P = ((pred) is det :-
            trace [ io(!DebugIO) ] ( io__write_string("P\n", !DebugIO
    ) )
        ),
    % this actually calls the predicate bound to `P`.
    P,
    io__write_string("\n", !IO).
```

You can't do the same to functions with no arguments. For example, this will complain about a type mismatch between `func = int` and the required `int`:

```
main(!IO) :-
    P = ((func) = 3),
    io__write_int(P, !IO),
    io__write_string("\n", !IO).
```

You can't solve this by writing `P:int` (which states that `P` should be of type `int`) either, and writing `P()` is syntax error. In order to call this, you need to use `apply`:

```
main(!IO) :-
    P = ((func) = 3),
    io__write_int(apply(P), !IO),
    io__write_string("\n", !IO).

% compiles and prints "3".
```

`apply` is a built-in expression that applies its first argument on its second, third, etc. arguments. For example, assume we have `P = (func(A, B, C) = A + B + C)`, the expression `apply(P,3,4,5)` would have the same result as `P(3,4,5)`. A similar construct named `call` is also available for calling predicates.

Predicates and functions, when applied to a number of arguments that is less than required, is automatically curried. For example, you can write `=<(0)` to create a function that can be used to filter out all the values that are equal or larger than zero (since `=<(A, B)` actually means `A =< B`, in our case we're effectively writing `0 =<`):

```
:- pred write_int_list(list(int)::in, io.state::di, io.state::uo) is
    det.
write_int_list([], !_).
write_int_list([X|Xs], !IO) :-
    io__write_int(X, !IO), io__write_string(", ", !IO),
    write_int_list(Xs, !IO).

main(!IO) :-
    list__filter(=<(0), [0, 1, 3, -2, 3, -4], Res),
    write_int_list(Res, !IO),
    io__write_string("\n", !IO).

% writes "0, 1, 3, 3, "
```

Built-in language constructs like `=` and `apply`, however, cannot be used this way, and requires one use an explicit anonymous predicate/function. For example, this is illegal:

```
% ...
lst__filter(\=(2), [1, 2, 3], Res)
```

...instead, you must write this or define a named predicate separately:

```
% ...
lst__filter((pred(X:int::in) is semidet :- X \= 2), [1, 2, 3], Res)
```

One thing to remember: you cannot use the whole state variable within an anonymous predicate/function; you can only use the "current value", whose semantics doesn't involve modifying the state-of-the-world and is thus allowed. For example, the compiler would complain about this code, saying that you can only use `!.IO` within the lambda:

```
main(!IO) :-
    P = (pred(Out:int::out) is det :-
            io__write_string("hey!\n", !IO),
            Out = 3
        ),
    P(_).
```

3.11 Exception handling

We already know that we can throw exceptions using `throw/1`. Now we learn how to handle them.

Handling with predicates

A few predicates have been provided in the `exception` module:[7]

- `try(Goal, Result)`: This goal succeeds regardless, with `Result` containing either `failed` (showing that `Goal` has failed without throwing exceptions), `succeeded(Val)` (showing that `Goal` has succeeded and producing a result of `Val` without throwing exception) or `exception(E)` (showing that `Goal` has thrown an exception, which is captured in `E`).

- `try_io(Goal, Result, !IO)`: Same as `try/2`, but with extra parameters for IO. texttttry/2 requires its goal to be a `pred(out)` and we can't directly use the IO state variable from the outside, thus dedicated predicate that would thread the state variable for us becomes necessary.

- `try_all(Goal, MaybeException, Solution)`: Collect all solutions of `Goal` which might throw exceptions into `Solution`. `MaybeException` would be bound with a value of type `maybe(univ)`; if an exception did get thrown, that `maybe` value would contain that exception; and if there has been no exception, it would simply be an empty value.

- `finally(Goal, Res, Cleanup, CleanupRes, !IO)`: The `Goal` and `Res` is the same as `try/2`, but `finally` would ensure to call `Cleanup` afterwards regardless of the result in `Res`, thus making this the “finally” construct in certain languages.

To unwrap thrown values from `throw/1` one needs to do some extra work when compared with other languages since the return value of `try/2` is a `exception_result(T)` which only contains the exception as an `univ`. For example, assume we have this goal:

```
    % this goal would definitely throw an exception since we
    % are indexing the 4th element of an empty list.
    try((pred(Out:int::out) is det :- det_index0([], 3, Out)), R),
```

[7]There are other predicates, which have slightly more niche usage so we will not cover them here.

...we would first need to unwrap the `exception_result(T)` part. We don't need to handle the `failed` case since our call of `try/2` uses a `det` goal, which results in `R` having the inst of `cannot_fail` which describes a subset of `exception_result` that only contains `succeeded/1` and `exception`[8]

```
    ( ( R = succeeded(K), io__write_string("succeeded!\n", !IO) );
      ( R = exception(K),
        % ... unwrap exception here.
      )
    )
.
```

`K` thus contains whatever values our goal have thrown. Mercury's standard libraries throw a dedicated type named `software_error`, which is a simple wrapper around a string:

```
        det_univ_to_type(K, KE:software_error),
        KE = software_error(KStr),
        io__write_string("exception! string: ", !IO),
        io__write_string(KStr, !IO) )
```

If what we have thrown are simpler things, e.g. a string, the handling would be simpler:

```
main(!IO) :-
    try((pred(Out:int::out) is det :- throw("this is errmsg")),
        R),
    ( ( R = succeeded(K), io__write_string("succeeded!\n", !IO) );
      ( R = exception(K),
        det_univ_to_type(K, KE:string),
        io__write_string("exception! string: " ++ KE ++ "\n", !IO)
      )
    )
.
```

`finally/6`

As mentioned above, `finally` is like the "finally" construct in other languages (unlike `try` which has `try_io`, `finally` only has one version, which supports threading in IO), but one might be surprised that `finally` rethrow the thrown exception *after* the cleanup goal has been successfully run without itself throwing an exception. To properly mimic the behaviour of "try ... catch ... finally", one should use `try` *inside* a call to `finally`.

Handling with `try` goals

To create anonymous predicates every time we want to `try` something isn't really that ergonomical. Good thing that Mercury provides a built-in `try`

[8]For how this works, see Chapter 4.

construct. Its grammar is as follows:

```
try Params Goal
then ThenClause
else ElseClause
catch Term1 -> CatchClause1
catch Term2 -> CatchClause2
...
catch_any CatchAnyVar -> CatchAnyClause
```

Seems awfully complicated, at least more so than other conventional languages. Let's break it down bit by bit:

- The `then` clause and the `else` clause are here to deal with cases where `Goal` does *not* throw an exception. If we only look at this part, it works similar to an if-then-else: if `Goal` succeeded, `ThenClause` is executed, or else `ElseClause` is executed. The `then` clause is mandatory.

  Variables bound inside `Goal` would be available inside `ThenClause`.

- The `Params` part may look similar to the parameters in trace goals considering the language reference do mention (and only mentioned) an `io` parameter, but in this case this `io` parameter works exactly the opposite way. In a trace goal, `io` *gives* you an IO state variable that you can use inside the trace goal; but in a try goal, it's *you* who have to pass in an IO state variable from the outside via `io` so that you can do IO inside `Goal`.

  – If we choose to thread in a state variable, we can use that state variable to perform IO within `Goal`, but `Goal` itself cannot fail (and thus the try goal is not allowed to have an `else` clause); if we choose not to use any state variable, we can't do IO within `Goal`, but it is allowed to fail (and subsequently get handled by `else` clause).

- If `Goal` throws an exception, it will get unified with `Term1`, `Term2`, etc., until the unification succeeds, in which case the corresponding `catch` clause is executed. `Terms` can be a lot of things like actual literals (which checks if the exception is exactly those values) and variables with type declarations (which checks if the exception is of those types).

- Finally, the `catch_any` is optional. Just like its name suggests, it's intended to be a "catch-all" clause. It binds whatever is thrown to a variable (`CatchAnyVar` does indeed have to be a variable) which is available within `CatchAnyClause`. If all the `catch` clauses of a try goal fail to unify with the exception and the goal does not have a catch_any clause, that exception is rethrown to the outside.

Here is an example modified from the one in the language reference:

```
:- pred p(string::out) is det.
p(Output) :-
    Output = "some string".

:- pred p_carefully(io.state::di, io.state::uo) is cc_multi.
p_carefully(!IO) :-
    (try [io(!IO)] (
         io.write_string("Calling p\n", !IO),
         p(Output)
    )
    then
        io.write_string("p returned: ", !IO),
        io.write(Output, !IO),
        io.nl(!IO)
    catch S ->
        io.write_string("p threw a string: ", !IO),
        io.write_string(S, !IO),
        io.nl(!IO)
    catch 42 ->
        io.write_string("p threw 42\n", !IO)
    catch N:int ->
        io.write_string("p threw other number\n", !IO)
    catch_any Other ->
        io.write_string("p threw something: ", !IO),
        io.write(Other, !IO),
        % Rethrow the value.
        throw(Other)
    ).
```

As the program above currently is, if we call `p_carefully`, it would print two lines of string: `Calling p` and `p returned: "some string"`. By throwing different things as exception in `p` we can change this behaviour. For example, changing `p`'s definition to this:

```
:- pred p(string::out) is det.
p(Output) :-
    Output = "some string",
    throw(42).
```

...would make the program print “p threw 42”.

One thing you might have noticed is the fact that the `catch S` clause seems to be capturing any kind of values; but the truth is the later use of `io.write_string` has restricted the type of `S` to `string`; this means that this particular `catch` clause would only catch strings. You can test it out by checking the difference of the output between the case where `p` throws a random string and the case where `p` throws a boolean value.

A try goal is `cc_multi` by default, regardless of what might get called inside `Goal`.

3.12 Advanced goals

Up till now we have seen the following forms of goals:

- Conjunction, written as `A, B`;
- Disjunction, written as `A; B`;
- Branching, written as `if Cond then A else B` or `(Cond -> A; B)`;
- Unification, written as `A = B`;
- Trace goals, written as `trace [Args] Goal`;

Fact is, there is much more than the one listed above. We will go through a few more of them in this section. Good thing is that Mercury provides a built-in try construct.

Other basic goals

- Success, written as `true`, is a goal that always succeed.
- Failure, written as `fail`, is a goal that always fail.
- Negation is written as `not Goal` or `\+ Goal`.
- Inequality, which is written as `A B`. This is equivalent to `not (A = B)`.
- Parallel conjunction, written as `A & B`, is a goal that executes both `A` and `B` in parallel. The actual behaviour depends on the backend (called "grade" in Mercury lingo), and in certain cases might act like normal conjunction. Both `A` and `B` must be either of determinism `det` or `cc_multi`.

Existential quantification

Within the same clause of the definition of a predicate/function, all variables share the same scope; for example, in the snippet below, variables `N`, `M` and `M1` all share the same scope; new variables within this clause are thus required to not share the same name:

```
fac(N, M) :-
    ( if ( N = 0 ) then ( M = 1 )
      else if ( N = 1 ) then ( M = 1 )
      else if ( N < 0 ) then ( M = 1 )
      else ( fac(N-1, M1), M = N * M1 )
    ).
```

Sometimes this might prove to be bothersome, especially in predicates that are more substantial. In this case, one might want to have separate scopes, where one can reuse certain variable names as if they have never been used. One can achieve this by using existential quantification, written as `some [Vars] Goal`; all variables declared in `[Vars]` can be used without fear within `Goal`. For example:

```
main(!IO) :-
    X = 3,
    some [X] (
        X = "blah\n",
        io.write_string(X, !IO)
    ),
    io.write_int(X, !IO).
```

This would compile without error (albeit the compiler would produce a warning complaining about `X` being reused) despite the variable `X` being bound to values of different types.

Technically all normal goals are implicitly existential quantified; making them explicit won't introduce any change in semantics and/or actual behaviour.

Mercury also support universal quantification goals, which are of the form `all [Vars] Goal` and are equivalent to `not (some [Vars] (not Goal))`.

Shorthands for certain compound goals

- Implication: `A => B`, which is equivalent to `not (A, not B)`.
- Reverse implication: `A <= B`, which is equivalent to `B => A`, which is in turn equivalent to `not (B, not A)`.
- Logical equivalence: `A <=> B`, which is equivalent to `(A => B), (B => A)`

3.13 Working with other people's libraries

CHAPTER 4

Advanced Concepts & Usage

4.1 Instantiatedness

As we have already seen in Chapter 2, 3 of the most common insts in Mercury describes the "bound-ness" of terms: `free` representing a free term, `bound` representing a bound term, and `ground` which represents a fully-bound term. This is not all there is to instantiatedness; in Mercury, you can also define custom insts to describe *states a term could have*. One common use of this semantics is to describe the determinism of a predicate/function parameter since type signatures does not contain determinism info. For example, the function `pred_to_bool` which converts a call to predicate to a boolean value has the following mode declaration; the first parameter is declared to have the type `(pred)` and the mode `(pred) is semidet`, which restricts the allowed range of values for the first parameter to predicate calls that are of determinism `semidet`:

```
:- func pred_to_bool((pred)::((pred) is semidet)) = (bool::out) is det
    .
```

...calling it with anything else, like a `multi`, would now cause a compile error:

```
:- pred my_pred(int::out) is multi.
my_pred(3).
my_pred(4).
my_pred(5).

main(!IO) :-
    X = pred_to_bool((my_pred(_))),
    % ...

% this outputs the following error:
```

```
% ex01.m:019: In clause for `main(di, uo)':
% ex01.m:019:   in argument 1 of call to function `bool.pred_to_bool
    '/1:
% ex01.m:019:   mode error: variable `V_8' has instantiatedness
% ex01.m:019:   `/* unique */ (pred is multi)',
% ex01.m:019:   expected instantiatedness was `(pred is semidet)'.
% ex01.m:019: In clause for predicate `main'/2:
% ex01.m:019:   warning: variable `X' occurs only once in this scope.
```

You can define such insts for algebraic datatypes as well. While doing that, you need to specify the datatype the inst is for by specifying `for type_name/arity`. For example, an inst for `maybe/1` which requires values to be a fully-bound `yes` can be defined as follow:

```
:- inst maybe_yes_ground for maybe/1
   ---> yes(ground).
```

A non-empty list can be defined as such:

```
:- inst non_empty_list for list/1
   ---> [ ground | ground ].
```

It can then be used to describe situations where the list must be non-empty, for example, in the mode declaration of the predicate `solutions/2` (in module `solutions`). This is a predicate that collects all the solutions of a specified goal into a list; and when that goal is of determinism `multi`, this list would be non-empty. The mode declaration of this situation can thus be as follows:

```
:- mode solutions(pred(out) is multi, out(non_empty_list)) is det.
:- mode solutions(pred(out) is nondet, out) is det.
```

The first mode declaration restricts the first parameter to those predicates that are of determinism multi; since the predicate call is multi, the call must have at least one solution, and for this reason we can provide more information to the second parameter and describe it as an non-empty list. We couldn't do the same for the second one, since nondet is the most general determinism possible; with a nondet predicate call, we can't say anything about the possible state of the list of solutions, since there can be no solutions, one solution or many solutions.

Higher-order insts

4.2 Modes

Higher-order modes

4.3 Multiple solutions

4.4 Definite Clause Grammar

4.5 Typeclasses

4.6 Existential Types

4.7 Purity System

CHAPTER 5

Parsing With Definite Clause Grammar

CHAPTER 6

Foreign Language Interface

CHAPTER 7

Debugging Mercury Programs