

Chapter 0

About this book

Last edited Apr 11 10:40

Disclaimer.

This is a work in progress.

What is this book for?

A basis for courses teaching or using Mercury. A tutorial for experienced programmers wishing to learn Mercury.

What version of Mercury is used in the book?

Any recent release-of-the-day (ROTD) compiler will do. The text as of today (10 April 2005) will work fine for Mercury release 0.12.

Who is it aimed at?

Students who have completed at least one year of a computer science degree, been exposed to a strongly, statically typed functional programming language (e.g., ML, Haskell, Miranda), and have some reasonable programming experience.

What is this book about?

This book is a tutorial on programming in Mercury. The book should provide the reader with enough information to become a competent Mercury programmer and knowledge sufficient to understand the Mercury Reference Manual, the Mercury Library Reference Manual and the Mercury Users' Guide. The book will describe key features of Mercury in detail, including the type system, the mode and determinism systems, and programming with nondeterminism.

What is this book not about?

This book will not teach you how to program in general, nor will it teach you a great deal of theory. It will not teach you how type inference works; it will (probably) not teach you about Herbrand universes; it will not include an extended treatise on IO in declarative programming languages; it will not (in the first place) cover black-belt Mercury programming techniques (e.g. dynamic type casts for non-ground types).

Where is the introduction?

I plan to write the introduction last. It will include a brief description of “what Mercury is all about” and an explanation of what the book is trying to do.

Where should I send comments?

Please subscribe to the Mercury users’ mailing list by sending an e-mail containing the word “subscribe” in the body (as opposed to the subject line) to `mercury-users-request@cs.mu.oz.au` (this is a very low bandwidth mailing list). Then you can send comments to `mercury-users@cs.mu.oz.au`. All feedback is very gratefully received, particularly with reference to missing topics.

Chapter 1

Mercury By Example

Last edited Feb 2 12:41

This chapter aims to convey through examples a basic feeling for how Mercury works. Because this is the first chapter, we may gloss over some fine detail and allow ourselves a certain latitude in precision, but this will not worry us as we will correct these deficiencies in later chapters. The approach taken here is to start by presenting the “obvious” solution to a problem and then introduce features of Mercury that allow for more elegant or efficient programs — although we are not necessarily trying to write the most efficient programs possible at this point! **XXX** *Is this last point even necessary?*

1.1 Hello, World!

It is slightly unfortunate that the “Hello, World!” program introduces no less than three advanced Mercury concepts, but since tradition dictates that tutorial texts start with “Hello, World!” we’ll just have to jump straight in with the knowledge that things will get easier thereafter.

We’ll start by presenting the complete program which we’ll assume we’ve typed into a file called `hello.m`:

```

:- module hello.
:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is det.
:- implementation.
main(IOState_in, IOState_out) :-
    io.write_string("Hello, World!\n", IOState_in, IOState_out).

```

Let's go through this line by line.

```

:- module hello.

```

Every module must start with a declaration like this giving the name of the module; the Mercury compiler will expect a module called `hello` in a file called `hello.m`.

```

:- interface.

```

An **interface** declaration introduces the part of module describing the things we are going to export (i.e., allow people to see).

```

:- import_module io.

```

This **import_module** declaration says that we need to use some of the names exported by the `io` module, which is part of the standard Mercury library.

```

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

```

This says that we are going to define a *predicate* called `main` with two arguments of type `io` (which happens to be defined in the `io` module), the first of which is a *destructive input* argument, and the second of which is a *unique output* argument; the **is det** part tells us that `main` is a *deterministic* predicate — that is, `main` always succeeds and will always compute the same output given the same input (we will see later that in Mercury some predicates can fail and some can have more than one solution).

Every Mercury program needs to export a predicate called `main`, which the compiler takes as the starting point for the Mercury program as a whole.

```

:- implementation.

```

Everything after an **implementation** declaration is considered private implementation detail not visible to other users of the module.

```

main(IOState_in, IOState_out) :-
    io.write_string("Hello, World!\n", IOState_in, IOState_out).

```

Finally, we have a *clause* defining `main`. A clause comprises a *head* and *body* separated by a `:-` symbol. The *head* of this clause tells us that this is a definition for `main` and names its two arguments `IOState_in` and `IOState_out`. The body of the clause, which is executed when `main` is called, consists of a single *goal* calling `io.write_string` (i.e., the `write_string` predicate defined in the `io` module) with a message (Mercury interprets the `\n` sequence in the string argument as a literal newline character) and the two `IOState` arguments.

Program variables in Mercury always start with a capital letter or underscore, while names of predicates, types, modules and so forth do not.

We can compile and run `hello.m` as follows (`$` indicates the command line prompt):

```
$ mmc --make hello
Making Mercury/int3s/hello.int3
Making Mercury/cs/hello.c
Making Mercury/os/hello.o
Making hello
$ ./hello
Hello, World!
```

Et voila! (By default, `mmc --make` will construct a local Mercury directory, if necessary, to hold intermediate files generated during compilation.) **XXX** *Check for consistency throughout whether we're using a Mercury subdirectory or not when compiling.*

At this point the reader is probably wondering about the meaning of the `io` type arguments `IOState_in` and `IOState_out`, and the strange `di` and `uo` *argument modes*. The short answer is that every predicate that performs I/O has to have an `io` type input argument describing the state of the world at the time the predicate is called and an `io` type output argument describing the state of the world after the call. This is how Mercury allows programs to communicate with the outside world without compromising its mathematical integrity.

Because it doesn't make much sense to try reusing an old `io` state — once the print job has finished, you can't get the ink back off the paper and into the toner cartridge again — `io` states are *unique*. This is where the `di` and `uo` argument modes come in: they are just like the ordinary `in` and `out` argument modes we'll meet in the next section, except that they also specify uniqueness. The Mercury compiler will not allow programs to copy unique values or reuse dead ones, which means that programs that do I/O are guaranteed to behave in the way we expect them to.

So what if we want to do more than one I/O operation? In this case we have to give names to each of the intermediate `io` states:

```
main(IOState_in, IOState_out) :-
    io.write_string("Hello, ", IOState_in, IOState_tmp1),
    io.write_string("World!", IOState_tmp1, IOState_tmp2),
    io.nl(IOState_tmp2, IOState_out).
```

The first call to `io.write_string` takes `IOState_in` as an input, destroys it in the process of writing its string argument, and produces `IOState_tmp1` as its result. Then the second call to `io.write_string` destroys `IOState_tmp1` and produces `IOState_tmp2`. Finally, `io.nl` (which just writes out a newline), destroys `IOState_tmp2` and returns `IOState_out`, which is the result of the call to `main`.

Naming all these intermediate states quickly becomes tedious, so Mercury provides us with syntactic sugar in the form of *state variables*:

```
main(!IO) :-
    io.write_string("Hello, ", !IO),
    io.write_string("World!", !IO),
    io.nl(!IO).
```

This code is transformed by the compiler into something equivalent to the preceding example: each occurrence of the `!IO` state variable actually stands for two normal variables, which are given intermediate names in the “obvious” way (the full details of the state variable transformation can be found in the Mercury Reference Manual). Note that there is no special significance in the name `IO`, we merely use `IO` by convention to stand for `io` state arguments.

Points to remember

- A module starts with a **module** declaration and is followed by an **interface** section and an **implementation** section.
- The interface section declares the things that are exported by the module.
- All declarations start with a `:-` symbol.
- Declarations and clauses always end with a full stop.
- We have to import a module before we can use things exported by that module.

- Every Mercury program must export a predicate called `main`.
- Clauses, which define predicates, go in the implementation section.
- Variable names start with a capital letter; names of modules, types, predicates and so forth start with a lower-case letter or use symbols (e.g., the `int` module uses `+` for addition).
- Every predicate that performs I/O must have an `io::di` argument and an `io::uo` argument and must be deterministic.
- We use a state variable to avoid having to name every `io` state.

1.2 The Fibonacci numbers

Another great tradition of computer science tutorial texts is to define a function to calculate numbers in the Fibonacci series. The Fibonacci series starts 1 1 2 3 5 8 13 21 34... Each number after the first two is computed as the sum of the preceding two numbers.

As before, we'll start by showing a complete program and then look at the interesting parts in more detail.

```

:- module fib.
:- interface.
:- import_module io.

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

:- implementation.
:- import_module int.

:- pred fib(int::in, int::out) is det.

fib(N, X) :-
  ( if   N =< 2
    then X = 1
    else fib(N - 1, A), fib(N - 2, B), X = A + B
  ).

main(!IO) :-
  fib(17, X),
  io.write_string("fib(17, ", !IO),
  io.write_int(X, !IO),
  io.write_string(")\n", !IO).

```

Building and running `fib.m`, we find that...

```

$ mmc --make fib
Making Mercury/int3s/fib.int3
Making Mercury/cs/fib.c
Making Mercury/os/fib.o
Making fib
$ ./fib
fib(17, 1597)

```

The first thing to note is the **import_module** declaration at the start of the implementation section. We need to import the `int` module from the Mercury standard library because it defines all the operations on integers, such as addition and comparison. We import it in the implementation section rather

than the interface section because that's the only place we refer to names defined in the `int` module.

Next the declaration

```
:- pred fib(int::in, int::out) is det.
```

says that we are going to define a predicate `fib` taking two `int` arguments, an input and an output, which always succeeds and always computes the same output given the same input.

```
fib(N, X) :-
  ( if   N =< 2
    then X = 1
    else fib(N - 1, A), fib(N - 2, B), X = A + B
  ).
```

The body of the `fib` definition uses an **if-then-else** goal to decide what to do; the **else** part is not optional; and the whole thing appears in parentheses. The condition `N =< 2` succeeds if `N` is less than or equal to 2 and *fails* otherwise (we'll learn more about **semidet** predicates like `=<` in later examples). If `N =< 2` then the *unification* `X = 1` is executed. Otherwise `fib` is called twice to compute the preceding two Fibonacci numbers in `A` and `B`, and `X` is unified with their sum.

Finally we have

```
main(!IO) :-
  fib(17, X),
  io.write_string("fib(17, ", !IO),
  io.write_int(X, !IO),
  io.write_string(")\n", !IO).
```

which calls `fib(17, X)`, unifying `X` with the result of computing the 17th Fibonacci number, then writes out the answer.

Now, just as `N - 1` computes `N` minus 1 and `A + B` computes the sum of `A` and `B`, it is possible to define `fib` so that `fib(N)` computes the `N`th Fibonacci number:

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

fib(N) = X :-
  ( if   N =< 2
    then X = 1
    else X = fib(N - 1) + fib(N - 2)
  ).
```

The **func** declaration introduces `fib` as a *function* with an `int` argument computing an `int` result. Mercury assumes that the input arguments to a function have mode `in` and the result has mode `out` and that the function as a whole is **det**. Functions calls appear in argument expressions to unifications, predicate calls and other function calls; predicate calls, on the other hand, never appear in expressions.

Having redefined `fib` as a function, we also have to change how it is called in `main`:

```
main(!IO) :-
    io.write_string("fib(17) = ", !IO),
    io.write_int(fib(17), !IO),
    io.nl(!IO).
```

To finish, here is one last refinement we might like to make to our definition of `fib`:

```
fib(N) = ( if N =< 2 then 1 else fib(N - 1) + fib(N - 2) ).
```

By using an **if-then-else** *expression* we can move the entire body into the head of the clause. Since the clause now has an empty body, Mercury requires that we omit the `:-` part.

It is worth noting that all three of our definitions are computationally identical and there should be no difference in the code generated by the Mercury compiler. Whether to use a predicate or function is largely a matter of taste; a good rule of thumb is that if a predicate has a single output argument, and is deterministic, then it is probably better expressed as a function.

Points to remember

- Import modules in the implementation section if they are only referred to in the implementation section.
- Non-unique arguments (i.e., non-io arguments) should use the `in` and `out` argument modes.
- Using functions can cut down on the number of intermediate results you need to name.
- **if-then-else** can be used as a goal and as an expression; it should appear in parentheses and the **else** part is mandatory.
- If a clause of a predicate or a function has an empty body (because the computation is described in the head), then the `:-` must be omitted.

1.3 Simple input

We can extend our Fibonacci program to read in `N` rather than have it hard-coded by changing the definition of `main`:

```

:- import_module list, string.

main(!IO) :-
  io.read_line_as_string(Result, !IO),
  ( if
    Result = ok(String),
    string.to_int(string.strip(String), N)
  then
    io.format("fib(%d) = %d\n", [i(N), i(fib(N))], !IO)
  else
    io.format("That's not a number...\n", [], !IO)
  ).

```

The `list` and `string` standard library modules are imported because we will need them in the definition of `main`.

`main` starts by calling `io.read_line_as_string`, which reads a whole line of input up to and including the next newline character. If all goes well then `Result` ends up unified to a value `ok(String)`, where `String` is the string of characters read in and `ok` is called a *data constructor*. Other possibilities for `Result` are `eof`, indicating the end-of-file has been reached, and `error(ErrorCode)`, indicating that something went wrong.

Then an **if–then–else** decides what to do. The **if** condition succeeds if `Result` is an `ok` value (unifying `String` with the argument) and if `string.to_int(string.strip(String), N)` succeeds. The `string.strip` function returns its argument minus any leading and trailing whitespace, including the terminating newline character, while the predicate `string.to_int` succeeds if its first argument is a string of decimal digits (unifying `N` with the corresponding number), and fails otherwise.

It's worth taking a slightly closer look at the unification in the **if** part of the **if–then–else** goal:

```
Result = ok(String)
```

We know that `Result` has a value at this point, so this kind of unification is known as a *deconstruction*: it only succeeds if the value in `Result` matches the pattern `ok(something)`, in which case it unifies `String` with the *something*.

The **then** and **else** arms of the **if–then–else** goal call the `io.format` predicate, which is rather like C's `printf` function. The first argument is a format string (where `%d` indicates a decimal integer, `%f` indicates a floating point value, `%s` indicates a string, and `%c` indicates a character) and the second argument is a list of the corresponding values in [brackets], tagged with `i`, `f`, `s` or `c` for `int`, `float`, `string` or `char` values respectively. Thus in

```
io.format("fib(%d) = %d\n", [i(N), i(fib(N))], !IO)
```

the `int N` is printed instead of the first `%d` in the format string and the `int` result of `fib(N)` is printed instead of the second `%d`. If `N = 17` we'd expect this goal to output

```
fib(17) = 1597
```

Just as the Mercury compiler doesn't know how to do anything with integers unless you import the `int` module, it doesn't know anything about lists unless you import the `list` module. So if you want to use `io.format` then you have to import the `list` and `string` modules.

Okay, so far so good. What if we want to input more than one number? That's easy to arrange:

```
main(!IO) :-
  io.read_line_as_string(Result, !IO),
  ( if
    Result = ok(String),
    string.to_int(string.strip(String), N)
  then
    io.format("fib(%d) = %d\n", [i(N), i(fib(N))], !IO),
    main(!IO)
  else
    io.format("I didn't expect that...\n", [], !IO)
  ).
```

Now the **then** branch calls `main` recursively to read in another number. Being a declarative language, recursion is Mercury's only looping construct. However, as with any decent declarative language compiler worth its salt, tail recursion like this is just as efficient as a `while` or `for` loop in any other language.

Firing up the compiler we get

```

$ mmc --make fib
Making Mercury/int3s/fib.int3
Making Mercury/cs/fib.c
Making Mercury/os/fib.o
Making fib
$ ./fib
| 10
fib(10) = 55
| 17
fib(17) = 1597
| 20
fib(20) = 6765
| ^D
I didn't expect that...

```

(The `|`s indicate input from the user and don't actually appear on the screen. `^D` indicates the user typing `Ctrl+D` to close the input stream; Windows users should use `^Z`.)

It would be good to handle the end-of-file condition more gracefully. The obvious way to do that is to add another case to the **if-then-else**:

```

main(!IO) :-
  io.read_line_as_string(Result, !IO),
  ( if
    Result = eof
  then
    io.format("bye bye...\n", [], !IO)
  else if
    Result = ok(String),
    string.to_int(string.strip(String), N)
  then
    io.format("fib(%d) = %d\n", [i(N), i(fib(N))], !IO),
    main(!IO)
  else
    io.format("I didn't expect that...\n", [], !IO)
  ).

```

Observe the deconstruction test of `Result` with `eof` — the `eof` data constructor has no argument list and indeed it is a syntax error to write `eof()` as one might in other languages.

Before we leave our `fib` example, let us introduce Mercury's *switch* goals. A switch goal is rather like C's `switch` statement and consists of a set of alternatives testing a given variable against different possible values it might have. Here is `main` rewritten to use a switch goal:

```

main(!IO) :-
  io.read_line_as_string(Result, !IO),
  (
    Result = eof,
    io.format("bye bye...\n", [], !IO)
  ;
    Result = ok(String),
    ( if string.toInt(string.strip(String), N)
      then io.format("fib(%d) = %d\n", [i(N), i(fib(N))], !IO)
      else io.format("that isn't a number\n", [], !IO)
    ),
    main(!IO)
  ;
    Result = error(ErrorCode),
    io.format("%s\n", [s(io.error_message(ErrorCode))], !IO)
  ).

```

A sequence of goals separated by semicolons is called a *disjunction* (the semicolon is usually pronounced “or”). If each *disjunct* deconstructs a particular variable against a set of mutually exclusive possibilities, then it is called a switch. In general it is good style to use a switch rather than a sequence of **if–then–else** goals since then, in most cases, the Mercury compiler will tell you if you’ve forgotten a possibility or counted the same possibility twice.

Points to remember

- Some types (such as the result type of `io.read_line_as_string`) use different data constructors for different values. These values can be tested using deconstruction unifications.
- A data constructor with no arguments, such as `eof`, is not followed by an argument list.
- `io.format` is Mercury’s version of C’s `printf`. To use it you must import `list` and `string` as well as the `io` module.
- You should omit the parentheses around an **if–then–else** that immediately follows the **else** part of another **if–then–else**.
- A disjunction is a sequence of goals separated by semicolons.
- A switch is a disjunction where each disjunct tests a particular variable against a different possibility. Where applicable, switches are generally preferable to **if–then–elses**.

1.4 rot13

Let's move on to a different example. This time we are going to implement the `rot13` “encryption” algorithm, which works by rotating the Roman alphabet by 13 places — in other words, `abcdefghijklmnopqrstuvwxyz` in the input becomes `nopqrstuvwxyzabcdefghijklm` in the output. Decryption is simple: just use `rot13` a second time! While `rot13` has the cryptographic strength of damp tissue paper, it is sometimes useful for obscuring information in an e-mail that the recipient may not yet wish to know, such as who won the Grand Final the day before.

Here's a first cut at a solution:

```

:- module rot13.
:- interface.
:- import_module io.

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

:- implementation.
:- import_module char, list, string.

main(!IO) :-
    io.read_char(Result, !IO),
    (
        Result = ok(Char),
        io.write_char(rot13(Char), !IO),
        main(!IO)
    ;
        Result = eof
    ;
        Result = error(ErrorCode),
        io.format("%s\n", [s(io.error_message(ErrorCode))], !IO)
    ).

:- func rot13(char) = char.

rot13(Char) = ( if    Char = 'a' then 'n'
                else if Char = 'b' then 'o'
                ...
                else if Char = 'z' then 'm'
                else if Char = 'A' then 'N'
                else if Char = 'B' then 'O'
                ...
                else if Char = 'Z' then 'M'
                else   Char
            ).

```

While this plainly works:

```
$ mmc --make rot13
Making Mercury/int3s/rot13.int3
Making Mercury/cs/rot13.c
Making Mercury/os/rot13.o
Making rot13
$ ./rot13
| Port Adelaide beat the Brisbane Lions 113 to 73 in the Grand Final.
Cbeg Nqrynvqr orng gur Oevfonar Yvbaf 113 gb 73 va gur Tenaq Svany.
| Cbeg Nqrynvqr orng gur Oevfonar Yvbaf 113 gb 73 va gur Tenaq Svany.
Port Adelaide beat the Brisbane Lions 113 to 73 in the Grand Final.
```

it's hardly going to win prizes for elegance or efficiency. A more experienced Mercury programmer might code `rot13` like this:

```
:- func rot13(char) = char.

rot13(CharIn) = ( if rot13b(CharIn, CharOut) then CharOut else CharIn ).

:- pred rot13b(char::in, char::out) is semidet.

rot13b('a', 'n').
rot13b('b', 'o').
...
rot13b('z', 'm').
rot13b('A', 'N').
rot13b('B', 'O').
...
rot13b('Z', 'M').
```

There are three new things here: the **semidet** *determinism category*, clauses with the arguments already “filled in”, and using more than one clause to define a predicate.

First off, the **semidet** determinism category means that `rot13b` will, for any given input, either fail or compute a single answer. Looking at the code we might guess (correctly) that `rot13b('z', X)` should succeed unifying `X = 'm'` (and never anything else), while `rot13b('7', X)` would fail.

Secondly, a clause like

```
rot13b('a', 'n').
```

is just syntactic sugar for

```
rot13b(V1, V2) :- V1 = 'a', V2 = 'n'.
```

Since we know from the **pred** declaration for **rot13b** that V_1 is an input and V_2 an output, the unification $V_1 = 'a'$ must be a deconstruction test and, if that succeeds, then the *construction* unification $V_2 = 'n'$ is carried out (a construction unification always succeeds because the “destination” variable, V_2 in this case, does not have a value before this point).

Finally, a sequence of clauses is syntactic sugar for a single clause whose body is a disjunction. Hence our code is transformed by the compiler into this:

```
rot13b(V1, V2) :- ( V1 = 'a', V2 = 'n'
                    ; V1 = 'b', V2 = 'o'
                    ...
                    ; V1 = 'z', V2 = 'm'
                    ; V1 = 'A', V2 = 'N'
                    ; V1 = 'B', V2 = 'O'
                    ...
                    ; V1 = 'Z', V2 = 'M' ).
```

The astute reader will immediately identify this as a switch on V_1 because each disjunct tests the input V_1 for a different possible value. One of the good things about switches is that the Mercury compiler will generate very efficient code for them, using a lookup-table or hash-table perhaps, which will certainly out-perform the long chain of **if–then–elses** in our first attempt.

(As an aside, the **is semidet** determinism declaration for **rot13b** tells the Mercury compiler that this predicate is expected to fail in some cases, so it will not warn us about missing possible values for the first argument. In the **fib** program the switch had to be exhaustive, so it could not fail, because **main** was declared to be **det**. Had we missed a possible case out of that switch, the compiler would have reported the missing case as an error. Similarly, because **rot13b** cannot have more than one solution for any input, the compiler will report an error if we have duplicate clauses matching the same input. The compiler won't warn us about such problems if we use chains of **if–then–elses**.)

Note that if we felt so inclined, we could make **rot13b** deterministic by including the translation of every possible character! Of course, there are many ways of coding **rot13** and while our implementation may not be the most concise, it is quite efficient and very easy to understand. Either way, our aim here was to look more closely at the concept of semideterminism and introduce the technique of making code more readable by using multiple clauses to define a predicate or function.

Points to remember

- Literal character values in Mercury are normally enclosed in single quotes. Some characters which the Mercury parser would normally expect to see used as infix function symbols, such as + and *, also need to be enclosed in parentheses: ('+') and ('*'). Full details of how special characters should be written can be found in the the Mercury Reference Manual.
- The **semidet** determinism category means that a predicate can have at most one *solution* for a given set of inputs (if it has no solution for the given inputs then it fails).
- Semideterministic predicates therefore often appear in the conditions of **if–then–else** goals.
- “Filling in” the arguments of a clause is just shorthand for omitting the equivalent unification goals.
- The compiler will view a predicate or function definition comprising several clauses as a disjunction. Multiple clauses are often easier to read than the equivalent disjunction.

1.5 Cryptarithms

In this example we try to demonstrate a little of what differentiates Mercury from conventional programming languages. A cryptarithm is an equation (usually just an addition) where the digits of each number have been replaced by letters, $DOG + ANT = CAT$ for example; a solution is a mapping from letters to digits that satisfies the equation. Leading letters cannot stand for zero and each letter must stand for a distinct digit. Here's a Mercury program to solve this particular cryptarithm:

```

:- module crypt.
:- interface.
:- import_module io.

:- pred main(io::di, io::uo) is cc_multi.

:- implementation.
:- import_module int, list, string.

main(!IO) :-
  io.format("DOG + ANT = CAT\n", [], !IO),
  ( if
    Ds0 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],    C0 = 0,
    pick( Ds0, G, Ds1),
    pick( Ds1, T, Ds2),
    T = (G + T + C0) mod 10,          C1 = (G + T + C0) / 10,
    pick( Ds2, O, Ds3),
    pick( Ds3, N, Ds4),
    A = (O + N + C1) mod 10, A \= 0, C2 = (O + N + C1) / 10,
    pick( Ds4, D, Ds5),
    pick( Ds5, A, Ds6),
    C = (D + A + C2) mod 10, D \= 0, C = (D + A + C2) / 10,
    pick( Ds6, C, -)
  then
    DOG = 100 * D + 10 * O + G,
    ANT = 100 * A + 10 * N + T,
    CAT = 100 * C + 10 * A + T,
    io.format("%d + %d = %d\n", [i(DOG), i(ANT), i(CAT)], !IO)
  else
    io.format("has no solutions\n", [], !IO)
  ).

:- pred pick(list(int)::in, int::out, list(int)::out) is nondet.

pick([X | Xs], X, Xs).
pick([X | Xs], Y, [X | Zs]) :- pick(Xs, Y, Zs).

```

At this early stage it would be wrong to try to explain exactly *how* this

program works. Instead, we will describe *what* it does and offer hints as to how it does it, but leave the technical explanation to later chapters.

The interesting part of this program is the condition of the **if–then–else** goal. The trick to understanding this code fragment is to consider it as a set of constraints on a solution rather than as a series of computations. (Concentrating on the “what” rather than the “how” is a hallmark of declarative programming.)

Ds_0 is the list of digits from 0 to 9, Ds_1 is Ds_0 with the digit for G removed, Ds_2 is Ds_1 with the digit for T removed, Ds_3 is Ds_2 with the digit for O removed, and so forth. In this way we ensure that each of D, O, G, A, N, T and C are different digits.

The digits for each letter are selected nondeterministically using the `pick` predicate. The goal `pick(Ds0, G, Ds1)`, for instance, picks a digit for G from Ds_0 and leaves the remainder in Ds_1 . Since Ds_0 contains ten members, there are ten possible solutions for G.

C_1 is the carry left over from the units column, C_2 is the carry left over from the tens column, and we introduce C_0 as a “carry in” of zero just to keep the structure of the program regular.

We verify that the units column works out with the goal $T = (G + T + C_0) \bmod 10$. Similarly the goal $A = (O + N + C_1) \bmod 10$ verifies the tens column is consistent and $C = (D + A + C_2) \bmod 10$ verifies the hundreds column works.

The goals $A \neq 0$ and $D \neq 0$ ensure that we don’t have any zeroes in the hundreds column, while $0 = (D + A + C_2) / 10$ ensures that there is no carry left over from the sum as a whole (`/`, the division function on ints, truncates the result towards zero).

Informally, Mercury tries to find a solution to the condition goal as a whole by considering each subgoal in turn. The `pick` goals are slightly special because they can have more than one possible solution; such goals are referred to as *choice points*. When the Mercury program executes a `pick` goal, it considers each possibility in turn until it finds one that works. It does this by *backtracking* whenever it encounters a goal that fails. Backtracking unwinds the computation to the last choice point (i.e., the most recent `pick` goal), undoing any unifications that have since occurred, and tries another alternative. If all of the possibilities for a given `pick` goal lead to failure, then backtracking continues to the preceding `pick` goal, and so on.

Eventually either a set of solutions to the `pick` goals will be found that is consistent with the other constraints or, if no such set exists, then this part of the program will fail.

At this point we can explain the **cc_multi** determinism category for `main`, which is different to **det** which we've used in all the preceding examples. **cc_multi** stands for *committed choice multideterministic*, which means that although this predicate may have multiple possible answers (there may be multiple solutions to `main`'s **if-then-else** condition) we will only be getting one of them! This extra constraint is necessary to allow a predicate with more than one possible solution to perform I/O; since we're not allowed to backtrack over predicates that do I/O, we have to state that we're content for `main` to stick to the first solution it sees, whichever one that happens to be.

As with `main`, we won't explain here *how* `pick` works suffice to give plain English descriptions of the clauses:

```
pick([X | Xs], X, Xs).
```

This clause says that given a list whose *head* is the item `X` and whose *tail* is the list of items `Xs`, we can remove `X` from the input leaving `Xs` as the remainder.

```
pick([X | Xs], Y, [X | Zs]) :- pick(Xs, Y, Zs).
```

This clause says that given a list with head `X` and tail `Xs`, we remove an item `Y` from `Xs` leaving remainder `Zs` (via the recursive call to `pick`), and the remainder of the original input then is the list whose head is `X` and whose tail is `Zs`.

Don't worry if much of this seems confusing: once some facility with thinking declaratively (i.e., "what" rather than "how") is acquired, one will be able to look at predicates like `pick` and complicated goals like the condition of `main`'s **if-then-else** and immediately understand what is meant. The thing to carry away from this example is the ease with which Mercury allows us to describe a solution to a complex search problem. We have done very little more than write a logical specification of the problem using Mercury syntax, all the tedious operational details are handled for us by the compiler! Elegance of this sort is one of the key things that makes Mercury such an attractive programming language.

Anyway, after all that hard work, let's just prove that all this magic actually works:

```
$ mmc --make crypt
Making Mercury/int3s/crypt.int3
Making Mercury/cs/crypt.c
Making Mercury/os/crypt.o
Making crypt
$ ./crypt
DOG + ANT = CAT
420 + 531 = 951
```

Success!

Points to remember

- Nondeterministic predicates such as `pick` can have multiple solutions for a given set of inputs. Mercury handles this by backtracking to the most recent choice point when a later goal fails.
- As far as possible, try to think declaratively rather than operationally: focus on what it is that is being computed rather than how the computation should proceed. Leave as much of that sort of detail to the compiler as you can.
- Unification goals are quite versatile: they can denote constructions, deconstructions, and equality tests.

Chapter 2

The Mercury type system

Last edited Jan 18 17:56

In this chapter we describe the primitive (i.e., built-in) Mercury types, and how to define and use new types. The style of this chapter is a little dry, so we advise the reader to skim through it the first time around, and then dip back into it for reference as occasion demands.

Mercury uses an expressive, statically checked type system similar to that of ML and Haskell. The type system is expressive in the sense that the compiler can always infer the type held in a particular variable; apart from one exception, which is described below, type casts of the kind found in Java and C programs are unnecessary. Static type checking means that any possible type violation in a program is detected at compile-time rather than at run-time. Many common programming errors are, therefore, simply inexpressible in Mercury. A handy secondary benefit of this approach to typing is that the compiler can generate highly optimized code.

Mercury also supports dynamic typing and type reflection for the rare cases where such things are necessary. Dynamic typing is supported by a universal type that can conceal values of any type at all; however, extracting a concealed value does require a checked run-time type cast operation. Type reflection allows a program to examine the structure of values and their types. By way of example, a generic function to compute hash codes for values of any type depends upon type reflection.

Important!

Mercury is a purely declarative language. This means that values are immutable: there is no destructive assignment. Every computation constructs a new value for each output. The compiler may well generate code that uses destructive assignment (e.g., for efficient array updates), but such things are not directly available to the Mercury programmer.

2.1 The primitive types: int, float, string, and char

Note! The underlying representation of the primitive types depends upon the compiler target (C, Java, .Net etc.) For the C targets `int` corresponds to `int`, `float` to `double`, `char` to `char` and `string` to `char *`; number ranges, representations, arithmetic overflow handling and so forth are dictated by the target machine.

int

Fixed precision integers are represented by the `int` type. Syntactically, an `int` is a sequence of digits, optionally preceded by a minus sign (there is no unary plus). The sequence of digits may be decimal, hexadecimal (if preceded by `0x`), octal (if preceded by `0o`), or binary (if preceded by `0b`).

Examples: decimal `-123`, `0`, `42`; hexadecimal `-0x7B`, `0x0`, `0x2A`; octal `-0o173`, `0o0`, `0o52`; binary `-0b1111011`, `0b0`, `0b101010`.

The sequence `0'x` denotes the character code for the character `x`. For example, on an ASCII system `0'a`, `0'b`, and `0'c` denote 97, 98, and 99 respectively.

The `int` standard library module must be imported to use any of the primitive `int` operations.

float

Floating point numbers are represented by the `float` type, which corresponds to the C `double` type. Syntactically, a `float` is a decimal floating point number (the decimal point is required), optionally preceded by a minus sign, optionally followed by an integer exponent.

These are all equivalent: 1.414, 1414e−3, .1414e1, 0.01414e2; either e or E is acceptable as the exponent separator.

The `float` standard library module must be imported to use any of the primitive float operations. Constants such as `pi` and `e` and more complex floating point operations, such as the trigonometric functions, are defined in the `math` standard library module.

string

A string constant is a sequence of characters enclosed in double quotes.

Examples: `" "`, `"Hello, World!\n"`, `"\“Lawks!\” I declared."`.

Certain characters have special syntax referred known as *character escapes*:

<code>\"</code>	double quote	<code>\\</code>	backslash
<code>\'</code>	single quote	<code>\a</code>	alert (“beep”)
<code>\b</code>	backspace	<code>\r</code>	carriage return
<code>\f</code>	form-feed	<code>\t</code>	tab
<code>\n</code>	newline	<code>\v</code>	vertical tab

Characters can also be specified by character code using the sequence `\xhh\`, where `hh` is a hexadecimal number, or `\ooo\`, where `ooo` is an octal number. the ASCII character `A`, for example, can also be written as `\x41\` or `\101\`.

Note! For arcane reasons, it is a Very Bad Idea to include the NUL character, `\x00\`, in strings.

A backslash at the end of a line is ignored in string constants. Thus

```
“abc\  
def”
```

is equivalent to just `“abcdef”`. Otherwise, literal newlines may appear in a string constant:

```
“pqr  
xyz”
```

is equivalent to `“pqr\nxyz”`.

The `string` standard library module must be imported to use any of the primitive string operations. In particular it defines `++`, the string concatenation function — for example, `“foo” ++ “bar” = “foobar”`.

char

Character constants are represented using the `char` type and, syntactically, are single characters (or character escapes or character codes as described above) enclosed in single quotes. Characters that could be interpreted as infix operators, such as `+` and `*`, should be further enclosed in parentheses.

Examples: `'A'`, `'\x41'`, `'\101'`, `'\"'`, `'\n'`, `('+')`, `('*')`.

XXX *Are we ASCII specific?*

The `char` standard library module must be imported to use any of the primitive operations on chars.

2.2 Tuples

A tuple is a fixed size vector of values. Syntactically, a tuple *type* is a comma separated sequence of type names enclosed in braces, whereas a tuple *value* is a comma separated sequence of values enclosed in braces.

Examples: `{111, 'b'}` is a value of type `{int, char}`; `{1.2, 3, "456"}` is a value of type `{float, int, string}`; `{"a", {"little", "contrived"}}` is a value of type `{string, {string, string}}`.

Note! Unlike lists (described below), tuples are constructed and deconstructed as complete entities. There is no such thing as the head or tail of a tuple. **XXX** *Should we include this note at all? Or move the whole tuples section after lists?*

Tuples are occasionally useful for aggregating small numbers of different types. More often than not it is better style to use a discriminated union type.

2.3 Lists

A list is a linear sequence of values sharing the same type. Syntactically, a list is a comma separated sequence of values enclosed in brackets.

Examples: `[]` denotes the empty list, regardless of the list type; `[1, 2, 3]` is a

value of type `list(int)`; `['a', 'b', 'c', 'd']` is a value of type `list(char)`; `[[1], [2, 3], [4]]` is a value of type `list(list(int))`.

It is an error to mix values of different types inside a list because there is no way to give a type to values such as `[1, "2", 3.4]`.

The `list` standard library module defines the `list` type and a wealth of list operations. If you forget to import the `list` module then the compiler will issue warnings like this:

```
foo.m:031: error: undefined symbol '[]/2'
foo.m:031: (the module 'list' has not been imported).
```

Lists are actually composed of two kinds of building block: `[]` (the empty list) and `[]` (the non-empty list constructor, pronounced “cons”). If `X` is an `int`, say, and `Xs` a `list(int)`, then `[](X, Xs)` is a `list(int)` whose *head* (first member) is `X` and whose *tail* (list of trailing members) is `Xs`.

Lists are so common that special syntactic sugar exists for them: `[X | Xs]` is shorthand for `[](X, Xs)`; `[X, Y, Z | Ws]` is shorthand for `[X | [Y | [Z | Ws]]]`; and `[X, Y, Z]` is shorthand for `[X, Y, Z | []]`.

2.4 Discriminated union types

Discriminated unions allow the definition of new, structured types. This example shows a representation of playing cards using discriminated union types:

```
:- type playing_card ----> card(rank, suit) ; joker.
:- type rank          ----> ace ; two ; three ; four
                          ;    five ; six ; seven ; eight
                          ;    nine ; ten ; jack  ; queen ; king.
:- type suit         ----> clubs ; diamonds ; hearts ; spades.
```

The *data constructors* defining the values of the discriminated union types appear to the right of the arrows: `suit` has four possible values, `rank` thirteen, and `playing_card` fifty three (fifty two possible card values from `card(ace, clubs)`, `card(two, clubs)`, all the way up to `card(king, spades)`, plus the `joker` option).

An exhaustive switch on a discriminated union type must test for every possible top-level data constructor. That is, an exhaustive switch on a

playing_card value need only test for card(., .) and joker rather than every specific card instance.

(The term “discriminated union” is used because a type denotes a union of sets of possible values, each of which is distinguished by its data constructor.)

Data constructors with named fields

The fields of a data constructor can be named:

```
:- type bank_account ---> account( name      :: string,
                                   account_no :: int,
                                   funds      :: float ).
```

We can use field names to access fields directly without having to first de-construct a bank_account value. That is, rather than writing

```
BankAcct = account(Name, AcctNo, Funds),
( if Funds >= RequestedSum then
  ... debit RequestedSum from BankAcct ...
else
  ... reject debit request ...
)
```

we can write

```
( if BankAcct^funds >= RequestedSum then
  ... debit RequestedSum from BankAcct ...
else
  ... reject debit request ...
)
```

The Mercury compiler automatically generates a *field access function* for every named field in a data constructor. The bank_account field access functions would be defined like this:

```
account(A, _, _)^name      = A.
account(_, B, _)^account_no = B.
account(_, _, C)^funds     = C.
```

Field access functions to “update” named fields are also generated:

```
( if BankAcct^funds >= RequestedSum then
  NewBankAcct = (BankAcct^funds := BankAcct^funds - RequestedSum)
else
  ... reject debit request ...
)
```

The expression `(BankAcct^funds := X)` returns a value identical to `BankAcct` except the `funds` field will contain `X`. The `bank_account` field access update functions would be defined like this:

```
( account(_, B, C)^name      := A ) = account(A, B, C).
( account(A, _, C)^account_no := B ) = account(A, B, C).
( account(A, B, _)^funds     := C ) = account(A, B, C).
```

Note that you do not have to name every field of a data constructor; unnamed fields can only be “read” or “updated” by explicitly deconstructing or constructing the entire data constructor value.

You are not allowed to use the same field name in different types defined in the same module. This is an error:

```
:- type cat ----> cat(name :: string).
:- type dog ----> dog(name :: string).
```

Instead use distinct field names, such as `cat_name` and `dog_name`, or use a single type with two data constructors (the same field name can be used in different data constructors of the same type.)

A field access may fail if a data type has more than one data constructor. For example, given

```
:- type playing_card ----> card(card_rank :: rank, card_suit :: suit) ; joker.
```

A goal featuring the expression `Card^card_rank` will fail if `Card` happens to be a `joker`.

Field accesses can be chained together.

```
:- type employee ----> employee(id :: int, contact :: contact_details).
:- type contact_details ----> contact_details(address :: string, phone :: int).
```

If `Employee` contains a value of type `employee` then the expression `Employee^contact^address` is the `address` field of the `contact` field of the `employee` constructor.

Nested fields can be updated. The expression `Employee^contact^address := NewAddr` denotes a copy of `Employee` with the `address` field of the `contact`

field of the `employee` data constructor updated to hold `NewAddr`.

Parentheses can change the meaning of an update expression:

`(Employee^contact)^address := NewAddr` denotes an updated copy of the `contact` field of the `employee` data constructor (i.e., the type of this expression is `contact_details`, not `employee`.)

One final remark: it is also possible to explicitly define field access functions, for instance for “virtual fields” that are computed rather than stored in a data constructor or for update functions that perform sanity checks on their arguments. User defined field access functions are described fully in Chapter **XXX**.

2.5 Polymorphic types

Polymorphic types are types parameterised by *type variables*. A polymorphic binary tree type carrying values at the branches could be defined like this:

```
:- type tree(T) ----> leaf ; branch(tree(T), T, tree(T)).
```

This is just a discriminated union type with a type argument, `T`. `T` can match any type at all, so `tree(int)`, `tree(string)`, `tree(list(char))` are all refinements of `tree(T)`.

Examples: `branch(branch(leaf, 1, leaf), 2, branch(leaf, 3, leaf))` is a value of type `tree(int)`; `branch(leaf, {'a', 65}, branch(leaf, {'b', 66}, leaf))` is a value of type `tree({char, int})`; and `leaf` is a value of every `tree` type.

The canonical example of a polymorphic data type is the list type defined in the list standard library module:

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

The `maybe` type defined in the `std_util` standard library module is another useful polymorphic type:

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

This type is commonly used to represent optional values (had the database community known about `maybe` types they never would have invented `NULLs` and wrecked the relational model...)

Once one has polymorphic types, it is natural to want to define polymorphic predicates and functions. The `list` module `length` function works for lists of every type thanks to the type argument in its signature (it is just convention that we have reused the name `T` here — any variable name would do):

```
:- func length(list(T)) = int.

length([])      = 0.
length([_ | Xs]) = 1 + length(Xs).
```

The first clause defines the length of the empty list to be 0; the second clause defines the length of a non-empty list `[_ | Xs]` to be 1 for the head (the underscore says we don't care what that happens to be) plus the length of the tail, `Xs`.

Here is an example of a polymorphic predicate to decide whether a given value resides in an ordered binary tree of the type we defined at the start of this section (by ordered we mean that smaller values appear to the left of larger values in the tree):

```
:- pred search(tree(T)::in, T::in) is semidet.

search(branch(L, X, R), Y) :-
  O = ordering(X, Y),
  ( O = (<), search(R, Y)
  ; O = (=)
  ; O = (>), search(L, Y)
  ).
```

The `ordering` function is built-in to Mercury and compares any two values of the same type, returning a result of type `comparison_result`:

```
:- type comparison_result ---> (<) ; (=) ; (>).
```

As you can see, data constructor names don't necessarily have to be alphanumeric. These data constructor names must appear in parentheses to stop the Mercury parser from interpreting them as infix operators. This is also an example of *overloading*, where the same name may be used for more than one purpose provided there is no ambiguity.

After calling `ordering`, `search` switches on `O` to decide what to do next: if the value at the current `branch`, `X`, is less than the value we are searching for, `Y`, then `search` should proceed down the right subtree, `R`. If `X = Y` then the search terminates successfully. Otherwise, `X` is greater than `Y` and the search should proceed down the left subtree, `L`. The absence of a clause for leaf values means that any search that reaches a leaf will fail.

2.6 Equivalence types

Readability is often improved by giving simple names to complex types or by using more meaningful names for a specific uses of general types:

```

:- type height == float. % In metres.
:- type radius == float. % In metres.
:- type volume == float. % In cubic metres.

:- func volume_of_cylinder(height, radius) = volume.
:- func volume_of_sphere(radius) = volume.

```

XXX *Move this stuff about comments to chapter 1.* (The % sign introduces a comment, which extends to the end of the line.) Here we define the types `height`, `radius` and `volume` to be equivalent to (i.e., interchangeable with) type `float`. We could have just declared `volume_of_cylinder` using

```

:- func volume_of_cylinder(float, float) = float.

```

but then we would be morally obliged to include a comment explaining which arguments correspond to which measurements.

Equivalence types can also be parameterized. For example:

```

:- type dictionary(Key, Value) == list({Key, Value}).

    % search(Dict, Key, Value) unifies Value if there is an association
    % for Key in Dict, but fails otherwise.
    %
:- pred search(dictionary(Key, Value)::in, Key::in, Value::out) is semidet.

search([{{K, V} | Dict], Key, Value) :-
    ( if Key = K then Value = V else search(Dict, Key, Value) ).

    % set(Dict, Key, Value) returns an updated version of Dict
    % associating Key with Value.
    %
:- func set(dictionary(Key, Value), Key, Value) = dictionary(Key, Value).

set(Dict, Key, Value) = [{{Key, Value} | Dict].

```

2.7 Abstract types

It is virtually always a Bad Idea to reveal implementation detail to the user of a module. Mercury ensures that predicate and function *definitions* are private to a module because they cannot appear in the **interface** section of a module. Abstract types allow the same kind of information hiding for types. An abstract type is one that is *declared* in the **interface** section of a module, but *defined* in the **implementation** section.

Here's how we would use abstract types if we wanted to export the dictionary type defined above:

```

:- module dictionary.
:- interface.

:- type dictionary(Key, Value).

:- pred search(dictionary(Key, Value)::in, Key::in, Value::out) is semidet.
:- func set(dictionary(Key, Value), Key, Value) = dictionary(Key, Value).

:- implementation.
:- import_module list.

:- type dictionary(Key, Value) == list({Key, Value}).

search([K, V] | Dict, Key, Value) :-
    ( if Key = K then Value = V else search(Dict, Key, Value) ).

set(Dict, Key, Value) = [Key, Value] | Dict.

```

Observe the **type** declaration in the **interface** section: it gives the name of the type and its arguments, but nothing else. Further down, in the **implementation** section, we give a definition for `dictionary`. At some later point we may reimplement `dictionary` as an ordered list or binary tree or some other more efficient structure. Such a change would not affect the interface of the dictionary module (because `dictionary` is an abstract type), so no changes would be required by users of the module.

2.8 Higher order types

Mercury considers functions and predicates to be values just as much as it does ints, strings and lists. Consider the higher order `map` function (as defined in the list standard library module) which takes a function from T_1

values to T_2 values, a list of T_1 values, and returns a list of T_2 values:

```
:- func map(func( $T_1$ ) =  $T_2$ , list( $T_1$ )) = list( $T_2$ ).

map(., []) = [].
map(F, [X | Xs]) = [F(X) | map(F, Xs)].
```

The first clause says that mapping over the empty list returns the empty list. The second clause says that mapping the function F over the non-empty list $[X | Xs]$ is the list whose head is $F(X)$ (i.e., the result of applying F to X) and whose tail is the result of mapping F over Xs .

The thing to observe here is the argument type $\mathbf{func}(T_1) = T_2$, which illustrates the syntax for function types.

This next example illustrates the syntax for predicate types (this predicate is also defined in the list standard library module):

```
:- pred filter(pred( $T$ ), list( $T$ ), list( $T$ ), list( $T$ )).
:- mode filter(in(pred(in) is semidet), in, out, out) is det.

filter(., [], [], []).
filter(P, [X | Xs], Ys, Zs) :-
    filter(P, Xs, Ys0, Zs0),
    ( if P(X) then Ys = [X | Ys0], Zs = Zs0
      else Ys = Ys0, Zs = [X | Zs0]
    ).
```

the goal $\mathbf{filter}(P, As, Bs, Cs)$ unifies Bs with the list of members of As that satisfy P and unifies Cs with the list of members of As that don't. The first clause says that filtering the empty list yeilds two empty lists. The second clause says that filtering $[X | Xs]$ through the predicate P is the result of filtering Xs through P and adding X to the first result if $P(X)$ (i.e., if P succeeds given X), or adding X to the second result if it doesn't.

The first new thing here is the separation of *type* information from *mode* information in the declarations for \mathbf{filter} . Mercury requires a separate **mode** declaration if you do not supply mode and determinism details in the **pred** declaration. The two declarations could be combined thus

```
:- pred filter( pred( $T$ ::in(pred(in) is semidet),
               list( $T$ ::in, list( $T$ ::out, list( $T$ ::out)) is det.
```

However, having separate **pred** and **mode** declarations highlights that the *type* of \mathbf{filter} 's higher order argument is written $\mathbf{pred}(T)$. (Separate **mode** declarations are examined in more detail in Chapter XXX.) XXX *Do I need*

to explain the parameterised in mode?

In general, higher order programming with predicates is more complicated than with functions because of the need to also specify the modes for the higher order arguments.

2.9 univ, the universal type

The `univ` type provides support for dynamically typed programming. Mercury's type system is so expressive that `univ` is hardly ever necessary. However, should you require it, here is (a slightly abridged version of) the interface to `univ` as defined in the `std_util` standard library module:

```
:- type univ.
:- func univ(T) = univ.
:- pred univ_to_type(univ::in, T::out) is semidet.
```

The `univ` function turns an argument of any type into a `univ` value (this is an example of overloading a name for a type and a function). The `univ_to_type` predicate turns a `univ` value into a value of type `T` (what `T` denotes depends upon the context of the call to `univ_to_type`) if that is the type of value contained in the `univ`, and fails otherwise.

We said earlier that a value like `[1, "2", 3.4]` would be rejected because it cannot be given a type. This is true, but we can achieve much the same end by writing `[univ(1), univ("2"), univ(3.4)]`, which does have a type, `list(univ)`.

To illustrate the use of `univ_to_type`, here is a program to print out univs:

```
:- module print_univs.
:- interface.
:- import_module io.

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

:- implementation.
:- import_module list, std_util, string.

main(!IO) :-
    print_univ(univ(1), !IO),
    print_univ(univ("2"), !IO),
    print_univ(univ(3.4), !IO),
    print_univ(univ({5, 6, 7}), !IO).
```

```
:- pred print_univ(univ::in, io::di, io::uo) is det.
```

```
print_univ(U, !IO) :-
  ( if   univ_to_type(U, C) then io.format("a char, %c\n",      [ c(C) ], !IO)
    else if univ_to_type(U, S) then io.format("a string, \"%s\" \n", [ s(S) ], !IO)
    else if univ_to_type(U, I) then io.format("an int, %d\n",      [ i(I) ], !IO)
    else if univ_to_type(U, F) then io.format("a float, %f\n",    [ f(F) ], !IO)
    else   io.format("no idea...\n", [], !IO)
  ).
```

Compiling and running this program, we get

```
> mmc --make print_univs
Making Mercury/int3s/print_univs.int3
Making Mercury/cs/print_univs.c
Making Mercury/os/print_univs.o
Making print_univs
> ./print_univs
an int, 1
a string, "2"
a float, 3.400000
no idea...
```

So how does `univ_to_type` know that `C` is a `char`, `S` a `string`, and so forth? The answer is the compiler automatically infers these types from context: the argument of a `c` data constructor in an `io.format` argument list must be a `char`; the argument of an `s` data constructor must be a `string`; similarly `I` must be an `int` and `F` a `float`. Information about each result type is supplied to the `univ_to_type` calls via extra arguments inserted by the Mercury compiler.

Full details of the run-time type information (RTTI) scheme are beyond the scope of this book. The interested reader is referred to the documentation for the `std_util` module in the Mercury Library Reference Manual and the relevant parts of the Mercury Reference Manual.

2.10 Useful types defined in the Mercury standard library

The Mercury standard library defines many useful types, the most common of which are examined in more detail in Chapter **XXX**. These include `bool` for boolean values, `graph` for graph processing, `list` for stacks and sequences, `map` for dictionaries, `queue` for first-in first-out (FIFO) queues, `pqueue` for priority queues, `random` for random numbers, and `set` for sets. The `std_util`

module defines miscellaneous utility types such as `univ`, `maybe`, `unit` (the “dummy” type), and `pair`. Many more types are defined in the standard library, but the above suffice for the majority of data structures.

2.11 More advanced types...

Mercury’s type system includes two more advanced aspects each with a separate chapter. Chapter **XXX** describes type classes and existentially quantified types which exist to support object oriented programming styles. Chapter **XXX** describes types with user-defined equality and comparison relations; these are so-called *non-canonical* types in which a given semantic value may be represented in more than one way.

XXX *I haven’t really mentioned type inference, nor have I mentioned explicit type qualification of local vars.*

Chapter 3

The Mercury mode system

Last edited Jun 7 16:38

Mercury programs are really just logical formulae written down using a particular syntax. The Mercury compiler, however, needs extra information to turn these logical formulae into something the computer can execute. Specifically, it needs to know which arguments of a predicate can be inputs and which outputs. This information is conveyed using *argument modes* and *determinism categories*.

This chapter explains the mode system and how it is used.

3.1 Predicates and procedures

Every predicate must have a declaration specifying which arguments can be inputs and which outputs. Consider the following implementation of a telephone directory in which phone numbers can be looked-up by name:

```
:- pred phone(string::in, int::out) is semidet.
```

```
phone("Ian", 66532).  
phone("Julien", 66532).  
phone("Peter", 66540).  
phone("Ralph", 66532).  
phone("Zoltan", 66514).
```

The style of **pred** declaration used here is called a **pred-mode** declaration and is syntactic sugar for two separate declarations:

```

:- pred phone(string, int).
:- mode phone(in, out) is semidet.

```

The plain **pred** declaration tells us the argument types; the **mode** declaration tells us the *argument modes* (in or out) and the corresponding *determinism category* (**semidet**).

It is natural to want to call some predicates in more than one way — if we want to use **phone** to perform “reverse look-ups”, for instance. In such cases more than one **mode** declaration is necessary (**pred-mode** shorthand can only be used for singly-moded predicates).

To allow reverse look-ups with **phone** all that is necessary is to (a) use separate **pred** and **mode** declarations and (b) add an extra **mode** declaration:

```

:- pred phone(string, int).
:- mode phone(in, out) is semidet.
:- mode phone(out, in) is nondet.

```

The first **mode** declaration

```

:- mode phone(in, out) is semidet.

```

says that if we call **phone** giving its first argument as an input and taking its second as an output, then the result is semideterministic: every name in the directory appears exactly once, but not all names are listed. The goal **phone**(“Harald”, HaraldsNum) will obviously fail, but **phone**(“Ralph”, RalphsNum) will succeed unifying RalphsNum with 66532.

The second **mode** declaration

```

:- mode phone(out, in) is nondet.

```

says that if we call **phone** with its second argument given as input and take its first argument as output, then the result is nondeterministic: it can fail because the goal **phone**(Person, 12345) will fail, but the goal **phone**(Person, 66532) has no less than three possible solutions — Person = “Ian”, Person = “Julien”, and Person = “Ralph” — each of which will be computed on backtracking.

Note! Each **mode** declaration specifies a *procedure* to be derived from the predicate definition. The compiler generates code separately for each procedure of a predicate, reordering goals to ensure that every variable is instantiated (e.g., by some earlier unification or call) before it is used in an input in a unification or call. Mode information is also used to decide whether a unification with

a data constructor is a construction or deconstruction.

The compiler verifies that the determinism category for a procedure properly describes the behaviour of the procedure. The compiler will issue an error if a procedure can fail or have multiple solutions when its declared determinism category says otherwise. Moreover, the compiler will report an error if a deterministic switch is incomplete, telling you which cases have been missed (Prolog programmers dream of having error detection like this...)

Implied modes

An *implied mode* is one where an output argument is supplied as an input in a procedure call. Consider the goal `phone("Ralph", 66540)`. The compiler gets around this situation by placing a new, temporary variable in the output position and then adding a unification goal, giving `phone("Ralph", Tmp), Tmp = 66540`.

3.2 The determinism categories

A determinism category tells us whether a particular procedure can fail and whether it may have more than one solution:

Determinism category	Number of solutions
det	1
semidet	≤ 1
multi	≥ 1
nondet	≥ 0
failure	0

There are three other determinism categories that are only occasionally needed: **erroneous**, which is used for predicates that only terminate by throwing an exception (exceptions are described in Chapter XXX), and **cc_multi** and **cc_nondet** which are used for committed-choice nondeterminism (see Chapter XXX).

Some examples

```
:- pred square(int::in, int::out) is det.
square(X, X * X).
```

square is **det**: it cannot fail and every input has a single solution for the output.

```
:- pred absolute_square_root(float::in, float::out) is semidet
absolute_square_root(X, AbsSqrtX) :-
    X >= 0.0,
    AbsSqrtX = math.sqrt(X).
```

absolute_square_root is **semidet**: it fails for negative inputs while non-negative inputs each have a single solution.

```
:- pred small_prime(int::out) is multi.
small_prime(2).
small_prime(3).
small_prime(5).
small_prime(7).
```

small_prime is **multi**: it cannot fail and it has more than one solution.

```
:- pred small_prime_factor(int::in, int::out) is nondet.
small_prime_factor(X, P) :-
    small_prime(P),
    X mod P = 0.
```

small_prime_factor is **nondet**: small_prime_factor(11, A), for instance, will fail, but small_prime_factor(6, A) has solutions A = 2 and A = 3.

Finally, the built-in predicate **false**, which takes no arguments, has determinism **failure**: it never succeeds. The opposite of **false** is the built-in predicate **true** which has no arguments and always succeeds (and is therefore **det**).

Note! The determinism category of a goal with no output arguments is either **det**, **semidet**, or **failure**. Consider the following:

```
:- pred has_small_prime_factor(int::in) is semidet.
has_small_prime_factor(X) :-
    small_prime(P),
    X mod P = 0.
```

Because there are no outputs, Mercury ensures that the goal has_small_prime_factor(15), say, will not succeed more than once, even though small_prime(P) has two solutions, P = 3 and P = 5, satisfying 15 mod P = 0.

3.3 Determinism

These rules specify how determinism categories for compound goals are derived (with a little experience this quickly becomes second nature). The determinism category of a goal is derived from the instantiation state of its arguments at the time the goal is executed.

Note! Remember that the compiler reorders the goals in a predicate separately for each **mode** declaration for the predicate. A running program does not make decisions about which procedures should be executed when calling predicates; this is decided in advance by the Mercury compiler.

Unifications

Whether a unification is a construction, deconstruction, assignment or equality test depends upon which variables are instantiated and which are not at the time the unification is executed.

A unification $X = \text{data_ctor}(Y_1, Y_2, Y_3)$ is a *construction* if Y_1 , Y_2 , and Y_3 are initially instantiated and X is not. Constructions are always **det**. Afterwards, X will be instantiated.

A unification $X = \text{data_ctor}(Y_1, Y_2, Y_3)$ is a *deconstruction* if X is initially instantiated. Afterwards, Y_1 , Y_2 , and Y_3 will be instantiated. Deconstructions are almost always **semidet** (in certain circumstances a deconstruction may have determinism category **det** if it is guaranteed to succeed or **failure** if it is guaranteed to fail).

A unification $X = Y$ is an *assignment* if precisely one of X or Y is initially instantiated. Afterwards, both variables will be instantiated. Assignments are always **det**.

A unification $X = Y$ is an *equality test* if both X and Y are initially instantiated. Equality tests are always **semidet**.

Procedure calls

For a predicate call $p(X_1, X_2, X_3)$, which procedure of p is executed depends upon which **mode** declaration for predicate p best matches the instantiation

states of X_1 , X_2 , and X_3 at the time the call is executed. The determinism category of the goal is that of the called procedure, adjusted for any implied modes (i.e., extra unifications added because some output arguments of the procedure are already instantiated at the time of the call).

For example, the goal `phone("Zoltan", ZoltansNumber)` is compiled as a call to the (in, out) **is det** procedure of `phone`. The goal `phone(Person, 66540)` is compiled as a call to the (out, in) **is nondet** procedure of `phone`. The goal `phone("Ralph", 66532)` requires an implied mode and may be compiled either as `phone("Ralph", Tmp), Tmp = 66532` or `phone(Tmp, 66532), Tmp = "Ralph"`, both of which are **semidet**.

Conjunction

A sequence of goals separated by commas, G_1, G_2, G_3, \dots , is called a *conjunction*. The commas are pronounced "and" and each subgoal is called a *conjunct*.

A conjunction can fail if any conjunct can fail.

A conjunction can succeed if every conjunct can succeed.

A conjunction can have multiple solutions if it can succeed and one or more conjuncts have multiple solutions.

Note! These rules are a conservative (i.e., safe) approximation. For example, the compiler will conclude that conjunction `small_prime(X), X = 4` is **semidet**, even though we can see that this goal has to fail.

Disjunction

A sequence of goals separated by semicolons, $(G_1 ; G_2 ; G_3 ; \dots)$, is called a *disjunction*. The semicolons are pronounced "or" and each subgoal is called a *disjunct*.

A disjunction can succeed if any disjunct can succeed.

A disjunction can have multiple solutions if more than one disjunct can succeed or one or more disjuncts can have multiple solutions.

Note! Switches are a special case. A switch is a disjunction that deconstructs a particular variable against a different data constructor in each disjunct. If, apart from the deconstructions, every disjunct is **det**, then the switch is **det** if the set of deconstructions is exhaustive and **semidet** if not.

For example, even though both **p** and **q** (below) define switches on **X**, **p** is **det** because its switch is exhaustive, whereas **q** is **semidet** because its switch is not:

```
:- type ott ---> one ; two ; three.
```

```
:- pred p(ott::in, int::out) is det.
```

```
p(X, Y) :- ( X = one, Y = 1 ; X = two, Y = 2 ; X = three, Y = 3 ).
```

```
:- pred q(ott::in, int::out) is semidet.
```

```
q(X, Y) :- ( X = one, Y = 1 ; X = three, Y = 3 ).
```

Note! Disjunction binds less tightly than conjunction:

```
( G11, G12, G13 ; G21 ; G31, G32 )
```

is equivalent to

```
( (G11, G12, G13) ; G21 ; (G31, G32) ).
```

Note! A definition spanning multiple clauses is equivalent to a definition using a single clause containing a disjunction. That is

```
p(one, 1).
```

```
p(two, 2).
```

```
p(three, 3).
```

is semantically and operationally identical to

```
p(X, Y) :- ( X = one, Y = 1 ; X = two, Y = 2 ; X = three, Y = 3 ).
```

Note! If any disjunct instantiates a variable that is used outside the disjunction, then every disjunct in the disjunction must also instantiate that variable. That is, the Mercury compiler will report a mode error if a program contains a disjunction that instantiates **X** in some disjuncts, but not others, and **X** is also needed outside the disjunction. For instance, the following is illegal because **Y**, which appears outside the disjunction, is instantiated in the first and second disjuncts, but not the third:

```
:- pred p(number::in, int::out) is det.
```

```
p(X, Y) :- ( X = one, Y = 1 ; X = two, Y = 2 ; X = three ).
```

Negation

A goal (**not** G) is called the *negation* of G. The negation fails if G succeeds, and vice versa. The negation succeeds if G fails and fails if G succeeds.

Note! G is said to occur inside a *negated context* and is not allowed to instantiate variables that also occur outside the negation.

Note! Negation binds more tightly than conjunction, hence **not** G₁, G₂, ... is equivalent to (**not** G₁), G₂, To negate a conjunction, put the conjunction in parentheses: **not** (G₁, G₂, ...)

Note! $X \setminus = Y$ is syntactic sugar for **not** (X = Y).

If-then-else goals

The declarative semantics for a goal (**if** Gc **then** Gt **else** Ge) are identical to those of (Gc, Gt ; (**not** Gc), Ge). The operational semantics are more efficient, though: if there are no solutions to Gc, the program immediately executes Ge.

If any of Gc, Gt, or Ge can fail then the **if-then-else** can fail.

If any of Gc, Gt, or Ge can have multiple solutions then the **if-then-else** can have multiple solutions.

Note! Gc is not allowed to instantiate variables that are used outside the **if-then-else**. This is because, semantically, Gc appears in a negated context. It is all right, however, for Gc to instantiate variables that are used by Gt.

Note! Execution can backtrack into Gc. For example,

(**if** small_prime(X), X > 2 **then** Y = X * X **else** Y = -1)

has solutions Y = 9, Y = 25, and Y = 49.

Prolog programmers take note: unlike Mercury, Prolog programs commit to the first solution of Gc. The Prolog equivalent of the above goal would have Y = 9 as its only solution, not Y = 25 or Y = 49.

Note! (Gc \rightarrow Gt ; Ge) is an alternative, albeit old-fashioned, syntax for (**if** Gc

then Gt **else** Ge).

3.4 Procedures and code reordering

The aim of this section is to give the reader some understanding of code reordering. This knowledge is not required to write Mercury programs, but it can help the programmer understand mode-related error messages from the compiler.

We will illustrate using the `append` predicate defined in the `list` standard library module. The declarative semantics of `append(Xs, Ys, Zs)` is that the list `Zs` is the concatenation of lists `Xs` and `Ys`. So `append([1], [2, 3], [1, 2, 3])` is true, but `append([2, 3], [1], [1, 2, 3])` is not.

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
```

```
append(Xs, Ys, Zs) :-
  (
    Xs = [], Zs = Ys
  ;
    Xs = [X | Xs0], append(Xs0, Ys, Zs0), Zs = [X | Zs0]
  ).
```

This code needs no reordering for the `(in, in, out)` **is det** mode, where `Xs` and `Ys` start off instantiated and `Zs` starts off uninstantiated. The first disjunct works like this:

```
1 Xs = []           Deconstruct Xs
2 Zs = Ys           Assign Zs
```

and the second disjunct works like this:

```
1 Xs = [X | Xs0]    Deconstruct Xs, instantiating X and Xs0
2 append(Xs0, Ys, Zs0) Call the (in, in, out) procedure, instantiating
                       Zs0
3 Zs = [X | Zs0]    Construct Zs
```

Because each disjunct deconstructs `Xs` in a different way, this disjunction is a switch. Because the switch is exhaustive, and the other goals in each disjunct are **det**, the switch as a whole is **det**.

The (out, out, in) **is multi** mode, where only Zs is initially instantiated, does require some reordering in order to ensure that every variable is instantiated before it is needed. The first disjunct becomes

- | | | |
|---|-----------|----------------|
| 1 | $Xs = []$ | Construct Xs |
| 2 | $Zs = Ys$ | Assign Ys |

and the second disjunct becomes

- | | | |
|---|--------------------------|------------------------------------------------------------------|
| 1 | $Zs = [X \mid Zs_0]$ | Deconstruct Zs , instantiating X and Zs_0 |
| 2 | $append(Xs_0, Ys, Zs_0)$ | Call the (out, out, in) procedure, instantiating Xs_0 and Ys |
| 3 | $Xs = [X \mid Xs_0]$ | Construct Xs |

Since this disjunction is not a switch and the first disjunct always leads to a solution, the disjunction as a whole is **multi** in this case.

Note! The Mercury compiler reorders code as little as possible. However, programmers should not write code that depends upon any particular order of evaluation — code can also be reordered by various optimizations! In particular, it is a bad idea to write something like (**if** $X \neq 0$, $Z = Y / X$ **then** ... **else** ...), assuming that the test for X being non-zero will guarantee that this code cannot lead to a division-by-zero error at run-time. It is certainly possible that the test and the division may be compiled the other way around.

3.5 Insts and subtypes

So far we have only talked about variables going from being uninstantiated to being instantiated. It turns out to be useful to also keep track of the possible values a variable can have when it is instantiated. Mercury uses **insts** for this purpose. An **inst** represents the possible instantiation states of a variable at a particular point in a program.

The most basic **insts** are **free**, meaning a variable is uninstantiated, and **ground**, meaning a variable is instantiated with some unknown value of the appropriate type.

The built-in **modes** in and out are defined using the following syntax:

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

That is, an **in** mode argument of a goal must be **ground** (i.e., be instantiated with some value) before the goal is executed and will also be **ground** afterwards, while an **out** mode argument must be **free** (i.e., not instantiated) before the goal is executed, but will be **ground** afterwards.

Note! If a goal fails or backtracks then the **insts** of its arguments stay the same as they were before the goal was tried.

Specialised insts

It is occasionally useful to define new **insts** matching only subsets of possible values that a variable might have. Consider the **inst** `non_empty_list` which is defined in the `list` standard library module:

```
:- inst non_empty_list == bound([ground | ground]).
```

```
:- pred head(list(T), T).
```

```
:- mode head(in, out) is semidet.
```

```
:- mode head(in(non_empty_list), out) is det.
```

```
head(Xs, X) :- Xs = [X | _].
```

The **inst** declaration defines `non_empty_list` to mean “bound to the list data constructor `[]` whose first argument has inst `ground` and whose second argument has inst `ground`”.

The first mode for `head` tells us that if all we know about the first argument is that it is **ground** (i.e., it could be bound to any value of type `list(T)`, including `[]`) then a call to `head` is **semidet**.

The second mode for `head` says that if we know the first argument is a non-empty list (i.e., whatever value it has, its top-most data constructor must be `[]` with two **ground** arguments) then a call to `head` is guaranteed to succeed.

The second **mode** declaration uses the built-in parameterised form of the **in** argument mode, which is defined like this:

```
:- mode in(I) == (I >> I).
```

where `I` is an **inst** parameter. `in(non_empty_list)` is therefore equivalent to writing `(non_empty_list >> non_empty_list)`. There is also a built-in parameterised **out** argument mode, defined thus:

```
:- mode out(l) == (free >> l).
```

When compiling the procedure for the second mode of `head`, the Mercury compiler uses the information about the `inst` of the first argument, `Xs`, to infer that the goal `Xs = [X | _]` must (a) be a deconstruction and (b) must succeed because whatever value `Xs` has matches the pattern `[_ | _]`.

Note! A value with a `bound(...)` **inst** can always be used in a context where a ground value is expected, but not the other way around.

XXX Mention the alternative **inst** definition syntax.

Recursively defined insts

It is possible to describe quite complicated instantiation states. The following **insts**, for instance, describe lists of even and odd lengths respectively:

```
:- inst even_length_list == bound([], [ground | odd_length_list]).
:- inst odd_length_list == bound([ground | even_length_list]).
```

The first **inst** declaration defines `even_length_list` to mean “bound *either* to `[]` *or* to `[]` with two arguments, the first having **inst** `ground` and the second having **inst** `odd_length_list` (multiple possibilities in a `bound` expression are separated by semicolons).

The second **inst** declaration defines `odd_length_list` to mean “bound to `[]` with two arguments, the first having **inst** `ground` and the second having **inst** `even_length_list`.”

Partial instantiation

A partially instantiated value is one whose **inst** is `bound(...)` where the ... part contains **free** sub-**insts**, either directly or indirectly.

Partial instantiation is not currently supported for several reasons, including the difficulty of analysing such code, the difficulty of maintaining such code, and the difficulty of compiling such code efficiently.

3.6 Uniqueness

A `bound(...)` `inst` is said to be *shared* — that is, it corresponds to a value that may be referred to, directly or indirectly, by more than one variable at a given point in the program.

Mercury has a special `inst`, `unique`, which is like `ground`, but it means that there is precisely one reference to the `unique` data at this point in the program. The counterpart to `unique`, is `clobbered`. A variable with **`inst`** `clobbered` may never be used again (e.g., because the value it refers to is now out-of-date or has been overwritten with something else).

The most common use of uniqueness is for managing IO. All the IO operations defined in the `io` standard library module include two arguments of type `io`, with modes `di` and `uo` respectively. `di` stands for “destructive input” and `uo` stands for “unique output”. These **`modes`** are built-in and defined thus:

```
:- mode di == (unique >> clobbered).
:- mode uo == (free >> unique).
```

To illustrate, consider these **`pred`** declarations taken from the `io` module:

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

and the following code snippet:

```
io.write_string("The meaning of life is ", IO0, IO1),
io.write_int(42, IO1, IO2),
io.nl(IO2, IO3)
```

The `io` type arguments denote “states of the world”. These `io` states are updated when IO actions are performed. One can never go back to an earlier state (you can’t unplay a piece of music or unprint a document), so each IO action clobbers the `io` state passed to it and produces and new `io` state as its result. Similarly, because one cannot copy the state of the world, `io` states have to be unique. These constraints ensure that the above code snippet executes in the expected order — that is, first the string “The meaning of life is ” will be printed (clobbering `IO0` and producing `IO1`), then the number 42 (clobbering `IO1` and producing `IO2`), and finally a newline (clobbering `IO2` and producing `IO3`).

Say we were to accidentally reuse `IO0` in the second goal:

```
io.write_string("The meaning of life is ", IO0, IO1),
io.write_int(42, IO0, IO2),
io.nl(IO2, IO3)
```

The Mercury compiler will report the following error (line 27 in file `foo.m` is the call to `io.write_string`):

```
foo.m:027: In clause for 'main(di, uo)':
foo.m:027:   in argument 2 of call to predicate 'io.write_string/3':
foo.m:027:   unique—mode error: the called procedure would clobber
foo.m:027:   its argument, but variable 'IO0' is still live.
```

Note! Procedures that can clobber arguments must have determinism category **det** or **cc_multi**. They must always succeed and produce a single result. The reason for this is that once an argument is clobbered, which could happen at any point during the execution of the procedure, there is no way of unclobbering it on failure or backtracking. Consequently it is also an error for code to backtrack into such procedures. For the rare cases where one needs to do such things, the reader is referred to the section on backtrackable destructive update in the Mercury reference manual which discusses “mostly uniqueness”.

Other modules in the standard library that use uniqueness are `array` and `store`. The `array` module implements arrays with $O(1)$ look-up and set operations. It achieves this by using destructive update for the array set operation. This is quite safe because arrays are unique: the ‘old’ version of the array is clobbered by the update operation so it can never be referred to again; the ‘new’ version of the array is simply the updated ‘old’ version. The `store` module allows one to construct safe, pointer-based structures. Pointer referents can be accessed and updated in $O(1)$ time by using the same technique as for arrays. Chapter XXX describes these modules in more detail.

3.7 Higher-order modes

XXX *Fill this out.*

3.8 Committed-choice nondeterminism

XXX *Fill this out.*

The **cc_nondet** and **cc_multi** modes. There may be multiple solutions to a cc predicate, but you will only get one of them.

The compiler will report an error if a program can backtrack into a committed-choice goal: all goals following a committed-choice goal must be *guaranteed* to succeed. Programming under this restriction is quite burdensome.

If all solutions to a committed-choice predicate are equivalent, in the sense that, no matter which solution you get, the observable behaviour of your program will be the same, then you can use the built-in function `promise_only_solution` to escape from the committed-choice context.