# Compact reference

Compact is a strongly statically typed, bounded smart contract language, designed to be used in combination with TypeScript for writing smart contracts for the three-part structure of Midnight, where contracts have the following components:

- a replicated component on a public ledger
- a zero-knowledge circuit component, confidentially proving the correctness of the former
- a local, off-chain component that can perform arbitrary code

Each contract in Compact can have four kinds of code:

- type declarations, to support all of the following
- declarations of the data that the contract stores in the public ledger
- declarations of `witness` functions, to be supplied in TypeScript
- `circuit` definitions, that serve as the operational core of a smart contract

A contract can also include code from external files, and it can declare and import modules.

Like TypeScript, Compact is an eager call-by-value language.



## Compact Types

Compact is **statically typed**: every expression in a Compact program has a static type. Named circuits and witnesses req annotation on each of their parameters, and they require a return type annotation. Anonymous circuit expressions can h

parameter and return type annotations. Constant binding statements can have an optional type annotation.

The language is **strongly typed**: the compiler will reject programs that do not type check. It will reject programs where a circuit or witness with a parameter type annotation is called with an incorrectly typed argument for that parameter. It will reject programs where a circuit with a return type annotation returns an incorrectly typed value. If an optional type annotation is omitted, the compiler will infer a type and it will reject programs where no such type can be inferred.

Types consist of built-in primitive types, user-defined types defined in the program, and generic type parameters in scope. When the term "type" occurs in this document without any other qualifier, it means either a primitive type, a user-defined type, or a generic type parameter in scope.

## Primitive types

The following are the primitive types of Compact:

- `Boolean` is the type of *boolean* values. There are only two values of `Boolean` type. They are the values of the expressions `true` and `false`.
- `Uint<m..n>`, where `m` is the literal `0` or a generic size parameter in scope and bound to `0`, and where `n` is a natural number literal or a generic size parameter in scope, is the type of *bounded unsigned integer* values between `0` and `n`, both inclusive. (The lower bound is currently required to be `0`.) `Uint` types with different bounds `0..n` are different types, although one may be a subtype of the other. In practice, there is a (large) maximum unsigned integer value determined by the zero-knowledge proving system. The Compact implementation will signal an error if a `Uint` type exceeds this maximum value.
- `Uint<n>`, where `n` is a non-zero natural number literal or a generic size parameter in scope and bound to a non-zero natural number, is the type of *sized unsigned integer* values with binary representations using up to `n` bits. This is the same type as `Uint<0..m>` where `m` is equal to `(2^n)-1`. Sized integer types can be seen as a convenience for programmers. `Uint<32>`, for

Feedback

example, can be more obvious and less error-prone than the equivalent `Uint<0..4294967295>`. Any Compact program that uses sized integer types can be rewritten to one that uses only bounded integer types.

- `Field` is the type of elements in the scalar prime field of the zero-knowledge proving system.
- `[T, ...]`, where `T, ...` are zero or more comma-separated types is the type of *tuple* values with element types `T, ...`. Note that tuples are heterogeneous: the element types can all be distinct. The *length* of a tuple type is the number of element types. Tuple types with different lengths are different types. Tuple types with the same lengths but where at least one element has different types are different types although one may be a subtype of the other.
- `Vector<n, T>`, where `n` is a natural number literal or else a generic size parameter in scope and `T` is a type, is a shorthand notation for the tuple type `[T, ...]` with `n` occurrences of the type `T`. Note that a vector type and the corresponding tuple type are two different ways of writing exactly the same type. Unless otherwise specified, type rules for vector types are derived from the rules for the corresponding tuple type.
- `Bytes<n>`, where `n` is a natural number literal or else a generic size parameter in scope, is the type of *byte array* values of length `n`. `Bytes` types with different lengths are different types. `Bytes` types are used in the Compact standard library for hashing. String literals in Compact also have a `Bytes` type, where `n` is the number of bytes in the UTF-8 encoding of the string.
- `Opaque<s>`, where `s` is a string literal, is the type of *opaque* values with tag `s`. The syntax of string literals in Compact is the same as in TypeScript. `Opaque` types with different tags are different types. Opaque values can be manipulated in witnesses but they are opaque to circuits. They are represented in circuits as their hash. The allowed tags are currently only `"string"` and `"Uint8Array"`.

## User-defined types

Users can define three kinds of types themselves: structures, enumerations, and contracts.

### Structure types

Structure types are defined by a declaration beginning with the keyword `struct`. Here are some examples:

```
struct Thing {
  triple: Vector<3, Field>,
  flag: Boolean,
}

struct NumberAnd<T> {
  num: Uint<32>;
  item: T
}
```

A *non-generic structure* declaration introduces a named structure type, such as `Thing` in the first example above. Each non-generic structure declaration introduces a distinct type, even when the content of the structure is identical to another.

Structure declarations can also be *generic structure* declarations, such as `NumberAnd` in the second example above. They have a non-empty list of comma-separated *generic parameter* names enclosed in angle brackets. The generic parameters of a generic structure declaration are in scope in its body.

Generic structure declarations do not introduce a type. To be used as a type, they must be *specialized* by providing a comma-separated list of *generic arguments*, e.g., `NumberAnd<Uint<8>>`. Generic arguments must be types, natural number literals, or a generic size parameter in scope. Generic structures must be fully specialized: the number of generic arguments must match the number of generic parameters.

- Specializations of the same generic structure to the same types are the same type.
- Specializations of the same generic structure to different types are different types.
- Specializations of different generic structures are always different types, even if the specializations are structurally equivalent.

- Specializations of generic structures are always different from non-generic structure types, even if the specialization is structurally equivalent to the non-generic type.

A structure declaration has a sequence of named fields which must be separated either by commas or by semicolons. Comma and semicolon separators cannot be mixed within a single structure declaration. A trailing separator is allowed, but not required.

Each structure field must have a type annotation.

Values of structure types are created with *structure creation* expressions. These consist of the structure type (so generic structures must be specialized), followed by a sequence of field values enclosed in curly braces (`{ }`). Field values can be given positionally, in the same order as they are declared in the `struct` declaration; or they can be named using the field names from the declaration. Named field values can appear in any order. Positional and named field values can be mixed in the same structure creation expression, but all the positional ones must come before any of the named ones. See Structure creation for the details.

Using the example declarations above, structure values could be created with `Thing {[0, 1, 2], true}` or `NumberAnd<Uint<8>> { item: 255, num: 0 }`.

Structures may not contain fields containing the same type as the structure, either directly or indirectly. For example, it is an error to use the following pair of declarations:

```
struct Even {
  predecessor: Odd
}

struct Odd {
  predecessor: Even
}

export circuit doesntWork(s: Even): Odd {
```

Feedback

```
  return s.predecessor;
}
```

### Enumeration types

Enumeration types are defined by a declaration beginning with the keyword `enum`. Here is an example:

```
enum Fruit { apple, pear, plum }
```

An enumeration declaration introduces a named enumeration type, such as `Fruit` in the example above. Each enumeration declaration introduces a distinct type.

An enumeration declaration has a sequence of named values separated by commas. A trailing separator is allowed but not required.

In the example above, the type `Fruit` has three values: `Fruit.apple`, `Fruit.pear`, and `Fruit.plum`.

### Contract types

As of this writing, declarations of contracts and the cross-contract calls they support are not yet fully implemented, but the keyword `contract` used to declare contracts is reserved for this use.

## Generic parameter references

Generic parameters are declared in generic module declarations, generic structure type declarations, generic circuit definitions, and generic witness declarations.
For generic modules, they are in scope within the module. For generic structures, they are in scope for the structure's fields. For generic circuits, they are in scope in the circuit's parameters, its return type annotation, and its body. In these scopes, a reference to a generic parameter (that is not otherwise shadowed by some other identifier binding) is either a type or a natural-number size.

## Subtyping and least upper bounds

There is a *subtyping* relation on Compact types. Informally, if a type `T` is a subtype of a type `S` then every value of type `T` is also a value of type `S` (equivalently, `S` is a supertype of `T`). In that case, Compact allows implicitly using a value of type `T` where a value of type `S` is expected, without any programmer-supplied conversion.

Subtyping is defined by the following rules:

- Any type `T` is a subtype of itself (subtyping is reflexive)
- `Uint<0..n>` is a subtype of `Uint<0..m>` if `n` is less than `m`
- `Uint<0..n>` is a subtype of `Field` for all `n`
- `[T, ...]` is a subtype of `[S, ...]` if the two tuple types have the length and each type `T` is a subtype of the corresponding type `S`

A circuit or witness can be called with argument expressions whose types are subtypes of the corresponding parameter type annotations. A constant binding statement with a type annotation can be initialized with an expression whose type is a subtype of the type annotation.

The *least upper bound* (with respect to subtyping) of a non-empty set of types {`T0`, ..., `Tn`} is a type `S` such that:

- `S` **is an upper bound:** `Ti` is a subtype of `S` for all `i` in the range 0..`n`, and
- `S` **is the least upper bound:** for all upper bounds `R` of the set of types {`T0`, ..., `Tn`}, `S` is a subtype of `R`.

Note that least upper bounds do not necessarily exist for all sets of types.

**Tuple and vector types:** We say that a tuple type `[T, ...]` with possibly distinct types `T, ...` "*has a vector type*" if the least upper bound `S` of the set of types {`T`, ...} exists. In that case, the tuple type has the vector type `Vector<n, S>` where `n` is the length of the tuple. Some operations over tuples (such as mapping and folding) require the tuple type to have a vector type.

Note that when a tuple type has a vector type, the tuple type is a subtype of the vector type. Perhaps surprisingly, vector types can be subtypes of tuple types as well. A vector type `Vector<n, T>` is a subtype of a tuple type `[S, ...]` if the tuple has length `n` and `T` is a subtype of each of the types `S, ...`. The means, for instance, that a vector could be passed to a circuit where a tuple is expected.

## Default values

Every type in Compact has a *default value* of that type. The default values are as follows:

- `Boolean`: the value of the literal `false`
- `Uint<0..n>` and `Uint<n>`: `0`
- `Field`: `0`
- `[T, ...]` where `T, ...` is a sequence of zero or more types: the tuple with the corresponding length, each of the default value of the corresponding type
- `Bytes<n>`: the byte array of length `n` with all zero bytes
- `Opaque<"string">`: an empty string, i.e., `""`
- `Opaque<"Uint8Array">`: a zero-length `Uint8Array`, i.e., `new Uint8Array(0)`
- structure types: the struct with all fields set to the default value of their type
- enumeration types: the first value listed in the declaration

## Representations in TypeScript

Compact's primitive types are represented in TypeScript as follows:

- `Boolean` - `boolean`
- `Field` - `bigint` with runtime bounds checks
- `Uint<n>` / `Uint<0..n>` - `bigint` with runtime bounds checks

Feedback

- `[T, ...]` - the TypeScript tuple type `[S, ...]` or else the TypeScript array type `S[]` with runtime length checks, where `S` is the TypeScript representation of the corresponding type `T`
- `Bytes<n>` - `Uint8Array` with runtime length checks
- `Opaque<"string">` - `string`
- `Opaque<"Uint8Array">` - `Uint8Array`

User-defined types are represented in TypeScript as follows:

- `enum` instances - a `number` with runtime membership checks
- `struct` instances with fields `a: A, b: B, ...` - an object `{ a: A, b: B, ... }` where `A`, `B`, ... are the TypeScript representations of the Compact types.

Note that other `Opaque` types are currently not supported.

# Include files

Compact supports code separation and namespaces through separate files and modules. The most basic of these is the statement

```
include "path/to/file";
```

which may appear at the top level of a source file or module. When encountered, the Compact compiler will search for a Compact source file at `path/to/file.compact` in the current directory and then relative to any of the directories in the `:`-separated environment variable `COMPACT_PATH`. This file *must* be found and will be included verbatim in place of the `include` statement.

# Modules, exports, and imports

Feedback

A module is a collection of definitions whose namespace is hidden from surrounding code. A module is defined with

```
module Mod1 {
  ...
}

module Mod2<T> {
  ...
}
```

By default, identifiers defined within the body of a module are visible only within the module, i.e., they are not exported from the module. Any identifier defined at or imported into the top level of a module can be exported from the module in one of two ways: (1) by prefixing the definition with the `export` keyword, or by listing the identifier in a separate `export` declaration. For example, the following module exports `G` and `S` but not `F`.

```
module M {
  export { G };
  export struct S { x: Uint<16>, y: Boolean }
  circuit F(s: S): Boolean {
    return s.y;
  }
  circuit G(s: S): Uint<16> {
    return F(s) ? s.x : 0;
  }
}
```

A module can be imported into another definition scope, bringing all its exported entries into that scope, potentially with a prefix. For instance:

```
module Runner {
  export circuit run(): [] {}
}
import Runner;
// run is now in scope
import Runner prefix SomePrefix_;
// SomePrefix_run is now in scope

module Identity<T> {
  export { id }
  circuit id(x: T): T {
    return x;
  }
}
import Identity<Field>;
// id is now in scope, with Field as type T
```

Compact's standard library can be imported by `import CompactStandardLibrary`. The standard library defines a number of useful types and circuits along with ledger ADTs such as `Counter`, `Map`, and `MerkleTree`.

When importing module `M`, if the program does not contain a visible module definition the compiler looks in the file system in the relative path of the current directory for `M.compact`. In the case where a module is defined in a different program (file), the program must only contain a top-level module definition. Otherwise, the compiler throws a static error stating that the program does not contain a single module definition. For example, the program `M.compact` below defines a module:

```
module M {
  export { F };
  export struct S { x: Uint<16>, y: Boolean }
  circuit F(x: S): Boolean {
    return S.y;
```

```
  }
}
// circuit cant_exists() : [] {}
// If cant_exists is uncommented, the compiler will throw an error when compiling
// test.compact
```

Then, `test.compact` imports `M`:

```
//module M {
//  export { G };
//  export struct S { x: Uint<16>, y: Boolean }
//  circuit G(x: S): Boolean {
//    return S.y;
//  }
//}
// If M is uncommented, the compiler will import this module and not the one
// defined in M.compact. In this case, the compiler will throw an error for
// exporting F.

import M;
export { F };
```

The import syntax allows the module to be identified by a string pathname. In this case, the compiler first looks for the imported module relative to the current directory (the path of the importing program) and then in the directories identified by `COMPACT_PATH`. Importing by a pathname allows importing multiple modules with the same name. For example, consider the program `M.compact`:

```
module M {
  export { F };
  export struct S { x: Uint<16>, y: Boolean }
  circuit F(x: S): Boolean {
```

```
    return S.y;
  }
}
```

And the program `A/M.compact`:

```
module M {
  export { F };
  export struct S { x: Uint<16>, y: Boolean }
  circuit F(x: S): Boolean {
    return S.y;
  }
}
```

And finally the program `test.compact` can export both `$F` and `A_F` but not `$G`:

```
module M {
  export { G };
  export struct S { x: Uint<16>, y: Boolean }
  circuit G(x: S): Boolean {
    return S.y;
  }
}

import "M" prefix $;
// this imports M.compact and not the module M defined above

import "A/M" prefix A_;

export { $F
```

```
  ,A_F
//  ,$G
// uncommenting this will result in an error
  };
```

# Top-level exports

The circuits exported at the top level of a contract (i.e., not merely exported from a module) are the entry points of the contract and may not take generic arguments. Although multiple circuits with the same name are allowed generally to support function overloading, it is a static error if more than one circuit with the same name is exported from the top level.

User-defined types exported from the top level of the main file can be used to describe the argument and return types of witnesses and exported circuits; these may accept generic arguments, but generic arguments not actually used as types are dropped in the exported type. For example:

```
export struct S<#n, T> { v: Vector<n, T>; curidx: Uint<0..n> }
```

is exported for use as a TypeScript type with the `T` parameter but not the `n` parameter, i.e.,:

```
export type S<T> = { v: T[]; curidx: bigint }
```

Ledger field names exported from the top level are visible for direct inspection by code outside of the contract via the generated TypeScript `ledger()` function.

# Circuits

The basic operational element in Compact is the `circuit`. This corresponds closely to a function in most languages but is compiled directly into a zero-knowledge circuit. Circuits are declared as:

```
circuit c(a: A, b: B, ...): R {
  ...
}

circuit id<T>(value: T): T {
  return value;
}
```

where `A`, `B`, ..., and `R` are types, and `a`, `b` are parameters. The circuit body itself is a sequence of statements, and each path through the body must end with a `return` statement, unless the return type is `[]`.

## Pure and impure circuits

A Compact circuit is considered *pure* if it computes its outputs from its inputs without reference to or modification of public state (via the ledger) or private state (via witnesses). In practice, the compiler considers a circuit to be impure if the body of the circuit contains a ledger operation, a call to any impure circuit, or a call to a witness.

Some external circuits defined in `CompactStandardLibrary` are witnesses; calls to these make the caller impure. The remainder are considered pure, so calls to those do not make the caller impure.

A Compact program can declare a circuit to be pure by prefixing the circuit definition with the `pure` modifier, which must follow the `export` modifier, if present, e.g.:

```
pure circuit c(a: Field): Field {
  ...
```

```
}

export pure circuit c(a: Field): Field {
  ...
}
```

The only effect of the `pure` modifier is that the compiler will flag the declaration as an error if its own analysis determines that the circuit is actually impure. The pure modifier allows an application to ensure that the circuit will be present in the `PureCircuits` type declaration and via the `pureCircuits` constant in the TypeScript module produced for a (correct) Compact program by the Compact compiler.

## Statements

A statement may be

- a `for` loop
- an `if` statement
- a `return` statement
- an `assert`
- a block - a sequence of statements in a nested scope, enclosed by curly braces
- a `const` binding statement
- an expression

### `for` loop

A `for` loop repeats for a fixed number of iterations, using one of the two syntaxes below:

```
for (const i of <vector>) <statement>

for (const i of <lower>..<upper>) <statement>
```

### `if` statement

An `if` statement is of one of the following forms:

```
if (testexpr)
  <statement>

if (testexpr)
  <statement>
else
  <statement>
```

### `return` statement

If a circuit's return type is `[]`,

```
return;
```

is a valid statement. Otherwise,

```
return <expr>;
```

for an expression `<expr>` of the declared return type is a valid return statement.

### `assert` statement

An assertion can be made with

```
assert(<expr>, "what constraint was violated");
```

where `<expr>` should evaluate to a `Boolean`. If `<expr>` evaluates to true, the assertion succeeds; if `<expr>` evaluates to false, it fails with the given message. Each assertion is checked at runtime and constrained in-circuit.

### `const` binding statement

A new constant can be brought into scope with

```
const x = <expr>;
```

A `const` binding statement can bind multiple constants. The constant bindings are performed from left-to-right.

```
const x = <expr>, y = <expr>;
```

A variable bound by a constant binding cannot be referenced before it is initialized.

```
circuit c(): Field {
  // const y = x; // rejected by Compactc
  const x = 1, y = x;
  // const y = x, x = 1; // rejected by Compactc
  return y;
}
```

Each of the constant bindings may optionally be annotated with a type, in which case the type is checked with the expression.

```
const x: T = <expr>;
```

```
const x: T = <expr>, y = <expr>;
```

The name of a constant may not be reused within a block, and constants cannot be reassigned, although they can be shadowed in a nested block:

```
circuit c(): Field {
  const answer = 42;
  // const answer = 12, answer = 1; // rejected by Compactc
  {
    const answer = 12;
    assert answer != 42 "shadowing didn't work!";
  }
  return answer; // returns 42
}
```

Constant initializer expressions are evaluated when the binding statement is executed.

## Expressions

This section describes the syntax of Compact expressions and provides their static typing rules and their evaluation rules.

The syntax of expressions is given by an EBNF grammar. We use the following notational conventions in the grammar:

- Terminals are in **`bold monospaced`** font
- Non-terminals are in *emphasized* font
- Alternation is indicated by a vertical bar (`|`)
- Repetition of zero or more items is indicated by enclosing them in curly braces (`{` and `}`)
- Optional items are indicated by enclosing them in square brackets (`[` and `]`)

A Compact expression either has a static type, or else it contains a type error. The static type of an expression, if it has one, is either a Compact type or else a ledger state type. If an expression in a program contains a type error, it means that the Compact compiler will not compile that program. Subexpressions of Compact expressions are always required to be well-typed (free from type errors).

Every Compact expression either evaluates to a value, or else it raises an exception. The evaluation of an expression is defined in terms of the evaluation of its subexpressions. If the evaluation of a subexpression raises an exception, then the evaluation of the containing expression will stop and raise the same exception.

### Literals

Compact has syntax for boolean, numeric, and string literal expressions.

```
expr → true
     | false
     | nat
     | str
     | pad ( nat , str )
```

**Boolean literals** are the reserved words `true` and `false`. The static type of a boolean literal is `Boolean`. It evaluates to one of the two boolean values.

Feedback

**Numeric literals** are non-negative decimal integers. They are either the number `0` or a sequence of digits starting with a non-zero digit. The static type of a natural number literal `n` is `Uint<0..n>`.

There is an implementation-defined maximum unsigned integer value. A numeric literal larger than this value will have an invalid static type `Uint<0..n>` for some `n` larger than the maximum unsigned integer. This expression will be a static type error unless the literal is smaller than the maximum `Field` value and is used in a type cast expression of the form `e as Field`.

A natural number literal evaluates to the unsigned integer value that it denotes.

**String literals** use TypeScript string literal syntax. Note that they can therefore be enclosed in either single (`'`) or double (`"`) quotes, and they can contain escaped characters. Strings are represented by their UTF-8 encoding. The **length** of a string is the length of its UTF-8 encoding. The static type of a string literal is `Bytes<n>` where `n` is the length of the string. It evaluates to a byte array containing its UTF-8 encoding.

In addition, the expression `pad(n, s)` is a string literal, where `pad` is a reserved word, `n` is a natural number literal and `s` is a string literal whose length must be less than or equal to `n`. The static type of a padded string literal `pad(n, s)` is `Bytes<n>`. It evaluates to a byte array containing the UTF-8 encoding of `s`, followed by `0` bytes up to the padded length `n`.

### Variable references

The syntax of Compact identifiers is the same as the syntax of TypeScript identifiers. A variable is an identifier that is bound as a parameter in a circuit declaration or else as a constant in a `const` binding statement.

```
expr → id
```

**Circuit parameters** are bound by circuit declarations. The static type of a circuit parameter reference is the declared static type given by the parameter declaration's type annotation. It evaluates to the value of the corresponding argument expression that was passed to the circuit call.

Feedback

**Constants** are bound by `const` binding statements. If the binding statement has a type annotation, then the constant reference's static type is the declared static type given by the type annotation, and the type of the right-hand side initializer expression must be a subtype of the declared type. If the binding statement does not have a type annotation, then the constant reference's static type is the inferred static type (that is, the type of the binding statement's initializer expression). A constant reference evaluates to the value of the binding statement's initializer expression.

Note that constant references can have ledger state types because they can be initialized with a ledger state type's default value.

### Default values of a type

Every Compact type and ledger state type has a default value.

```
expr → default < type >
     | default < ledger-adt >
```

The expression `default<T>`, where `T` is a Compact type or a ledger state type, has static type `T`. It evaluates to the default value of that type.

Note that default value expressions can have ledger state types.

### Circuit and witness calls

Circuits and witnesses, collectively referred to as functions, are called via an expression of the form `f(e, ...)`, where `f` is a function and `e, ...` is a sequence of zero or more comma-separated argument expressions.

```
expr      → fun ( [expr {, expr}] )
fun       → id [targs]
          | ( [var-or-arg {, var-or-arg}] ) : [type] => body
          | ( fun )
```

Feedback

```
targs       → < [targ {, targ}] >
targ        → nat
            | type
var-or-arg  → id
            | arg
arg         → id : type
body        → block
            | expr
```

The function expression can take several different forms:

A **function name** is the name of a circuit or witness from a circuit or witness declaration in scope.

An **anonymous circuit** is an inline circuit definition having the form `(P, ...) => body` or `(P, ...): R => body`, where `P, ...` are zero or more comma-separated parameter declarations, `R` is an optional return type annotation, and `body` is the circuit's body. Each parameter consists of an identifier `x` (the parameter's name) and an optional type annotation `: T` where `T` is a Compact type. The optional return type `R` is a Compact type. The body is either a block (a sequence of zero or more semicolon-delimited statements enclosed in curly braces (`{ }`)), or an expression.

There is no syntax for generic anonymous circuits. This is because circuits are not first-class values: they cannot be passed around or stored in data structures, they *must* be called. And generic circuits must be specialized to call them, so anonymous generic circuits would have to be immediately specialized. In that case, the programmer can just write the non-generic version themselves.

A **parenthesized function** has the form `(f)` where `f` is a function expression, that is either a function name, an anonymous circuit, or a parenthesized function.

Because functions are not first class, parameter names and constant names are not allowed as the function part of a call. Nor are arbitrary expressions allowed, for the same reason.

Because circuits and witnesses are not first class, parameter names and constant names are not allowed as the circuit or witness part of a call. Nor are arbitrary expressions allowed, for the same reason.

Generic functions cannot be called without explicitly specializing them with generic arguments enclosed in angle brackets. Calls to generic functions must be fully specialized: the number of generic arguments must match the number of generic parameters.

The *underlying function* of a function expression is a non-parenthesized function expression. For a function name it is the function name, for an anonymous circuit it is the anonymous circuit, and for a parenthesized function it is the underlying function of the parenthesized function expression.

Type checking a function call depends on the form of the underlying function.

- **For a named function:** Function names may be *overloaded*: more than one function of the same name may be present in the scope of a call to that function. A call whose underlying function is a name can thus have zero or more candidate functions, i.e., all of those that have the given name and are present in the scope of the call.

  A candidate function is not necessarily compatible with the number and kinds of the generic parameter values, nor with the number and types of the argument expressions provided at the call site. It is compatible if and only if the number of generic parameters is the same as the number of generic arguments, each generic argument is of the required kind (a size or a type), the number of declared parameters is the same as the number of argument expressions, and if the declared type of each parameter is a supertype of the corresponding argument expression. If exactly one candidate is compatible, the call is made to that candidate. It is a static type error otherwise (if there is not exactly one compatible candidate).

  The static type of a call to a named function is the return type of the called function.

- **For an anonymous circuit:** Parameters that have type annotations will be type checked, and parameters that do not have type annotations will have types inferred. The argument subexpressions are type checked and then:

  - If there is a type annotation for a parameter, it is a static type error if the type of the corresponding argument is not a subtype of the type annotation.

- If there is no type annotation for a parameter, the parameter's type is inferred as the type of the corresponding argument expression.

If there is a return type annotation, then the circuit's body is type checked under the assumption that the parameters have their declared or inferred types. It is a static type error if there is a returned subexpression whose static type is not a subtype of the return type annotation. A return statement of the form `return;` is type checked as if it returned a value of type `[]`. Every control flow path through a body that does not explicitly end with a return statement implicitly ends with one of the form `return;`.

If there is no return type annotation, then a return type is inferred from the body. The body is type checked under the assumption that the parameters have their declared or inferred types. The inferred return type is the least upper bound of the types of all the returned subexpressions, with the same treatment of `return;` and control-flow paths that do not end in an explicit return as described above.

The static type of a call to an anonymous circuit is the declared or inferred return type.

Calls are evaluated by evaluating the argument expressions in order from left to right. Then, if a circuit is being called, the statements in the body of the circuit are executed with the parameter names bound to the corresponding argument values. The value of a circuit call is the value returned from the execution of the body. If a witness is being called, the contract will invoke the TypeScript or JavaScript witness function with the argument values. The value of a witness call is the value returned by the witness function.

### Structure creation

Structure values are created with structure creation expressions. The expression `S {f, ...}` is a structure creation expression, where `S` is a structure type and `f` is a sequence of zero or more comma-separated field value specifiers.

A field value specifier can be one of three things:

- A **positional** field value is an expression. Evaluating the expression gives the value for the field. Positional field values must be given in the order that fields are declared in the corresponding structure declaration.

- A **named** field value is of the form `id: e` where `id` is a field name and `e` is an expression. Evaluating the expression gives the value for the corresponding named field. Named fields can appear in any order. If named and positional fields are mixed, all the named fields must appear after all the positional fields.
- A **spread** expression is of the form `...e` where `...` is the literal three dots (ellipsis) token and `e` is an expression. Evaluating the expression must give a value of the same structure type as the one being created. The fields of the created structure are given values from the spread structure as described below. If there is a spread expression, it must occur as the first field value specifier and all other specifiers must be named field values.

```
expr       → tref { [new-field {, new-field}] }
new-field  → expr
           | id : expr
           | ... expr
tref       → id [targs]
```

The examples below demonstrate the use of positional and spread field values:

```
struct S { a: Uint<32>, b: Boolean, c: Bytes<8> }
circuit f(x: Uint<32>, y: Boolean, z: Bytes<8>): S {
  const s1 = S { c: z, a: x, b: y };
  // Alternatively, s1 can be created with the positional syntax S { x, y, z }
  // or a mix of positional and named field values S { x, c: z, b: y }.

  const s2 = S { ...s1, b: true };
  // s2 is created using the spread syntax.  So, s2 has the same field values
  // as s1 except that b is true.

  const s3 = S { ...s2, c: 'abcdefgh' };
  // s3 is also created using the spread syntax.  s3 has the same field values
  // as s2 except that c is 'abcdefgh'.
```

Feedback

```
  return s3;
}
```

The structure type must come from a structure declaration in scope. If the structure is generic, then it must be explicitly specialized with generic arguments enclosed in angle brackets. Generic structures must be fully specialized: the number of generic arguments must match the number of generic parameters.

The static type of a non-generic structure creation expression is the named structure type.

The generic arguments to a generic structure can be types, natural number literals, or the names of generic parameters in scope. A generic structure creation expression is type checked by substituting the generic arguments for the generic parameters in the structure's definition. The static type of a generic structure creation expression is a structure with the same name as the generic type and field types obtained by substituting the generic arguments for the generic parameters in the structure's declaration.

**If there is no spread field specifier:** It is a static error if the number of field specifiers does not match the number of fields in the corresponding structure declaration (a value must be given for every field). It is a static error if a named field specifier occurs before a positional field specifier. It is a static error if a field name occurs more than once, and it is a static error if a field name occurs that is not the name of a field in the corresponding structure declaration. It is a static type error if the type of a positional field subexpression is not a subtype of the declared type of the (positionally) corresponding field in the structure declaration. It is a static type error if the type of a named field subexpression is not a subtype of the declared type of the corresponding (named) field in the structure declaration.

**If there is a spread field specifier:** It is a static error if the spread field specifier does not come first in the sequence. It is a static type error if the type of the spread subexpression is not the same type as structure to be created. It is a static error if there are any positional field specifiers. It is a static error if a field name occurs that is not the name of a field in the corresponding structure declaration. It is a static type error if the type of a named field subexpression is not a subtype of the declared type of the corresponding (named) field in the structure declaration.

A structure creation expression is evaluated by evaluating the field specifier subexpressions in order from left to right. The result is a structure value whose fields have values based on the corresponding field specifier: if there is a positional or named field specifier for the field, the field value is the value of the subexpression, otherwise there must be a spread expression and the field value is the value of the corresponding field in the (structure) value of the spread subexpression.

### Tuple creation

Tuple values are created with expressions of the form `[e, ...]` where `e, ...` is a sequence of zero or more comma-separated argument expressions. A non-empty sequence can have an optional trailing comma. The *length* of a tuple is the number of subexpressions in the tuple creation expression.

```
expr → [ [expr {, expr}] [,] ]
```

The static type of a tuple creation expression is `[T, ...]` with the number of types matches length of the tuple and each type `T` is the type of the corresponding expression.

It is evaluated by evaluating the subexpressions from left to right. Its value is a tuple whose length is the number of subexpressions and whose elements are the subexpression values.

### Parenthesized expressions

Compact allows parenthesized expressions of the form `(e)`, where `e` is an expression. They can be used to control the order of evaluation.

```
expr → ( expr )
```

The type of the parenthesized expression is the type of the subexpression.

Its value is the value of the subexpression.

### Sequence expressions

Expressions can be sequenced for their side effects. An expression of the form `(e0, e1, e2, ...)` or `e0, e1, e2, ...` where `e0, e1, e2, ...` is a sequence of two or more comma-separated expressions is a sequence expression. The latter form where parentheses can be omitted is only allowed in some context which can be viewed in the formal grammar of Compact.

```
expr → ( expr {, expr} , expr )
     | expr {, expr} , expr
```

The static type of a sequence expression is the static type of the last subexpression.

It is evaluated by evaluating the subexpressions from left to right. Its value is the value of the last subexpression.

### Ledger expressions

A Compact program interacts with its public state by invoking operations upon the ledger or ledger state types. There are two different forms of ledger calls.

**Kernel operations** are operations that do not depend on specific ledger state. They can be invoked by expressions of the form `k.op(e, ...)`, where `k` is the name of a ledger field declared to have the special ADT type `Kernel`, `op` is the name of a builtin kernel operation and `e, ...` is a comma separated sequence of zero or more argument expressions. The `CompactStandardLibrary` predefines the ledger field name `kernel` to have ledger type `Kernel`, so for example, the built-in `self` operation can be called from a circuit as follows:

```
import CompactStandardLibrary;
circuit f(): ContractAddress {
  return kernel.self();
}
```

**Ledger ADT operations** are operations on the program's public ledger state. They are invoked by expressions of the form `member.op(e, ...)...`, where `member` is a ledger field name (declared via a `ledger` field declaration) and `.op(e, ...)...` are a sequence of zero or more ledger ADT operation invocations, where each `op` is the name of a ledger ADT operation and each `e, ...` is a sequence of zero or more comma-separated argument expressions.

```
expr            → ledger ledger-accessor
                | ledger . id {ledger-accessor}
ledger-accessor → . id ( [expr {, expr}] )
```

Note that ledger ADT operations can be chained, because the result of a ledger ADT operation might itself have a ledger state type. Kernel operations cannot be chained, because the kernel is not a ledger state type and is not returned by any operation.

The static type of a kernel operation expression is the return type of the corresponding kernel operation according to the ledger data types reference.

The static type of a ledger ADT operation expression is the return type of the corresponding ADT operation according to the ledger data types reference. Note that this might be a Compact type or it might be a ledger ADT type. Values of ledger ADT types can have ADT operations invoked (immediately) on them, but any other use would be a static type error.

Kernel operations are evaluated by evaluating the argument subexpressions in order from left to right and then invoking the corresponding kernel operation with the argument values.

A ledger ADT operation `member` (that is not followed by a ledger accessor) is implicitly an invocation of the `read` operation. It is evaluated as if it were `member.read()`.

A ledger ADT operation `member.op(e, ...)` is evaluated by evaluating the argument subexpressions from left to right, and then invoking the operation `op` on the public ledger member `member` with the argument values.

Feedback

A ledger ADT operation `adt.op(e, ...)` where `adt` is itself a ledger ADT operation is evaluated by first evaluating `adt`, then evaluating the argument subexpressions from left to right, and then invoking the operation `op` on the ledger ADT value of `adt` with the argument values.

### Element and member access expressions

Compact has expressions for accessing the elements of tuple values and the members of structure values. An expression of the form `e[n]` where `e` is an expression and `n` is a numeric literal is a tuple element access. An expression of the form `e.id` where `e` is an expression and `id` is the name of a structure member is a structure member access.

```
expr → expr [ nat ]
     | expr . id
```

Tuple element accesses are type checked by checking the type of the subexpression. It is a type error if this type is not a tuple type `[T, ...]`. It is a type error if the length of the tuple is less than or equal to the numeric literal in the expression. The type of the expression is the corresponding element type `T` at position `n` in the tuple type.

Tuple element accesses are evaluated by evaluating the subexpression. The value of the expression will be the element value at the given zero-based index. The subexpression will have a tuple value and the element access will not be out of bounds, because the expression is well-typed.

Member access expressions are type checked by checking the type of the subexpression. It is a type error if this type is not a structure type. It is a type error if the structure type does not contain a member with the same name as the name in the expression. The type of the expression is the type of the corresponding named member of the subexpression's structure type.

Member access expressions are evaluated by evaluating the subexpression. The value of the expression will be the member value with the given name. The subexpression will have a structure value and the name will exist, because the expression is well-typed.

### Boolean negation expressions

Compact has unary boolean negation expressions of the form `!e` where `e` is an expression.

```
expr → ! expr
```

A boolean negation expression is type checked by checking the type of the subexpression. It is a type error if this type is not `Boolean`. The type of the expression will be `Boolean`.

Negation expressions are evaluated by evaluating the subexpression. The value of the expression will be `true` if the value of the subexpression is `false` and vice versa. The subexpression will have a boolean value because the expression is well-typed.

### Binary arithmetic expressions

Binary arithmetic expressions are of the form `e0 op e1` where `e0` and `e1` are expressions and `op` is one of Compact's binary arithmetic operators. The binary arithmetic operators are **add** (`+`), **subtract** (`-`) and **multiply** (`*`).

```
expr → expr + expr
     | expr - expr
     | expr * expr
```

Arithmetic expressions require the type of both subexpressions to be numeric types, that is, either a `Field` or a `Uint`. The type of the result will depend on the types of the subexpressions as follows:

- If either subexpression has type `Field`, the result will have type `Field`
- Otherwise the left subexpression will have type `Uint<0..m>` and the right subexpression will have type `Uint<0..n>` for some bounds `m` and `n`. The type of the result depends on the operation as follows:
  - For add, the result will have type `Uint<0..m+n>`
  - For subtract, the result will have type `Uint<0..m>`
  - For multiply, the result will have type `Uint<0..m*n>`

For arithmetic operations with `Uint` result types, it is a static type error if the result's bound would be greater than the maximum unsigned integer.

Arithmetic expressions are evaluated by first evaluating the subexpressions in order from left to right. Integer addition, subtraction, or multiplication is then used on the subexpression values. The overflow and underflow behavior differs for `Field` and `Uint` operations:

- `Field` arithmetic overflow and underflow wraps around 0; that is, the result of an arithmetic operation whose result is a `Field` is the actual arithmetic value modulo `k`, where `k` is one more than the maximum field value.
- `Uint` addition and multiplication cannot overflow: the static type of the result will always be large enough to hold the result value
- `Uint` subtraction checks if the value of the right subexpression is greater than the value of the left subexpression. If so, it is a runtime error (the result would be negative). Otherwise the unsigned subtraction is performed.

The static typing rules imply that if `Field` arithmetic semantics is desired, then at least one of the operands must have static type `Field`.

### Type cast expressions

Type cast expressions in Compact are of the form `e as T` where `e` is an expression, `as` is a reserved words, and `T` is a Compact type.

TypeScript-style casts of the form `<T>e` are not supported in Compact.

*expr* → *expr* **as** *type*

Type cast expressions are type checked by checking the type of the subexpression. If the cast from the subexpression's type to the type `T` named in the type cast is allowed, then the static type of the expression will be `T`. Otherwise, it is a static type error.

Upcasts, i.e., casts from a type to a supertype, are allowed but never required and never result in a static or run-time error.

Feedback

The table below describes the allowed type casts. Casting between types not shown in the table is not allowed. The entries in the table can be one of:

- **static:** the type cast only changes the static type and does not have any effect at runtime
- **conversion:** the type cast always succeeds but has the runtime effect of converting between different source and target representations, which normally has a low cost
- **checked:** the type cast is checked at runtime and can fail
- **no:** the type cast is not allowed
- a number: see the corresponding note below the table

| | | TO | | | |
|---|---|---|---|---|---|
| | | `Field` | `Uint<0..n>` | `Boolean` | `Bytes<n>` |
| FROM | `Field` | static | checked | 1 | 2 |
| | `Uint<0..m>` | static | 3 | conversion | no |
| | **enum type** | conversion | no | no | no |
| | `Boolean` | conversion | 4 | no | no |
| | `Bytes<m>` | 5 | no | no | 6 |

1. `Field` to `Boolean`: the value `0` is converted to `false` and all other values are converted to `true`.
2. `Field` to `Bytes<n>`: the value of the field is converted into `Bytes` of the given length, with the least-significant byte of the field occurring first in the `Bytes`. The `Bytes` will be padded to the length with trailing zeros. It is a runtime error if the f

not fit in the length.

3. `Uint<0..m>` to `Uint<0..n>`: if `m` is less than or equal to `n` this is a static cast. Otherwise it is checked and will fail at runtime if the value is greater than `n`.
4. `Boolean` to `Uint<0..n>`: If `n` is not 0 then this is a conversion of `false` to 0 and `true` to 1. Otherwise, it is checked and will fail at runtime if the value is `true` (and convert `false` to 0).
5. `Bytes<m>` to `Field`: the bytes are converted into a field with the least-significant byte of the field occurring first in the `Bytes`. It is a runtime error if the result would exceed the maximum `Field` value.
6. `Bytes<m>` to `Bytes<n>`: the cast is a static cast if `m` equals `n`, and is not allowed otherwise.

Allowed type casts are evaluated by first evaluating the subexpression. Then, if the cast is static, the result is the subexpression's value interpreted as the type `T` mentioned in the cast expression. If the cast is a conversion, the JavaScript representation of the subexpression's value is converted into the representation of a value of type `T`. If the cast is checked, the check is performed before conversion and the cast fails (at runtime) if the check fails. The exceptions noted in the table are evaluated as described above after evaluating the subexpression.

### Relational comparison expressions

Relational comparison expressions are of the form `e0 op e1` where `e0` and `e1` are expressions and `op` is one of Compact's relational operators. The relational operators are **equals** (`==`), **not equals** (`!=`), **less than** (`<`), **greater than** (`>`), **less than or equals** (`<=`), and **greater than or equals** (`>=`).

```
expr → expr == expr
     | expr != expr
     | expr < expr
     | expr > expr
     | expr <= expr
     | expr >= expr
```

Equals and not equals require the types of the subexpressions to be in the subtype relation. That is, the type of the first subexpression must be a subtype of the type of the second subexpression, or else the type of the second subexpression must be a subtype of the type of the first subexpression.

Less than, greater than, less than or equals, and greater than or equals require the type of both subexpressions to be unsigned integer types (note that `Field` cannot be compared with these operators).

The type of the result is `Boolean`.

Relational comparison expressions are evaluated by evaluating the subexpressions in order from left to right. Then the comparison is performed as described below.

### Equals

The comparison that is performed depends on the type of the operands:

- **`Boolean`:** if the operands have type `Boolean`, then the values must be the same boolean value. Both operands will have type `Boolean` due to the static typing rules.
- **`Uint`:** if the operands have unsigned integer types, then the integer values must be equal. Both operands will have unsigned integer types due to the static typing rules.
- **`Field`:** if either operand has type `Field`, then the integer values of the operands must be equal. Both operands will have numeric (`Field` or unsigned integer) types due to the static typing rules.
- **`Bytes:`** if the operands have bytes types, then the corresponding bytes at each index must be equal. Both operands will have bytes types and their lengths will be equal due to the static typing rules.
- **`Tuple:`** if the operands have tuple types, then the corresponding element values at each index must be equal according to these rules, based on the static element types. Both operands will have tuple types, they will have the the same length, and their element types will be in the subtype relation due to the static typing rules.

- `Opaque :` if the operands have opaque types, then the runtime values must be equal according to JavaScript's strict equality (`==`) operator. Both operands will have the same opaque type due to the static typing rules.
- **structure type:** if the operands have structure types, then the corresponding values of each field must be equal according to these rules, based on the field types. Both operands will have the same structure type due to the static typing rules.
- **enum type:** if the operands have enum types, then they must be the same enum value. Both operands will have the same enum type due to the static typing rules.

### Not equals

The operands are compared according to the rules for equals above, and then the boolean result is negated.

### Less than, greater than, less than or equals, and greater than or equals

The integer values of the operands are compared according to the relational operation. Both operands will have unsigned integer types due to the static typing rules.

### Short-circuit logical expressions

Compact supports short-circuit logical expressions of the form `e0 op e1` where `e0` and `e1` are expressions and `op` is one of the logical operators **or** (`||`) or **and** (`&&`).

```
expr → expr || expr
     | expr && expr
```

Logical expressions require the type of the left subexpression to be `Boolean`. The type of the right subexpression must be a supertype of `Boolean`. The only supertype of `Boolean` is `Boolean` itself. The entire expression will have the same type as the type of the right subexpression.

Feedback

Logical expressions are evaluated by first evaluating the left subexpression. Then, the value of that expression determines the value of the entire expression as follows:

- For or, if the value of the left subexpression is `false` then the right subexpression is evaluated and its value is the value of the entire expression. Otherwise, the right subexpression is **not** evaluated and the value of the left subexpression, implicitly cast to the type of the entire expression, is the value of the entire expression.
- For and, if the left subexpression is `true` then the right subexpression is evaluated and its value is the value of the entire expression. Otherwise, the right subexpression is **not** evaluated and the value of the left subexpression, implicitly cast to the type of the entire expression, is the value of the entire expression.

### Conditional expressions

Compact supports conditional expressions of the form `e0 ? e1 : e2` where `e0`, `e1`, and `e2` are expressions.

```
expr → expr ? expr : expr
```

Conditional expressions require the type of `e0` to be `Boolean`. The types of `e1` and `e2` must be in the subtype relation. That is, either the type of `e1` is a subtype of the type of `e2` or else the type of `e2` is a subtype of the type of `e1`.

The type of the entire expression is the type of `e2` if `e1` is a subtype of `e2` and the type of `e1` if `e2` is a subtype of `e1`.

Conditional expressions are evaluated by first evaluating `e0`. Then, the value of that expression determines which of the other subexpressions is evaluated:

- if the value of `e0` is `true`, then `e1` is evaluated and its value is the value of the entire expression
- if the value of `e0` is `false`, then `e2` is evaluated and its value is the value of the entire expression

The evaluation rules ensure that only one of `e1` and `e2` is evaluated.

Feedback

### Map and fold expressions

Compact supports expressions that perform the higher-order operations *map* and (left) *fold* over tuples that have a vector type (not arbitrary tuples).

Map expressions have the form `map(f, e, e, ...)` where `map` is a keyword, `f` is a circuit or witness taking at least one argument, and the `e`s are expressions. A circuit or witness taking *n* arguments can be mapped over *n* argument vectors by providing *n* vector subexpressions to the `map`.

Fold expressions have the form `fold(f, init, e, e, ...)` where `fold` is a keyword, `f` is a circuit or witness, and `init` and the `e`s are expressions. A circuit or witness taking *n*+1 arguments can be folded over an initial value `init` and *n* vectors by providing *n* vector subexpressions to the `fold`.

```
expr → map ( fun , expr {, expr} )
     | fold ( fun , expr , expr {, expr} )
```

The syntax of the circuit or witness is given by the grammar production for *fun* in the section **Circuit and witness calls** above.

A map expression is type checked by checking the type of the witness or circuit `f` to find its parameter types and its return type `R`. `f` must have at least one parameter. The map expression must have the same number of vector subexpressions as the number of parameters of `f`. Each of the vector subexpressions must have a vector type and all these vector types must have the same length `n`. If the type of the *i*th parameter to `f` is `T`, then the type of the *i*th vector subexpression must be `Vector<n, S>` where `S` is a subtype of `T`. The type of entire expression is `Vector<n, R>`.

A fold expression is type checked by checking the type of the witness or circuit `f` to find its parameter types and its return type `R`. `f` must have at least two parameters, and the type of the first parameter must be the same type as the return type `R`. The fold expression must have one fewer vector subexpression than the number of parameters of `f`. The subexpression `e` gives the initial value for the fold. It must have a type which is a subtype `R`. Each of the vector subexpressions must have a vector type and all t

Feedback

types must have the same length `n`. If the type of the *i*+1th parameter of `f` is `T` then the type of the *i*th vector subexpression must be `Vector<n, S>` where `S` is a subtype of `T`. The type of the entire expression is `R`.

Map expressions are evaluated by evaluating the vector subexpressions from left to right. These values are the input vector values. The witness or circuit `f` is then applied in turn, from index 0 up to index `n`-1, to arguments taken from the input vector values. The result is a vector of length `n` where each *i*th element is the result of applying `f` to the *i*th element of the corresponding input vector values.

Fold expressions are evaluated by evaluating the initial value expression `e` and then evaluating the vector subexpressions from left to right. The values of the vector expressions are the input vector values. The witness or circuit `f` is then applied in turn, from index 0 up to index `n`-1, to an accumulator value argument and arguments taken from the input vector values. The 0th (initial) accumulator value is the value of the expression `e`, and each subsequent *i*+1th accumulator value is the result of applying `f` to the *i*th accumulator value and to the *i*th element of the corresponding input vector values. The result is the `n`th (final) accumulator value where `n` is the length of the input vectors.

### Ledger assignment expressions

Compact has ledger assignment statements. They have the form `lhs op e` where `lhs` is a ledger expression as defined in the section **Ledger expressions** above, `op` is one of the assignment operators `=` (assignment), `+=` (addition assignment), or `-=` (subtraction assignment), and `e` is an expression.

```
expr → ledger . id ledger-accessor = expr
     → ledger . id ledger-accessor += expr
     → ledger . id ledger-accessor -= expr
```

`lhs = e` is shorthand for `lhs.write(e)`. `lhs += e` is shorthand for `lhs.increment(e)`. `lhs -= e` is shorthand for `lhs.decrement(e)`.

Ledger assignment statements are type checked exactly as if they were their longer equivalent invoking a ledger ADT o

Feedback

They are evaluated exactly as if their longer equivalent were evaluated as a ledger expression.

# Declaring witnesses for private state

Compact code can call code external to the zero-knowledge circuits to read and update private state via `witness` functions.

A `witness` function declaration can appear anywhere a `circuit` definition can appear, including within modules. A witness function does not have a body, and its implementation is instead an input to the contract in the TypeScript target. For instance:

```
witness something(x: Boolean): Field;
```

A `witness` function can be called in the same way as a `circuit`.

> **DANGER**
>
> Do not assume in your contract that the code of any `witness` function is the code that you wrote in your own implementation. Any DApp may provide any implementation that it wants for your `witness` functions. Results from them should be treated as untrusted input.

# Declaring and maintaining public state

Compact code can declare public state through `ledger` declarations.

A ledger declaration defines one piece of information the contract stores in Midnight's public ledger. Multiple ledger declarations can appear in a program, or none. They can appear anywhere circuit definitions can appear, including within modules.

A ledger declaration associates a ledger field name with one of a set of predefined ledger ADT types For instance:

```
ledger val: Field;
export ledger cnt: Counter;
sealed ledger u8list: List<Uint<8>>;
export sealed ledger mapping: Map<Boolean, Field>;
```

## Ledger state types

In a `ledger` declaration, the following types are valid:

- `T`, for any Compact type `T`
- `Counter`
- `Set<T>`, for any Compact type `T`
- `Map<K, T>`, for any Compact types `K` and `T`
- `Map<K, V>`, for any Compact type `K` and ledger state type `V` (see the following section)
- `List<T>`, for any Compact type `T`
- `MerkleTree<n, T>`, for a compile time integer `1 < n <= 32`, and any Compact type `T`
- `HistoricMerkleTree<n, T>`, for a compile time integer `1 < n <= 32`, and any Compact type `T`

Each ledger type supports a set of operations, which can be invoked with

```
<field name>.<operation>(<arguments ...>)
```

A ledger field that is declared with a Compact type `T` implicitly has the type `Cell<T>`. `Cell` has operations such as `re` and `reset_to_default`.

The `read` and `write` operations for the `Cell` type of any Compact type `T` have syntactic sugar. If `x` is a field in the ledger of any Compact type, then you may write

```
x       // expression equivalent to x.read()
x = val // statement equivalent to x.write(val)
```

The `read`, `increment`, and `decrement` operations of type `Counter` type also have syntactic sugar. If `c` is a `Counter` field in the ledger, then you may write

```
c        // expression equivalent to c.read()
c += val // statement equivalent to c.increment(val)
c -= val // statement equivalent to c.decrement(val)
```

A comprehensive list of operations can be found in the Compact ledger data type documentation.

## Nested state types in the `Map` type

The only ledger state type in which values of other state types may be held is `Map`. The key values in a `Map` must be non-state types (simple Compact types), but the mapped values may be counters, sets, lists, other maps, and so on.

Here is a small example:

```
import CompactStandardLibrary;

ledger fld: Map<Boolean, Map<Field, Counter>>;

export circuit initNestedMap(b: Boolean): [] {
  fld.insert(b, default<Map<Field, Counter>>);
```

Feedback

```
}

export circuit initNestedCounter(b: Boolean, n: Field): [] {
  fld.lookup(b).insert(n, default<Counter>);
}

export circuit incrementNestedCounter(b: Boolean, n: Field, k: Uint<16>): [] {
  fld.lookup(b).lookup(n).increment(k);
}

export circuit readNestedCounter1(b: Boolean, n: Field): Uint<64> {
  return fld.lookup(b).lookup(n).read();
}

export circuit readNestedCounter2(b: Boolean, n: Field): Uint<64> {
  return fld.lookup(b).lookup(n);
}
```

In this example,

- `fld` is bound to a `Map` from `Boolean` values to `Map`s from `Field` values to `Counter`s
- `initNestedMap` can be used to create the inner `Map` for a particular outer-`Map` key
- `initNestedCounter` can be used to create a `Counter` for a given outer-`Map` key and a given inner-`Map` key
- `incrementNestedCounter` can be used to increment an existing `Counter` for a given outer-`Map` key and a given inner-`Map` key
- either `readNestedCounter1` or `readNestedCounter2` can be used to read the value of an existing `Counter` for a given outer-`Map` key and a given inner-`Map` key.

Notes:

Feedback

1. Nesting is permitted only within `Map` values. That is, nesting is not permitted in `Map` keys or within any ledger state type other than `Map`.
2. Nested values must be initialized before first use. The syntax `default<T>` is used to create default ledger state type values, just as it can be used to create default Compact type values.
3. Ledger state type values are not first-class objects, so when accessing a nested value, the entire indirection chain must be used. For example, the following will result in a compiler error:

```
export circuit incrementNestedCounter(b: Boolean, n: Field, k: Uint<16>): [] {
  fld.lookup(b); // ERROR: incomplete chain of indirects
}
```

4. When the last lookup is a read of a base type one can omit the explicit `read()` indirect, as illustrated by the definitions of `readNestedCounter1` and `readNestedCounter2` above, which have the same behavior.
5. For convenience, local variables can hold default ledger state type values, so the following definition of `initNestedMap` is equivalent to the one above.

```
export circuit initNestedMap(b: Boolean): [] {
  const t = default<Map<Field, Counter>>;
  fld.insert(b, t);
}
```

## Sealed and unsealed ledger fields

Any ledger field can be optionally marked *sealed* by prefixing the ledger field declaration with the keyword `sealed`. A sealed field cannot be set except during contract initialization. That is, its value can be modified only by the contract constructor (if any), either directly within the body of the constructor or via helper circuits called by the constructor. The `sealed` keyword must come after the `export` keyword (if present) and before the `ledger` keyword, as in the following example:

```
sealed ledger field1: Uint<32>;
export sealed ledger field2: Uint<32>;

circuit init(x: Uint<32>): [] {
  field2 = x;
}

constructor(x: Uint<16>) {
  field1 = 2 * x;
  init(x);
}
```

It is a static error if a sealed ledger field can be set by any code that is reachable from an exported circuit.

# Contract constructor

A contract can be initialized via a contract constructor defined at the program's top level. The constructor, if any, is typically used to initialize public state and can also be used to initialize private state through witness calls. A constructor can take zero or more arguments. For example:

```
enum STATE { unset, set }
ledger state: STATE;
ledger value: Field;

constructor(v: Field) {
  value = disclose(v);
  state = STATE.set;
}
```

At most one contract constructor can be defined for a contract, and it must appear only at the program top level, i.e., it cannot be defined in a module. To initialize ledger fields that are visible only within a module, the constructor can call a circuit that is exported from the module. For example:

```
module PublicState {
  enum STATE { unset, set }
  ledger state: STATE;
  ledger value: Field;
  export circuit init(v: Field): [] {
    value = disclose(v);
    state = STATE.set;
  }
}

import PublicState;

constructor(v: Field) {
  init(v);
}
```

## Runtime representations and type/bounds checks

The TypeScript type representing a Compact type is defined in Representations in TypeScript.

Compact represents values exactly as TypeScript represents values, i.e., as ordinary JavaScript values. So a Compact boolean is represented at run time as a JavaScript boolean, a Compact tuple is represented as a JavaScript array, and enum values are represented by numbers.

To maintain type safety, Compact verifies at run time that values passed by an outside caller to an exported circuit or re[illegible] outside witness have the expected types. This is necessary even when the caller or witness is written in properly typed Typ[illegible]

because some Compact types have size and range limits that are not expressible via the TypeScript type system:

- `Field` values are limited by a maximum field value
- `Uint` values are limited by the declared bounds
- `Bytes` and tuple values are limited by their lengths
- enum values are limited by the maximum index for the enum elements

It is also necessary because compile-time type checks are easily defeated in TypeScript and are nonexistent when a caller or witness is coded in JavaScript.

Certain values to be stored in public state require size, length, and other properties to be maintained explicitly, because these properties cannot be determined solely from the JavaScript representation of the value. For this purpose, the `@midnight/compact-runtime` package provides explicit runtime types satisfying the `CompactType<T>` interface, where `T` is the corresponding TypeScript type. This representation is *not* user-facing most of the time, except when replicating the behavior of the operations implemented in `@midnight/compact-runtime`.

The following constructors can be used to create a `CompactType` instance for a primitive type:

- `Boolean` - `new CompactTypeBoolean()`
- `Field` - `new CompactTypeField()`
- `Uint<0..n>` - `new CompactTypeUnsignedInteger(n, length)`, where `length` is the number of bytes required to store `n`
- `Uint<n>` - as `Uint<0..(2 ** n) - 1>`
- `Bytes<n>` - `new CompactTypeBytes(n)`
- `Vector<n, T>` - `new CompactTypeVector(n, rt_T)`, where `rt_T` is the runtime type of `T`
- `Opaque<"String">` - `new CompactTypeString()`
- `Opaque<"Uint8Array">` - `new CompactTypeUint8Array()`.

Feedback

For user-defined types, structures are not currently easily constructed at runtime and require implementing `CompactType<T>` manually or using compiler internals. Enumerations are exposed through `new CompactTypeEnum(maxValue, length)`, where `maxValue` is the maximum permissible integer assignable, and `length` its representation length in bytes (almost always 1).

# TypeScript target

When compiled, a contract generates several artifacts. Key to these are the exported circuits from the contract's top level. These are divided into two categories: pure circuits and impure circuits.

In the `contract` directory, the semantics of the contract is encoded in TypeScript, in the form of a `index.cjs` JavaScript implementation file and a `index.d.cts` type declaration file. For most uses, it is recommended to rely on the information and interface provided in `index.d.cts`.

For each of the impure circuits, a zero-knowledge prover/verifier key pair is also generated, as well as instructions for proof generation. These can be found in the output directory's `keys` and `zkir` subdirectories respectively.

## Structure of the exported TypeScript

The exported TypeScript exposes a number of declarations that can be used to interact with the contract from any TypeScript application. Some of these also require use of the `@midnight/compact-runtime` library, which all contracts depend upon and which implements key built-in behaviors.

A contract exports the following in the TypeScript module:

- The TypeScript type corresponding to each user-defined type exported from the contract's top level
- A `Witnesses<T>` type, which describes the format external witnesses must satisfy to instantiate the contract
- A `ImpureCircuits<T>` type, which describes the set of impure circuits exported from the contract's top level

- A `PureCircuits` type, which describes the set of pure circuits exported from the contract's top level
- A `Circuits<T>` type, which describes the set of all exported circuits
- A `Contract<T, W extends Witnesses<T> = Witnesses<T>>` class, which:
  - can be constructed by passing in an instance of `W`
  - exposes members `circuits: Circuits<T>` and `impureCircuits: ImpureCircuits<T>`
  - provides initial contract states via `initialState(privateState: T): [T, runtime.ContractState]`
- A constant `pureCircuits: PureCircuits` object, providing all pure circuits as pure functions
- A `Ledger` type, providing views into a current ledger state, by permitting direct calls of all read functions of `ledger` objects, as well of some TypeScript specific ones that cannot be called from Compact, such as iterators
- A `ledger(state: runtime.StateValue): Ledger` constructor of the `Ledger` type, giving access to the values of exported ledger fields.

The argument `T` for a number of these should be interpreted as the type of the local/private state. For the most part, `circuit` and `witness` functions are translated simply by translating their Compact types into corresponding TypeScript types for parameters and return values. For `PureCircuits`, this is all that happens, for the other `_Circuits` instances, they receive an additional first parameter of type `runtime.CircuitContext<T>`, and their result type `R` is wrapped in `runtime.CircuitResults<T, R>`. For `Witnesses`, they receive an additional first parameter of type `runtime.WitnessContext<Ledger, T>`, and their result type `R` is wrapped in `[T, R]`. See the runtime API docs for the details of these types. This wrapping makes the entirety of the contract code *functional*, ensuring calls have no hidden side effects.

Feedback