

# **The Mercury Programming Language**

**Melbourne Compose Talk, November 2025**

**David Overton**

# What is Mercury?

- Logic/functional programming language
- Purely declarative
  - no side effects
  - referential transparency
- Strict (i.e. not lazy)
- Strong static type system
- Strong mode and determinism systems
- Efficient execution model

## A Bit of History

- Started in **1994** at the University of Melbourne
- Creators: **Zoltan Somogyi, Fergus Henderson, Thomas Conway**, et al.
- Goal: industrial-strength logic programming
- Original compiler written in the intersection of Mercury and NU-Prolog until it could bootstrap itself
- Became a focus of PL / LP research at Melbourne, Monash and RMIT, as well as further afield (KU Leuven, Belgium; Uppsala, Sweden)
- Several commercial users
- No longer an active research project, but still under development by Zoltan and others

# What is Logic Programming?

- Developed in the 1970s based on predicate logic
- Key features:
  - Clauses (facts and rules)
  - Logical connectives (and, or, not)
  - Unification
  - Backtracking for search

# Prolog

```
% facts
parent(alice, bob).
parent(alice, barb).
parent(bob, carol).
parent(bob, charlie).
parent(barb, chris).
parent(barb, cate).
```

```
% rule
grandparent(A, C) :-
    parent(A, B),
    parent(B, C).
```

```
% query
?- grandparent(alice, X).
    X = carol ; X = charlie ; X = chris ; X = cate.
```

# Predicate logic

```
grandparent(A, C) :-  
  parent(A, B),  
  parent(B, C).
```

<marp-pre>

$$\forall A \forall C \text{ grandparent}(A, C) \leftarrow \\ \exists B \text{ parent}(A, B) \wedge \text{parent}(B, C)$$

</marp-pre>

- Implicit variable quantification
- `:-` is  $\leftarrow$  (reverse implication or 'if')
- `,` is  $\wedge$  (logical conjunction or 'and')

# Problems with Prolog

- Lack of static types, modes and determinism
  - No help from compiler to catch potential errors
  - Hard for compiler to generate efficient code
  - Negation can be unsound
  - Requires use of non-logical constructs (such as *cut*) for efficiency
- Impure features, e.g. I/O
  - Makes it harder to reason about code logically

# Mercury example

```
:- pred append(list(T), list(T), list(T)).  
:- mode append(in, in, out) is det.  
  
append([], Ys, Ys).  
append([X | Xs], Ys, [X | Zs]) :-  
    append(Xs, Ys, Zs).
```

compare Haskell code:

```
append :: [a] -> [a] -> [a]  
append [] ys = ys  
append (x:xs) ys = x : append xs ys
```

- Relation, not a function, but otherwise similar in structure to Haskell equivalent
- Prolog syntax for variables (upper case), function symbols (lower case), and lists `[X | Xs]`



Can write as a single clause:

## Mercury

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

- Comma for conjunction (**and**)
- Semicolon for disjunction (**or**)
- Unification ( = ) for pattern matching

## Haskell

```
append xs ys =  
  case xs of  
    []      -> ys  
    (x : xs') ->  
      x : append xs' ys
```

- Disjuncts are non-overlapping because `Xs` unifies with two different list constructors

**Aside:** Mercury also has functions so we could write `append` as a function:

```
:- func append(list(T), list(T)) = list(T).  
  
append([], Ys) = Ys.  
append([X | Xs], Ys) = [X | append(Xs, Ys)].
```

compare Haskell code:

```
append :: [a] -> [a] -> [a]  
append [] ys = ys  
append (x:xs) ys = x : append xs ys
```

but ...

... `append` as a predicate can have other *modes*:

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

```
append([], Ys, Ys).  
append([X | Xs], Ys, [X | Zs]) :-  
    append(Xs, Ys, Zs).
```

```
?- append(Xs, Ys, [1, 2, 3]).
```

multiple solutions:

```
Xs = [],      Ys = [1, 2, 3]  
; Xs = [1],   Ys = [2, 3]  
; Xs = [1, 2], Ys = [3]  
; Xs = [1, 2, 3], Ys = []
```

(uses mode `append(out, out, in)`)

Solution space explored through depth-first search and backtracking.

## Another example

```
:- pred member(T, list(T)).  
:- mode member(in, in) is semidet.  
:- mode member(out, in) is nondet.
```

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

Existential quantification:

```
?- some [X] (  
    member(X, [1, 2, 3]),  
    X > 1).  
true.
```

Universal quantification:

```
?- all [X] (  
    member(X, [1, 2, 3]) =>  
    X > 1).  
fail.
```

# Types

Features of the Mercury type system:

- strong static type system
- algebraic data types
- subtypes
- type inference (with ad-hoc overloading)
- higher order types
- record types with named fields
- type classes
- existential types
- runtime type information (reflection)
- no higher-kinded types 🤔

## Example type definitions

	Mercury	Haskell
<b>enum</b>	<pre><code>:- type bool ---&gt; yes ; no.</code></pre>	<pre><code>data Bool = True   False</code></pre>
<b>polymorphic type</b>	<pre><code>:- type maybe(T)     ---&gt; yes(T)     ;    no.</code></pre>	<pre><code>data Maybe a     = Just a       Nothing</code></pre>
<b>type alias</b>	<pre><code>:- type width == float.</code></pre>	<pre><code>type Width = Double</code></pre>
<b>newtype</b>	<pre><code>:- type counter     ---&gt; counter(int).</code></pre>	<pre><code>newtype Counter     = Counter Int</code></pre>

# Higher order types

```
:- pred map(pred(T, U), list(T), list(U)).  
:- mode map(pred(in, out) is det, in, out) is det.
```

```
map(_, [], []).  
map(P, [X | Xs], [Y | Ys]) :-  
    P(X, Y),  
    map(P, Xs, Ys).
```

## Currying

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

```
?- map(add(1), [1, 2, 3], Ys).  
Ys = [2, 3, 4]
```



# Modes

- Describe data flow through *instantiation states* of variables

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

- Mode declarations for a predicate must give a mode for each argument:

```
:- mode append(in, in, out).
```

- Functions have a default mode where arguments have mode `in` and the function result has mode `out`, unless otherwise specified.

# Determinism

Each mode of a predicate or function is categorised by whether or not it can fail and how many solutions it can produce:

- `det` : exactly one solution
- `semidet` : at most one solution (can fail or succeed once)
- `multi` : at least one solution
- `nondet` : zero or more solutions
- `failure` : no solutions (always fails)
- `erroneous` : never returns (infinite loop, exception or runtime error)

max solutions	0	1	>1
cannot fail	erroneous	det	multi
can fail	failure	semidet	nondet

# I/O and unique modes

- Mercury is a *pure* declarative language, so how do we do I/O?
- Thread a “state of the world” through predicates that do I/O

```
:- pred write_string(string, io, io).
```

- But we want to make sure an `io` state is never re-used, even when backtracking.

```
:- mode write_string(in, di, uo) is det.
```

- `di` : destructive input
- `uo` : unique output

# Unique modes

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

- `unique` : a unique reference to this value
- `clobbered` : no references to the value (it may have been destroyed or destructively updated)
- The compiler ensures that the `di` argument is a unique reference and is never used again after the call.
- Requires that calls doing I/O can't fail and can't be retried via backtracking.

# I/O example

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

main(I00, I0) :-
    io.write_string("What is your name?\n", I00, I01),
    io.read_line_as_string(Result, I01, I02),
    (
        Result = ok(String),
        io.format("Hello %s, nice to meet you!\n", [s(strip(String))], I02, I03),
        main(I03, I0)
    ;
        Result = eof,
        io.write_string("Ok, bye!\n", I02, I0)
    ;
        Result = error(Err),
        io.write_string("Error: ", I02, I03),
        io.print(Err, I03, I0)
    ).
```

## State variables

- Threading those numbered `I0n` variables through the code can get tedious very fast.
- Haskell uses a state monad to hide the I/O state.
- Mercury has a difference solution: *state variables*.

# State variables

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

main(!IO) :-
  io.write_string("What is your name?\n", !IO),
  io.read_line_as_string(Result, !IO),
  (
    Result = ok(String),
    io.format("Hello %s, nice to meet you!\n", [s(strip(String))], !IO),
    main(!IO)
  ;
    Result = eof,
    io.write_string("Ok, bye!\n", !IO)
  ;
    Result = error(Err),
    io.write_string("Error: ", !IO),
    io.print(Err, !IO)
  ).
```



# Module system

- Modules have `interface` and `implementation` sections.
- Only things declared in the `interface` section are exported.
- There are also submodules, either nested or in separate files.

## Module example

```
:- module hello.  
:- interface.  
:- import_module io.  
  
:- pred main(io::di, io::uo) is det.  
  
:- implementation.  
  
:- import_module list, string.  
  
main(!IO) :-  
    ...
```

# Abstract types

- A type declared in the module interface, but defined in the implementation is an *abstract type*

```
:- module set.  
:- interface.  
  
:- type set(T).  
  
:- implementation.  
  
:- type set(T)  
    ---> set(list(T)).
```

# Tooling

- `mmc` : Melbourne Mercury Compiler
  - Backends targetting C, Java and C#
  - C backend uses Boehm GC
- `mdb` : Mercury debugger
  - supports `retry` , which is awesome.
- `mprof` : time and memory profiler
- standard library: lots of useful data structures

# Who uses Mercury?

- Academic research projects, such as
  - HAL constraint logic programming language (Melbourne/Monash Uni)
  - G12 constraint programming platform (NICTA)
- Mission Critical IT — ODASE ontology platform
- Opturion — optimisation platform (commercial spin-off from G12)
- YesLogic — Prince HTML+CSS to PDF typesetting software

# Example program: Sudoku

4	1		3				6	5
			2				4	
				5	4	8	1	
	5				2			
6	4						7	2
			6				5	
	8	4	7	1				
	6				3			
5	7				6		8	1

(c) Daily Sudoku Ltd 2025. All rights reserved.

Daily Sudoku: Tue 18-Nov-2025