Cambria Language Reference

A Functional Language with
Parametrized Algebraic Effects and Handlers

A complete guide for programming in Cambria

Cambria is an experimental language under active development. This reference describes the implementation at the time of writing. The interpreter is the authority when the two disagree.

Contents

1 Running Programs

```
cabal build
cabal run cambria -- <file.cba>
```

Programs are written in `.cba` files. The interpreter parses, desugars, type-checks, and evaluates the program, printing the final result together with its inferred type.

Output format: `<result> : <type>` (e.g. `Pure: 5 : Int!{}`).

2 Values and Types

Cambria distinguishes between **values** (inert data) and **computations** (potentially effectful expressions). This is a fine-grained call-by-value discipline.

2.1 Primitive Types

Type	Literal examples	Description
`Unit`	`()`	The unit value
`Int`	`0`, `42`, `-3`	Arbitrary-precision integers
`Bool`	`true`, `false`	Booleans
`Double`	`1.0`, `0.5`, `-0.25`	Floating-point numbers
`Str`	`"hello"`, `"a\nb"`	Strings (with escapes)
`Name`	(produced by `!fresh`)	Opaque unique identifiers

String escape sequences: `\n` (newline), `\t` (tab), `\"` (double quote), `\\` (backslash).

Numeric Literals

Point notation separates the two numeric types: `1` is an `Int` and `1.0` is a `Double`. A `Double` literal needs digits on both sides of the point, so `1.` and `.5` are not literals.

A `-` written directly against the digits belongs to the literal, so `-3` and `(-3, 4)` and `3 - -2` all contain a negative literal. Where an operand comes first there is nothing to negate and the same text is a subtraction, so `n-1`, `n - 1` and `n -1` all subtract.

The one thing to know is that a signed literal cannot be an application argument on its own: `f -3` is the subtraction `f - 3`, and applying `f` to `-3` is written `f (-3)`.

2.2 Compound Types

Pairs `A * B`

```
(1, 2)              -- Int * Int
((1, 2), true)      -- (Int * Int) * Bool
```

Sum Types (Either) `A + B`

```
inl 42              -- creates a left injection
inr true            -- creates a right injection
```

Lists `List A`

```
[]                  -- empty list
1 :: 2 :: 3 :: []   -- list [1, 2, 3]
```

Maps `Map K V`

```
empty                       -- empty map
insert ((key, val), m)      -- insert into map
```

2.3 Type Syntax

Types are written in `effect` declarations and in explicit annotations (Section 15), using this syntax:

Syntax	Meaning
`Unit`	Unit type
`Void`	Empty type (no values)
`Int`	Integer type
`Bool`	Boolean type
`Double`	Floating-point type
`Str`	String type
`Name`	Fresh-name identifier type
`A * B`	Pair type (product)
`A + B`	Sum type (either)
`Map K V`	Map type
`List A`	List type
`A -> B!{ops}`	Function type
`A!{ops} => B!{ops}`	Handler type
`$p`	Abstract type parameter
`(A)`	Parenthesised type

Precedence: `+` binds looser than `*`, so `Int * Bool + Str` means `(Int * Bool) + Str`. Neither operator associates, so a chain of three or more needs parentheses.

A function type carries the computation type of its result, effects included, so an operation can take a callback:

```
effect !atomically : (Unit → Int!{ get : Unit ⇝ Int }) ⇝ Int.
```

A handler type is written with a fat arrow between two computation types, the one it consumes and the one it produces (Section 14.5).

```
(handler { return x → return x } : Int!{} ⇒ Int!{})
```

`Void` has no values and no literal. It is the sensible result type for an operation that never returns, such as a jump or a thread halting (`effect !stop : Unit ⇝ Void.`). The built-in `absurd` turns a `Void` into anything.

3 Values and Computations

Cambria programs are built from two syntactic categories:

- **Values**: inert data that can be passed around (integers, booleans, functions, pairs, etc.)
- **Computations**: expressions that may perform effects or produce a value

A top-level program is a computation.

3.1 `return`

Lifts a value into a computation that does nothing and produces that value:

```
return 42
return (1, 2)
return ()
```

3.2 `do ... <- ... in ...`

Sequences computations, binding the result of the first to a name used in the second:

```
do x ← return 42 in
do y ← return 10 in
return (x + y)
```

The bound name is available in everything after `in`.

3.3 Semicolon `;`

A shorthand for `do _ ← ... in ...` that sequences two computations, discarding the first result:

```
!print "hello" ; !print "world" ; return ()
```

This desugars to:

```
do _ ← !print "hello" in
do _ ← !print "world" in
return ()
```

3.4 `let` Bindings

`let` binds the result of an expression or computation to a pattern:

```
let x = 5 in
let (a, b) = (1, 2) in
let y = !get r in       -- the right-hand side may perform effects
return (x + a + b + y)
```

`let x = e in c` is shorthand for `do x ← return e in c`. A computation on the right is lifted into the binding by ANF conversion (Section 16), so it runs once, where the `let` stands, however many times `x` is used afterwards.

One `let` may bind several names, separated by commas. The bindings are nested left to right, so each right-hand side sees the ones before it, and their effects happen in that order:

```
let x = 1,
    y = x + 2,
    w = !get r
in return (x + y + w)
```

With argument patterns, `let` defines a function, and `let rec` a recursive function:

```
let double x = x + x in
let addPair (x, y) = x + y in
let rec fact n =
  if n == 0 then return 1
  else do r ← fact (n - 1) in return (n * r)
in fact 5
```

These are shorthand for binding a `fun` or `rec` value: `let f x = c in ...` desugars to `do f ← return (fun x → c) in ...`.

Under `let rec` the bindings of a group are mutually visible, so each may call the others:

```
let rec even n = if n == 0 then return true  else odd (n - 1),
        odd  n = if n == 0 then return false else even (n - 1)
in even 10
```

This desugars to a single recursive thunk returning the tuple of definitions, together with a wrapper binding for each name (Section 16); the definitions are separate components of the tuple, so they need not share a type.

A `let` may also bind an infix operator by carrying a fixity, which is the only way to introduce one (Section 7.7):

```
let infixl 8 xs <+ x = return (x :: xs) in
[] <+ 1 <+ 2
```

4 Functions

4.1 Anonymous Functions (`fun`)

```
fun x → return (x + 1)
```

Multi-argument functions are written with multiple variables:

```
fun x y → return (x + y)
```

This desugars to a curried function: `fun x → return (fun y → return (x + y))`.

Function bodies are computations (they can perform effects).

4.2 Recursive Functions (`rec`)

```
rec factorial n →
  if n == 0 then return 1
  else do r ← factorial (n - 1) in
       return (n * r)
```

The first identifier after `rec` is the function's own name (for recursion), followed by argument patterns:

```
rec f x y → ...    -- f is the recursive name, x and y are arguments
```

4.3 Function Application

Function application is juxtaposition:

```
f x           -- apply f to x
f (x, y)      -- apply f to the pair (x, y)
```

Application is a computation and may trigger effects. Since functions take a single argument, multi-argument application is curried:

```
do add ← return (fun x y → return (x + y)) in
add 1 2
```

Here `add 1` returns a closure, and then `2` is applied to that closure.

5 Pattern Matching

Patterns appear in `do`-bindings, function arguments, `rec` arguments, `let` bindings, handler clauses, and the cases of a `case`.

Binding positions (`do`, `fun`, `rec`, `let`, handler clauses) accept only *irrefutable* patterns, which always match:

- A variable name (e.g. `x`): binds the matched value
- A wildcard `_`: matches anything, discards the value
- The unit pattern `()`
- A pair pattern `(p1, p2)`: destructs a pair, matching each component with a sub-pattern

Cases additionally accept *refutable* patterns, which can fail to match (Section 6):

- Sum patterns `inl` *p* and `inr` *p*
- List patterns `[]` and *p1* `::` *p2*
- Literal patterns: integers, doubles, booleans, and strings. Numeric literals may be signed, so `-1` and `-0.25` are patterns; a literal pattern is tested with `==`

All of these nest freely, e.g. `inl (x :: (1, _) :: rest)`.

5.1 Wildcards

Use _ anywhere a pattern is expected to discard a value:

```
do _ ← !print "hello" in return ()
fun _ → return 42
rec f _ → return 0
```

Wildcards also work inside pair patterns:

```
do (x, _) ← return (1, 2) in return x
fun (_, y) → return y
```

5.2 Pair Patterns

Pairs can be destructured wherever patterns are accepted:

```
-- In do-bindings
do (x, y) ← return (1, 2) in return (x + y)

-- In function arguments
fun (x, y) → return (x + y)

-- Nested patterns
fun ((a, b), c) → return (a + b + c)

-- In handler operation clauses
set (ref, val) k → ...
```

Pair patterns desugar into pair projections (Section 16).

6 Control Flow

6.1 If-Then-Else

```
if condition then computation1 else computation2
```

The condition is a value of type `Bool`, and both branches are computations:

```
if x == 0 then return "zero" else return "nonzero"
```

6.2 Case Expressions

`case` matches a value against a sequence of cases:

```
case expr of {
  pattern1 → computation1,
  pattern2 → computation2,
  ...
}
```

The scrutinee may be a computation; it is evaluated once, before matching. Cases are tried top to bottom and the first matching case wins, so an earlier case may overlap a later one:

```
case xs of {
  []        → return 0,
  x :: rest → return x
}

case n of {
  0 → return "zero",
  1 → return "one",
  _ → return "many"
}

case v of {
  inl (x :: _)   → return x,
  inl []         → return 0,
  inr (true, y)  → return y,
  inr (false, _) → return 99
}
```

Matching compiles down to the kernel's binary sum `case`, list destructuring and equality tests (Section 16); a two-case `inl`/`inr` match maps directly onto the kernel construct. If no case matches at runtime the program aborts with a pattern-match failure, so end with an irrefutable case when the earlier cases are not exhaustive.

An irrefutable case must be the last one written. Since it always matches, any case placed after it could never run, and such a match is rejected at compile time.

7 Operators

7.1 Arithmetic

Operator	Types	Result
`+`	`Int * Int`	`Int`
`-`	`Int * Int`	`Int`
`*`	`Int * Int`	`Int`
`/`	`Int * Int`	`Double`

Note: `/` always produces a `Double`, even for integer operands.

7.2 Comparison

Operator	Types	Result
`==`	`a * a`	`Bool`
`/=`	`a * a`	`Bool`
`<`	`Int * Int`	`Bool`
`>`	`Int * Int`	`Bool`
`<=`	`Int * Int`	`Bool`
`>=`	`Int * Int`	`Bool`

The `==` and `/=` operators are polymorphic. They work on any type where structural equality is defined (integers, doubles, booleans, strings, unique values, pairs, lists, maps, sum types). The

four orderings are on `Int`, like the other arithmetic. All six are non-associative, so `a < b < c` is a parse error.

7.3 Boolean Connectives

Operator	Types	Result
`&&`	`Bool * Bool`	`Bool`
`||`	`Bool * Bool`	`Bool`

```
if x == 0 || 0 < y && y ≤ 10 then return true else return false
```

Both are right-associative, and `||` binds looser than `&&`, so the condition above reads `(x == 0) || ((0 < y) && (y ≤ 10))`. Neither is short-circuiting: they are ordinary functions on a pair, so both arguments are evaluated.

7.4 String Concatenation

Operator	Types	Result
`++`	`Str * Str`	`Str`

7.5 List Cons

Operator	Types	Result
`::`	`a * List a`	`List a`

```
1 :: 2 :: 3 :: []    -- builds [1, 2, 3]
```

`::` is right-associative.

7.6 Operator Precedence (Low to High)

Every infix operator sits at one of ten numbered levels, 0 (loosest) to 9 (tightest). The built-in operators occupy seven of them, and levels 0, 8 and 9 are left free for operator bindings (Section 7.7):

Level	Operators	Associativity
0	*(free)*	
1	`::` (cons)	right
2	`||` (disjunction)	right
3	`&&` (conjunction)	right
4	`== /= < > <= >=` (comparison)	none
5	`++` (string concat)	left
6	`+ -` (additive)	left
7	`* /` (multiplicative)	left
8, 9	*(free)*	

Sequencing with `;` is looser than every level, and function application binds tighter than every level.

All infix operators desugar into function applications. For example, `x + y` becomes `(+) (x, y)`.

The compound computation forms are looser still. The body of a `do` or a `let`, the branches of an `if`, the cases of a `case`, the body of a `with ... handle`, and the scope of an `effect` declaration all extend as far to the right as they can, across `;` and to the end of the enclosing computation. So

```
do x ← return 1 in return x ; return x
```

binds `x` in both computations of the sequence, rather than closing the `do` at the semicolon. This is what lets a handler be written as a prefix, with its body simply following it:

```
with handler { ... } handle

effect !op : Unit ⇝ Int.
!op () ; !op ()
```

Parenthesise a computation to end its scope early.

7.7 User-Defined Operators

An operator is not a separate kind of thing: it is an ordinary variable whose name happens to be symbolic, and the built-in operators are exactly that already (`+` names the primitive that adds a pair). So an operator is introduced by a `let` carrying a fixity, and the two spellings below bind the same thing:

```
let infixr 0 a ⇒ b = !mkImpl (a, b) in     -- operands named
let infixl 8 (<+) = snoc in                -- an existing function
```

The first names the operands and takes the body directly. The second gives the operator a function, which must accept a *pair*, since `a OP b` always elaborates to `(OP) (a, b)` with the operands evaluated left to right. Writing a curried function there is a type error rather than a special case. Both forms are available under `let rec`, where the operator is in scope in its own body, and in a group of them, where every operator in the group is in scope in every body.

`infixl`, `infixr` and `infix` give a left-, right- and non-associative operator respectively, at the given precedence level, on the same 0 to 9 scale as the built-in operators above. Levels 0, 8 and 9 are unoccupied, so a binding there is either looser or tighter than everything built in; the remaining levels tie with the built-in group listed against them.

Because the fixity belongs to the binding, it has exactly the binding's scope. An operator may shadow a built-in one, and both its meaning and its associativity revert at the end of the enclosing computation:

```
let inner =
  let infixr 2 a - b = return 0 in
  1 - 2 - 3          -- 0, using the shadowing definition
in
return (1 - 2 - 3)   -- -4, subtraction again, left-associative
```

Operator names are sequences of the symbol characters `+ - * / < > = & | ^ @ % ? ~`. Exactly four spellings are unavailable, because the grammar needs them elsewhere: `->`, `<-`, `~>` and `=`. Everything else in that alphabet is free, including every built-in operator and `=>`, which a binding shadows in the ordinary way. Note that `:` is not in the alphabet, so `::` and type ascription stay unambiguous.

Using an operator with no fixity in scope is an error, and non-associative operators cannot be chained.

Parenthesising an operator gives back the plain variable, so operators can be applied prefix and passed as arguments:

```
(+) (3, 4)                              -- 7
fold ((+), (0, 1 :: 2 :: 3 :: []))      -- 6
```

With the `=>` binding above, Peirce's law is written

```
((p ⇒ q) ⇒ p) ⇒ p
```

which elaborates to `(⇒) ((⇒) ((⇒) (p, q), p), p)`, and so to three calls of `!mkImpl`.

8 Effects and Operations

Effects are the core feature of Cambria. An **effect operation** is invoked with the `!` prefix:

```
!print "hello"
!flip ()
!get a
!set (a, 42)
```

An operation invocation `!op arg` is a computation that performs the named effect and produces a result. The result depends on how the enclosing handler interprets the operation.

8.1 Built-in vs User-Defined Operations

Some operations are built-in and handled by the runtime (see Section 13). Operations can be:

1. **Declared** with `effect` to enforce a type signature, and
2. **Handled** by an enclosing `with ... handle ...` block

If an operation is invoked but no handler catches it, it propagates outward. If it reaches the top level and is not a built-in, the program reports an unhandled effect.

9 Handlers

Handlers are the mechanism for defining the meaning of effect operations. A handler intercepts operations performed within its body and provides implementations for them.

9.1 Handler Syntax

```
with [$p → Type] handler {
  return x → returnBody,
```

```
  opName pattern contName → opBody,
  finally pattern → finallyBody
} handle

computation
```

A handler is a value (of handler type) applied to a computation via `with ... handle ...`. The optional `[$p → Type, ...]` list between `with` and the handler instantiates the handler's abstract type parameters (Section 10); it is omitted when there are none.

The handled computation runs to the end of the enclosing computation (Section 7), so a stack of handlers reads as a prefix and needs no parentheses. Wrapping the body in `(...)` is still allowed, and is how a handler is closed off early.

9.2 Handler Clauses

A handler can contain three kinds of clauses, separated by commas. The ordering is not important.

9.2.1 Return Clause: `return x -> ...`

Transforms the final value produced by the handled computation:

```
handler {
  return x → return (x + 1)
}
```

If omitted, it defaults to the identity clause: `return x → return x`.

9.2.2 Operation Clause: `opName pattern contName -> ...`

Handles an effect operation. When `!opName arg` is invoked inside the handled computation:

- `pattern` binds the argument `arg`
- `contName` binds the **continuation**, which is a function representing "the rest of the computation after the operation"

```
handler {
  get a k → k (lookup (a, s)),
  set x k → k ()
}
```

Calling the continuation `k` with a value resumes the computation as if the operation returned that value. You can:

- Call `k` once (normal resumption)
- Call `k` multiple times (nondeterminism, backtracking)
- Not call `k` at all (abort, short-circuit)
- Call `k` inside another computation (transform the result)

9.2.3 Finally Clause: `finally f -> ...`

Transforms the entire result of the handler after the return clause has been applied:

```
handler {
  return x → return (fun _ → return x),
  get a k → return (fun s → k (lookup (a, s)) s),
  finally s → s empty
}
```

The finally clause is particularly useful for the "state-passing" pattern where the return clause wraps the result in a function, and `finally` applies that function to an initial state.

9.3 Handler as a Value

Handlers are first-class values. You can bind them to variables:

```
do h ← return (handler {
  return x → return x,
  decide v k → max (k true, k false)
}) in
with h handle ...
```

9.4 Nested Handlers

Handlers can be nested. The innermost handler for a given operation takes priority:

```
with outerHandler handle
with innerHandler handle
-- innerHandler's operations are checked first
...
```

If an inner handler does not handle an operation, it propagates to the outer handler. The inner handler automatically wraps the continuation to maintain its own handling scope (this is called "deep handling").

9.5 Deep Handling

When a handler catches an operation and calls the continuation `k`, the handler is automatically re-installed around the continuation. This means subsequent operations in the same computation are also caught by the same handler. This is the standard "deep handler" semantics.

10 Type Parameters (`$p`)

Type parameters are Cambria's mechanism for **abstract types**. Types whose concrete representation is hidden from the code that uses them but known to the handler that provides them.

10.1 Declaring Operations with Type Parameters

```
effect !ref : Int ⇝ $p.
effect !get : $p ⇝ Int.
```

```
effect !set : $p * Int ⇝ Unit.
```

Here `$p` is an abstract type. Code that uses these operations knows that `!ref` produces a `$p` and `!get` consumes a `$p`, but it cannot inspect or construct `$p` values directly. Attempting to use a `$p` value where a concrete type is expected (e.g. `a + 1` where `a : $p`) is a type error.

10.2 Instantiating Type Parameters in Handlers

A handler application instantiates a type parameter with a `[$p -> Type]` list placed between `with` and the handler:

```
with [$p → Name] handler {
  return x → ...,
  ref x k → do a ← !fresh () in k a,
  get a k → ...,
  set x k → ...
} handle

effect !ref : Int ⇝ $p.
effect !get : $p ⇝ Int.
effect !set : $p * Int ⇝ Unit.
...
```

When the handler instantiates `$p -> Name`, all occurrences of `$p` in the handled body's effect signatures are replaced with `Name` for type-checking purposes.

10.3 Multiple Type Parameters

Different parameters can be used for different abstract types:

```
with [$p → Name, $q → Int] handler { ... } handle ...
```

10.4 Nested Parameter Instantiation

A type parameter can be instantiated to a type involving another parameter:

```
with [$p1 → $p2 + Int] handler {
  ...
} handle ...
```

10.5 Parameter Coverage Checking

If a handler instantiates a type parameter, it **must** handle all operations whose signatures mention that parameter. Otherwise, the type checker rejects the program:

```
-- ERROR: handler instantiates $p but does not handle !set
with [$p → Name] handler {
  return x → return x,
  get a k → k 42,
  ref x k → k ()
} handle
```

```
effect !get : $p ⇝ Int.
effect !set : $p * Int ⇝ Unit.    -- not handled!
effect !ref : Int ⇝ $p.
...
```

The operations that would be left over are reported by name: `Forwarded operations ["set"] mention type parameters bound by this handler`. A handler that instantiates `$p` is the only place where `$p` has a meaning, so nothing mentioning it may be passed further out.

10.6 Parameter Escape Checking

The same reasoning applies to values, not just to operations. A `$p`-typed value may not escape the handler that instantiates `$p`. The type checker rejects a body that leaks one into the surrounding context:

```
-- ERROR: $p escapes through f, which comes from outside the handler
return (fun f →
  with [$p → Int] handler {
    return x → return x,
    get a k → k 5
  } handle
  effect !get : Unit ⇝ $p.
  do x ← !get () in
  f x)
```

11 Effect Declarations

11.1 `effect`

Declares the type signature of an effect operation for use in subsequent code:

```
effect !opName : ArgType ⇝ RetType.
```

The declaration must end with a `.` (dot) and is followed by the computation that uses the operation.

Examples:

```
effect !get : $p ⇝ Int.
effect !set : $p * Int ⇝ Unit.
effect !ref : Int ⇝ $p.
effect !ask : Unit ⇝ Int.
effect !urn : Unit ⇝ $p.
effect !sample : $p ⇝ Bool.
```

Declarations serve two purposes:

1. They tell the type checker the argument and return types of operations.
2. They declare if the operations use abstract types.

12 Built-in Functions

These are available in every program without any declaration.

12.1 Pair Operations

Function	Type	Description
`fst`	`a * b -> a`	First component
`snd`	`a * b -> b`	Second component

12.2 Arithmetic / Comparison

Function	Type	Description
`+`	`Int * Int -> Int`	Addition
`-`	`Int * Int -> Int`	Subtraction
`*`	`Int * Int -> Int`	Multiplication
`/`	`Int * Int -> Double`	Division
`max`	`Int * Int -> Int`	Maximum
`<`	`Int * Int -> Bool`	Less than
`>`	`Int * Int -> Bool`	Greater than
`<=`	`Int * Int -> Bool`	Less than or equal
`>=`	`Int * Int -> Bool`	Greater than or equal
`&&`	`Bool * Bool -> Bool`	Conjunction
`||`	`Bool * Bool -> Bool`	Disjunction
`==`	`a * a -> Bool`	Structural equality
`/=`	`a * a -> Bool`	Structural inequality

12.3 String Operations

Function	Type	Description
`++`	`Str * Str -> Str`	Concatenation
`hash`	`Name -> Str`	Convert a name to a string

12.4 List Operations

Function	Type	Description
`::`	`a * List a -> List a`	Prepend element
`null`	`List a -> Bool`	Is empty?
`[]`	`List a`	Empty list constant

Lists are taken apart by pattern matching on `[]` and `::` (Section 5); there is no destructor function in the surface.

12.5 Map Operations

Function	Type	Description
`empty`	`Map k v`	Empty map constant
`insert`	`(k * v) * Map k v -> Map k v`	Insert or update entry
`remove`	`k * Map k v -> Map k v`	Remove entry by key
`lookup`	`k * Map k v -> v`	Lookup by key (partial)
`member`	`k * Map k v -> Bool`	Key exists?

Note: `lookup` will error at runtime if the key is not found. Check with `member` first if unsure.

12.6 The Empty Type

Function	Type	Description
`absurd`	`Void -> a`	Anything follows from `Void`

An operation declared to return `Void` never resumes, so its result can be turned into whatever the surrounding code needs:

```
effect !stop : Unit ⇝ Void.
do v ← !stop () in absurd v
```

13 Built-in IO Effects

These operations are handled at the top level by the runtime and perform actual I/O:

Operation	Signature	Description
`!fresh ()`	`Unit ⇝ Name`	Fresh unique identifier
`!print s`	`Str ⇝ Unit`	Print string to stdout
`!read ()`	`Unit ⇝ Str`	Read line from stdin
`!flip ()`	`Unit ⇝ Bool`	Random boolean (fair coin)
`!bernoulli p`	`Double ⇝ Bool`	Random boolean with probability `p`
`!uniform ()`	`Unit ⇝ Double`	Random double in $[0, 1)$

These do not need to be declared; they are always available. They are caught at the outermost level after all user-defined handlers have had a chance to intercept them.

14 The Type System

Cambria uses an extended Hindley-Milner type inference system with effect types. The only annotation that must be supplied is the arity of any operation whose signature mentions an abstract parameter type (`$p`), which are introduced with an `effect` declaration. Explicit annotations are nonetheless available where desired.

14.1 Computation Types

Every computation has a type of the form:

```
ValueType!{effects}
```

For example:

- `Int!{}`: a pure computation producing an `Int`
- `Bool!{ flip : Unit ⇝ Bool }`: a computation that may invoke `!flip` and produces a `Bool`
- `Unit!{ print : Str ⇝ Unit, fresh : Unit ⇝ Name }`: may print and generate uniques

14.2 Effect Rows

Effects are tracked as a set of operation signatures. When operations are handled, they are removed from the effect set. A fully handled program has an empty effect set `{}` (plus any built-in IO effects that are handled by the runtime).

Effect rows can be either **closed** (a fixed set of operations) or **open** (a set of known operations plus an effect variable representing unknown additional effects). Open effect rows allow polymorphism over effects.

The two print differently. A closed row is just its operations between braces, and an open row adds a bar and the effect variable:

Printed	Meaning
`{}`	Closed and empty: no effects at all
`{ print : Str ⇝ Unit }`	Closed: exactly this operation
`{_e1}`	Open and empty: any effects
`{ print : Str ⇝ Unit | _e1 }`	Open: this operation and possibly more

14.3 Polymorphism

Functions defined with `do ... ← return (fun ...) in ...` or `do ... ← return (rec ...) in ...` are generalised so their types are polymorphic. The generalisation happens at the `do`-binding site.

```
do id ← return (fun x → return x) in
-- id has type: forall a. a → a!{e}
do _ ← id 42 in          -- instantiated at Int
do _ ← id true in         -- instantiated at Bool
return ()
```

Recursive functions are also polymorphic:

```
let rec map f xs = case xs of {
  []       → return [],
  x :: xs' → f x :: map f xs'
} in
-- map : forall a b. (a → b!S) → List a → List b!S
```

14.4 Type Parameter Abstraction

Type parameters (`$p`) are existentially quantified from the body's perspective. The body knows operations involve `$p` but cannot see its concrete type. Only the handler knows the instantiation.

This enables abstraction: you can write polymorphic functions that work over `$p` without knowing what it is:

```
-- This swap function works at both Int and $p
do swap ← return (fun (x, y) → return (y, x)) in
swap (1, 2)              -- at Int
do a ← !ref 100 in
do b ← !ref 200 in
swap (a, b)              -- at $p (Name, but swap doesn't know)
```

14.5 Handler Types

Handlers have types of the form:

$$\texttt{CompType}_{\texttt{in}} \texttt{ => } \texttt{CompType}_{\texttt{out}}$$

The input computation type includes the effects the handler catches. The output computation type is the type of the result after handling. Both rows are usually open, so the collecting handler of Section 18 has a type of the shape

```
(a!{ print :  Str ⇝ Unit | e } => (a * Str)!{e'})
```

A handler type records no parameter instantiation. The `[$p -> Type]` list belongs to the `with ... handle` application rather than to the handler value, so it is checked where the handler is applied, not where it is written.

Written out in an annotation, the fat arrow separates the two computation types. The handler below catches `!flip` and so discharges it, turning an `Int` computation that may flip into a pure one:

```
do h ← return ((handler {
  return x → return x,
  flip _ k → k true
}) : Int!{flip : Unit ⇝ Bool} ⇒ Int!{}) in
with h handle
do b ← !flip () in
if b then return 1 else return 2
```

15 Explicit Annotations

Cambria infers types without help, so annotations are never required except for the arity of an operation whose signature mentions a parameter type (Section 11). They are available as a way to pin a type or document intent.

An annotation is a parenthesised expression followed by `:` and a type.

15.1 Value Annotations

```
return (5 : Int)
do x ← return 7 in return (x : Int)
return ((inl 5) : Int + Bool)
```

The annotation sits on the expression, not on the surrounding computation, which is why the second line annotates `x` inside the `return` rather than the `do`-binding. Annotating a polymorphic

expression pins its instance. Without the annotation, `inl 5` would leave the right-hand summand undetermined.

A value annotation may be a function type, in which case the result is a computation type and carries a row:

```
do f ← return ((fun x → return x) : Int → Int!{}) in f 4
```

Annotating that same function `Int -> Bool!{}` is a type error, because the body returns its argument.

15.2 Computation Annotations

A computation type is a value type, `!`, and a braced row:

```
(return 3 : Int!{})
(return (1, true) : Int * Bool!{})
(!print "hello" : Unit!{print : Str ⇝ Unit})
```

The row is checked as well as the value type. A row that omits an effect the body actually performs is rejected, so `(!print "boom" : Unit!{})` fails with an effect mismatch, as do two nested annotations that disagree.

15.3 Open Rows in Annotations

A row written out in full is **closed**. The computation may only perform the effects it lists. Ending the row with `...` makes it **open**, so other operations are permitted:

```
(do _ ← !print "hi" in !fresh () : Name!{print : Str ⇝ Unit, ...})
```

The body also performs `!fresh`, which the closed form would reject; the ellipsis lets the row carry it, and the inferred type comes out with both operations.

The ellipsis stands for the row variable of Section 14, so an open annotation constrains the row from below rather than fixing it.

16 Desugaring

The parser produces a "sugared" AST that is desugared into a simpler core AST before type-checking and evaluation. Understanding desugaring helps explain how syntactic sugar works.

16.1 Multi-Argument Functions

```
fun x y z → body
```

desugars to:

```
fun x → return (fun y → return (fun z → body))
```

Each additional argument becomes a nested function that returns the next closure.

16.2 Multi-Argument Rec

```
rec f x y → body
```

desugars to:

```
rec f x → return (fun y → body)
```

16.3 Pair Patterns

```
do (x, y) ← comp in body
```

desugars to:

```
do _tmp ← comp in
do x ← _fst _tmp in
do y ← _snd _tmp in
body
```

Nested pair patterns work recursively:

```
fun ((a, b), c) → body
```

desugars to a function on a fresh variable, with projections extracted via `_fst` and `_snd`.

16.4 `let` Bindings

```
let x = e in body
let f p q = c in body
let infixl 8 p <+ q = c in body
```

desugar to:

```
do x ← return e in body
do f ← return (fun p q → c) in body
do (<+) ← return (fun (p, q) → c) in body
```

with `fun` replaced by `rec f` under `let rec`. There is no second rule for a computation on the right: it stands where a value is expected, so it is lifted by the same ANF conversion that lifts one anywhere else. An operator binding is the same `do`, binding the symbol as a variable that takes a pair; the fixity is consumed while desugaring and never reaches the core.

16.5 Mutual Recursion

```
let rec f x = c1, g y = c2 in body
```

desugars to a single recursive thunk returning the tuple of definitions, plus a wrapper binding for each name that fetches its component from the thunk and applies it (Bekic's construction):

```
do h ← return (rec h _ →
     return ( fun x → do (f, g) ← h () in c1
            , fun y → do (f, g) ← h () in c2 )) in
do f ← return (fun v → do (f, g) ← h () in f v) in
do g ← return (fun v → do (f, g) ← h () in g v) in
body
```

Each definition's body begins with the same fetch, which is what lets `c1` call `g` and `c2` call `f`. Because the definitions are separate components of the tuple rather than branches of one function, they need not share a type.

16.6 Multi-Clause `case`

A two-case `inl`/`inr` match on irrefutable sub-patterns is the kernel `case` construct directly. Anything else compiles to a decision tree: each case becomes a test that either enters the body or calls a fall-through thunk holding the remaining cases.

```
case xs of { [] → c1, y :: ys → c2 }
```

desugars to:

```
do _rest ← return (fun _ → {- case 2 -}) in
do _tmp ← _uncons xs in
case _tmp of {
  inl _ → c1,
  inr _ → _rest ()
}
```

List patterns go through `_uncons`, pair patterns through `_fst` and `_snd`, and literal patterns through `_eq` followed by an `if`. When every case fails the tree ends in `_matchfail`, which aborts.

`_fst`, `_snd` and `_eq` are internal aliases for `fst`, `snd` and ==. Desugaring goes through them rather than through the public names so that shadowing a primitive cannot change what a pattern means. `_uncons` and `_matchfail` have no public counterpart and exist only for desugaring. A leading underscore is not a legal identifier in the surface, so no program can bind or even mention one, which is also why the generated `_rest` and `_tmp` above cannot collide with a variable of the user's.

16.7 Infix Operators

All infix operators desugar to function applications on pairs:

```
x + y    ⟶  (+) (x, y)
x == y   ⟶  (==) (x, y)
x ++ y   ⟶  (++) (x, y)
x ::  y  ⟶  (::)  (x, y)
```

Declared operators are no different: `x <+ y` becomes `(<+) (x, y)`, a reference to the variable the binding introduced.

The parser does not resolve precedence itself. It collects a run of infix operators into a flat chain, and the desugarer reassociates the chain by precedence climbing against the fixities in scope, which is why a new operator needs no change to the grammar.

16.8 Semicolons

```
comp1 ; comp2
```

desugars to:

```
do _ ← comp1 in comp2
```

16.9 Inline Computations in Value Position

When a computation appears where a value is expected (e.g. as a function argument), it is lifted into a `do`-binding:

```
f (!get a)
```

desugars to:

```
do _tmp ← !get a in f _tmp
```

This is called **ANF conversion** (A-normal form) and happens automatically.

16.10 Default Handler Clauses

If no `return` clause is given, the default is `return x → return x`.

17 Evaluation Model

Cambria uses **call-by-value** evaluation with **deep effect handlers**.

17.1 Values vs Computations at Runtime

- **Values** are fully evaluated data: integers, closures, pairs, etc.
- **Computations** either produce a **pure result** or an **impure result** (an unhandled operation).

The evaluator returns one of:

- `Pure v`: the computation finished with value `v`
- `Impure op arg`: the computation performed operation `!op arg` and is suspended, waiting for a result via the continuation

17.2 Function Evaluation

- `fun x → body` creates a **closure** capturing the current environment.

- `rec f x → body` creates a closure with a self-reference for recursion (via a knot-tying `let rec`).
- Application substitutes the argument into the closure's environment and evaluates the body.

17.3 Handler Evaluation

When `with h handle c` is evaluated:

1. Evaluate `c` in the current environment.
2. If `c` produces `Pure v`:
   - Apply the handler's return clause: evaluate the return body with `v` bound to the return pattern variable.
3. If `c` produces `Impure op arg k`:
   - If the handler has a clause for `op`: evaluate the clause body with `arg` bound to the parameter pattern and `k` bound to a **deep-handled continuation** (a new closure that re-wraps the continuation with the same handler).
   - If the handler does not handle `op`: propagate the impure result upward, wrapping the continuation so the handler remains installed.

The **deep handling** of continuations means that when an operation clause calls `k value`, the handler is automatically re-installed around the remaining computation.

17.4 Finally Clause Evaluation

The `finally` clause behaves as a `do`-binding around the whole handler application, so

```
with handler { ..., finally f → body } handle c
```

evaluates like

```
do f ← (with handler { ... } handle c) in body
```

This is why the finally clause sees the fully-processed result of the handler.

17.5 State-Passing Pattern

The most common handler pattern threads state through continuations:

```
with handler {
  return x  → return (fun _ → return x),
  get a k   → return (fun s → k (lookup (a, s)) s),
  set x k   → return (fun s → k () (insert (x, s))),
  ref x k   → do a ← !fresh () in
                 return (fun s →
                   k a (insert (((a, x), s))))),
  finally s → s empty
} handle (...)
```

How it works:

1. **return** wraps the final value in a function that ignores state.
2. **get** returns a function that reads from state and passes it to the continuation.
3. **set** returns a function that updates state and passes `()` to the continuation.
4. **ref** allocates a fresh unique ID and returns a function that extends the state.
5. **finally** applies the resulting state-function to an initial empty map.

The result of the handler is a function *State* → *Result*. The `finally` clause seeds it with `empty`.

18 Complete Examples

18.1 Hello World

```
!print "Hello, world!"
```

18.2 Local Mutable State

```
with [$name → Int] handler {
  return x   → return (fun _ → return x),
  fresh _ k  → return (fun n → k n (n+1)),
  eq (a,b) k → k (a == b),
  finally f  → f 0
} handle

effect !fresh : Unit ⇝ $name.
with [$loc → $name] handler {
  return x    → return (fun _ → return x),
  get a k     → return (fun s → k (s a) s),
  set (a,x) k → return (fun s →
                  k () (fun b → if !eq (a, b) then return x else s b)),
  ref x k     → do a ← !fresh () in return (fun s →
                  k a (fun b → if !eq (a, b) then return x else s b)),
  finally s   → s (fun _ → return 0)
} handle

effect !ref : Int ⇝ $loc.
effect !get : $loc ⇝ Int.
effect !set : $loc * Int ⇝ Unit.
do a ← !ref 2 in
do b ← !ref 3 in
do _ ← !set (a, !get b) in
!get a
```

The state is encoded as a function from locations to values, and a separate outer handler supplies the fresh names used as locations. Each handler is a prefix of the computation it handles, so no parentheses are needed. The program allocates two references (holding 2 and 3), copies `b`'s value into `a`, and reads `a` back. Result: `Pure: 3`. This is `examples/local_state.cba`.

18.3 Nondeterministic Maximisation

```
with handler {
```

```
  decide v k → max (k true, k false)
} handle

do x1 ← if !decide () then return 15 else return 30 in
do x2 ← if !decide () then return 5  else return 10 in
return (x1 - x2)
```

The handler explores all branches and returns the maximum. The `decide` operation calls the continuation twice: once with `true`, once with `false`. Result: `Pure: 25` ($30 - 5$).

18.4 Collecting Output

```
with handler {
  return x → return (x, ""),
  print s k → do (v, acc) ← k () in
    return (v, s ++ acc)
} handle

!print "A"; !print "B"; !print "C"
```

Instead of printing to stdout, this handler collects all printed strings into an accumulator. Result: `Pure: ((), "ABC")`.

18.5 Polymorphic Functions with Abstract Types

```
with [$name → Int] handler {
  return x   → return (fun _ → return x),
  fresh _ k  → return (fun n → k n (n+1)),
  eq (a,b) k → k (a == b),
  finally f  → f 0
} handle

effect !fresh : Unit ⇝ $name.
with [$loc → $name] handler {
  return x    → return (fun _ → return x),
  get a k     → return (fun s → k (s a) s),
  set (a,x) k → return (fun s →
                  k () (fun b → if !eq (a, b) then return x else s b)),
  ref x k     → do a ← !fresh () in return (fun s →
                  k a (fun b → if !eq (a, b) then return x else s b)),
  finally s   → s (fun _ → return 0)
} handle

effect !ref : Int ⇝ $loc.
effect !get : $loc ⇝ Int.
effect !set : $loc * Int ⇝ Unit.

-- Polymorphic map: forall a b. (a → b!S) → List a → List b!S
let rec map f xs = case xs of {
  []       → return [],
  x :: xs' → f x :: map f xs'
} in
do vals ← 10 :: 20 :: 30 :: [] in

-- (1) map at (Int → $loc): allocate a ref for each number
```

```
do refs ← map (fun n → !ref n) vals in

-- (2) map at ($loc → Bool): read each ref and test it against 20
do checks ← map (fun n → !get n == 20) refs in

return checks
```

The same polymorphic `map` is used at two different type instantiations: `Int -> $loc`, allocating a reference per element, and `$loc -> Bool`, reading each reference and comparing it. `map` never inspects what `$loc` is. Result: `Pure: [False, True, False]`. This is `examples/poly_map.cba`.

18.6 Two Implementations of a Probabilistic Urn

A key strength of parametrized handlers is that the *same* abstract interface can be given completely different implementations by swapping the handler. The following two programs both implement Pólya's urn with the same body code and the same effect interface:

```
effect !newurn : Unit ⇝ $urn.
effect !sample : $urn ⇝ Bool.
```

`!newurn` creates a new urn and `!sample` draws a boolean sample from it. The abstract type `$urn` hides what an urn actually is. The shared body draws two samples and returns them as a pair:

```
do urn ← !newurn () in
do b1 ← !sample urn in
do b2 ← !sample urn in
return (b1, b2)
```

Implementation 1: Real-number parameter (`$urn -> Double`)

The simplest implementation instantiates `$urn` to `Double`. An urn is just a bias drawn once from the uniform distribution, and each sample is an independent Bernoulli trial with that bias:

```
with [$urn → Double] handler {
  newurn _ k → k (!uniform ()),
  sample p k → k (!bernoulli p)
} handle

effect !newurn : Unit ⇝ $urn.
effect !sample : $urn ⇝ Bool.
do urn ← !newurn () in
do b1 ← !sample urn in
do b2 ← !sample urn in
return (b1, b2)
```

The handler is flat: `newurn` draws a `Double` and `sample` passes it directly to `!bernoulli`. No state is needed. Result: a pair of booleans, e.g. `Pure: (False, True)`. This is `examples/prob_polya_bias.c`

Implementation 2: Pólya urn via local state (`$urn -> $loc`)

The Pólya urn implementation instantiates `$urn` to a location `$loc` of the local-state effect, storing a pair of counts (red, blue). Each sample draws a colour with probability proportional to the counts and then adds a ball of that colour, so the urn is biased towards the colour already drawn ("rich get richer"). It reuses the function-encoded local state from above, with its fresh-name handler outside that in turn:

```
with [$name → Int] handler {
  return x   → return (fun _ → return x),
  fresh _ k  → return (fun n → k n (n+1)),
  eq (a,b) k → k (a == b),
  finally f  → f 0
} handle

effect !fresh : Unit ⇝ $name.
with [$loc → $name] handler {
  return x    → return (fun _ → return x),
  get a k     → return (fun s → k (s a) s),
  set (a,x) k → return (fun s →
                  k () (fun b → if !eq (a, b) then return x else s b)),
  ref x k     → do a ← !fresh () in return (fun s →
                  k a (fun b → if !eq (a, b) then return x else s b)),
  finally s   → s (fun _ → return (0, 0))
} handle

effect !ref : Int * Int ⇝ $loc.
effect !get : $loc ⇝ Int * Int.
effect !set : $loc * (Int * Int) ⇝ Unit.
with [$urn → $loc] handler {
  newurn _ k → k (!ref (1, 1)),
  sample urn k →
    do (red, blue) ← !get urn in
    do isRed ← !bernoulli (red / (red + blue)) in
    do _ ← if isRed then !set (urn, (red + 1, blue))
                    else !set (urn, (red, blue + 1)) in
    k isRed
} handle

effect !newurn : Unit ⇝ $urn.
effect !sample : $urn ⇝ Bool.
do urn ← !newurn () in
do b1 ← !sample urn in
do b2 ← !sample urn in
return (b1, b2)
```

Several things to note:

- The **innermost handler** implements the urn on top of local state: `newurn` allocates a reference holding the initial counts `(1, 1)`, and `sample` reads the counts, draws a biased coin, updates the counts, and returns the result.
- It instantiates `$urn -> $loc`: the urn parameter becomes a location of the *enclosing* local-state handler, itself an abstract parameter.
- The **middle handler** provides the function-encoded local state (`$loc -> $name`), and the **outermost handler** supplies the fresh location names (`$name -> Int`).

- The **body code is identical** to Implementation 1. Only the handler stack changes.

This is `examples/prob_polya_state.cba`.

Both handlers induce the same distribution on the returned pair (Pólya's identity), so by parametricity the body cannot tell them apart.

19 Grammar Summary

```
program := computation

-- Values
value   := atom
         | inl atom | inr atom
         | fun patterns → computation
         | rec name patterns → computation
         | handler { clauses }

atom    := () | [] | integer | double
         | boolean | string
         | (expr, expr) | name

integer := digit+
double  := digit+ . digit+

-- A '-' glued to the digits is part of the
-- literal wherever an expression may start,
-- and a subtraction after an operand.  An
-- application argument takes parentheses:
-- f (-3), since f -3 is the subtraction f - 3.

-- Explicit annotations
expr    := ( expr : type )
comp    := ( comp : compType )

-- Computations
comp    := comp ; comp
         | return expr
         | do pattern ← comp in comp
         | if expr then comp else comp
         | case expr of { inl p → comp,
                          inr p → comp }
         | with paramInsts expr handle comp
         | effect !op : type ⇝ type . comp
         | expr expr
         | !op expr
         | expr BINOP expr

BINOP   := :: | || | && | == | ≠ | < | >
         | ≤ | ≥ | ++ | + | - | * | /

-- The trailing comp of do, if, case, with and effect
-- extends as far to the right as it can, across ';'.
-- Parenthesise a comp to end its scope early.

-- Patterns
pattern := name | _ | (pattern, pattern)
```

```
-- Handler clauses
clause  := return pattern → comp
         | opName pattern contName → comp
         | finally pattern → comp

-- Parameter instantiations (optional, between with and the handler)
paramInsts := [ paramList ]
paramList  := $param → type
            | $param → type , paramList

-- Types (in declarations and annotations)
type     := typeSum
          | typeSum → compType
          | compType ⇒ compType
typeSum  := typeProd
          | typeProd + typeProd
typeProd := typeAtom
          | typeAtom * typeAtom
typeAtom := Unit | Int | Bool | Double
          | Str | Name | Void
          | Map typeAtom typeAtom
          | List typeAtom
          | $param | (type)

compType := type ! { effList }
effList  := empty | ...
          | opSig | opSig , effList
opSig    := opName : type ⇝ type

-- A trailing ... makes the row open.
```

20 Comments

Line comments start with `--`:

```
-- This is a comment
return 42  -- inline comment
```

There are no block comments.